VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

TVORBA SIMULAČNÍCH MODELŮ JAZYKA P4

CREATION OF SIMULATION MODELS FOR P4 LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MIROSLAV BULIČKA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MARTÍNEK, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Bulička Miroslav**
Program: Informační technologie
Název: **Tvorba simulačních modelů jazyka P4**
Development of Simulation Models of P4 Language
Kategorie: Počítačové sítě
Zadání:

1. Seznamte se s jazykem P4 a jeho využitím v oblasti síťových aplikací.
2. Seznamte se se simulačním modelem jazyka P4 (BMv2) a zaměřte se na problematiku vytváření vlastních simulačních modelů pro specifické cílové platformy.
3. Navrhněte a implementujte nový simulační model jazyka P4 pro karty rodiny COMBO vyvíjené v rámci sdružení CESNET. Při návrhu zohledněte např. možnost připojení externích modulů.
4. Funkčnost vytvořeného modelu ověřte na vhodných síťových aplikacích.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího pokračování projektu.

Literatura:

- Dle pokynů vedoucího práce.

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

Vedoucí práce: **Martínek Tomáš, Ing., Ph.D.**
Konzultant: Benáček Pavel, Ing., CESNET
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 9. ledna 2020

Abstrakt

Sdružení CESNET vyvíjí nástroj umožňující spouštět P4 programy na programovatelném hradlovém poli. Tento nástroj se využívá při návrhu digitálního obvodu, který prochází procesem verifikace. Pro verifikaci je prozatím využíván simulační model, který plně neodpovídá požadavkům. Bakalářská práce se zabývá seznámením s jazykem P4 a tvorbou jeho simulačních modelů. Práce má za cíl zejména vytvoření simulačního modelu, který přesněji odpovídá požadavkům sdružení CESNET tak, aby mohl být nahrazen dosavadní simulační model využívaný ve verifikačním prostředí. Tvorba modelu probíhala pomocí projektu BMv2, který umožňuje psaní simulačních modelů pro jazyk P4. Jako výchozí simulační model byl zvolen Simple switch, který je ve verifikačním prostředí prozatím využíván a který byl upravován podle požadavků. Výsledkem práce je simulační model odpovídající požadavkům sdružení CESNET.

Abstract

CESNET association is developing tool that allows running P4 programs on field programmable gate array. This tool is used in design of digital circuit, which goes through verification process. The verification uses behavioral model, which does not fully meet requirements. This thesis deals with introduction to P4 language, creation of behavioral models and developing of behavioral model, which meets the requirements of CESNET association. Current behavioral model, used in verification process, will be replaced by developed model. Project Behavioral Model version 2 was used for developing of behavioral model. Behavioral model simple switch was used as default model. This model is edited based on requirements of CESNET association. Result of this thesis is behavioral model which meets the requirements.

Klíčová slova

jazyk P4, simulační model, Behaviorální model verze 2, software-defined networking

Keywords

P4 language, behavioral model, Behavioral model version 2, software-defined networking

Citace

BULIČKA, Miroslav. *Tvorba simulačních modelů jazyka P4*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Martínek, Ph.D.

Tvorba simulačních modelů jazyka P4

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Martínka, Ph.D. Další informace mi poskytl Ing. Pavel Benáček, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Miroslav Bulička
27. května 2020

Poděkování

Chtěl bych poděkovat především vedoucímu práce Ing. Tomáši Martínkovi, Ph.D. za konzultace týkající se této práce. Dále bych chtěl poděkovat Ing. Pavlu Benáčkovi, Ph.D. za poskytnutí dalších informací. Nakonec bych rád poděkoval celému sdružení CESNET za poskytnutí externího zadání bakalářské práce.

Obsah

Kapitola 1

Úvod

S dnešní dobou souvisí rychlý vývoj ve všech oblastech techniky. Jinak na tom není ani odvětví počítačových sítí, kde neustále rostou požadavky například na rychlost přenosu a jednodušší správu sítí. Řešení těchto věcí přináší software-defined networking, zkráceně SDN (viz kapitola 2), což je architektura umožňující vytvořit flexibilnější, dynamičtější a centralizovanější síť.

Součástí SDN je i jazyk P4 (viz podkapitola 2.1), který umožňuje definovat chování síťové vrstvy (viz kapitola 2). Toto chování si definuje sám administrátor. Jazyk P4 je platformě nezávislý a tak umožňuje spustit P4 program na různých platformách. Sdružení CESNET vytváří nástroj, který umožňuje spouštět P4 programy na programovatelném hradlovém poli (Field Programable Gate Array - FPGA). Tento nástroj se používá při návrhu digitálního obvodu, který prochází i procesem verifikace.

Verifikace probíhá tak, že je porovnán výstup výše zmíněného digitálního obvodu s výstupem simulačního modelu, který simuluje zpracování paketů stejným P4 programem jako digitální obvod. Simulační model využívaný ve verifikaci však v současné době neodpovídá architektuře digitálního obvodu. Z toho důvodu je hlavním úkolem této práce navrhnout a implementovat simulační model, pomocí kterého se bude ověřovat vygenerovaný popis pro FPGA a který odpovídá architektuře digitálního obvodu.

Práce je členěna na kapitoly. Kapitola 2 popisuje jazyk P4, cílová zařízení pro P4, projekt Behavioral Model v2 s již existujícími simulačními modely a vytváření vlastních simulačních modelů pomocí tohoto projektu. Kapitola 3 popisuje návrh na úpravy referenčního modelu tak, aby splňoval požadavky zadání a samotnou implementaci simulačního modelu. Kapitola 4 popisuje průběh testování a výsledky dosažené v této práci.

Kapitola 2

Teoretická část

Za účelem korektního pochopení implementace simulačního modelu jsou v teoretické části popsány technologie a slovní pojmy, které se v této práci vyskytují. Kapitola začíná stručným úvodem do jazyka P4 (viz podkapitola 2.1), jehož programy se spouštějí v simulačním modelu. Následuje popis cílových P4 zařízení (viz podkapitola 2.2). Kapitola končí popisem projektu Behavioral Model verze 2 (BMv2) (viz podkapitola 2.3), pomocí kterého lze vytvářet nové simulační modely.

V této kapitole se vyskytují pojmy týkající se architektury počítačových sítí, která se skládá ze síťové vrstvy (data plane), kontrolní vrstvy (control plane) a dohledové vrstvy (management plane). Dalším z vysvětlených pojmů je software-defned networking, jehož součástí je i jazyk P4, který je pro tuto práci důležitý. V následujících odstavcích jsou tyto pojmy krátce vysvětleny pro korektní pochopení dalších částí bakalářské práce.

Síťová vrstva (data plane) je část architektury počítačových sítí, která na základě informací ze směrovací tabulky přeposílá pakety na další skok po cestě do cílové destinace. [6]

Kontrolní vrstva (control plane) je část architektury počítačových sítí, která spravuje informace v směrovacích tabulkách. Zajišťuje jejich konfiguraci, řízení a výměnu informací mezi těmito tabulkami. [6]

Software-defined networking (SDN) [6] je architektura sítě, která ji dělá více flexibilní, dynamickou a centralizovanou. Tato architektura od sebe odděluje síťovou vrstvu a kontrolní vrstvu. Hlavním prvkem sítě je SDN-kontrolér, který poskytuje centralizovaný pohled na celou síť a umožňuje administrátorům konfigurovat, jak mají jednotlivé směrovače a přepínače směrovat provoz na síti, bez potřeby konfigurovat každé zařízení zvlášť. Další výhodou je bezpečnost, jelikož kontrolér poskytuje administrátorovi sledování provozu v síti a umožňuje mu nasazovat bezpečnostní politiky. Například pokud kontrolér usoudí, že je provoz na síti podezřelý, může pakety přesměrovat, či zahodit. Jedním z prvních standardů SDN je OpenFlow [12], což je standard definující komunikační protokol v SDN prostředí, který umožňuje přímou komunikaci SDN kontroléru se síťovou vrstvou síťových zařízení.

2.1 Jazyk P4

Programming Protocol-independent Packet Processors neboli P4 [10, 3, 4, 2] je protokolově nezávislý programovací jazyk, který byl původně navržen pro programování přepínačů, avšak jeho rozsah byl rozšířen na velké množství síťových prvků, jako jsou například směrovače a síťové karty, které budou dále v bakalářské práci nazývány jednotným pojmem cílové P4 zařízení. Tento jazyk vyjadřuje, jakým způsobem jsou zpracovávány pakety v síťové vrstvě cílových P4 zařízení. Částečně také definuje rozhraní pomocí kterého komunikuje síťová vrstva a kontrolní vrstva. Konkrétní funkcionalitu kontrolní vrstvy však P4 program specifikovat nemůže.

