VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

LIBYANG - MODUL PRO PYTHON 3

LIBYANG - PYTHON 3 MODULE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DAVID SEDLÁK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JAN KUČERA**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Sedlák David**
Program: Informační technologie
Název: **Libyang - modul pro Python 3**
Libyang - Python 3 Module
Kategorie: Softwarové inženýrství

Zadání:

1. V teoretické části práce se seznamte s modelovacím jazykem YANG a knihovnou libyang, která jej implementuje.
2. Proveďte porovnání výkonnosti jednotlivých možností implementace obalové knihovny (language binding) pro jazyk Python 3.
3. Navrhněte vhodné programové rozhraní knihovny libyang pro jazyk Python 3.
4. Toto programové rozhraní implementujte jako obalovou knihovnu.
5. Vytvořenou knihovnu otestujte a připravte sadu příkladů použití.
6. V závěru diskutujte výsledky a možnosti dalšího pokračování práce.

Literatura:

- CLAISE, Benoit, CLARKE, Joe, LIDNBLAD, Jan. Network Programmability with YANG: Data Modeling-Driven Management with YANG. Addison Wesley, 2019. ISBN 978-0-13-518039-6.
- https://github.com/CESNET/libyang

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

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

Vedoucí práce: **Kučera Jan, Ing.**
Konzultant: Krejčí Radek, RNDr., CESNET
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 31. července 2020
Datum schválení: 25. října 2019

Abstrakt

Bakalářská práce se zabývá interakcí jazyků C a Python 3. Hlavním cílem bylo vytvoření uživatelsky přívětivého Python 3 rozhraní pro knihovnu libyang2, která své základní rozhraní poskytuje v jazyce C. Výsledné Python 3 rozhraní je vytvořené s využitím CFFI balíčku, ten umožňuje jednoduché použití v rámci jazyka Python 3 a adekvátním způsobem využívá jeho možností. Součástí práce je také srovnání několika vybraných přístupů a nástrojů, které interakci mezi jazyky C a Python 3 umožňují, včetně měření a porovnávání režie spojené s jejich použitím. Implementované rozhraní poslouží ke zjednodušení integrace podpory modelovacího jazyka YANG do síťových aplikací a zařízení.

Abstract

This bachelor thesis deals with interaction between C and Python 3 languages. The main goal was to provide user friendly Python 3 binding for libyang2 library which provides interface for C language. Final Python 3 interface is created with help of CFFI package, is simply usable in Python 3 and adequately uses rich features of Python 3 language. Comparison of few approaches and tools to interact with C library from Python code is also covered in the thesis, including measurements of call overhead caused by them. Implemented Python 3 interface will help with integration of YANG modeling language in network applications and devices.

Klíčová slova

libyang2, YANG, obalová knihovna pro Python, design aplikačního rozhraní, CFFI, C extension, Cython, Ctypes

Keywords

libyang2, YANG, Python binding, Python wrapper, API design, CFFI, C extension, Cython, Ctypes

Citace

SEDLÁK, David. *Libyang - modul pro Python 3*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Kučera

Libyang - modul pro Python 3

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením inženýra Jana Kučery. Další informace mi poskytli RNDr. Radek Krejčí. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
David Sedlák
31. července 2020

Poděkování

Tímto bych rád poděkoval především svému konzultantovi Radku Krejčímu za cenné rady, které mi poskytl ke všem aspektům práce a stejně tak vedoucímu práce Janu Kučerovi.

Obsah

Kapitola 1

Úvod

Počítačové sítě se stávají stále větší součástí lidského života ve vyspělé společnosti. Na jejich správném fungování jsou závislé velké korporace, banky, školy, nebo dokonce i mnohé aspekty našeho osobního života. Jsou zkrátka nezbytné k provádění mnohých činností, které dnes již považujeme za samozřejmé a život si bez nich už takřka nedokážeme představit. S tím se však v celkovém měřítku zvyšuje i jejich velikost a komplexita. Čím jsou jednotlivé sítě složitější a rozsáhlejší, tím jsou kladeny větší požadavky na jejich návrh, uvedení do provozu a následnou správu a monitorování sítě.

Některé z těchto aspektů, konkrétně uvedení sítě do provozu, její správa a monitorování, dnes již mohou být výrazně zjednodušeny s využitím přístupu nazývaného DMDNM (Data Model Driven Network Management, *česky: modelově orientovaný přístup ke správě sítě*). S využitím tohoto inovativního přístupu mohou být sítě a služby na nich provozovány uvedeny do chodu rychleji než kdy dříve a při jejich uvedení do provozu nemusí docházet k chybám zaviněným lidskou nepozorností. Během následného provozu sítě mohou být chyby objeveny, diagnostikovány a opraveny rychleji než při konvenčních přístupech. Správa sítě postavené kolem DMDNM přístupu může být díky využití automatizace podstatně snažší, to umožňuje snížit i náklady, které jsou s provozem sítě spojené.

Jak už název napovídá, v případě DMDNM přístupu ke správě sítě, jednu z hlavních komponent tvoří datové modely. Ty jsou využívány k popisu rozhraní pro správu jednotlivých zařízení a služeb na síti. Díky existenci jednotného a přesně definovaného rozhraní mohou být jednotlivé části sítě jednoduše programovatelné a většina úkonů může být automatizována. Standardním jazykem pro modelování se v této oblasti stal k tomu určený modelovací jazyk YANG. Modely jsou následně využívány některými protokoly pro správu sítě, kterými jsou například NETCONF a RESTCONF.

Aby potenciál tohoto přístupu ke správě sítě mohl být naplno využit, musí existovat nástroje, které jeho integraci a využití umožňují a usnadňují. Čím více takových nástrojů bude, tím jednodušší a atraktivnější bude jejich používání pro správce sítí a programátory síťových služeb. Jedním z nástrojů, které k jeho rozšíření napomáhají, je knihovna libyang. Ta konkrétně usiluje o usnadnění integrace podpory modelovacího jazyka YANG do síťových služeb a zařízení.

Libyang umožňuje zpracování a validaci datových modelů popsaných pomocí modelovacího jazyka YANG. Dále umožňuje zpracování, validaci a manipulaci instančních dat serializovaných pomocí k tomu určených formátů [nejčastěji XML (Extensible Markup Language, *česky: rozšiřitelný značkovací jazyk*) a JSON (JavaScript Object Notation, *česky: javaskriptový objektový zápis*)]. Poskytuje API (Application Programming Interface, *česky: rozhraní pro programování aplikací*) v jazyce C.

Mým cílem je vytvořit ke knihovně libyang rozhraní, díky kterému bude většina jejích funkcí jednoduše použitelná z jiného a dnes velmi populárního programovacího jazyka Python.

Rozhraní, které umožňuje použití knihovny v jiném jazyce, než ve kterém byla původně vytvořena, se běžně nazývá language binding (*česky: obalová knihovna*). Cílem práce je tedy vytvořit Python binding pro knihovnu libyang.

V době tvorby této práce je dokončována implementace nové verze knihovny libyang (libyang 2.x). Cílem tvorby této nové verze je odstranit nedostatky, které se v aktuální verzi (libyang 1.x) nacházejí. Současná verze knihovny obsahuje dokonce dvě různá rozhraní, které umožňují použití v rámci jazyka Python. Jedna verze Python rozhraní pro libyang 1.x však příliš neodráží podstatně vyšší úroveň abstrakce, kterou Python oproti jazyku C nabízí, a její použití v rámci jazyka Python tak nepůsobí velmi přirozeně a intuitivně. Druhá existující verze zase nabízí pro Python odpovídající úroveň abstrakce, ale pokrývá jen omezenou část funkcí knihovny a původně byla vytvořena pro jeden konkrétní případ užití. To je jeden z hlavních problémů, které se touto prací snažím vyřešit a pro libyang 2.x vytvořit jeden Python binding, který bude pokrývat většinu funkcí a zároveň jeho používání v rámci jazyka Python bude pohodlné a přirozené. Nová verze knihovny s sebou přináší i nevyhnutelné změny API a nezaručuje tedy úplnou zpětnou kompatibilitu s předchozí verzí. Díky tomu se naskýtá ideální příležitost k provádění významnějších změn i v rozhraní pro Python.

Pro vytvoření kvalitního a dobře použitelného API je nutné pochopit, jakým způsobem a k čemu je knihovna využívána, stejně tak jako její architekturu a některé implementační detaily nejdůležitějších částí. Z toho důvodu se v kapitole 2 obecně zabývám modelovacím jazykem YANG, možnostmi jeho využití a hlavními částmi knihovny libyang 2.x.

Datové modely i instanční data, která libyang zpracovává, mohou být v některých případech rozsáhlé a komplexní, práce s nimi proto může být výpočetně náročná. Libyang se tento proces snaží v mnoha ohledech optimalizovat, což mu umožňuje i velké vstupy zpracovat poměrně rychle a efektivně. Language binding je primárně zodpovědný za transformaci datových struktur z formátu používaného v prostředí, pro který je vytvořen, do datových struktur používaných v prostředí, ve kterém je napsána původní knihovna. Konkrétně Python binding pro knihovnu libyang bude často provádět transformace z Python objektů na datové struktury jazyka C a naopak. Existují obavy o tom, že v případě implementace language bindingu pro libyang nevhodným způsobem, by mohlo docházet ke zbytečně časté nebo neefektivní transformaci datových struktur, což by mohlo vést ke zbytečnému snížení výkonnosti knihovny. Proto se v kapitole 3 zabývám srovnáním několika vybraných nástrojů a přístupů, které lze při implementaci tohoto konkrétního language bindingu použít, a zaměřuji se na porovnání jednotlivých nástrojů z pohledu výkonnosti, udržovatelnosti a náročnosti na použití.

Dále v kapitole 4 popisuji použité ideologie převodu některých konceptů jazyka C na koncepty jazyka Python a nejdůležitější prvky navrženého rozhraní. V kapitole 5 popisuji některé zajímavější problémy, na které jsem během implementace narazil. V kapitole 6 popisuji jakým způsobem byla implementace rozhraní testována. V kapitole 7 diskutuji výsledky práce a možnosti jejího dalšího pokračování. A v příloze A na konkrétních příkladech ukazuji využití implementovaného rozhraní.

Kapitola 2

Modelovací jazyk YANG a knihovna libyang 2.x

Kapitola obsahuje obecné informace o modelovacím jazyku YANG a následně o knihovně libyang 2.x, která jej implementuje.

2.1 Modelovací jazyk YANG

Tato sekce se nejprve věnuje modelovacímu jazyku YANG, některým jeho vlastnostem a možnostem jeho využití. Dále jsou uvedeny některé zajímavější vlastnosti jazyka YANG, kvůli kterým může být jeho zpracování často problematické. Součástí sekce je i příklad jednoduchého YANG modulu a jeho instančních dat. Cílem sekce není encyklopedické popsání všech prvků jazyka, ani vyčerpávající přepisování standardu. Obsáhlost jazyka YANG a všech jeho částí je mimo rozsah tohoto dokumentu a k jeho komplexnímu pochopení je téměř nutné nahlédnout do standardu.

2.1.1 Základní vlastnosti a využití

Po vytvoření protokolu NETCONF, který nabízí mechanismy k instalaci, úpravám a odstranění konfigurace síťových zařízení [12, Abstrakt], bylo nutné vytvořit jednotný modelovací jazyk, pomocí kterého bude možné jednoduše specifikovat strukturu a význam dat, které protokoly pro správu sítě přenášejí. Proto byl (původně jako doplněk protokolu NETCONF [7, sekce 1.]) vytvořen modelovací jazyk YANG. Ve verzi 1.0 byl poprvé standardizován v roce 2010 v podobě RFC 6020 [7]. O 6 let později vznikl standard pro verzi 1.1, který je popsaný v RFC 7950 [8]. Verze 1.1 přináší několik oprav a počítá s použitím i mimo protokol NETCONF [8, sekce 4.1.]. Ve většině případů je verze 1.1 zpětně kompatibilní s verzí 1.0 [8, sekce 1.].

V porovnání s obecnějšími modelovacími jazyky, kterým je například XML Schema, má YANG v této oblasti výhodu v tom, že se primárně zaměřuje na popis dat využívaných protokoly pro správu sítě [8, sekce 4.1.]. Nesnaží se být všeobecným modelovacím jazykem [8, sekce 4.1.]. Tato vlastnost při vytváření YANG modelů zužuje problémovou doménu a dovoluje se soustředit na to, co je v rámci správy sítě důležité.

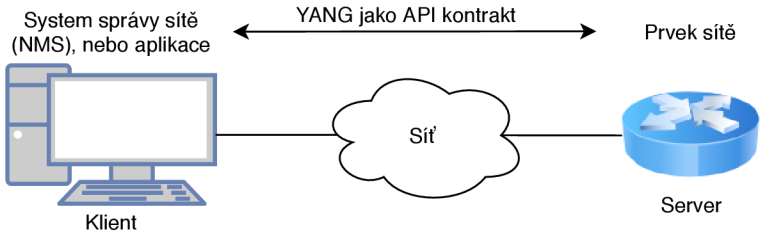
YANG modeluje hierarchické uspořádání dat jako stromovou strukturu, ve které má každý uzel specifikované jméno a množinu podřazených uzlů, nebo hodnotu. [8, sekce 4.1.].

Model v jazyce YANG je definován pomocí množiny modulů [8, sekce 4.2.1.]. YANG modely, případně konkrétní moduly, se často označují jako schéma, nebo schémata. Každý modul je tvořen třemi částmi, a to hlavičkou modulu, revizemi a tělem modulu [8, sekce 4.2.1.]. V hlavičce modulu jsou obsaženy všeobecné informace o něm, jako například namespace modulu nebo jeho prefix. Část s revizemi obsahuje informace o předchozích verzích modulu a o tom, kolik v něm od jeho vydání bylo provedeno změn. Samotné tělo modulu pak obsahuje datový model, který daný modul reprezentuje.

Využití

Pomocí jazyka YANG mohou být modelována všechna data, která se posílají mezi NETCONF klientem a serverem. To zahrnuje modelování konfiguračních a stavových dat, modelování vzdáleného volání procedur (RPC) a notifikací [8, sekce 4.1.].

Běžně se YANG využívá k popisu API jednotlivých prvků na síti. Lze pomocí něj vytvořit tzv. API kontrakt, tedy přesně určit, jak má vypadat rozhraní mezi klientem a serverem v jednotlivých situacích [9, kapitola 2]. Využití YANG modelu jako API kontraktu je ilustrováno v obrázku 2.1. Klientem je v tomto kontextu NMS (Network Management System, *česky: systém správy sítě*), nebo jiná aplikace, která k rozhraní pro správu daného prvku sítě nějakým způsobem přistupuje. Co všechno server klientovi umožňuje, je dáno tím, jaké YANG modely implementuje.



Obrázek 2.1: Příklad využití jazyka YANG jako API kontraktu, s úpravami převzato z [9, kapitola 2]

Příklady některých zajímavějších vlastností jazyka

Vlastnosti uvedené v této podsekci z mého pohledu patří k zajímavým vlastnostem jazyka YANG. Při strojovém zpracování jazyka však mohou být uvedené vlastnosti v různých ohledech problematické a zpracování jazyka YANG je celkově náročné k implementaci.

Rozšíření (YANG výraz *extension*)

Součástí jazyka YANG je i mechanismus pro jeho rozšíření dle potřeb návrháře. Pomocí příkazu extension je definován nový příkaz, který může být používán stejným způsobem jako základní vestavěné příkazy [8, sekce 7.19.]. Tímto způsobem definované nové příkazy mohou být importovány a využívány i v jiných modulech, než ve kterých byly původně definovány [8, sekce 7.19.].

Deviace (YANG výraz *deviation*)

Přestože v ideálním světě by se to nemělo stávat, často existují zařízení, která implementují nějaký standard, ale nejsou schopna podporovat všechny jeho části. V takovém případě zařízení stále může využívat běžný model, který daný standard popisuje. S využít mechanismu deviace může dát najevo, že daný model nepodporuje věrně a zároveň specifikovat, ve kterých částech a jak se od standardního modelu odlišuje [8, sekce 7.20.3.].

Augmentace (YANG výraz *augment*)

Často se stává, že k existujícímu standardu si chce daný výrobce v rámci svojí implementace něco přidat. Pokud v takovém případě pro standard už existuje standardní model, může ho výrobce využít a pomocí augmentace si do něj přidat, co potřebuje. Augmentace umožňuje přidání datových uzlů do stromové struktury jiného modulu, přičemž její provedení může být podmíněno splněním nějaké podmínky [8].

Omezující podmínky

Existuje mnoho způsobů, jak v modulech specifikovat různá omezení, která musí být splněna k tomu, aby daná instanční data mohla být považována za validní. Některé uzly mohou být označeny jako povinné a prvky seznamu mohou být označeny jako unikátní. Pomocí výrazu must je možné specifikovat omezení na základě *XPath*[1] výrazu [8, sekce 7.5.3.]. Při definici nových typů lze přidat omezení, která musí platit pro jednotlivé instance daného typu. Taková omezení mohou být založena na délce, rozsahu, nebo regulárním výrazu, který daná hodnota musí splňovat.

2.1.2 Příklad modulu a jeho instančních dat

```
module interfaces {
      namespace "urn:example:interfaces";
      prefix "ex";
      container "interfaces" {
            list interface {
                  key "name";
                  leaf name {
                        type string;
                        description "Name of the interface.";
                  }
                  leaf mtu {
                        type uint32;
                        description "The MTU of the interface.";
                  }
            }
      }
}
```

Výpis 2.1: Jednoduchý YANG modul reprezentující seznam rozhraní. Rozhraní je popsáno pomocí názvu a MTU, kde název rozhraní tvoří klíč seznamu, s úpravami převzato z [8]

[1] XPath je jazyk, který umožňuje adresovat části XML dokumentů [10].

```
<interfaces xmlns="urn:example:interfaces">

      <interface>
            <name>eth0</name>
            <mtu>1500</mtu>
      </interface>

      <interface>
            <name>wlps0</name>
            <mtu>2304</mtu>
      </interface>

</interfaces>
```

Výpis 2.2: Ukázka instančních dat modulu z ukázky 2.1 ve formátu XML. Data reprezentují dvě rozhraní s názvy eth0 a wlps0 a jejich MTU.

2.2 Knihovna libyang 2.x

V této sekci je nejprve uveden stručný popis knihovny libyang 2.x a jejího běžného využití. Následně se zaměřuje na její architekturu a některé implementační detaily. Naprostá většina informací využitých v této sekci byla získána z dokumentačních komentářů obsažených ve zdrojových kódech knihovny libyang 2.x. Obecnější forma dokumentace zatím nebyla vytvořena.

2.2.1 Popis knihovny

Knihovna je implementována v jazyce C pro GNU/Linux. V jazyce C poskytuje také své API, které tvoří 201 funkcí, 15 datových typů, 95 datových struktur, 18 enumerací a 168 maker [16]. V době tvorby této práce je knihovna libyang 2.x stále ve vývoji a některé funkce ještě nejsou plně podporovány, zároveň stále dochází k drobným změnám jejího rozhraní. Vydání beta verze je plánováno na červenec 2020, vydání plné verze pak na září 2020.

Libyang při načítání YANG modulů využívá konceptu kontextů, kdy kontext představuje množinu modulů, které jsou v něm načteny. Uživatel může vytvářet libovolné množství kontextů a následně do nich načítat moduly a dále s nimi pracovat. Díky tomu je možné, aby jedna aplikace jednoduše pracovala s teoreticky neomezeným počet vzájemně nezávislých množin modulů. Veškerá práce s daty pak probíhá v závislosti na použitém kontextu.

Hlavní funkce

Knihovna uživatelům poskytuje celou řadu funkcí. Mezi ty hlavní patří následující.

- Parsování, validace a tisk modulů ve formátech YANG, YIN[2].
- Parsování, validace a tisk instančních dat ve formátu XML a LYB[3].
- Manipulace s instančními daty.

[2]Definice formátu YIN je součástí standardu, který popisuje jazyk YANG. Jedná se o jeho ekvivalent mapovaný do XML.

[3]Binární formát LYB byl vytvořen jedním z autorů knihovny libyang za účelem optimalizací.

- Podpora pro rozšíření jazyka YANG, uživatelské typy a YANG metadata.

Tvorba nové verze

Po vytvoření původní verze knihovny libyang 1.x byly upřesňovány některé nejasné vlastnosti jazyka YANG z úvodního standardu, se kterými původní návrh knihovny nepočítal. Udržovat knihovnu aktuální a zároveň opravovat nově nalezené chyby bylo stále složitější a do budoucna neudržitelné. Současně autoři zpětně viděli některé části knihovny, které by podle nich mohly být interně realizovány lépe [16, issue 880].

Tyto zpětně nevhodně navržené části však razantně ovlivňovaly zbytek knihovny, proto začala vznikat nová verze libyang 2.x, která nezaručuje úplnou zpětnou kompatibilitu s rozhraním libyang 1.x. Kromě změn rozhraní libyang 2.x obsahuje dvě významné interní změny, a to rozdělení struktur k ukládání schémat na dva typy: na struktury pro načtená schémata[4] a na struktury pro kompilovaná schémata. Zároveň odděluje vyhodnocení referencí do externích modulů od samotného načítání modulu [16, issue 880].

Druhou interně velmi významnou změnou je odstranění možnosti odebrat modul z kontextu. Tím, jak jsou nejrůznější části jednotlivých YANG modulů vzájemně propojené, bylo odstranění modulu z kontextu velmi náročné a komplikovalo mnoho dalších částí knihovny. Tato funkcionalita nebyla často využívána, nebo se jejímu použití dalo snadno vyhnout [16, issue 880].

Běžné využití knihovny

Prvním krokem, který by měl uživatel při běžném použití knihovny provést, je tedy vytvoření nového kontextu pomocí k tomu určené funkce z API knihovny[5]. Po vytvoření kontextu se uživatel dostane do stádia, které by se dalo označit jako budování kontextu. V tomto okamžiku by se do kontextu měly načíst všechny moduly, které v něm uživatel požaduje. Jednotlivé moduly je možné do kontextu přidávat hned několika způsoby a z různých zdrojů (ze souboru, z paměti atd...).

Během načítání modulu libyang 2.x provede jeho syntaktickou validaci. V běžných případech následně modul zkompiluje a implementuje do kontextu. Kompilací libyang 2.x v modulu vyhodnotí všechny reference do externích modulů a provede jeho úplnou validaci. Modul po kompilaci tak neobsahuje všechna syntaktická data, která obsahoval před kompilací, z toho důvodu se v kontextu standardně ukládá současně kompilovaná i parsovaná verze modulu[6]. Kompilovaná verze obsahuje všechny informace, které jsou potřebné k validaci datových stromů. Parsovaná verze na druhou stranu zahrnuje všechny informace, které jsou třeba k serializaci modulu zpět do textové podoby.

Po vybudování kontextu může uživatel přistupovat k jednotlivým modulům[7]. Uživatel má přístup ke všem vlastnostem jednotlivých modulů a k celé stromové struktuře, která je v nich definována. Může přistupovat k parsovaným i kompilovaným verzím jednotlivých modulů. Uživatel by moduly ani žádné jejich části neměl modifikovat, jsou mu určeny pouze ke čtení.

[4] Načtená schémata se v rámci knihovny libyang 2.x označují jako parsovaná schémata.

[5] Funkce pro vytvoření kontextu se jmenuje `ly_ctx_new()`.

[6] Do budoucna je v plánu implementovat možnost po kompilaci smazat parsovanou verzi modulu pro šetření místa, v případě, že s ní uživatel nepotřebuje pracovat.

[7] I přesto, že libyang 2.x tomu v aktuální verzi nebrání, uživatel by neměl z modulů obsažených v kontextu používat ani ukládat žádné ukazatele předtím, než budování kontextu dokončí. Implementací dalších modulů do kontextu se totiž mohou některé ukazatele obsažené uvnitř struktur modulů změnit a původní zneplatnit.

Další důležitou částí je práce s daty. Po vybudování kontextu může uživatel načítat instanční data z formátu XML[8]. S instančními daty smí následně manipulovat, kompletně datový strom měnit a po provedení změn může celý strom opět validovat vůči schématu. Libyang 2.x umožňuje vytvoření celého datového stromu i bez načítání ze serializačních formátů. Celý datový strom může být vytvořen použitím několika funkcí z API knihovny.

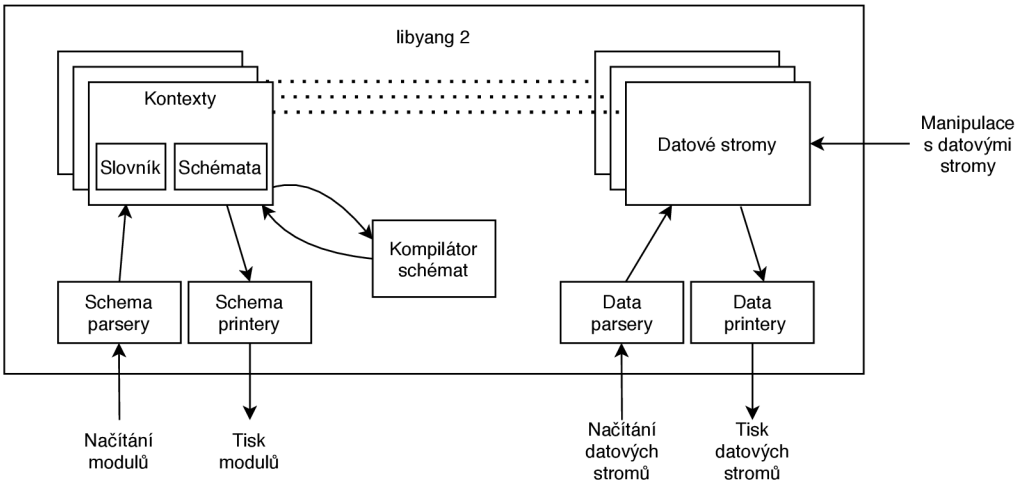
Knihovna libyang 2.x umožňuje moduly i datové stromy ze svých struktur serializovat do textové podoby. V rámci dokumentace a rozhraní knihovny libyang je serializace označována jako tisk.

Architektura

Architekturu knihovny tvoří následující klíčové prvky.

- Kontexty, do kterých se ukládají jednotlivé moduly. Součástí každého kontextu je i slovník[9].
- Schema a data parsery se používají pro načítání schémat a datových stromů z podporovaných formátů.
- Schema a data printery slouží k serializaci schémat uložených v kontextu a vytvořených datových stromů do textové podoby.
- Datové stromy, které může uživatel různými způsoby vytvářet a manipulovat s nimi.

Na obrázku 2.2 je ve zjednodušené podobě znázorněna interakce hlavních částí knihovny.



Obrázek 2.2: Zjednodušená ilustrace interakce hlavních částí knihovny, kde plné čáry představují tok dat a přerušované čáry interní propojení daných částí.

2.2.2 Implementační detaily

Libyang 2.x často využívá datový typ, který označuje *sized-array*. Jedná se o běžné pole, které má hned před svým začátkem uloženou informaci o počtu uložených prvků. Manipulace s tímto typem je vyřešena pomocí maker obsažených v API knihovny.

[8] Podpora formátu JSON není v nové verzi knihovny zatím zcela dokončena.

[9] Slovník v kontextu se využívá k šetření jinak zbytečně využitého místa při opakovaném ukládání stejných řetězců.

Libyang 2.x kromě sized-array obsahuje vlastní implementaci i několika běžnějších ADT (Abstract Data Type, *česky: abstraktní datový typ*) a využívá je i ve svém rozhraní. Některé funkce vrací instanci datového typu množina, se kterým uživatel opět může pracovat pomocí funkcí z API knihovny. Dalším častým datovým typem je lineární seznam, pomocí kterého libyang 2.x v některých případech vytváří kolekce. Pomocí lineárního seznamu jsou organizovány třeba podřazené uzly rodičovského uzlu ve stromových strukturách.

Některé struktury jsou vzájemně kompatibilní díky tomu, že je jejich začátek z podstatné části tvořen stejnými položkami. Libyang 2.x tento mechanismus využívá k vytvoření typové hierarchie, kdy jedna obecná struktura zastřešuje celou řadu specializovanějších struktur. Informace, podle které lze určit, o jakou specializovanější strukturu se jedná, je uložena v kompatibilní části struktury, a je tak vždy možné z obecné struktury určit, o kterou specializovanou strukturu se jedná a obecnou strukturu na ni přetypovat.

V několika částech knihovny je využívána i práce s jednotlivými bity a bitovými maskami. Při volání některých funkcí může být nastavení různých bitů zkombinováno v rámci jednoho argumentu. Chování dané funkce se tímto způsobem dá ve velké míře konfigurovat i bez toho, aby funkce přijímala velké množství povinných argumentů. Některé struktury obsahují položku `flags`, která je využita jako bitové pole, nastavení jednotlivých bitů v poli pak indikuje, zda instance struktury splňuje vlastnost, která je daným bitem reprezentována.

Kapitola 3

Python binding

Termín language binding označuje rozhraní umožňující interagovat s knihovnou z jiného jazyka, než ve kterém byla původně vytvořena. Konkrétním případem je Python binding (často označován také jako Python wrapper) pro knihovnu jazyka C, jehož úkolem je překlenout rozdíly v běhových prostředích jazyků C a Python. V ideálním případě by Python binding měl reflektovat i rozdíly v úrovních abstrakce, které mezi jazyky C a Python existují.

V této kapitole jsou nejprve krátce uvedeny jazyky Python a C a jejich hlavní rozdíly, které Python binding musí zohledňovat.

Následně jsou uvedeny tyto 4 přístupy běžně využívané při tvorbě Python bindingu pro knihovnu jazyka C, včetně měření výkonnosti výsledného rozhraní při použití jednotlivých přístupů.

- Využití API Python interpretu k definici rozšiřujicího modulu v jazyce C/C++. Tento přístup je dále v textu označován jako C extension.
- Využití Python balíčku CFFI k tvorbě základního wrapperu, nad kterým je následně vytvořen Python balíček s rozhraním pro uživatele. Tento Přístup je dále v textu označován jen jako CFFI.
- Využití Python balíčku Ctypes k tvorbě základního wrapperu, nad kterým je následně vytvořen Python balíček s rozhraním pro uživatele. Tento přístup je dále v textu označován jako Ctypes.
- Využití jazyka Cython. Tento přístup je dále v textu označován jen jako Cython.

Nástroj SWIG není v kapitole uvažován z důvodu předchozích špatných zkušeností s udržovatelností SWIG wrapperu ze strany libyang vývojářů. SWIG má oproti ostatním přístupům uvedeným v kapitole nevýhodu ve svojí komplexnosti, která je způsobena jeho obecností. Cílem práce je vytvořit kvalitní language binding pro jazyk Python, ne pro více jazyků. V případě, že by bylo cílem vytvořit language binding pro několik jazyků najednou, připadal by SWIG v úvahu jako velmi zajímavá varianta.

V závěru kapitoly jsou diskutovány výsledky měření a výběr konkrétního přístupu k implementaci, který je v této práci použit. I přesto, že se při porovnáních zaměřuji na Python binding pro knihovnu libyang 2.x, je možné většinu z nich zobecnit a aplikovat na Python binding pro libovolnou jinou knihovnu napsanou v jazyce C.

3.1 Jazyk C versus Python

Jazyk C

Jazyk C je kompilovaný programovací jazyk pro obecné použití. Nejedná se o „příliš vysokoúrovňový“ ani „velký“ jazyk, ale díky jen malým omezením programátora a svojí obecnosti je pro mnohé aplikace vhodnější a efektivnější než zdánlivě mocnější jazyky [15].

Počátky jazyka C sahají až do sedmdesátých let dvacátého století [15], ale i tak je to stále jeden z nejrozšířenějších programovacích jazyků dnešní doby [6], a to i přesto, že je v porovnání s modernějšími jazyky (kterými jsou například Java, Python, nebo JavaScript) dost nízkoúrovňový a jeho používání není oproti těmto modernějším jazykům pro programátora příliš komfortní. Hlavními výhodami, které jazyk C přináší, jsou však rychlost, rozšířenost a v mnoha ohledech i jeho blízkost k hardwaru.

V současnosti nalézá využití při tvorbě systémového softwaru, kterým jsou například ovladače zařízení, nebo obecně při tvorbě firmwaru a operačních systémů. Stejně tak je často používaný pro tvorbu nejrůznějších knihoven, kompilátorů, interpretů a softwaru pro vestavěné aplikace.

Python

Jeden z nejpopulárnějších programovacích jazyků dnešní doby [6] s multiparadigmatickým přístupem. Opět se jedná o univerzální jazyk, který není přímo zaměřený na žádnou konkrétní oblast. Má velmi širokou škálu využití.

K hlavním vlastnostem, se kterými byl Python navržen, patří jednoduchá čitelnost a udržovatelnost kódu. Tohoto cíle se Python mimo jiné snaží dosáhnout neobvyklým způsobem syntaxe a oddělováním logických bloků pomocí bílých znaků, jako jsou mezery a tabulátory [4, sekce 2.1.8.]. Mezi Python programátory dokonce existuje přídavné jméno „pythonic“(*česky pythonický*) označující kód, který správně využívá vlastností jazyka k tomu, aby požadovaného chování dosáhl přehledným a pokud možno i jednoduchým způsobem.

Python má hned několik implementací. Tradiční a nejčastěji používaná implementace, kterou spravuje Python Software Foundation, se jmenuje CPython [4, sekce 1.1.]. Alternativní implementace nejčastěji nabízí kompatibilitu s jinými jazyky, specifickými platformami, nebo vylepšení výkonnosti. Jako příklady alternativních implementací mohou být uvedeny IronPython, Jython, MicroPython, nebo PyPy [4, sekce 1.1.].

Své místo si jazyk našel například jako nástroj pro psaní jednoduchých skriptů, při automatizaci nejrůznějších činností, při analýze dat nebo třeba v oblasti strojového učení.

Hlavní rozdíly

Tato podsekce obsahuje shrnutí nejdůležitějších rozdílů v jazycích Python a C, které musí Python binding zohledňovat.

Datové typy

V jazyce C existují tři skupiny základních datových typů, a to celočíselné typy, typy pro čísla s plovoucí řádovou čárkou a prázdný typ `void`. S použitím základních typů a dalších prostředků jazyka jsou definovány odvozené typy [15]. Python nabízí mnohem větší množství vestavěných typů a pro všechny základní typy jazyka C má vlastní typ pro reprezentaci stejného druhu dat. U celočíselného typu v jazyce Python však není omezen jeho rozsah [5], jak tomu je u celočíselných typů jazyka C.

Textové řetězce jsou v jazyce C standardně reprezentovány jako pole znaků ukončené nulovým znakem, zatímco Python má pro jejich reprezentaci vestavěný typ `str` [5].

Ukazatele a reference

Všechny proměnné jazyka C odpovídají oblasti paměti, která jim patří, a všechny argumenty jsou funkcím předávány hodnotou. Pokud má být obsah proměnné uvnitř funkce změněn, musí být funkci předán ukazatel na paměť, která dané proměnné patří. Práce s ukazateli je v jazyce C běžná a velmi často využívaná.

V jazyce Python jsou všechny proměnné reference na instanci objektu. Argumenty jsou funkcím sice také předávány hodnotou, ale vždy se jedná o hodnotu reference na daný objekt. Pokud je referencovaný objekt uvnitř funkce změněn, projeví se změny i mimo funkci. Python standardně neumožňuje získat ukazatel do paměti.

Správa paměti

Při práci s pamětí v jazyce C je programátor zodpovědný za alokaci paměti a její uvolnění [15, sekce 7.8.5.], zatímco Python obsahuje mechanismy, které programátora od přímé alokace paměti odstiňují a o alokaci i uvolňování paměti se stará interpret [3, sekce 1.10.]. Python binding tak musí zajistit, aby paměť potřebná při práci s knihovnou jazyka C byla správně alokována a následně i správně uvolněna. V případě, že automatické uvolňování paměti nelze zajistit, musí uživateli alespoň poskytnout způsob, jak lze danou paměť uvolnit.

3.2 Vybrané přístupy k implementaci

V této sekci jsou uvedeny základní vlastnosti vybraných přístupů k implementaci. Informace v této sekci jsou tvořeny spojením mých osobních zkušeností získaných při experimentování s jednotlivými přístupy a informací z dokumentací jednotlivých přístupů.

C extension

Standardní způsob pro tvorbu modulů jazyka Python v C a C++ . CPython poskytuje API, pomocí kterého mohou být definovány nové moduly [3]. Stejné API je interně využíváno i v rámci některých sofistikovanějších přístupů, které jsou popsány dále v této kapitole.

Výhody

V tomto případě dostává programátor velkou kontrolu nad tím, co všechno bude výsledný modul umožňovat, a interně dokáže definovat a využívat části, které nebudou v prostředí jazyka Python dostupné. Tato vlastnost může být vhodná, pokud je žádoucí, aby uživatel nemohl přistupovat k některým interním vlastnostem modelem definovaných typů.

Celý modul je popsaný jazykem C/C++. Díky tomu, že je modul popsaný jedním jazykem, který je navíc pro většinu programátorů dobře známý, není nutné učit se mnoho nových věcí k tomu, aby se programátor v implementaci již existujícího balíčku zorientoval a mohl jej spravovat. Pro programátory s výrazně vyšší znalostí jazyka C než jazyka Python, mohou být principy tohoto přístupu v porovnání s ostatními (vyžadujícími vyšší znalosti jazyka Python) mnohem lépe pochopitelné. Dá se očekávat, že API, které CPython poskytuje, se nebude zásadně měnit a nebylo by tedy v budoucnu nutné modul kvůli závislosti na tomto API změnám rozhraní. Díky tomu, že tvůrci knihovny libyang jsou primárně

C programátoři, mohl by pro ně tento přístup být nejpraktičtější a v zásadě by pro ně k porozumění nevyžadoval učení ničeho nového.

Nevýhody

Tvůrce balíčku nedostane „nic zadarmo“. Vytvoření kompletního a dobře ošetřeného balíčku oproti ostatním přístupům vyžaduje napsání velkého množství kódu a v porovnání s ostatními přístupy je pro programátora náročnější, zdlouhavější a pracnější.