Chod P4 zařízení je rozdělen do dvou režimů a to inicializace a běžný chod. Při inicializaci se do zařízení nahrává přeložený P4 program specifikující funkcionalitu zařízení. P4 program je přeložen pomocí P4 překladače, který byl dodán spolu se zařízením, do reprezentace nejvhodnější pro dané zařízení. To umožňuje kompilovat P4 program pro běh na libovolném síťovém zařízení, bez ohledu na jeho vnitřní architekturu. Lze tedy kompilovat například na CPU, GPU, síťovém procesoru nebo FPGA. Po inicializaci zařízení přechází do režimu běžného chodu, ve kterém zařízení zpracovává pakety dle P4 programu nahraného v režimu inicializace.

Cílové P4 zařízení se liší od tradičních síťových zařízení v několika podstatných věcech. Funkcionalita síťové vrstvy není fixní, ale je definována P4 programem. Při inicializaci je síťová vrstva konfigurována tak, aby implementovala funkcionalitu popsanou v P4 programu. Tato vrstva nemá žádnou vestavěnou znalost existujících sítových protokolů.

Kontrolní vrstva zařízení komunikuje se síťovou vrstvou pomocí API. Toto API umožňuje kontrolní vrstvě spravovat *match-action* tabulky a externí objekty. *Match-action* tabulky i externí objekty se nachází v síťové vrstvě zařízení, kde však nejsou fixovány, jelikož jsou definovány P4 programem.

Výhody jazyka P4

Oproti tradičnímu zpracování paketů, kterým je myšleno spravování založené na psaní mikrokódu nad vlastním hardwarem, poskytuje jazyk P4 tyto výhody [10]:

- **Flexibilita** - P4 umožňuje vyjádřit mnoho politik přesměrování paketů jako programy.
- **Expresivita** - P4 umožňuje vyjádřit hardwarově nezávislé algoritmy zpracování paketů využívající výhradně obecné operace a tabulkové vyhledávání. Tyto programy jsou přenosné na cílové P4 zařízení implementující stejnou architekturu.
- **Mapování a správa zdrojů** - P4 programy popisují prostředky pro ukládání dat abstraktně. Překladač mapuje tyto uživatelem definované pole na dostupné hardwarové zdroje a spravuje nízkoúrovňové detaily jako je alokace a plánování.
- **Softwarové inženýrství** - P4 programy poskytují výhody, kterými jsou: kontrola typů, schování informací a odmítnutí softwaru v případě neúspěchu při mapování.
- **Knihovny komponent** - Knihovny komponent poskytované výrobcem mohou být užity k zabalení hardwarově specifických funkcí do přenosných vysokoúrovňových P4 konstrukcí.

- **Oddělení vývoje hardwaru a softwaru** - Výrobci cílových P4 zařízení mohou použít abstraktní architektury k odděleni vývoje nízkoúrovňových architektonických detailů od vysokoúrovňového zpracování.
- **Debuggování** - Výrobci mohou poskytnout softwarové modely architektury k pomoci ve vývoji a debuggování P4 programů.

Základní konstrukční prvky jazyka P4

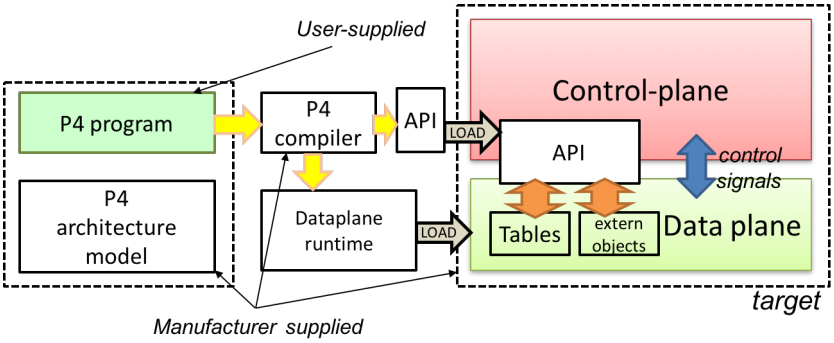
Jazyk P4 poskytuje tyto základní prvky [10]:

1. **Typy hlaviček** - popisující u každé hlavičky obsažené v paketu její formát, čímž jsou myšlena pole hlavičky a jejich velikost.
2. **Bloky pro extrakci hlaviček** - tyto kontrolní bloky popisují povolené sekvence hlaviček, jak tyto sekvence poznat a které hlavičky a pole extrahovat z paketu.
3. **Match-action tabulky** - spojují uživatelem definované klíče s akcí, která se má provést s paketem. Hodnota klíče a akce tvoří pravidlo v tabulce.
4. **Akce** - je část kódu popisující jak mají být upravena metadata a hlavičky paketu. Akce může mít parametry, jejichž hodnoty jsou v kódu akce využity.
5. **Kontrolní bloky** - vytvářejí vyhledávací klíč, pomocí kterého poté vyhledávají v tabulce pravidlo a následně provedou akci specifikovanou v nalezeném pravidle.
6. **Bloky pro složení paketu** - kontrolní bloky opětovně sestavující paket, ze kterého byly extrahované hlavičky v bloku pro extrakci hlaviček.
7. **Externí objekty** - jsou objekty specifické pro danou architekturu, se kterými může P4 program pracovat skrz definované API. Chování objektu je pevně dané a nemůže být naprogramované v P4 kódu.
8. **Uživatelem definovaná metadata** - jsou struktury definované uživatelem, které jsou součástí každého paketu.
9. **Vnitřní metadata** - jsou struktury poskytované modelem architektury. Tyto struktury jsou součástí každého paketu.

2.2 Cílové zařízení

Cílové P4 zařízení je libovolné síťové zařízení, na kterém jsou spouštěny P4 programy definující jeho funkcionalitu. Aby bylo možné toto zařízení využít v provozu, musí k němu výrobce zařízení a uživatel dodat další prvky. Obrázek 2.1 popisuje běžný pracovní postup pro spuštění cílového zařízení.

1. Výrobce dodá cílové zařízení, model architektury a překladač pro daný model.
2. Uživatel naprogramuje P4 program, popisující chování síťové vrstvy. Tento program musí respektovat rozhraní výrobcem dodaného modelu architektury.



Obrázek 2.1: Cílové zařízení a komponenty k němu potřebné. Převzato z [10]

3. P4 program je kompilován v překladači, který má dva výstupy. Prvním výstupem je konfigurace síťové vrstvy. Druhým výstupem je P4info, což jsou metadata specifikující *match-action* tabulky a externí objekty instanciované v P4 programu.

4. Cílové zařízení je možné spustit a nakonfigurovat pomocí výstupů P4 překladače.

Výsledkem tohoto procesu je zařízení implementující směrovací logiku popsanou v P4 programu.

Model architektury

Cílové zařízení obsahuje fixní části a části programovatelné pomocí P4. Tyto programovatelné části se nazývají kontrolní bloky a mají předem definované rozhraní. Toto rozhraní definuje P4 program zvaný model architektury [10]. Kromě definice rozhraní programovatelných bloků definuje model architektury také rozhraní externích objektů a metadata, se kterými bude zařízení pracovat.

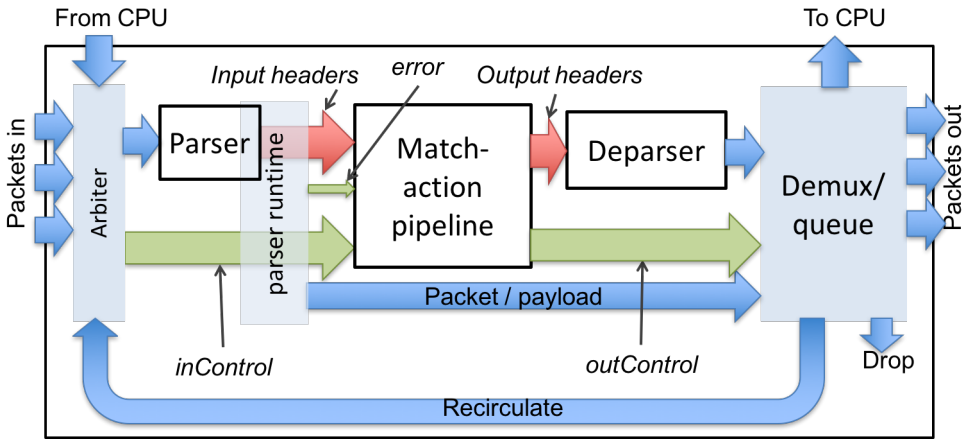
Cílové P4 zařízení interaguje s P4 programem přes sadu kontrolních registrů nebo signálů. Vstupní řízení poskytuje P4 programu informace (například vstupní port, na kterém byl přijat paket). Výstupní řízení může ovlivnit chování cílového P4 zařízení (například změnou výstupního portu pro paket). Registry a signály jsou reprezentovány v P4 jako vnitřní metadata, která přísluší každému paketu. Detaily interpretace vnitřních metadat jsou specifické pro každý model architektury.

P4 program může využívat služby implementované externími objekty, které jsou poskytované architekturou. Jejich implementace není v P4 programu specifikována, avšak jejich rozhraní ano. Rozhraní pro externí objekt popisuje každou operaci, kterou poskytuje a to i s jejími parametry a návratovými typy. Obecně P4 programy nepředpokládají přenositelnost na jiné architektury, avšak měly by být přenositelné na všechna cílová P4 zařízení implementující stejnou architekturu.