Správa paměti v modulu je plně v režii programátora, což s sebou nese velký prostor k chybám.

Kvůli tomu, že přístup přímo využívá API, které poskytuje CPython, výsledek nemusí být, a často nejspíše nebude, kompatibilní s alternativními implementacemi jazyka Python. Funkčnost a stabilita je zaručena pouze v kombinaci s implementací CPython. Pro programátory bez větších znalostí jazyka C může být porozumění implementaci modulu problematické. I přesto, že tento přístup má teoreticky potenciál k tomu být nejefektivnější, nebude v základu obsahovat optimalizace, které autoři dále uvedených nástrojů použili. Pro získání nejlepších praktických výsledků, by bylo pravděpodobně nutné, podobné optimalizační mechanismy implementovat přímo do modulu samotného.

CFFI

CFFI (C Foreign Function Interface for Python) je Python modul, který umožňuje interagovat téměř s jakýmkoliv kódem jazyka C z prostředí jazyka Python [18]. Modul nabízí dva módy fungování, a to v závislosti na ABI (Application Binary Interface), nebo na API.

V případě ABI módu umožňuje CFFI načíst dynamickou knihovnu a přímo s ní pracovat. Při použití v API módu je modul přeložen do nativního kódu a využívá API pro tvorbu nativních modulů konkrétní Python implementace.

V obou případech CFFI nad knihovnou jazyka C automaticky pomocí obdržených definic vytvoří mezivrstvu, díky které je knihovna v jazyce Python využitelná téměř stejným způsobem jako v jazyce C. K tomu, aby její používání bylo uživatelsky přívětivé a odpovídalo standardům jazyka Python, je nutné nad touto mezivrstvou vytvořit ještě uživatelské rozhraní.

Pro fungování modul potřebuje informace o částech knihovny, se kterými má umět pracovat, z hlavičkových souborů knihovny [18]. Modul však nedokáže zpracovat všechny výrazy, které se v hlavičkových souborech jazyka C využívají, proto nelze pouze načíst celý hlavičkový soubor, ale musí být nejprve vhodným způsobem upraven.

Vestavěné i odvozené datové typy jazyka C dokáže CFFI mezi oběma jazyky mapovat automaticky. Umožňuje typy jazyka C i přímo alokovat, a v případech, kdy je paměť alokována přes CFFI, postará se o jejich automatické uvolnění.

Výhody

Veškerý kód rozhraní je definovaný v jazyce Python. Tvůrci rozhraní CFFI modul poskytuje řadu funkcí, které mu jeho tvorbu významně usnadní. API CFFI modulu je v porovnání s tím, které je používané u C extension a Ctypes, podstatně jednodušší a z osobních zkušeností jeho pochopení vyžaduje méně času než v případě C extension a Ctypes. V tomto případě je, ze všech zde popisovaných alternativ, tvorba modulu z mého pohledu pro programátora nejjednodušší a nejvíce přívětivá, obzvláště pokud má předchozí zkušenosti s programováním v jazyce Python.

Výsledný modul by měl být kompatibilní se všemi alternativními implementacemi jazyka Python, pro které existuje CFFI modul.

Další podstatnou výhodou je existující komunita zkušených programátorů, kteří stojí za tvorbou neúplného Python rozhraní pro libyang 1.x. V případě použití tohoto přístupu se dá předpokládat větší podpora existující komunity při jeho údržbě.

Nevýhody

Rozhraní knihovny je při vytváření nutné duplikovat, vyžaduje závislost na CFFI modulu. Pokud by se v budoucnu rozhraní CFFI modulu změnilo, bylo by nutné tyto změny reflektovat.

Ctypes

Jedná se o modul pro Python, který umožňuje volat funkce z prostředí jazyka C. Součástí Ctypes modulu jsou i třídy, jejichž instance lze použít jako náhradu primitivních datových typů při volání funkcí z jazyka C. K vytvoření obdoby struktur jazyka C lze vytvořit novou třídu, která bude obsahovat potřebné ekvivalenty datových typů [1].

Při tvorbě modulu tak vzniká jakási mezivrstva Python objektů, které je možné předat funkcím jazyka C a zároveň je možné s nimi pracovat z jazyka Python. Nad touto mezivrstvou, která přesně odpovídá definicím z hlavičkového souboru knihovny, je však, stejně jako v případě CFFI, nutné vytvořit ještě samotné rozhraní pro uživatele.

Výhody

Veškerý kód výsledného balíčku je napsaný v jednom jazyce, a to v jazyce Python. Oproti přístupu C extension je výsledný kód přehlednější a jednodušší. Tvůrce balíčku nepotřebuje větší znalosti jazyka C. Ctypes modul je součástí standardní knihovny jazyka Python. Výsledný balíček není závislý na konkrétní implementaci jazyka Python, ale měl by být použitelný ve všech implementacích, které obsahují Ctypes modul.

Nevýhody

Popis rozhraní knihovny, se kterou má Ctypes modul pracovat, je nutné pomocí alternativní syntaxe duplikovat v jazyce Python, případné změny v rozhraní. Duplikace informací o rozhraní je větší problém než v případě CFFI, a to hlavně z důvodu alternativní syntaxe. Tvorba balíčku je pracnější než v případě CFFI, vyžaduje závislost na Ctypes modulu. Pokud by se v budoucnu rozhraní Ctypes modulu změnilo, bylo by nutné tyto změny reflektovat.

Cython

Samostatný jazyk, který je v zásadě nadmnožinou jazyka Python. Umožňuje spojovat C a Python velmi jednoduše. Cython kód je následně zkompilován a vytvoří se z něj klasický Python modul, který je možné importovat v CPythonu [2].

Výsledek po kompilaci funguje na stejném principu jako v případě C extension. Celý modul je tak ve výsledku tvořen nativním kódem a má potenciál k tomu, být velmi rychlý. Jeden z účelů, ke kterým se Cython běžně využívá, je optimalizace již existujícího Python kódu. Samotná kompilace Python kódu pomocí překladače CPython vede k výraznému zrychlení. Do kódu lze navíc přidat informace o typech jednotlivých proměnných, které

v základu Python kód neobsahuje. Díky informacím o typech je pak kompilátor schopen kód zpracovat ještě lépe a vyprodukovat efektivnější kód.

Výhody

Výsledný kód by měl být z uvedených přístupů nejefektivnější. Po pochopení základů jazyka je propojení knihovny jazyka C s Pythonem podobně jednoduché jako při využití CFFI.

Nevýhody

Stejně jako u Ctypes a CFFI je nutné duplikovat popis rozhraní knihovny. V tomto případě ale stejně jako u CFFI není využívána alternativní syntaxe a často lze použít definice z hlavičkového souboru pouze s drobnými úpravami, díky čemuž je popis rozhraní pro Cython skoro tak jednoduchý jako u CFFI. Funkčnost výsledného modulu je zaručena pouze u tradiční implementace CPython. Celý modul je popsán v jazyce Cython, který v mnoha ohledech připomíná Python, ale s prvky jazyka C. Porozumění implementaci modulu a jeho údržba tedy vyžaduje znalost dalšího jazyka. Výsledný modul je závislý na Cython kompilátoru.

3.3 Porovnání výkonnosti přístupů k implementaci

V této sekci jsou popsány způsoby měření výkonnosti jednotlivých přístupů k implementaci Python bindingu.

Z toho důvodu, že jsem chtěl mít kontrolu nad všemi částmi měření, jsem pro jeho účely vytvořil tři knihovny jazyka C, a to knihovny *libmathops*, *libparams* a *libbintree*. Knihovna libmathops poskytuje základní matematické operace jako sčítání, druhou mocninu, odmocninu atd. Knihovna libparams poskytuje dvě datové struktury s různou velikostí a funkce, které ukazatele na tyto struktury přijímají jako parametry. To poslouží ke zjištění toho, jak jednotlivé přístupy dokáží pracovat se strukturami a jakým způsobem velikost struktur ovlivňuje výkonnost daného přístupu. Knihovna libbintree zase poskytuje naivní implementaci binárního vyhledávacího stromu[1]. Následně jsem nad všemi třemi knihovnami vytvořil identická rozhraní pomocí přístupů dříve uvedených v této kapitole.

Kombinace těchto tří knihoven mi dovoluje měřit, jak se výsledky jednotlivých přístupů chovají při manipulaci se všemi základními datovými typy jazyka C, datovými strukturami, ukazateli, při volání funkcí, předávání parametrů a při správě paměti.

Při všech měřeních je pro eliminování chyby způsobené během ostatních procesů naměřeno větší množství opakování[2] měřeného případu a následně je z nich vybrán ten nejrychlejší, který je považován za výsledek. Chyby měření, které jsou způsobeny kvůli běhu ostatních procesů a vzájemného soupeření procesů o systémové prostředky, mohou měřený případ pouze zpomalit, nikoliv zrychlit, proto je za relevantní považováno nejrychlejší naměřené opakování. V době běhu testů navíc systém nebyl jiným způsobem využíván.

Všechna měření byla provedena na stroji s procesorem Intel Core i5-7300HQ, 8 GB operační paměti a operačním systémem Fedora 32.

[1] Implementace binárního stromu byla vybrán z toho důvodu, že naprostá většina knihovny libyang 2.x pracuje se stromovými strukturami

[2] Počet opakování je závislý na složitosti konkrétního měřeného případu, jednoduché případy jsou opakovány milionkrát, zatímco složitější případy obsahující cykly jsou opakovány tisíckrát.

Použité technologie

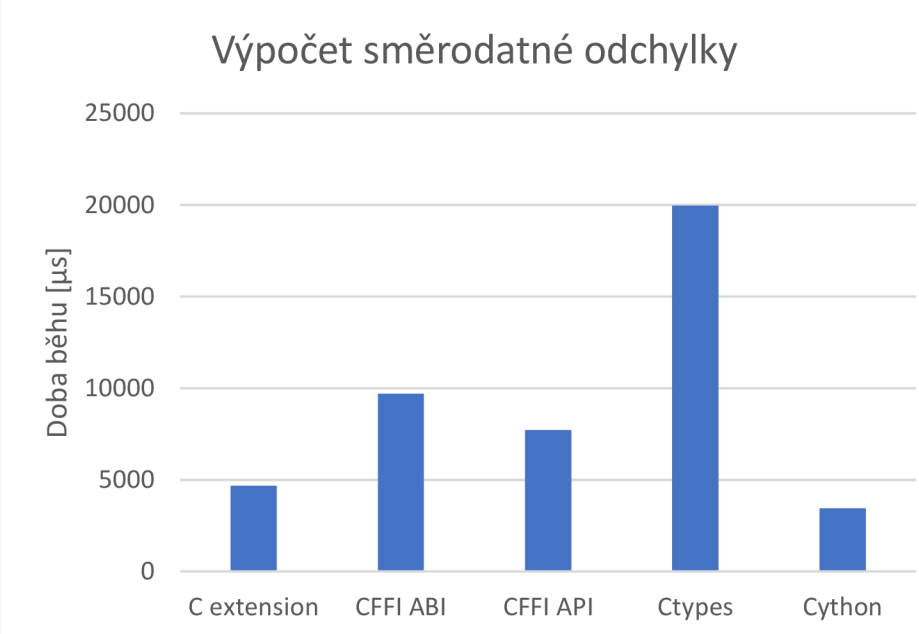
Pro měření rychlosti jednotlivých implementací byl použit modul *timeit*, který umožňuje jednoduchým způsobem měřit rychlost Python kódu a automaticky se vyhýbá chybám, ke kterým při měření doby běhu programu často dochází [5, kapitola Debugging and Profiling].

Měřené případy a jejich výsledky

V této podsekci jsou vysvětleny měřené případy a prezentovány jejich výsledky. Výsledky jsou prezentovány formou grafů a stručným komentářem autora. Kompletní a přesné výsledky všech měření, včetně spustitelných měřících případů, jsou součástí přiloženého paměťového média.

Výpočet statistické odchylky

S využitím Python bindingů pro knihovnu libmathops jsem implementoval algoritmus pro výpočet statistické odchylky ze seznamu čísel. Při měření byl použit list, který obsahoval 10 000 hodnot. Algoritmus implementovaný v jazyce Python tak počítal statistickou odchylku z 10 000 čísel pomocí aritmetických operací implementovaných v jazyce C. Výsledky tohoto měření jsou zobrazeny v grafu 3.1. Na ose x jsou uvedeny jednotlivé měřené přístupy a na ose y doba běhu nejrychlejšího opakování pro daný přístup v mikrosekundách.



Obrázek 3.1: Výsledky měření výpočtu statistické odchylky pomocí matematických operací knihovny jazyka C z prostředí jazyka Python.

Dle očekávání nejlepších výsledků dosáhl přístup využívající Cython, C extension a CFFI jsou na tom ale dost podobně a nejsou ani více než třikrát pomalejší, zatímco Ctypes je nejpomalejší. Tento měřený případ využívá velký počet volání velmi jednoduchých funkcí, které vracejí hodnotu téměř instantně. Naměřené rozdíly v jednotlivých přístupech tak reprezentují převážně jejich režii spojenou s konverzí a předáváním parametrů. Toto měření tedy ukazuje, jak velká režie je spojena s invokací funkce a předáním parametrů u jed-

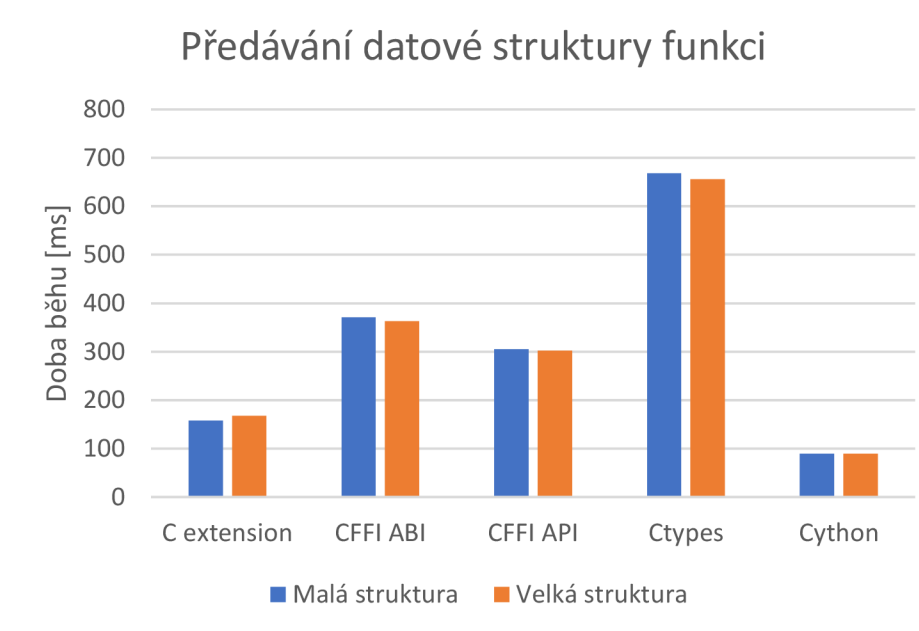
notlivých přístupů. Parametry funkcí však byly poměrně jednoduché, všechny parametry a návratové hodnoty funkcí byly typu `float`. Vliv typu parametrů je zkoumán v následujících dvou měřených případech.

Předávání datových struktur funkci

Předávání struktur do funkcí bude Python binding pro libyang 2.x provádět nejčastěji. Proto jsem vytvořil testovací případ, který měří režii s tím spojenou. Zároveň srovnávám předání malé a velké struktury.

Jako malá struktura byla použita struktura, která v rámci architektury, na které bylo prováděno měření, měla 8 bajtů. Jako velká struktura byla použita struktura s velikostí 536 bajtů. Velikost velké struktury přibližně odpovídá největším strukturám, které jsou součástí veřejného API knihovny libyang 2.x.

Výsledky tohoto měření jsou zobrazeny v grafu 3.2. Na ose x jsou uvedeny jednotlivé měřené přístupy a na ose y doba běhu nejrychlejšího opakování pro daný přístup v milisekundách.



Obrázek 3.2: Výsledky měření předávání datové struktury jako parametr do funkce jazyka C z prostředí jazyka Python.

Přístup využívající Cython je i v tomto případě nejrychlejší, těsně za ním je C extension, které je přibližně jedenapůlkrát pomalejší. Přibližně třikrát pomalejší než Cython je pak CFFI a nejpomalejší je opět přístup využívající Ctypes. I v tomto případě se jedná o velmi jednoduché funkce a rozdíl v režiích u jednotlivých přístupů tak byl dobře měřitelný. Meření také ukazuje, že rozsah velikostí struktur využívaných v API knihovny libyang 2.x není dostatečně velký k tomu, aby u jednotlivých případů způsobil znatelnější zpomalení, rozdíly mezi výsledky při předávání velké a malé struktury v tomto případě nejsou měřitelné a velikost datových struktur používaných ve veřejném API libyang 2.x tak pro výběr způsobu implementace Python bindingu nehraje roli.

Výsledky jsou velmi podobné jako při výpočtu statistické odchylky 3.1. To naznačuje, že konkrétní typ parametru není příliš důležitý a všechny přístupy dokáží Python objekt převést na strukturu jazyka C podobně efektivním způsobem, jako tomu je v případě základních datových typů jazyka C.

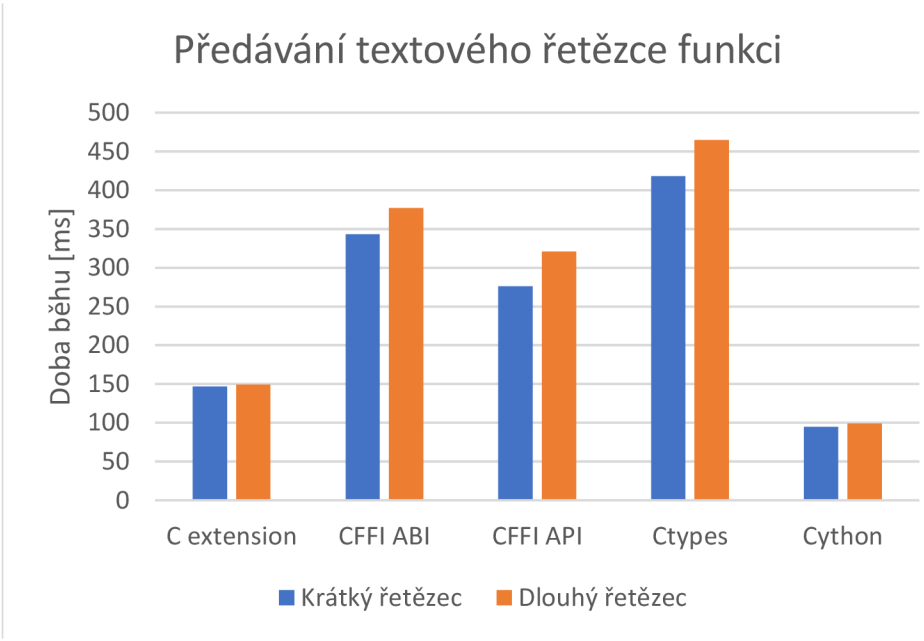
Předávání řetězců funkci

Dalším případem, který jsem se rozhodl změřit, je předávání textového řetězce do funkce. Při předávání řetězce musí Python binding řetězec převést z Python objektu, který jej reprezentuje, na standardní reprezentaci řetězce v jazyce C.

Pro účely měření byly použity dvě různé délky řetězce, a to krátký řetězec o délce 10 znaků a dlouhý řetězec o délce 2 000 000 znaků.

Výsledky jsou velmi podobné jako u předchozích měření. Zdá se tedy, že na konkrétním datovém typu, který je funkci předáván, příliš nezáleží a ve všech případech k převodu dochází podobně efektivním způsobem.

Výsledky tohoto měření jsou zobrazeny v grafu 3.3. Na ose x jsou uvedeny jednotlivé měřené přístupy a na ose y doba běhu nejrychlejšího opakování pro daný přístup v milisekundách.



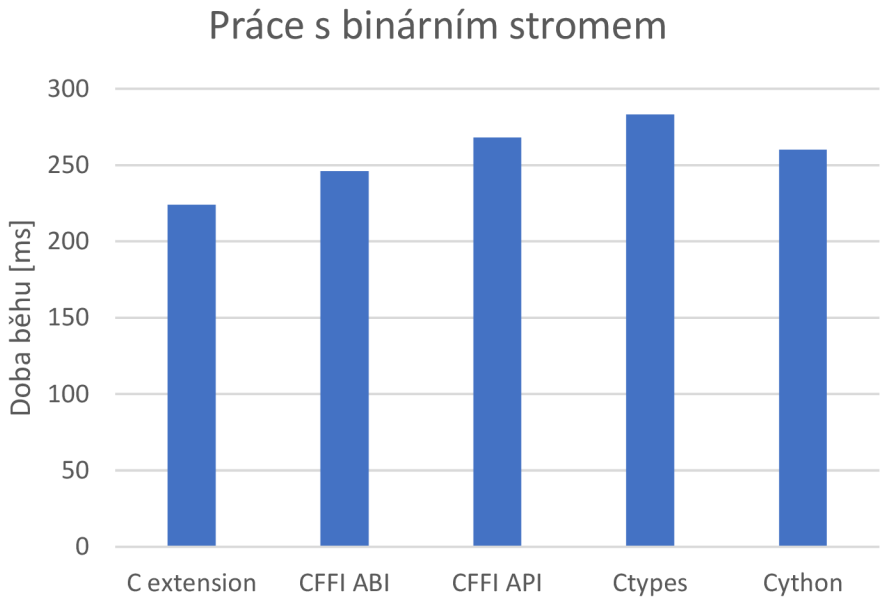
Obrázek 3.3: Výsledky měření předávání textového řetězce jako parametr funkci knihovny jazyka C z prostředí jazyka Python.

Práce s binárním stromem

Z toho důvodu, že libyang 2.x velmi často pracuje se stromovými strukturami, jsem se rozhodl jednu z nich zahrnout i do měření. Během tohoto případu bylo do binárního stromu postupně vkládáno 10 000 uzlů. Po vložení všech uzlů byl postupně měněn datový obsah u poloviny z nich.

Výsledky tohoto případu jsou u jednotlivých případů velmi podobné, to je způsobeno větší složitostí jednotlivých funkcí než u předchozích případů. Režie spojená s konverzí parametrů tím pádem není v porovnání s dobou provádění samotné funkce tak významná.

Výsledky tohoto měření jsou zobrazeny v grafu 3.4. Na ose x jsou uvedeny jednotlivé měřené přístupy a na ose y doba běhu nejrychlejšího opakování pro daný přístup v milisekundách.



Obrázek 3.4: Výsledky měření práce s knihovnou jazyka C, poskytující implementaci binárního vyhledávacího stromu z prostředí jazyka Python.

Zhodnocení a výběr přístupu k implementaci

Měření ukázala, že mezi uvedenými přístupy k implementaci nejsou markantní rozdíly v rychlosti. Je nutné si uvědomit, že režie spojená s převodem typů je přítomna pouze při invokaci funkce jazyka C a přijímání návratové hodnoty funkce. Proto hraje zásadní roli, s jakou granularitou a jak složité funkce jsou invokovány. V případě, kdy byly používány jednoduché funkce, je rozdíl mezi jednotlivými přístupy mnohem znatelnější než v případě, kdy byly používány složitější funkce.

Kvůli tomu, že rozdíly v efektivitě jednotlivých přístupů nejsou při volání složitějších funkcí příliš významné, rozhodl jsem se při výběru implementace klást velký důraz i na jiné aspekty jako jsou jednoduché použití, udržovatelnost výsledného rozhraní a obeznámení s technologiemi u vývojářů libyang 2.x a členů opensource komunity, která kolem knihovny libyang v posledních letech vznikla.

Z toho důvodu bude implementace v rámci této práce provedena pomocí přístupu využívající CFFI, konkrétně v API módu. Při měřeních si přístup využívající CFFI nevedl vůbec špatně. Použití CFFI i následné udržování rozhraní, které je pomocí něj vytvořené, je jednoduché a pro programátora přívětivé. Navíc se dá navázat na neúplnou implementaci, která byla pro libyang 1.x pomocí CFFI vytvořena.

Kapitola 4

Návrh rozhraní

Tato kapitola se nejprve zaměřuje na obecný přístup k návrhu a princip převodu některých konceptů popsaných v části 2.2.2 na koncepty jazyka Python. Následně popisuje hlavní části navrženého rozhraní. V přiložených diagramech jsou využity prvky převzaté z jazyka UML [11] ve spojení s anotacemi pro popis typů [20] a standardními konvencemi pro pojmenovávání identifikátorů [21] využívaných v rámci jazyka Python. Pro zjednodušení nejsou v diagramech zahrnuty vlastnosti třídy `object`, která v jazyce Python 3 definuje základní rozhraní objektů, a třídy v jazyce Python 3 jsou z ní implicitně odvozené. Některé části rozhraní jsou inspirovány neúplným Python 3 rozhraním pro libyang 1.x [14].

4.1 Převod konceptů

Určitou podobnost v obou rozhraních jsem se snažil cíleně zachovat. V případě, že někdo dokáže pracovat se základním rozhraním pro jazyk C, měl by při práci s rozhraním pro Python být i bez čtení další dokumentace schopný přibližně odvodit, jak se jednotlivé metody chovají.

Pro velkou část struktur z rozhraní jsem navrhnul odpovídající třídu, která zastřešuje přístup k datům struktury a zároveň shlukuje operace, které nad ní mohou být provedeny. Jednoduché vlastnosti struktur, které odpovídají jedné hodnotě, jsou transformovány na *property*[1], zatímco přístup ke složitějším hodnotám, které představují kolekce, je zajištěn pomocí veřejných metod. Toto rozhodnutí jsem učinil z důvodu možnosti v budoucnu ke složitějším metodám přidat nepovinné argumenty bez nutnosti provedení zpětně nekompatibilních změn rozhraní. U jednoduchých atributů jako například jméno nebo prefix modulu do budoucna žádnou změnu ani rozšíření při jejich získávání neočekávám a přijde mi mnohem přirozenější přistupovat k nim jako k atributům.

Některé koncepty a ADT, které libyang 2.x využívá, je pro jednoduché použití v rámci jazyka Python nutné převést na jejich Python ekvivalenty. Pracovat s konkrétními bity, nebo s nestandardním rozhraním datového typu, který má ve standardní knihovně jazyka Python svoji implementaci a dlouhodobě zažité rozhraní, by sice bylo možné, ale pro uživatele rozhodně ne příliš pohodlné a intuitivní.

[1] Vlastnost objektu, ke které se přistupuje jako k atributu, ale ve skutečnosti je realizována pomocí jedné nebo více metod.

Kolekce

K uchování kolekcí libyang 2.x využívá několik typů, které mají ve standardní knihovně jazyka Python přímý ekvivalent. Pole a lineární seznamy mohou být převedeny na instance typu `list` a množiny zase na instance typu `set`. Místo přímého převodu na novou instanci dané kolekce, který vede na duplikaci dat, je však z mého pohledu vhodnější nad konkrétními implementacemi datových typů použitých v knihovně libyang 2.x buď vytvořit rozhraní, díky kterému budou použitelné stejným způsobem jako ekvivalentní typy ze standardní knihovny, nebo použít generátory[2]. Z generátoru lze v Pythonu snadno vytvořit instance typu `list` i `set` a ve spojení s dalšími možnostmi jazyku Python jsou generátory velmi mocným nástrojem. Ve všeobecnosti je lepší nabízet uživatelům funkce nebo metody, které vrací generátor než ty, které vrací seznam [22]. Generátory nevyžadují téměř žádnou paměť a uživatelé pak mají možnost pracovat přímo s generátorem, nebo z něj v případě potřeby vytvořit libovolnou kolekci [22]. Z tohoto důvodu jsou v navrženém rozhraní v naprosté většině případů pole, lineární seznamy i množiny z knihovny libyang 2.x převáděny na generátory.

Bitová pole a přepínače

V mnoha případech je v implementaci knihovny libyang 2.x využita přímá manipulace s jednotlivými bity. V rámci možností jazyka C je to často způsob, jak efektivně a pohodlně ukládat hodnoty, které nabývají pouze dvou stavů, nebo konfigurovat složitější chování funkce. Na úrovni jazyka Python je sice možné s jednotlivými bity do jisté míry manipulovat, ale z pohledu uživatele to není příliš pohodlné a osobně manipulaci s konkrétními bity nepovažuji za prostředek, jenž by se měl objevovat v Python rozhraní, které je určené pro uživatele. Python nabízí možnosti, jimiž lze manipulaci s jednotlivými bity od uživatele odstínit a nahradit ji způsobem, který uživateli zpříjemní práci s rozhraním a povede na vznik čitelnějšího kódu. Použití kombinace nastavených bitů při volání funkcí nahrazuji volitelnými pojmenovanými parametry, které předpokládají hodnoty typu `bool`. V případě bitových polí u struktur zase nedávám uživateli k dispozici celé pole, ale pomocí veřejně dostupných metod mu zpřístupňuji vlastnosti, které jednotlivé bity reprezentují.

Typová hierarchie

K vytvoření typové hierarchie, která je v knihovně libyang 2.x vytvořena pomocí z části kompatibilních struktur, je v Pythonu možné využít konceptu dědičnosti. Obecnější třída může obsahovat všechny atributy a metody, které jsou pro podřazené typy společné. Podřazené typy pak všechny společné vlastnosti získají díky dědičnosti od nadřazeného typu a nebude tak docházet k duplikaci kódu a zároveň bude zachována kompatibilita obou typů.

Zpracování chyb

V případě, že dojde k chybě při volání funkcí knihovny libyang 2.x, nebo nastane jiné neočekávané chování spojené s knihovnou, bude vyvolána výjimka typu `LibyangError` obsahující popis poslední chyby, kterou knihovna uložila do svého logu. Tato vlastnost může být pro budoucí uživatele knihovny velmi užitečná při hledání chyb v programu.

[2] Generátory jsou v jazyce Python funkce, které postupně vracejí prvky z kolekce hodnot, kterou reprezentují.

4.2 Práce s kontexty

Pro pokrytí většiny práce s kontextem jsem navrhl třídu `Context`. Třída `Context` je vyobrazena v diagramu 4.1.

Manipulace s vlastnostmi kontextu

Pro úpravu chování kontextu po jeho vytvoření lze využít následující property třídy `Context`.

- Property `searchdirs` uživatel může pouze číst, vrací instanci třídy `ContextSearchdirsSet`, pomocí které mohou být odebírány, přidávány a prohlíženy aktuálně nastavené cesty k prohledávání. `ContextSearchdirsSet` implementuje stejné rozhraní jako vestavěný typ `set`.
- Všechny následující property svým chováním odpovídají přepínačům *Context options* z knihovny libyang 2.x [16]. Property `noyanglibrary` může uživatel pouze číst. `allimplemented`, `disable_searchdir_cwd`, `disable_searchdirs`, `trusted` a `prefer_searchdirs` může uživatel číst i modifikovat.

Načítání modulů

- `parse_module` slouží k načtení modulu z textového řetězce, nebo ze souboru. Zdroj obsahující definici modulu metoda přijímá prostřednictvím parametru `source`. Zda se jedná o soubor, nebo řetězec metoda rozpozná automaticky. Pro výběr formátu načítaného modulu může uživatel použít parametr `fmt`, který akceptuje řetězce `yang` (v případě formátu YANG) a `yin` (v případě formátu YIN). Není-li hodnota parametru `fmt` při invokaci metody specifikována, předpokládá se jako výchozí hodnota `yang` a očekává se, že načítaný modul je specifikován ve formátu YANG. Metoda vrací instanci třídy `SModule`.
- `load_module` se pokusí najít a načíst modul s danými parametry. Jméno modulu, který má být načten, je specifikováno povinným parametrem `name`. Pomocí volitelného parametru `revision` může být upřesněna požadovaná revize modulu. Pokud revize není specifikována, vyhledává se modul s nejnovější revizí. Modul je hledán v cestách k prohledávání, které jsou v kontextu nastaveny. V případě, že už je v kontextu načten modul s požadovaným jménem, metoda vrací již načtený modul bez ohledu na jeho revizi a další hledání modulu neprovádí. Metoda vrací nově načtenou, nebo v kontextu již nalezenou instanci třídy `SModule`.
- Metoda `reset_latest` resetuje cache obsahující informace o nejnovější revizi modulů v kontextu. Uživatel by tuto metodu měl invokovat v případě, že se obsah cest k vyhledávání modulů změní až po vytvoření modulu, nebo po přidání poslední cesty do množiny cest.

Vyhledávání modulů

Pro získání načtených modulů z kontextu jsou k dispozici následující metody:

- `get_modules` nepřijímá žádné parametry a po invokaci vrací generátor všech modulů načtených v kontextu.

- Metody `get_module`, `get_module_implemented` a `get_module_latest` slouží k vyhledání jednoho konkrétního modulu. Všechny metody z této skupiny mají nepovinný argument `use_namespace`, který při nastavení na hodnotu `True` způsobí, že modul bude vyhledáván podle atributu `namespace` a ne podle atributu `name`. Defaultně je `use_namespace` nastavený na hodnotu `False` a metody vyhledávají modul podle jména. Metoda `get_module` vyhledává modul se zadaným klíčem a revizí, `get_module_implemented` vyhledává implementovaný modul se zadaným klíčem a metoda `get_module_latest` vyhledává modul s nejnovější revizí a zadaným klíčem.

Načítání dat

- Metoda `parse_data` slouží k načtení instančních dat z textového řetězce, nebo ze souboru. Zdroj obsahující definici dat metoda přijímá prostřednictvím parametru `source`. Zda se jedná o soubor, nebo řetězec metoda rozpozná automaticky. Pro výběr formátu načítaných dat může uživatel použít parametr `format`, který přijímá řetězce `xml` (v případě formátu XML) a `lyb` (v případě formátu LYB). Není-li hodnota parametru format při invokaci metody specifikována, předpokládá se jako výchozí hodnota `xml` a očekává se, že načítaná data jsou ve formátu XML. Metoda vrací instanci třídy `DNode`. Ostatní parametry, které metoda přijímá svým chováním i názvem odpovídají přepínačům *Data validation options* a *Data parser options* z knihovny libyang 2.x [16].

- Metoda `dnode_new_path` vytváří nový datový uzel na základě cesty. Cesta je specifikována povinným parametrem `path`, volitelná hodnota může být specifikována parametrem `value`, případný typ hodnoty je specifikován argumentem `value_type`. Hodnota je využita pouze při vytváření uzlů typu leaf, leaf-list, anyxml a anydata. Typ hodnoty je požadován pouze v případě vytváření uzlu typu anydata, nebo anyxml a je specifikován parametrem `value_type`, který přijímá řetězce `datatree`, `string`, `xml`, `json`, `lyb`. Ostatní parametry, které metoda přijímá, svým chováním i názvem odpovídají přepínačům *Data path creation options* z knihovny libyang 2.x [16].

Speciální metody

- Metoda `destroy` umožňuje uživateli explicitně uvolnit všechny systémové prostředky, které jsou pro kontext vyhrazeny, a informuje všechny instance tříd, které vlastní v kontextu uložená data o tom, že jejich data nejsou nadále platná. Po invokaci této metody by uživatel neměl pracovat s kontextem, ani s žádnou z jeho částí.

- Inicializátor `__init__` přijímá jako argumenty vlastnosti, které lze kontextu při vytvoření nastavit. Jejich názvy odpovídají přepínačům *Context options* z knihovny libyang 2.x [16]. Jako hodnoty jednotlivých vlastností jsou předpokládány hodnoty typu `bool`. Vlastnost kontextu `noyanglibrary` může být nastavena pouze při vytváření kontextu. Cesty k vyhledávání mohou být nastaveny pomocí parametru `searchdirs`, který přijímá iterable textových řetězců.

- Díky kombinaci metod `__enter__` a `__exit__` může být instance třídy použita jako tzv. *context manager* [19] s klíčovým slovem `with` [19].

- Finalizátor `__del__` [17] se postará o uvolnění všech prostředků, které jsou pro `Context` vyhrazeny, a informuje všechny instance tříd, které vlastní v kontextu uložená

data o tom, že jejich data nejsou nadále platná. Je automaticky invokován ve chvíli, kdy na daný kontext, ani žádnou z jeho částí uživatel nemá uloženou referenci a nemá ke kontextu tedy žádným způsobem přístup.

- Metoda `__iter__` umožňuje nad kontextem iterovat. Metoda vrací generátor všech modulů načtených v kontextu.

Obrázek 4.1: Třída `Context`

4.3 Práce se schématy

Jako hlavní prostředek pro práci se schématy jsem navrhnul třídu `SModule`. Třída `SModule` je vyobrazena v diagramu 4.2. Práci s parsovanou verzí schématu pak zajišťuje třída `SPModule`, která je popsána v podkapitole 4.4 a práci s kompilovanou verzí schématu zase třída `SCModule`, která je popsána v podkapitole 4.5.

Základní vlastnosti modulu

Přístup k základním vlastnostem modulu zajišťují následující property třídy `SModule`. Všechny jsou určeny pouze ke čtení.

- `parsed` slouží k přístupu k parsované verzi modulu a vrací instanci třídy `SPModule`.
- `compiled` slouží k přístupu ke kompilované verzi modulu a vrací instanci třídy `SCModule`.
- `context` vrací kontext, ve kterém je daný modul načten.
- `name` vrací název modulu.
- `description` vrací popis modulu.
- `reference` vrací hodnotu YANG výrazu reference, který k modulu patří.
- `revision` vrací volitelnou nejnovější revizi modulu.
- `namespace` vrací povinnou hodnotu YANG výrazu namespace, který k modulu patří.
- `prefix` vrací povinný prefix modulu.
- `filepath` vrací cestu k souboru modulu, pouze pokud byl modul načten ze souboru.
- `organization` vrací volitelné informace o organizaci, která modul vytvořila.
- `contact` vrací volitelnou hodnotu YANG výrazu contact, který k modulu patří.
- `implemented` vrací hodnotu `True` pokud je modul v kontextu implementovaný, jinak vrací hodnotu `False`.
- `version` vrací řetězec `YANG 1.0` v případě, že byl modul vytvořen podle jazyka YANG verze 1.0 a řetězec `YANG 1.1` v případě, že byl modul vytvořen podle jazyka YANG verze 1.1.