Příklad modelu architektury

Na obrázku 2.2 je popsána ukázková architektura [10]. Světle modré bloky *Arbiter*, *parser runtime* a *Demux/queue* mají pevně danou funkcionalitu. Bílé bloky *Parser*, *Match-action pipeline* a *Deparser* jsou bloky programovatelné pomocí P4. Červené šipky ukazují tok uživatelem definovaných dat, kterými jsou hlavičky paketu a případná uživatelem definovaná

metadata. Zelené šipky ukazují výměnu informací mezi bloky s pevnou funkcionalitou a programovatelnými bloky v modelu architektury. Těmito informacemi jsou myšlena vnitřní metadata.



Obrázek 2.2: Příklad modelu architektury. Převzato z [10]

Arbiter neboli rozhodčí blok přijímá pakety z jednoho z fyzických portů, z portu připojeného na CPU, nebo z recirkulačního portu. V případě, kdy je dostupných více paketů k přijetí, probíhá výběr konkrétního paketu, který bude zpracován. Z rozhodčího bloku putuje paket do bloku *Parser*, kde se extrahují hlavičky podle funkcionality naprogramované v P4 programu a následně se tyto hlavičky pošlou bloku *Match-action pipeline*. Po přijetí rozhodčím blokem se také nastaví vstupní port ve vnitřních metadatech *inCtrl*. Tato vnitřní metadata jsou stejně jako hlavičky zaslána bloku *Match-action pipeline*. *Parser runtime* blok pracuje ve shodě s blokem *Parser*. Poskytuje příznak *parse error* bloku *Match-action pipeline* a obsah paketu neobsahující extrahované hlavičky poskytuje *Demux* bloku. V bloku *Match-action pipeline* probíhá zpracování hlaviček paketu. Zpracováním je myšleno vyhledání akce, která se na paket aplikuje, v každé z tabulek z tohoto kontrolního bloku. Následně nastaví výstupní port ve vnitřních metadatech *outCtrl* a ta zašle *demux* bloku. *Deparser* složí paket a zašle ho *demux* bloku. *Demux* blok odesílá paket na daný výstupní port specifikovaný v *outCtrl*, který byl nastaven v *Match-action pipeline* bloku.

Deklarace modelu architektury

V modelu je možné krom deklarace programovatelných bloků deklarovat vlastní datové typy, konstantní proměnné a struktury pomocí stejných klíčových slov jako v jazyce C. Dále je nutná deklarace vnitřních metadat, která jsou deklarována pomocí struktury. Následující útržky kódu jsou deklarací jednotlivých programovatelných bloků z obrázku 2.2. Prvním z nich je blok pro extrakci hlaviček zvaný *Parser*.

```
parser Parser<H>(packet_in b, out H parsedHeaders);
```

V tomto útržku kódu `<H>` určuje typ hlaviček a je definován uživatelem při implementaci funkcionality tohoto modelu architektury. Blok *Parser* čte svůj vstup z argumentu *b* typu `packet_in`, což je předdefinovaný P4 externí objekt, reprezentující vstupní paket. *Parser* extrahuje hlavičky, které jsou následně zapsány do argumentu *parsedHeaders*. Klíčové slovo `out` určuje, že argument je výstupní.

Následuje deklarace kontrolního bloku *Pipe*.

```
control Pipe<H>(inout H headers,
                in error parseError,
                in InControl inCtrl,
                out OutControl outCtrl);
```

Klíčové slovo `inout` u argumentu *headers* značí, že je jak vstupem, tak výstupem tohoto bloku. Do kontrolního bloku *Pipe* jsou předány hlavičky, které extrahoval blok *Parser*. Hlavičky jsou v tomto bloku následně upraveny podle implementované funkcionality. Dalším vstupem je *parseError* značící chybu, která mohla nastat v kontrolním bloku *Parser*. Posledním vstupem je *inCtrl*, což je informace od architektury doprovázející vstupní paket. Touto informací může být vstupní port. Výstupní argument *outCtrl* je analogicky k *inCtrl* informace doprovázející výstupní paket a může jí být výstupní port.

Dále paket putuje do bloku *Deparser*.

```
control Deparser<H>(inout H outputHeaders,
                    packet_out b);
```

Vstupem tohoto bloku jsou hlavičky upravené v bloku *Pipe*. Tyto hlavičky jsou následně vloženy zpět do paketu, ze kterého byly extrahovány. Tento paket s upravenými hlavičkami je výstupem bloku *Deparser*.

Dalším krokem je deklarace balíčku nejvyšší úrovně.

```
package VSS<H>(Parser<H> p,
               Pipe<H> map,
               Deparser<H> d);
```

V P4 programu implementujícím funkcionalitu zde popsaného modelu architektury musí uživatel instanciovat tento balíček. Stejně tak musí být instanciovány bloky *Parser*, *Pipe* a *Deparser*, které jsou popsány výše a také musí být implementována jejich funkcionalita. V případě tohoto balíčku je typ `<H>` množina hlaviček. Příklad P4 programu implementujícího tuto architekturu se nachází v příloze A.

Externí třídy

Externí třídy jsou třídy, se kterými může pracovat P4 program skrz API, avšak jejich vnitřní chování je pevně definované a není možné jej naprogramovat v jazyce P4. Deklarace jejich rozhraní se nachází v modelu architektury, kde se označuje klíčovým slovem `extern`. Příkladem externí třídy je třída zajišťující výpočet kontrolního součtu. Příkladem deklarace konkrétní třídy [10] v modelu architektury je následující kód.

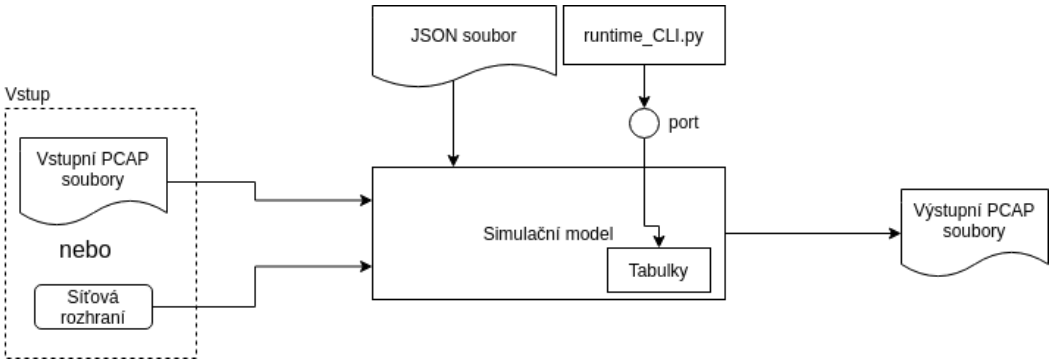
```
extern Checksum16 {
    Checksum16();
    void clear();
    void update<T>(in T data);
    void remove<T>(in T data);
    bit<16> get();
}
```

Tato deklarace popisuje metody poskytované externí třídou. Návratovou hodnotou metody `get()` je šestnáctibitová hodnota reprezentující vypočítaný kontrolní součet. Vstupním argumentem metody `void update<T>(in T data)` může být libovolné pole, seznam polí,

hlavička a podobně. Datový typ přijímaný touto metodou závisí na její konkrétní implementaci a z deklarace rozhraní jej nelze poznat.

2.3 Behaviorální model verze 2

Behaviorální model verze 2 (BMv2) [9] je projekt psaný v jazyce C++11, který slouží k tvorbě simulačních modelů simulujících cílové P4 zařízení. Kromě tříd umožňujících programování simulačních modelů obsahuje několik již vytvořených modelů. Modely slouží především jako nástroj umožňující jednodušší vývoj a testování P4 programu pro síťovou a kontrolní vrstvu fyzického P4 zařízení, které daný simulační model simuluje.



Obrázek 2.3: Vstupy a výstupy simulačního modelu

Obrázek 2.3 znázorňuje vstupy a výstupy simulačního modelu. Prvním vstupem jsou buď PCAP soubory, ze kterých simulační model získává pakety ke zpracování, nebo reálná síťová rozhraní, která jsou odposlouchávána za účelem získání vstupních paketů. Druhým vstupem modelu je JSON soubor generovaný z P4 programu pomocí P4 překladače. Tento soubor obsahuje popis prvků z P4 programu v takovém formátu, který dokáže simulační model interpretovat tak, aby implementoval zpracování paketů popsáné v P4 programu. Tento formát je nezávislý na architektuře. Posledním vstupem jsou pravidla tabulek, která jsou do simulačního modelu vkládaná pomocí programu `runtime_CLI`. Výstupem modelu jsou PCAP soubory obsahující zpracované pakety.