Metody pro práci s YANG výrazy feature patřící modulu.

Všechny metody pro aktivaci a deaktivaci přijímají nepovinný pojmenovaný parametr `force`, jehož nastavení způsobí, že daná operace bude provedena bez ohledu na ostatní feature, které modul má.

- `feature_enable` aktivuje specifikovanou feature daného modulu s daným názvem. Název je specifikován povinným parametrem `name`.
- `feature_enable_all` aktivuje všechny feature daného modulu.

- `feature_disable` deaktivuje specifikovanou feature daného modulu. Název je specifikován povinným parametrem `name`.
- `feature_disable_all` deaktivuje všechny feature daného modulu.
- `feature_value` vrací hodnotu konkrétní feature modulu s daným názvem. Název je specifikován povinným parametrem `name`. Návratová hodnota je typu `bool`.

Implementování modulu do kontextu

Implementování modulu do kontextu může být explicitně provedeno voláním metody `implement`. Metoda nepřijímá žádné parametry a může být invokována i nad již implementovaným modulem, v takovém případě nic nemění.

Serializace modulu

Pro serializaci modulu do textové podoby slouží metoda `print`. Přijímá volitelný parametr `target` specifikující cíl pro tisk, který v současné verzi může být pouze reference na soubor otevřený pro zápis. Tento návrh funkce však dovoluje do budoucna přidat další cíle pro tisk bez toho, aby musely být provedeny zpětně nekompatibilní změny rozhraní. Pokud není cíl pro tisk zadán, vrací metoda textový řetězec se serializovaným modulem.

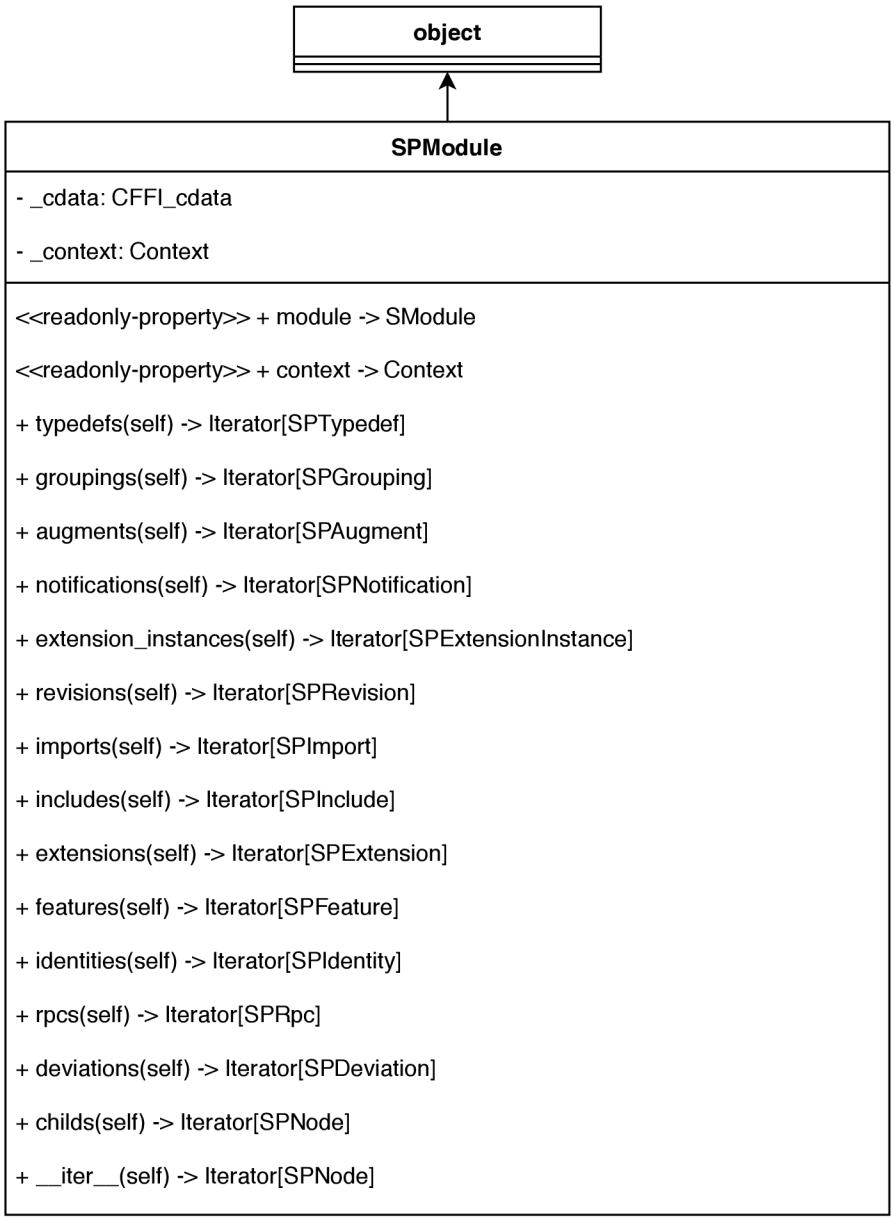
Validace datového stromu vůči modulu

Pro validaci datového stromu vůči modulu lze použít metodu `validate_tree`. Ta akceptuje parametr tree typu `DNode`, který reprezentuje uzel nejvyšší úrovně datového stromu pro validaci. Ostatní parametry, které metoda přijímá, svým chováním i názvem odpovídají přepínačům *Data validation options* z knihovny libyang 2.x [16].

Obrázek 4.2: Rozhraní třídy `SModule`

4.4 Přístup k parsované verzi modulu

Pro přístup k parsované verzi modulu a jeho stromové struktuře jsem navrhnul třídy `SPModule` a `SPNode`. Ze třídy `SPNode` jsou odvozeny všechny třídy, jejichž instance se mohou nacházet ve stromové struktuře parsovaného modulu. Konkrétně se jedná o tyto specializované třídy: `SPContainer`, `SPLeaf`, `SPLeafList`, `SPList`, `SPCase`, `SPAnydata`, `SPAnyxml`, `SPUses`, `SPAction`, `SPRpc`, `SPAugment`, `SPGrouping`, `SPNotification` a `SPInputOutput`. Pro jednoduchost nejsou detailní vlastnosti specializovaných tříd uváděny. Detaily o nich mohou být dohledány ve zdrojovém kódu a dokumentačních komentářích. Třída `SPModule` je vyobrazena v diagramu 4.3 a třída `SPNode` v diagramu 4.4.



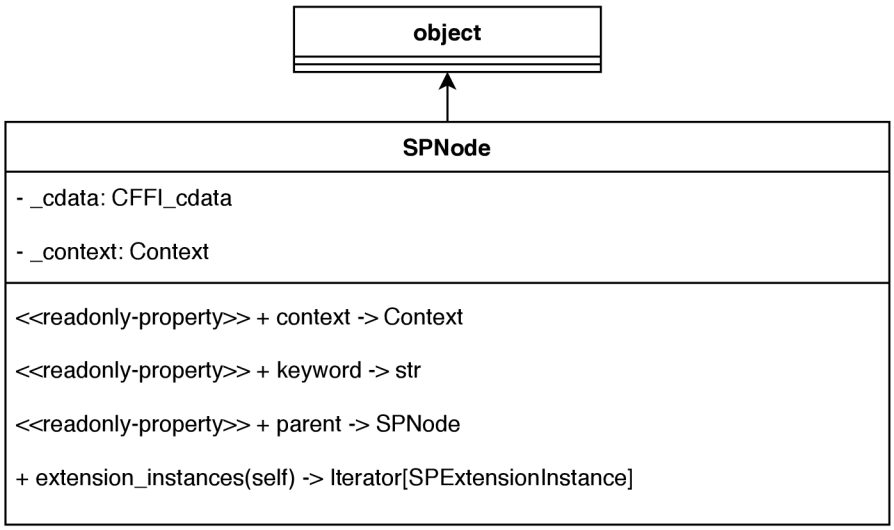
Obrázek 4.3: Třída `SPModule`

Základní vlastnosti parsovaného modulu

- Property `module` slouží pouze ke čtení a vrací instanci třídy `SModule`, jehož parsovanou verzi tento modul představuje.
- Property `context` slouží pouze ke čtení, vrací instanci třídy `Context`, do které tento modul patří.
- Metody `typedefs`, `groupings`, `augments`, `notifications`, `extension_instances`, `revisions`, `imports`, `includes`, `extensions`, `features`, `identities`, `rpcs` a `deviations` slouží k získání všech YANG výrazů daného typu definovaných v modulu. Jednotlivé generátory vrací instance tříd, které konkrétní YANG výraz reprezentují, návratové typy jednotlivých metod jsou zobrazeny v diagramu 4.3.
- Metoda `childs` vrací generátor všech uzlů nejvyšší úrovně, které jsou v modulu definovány. Generátor vrací instance třídy `SPNode` a od ní odvozených specializovaných typů.

Speciální vlastnosti parsovaného modulu

Díky implementaci metody `__iter__` je nad modulem možné iterovat. Metoda vrací generátor všech uzlů nejvyšší úrovně.



Obrázek 4.4: Třída `SPNode`

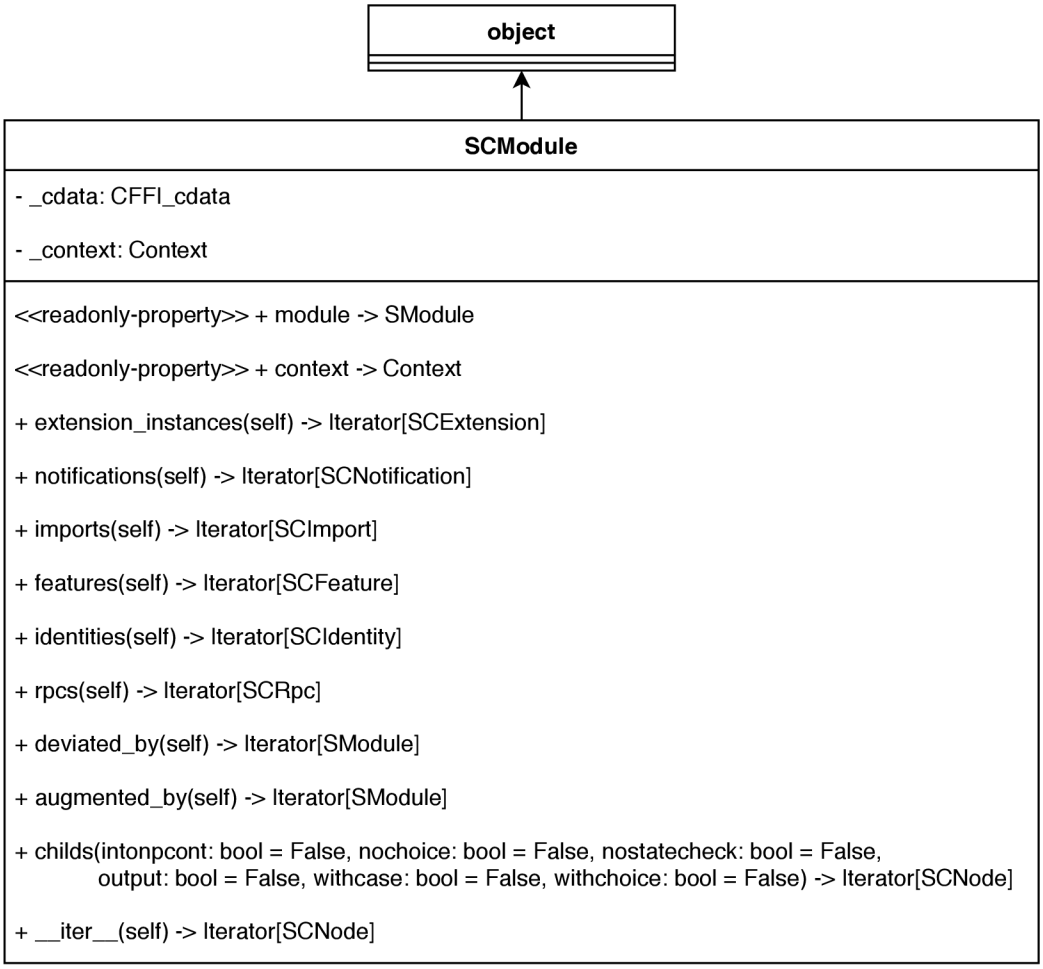
Základní vlastnosti parsovaného uzlu

- Property `context` je určena pouze ke čtení a slouží k získání kontextu, do kterého modul patří. Vrací instanci třídy `Context`.
- Property `keyword` je určena pouze ke čtení a vrací textovou reprezentaci YANG výrazu, který uzel reprezentuje. Návratová hodnota se liší u jednotlivých specializovaných tříd.

- Property `parent` je určena pouze ke čtení a vrací nadřazený uzel. V případě, že se jedná o uzel nevyšší úrovně a nemá tedy nadřazený uzel, vrací hodnotu `None`.
- Metoda `extension_instances` slouží k získání instancí rozšíření, které jsou v daném uzlu použity. Vrací generátor typu `SPExtensionInstance`.

4.5 Přístup ke kompilované verzi modulu

Pro přístup ke kompilované verzi modulu a jeho stromové struktuře jsem navrhnul třídy `SCModule` a `SCNode`. Ze třídy `SCNode` jsou odvozeny všechny třídy, jejichž instance se mohou nacházet ve stromové struktuře parsovaného modulu. Konkrétně se jedná o tyto specializované třídy: `SCContainer`, `SCCase`, `SCCHoice`, `SCLeaf`, `SCLeafList`, `SCList`, `SCAnydata`, `SCAnyxml`, `SCAction`, `SCRpc`, `SCNotification`. Pro jednoduchost nejsou detailní vlastnosti specializovaných tříd uváděny. Třída `SCModule` je vyobrazena v diagramu 4.5 a třída `SCNode` v diagramu 4.6.



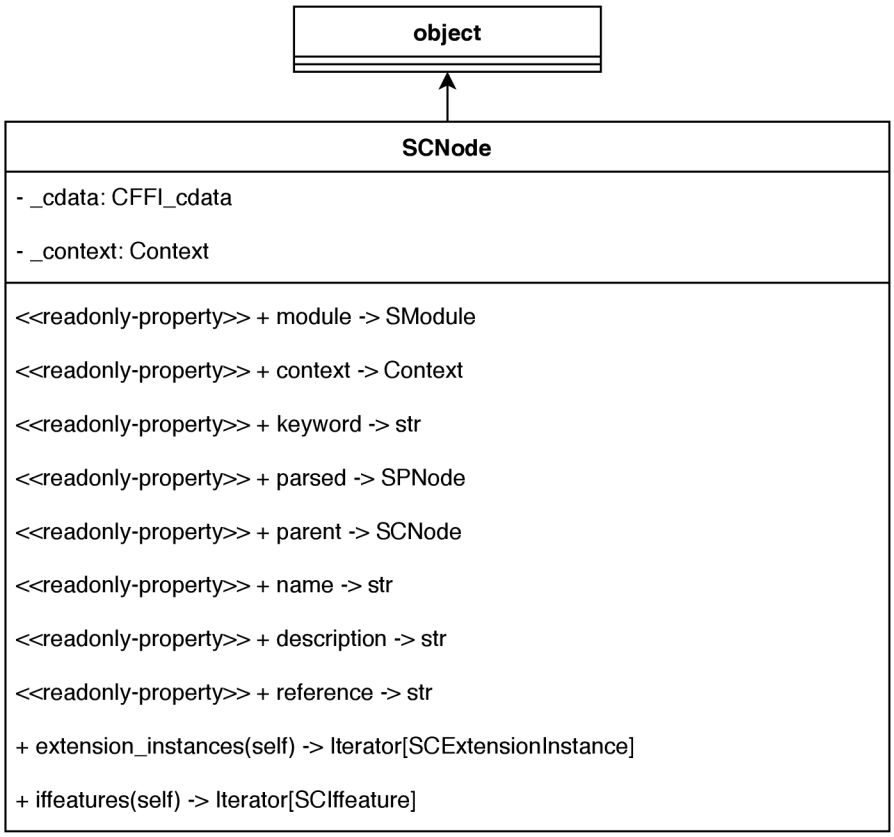
Obrázek 4.5: Třída `SCModule`

Základní vlastnost kompilovaného modulu

- Property `module` slouží pouze ke čtení a vrací instanci třídy `SModule`, jehož kompilovanou verzi tento modul představuje.
- Property `context` slouží pouze ke čtení, vrací instanci třídy `Context`, do které tento modul patří.
- Metody `notifications`, `imports`, `features`, `identities`, `rpcs` slouží k získání všech YANG výrazů daného typu, které modulu patří. Jednotlivé generátory vrací instance tříd, které konkrétní YANG výraz reprezentují. Návratové typy jednotlivých metod jsou zobrazeny v diagramu 4.5.
- Metoda `childs` vrací generátor všech uzlů nejvyšší úrovně, které modulu patří. Generátor vrací instance třídy `SCNode` a od ní odvozených specializovaných typů.

Speciální vlastnosti kompilovaného modulu

Díky metodě `__iter__` je možné nad instancemi třídy `SCModule` iterovat, vrací generátor uzlů nejvyšší úrovně. Generátor vrací instance třídy `SCNode` a od ní odvozených specializovaných typů.



Obrázek 4.6: Třída `SCNode`

Základní vlastnosti kompilovaného uzlu

- Property `module` je určena pouze ke čtení a vrací instanci třídy `SModule`, ke kterému uzel patří.
- Property `context` slouží pouze ke čtení, vrací instanci třídy `Context`, do které tento modul patří.
- Property `keyword` je určena pouze ke čtení a vrací textovou reprezentaci YANG výrazu, který uzel reprezentuje. Návratová hodnota se liší u jednotlivých specializovaných tříd.
- Property `parsed` je určena pouze ke čtení a vrací parsovanou verzi daného uzlu. Návratová hodnota je instance třídy `SCNode` a od ní odvozené specializované typy.
- Property `name` slouží pouze ke čtení a vrací název uzlu. Návratová hodnota je typu `str`.
- Property `description` je určena pouze ke čtení a vrací popis uzlu. Návratová hodnota je typu `str`.
- Property `reference` je určena pouze ke čtení a vrací hodnotu YANG výrazu reference, který k danému uzlu patří. Návratová hodnota je typu `str`.
- Metoda `extension_instances` slouží k získání instancí rozšíření, které k danému uzlu patří. Vrací generátor typu `SCExtensionInstance`.
- Metoda `iffeatures` slouží k získání YANG výrazů iffeature, které k danému uzlu patří. Vrací generátor typu `SCIffeature`.

4.6 Práce s daty

Pro přístup k datovým stromům a manipulaci s nimi slouží primárně třída `DNode`, která je vyobrazena v diagramu 4.7.