Spuštění simulačního modelu probíhá přes příkazový řádek. Každý simulační model obsahuje obecné argumenty upravující jeho funkcionalitu. Tyto obecné argumenty jsou společné pro všechny modely. Mimo tyto obecné argumenty má možnost poskytovat i argumenty specifické pro daný model. Specifické argumenty je nutné psát za obecné argumenty a musí od nich být odděleny oddělovačem "--". Následující příkaz je příklad spuštění s důležitými základními argumenty a jedním specifickým argumentem. Pro kompletní výčet argumentů daného simulačního modelu lze použít argument `--help`

```
./<simulacni_model> -i 0@<iface0> --use-files <time> <prog>.json -- --s-opt
```

- `-i 0@<iface0>` - Připojí k simulačnímu modelu jedno síťové rozhraní. Číslo 0 v hodnotě argumentu určuje číslo portu pro toto rozhraní, za kterým následuje název rozhraní oddělený zavináčem. Tento argument je v příkazu obsažen pro každé rozhraní, se kterým bude model pracovat. Každé rozhraní musí mít jiné číslo portu.

- `--use-files <time>` - Značí že model použije PCAP soubory s názvy `<iface0>_in.pcap`, ze kterých bude číst vstupní pakety a `<iface0>_out.pcap` do kterých zapíše zpracované pakety. Hodnota `<time>` udává čas v sekundách, po který simulační model čeká, než začne zpracovávat pakety ze vstupních souborů. Tato časová prodleva slouží k naplnění tabulek simulačního modelu. Pokud tento příznak není obsažen v příkazu spuštění modelu, jsou odposlouchávána reálná síťová rozhraní specifikována argumenty `-i`.
- `<prog>.json` - Připojuje k simulačnímu modelu JSON soubor udávající funkcionalitu simulačního modelu.
- `--s-opt` - Příklad argumentu specifického pro daný model. Jeho název a funkcionalitu určuje tvůrce simulačního modelu.

Plnění tabulek

Obrázek 2.3 také ukazuje připojení programu `runtime_CLI` [9] k simulačnímu modelu. Tento program slouží jako příkazový řádek, pomocí kterého lze naplnit tabulky simulačního modelu pravidly. Program se připojí k Thrift RPC (Remote Procedure Call, neboli vzdálené volání procedur) serveru, přes který se napojí na simulační model. Není-li při spuštění pomocí argumentu `--thrift port <cislo_portu>` specifikovné číslo portu, na kterém Thrift RPC server běží, je zvolena implicitní hodnota 9090. Pro úspěšné předávání zpráv přes Thrift RPC server musí být hodnota portu v simulačním modelu a programu `runtime_CLI` totožná. Jedna instance `runtime_CLI` může být připojena pouze k jednomu simulačnímu modelu. Příkladem spuštění jsou příkazy:

1. `./runtime_CLI.py --thrift-port 9091`
2. `./runtime_CLI.py --thrift-port 9091 < ./prikazy.txt`

Využitím první možnosti lze v terminálu zadávat příkazy sloužící k plnění tabulek simulačního modelu. Druhá možnost provede příkazy uvedené v souboru `./prikazy.txt`.

Program `runtime_CLI` poskytuje širokou sadu příkazů. Jejich následný výčet není kompletní, ale jedná se o nejdůležitější příkazy. Příklad konkrétního využití příkazu je psán pro tabulku *ipv4_lpm* nacházející se v kontrolním bloku *MyIngress* s akcemi *drop* pro zahození paketu a *ipv4_forward* pro přeposlání paketu.

- **table_set_default <jméno tabulky> <jméno akce> <parametry akce>** - Nastaví implicitní akci pro danou tabulku. Nebude-li pro paket v tabulce nalezeno žádné pravidlo, provede se tato akce. Příklad využití na konkretní tabulce:

  ```
  table_set_default MyIngress.ipv4_lpm MyIngress.drop
  ```

  Tento příkaz nastaví v tabulce *ipv4_lpm* implicitní akci *drop*.

- **table_add <jméno tabulky> <jméno akce> <hodnota, na kterou je akce využita> => <parametry akce> [priorita]** - Přidá konkrétní pravidlo do dané tabulky. Parametry akce jsou specifické pro každou akci. Existuje-li pro jednu hodnotu více pravidel, rozhoduje priorita pravidla. Nemají-li pravidla pro tuto hodnotu prioritu, je zvoleno pravidlo, které bylo přidáno jako první. Příklad využití v konkrétní tabulce:

```
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward
                                0x7f000000/8 => 0x424242424242 0
```

Příkaz nastaví v tabulce *ipv4_lpm* pravidlo pro pakety s IP adresami 127.0.0.0 až 127.0.0.255. Na pakety v daném rozsahu IP adres bude aplikována akce *ipv4_forward* s parametry *0x424242424242* a *0*.

- **table_delete <jméno tabulky> <pořadí přidání do tabulky>** - Odstraní pravidlo ze zadané tabulky. Pravidlo se vybírá podle pořadí přidání do tabulky s tím, že indexování začíná od nuly. Příklad využití:

```
table_delete MyIngress.ipv4_lpm 0
```

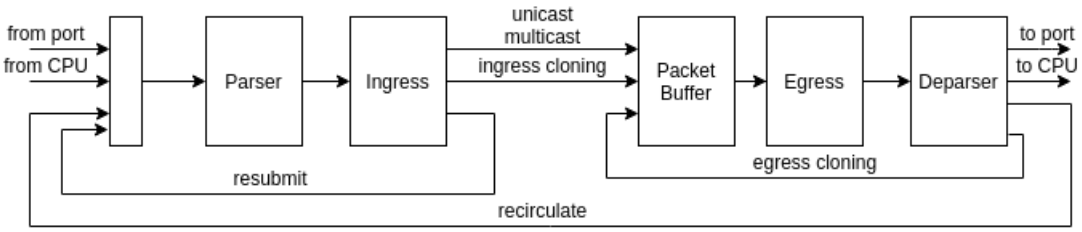
Tento příkaz odstraní první pravidlo z tabulky *ipv4_lpm*.

Existující modely

Projekt BMv2 obsahuje několik již vytvořených simulačních modelů, nad kterými lze spouštět P4 programy v případě, že implementují model architektury těchto simulačních modelů. Následuje popis dvou existujících simulačních modelů a to *Simple switch* a *Portable switch architecture*.

Simple switch

Simulační model *Simple switch* [1] je jedním z nejjednodušších existujících modelů. Lze ho využít pro verifikaci P4 kódu implementujícího softwarový přepínač. Tento simulační model pracuje s modelem architektury `v1model.p4`. Tato architektura definuje funkční bloky *Parser*, *Ingress*, *Egress* a *Deparser*. Dále poskytuje velkou škálu externích tříd a metod implementujících například čítače a výpočet kontrolního součtu. Dále obsahuje kontrolní bloky *Compute checksum* a *Verify checksum*. Tyto bloky nejsou plně programovatelné pomocí P4, jelikož umožňují pouze volání externích funkcí pro výpočet a ověření kontrolního součtu.



Obrázek 2.4: Průchod paketu modelem Simple switch

Obrázek 2.4 zobrazuje možné cesty paketu v simulačním modelu. Paket může být přijat z portu rozhraní, CPU portu, recirkulačního portu, nebo z portu opětovného zaslání. Po extrakci hlaviček v bloku *Parser* je zpracován v bloku *Ingress*, který provádí vstupní zpracování paketu. V tomto bloku lze provést vstupní klonování, multicast a zaslat paket k opětovnému zpracování. Ze vstupního zpracování je paket předán do fronty, ze které je vybrán na výstupní zpracování v bloku *Egress*. Ve výstupním zpracování je možno paket

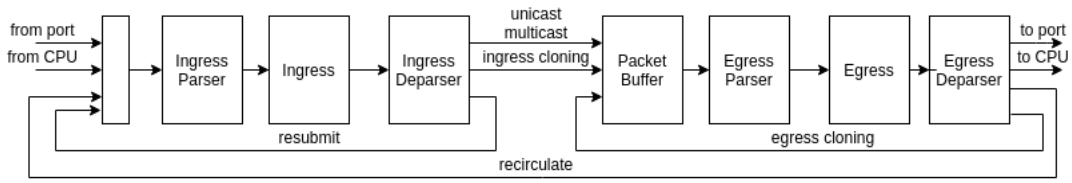
klonovat a provést rozhodnutí, zda bude paket recirkulován. Nakonec je paket odeslán do bloku *Deparser*, kde jsou do něj vloženy hlavičky, které byly dříve extrahovány. Z bloku *Deparser* je paket odeslán na port rozhraní, CPU port, recirkulační port, nebo port opětovného zaslání. Model architektury `v1model` pracuje s třemi různými hlavičkami vnitřních metadat. Prvními z nich jsou *Standard_metadata*, jejichž prvních 5 položek je povinných. Dalšími metadaty jsou *intrinsic_metadata* obsahující metadata pro muticast a časové značky přijetí a odeslání paketu. Posledními metadaty jsou *queueing_metadata* obsahující informace o stavu fronty mezi bloky *Ingress* a *Egress* v době, kdy do ní byl paket zařazen. *Intrinsic_metadata* a *queueing_metadata* nejsou povinná a tak s nimi P4 program tvořený pro model architektury `v1model` nemusí vůbec pracovat. Položky metadat *standard_metadata* jsou následující:

- **ingress_port** - číslo portu, na kterém byl paket přijat
- **packet_length** - délka paketu v bajtech
- **instance_type** - příznak typu paketu: normální, recirkulace, opětovné zaslání
- **egress_spec** - výstupní port paketu
- **egress_port** - výstupní port paketu, pouze pro čtení
- **checksum_error** - příznak neúspěchu metody verify_checksum()
- **parser_error** - příznak chyby vzniklé v bloku *Parser*

Po zpracování všech paketů není program ukončen. Místo toho hlavní vlákno simulačního modelu cyklí v nekonečné smyčce `while (true) std::this_thread::sleep_for (std::chrono::seconds(100));` a ukončení se z tohoto důvodu provádí signálem `SIGINT`.

Portable Switch Architecture

Portable switch architecture [13] je simulační model popisující běžné schopnosti síťového přepínače. Pracuje s modelem architektury `psa.p4`, který deklaruje programovatelné bloky *Ingress parser*, *Ingress*, *Ingress deparser*, *Egress parser*, *Egress* a *Egress deparser*. Dále obsahuje bloky *Packet Replication Engine* a *Buffering Queueing Engine*, které nejsou plně programovatelné pomocí P4, ale lze v nich volat pouze dovolené externí metody.



Obrázek 2.5: Průchod paketu modelem PSA switch. Převzato z [13]

Obrázek 2.5 popisuje průchod paketu tímto simulačním modelem. Porty jsou totožné s modelem *Simple switch*. Z paketů jsou extrahovány hlavičky v bloku *Ingress parser* a následně putují do bloku *Ingress*, po kterém jsou na rozdíl od modelu *Simple switch* opět složeny v bloku *Ingress deparser*. Odtud je mimo běžného unicastu paket možné zaslat nezměněný zpět bloku *ingress parser*, provést vstupní klonování nebo multicast. Pokud

paket nebyl zaslán zpět do bloku *Ingress parser*, je vložen do fronty odkud pokračuje do bloku *Egress parser*, kde jsou opět extrahovány hlavičky. Dále paket pokračuje do bloku *Egress*, kde je provedeno výstupní zpracování. Nakonec je paket složen v bloku *Egress deparser*. V této části je možné paket recirkulovat zpět do bloku *Ingress parser*, nebo provést výstupní klonování, které klonované pakety pošle zpět do bloku *Egress parser*. Ukončení tohoto simulačního modelu probíhá stejně jako v modelu *Simple switch*.

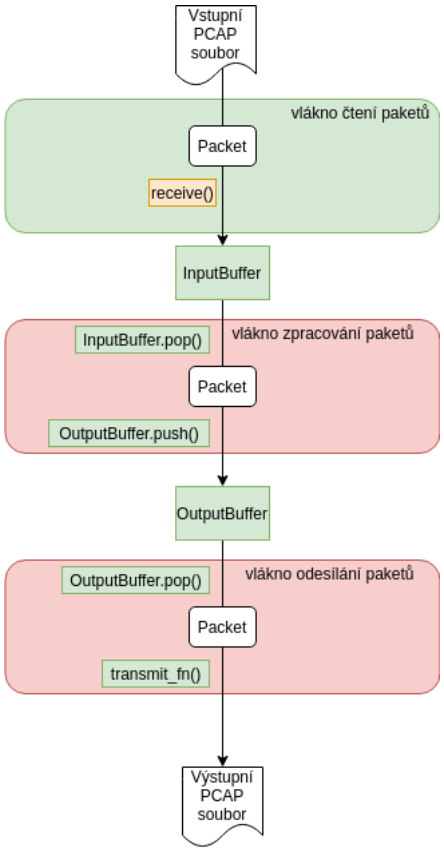
Model deklaruje vnitřní metadata nacházející se v sedmi hlavičkách. Jsou jimi:

- vstupní metadata pro blok ingress parser
- vstupní metadata pro blok ingress
- výstupní metadata pro blok ingress
- vstupní metadata pro blok egress parser
- vstupní metadata pro blok egress
- výstupní metadata pro blok egress
- vstupní metadata pro blok egress deparser

Většina těchto hlaviček obsahuje vstupní, nebo výstupní port a typ instance paketu. V metadatech vstupního a výstupního zpracování se nachází časové značky, příznaky zahození, klonování nebo opětovného zaslání paketu.

Implementace vlastního simulačního modelu

Při tvorbě vlastního simulačního modelu je nutné vytvořit třídu tohoto modelu, ve které je třeba vyřešit přijetí paketu, zpracování paketu a odeslání paketu. Práce simulačního modelu je rozdělena do několika vláken. Hlavní vlákno obstarává inicializaci zařízení, spuštění Thrift RPC serveru a spuštění dalších vláken simulačního modelu. Mimo toto hlavní vlákno obsahuje simulační model minimálně další tři vlákna. Jsou jimi: vlákno pro čtení paketů, vlákno pro zpracování paketů a vlákno pro odesílání paketů. Obrázek 2.6 zjednodušeně popisuje cestu paketu ze vstupního PCAP souboru, přes všechna vlákna až do výstupního PCAP souboru.



Obrázek 2.6: Více-vláknové zpracování uvnitř simulačního modelu

Zelená barva znázorňuje metody a objekty, jejichž implementaci obstarává projekt BMv2. Červená barva značí, že metodu musí implementovat programátor modelu. Oranžová barva značí, že metodu poskytuje projekt BMv2, avšak volá metody, které musí implementovat programátor simulačního modelu. Vlákno pro čtení paketů získá z PCAP souboru každý paket a v čase daném časovou značkou paketu ho vloží do vstupní fronty pomocí metody `receive()`. Ze vstupní fronty je paket vybrán metodou `pop()`, která je součástí implementace fronty. Ve vlákně pro zpracování paketů je před odesláním paket zpracován P4 programovatelnými bloky. Následně je metodou `push()` vložen do výstupní fronty, odkud je ve vlákně pro odesílání paketů vybrán metodou `pop()` a následně je zapsán pomocí metody `transmmit_fn()` do výstupního PCAP souboru.

Tvůrce simulačního modelu musí také provést inicializaci simulačního modelu a spuštění výše zmíněných vláken. Provedení těchto událostí bývá obvykle implementováno ve funkci *main*. Obrázek 2.7 znázorňuje postup volání metod ve funkci *main*. Funkcionalita zelených bloků je poskytnuta metodami, které jsou součástí projektu BMv2. Metody značené červeně musí uživatel naprogramovat. Oranžově značené metody jsou poskytnuté projektem BMv2, avšak využívají metody, které musí implementovat programátor simulačního modelu.

Obrázek 2.7: Main

Nejprve je instanciována třída simulačního modelu, která je následně inicializována parametry zadanými v příkazové řádce. Dále je spuštěn Thrift server, ke kterému je simulační model následně připojen. Nakonec jsou spuštěna všechna vlákna pracující s pakety.

Funkce main

V této sekci jsou popsány všechny základní možnosti jak provést kroky z obrázku 2.7. Všechny kroky, kromě inicializace lze provést pouze jedním způsobem. Pro inicializaci simulačního modelu však projekt BMv2 nabízí hned dvě možnosti.

Inicializace metodou `init_from_command_line_options()` Inicializaci pomocí této metody lze použít v případě, kdy programátor nepotřebuje přístup k argumentům z příkazové řádky před samotnou inicializací. Následující odrážky popisují jednotlivé kroky v pořadí, v jakém je třeba je provést. Tučně zvýrazněné kroky jsou povinné a ostatní kroky jsou volitelné

1. Instanciace třídy, která extrahuje specifické argumenty
   ```
   bm::TargetParserBasicWithDynModules specificke_argumenty_parser;
   ```
2. Přidání specifických argumentů
   ```
   specificke_argumenty_parser.add_flag_option("Název argumentu",
   "Nápověda k argumentu");
   //...Přidání dalších argumentů...
   ```
3. **Instanciace třídy vytvořeného simulačního modelu.**
   ```
   simulacni_model = new SimulacniModel();
   ```
4. **Inicializace simulačního modelu pomocí argumentů z příkazové řádky.**
   ```
   int status = simulacni_model->init_from_command_line_options(argc,
   argv, &specificke_argumenty_parser);
   if (status != 0) std::exit(status);
   ```
5. Práce se specifickými argumenty.
   ```
   bool specificky_argument = false;
   specificke_argumenty_parser.get_flag_option("Název argumentu",
   &specificky_argument)
   ```
6. **Spuštění Thrift serveru.**
   ```
   int thrift_port = simulacni_model->get_runtime_port();
   bm_runtime::start_server(simulacni_model, thrift_port);
   ```

7. **Spuštění zpracování paketů.**
   `simulacni_model->start_and_return();`

8. **Čekání na zpracování paketů**
   `while(true) std::this_thread::sleep_for(std::chrono::seconds(100));`

V případě, kdy programátor potřebuje přidat specifické argumenty do simulačního modelu, musí být instanciována třída pro extrakci těchto argumentů. Této instanci musí být předány informace o specifických argumentech. Těmito informacemi jsou název argumentu a text, který bude vypisovat argument `--help`. Instanciace třídy simulačního modelu probíhá pomocí konstruktoru. K inicializaci pomocí argumentů z příkazové řádky slouží metoda `init_from_command_line_options`, kterou třída simulačního modelu dědí od třídy `SwitchWContexts`. Základní volání této metody obsahuje pouze argumenty `argc` a `argv`, což jsou argumenty, se kterými byl spuštěn simulační model a jejich počet. Byla-li instanciována třída pro extrakci specifických argumentů, musí být této metodě předán ukazatel na ni jako třetí argument. Pokud metoda skončí neúspěchem, je program ukončen. Pro spuštění Thrift serveru je nejprve získán port, na kterém server poběží. Pokud nebyl port specifikován příznakem `--thrift-port`, je číslo portu 9090. Následně je server na tomto portu spuštěn. Spuštění zpracování paketů probíhá pomocí metody `start_and_return`, kterou třída simulačního modelu dědí od třídy `SwitchWContexts`. Poté program cyklí v nekonečné smyčce a po zpracování paketů musí být ukončen signálem SIGINT.