Základní vlastnosti datového uzlu

- Property `Context` je určena pouze ke čtení a slouží k získání kontextu, ke kterému datový uzel patří. Návratová hodnota je typu `Context`.
- Propert `schema` je určena pouze ke čtení, vrací `SCNode`.
- Property `parent` slouží pouze ke čtení a vrací rodičovský uzel daného uzlu. V případě, že se jedná o uzel nejvyšší úrovně a nemá tedy rodičovský uzel, vrací hodnotu `None`. Návratová hodnota je typu `DNode`.
- Property `name` vrací název uzlu. Návratová hodnota je typu `str`.
- Property `module` je určena pouze ke čtení a vrací modul, ke kterému datový uzel patří. Návratová hodnota je typu `SModule`.
- Metoda `siblings` vrací generátor uzlů na stejné úrovni. Generátor vrací typ `DNode`.
- Metoda `metadata` vrací generátor metadat datového uzlu. Generátor vrací typ `DMeta`.

Validace

Metoda `validate` validuje datový strom vůči kontextu, v rámci kterého byl vytvořen. Metoda přijímá dva volitelné pojmenované parametry, jejichž pojmenování a chování odpovídá přepínačům *Data validation options* z knihovny libyang 2.x [16].

Vyhledávání

Metoda `find_xpath` vyhledá uzly specifikovány pomocí XPath výrazu. Výraz je specifikován pomocí povinného parametru `xpath`. Metoda vrací generátor všech uzlů, které odpovídají výrazu. Generátor vrací typ `DNode`.

Rozšiřování stromu

Datový strom lze rozšiřovat pomocí těchto metod:

- Metoda `newchild_any` vytvoří nový podřazený uzel, který ve schématu odpovídá YANG výrazu anydata, nebo anyxml. Jméno uzlu definovaného ve schématu specifikuje povinný parametr `name`. Hodnotu uzlu určuje povinný parametr `value` a její typ povinný parametr `value_type`. Volitelně lze uvést modul, ve kterém je uzel s názvem `name` definován pomocí parametru `module`. Pokud modul není specifikován, předpokládá se modul patřící k rodičovskému uzlu.

- Metoda `newchild_inner` vytvoří nový podřazený uzel, který ve schématu odpovídá YANG výrazu container, notification, rpc, nebo action. Název uzlu ve schématu je specifikován povinným parametrem `name`. Volitelně lze uvést modul, ve kterém je uzel s názvem `name` definován pomocí parametru `module`. Pokud modul není specifikován, předpokládá se modul patřící k rodičovskému uzlu.

- Metoda `newchild_list` vytvoří nový podřazený uzel, který ve schématu odpovídá YANG výrazu list. Název uzlu ve schématu je specifikován povinným parametrem `name`. Parametrem `keys` lze specifikovat klíče listu. Volitelně lze uvést modul, ve kterém je uzel s názvem `name` definován pomocí parametru `module`. Pokud modul není specifikován, předpokládá se modul patřící k rodičovskému uzlu.

- Metoda `newchild_term` vytvoří nový podřazený uzel, který ve schématu odpovídá YANG výrazu leaf, nebo leaflist. Název uzlu ve schématu je specifikován povinným parametrem `name` a hodnota uzlu povinným parametrem `value`. Volitelně lze uvést modul, ve kterém je uzel s názvem `name` definován pomocí parametru `module`. Pokud modul není specifikován, předpokládá se modul patřící k rodičovskému uzlu.

- Metoda `newchild_opaq` vytvoří nový podřazený uzel, který není definován ve schématu. Název uzlu je specifikován parametrem `name` a jeho hodnota parametrem `value`. Volitelně lze uvést modul, parametrem `module`. Pokud modul není uveden, předpokládá se modul rodičovského uzlu.

- Metoda `merge` do datového stromu sloučí jiný datový strom. Argumentem `source` je specifikován kořen stromu, který má být sloučen. Parametr `with_siblings` specifikuje, zda má být `source` sloučen včetně svých sousedních uzlů. Ostatní parametry odpovídají přepínačům data merge options z knihovny libyang 2.x [16].

- Metody `insert_child`, `insert_before`, `insert_after`, `insert_sibling` do datového stromu vloží nový uzel. Všechny přijímají parametr `node`, který označuje uzel, jenž má být přidán. Pokud uzel `node` patří do jiného datového stromu, je z tohoto stromu odebrán.

Odebrání uzlů

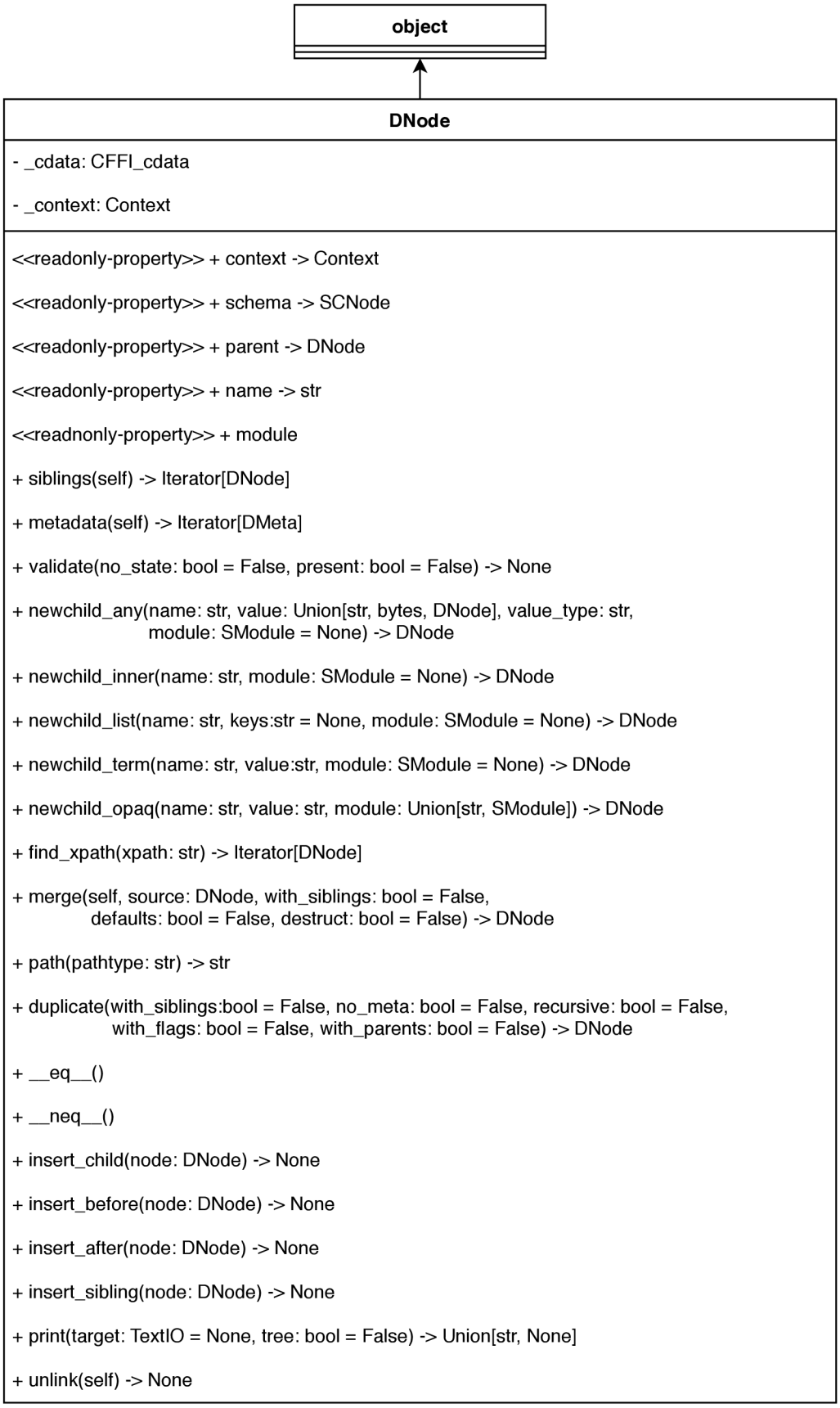
Uzel a podstrom, který reprezentuje, je možné ze stromu odebrat invokací metody `unlink`.

Serializace

Pro serializaci dat slouží metoda `print`, jež přijímá volitelný parametr `target`, kterým je možné určit soubor, do kterého má být výstup zapsán. Pokud není `target` specifikován, vrací metoda serializovaný řetězec jako návratovou hodnotu.

Speciální vlastnosti datového uzlu

Díky metodám `__eq__` a `__neq__` mohou být jednotlivé uzly mezi sebou korektně porovnávány pomocí standardních operátorů jazyka Python.



Obrázek 4.7: Třída `DNode`

Kapitola 5

Implementace

V této kapitole jsou shrnuty implementační detaily práce, včetně některých zajímavých problémů, se kterými jsem se při implementaci potýkal, a popis jejich řešení. Některé části implementace jsou inspirovány implementací neúplného Python rozhraní pro libyang 1.x [14] a skripty pro překlad modulu vytvořeného s využitím CFFI a spouštění testů jsou z ní přímo převzaty.

V přiložených diagramech jsou využity stejné zvyklosti jako u diagramů v kapitole 4.

5.1 Členění zdrojového kódu

Implementace je rozdělena do dvou částí. První jsou definice a skripty potřebné pro vygenerování základního Python wrapperu pomocí CFFI, který je v kapitole i v kódu označován názvem `_libyang 2.x`. Druhou důležitou částí je balíček `libyang 2.x-python` implementující rozhraní určené pro uživatele.

Naprostá většina implementační práce byla spojena s implementací obalovacích tříd[1].

Implementace balíčku `libyang 2.x-python` je rozdělena do následujících Python modulů:

- `context`, ve kterém je obsažena třída `Context` a všechny ostatní třídy tvořící rozhraní nad libyang 2.x kontexty.
- `data` obsahující třídu `DNode` a všechny ostatní třídy, které tvoří rozhraní nad datovými stromy. Jeho součástí jsou také funkce, které se interně využívají v souvislosti s datovými stromy.
- `log` s definicemi pro konfiguraci logování.
- `schema` se třídou `SModule`, která tvoří obecné rozhraní nad oběma typy schématu. Obsahuje také funkce, které jsou interně využívány v souvislosti se schématy.
- `schema_parsed` s definicí třídy `SPModule` a ostatních tříd, které tvoří rozhraní nad parsovanými schématy.
- `schema_compiled`, ve kterém je obsažena definice třídy `SCModule` a všech ostatních tříd, které tvoří rozhraní nad kompilovanými schématy.

[1] Třídy, jejichž instance slouží k obalení dat, které odpovídají datovým strukturám knihovny libyang 2.x.

- `utils` obsahuje všechny definice, které jsou využívány v ostatních modulech a zároveň přímo neodpovídají primárnímu zaměření žádného z nich.

Pro automatizované vygenerování balíčku `_libyang 2` pomocí CFFI jsou vytvořeny tyto soubory:

- `cdefs.h` s deklaracemi z rozhraní knihovny libyang 2.x, které má CFFI obalit.
- `source.c` s definicemi jazyka C, které jsou ke knihovně pro účely tvorby Python rozhraní přidány.
- `build.py`, ve kterém jsou informace o tom, jak a z čeho má být pomocí CFFI `_libyang2` modul vygenerován. Obsah tohoto souboru byl s úpravami převzat z [14].

5.2 Některé problémy a jejich řešení

V této sekci jsou zmíněny některé zajímavější problémy, na které jsem během implementace narazil, a způsoby, jakými jsem je vyřešil.

Nejednoznačnost typů a struktur knihovny libyang 2.x

Několik datových struktur použitých v knihovně libyang 2.x je využíváno k reprezentaci více různých druhů dat. Například strukturu `struct lysp_type_enum` libyang 2.x využívá k reprezentaci YANG výrazů `enum` i `bit`. O který konkrétní výraz se jedná, je jednoznačné až z toho, kde je struktura použita. Kvůli tomu není vždy ze samotného typu dat jednoznačné, která obalovací třída by nad nimi měl být instanciována.

V jazyce Python je na rozdíl od jazyka C možné dotazovat se na typy jednotlivých proměnných v rámci programu. Výrazy enum a bit se navíc dle [8] liší v názvu jedné položky (i když její obsah je stejného typu), proto bych reprezentaci těchto výrazů v rozhraní pro Python rád oddělil. Stejný problém nastává i v několik dalších případech, kdy libyang reprezentuje více YANG výrazů pomocí jedné struktury.

Dalším důvodem vzniku nejednoznačností mezi typem dat a instancí obalovací třídy, která pro ni má být vytvořena, je datová hierarchie, ve které jsou podřazené typy reprezentovány instancemi nadřazených typů. To, jakým způsobem libyang 2.x implementuje typovou hierarchii, bylo popsáno v kapitole 2.

Řešení nejednoznačnosti typů je vysvětleno v následující podsekci.

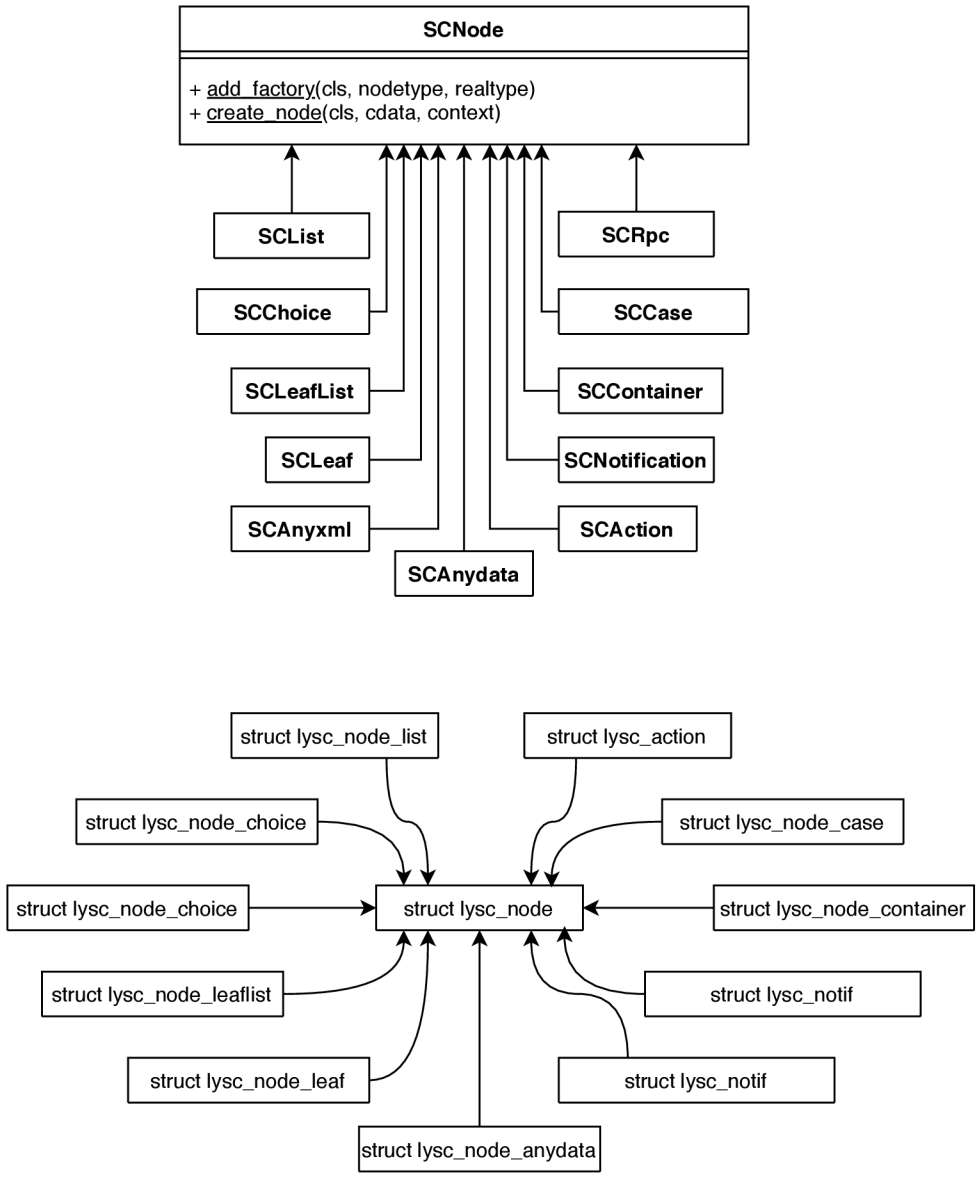
Vytváření instancí obalovacích tříd pro hierarchické typy

Každá typová hierarchie, kterou libyang 2.x definuje, má specifická pravidla pro rozlišení konkrétního typu. Proto jsem třídu reprezentující typ na nejvyšší úrovni vytvořil s využitím návrhového vzoru tovární metoda (*factory method*). Třída reprezentující typ, který je v hierarchii na nejvyšší úrovni, tudíž obsahuje veškerou logiku, která je s vytvářením obalovacích tříd pro danou typovou hierarchii spojena.

Obalovací třídy pro typy z dané hierarchie se registrují pomocí dekorátoru, jenž třída na nejvyšší úrovni poskytuje.

Příklad jedné z typových hierarchií, kterou knihovna libyang 2.x definuje, a její odpovídající reprezentace v rozhraní balíčku `libyang 2.x-python` je vyobrazena na obrázku 5.1. Třída SCNode je prostřednictvím metody `create_node` zodpovědná za vytváření instance obalovací třídy pro všechna data typu `struct lysc_node`. To kterou má pro obalení vybrat

třídu, rozpozná podle atributu `nodetype`. Kromě výběru obalovací třídy musí při vytváření zajistit i přetypování dat na odpovídající typ jazyka C. Odvozené třídy se do role „továren“ registrují pomocí dekorátoru `add_factory`.



Obrázek 5.1: Implementace typové hierarchie s využitím nadřazeného typu jako tovární metody (nahoře) a odpovídající typová hierarchie knihovny libyang 2.x (dole). U tříd pro jednoduchost nejsou uváděny vlastnosti, které nesouvisí s reprezentací a instanciací odpovídající typové hierarchie.

Sjednocení instanciace obalovacích tříd

Pro sjednocení a automatizaci výběru konkrétního typu obalovací třídy jsem vytvořil třídu `WrapperFactory`, která využívá návrhového vzoru tovární metoda.

Třída `WrapperFactory` je ilustrována na obrázku 5.2. Metody s prefixem `register` poskytují dekorátory pro registraci jednotlivých „továren“. Metoda `wrap` slouží k obalení konkrétní instance dat. Metoda `arr2gen` slouží k převodu sized_array na generátor instancí obalovacích tříd, které jednotlivé prvky pole reprezentují. Metoda `ll2gen` má za úkol převod spojovaného seznamu na generátor instancí obalovacích tříd, které jednotlivé prvky seznamu reprezentují.