Třída pro extrakci specifických argumentů umožňuje přidání argumentu bez hodnoty pomocí metody `add_flag_option`, argumentu s hodnotou string pomocí `add_string_option`, argumentu s celočíselnou hodnotou pomocí `add_int_option` a argumentu s celočíselnou hodnotou bez znaménka pomocí `add_uint_option`. Pro získání hodnoty specifického argumentu slouží metody `get_flag_option`, `get_string_option`, `get_int_option` a `get_uint_option`. Argumenty těchto metod jsou název argumentu z příkazové řádky, který má být získán a ukazatel na proměnnou, do které se má hodnota uložit.

Inicializace pomocí metody `init_from_options_parser` Inicializace pomocí této metody je využita v případě, kdy potřebuje programátor přístup k argumentům před samotnou inicializací simulačního modelu. Tučně zvýrazněné kroky je nutné provést.

1. Instanciace třídy pro extrakci specifických argumentů a přidání specifických argumentů
   `Stejné jako v předchozí možnosti`

2. **Instanciace třídy pro extrakci základních argumentů**
   `bm::OptionsParser parser;`

3. **Extrakce základních a specifických argumentů**
   `parser.parse(argc, argv, &specificke_argumenty_parser);`

4. Práce s argumenty
   ```
   bool use_files_arg = parser.use_files
   bool enable_swap_flag = false;
   specificke_argumenty_parser.get_flag_option("enable-swap",
   &enable_swap_flag)
   ```

5. **Instanciace simulačního modelu**
   `simulacni_model = new SimulacniModel();`

6. **Inicializace simulačního modelu pomocí instance třídy pro extrakci základních argumentů**
   ```
   status = simulacni_model->init_from_options_parser(parser);
   if(status != 0) std::exit(status);
   ```

7. **Spuštění zpracování paketů a čekání na jejich zpracování**
   `Stejné jako v předchozí možnosti`

Po instanciaci třídy pro extrakci specifických argumentů a přidání specifických argumentů je nutné instanciovat třídu pro extrakci základních argumentů. Poté se provádí samotná extrakce argumentů pomocí metody `parse`. Argumenty této metody jsou `argc`, `argv` a ukazatel na instanci třídy pro extrakci specifických argumentů v případě, že byly specifické argumenty přidány do simulačního modelu. Instance pro extrakci základních argumentů umožňuje přistupovat základním argumentům přímo. Pro přístup k specifickým argumentům musí být využity metody zmíněné v předchozí možnosti. Následuje instanciace třídy simulačního modelu a jeho inicializace, která v tomto případě probíhá pomocí metody `init_from_options_parser`, jejímž argumentem je instance třídy pro extrakci základních příznaků.

Předání TransportIface a DevMgrIface Metody pro inicializaci simulačního modelu `init_from_options_parser` a `init_from_command_line_options` umožňují předání vlastních instancí tříd `TransportIface` a `DevMgrIface`. Předání instancí se používá v případě, kdy programátor vytvořil vlastní implementaci pro jedno z těchto rozhraní. Následující příkazy jsou příkladem inicializace s předáním ukazatele na vlastní implementaci `DevMgrIface`.

```
DevMgrIface *ptr_to_my_dev_mgr = nullptr;
ptr_to_my_dev_mgr = new FilesDevMgrImpMod(false,parser.wait_time);
simulacni_model->init_from_options_parser(parser,nullptr,
std::unique_ptr<DevMgrIface>(ptr_to_my_dev_mgr));}
```

Nejprve je vytvořen ukazatel na rozhraní `DevMgrIface` a následně je instanciována jeho vlastní implementace. Ukazatel je následně předán metodě `init_from_oprions_parser`, nebo `init_from_command_line_options`. Předání třetího argumentu jako `nullptr` značí, že nebyla předána vlastní implementace rozhraní `TransportIface`. V případě, kdy by ji chtěl programátor předat, je postup stejný jako u rozhraní `DevMgrIface`. Ostatní kroky lze provést stejně jako u jedné z předchozích možností.

Implementace třídy simulačního modelu

Pro simulační model musí být vytvořena třída, která musí dědit od jedné z tříd `Switch` nebo `SwitchWContexts`. V této třídě je nutné implementovat několik metod. Následující tři virtuální metody jsou součástí třídy `SwitchWContexts` a musí být v třídě simulačního modelu přepsány:

- `virtual int receive_(port_t port_num, const char *buffer, int len)`,
- `virtual void start_and_return_()`
- `virtual void reset_target_state_()`.

Metoda `receive_` slouží k přijetí paketu, metoda `start_and_return_` je používána pro spuštění zpracovávání paketů, metoda `reset_target_state_` slouží k uvedení simulačního modelu do požadovaného stavu.

Dále je nutné implementovat:

- konstruktor třídy
- metodu pro zpracování paketů
- metodu pro odesílání paketů
- instanciaci vstupní a výstupní fronty

Virtuální metody Následuje popis, k čemu každá z virtuálních metod slouží, kde je volána a jak ji v simulačním modelu implementovat.

receive__ Tato metoda slouží k přijetí nového paketu, nastavení požadovaných metadat paketu a jeho vložení do vstupní fronty. Argumenty metody `receive_` jsou:

- port_t port_num - číslo portu, na kterém byl paket přijat
- const char *buffer - zásobník na kterém je uložen paket
- int len - délka paketu v bajtech

V této metodě je nutné provést několik operací:

1. Vytvoření nové instance třídy `Packet`.
   ```
   auto packet = new_packet_ptr(port_num, packet_id++, len,
   bm::PacketBuffer(len + 512, buffer, len));
   ```
2. Získání ukazatele na instanci třídy `PHV`, nacházející se v instanci třídy `Packet`.
   ```
   PHV *phv = packet->get_phv();
   ```
3. Nastavení potřebných vnitřních metadat.
   ```
   phv->get_field("standard_metadata.ingress_port").set(port_num);
   //...Nastavení dalších metadat...
   ```
4. Vložení ukazatele na instanci `Packet` do vstupní fronty.
   ```
   input_buffer.push_front(std::move(packet))
   ```

Vytvoření nové instance třídy `Packet` probíhá pomocí metody `new_packet_ptr` třídy `SwitchWContexts`, jejíž návratová hodnota je ukazatelem na vytvořenou instanci třídy `Packet`. Metodě je předáno číslo portu, ID paketu, délka paketu a zásobník, do kterého bude paket vložen. Proměnná `len + 512` v konstruktoru třídy `PacketBuffer` zajišťuje, že v zásobníku bude 512 bajtů volného prostoru pro případ, kdy by byly simulačním modelem přidány, nebo prodlouženy hlavičky paketu. Dále je za účelem možnosti přístupu k metadatům paketu získán ukazatel na instanci třídy `PHV`. Následně je možné nastavovat požadovaná metadata. Nakonec musí být paket vložen do vstupní fronty. Pro tuto frontu je vhodné využít třídu `Queue`, která je součástí projektu BMv2. Deklarací fronty je `Queue<std::unique_ptr<Packet» input_buffer;`.