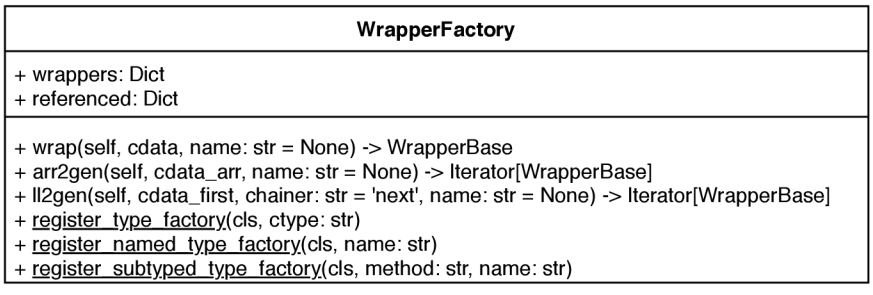
Instance třídy `WrapperFactory` je součástí všech instancí třídy `Context` a ve většině případů dokáže automaticky odvodit, který obalovací typ má pro která data použít. Kvůli tomu, že instanciace obalovacích tříd je v rozhraní velmi častá, je jeho zjednodušení žádoucí.

Jednotlivé „továrny“ je nutné do `WrapperFactory` nejprve přidat a to pomocí jednoho z dekorátorů, které poskytuje. „Továrny“ je možné přidávat podle následujících kritérií:

- Podle názvu typu dat, která k přidávané obalovací třídě patří. Tento způsob používají třídy, které přímo a jednoznačně odpovídají jedné konkrétní datové struktuře knihovny libyang 2.x.
- Podle názvu, který je předávaný jako volitelný parametr. Tento způsob umožňuje použití `WrapperFactory` i pro data s nejednoznačným typem. Využívají ho obalovací třídy, které obalují datový typ, jenž má v rozhraní libyang 2.x více reprezentací.
- Podle názvu typu dat, názvu obalovací metody a třídy zastupující danou hierarchii. Tento způsob využívají obalovací třídy zastupující typovou hierarchii.

Přesunutí instanciace obalovacích tříd na jedno místo a jejich sjednocení zjednodušuje vzájemné propojení jednotlivých tříd. Například třídy z modulu `schema` tak nemusejí importovat třídy z modulu `schema_parsed`, i když v některých případech instance tříd definovaných v modulu `schema_parsed` vracejí. Díky tomu jsou na sobě jednotlivé moduly mnohem méně závislé a minimalizují se šance, že případné další změny způsobí cyklické závislosti, se kterými se Python obecně nedokáže sám vypořádat.

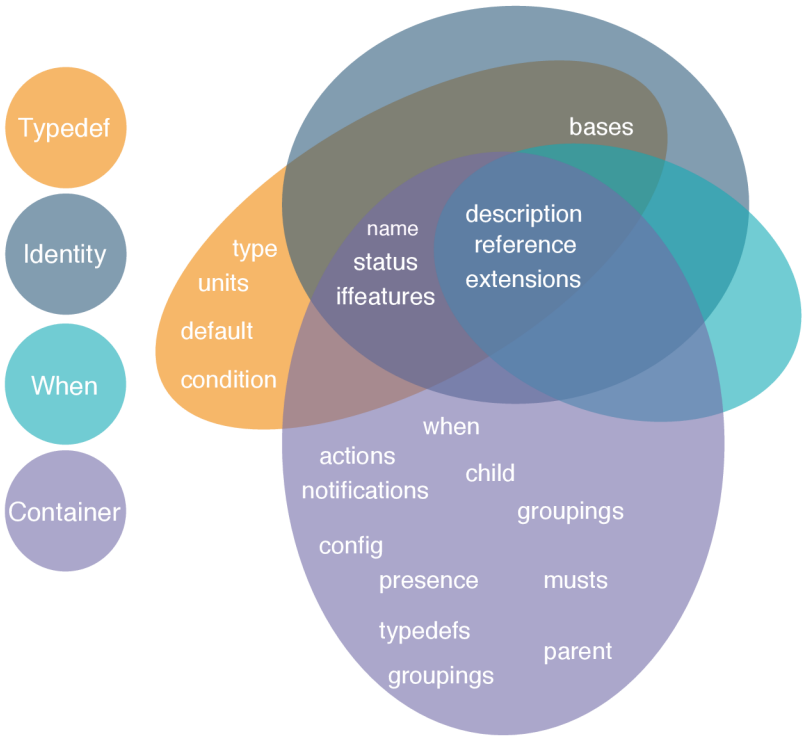
Vytváření instancí obalovacích tříd pouze na jednom místě navíc přináší vyšší kontrolu nad jejich instancemi, čehož je využito při správě paměti a umožnilo to vyřešit problém s duplikací identity instancí obalovacích tříd. Před implementací `WrapperFactory` se stávalo, že jedna instance dat z úrovně knihovny libyang 2.x mohla být na Python úrovni reprezentována několika různými obalovacími třídami. `WrapperFactory` tento problém řeší tak, že si pomocí slovníku ukládá reference na již vytvořené instance obalovacích tříd. V případě, že data jejichž obalení je požadováno už obalena byla, je vrácena existující instance obalovací třídy a nová se nevytváří.



Obrázek 5.2: Implementace třídy `WrapperFactory` pro sjednocení instanciace obalovacích tříd.

Opakující se vlastnosti obalovacích tříd

Jazyk YANG obsahuje velké množství výrazů. Pro reprezentaci většiny z nich má libyang 2.x specifickou datovou strukturu. Tyto datové struktury, stejně jako výrazy jazyka YANG, mají několik částečně konjunktních vlastností. Příklad průniku vlastností mezi jednotlivými strukturami je obsahem diagramu 5.3. V diagramu jsou pro zjednodušení zobrazeny vlastnosti pouze čtyř typů. libyang 2.x má však takto provázaných typů četné desítky a některé z nich jsou ještě výrazně složitější než vyobrazení v diagramu. Modelovat systém s takto provázanými vlastnosti pomocí třídní hierarchie obsahující pouze jednoduchou dědičnost by bylo bez duplikace kódu na úrovni některých rodičovských tříd velmi obtížné.



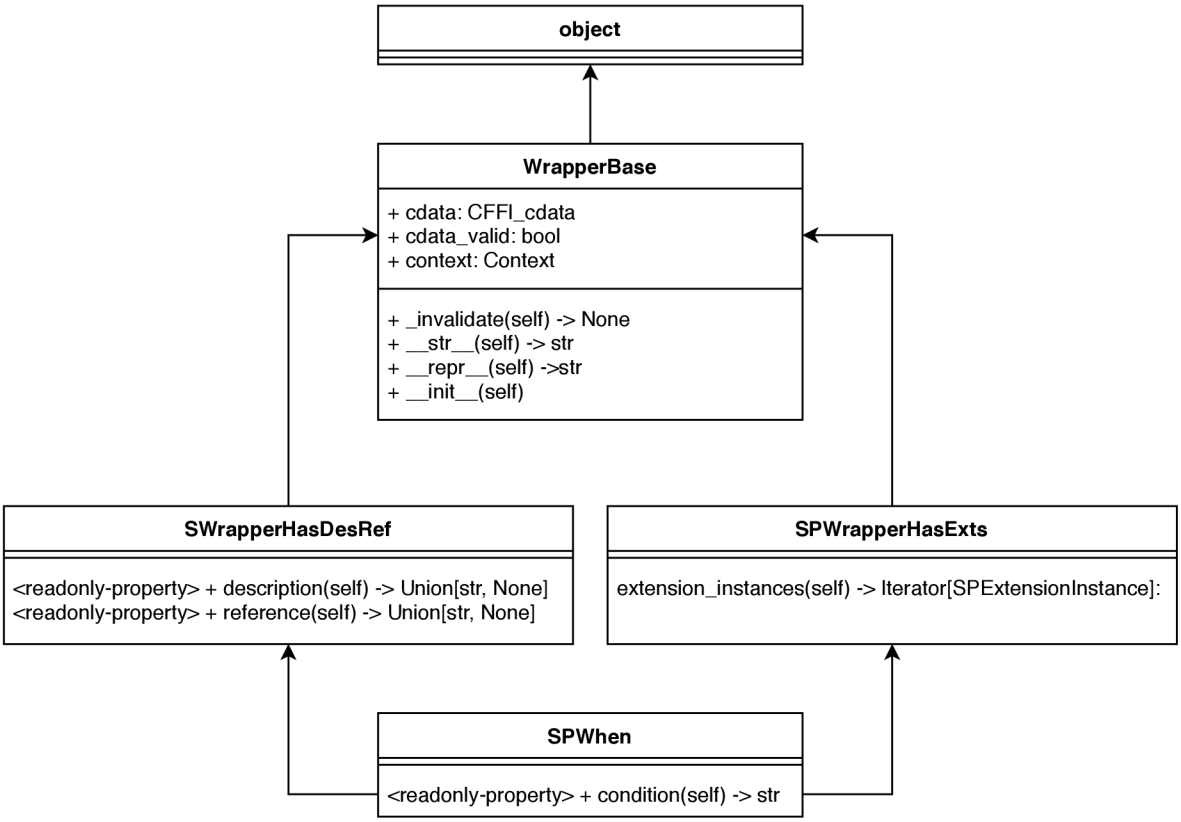
Obrázek 5.3: Diagram zobrazující průnik vlastností několika YANG výrazů.

Proto jsem konjunktní vlastnosti implementoval pomocí *mix-in*[2] tříd. To mi umožnilo obalovací třídy vytvořit bez duplikace kódu. Zároveň to ovšem znamená, že se implementace jednotlivých obalovacích tříd interně výrazně liší od podoby prezentované v kapitole 4, podstatné části jejich rozhraní však zůstávají beze změn. Ukázka implementace jedné z obalovacích tříd je na obrázku 5.4. Obrázek zahrnuje i její kompletní třídní hierarchii. Pomocí velmi podobné hierarchie jsou vytvořeny i ostatní obalovací třídy. Třídy `SWrapperHasDesRef` a `SWrapperHasExts` jsou v roli mix-in tříd.

Správa paměti

Python pro automatickou správu paměti používá systém počítání referencí v kombinaci s garbage collectorem pro uvolňování referenčních cyklů [3, sekce 1.10.]. U paměti aloko-

[2]Mix-in je v programování styl vývoje programového vybavení, při kterém jsou části funkcionality vytvořeny ve třídě, která je následně „vmíchána“ do ostatních tříd [13].



Obrázek 5.4: Diagram zobrazující kompletní hierarchii dědičnosti pro obalovací třídu `SPWhen`. Třídy `SWrapperHasDesRef` a `SWrapperHasExts` jsou v roli mix-in tříd.

vané interně v knihovně libyang 2.x však automatická správa paměti není a paměť, kterou libyang 2.x alokuje musí také uvolnit, jinak bude docházet k úniku paměti. libyang 2.x poskytuje několik funkcí, které slouží k uvolnění paměti, kterou alokuje.

Z rozhraní pro Python je třeba pomocí funkcí z API knihovny libyang 2.x uvolňovat paměť ve dvou případech, a to při finalizaci kontextu a při finalizaci datového stromu.

Třídy, které slouží k reprezentaci kontextu i datových stromů, uživateli nabízí metodu `destroy`, pomocí které mohou být explicitně uvolněny jejich systémové prostředky.

V případě kontextu bylo navíc dosaženo automatizované správy paměti pomocí odkazů ze všech instancí obalovacích tříd, které s daty z kontextu pracují. Python tak nebude instance třídy `Context` automaticky uvolňovat, dokud na něj, nebo na kteroukoliv z jeho částí, je k dispozici reference. Ve chvíli, kdy na instanci třídy `Context` neexistují dosažitelné reference, dojde k jejímu automatickému uvolnění. To zahrnuje i invokaci finalizační metody `__del__`, která v případě třídy `Context` zajistí i uvolnění paměti interně alokované knihovnou libyang 2.x.

Pro případ, že uživatel instance tříd `Context` a `DNode` (nebo z `DNode` odvozených tříd) uvolní explicitně metodou `destroy`, dojde k informování všech obalovacích tříd, které obalují data z daného kontextu nebo datového stromu, že její data nadále nejsou platná a práce s nimi způsobí vyvolání výjimky `RuntimeError`.

I přesto, že třída `Context` automatickou správu paměti umožňuje, pořád je doporučeno ji používat v kombinaci s výrazem `when` jako context manager jazyka Python 3.

V případě datových stromů se automatickou správu paměti nepodařilo zajistit hlavně kvůli širokým možnostem jejich vzájemného propojování a průběžné modifikace. Uživatel tak musí po dokončení práce s datovými stromy vytvořené stromy uvolnit invokací metody `destroy`. V opačném případě budou automaticky uvolněny až při uvolňování kontextu, do kterého patřily. Několik týdnů před termínem dokončení této práce však byla do knihovny libyang 2.x přidána funkcionalita *data diff*, která implementaci automatické správy paměti usnadní i v případě datových stromů, a v následujících měsících se ji chystám přidat.

Interakce s funkčními makry

CFFI si nedokáže automaticky poradit s funkčními makry, proto jsem pro práci s nimi vytvořit nativní funkci, která k danému funkčnímu makru poskytuje rozhraní, se kterým už si CFFI dokáže poradit a vygenerovat její ekvivalent invokovatelný z prostředí jazyka Python.

Vlastnosti implementace důležité pro opensource projekt s aktivní komunitou

Kód, který jsem během práce vytvořil, by se po dokončení a vydání plné verze knihovny libyang 2.x měl stát oficiálním balíčkem projektu a stejně jako v aktuální verzi se očekává aktivní přístup opensource komunity při údržbě a implementaci nových funkcí.

S touto myšlenkou jsem při implementaci k jednotlivým částem přidával dokumentační komentáře, anotace typů a využíval jsem jednotný styl pro zápis kódu převzatý z předchozí verze Python rozhraní pro libyang 1.x [14]. Tyto vlastnosti usnadní orientaci v kódu všem, kteří ho budou v budoucnu využívat nebo nějakým způsobem upravovat. Dokumentační komentáře navíc po vydání finální verze knihovny libyang 2.x a balíčku libyang 2.x-python poslouží při generování dokumentace.

Během implementace vznikla také rozsáhlá sada automatizovaných testů. Podrobnější popis testování je obsahem kapitoly 6.

Kapitola 6

Testování

V této kapitole je popsán způsob testování výsledného rozhraní.

Funkčnost výsledné implementace byla ověřována ve třech částech, a to během implementace pomocí jednotkových testů pro jednotlivé obalovací třídy, následně pomocí několika složitějších testů, které využívají více částí rozhraní najednou, a následně ručně při vytváření ukázek použití.

Použité technologie

Při vytváření testů jsem použil testovací framework *unittest*. Framework unittest podporuje automatizaci testů, sdílení kódu pro inicializaci a finalizaci testovacího prostředí a je součástí standardní knihovny jazyka Python 3 [5, sekce unittest — Unit testing framework].

Testování dílčích částí

Testování dílčích částí bylo zajištěno během tvorby implementace pomocí jednotkových testů. Jednotkové testy byly během implementace velmi důležité z toho důvodu, že se rozhraní knihovny libyang 2.x průběžně měnilo.

Každá obalovací třída má svůj vlastní unittest, který testuje její metody pro přístup k atributům struktur knihovny libyang 2.x, kterou obalují. Jednotlivé unit testy využívají mock objektů. Testy jednotlivých částí jsou tak nezávislé a vždy testují vlastnosti jedné konkrétní obalovací třídy v kombinaci s daty, která obaluje.

Jak bylo uvedeno v kapitole 5 jednotlivé obalovací třídy mají několik sdílených vlastností, které mohou být testovány stejně. Proto jsem sdílený testovací kód oddělil do samostatných tříd. Při testování sdílené vlastnosti pak pouze invokuji metodu, která danou vlastnost otestuje na kombinaci právě testovaných obalovaných dat a obalovací třídy.

Testování celku

Správné fungování dílčích částí však ještě není zárukou úspěchu, proto jsem vytvořil i několik složitějších testů, ve kterých je využíváno více částí rozhraní najedou. To umožňuje zjistit, zda spolu jednotlivé části rozhraní správně spolupracují, a ověřit tak, že je výsledná implementace funkční.

Pokrytí kódu

Automatizovaných testů je celkově téměř 600 a dohromady pokrývají více než 90 % kódu rozhraní. Testy budou v budoucnu využity v systému pro kontinuální integraci.

Kapitola 7

Závěr

Cílem této bakalářské práce bylo vytvořit rozhraní pro knihovnu libyang 2.x, díky kterému bude vhodným způsobem použitelná v rámci programovacího jazyka Python 3. Tento cíl byl úspěšně splněn.

V práci jsou obsaženy všeobecné informace o modelovacím jazyku YANG a knihovně libyang 2.x, která jej implementuje. Po nastudování této oblasti jsem provedl prvotní průzkum přístupů k tvorbě Python rozhraní pro knihovnu jazyka C, vybral jsem z nich ty, které se pro můj účel zdály jako adekvátní a nastudoval si o nich mnohem více podrobností. Následně jsem s využitím vybraných přístupů vytvořil několik testovacích případů pro měření výkonnosti, díky kterým jsem zjistil, jak velký dopad má použití jednotlivých přístupů na výkonnost výsledné Python 3 knihovny. S využitím závěrů získaných z měření jsem vybral přístup, který jsem následně k implementaci použil. Navrhnul jsem uživatelsky přívětivé, objektově orientované a „pythonické“ rozhraní pro knihovnu libyang 2.x. Toto rozhraní jsem následně implementoval jako knihovnu pro Python 3. Implementaci jsem otestoval pomocí velkého množství automatizovaných unit testů a několika automatizovaných end to end testů, které dohromady testují více než 90 % rozhraní knihovny. Následně jsem vytvořil sadu ukázek použití rozhraní, během čehož jsem výslednou implementaci testoval i ručně.

Implementované rozhraní umožňuje s knihovnou libyang 2.x pracovat pro Python 3 přirozeným způsobem, hojně využívá iterátorů a magických metod jazyka Python. Odstiňuje uživatele od práce s bitovými poli, přepínači a pro zpracování chyb používá systém výjimek. Díky tomu bude výsledek práce v budoucnu napomáhat rozšíření DMDNM přístupu ke správě sítí a všeobecně při integraci podpory modelovacího jazyka YANG do síťových zařízení a aplikací.

Během práce pro mě bylo obtížné pracovat s nedokončenou knihovnou a stavět na rozhraní, které se v některých ohledech průběžně měnilo. Závislost na nestabilním rozhraní mi ukázala praktickou důležitost automatizovaných testů a významným způsobem jsem si díky práci rozšířil znalosti jazyka Python a možností jeho interakce s jazykem C.

Po vydání plné verze knihovny libyang 2.x by se moje implementace měla stát oficiálním Python rozhraním pro projekt libyang 2.x. Mám v plánu se sdružením CESNET dále spolupracovat, výsledné rozhraní dále zdokonalovat a přidávat do něj nové funkce, které se v knihovně libyang 2.x budou průběžně objevovat. Zároveň bych se rád účastnil integrace nového Python rozhraní do projektu sysrepo-python, který po vydání plné verze knihovny libyang 2.x bude na tuto novou verzi přecházet. Chtěl bych také dokončit automatizaci správy paměti i pro datové stromy. Ta je nyní možná díky funkcionalitě, která do knihovny libyang 2.x přibyla v posledních týdnech před odevzdáním práce a neměl jsem tedy čas její podporu do práce plně integrovat.

Seznam zkratek

ABI Application Binary Interface

ADT Abstract Data Type, *česky: abstraktní datový typ*

API Application Programming Interface, *česky: rozhraní pro programování aplikací*

DMDNM Data Model Driven Network Management, *česky: modelově orientovaný přístup ke správě sítě*

JSON JavaScript Object Notation, *česky: javaskriptový objektový zápis*

NMS Network Management System, *česky: systém správy sítě*

XML Extensible Markup Language, *česky: rozšiřitelný značkovací jazyk*

Literatura

[1] *Ctypes — A foreign function library for Python* [online]. Version 3.8.5. Beaverton, USA: Python Software Foundation, 22-11-2019 [cit. 22-11-2019]. Dostupné z: https://docs.python.org/3/library/ctypes.

[2] *Cython's Documentation* [online]. Version 3.0 alpha 4. [cit. 5-1-2019]. Dostupné z: https://cython.org/#documentation.

[3] *Extending and Embedding the Python Interpreter* [online]. Version 3.8.5. Beaverton, USA: Python Software Foundation, 26-7-2020 [cit. 26-7-2020]. Dostupné z: https://docs.python.org/3/extending/.

[4] *The Python Language Reference* [online]. Version 3.8.5. Beaverton, USA: Python Software Foundation, 26-7-2020 [cit. 26-7-2020]. Dostupné z: https://docs.python.org/reference.

[5] *The Python Standard Library* [online]. Version 3.8.5. Beaverton, USA: Python Software Foundation, 20-7-2020 [cit. 21-7-2020]. Dostupné z: https://docs.python.org/3/library/.

[6] Benson, D. Overview - For U & Me: Yet Another Top Ten List of Popular Programming Languages. *Open Source for You*. New Dehli, India: Athena Information Solutions Pvt. Ltd. Únor 2019. ISSN 2456-4885. Dostupné z: http://search.proquest.com/docview/2184591579/.

[7] Bjorklund, M. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)* [Internet Requests for Comments]. RFC 6020. Internet Engineering Task Force (IETF), říjen 2010. Dostupné z: http://www.rfc-editor.org/rfc/rfc6020.txt.

[8] Bjorklund, M. *The YANG 1.1 Data Modeling Language* [Internet Requests for Comments]. RFC 7950. Internet Engineering Task Force (IETF), srpen 2016. Dostupné z: http://www.rfc-editor.org/rfc/rfc7950.txt.

[9] Claise, B., Clarke, J. a Lindblad, J. *Network Programmability with YANG: The Structure of Network Automation with YANG, NETCONF, RESTCONF, and gNMI*. 1. vyd. Addison-Wesley Professional, 2019. ISBN 978-0135180396.

[10] Clark, J. a DeRose, S. *XML Path Language (XPath) Version 1.0*. W3C Recommendation. W3C, listopad 1999. Dostupné z: https://www.w3.org/TR/1999/REC-xpath-19991116/.

[11] Cook, S., Bock, C., Rivett, P., Rutt, T., Seidewitz, E. et al. *Unified Modeling Language (UML) Version 2.5.1.* Standard. Object Management Group (OMG), prosinec 2017. Dostupné z: https://www.omg.org/spec/UML/2.5.1.

[12] Enns, R., Bjorklund, M., Schönwälder, J. a Bierman, A. *Network Configuration Protocol (NETCONF)* [Internet Requests for Comments]. RFC 6241. Internet Engineering Task Force (IETF), červen 2011. Dostupné z: http://www.rfc-editor.org/rfc/rfc6241.txt.

[13] Esterbrook, C. Using Mix-ins with Python. *Linux Journal.* USA: Specialized System Consultants, Inc. (SSC). Duben 2001, č. 84, s. 114, 116, 118, 120–121, [cit. 22-4-2020]. ISSN 1938-3827. Dostupné z: http://noframes.linuxjournal.com/lj-issues/issue84/4540.html.

[14] Jarry, R. *Libyang-python* [online]. GitHub, 2020. Dostupné z: https://github.com/CESNET/libyang-python.

[15] Kernighan, B. W. a Ritchie, D. M. *C Programming Language, 2nd Edition.* 2. vyd. Prentice Hall, duben 1988. ISBN 978-0131103627. Dostupné z: https://www.amazon.com/dp/0131103628?tag=xarg-20.

[16] Krejčí, R. a Vaško, M. *Libyang* [online]. GitHub, 2020. Dostupné z: https://github.com/CESNET/libyang/tree/master.

[17] Pitrou, A. *Safe object finalization.* PEP 442. Python Software Foundation, květen 2013. Dostupné z: https://www.python.org/dev/peps/pep-0442/.

[18] Rigo, A. a Fijalkowski, M. *CFFI documentation* [online]. Version 1.14. [cit. 16-3-2020]. Dostupné z: https://cffi.readthedocs.io/.

[19] Rossum, G. van a Coghlan, N. *The "with"Statement.* PEP 343. Python Software Foundation, květen 2005. Dostupné z: https://www.python.org/dev/peps/pep-0343/.

[20] Rossum, G. van, Lehtosalo, J. a Langa Łukasz. *Type Hints.* PEP 484. Python Software Foundation, září 2014. Dostupné z: https://www.python.org/dev/peps/pep-0484/.

[21] Rossum, G. van, Warsaw, B. a Coghlan, N. *Style Guide for Python Code.* PEP 8. Python Software Foundation, červenec 2001. Dostupné z: https://www.python.org/dev/peps/pep-0008/.

[22] Summerfield, M. Python Generators. *Micro Mart.* London, United Kingdom: Dennis Publishing Ltd. Červenec 2016, č. 1423, s. 62–63. ISSN 1473-0251. Dostupné z: http://search.proquest.com/docview/1812379654/.

Příloha A

Ukázky použití

Tato sekce obsahuje příklady použití jednotlivých částí implementovaného rozhraní pro jazyk Python 3. Jednotlivé příklady jsou vyobrazeny formou napodobující interaktivní Python konzoli. Jednotlivé příklady ve formě skriptů jsou součástí přiloženého paměťového média.

K ukázání některých funkcí rozhraní je využit YANG modul a jeho instanční data z ukázky 2.1 v kapitole 2 Pro přehlednost a návaznost textu modul a jeho instanční data uvádím znovu i zde. Ukázky předpokládají, že uvedený YANG modul je uložen ve složce modules a instanční data ve složce data, nacházející se v pracovním adresáři.

```
module interfaces {
    namespace "urn:example:interfaces";
    prefix "ex";
    container "interfaces" {
         list interface {
              key "name";
              leaf name {
                   type string;
                   description "Name of the interface.";
              }
              leaf mtu {
                   type uint32;
                   description "The MTU of the interface.";
              }
         }
    }
}
```

Výpis A.1: Jednoduchý YANG modul reprezentující seznam rozhraní. Rozhraní je popsáno pomocí názvu a MTU, kde název rozhraní tvoří klíč seznamu, s úpravami převzato z [8]

```
<interfaces xmlns="urn:example:interfaces">

    <interface>
          <name>eth0</name>
          <mtu>1500</mtu>
    </interface>

    <interface>
          <name>wlps0</name>
          <mtu>2304</mtu>
    </interface>

</interfaces>
```

Výpis A.2: Ukázka instančních dat modulu z ukázky A.1 ve formátu XML. Data reprezentují dvě rozhraní s názvy eth0 a wlps0 a jejich MTU.

Práce s kontextem

Vytvoření a konfigurace kontextu

```
>>> importování třídy Context z libyang2 modulu
>>> from libyang2 import Context
>>>
>>> # vytvoření kontextu s výchozím nastavením
>>> ctx = Context()
>>> # přístup k nastavením kontextu
>>> ctx.allimplemented
False
>>> ctx.searchdirs
ContextSearchdirsSet()
>>> # změna nastavení kontextu
>>> ctx.allimplemented = True
>>> ctx.allimplemented
True
>>> # přidání cesty do cest k prohledávání
>>> ctx.searchdirs.add("./modules")
>>> ctx.searchdirs
{'/home/xsedla1d/school/bc-thesis/implementation/examples/modules'}
>>> # vymazání cest k vyhledávání
>>> ctx.searchdirs.clear()
>>> # přidání více cest najednou
>>> ctx.searchdirs.update(["/etc", "/tmp"])
>>> ctx.searchdirs
{'/etc', '/tmp'}
```

Výpis A.3: Příklad možností vytvoření a konfigurace kontextu

Načítání modulů

```
>>> importování třídy Context z libyang2 modulu
>>> from libyang2 import Context
>>>
>>> # vytvoření kontextu s nastavením cest k prohledávání na "./modules"
>>> with Context(searchdirs=["./modules"]) as ctx:
      # načtení modulu interfaces z cest k prohledávání
      ctx.load_module("interfaces")
      <libyang2.schema.SModule: <'SModule': 'interfaces'>>
>>>
>>> # vytvoření kontextu s výchozím nastavením
>>> with Context() as ctx:
      # otevření souboru pro čtení
      with open("./modules/interfaces.yang") as f:
      # načtení modulu ze souboru
      ctx.parse_module(f)
      <libyang2.schema.SModule: <'SModule': 'interfaces'>>
>>>
>>> # načtení modulu z řetězce, proměnná mod obsahuje modul interfaces
>>> with Context() as ctx:
      # načtení modulu interfaces z řetězce, proměnná
      # mod obsahuje textovou podobu řetězce
      ctx.parse_module(mod)
      <libyang2.schema.SModule: <'SModule': 'interfaces'>>
```

Výpis A.4: Příklad možností načítání modulů do kontextu

Vypsání všech modulů načtených v kontextu

```
>>> importování třídy Context z libyang2 modulu
>>> from libyang2 import Context
>>> # vytvoření kontextu s nastavením cesty k prohledávání na "./modules"
>>> ctx = Context(searchdirs=["modules"])
   # načtení modulu interfaces z cest k prohledávání
>>> ctx.load_module("interfaces")
<libyang2.schema.SModule: <'SModule': 'interfaces'>>
>>> for module in ctx:
          print(module.name)
ietf-yang-metadata
yang
ietf-inet-types
ietf-yang-types
ietf-datastores
ietf-yang-library
interfaces
```

Výpis A.5: Příklad možností iterování nad kontextem. Vypsání názvu všech modulů obsažených v kontextu. Modul interfaces byl načten do kontextu explicitně, ostatní uvedené moduly jsou v libyang 2.x kontextu implicitně.

Práce se schématy

Převod modulu z formátu YANG do formátu YIN

```
>>> importování třídy Context z libyang2 modulu
>>> from libyang2 import Context
>>> # vytvoření kontextu
>>> with Context() as ctx:
      # načtení modulu intefaces ve formátu YANG
      # ze souboru interfaces.yang
      with open("modules/interfaces.yang") as f:
         mod = ctx.parse_module(f)
         mod_str = mod.print()
         # serializace modulu do formátu YIN
         mod_str = mod.print(fmt="yin")
         print(mod_str)

<?xml version="1.0" encoding="UTF-8"?>
<module name="interfaces"
       xmlns="urn:ietf:params:xml:ns:yang:yin:1"
       xmlns:ex="urn:example:interfaces">
   <namespace uri="urn:example:interfaces"/>
   <prefix value="ex"/>
   <container name="interfaces">
   <list name="interface">
       <key value="name"/>
       <leaf name="name">
       <type name="string"/>
       <description>
           <text>Name of the interface.</text>
       </description>
       </leaf>
       <leaf name="mtu">
       <type name="uint32"/>
       <description>
           <text>The MTU of the interface.</text>
       </description>
       </leaf>
   </list>
   </container>
</module>
```

Výpis A.6: Příklad možností serializace modulu, v ukázce je modul načtený z formátu YANG serializován do formátu YIN.

Prohlížení parsované verze modulu

```
>>> # importování třídy Context z libyang2 modulu
>>> from libyang2 import Context
>>> # vytvoření kontextu a nastavení cesty k prohledávání na ./modules
>>> ctx = Context(searchdirs=["./modules"])
>>> # načtení modulu interfaces
>>> interfaces_mod = ctx.load_module("interfaces")
>>>
>>> first = []
>>> second = []
>>> third = []
>>>
>>> # iterace nad první úrovní stromové struktury
>>> for node0 in interfaces_mod.parsed:
        first.append(node0)
        # iterace nad druhou úrovní stromové struktury
        for node1 in node0.childs():
            second.append(node1)
            # iterace nad třetí úrovní stromové struktury
            for node2 in node1.childs():
                third.append(node2)
>>> first
[<libyang2.schema_parsed.SPContainer: <'SPContainer': 'interfaces'>>]
>>> second
[<libyang2.schema_parsed.SPList: <'SPList': 'interface'>>]
>>> third
[<libyang2.schema_parsed.SPLeaf: <'SPLeaf': 'name'>>,
<libyang2.schema_parsed.SPLeaf: <'SPLeaf': 'mtu'>>]
```

Výpis A.7: Vytvoření seznam uzlů prvních tří nejvyšší úrovní parsované verze modulu interfaces

Prohlížení kompilované verze modulu

```
>>> # importování třídy Context z libyang2 modulu
>>> from libyang2 import Context
>>> # vytvoření kontextu a nastavení cesty k prohledávání na ./modules
>>> ctx = Context(searchdirs=["./modules"])
>>> # načtení modulu interfaces
>>> interfaces_mod = ctx.load_module("interfaces")
>>>
>>> first = []
>>> second = []
>>> third = []
>>>
>>> # iterace nad první úrovní stromové struktury
>>> for node0 in interfaces_mod.compiled:
        first.append(node0)
        # iterace nad druhou úrovní stromové struktury
        for node1 in node0.childs():
            second.append(node1)
            # iterace nad třetí úrovní stromové struktury
            for node2 in node1.childs():
                third.append(node2)

>>> first
[<libyang2.schema_compiled.SCContainer: <'SCContainer': 'interfaces'>>]
>>> second
[<libyang2.schema_compiled.SCList: <'SCList': 'interface'>>]
>>> third
[<libyang2.schema_compiled.SCLeaf: <'SCLeaf': 'name'>>,
<libyang2.schema_compiled.SCLeaf: <'SCLeaf': 'mtu'>>]
```

Výpis A.8: Vytvoření seznamu uzlů prvních tří nejvyšších úrovní kompilované verze modulu interfaces. V tomto jednoduchém případě se zdá být stromová struktura modelů stejná pro parsovanou i kompilovanou verzi. Při složitějších případech by však jejich struktura mohla být úplně jiná, objekty parsovaného modulu a kompilovaného mají navíc různé vlastnosti.

Práce s datovými stromy

Načtení datového stromu

```
>>> # importování třídy Context z libyang2 modulu
>>> from libyang2 import Context

>>> with Context(searchdirs=["modules"]) as ctx:
        # otevření souboru s daty
        with open("data/interfaces.xml", "r") as f:
            # načtení modulu interfaces z cest k prohledávání
            mod = ctx.load_module("interfaces")
            # načtení instančních dat ze souboru s nastavením na validaci
            # pouze vůči modulum načtenych dat, ne vůči celému kontextu
            ctx.parse_data(f, validate_present=True)

<'DContainer': 'interfaces'>

>>> with Context(searchdirs=["modules"]) as ctx:
        mod = ctx.load_module("interfaces")
        # načtení instančních dat s proměnné, proměnná data_str obsahuje
        # definici instančních dat pro modul interfaces
        ctx.parse_data(data_str, validate_present=True)

<'DContainer': 'interfaces'>
```

Výpis A.9: Ukázka možností načítání instančních dat.

Prohlížení datového stromu

```
>>> # importování třídy Context z libyang2 modulu
>>> from libyang2 import Context
>>>
>>> first = []
>>> second = []
>>>
>>> # vytvoření kontextu s nastavenou cestou k prohledávání
>>> with Context(searchdirs=["modules"]) as ctx:
        # otevření souboru s instančními daty
        with open("data/interfaces.xml", "r") as f:
            # načtení modulu interfaces z cest k prohledávání
            mod = ctx.load_module("interfaces")
            # načtení instančních dat
            tree = ctx.parse_data(f, validate_present=True)

            # iterování nad první úrovní stromové struktury
            for node1 in tree:
                first.append(node1)
                # iterování nad druhou úrovní stromové struktury
                for node2 in node1:
                    second.append(node2)
            print(first)

[<libyang2.data.DList: <'DList': 'interface'>>,
 <libyang2.data.DList: <'DList': 'interface'>>]

            print(second)

[<libyang2.data.DLeaf: <'DLeaf': 'name'>>,
<libyang2.data.DLeaf: <'DLeaf': 'mtu'>>,
<libyang2.data.DLeaf: <'DLeaf': 'name'>>,
<libyang2.data.DLeaf: <'DLeaf': 'mtu'>>]
```

Výpis A.10: Ukázka vytvoření seznamu uzlů prvních dvou nejvyšších úrovní datového stromu modulu interfaces.

Serializace datového stromu

```
>>> # importování třídy Context z libyang2 modulu
>>> from libyang2 import Context
>>>
>>> # vytvoření kontextu s nastavenou cestou k prohledávání
>>> with Context(searchdirs=["modules"]) as ctx:
        # otevření souboru pro čtení
        with open("data/interfaces.xml", "r") as f:
            # načtení modulu ze souboru
            mod = ctx.load_module("interfaces")
            f = open("data/interfaces.xml", "r")
            tree = ctx.parse_data(f, validate_present=True)
            tree_str = tree.print(format=True)
            print(tree_str)

<interfaces xmlns="urn:example:interfaces">
    <interface>
        <name>eth0</name>
        <mtu>1500</mtu>
    </interface>
    <interface>
        <name>wlps0</name>
        <mtu>2304</mtu>
    </interface>
</interfaces>
```

Výpis A.11: Ukázka serializace datového stromu.

Rozšiřování a validace datového stromu

```
>>> # importování třídy Context z libyang2 modulu
>>> from libyang2 import Context
>>>
>>> # vytvoření kontextu a nastavení cesty k prohledávání
>>> with Context(searchdirs=["modules"]) as ctx:
        # otevření souboru s daty ke čtení
        with open("data/interfaces.xml", "r") as f:
            # načtení modulu interfaces z cest k prohledávání
            mod = ctx.load_module("interfaces")
            # načtení datového stromu
            tree = ctx.parse_data(f, validate_present=True)

            # přidání rozhraní eth1 do seznamu rozhraní
            new_interface = tree.newchild_list("interface", "[name='eth1']")
            # nastavení MTU přidaného rozhraní
            new_interface.newchild_term("mtu", "3200")

            # validace datového stromu, argument presente způsobuje
            # validaci pouze pritomných dat
            tree.validate(present=True)
            # serializace datového stromu
            print(tree.print(format=True))

<interfaces xmlns="urn:example:interfaces">
    <interface>
        <name>eth0</name>
        <mtu>1500</mtu>
    </interface>
    <interface>
        <name>wlps0</name>
        <mtu>2304</mtu>
    </interface>
    <interface>
        <name>eth1</name>
        <mtu>3200</mtu>
    </interface>
</interfaces>
```

Výpis A.12: Ukázka rozšiřování a validace datového stromu.