Metoda `receive_` je volána uvnitř metody `receive(port_t port_num, const char *buffer, int len)` třídy `SwitchWContexts`. Metoda `receive` je volána, když simulační model příjme nový paket. Toto volání lze vidět na obrázku 2.6.

start_and_return_() Metoda slouží k inicializaci prostředků, u kterých ji není možné provést v konstruktoru. Nejdůležitější operací prováděnou v této metodě je spuštění vláken zpracovávajících paket. Příkladem může být spuštění vláken zpracovávajících pakety. V následujícím příkladu implementace je metodou `pipeline_thread` myšlena metoda pro zpracování paketů a metodou `transmit_thread` metoda pro odesílání paketů.

```
std::thread t1(&SimulacniModel::pipeline_thread, this);
t1.detach();
std::thread t2(&SimulacniModel::transmit_thread, this);
t2.detach();
```

Metoda je volána uvnitř metody `start_and_return()` třídy `SwitchWContexts`. Volání metody `start_and_return` probíhá za účelem spuštění všech požadovaných vláken a její volání lze vidět na obrázku 2.7.

reset_target_state_() Metoda `reset_target_state_()` je volána uvnitř metody `reset_target_state()` třídy `SwitchWContexts`. Přepsání této metody není povinné. Metoda je využita pokaždé, když je kontrolní vrstvou volána metoda `reset_state()` a slouží k resetování zařízení do požadovaného stavu. Například simulační model Simple switch využívá metodu k resetování stavu replikačního stroje pro multicast.

Metody k implementaci Následující metody je nutné implementovat v třídě simulačního modelu

Konstruktor Konstruktor simulačního modelu slouží k inicializaci proměnných a deklaraci metadat. Implementace konstruktoru je pro každý simulační model jiná a odvíjí se od prostředků, které simulační model využívá, avšak každý konstruktor by měl instanciovat vstupní a výstupní frontu a deklarovat metadata, se kterými simulační model pracuje. Příkladem jednoduchého konstruktoru je kód:

```
SimulacniModel() : input_buffer(1024), output_buffer(128) {
  add_required_field("standard_metadata", "ingress_port");
  add_required_field("standard_metadata", "egress_port");
  force_arith_header("standard_metadata");
}
```

Tento kód inicializuje vstupní a výstupní frontu a specifikuje povinné položky vnitřních metadat s názvem *standard_metadata*. Metody pro přidání metadat jsou součástí třídy `SwitchWContexts`.

Metoda pro zpracování paketů Simulační model musí implementovat metodu spuštěnou jako vlákno, která provádí zpracování paketů. V této metodě musí být volány kontrolní P4 bloky v pořadí, jaké je určeno v modelu architektury.

Následující popis metody je určen pro model architektury obsahující bloky *Parser*, *Pipeline* a *Deparser*. Nejedná se o kompletní metodu, ale o výčet nejdůležitějších operací, které je v této metodě nutné provést. Metoda obsahuje nekonečný cyklus, ve kterém je nutné:

1. Získat ukazatele na kontrolní P4 bloky a deklarovat ukazatel na třídu PHV. Tento krok je vhodné provést před nekonečným cyklem.
   ```
   Pipeline *pipeline = this->get_pipeline("pipeline");
   Parser *parser = this->get_parser("parser");
   ```

```
Deparser *deparser = this->get_deparser("deparser");
PHV *phv;
```

2. Získat paket ze vstupní fronty.
```
std::unique_ptr<Packet> packet;
input_buffer.pop_back(&packet);
```

3. Získat přístup k metadatům paketu.
```
phv = packet->get_phv();
```

4. Aplikovat na paket zpracování blokem *Parser* a nastavit vnitřní metadata paketu, která je po extrakci hlaviček nutná nastavit. Nastavení metadat závisí na modelu architektury, který simulační model implementuje.
```
parser->parse(packet.get());
```

5. Aplikovat na paket zpracování kontrolním blokem *Pipeline* a nastavit vnitřní metadata paketu, které je po zpracování paketu nutná nastavit. Dále lze s těmito metadaty pracovat.
```
pipeline->apply(packet.get());
```

6. Nastavit výstupní port paketu. P4 kód umožňuje měnit výstupní port ve vnitřních metadatech. Avšak pro odeslání přes správné rozhraní musí být výstupní port nastaven i v třídě `Packet`
```
int egress_spec = phv->get_field("intrinsic_metadata.egress_port")
.get_int();
packet->set_egress_port(egress_spec);
```

7. Zahodit paket, bylo-li tak určeno P4 kódem. P4 kód nastaví výstupnímu portu hodnotu, která značí, že má být paket zahozen. V tomto případě je hodnota 511. Zahození je provedeno tak, že paket není vložen do výstupní fronty.
```
int egress_port = phv->get_field("standard_metadata.egress_port")
.get_int();
if(egress_port == 511) continue;
```

8. Aplikovat na paket zpracování blokem *Deparser*.
```
deparser->deparse(packet.get());
```

9. Vložit paket do výstupní fronty.
```
output_buffer.push_front(std::move(packet));
```

Všechny výše využívané metody jsou součástí projektu BMv2. Práce s vnitřními metadaty plně závisí na modelu architektury, který simulační model implementuje. V případě, kdy model architektury obsahuje více kontrolních bloků zpracovávajících paket, je možné pro každý z těchto bloků implementovat jednu metodu. Příkladem je model architektury v1model, který deklaruje dva bloky zpracovávající paket a to *Ingress* a *Egress*. V simulačním modelu *Simple switch*, který implementuje tento model je implementována metoda pro každý z těchto bloků. V takovém případě je nutné řešit předání mezi těmito metodami pomocí fronty.

Metoda pro odesílání paketů Simulační model musí mít metodu, která se chová jako vlákno zapisující pakety do výstupního PCAP souboru. V metodě je nekonečný cyklus, ve kterém je nutné:

1. Získat paket z výstupní fronty
   `std::unique_ptr<Packet> packet;`
   `output_buffer.pop_back(&packet);`

2. Zapsat paket do výstupního PCAP souboru
   `transmit_fn(packet->get_egress_port(), packet->data(),`
   `packet->get_data_size());`

Metoda `void transmit_fn(port_t port_num, const char *buffer, int len)` je součástí třídy `DevMgr` a slouží k zápisu paketu do příslušného PCAP souboru. Jejími argumenty jsou výstupní port paketu, zásobník, ve kterém jsou uložena data paketu a délka paketu.

Důležité třídy BMv2

BMv2 je velmi robustní projekt obsahující velké množství tříd a metod. K tvorbě vlastního simulačního modelu však není potřebná kompletní znalost všech tříd, jelikož v některých třídách jsou řešeny nízkoúrovňové problémy, se kterými se následně programátor nemusí zabývat. Tím je umožněna jednodušší tvorba vlastních modelů.

V rámci pochopení chodu a programování simulačních modelů je v následující sekci popsán účel nejdůležitějších tříd v projektu BMv2, se kterými bude programátor při tvorbě simulačního modelu pracovat. V příloze B je diagramem nejdůležitějších tříd. Třídy v diagramu obsahují pouze metody a proměnné, které jsou v práci zmíněny. Metody dále neobsahují vstupní parametry. Ty jsou obsaženy v textu popisu třídy, které je metoda součástí.

SwitchWContexts a Switch

Je nutné, aby třída simulačního modelu dědila od třídy `Switch`, nebo `SwitchWContexts`. `SwitchWContexts` umožňuje programovat simulační model s libovolným počtem objektů `Context` avšak velké množství simulačních modelů více objektů `Context` nepotřebuje. Více objektů třídy `Context` umožňuje, aby byl simulační model nakonfigurován více P4 programy. To by umožňovalo simulačnímu modelu mít dvě různé definice jednoho P4 programovatelného bloku. Jeden objekt třídy `Context` udržuje informace o jedné konfiguraci. Překladač p4c však zatím tuto funkcionalitu nepodporuje. To je také jedním z důvodů, proč v projektu BMv2 existuje třída `Switch`, která dědí od třídy `SwitchWContexts` a která pracuje pouze s jedním objektem třídy `Context`.

Třída `SwitchWContexts` mimo jiné obsahuje:

- Proměnnou `arith_objects` obsahující vnitřní metadata a metodu `force_arith_header(const std::string &header_name)` pro jejich přidání do modelu.
- Metodu `bool field_exists(cxt_id_t cxt_id, const std::string &header_name, const std::string &field_name)` pro zjištění, zda byla položka metadat definovaná ve vstupním JSON souboru.
- Informaci o povinných polích vnitřních metadat nacházející se v proměnné `required_fields`.

- Proměnnou `std::string current_config` obsahující aktuálně využívaný JSON soubor.
- Proměnnou `thrift_port`, ve které je uložen port, na kterém je spuštěn Thrift server.
- Metody pro inicializaci zařízení podle argumentů z příkazové řádky.
```
int init_from_command_line_options(int argc, char *argv[],
TargetParserIface *tp = nullptr,
std::shared_ptr<TransportIface> my_transport = nullptr,
std::unique_ptr<DevMgrIface> my_dev_mgr = nullptr);

init_from_options_parser(const OptionsParser &parser,
std::shared_ptr<TransportIface> my_transport = nullptr,
std::unique_ptr<DevMgrIface> my_dev_mgr = nullptr);
```
- Metodu pro vytvoření nového objektu třídy `Packet`.
```
std::unique_ptr<Packet> new_packet_ptr(cxt_id_t cxt_id,
port_t ingress_port, packet_id_t id, int ingress_length,
PacketBuffer &&buffer);
```
- Výše zmíněné metody `start_and_return()`, `receive()` a `reset_target_state()`
- Výše zmíněné virtuální metody `start_and_return_()`, `receive_()` a `reset_target_state_()`
- Metody ze sekce runtime interface, které umožňují například správu tabulek a čítačů.

Třída `Switch`

- Implementuje metody `Pipeline *get_pipeline(const std::string &name)`, `Parser *get_parser(const std::string &name)` a `Deparser *get_deparser(const std::string &name)` vracející ukazatel na daný P4 kontrolní blok.
- Přetěžuje metody třídy `SwitchWContexts`, které pracují s konkrétním objektem třídy `Context`.

DevMgr a DevMgrIface

Třída `DevMgr` je rodičovskou třídou pro třídu `SwitchWContexts`. Obsahuje funkce pro odesílání paketů, čtení paketů a správu portů. Poskytuje rozdílné implementace, které implementují rozhraní `DevMgrIface`. V třídě `DevMgr` je obsažen ukazatel na jednu z implementací, kterými jsou:

- `BmiDevMgrImp` - Je implementace využívající knihovnu BMI k přijímání a odesílání paketů. V tomto případě je tok paketů odposloucháván z reálného síťového rozhraní. Výstupní pakety jsou zapisovány do PCAP souboru v případě, že je tak specifikováno pomocí příznaku `--pcap` při spuštění simulačního modelu.
- `PacketInDevMgrImp` - Je implementace využívající nanomsg PAIR socket k zasílání a přijímání paketů.
- `FilesDevMgrImp` - Je implementace, která čte a zapisuje pakety do PCAP souborů pomocí tříd `PcapFilesReader` a `PcapFilesWriter`.

PcapFilesReader a PcapFilesWriter

Třídy `PcapFilesReader` a `PcapFilesWriter` slouží ke čtení nebo zápisu paketů do PCAP souborů. Objekt třídy `FilesDevMgrImp` spouští vlákno pro čtení a řídí ho. Funkce `scan()` třídy `PcapFilesReader` následně prochází vstupní PCAP soubory a pakety následně podle časových značek zasílá do vstupní fronty simulačního modelu. Po jejich zpracování jsou pakety zaslány do výstupního fronty, odkud jsou vybírány a zapisovány do výstupních souborů.

Packet a PHV

`Packet` je jednou ze základních tříd BMv2, která reprezentuje paket během jeho průchodu simulačním modelem. Tato třída je finální. Vznikne-li při tvorbě simulačního modelu potřeba rozšířit tuto třídu, je nutné použít kompozici místo dědičnosti. Dále umožňuje třídám `SwitchWContexts` a `Switch` přistupovat k jejím *private* a *protected* prvkům pomocí klíčového slova *friend*.

Projde-li paket několika bloky pro extrakci hlaviček a bloky pro složení paketu, stále ho reprezentuje stejná instance této třídy. Každá instance obsahuje `packet_id`, které je přiřazeno při instanciaci objektu. Je-li nutné paket v simulačním modelu duplikovat, je vytvořena nová instance pro kopii. Proměnná `packet_id` originálu a kopie je totožná. Pro rozlišení kopií slouží proměnná `clone_id`. Každá instance obsahuje 4 obecné registry, které může programátor simulačního modelu využít pro libovolný účel. Data paketu jsou v této třídě uložena do instance třídy `PacketBuffer`. Třída obsahuje metody `int get_egress_port()` a `void set_egress_port(int port)` pro získání a nastavení výstupního portu. Stejné metody existují i pro vstupní port. Dalšími metodami jsou metody pro klonování paketu, získání obsahu paketu a nastavení chyby kontrolního součtu. Dále obsahuje jednu instanci třídy `PHV`, která slouží k uložení dat extrahovaných z paketu pomocí bloku pro extrakci hlaviček. Dále poskytuje metody pro práci s jednotlivými hlavičkami a poli v těchto hlavičkách. Skládá se z vektoru instancí hlaviček, kde se každá tato instance skládá z vektoru instancí jednotlivých polí v hlavičce.

Queue

Třída `Queue` poskytuje základní implementaci fronty. Implementace poskytuje zamykání pomocí semaforů a tudíž je bezpečná při používání stejné fronty více vlákny. Tato třída využívá implementaci fronty `std::deque`. Potřebuje-li programátor, aby simulační model podporoval pokročilejší funkcionality, například prioritní fronty, musí vyvořit vlastní implementaci fronty.

Parser, Deparser, Pipeline

Třída Parser poskytuje metodu `parse(Packet *pkt)` spouštějící extrakci hlaviček z paketu. Třída `Pipeline` poskytuje metodu `apply(Packet *pkt)`, spouštějící zpracování paketu daným kontrolním blokem. Třída `Deparser` poskytuje metodu `deparse(Packet *pkt)`, která vloží extrahované hlavičky zpět do paketu. Při tvorbě simulačního modelu se programátor s dalšími metodami těchto tříd nesetká.

ExternType, Data

Třída `ExternType` slouží k implementaci vlastních externích tříd. Každá externí třída dědí od třídy `ExternType`. Tato třída umožňuje přístup k aktuálně zpracovávanému paketu pomocí metody `Packet &get_packet()`. Dále slouží k registraci externích tříd.

Třída `Data` je jedním ze základních datových typů pro argumenty metod externích tříd. Její hodnotou může být libovolná číselná hodnota. Poskytuje metody pro nastavení hodnoty, získání hodnoty a provádění matematických operací nad dvěma instancemi této třídy

Implementace pokročilých vlastností

V této sekci se práce zabývá změnami v kódu simulačního modelu za účelem implementace pokročilejších funkcionalit modelu.

Recirkulace

Funkcionalita recirkulace je zaslání paketu na opětovné zpracování po tom, co proběhne jeho zpracování. Zda bude paket recirkulován je určeno v P4 kódu. Toto určení je nejjednodušší provést přidáním příznaku pro recirkulaci do metadat modelu architektury. V samotném P4 kódu je tento příznak nastaven při požadavku na recirkulaci. V metodě pro zpracování paketů ihned za dokončením zpracování paketu pomocí bloku *Deparser* je třeba zjistit, zda má být paket recirkulován, což lze zjistit z metadat. Paket, který má být recirkulován je místo odeslání do výstupní fronty odeslán zpět do vstupní fronty.

Příkladem implementace recirkulace je tento kód:

```
\\... zpracovani paketu ...
int recirc_flag = phv->get_field("standard_metadata.recirc_flag")
        .get_int();
deparser->deparse(packet.get());
if(recirc_flag == 0) {
    input_buffer.push_front(std::move(packet));
    continue;
}
\\.. pokracovani zpracovani paketu ...
```

Multicast

Pro multicast je v simulačním modelu vhodné doplnit dvě položky metadat. Jednu pro multicastovou skupinu paketu a druhou pro replikační ID paketu z multicastové skupiny. Dále musí být v konstruktoru simulačního modelu inicializován replikační stroj. Tento stroj je modifikovatelný kontrolní vrstvou. Inicializace probíhá pomocí kódu:

```
SimulacniModel() : ..., pre(new McSimplePreLAG()), ... {
  //...
  add_component<McSimplePreLAG>(pre);
  //...
}
```

V metodě zpracování paketu lze multicast provést takto:

```
\\...
```

```
unsigned int mgid = phv->get_field("standard_metadata.mcast_grp")
        .get_uint();
if (mgid != 0) {
  const auto pre_out = pre->replicate({mgid});

  for(const auto &out : pre_out){
    std::unique_ptr<Packet> packet_copy =
            packet->clone_with_phv_ptr();
    phv->get_field("standard_metadata.egress_port")
            .set(out.egress_port);
    phv->get_field("standard_metadata.egress_rid")
            .set(out.rid);
    output_buffer.push_front(std::move(packet));
  }
  continue;
}
\\...
```

Po získání čísla multicastové skupiny je volána metoda `replicate(mgid)`, která pro danou multicastovou skupinu vrací pole struktur `McOut`, ve které se nachází replikační ID multicastové kopie a výstupní port multicastové kopie. Poté je v cyklu tvořen pro každý prvek ze získaného pole nový paket pomocí metody `clone_with_phv_ptr()`, která vrací ukazatel na klonovaný paket. Tento paket se od původního liší pouze v hodnotě `copy_id`, která je je součástí paketu. Do metadat klonovaného paketu jsou nastaveny hodnoty výstupního portu a relplikační ID paketu. Nakonec je paket odeslán do výstupní fronty. Původní paket je zahozen.

Prioritní fronty

V simulačním modelu je implementace prioritních front možná. K jejich implementaci je nutné vytvořit vlastní vstupní frontu. Jednoduchým příkladem může být požadavek vyšší priority recirkulovaných paketů. V takovém případě jsou potřebné dvě fronty. Jedna s vyšší prioritou, do které budou vkládány recirkulované pakety a jedna s nižší, do které bude vkládán zbytek paketů. Při odebrání paketu z fronty je nejprve zjištěno, zda se nějaký paket nachází ve frontě s vyšší prioritou. Pokud ano, je paket vybrán z této fronty. V opačném příkladě je paket vybrán z fronty s nižší prioritou. Při implementaci vlastních front je vhodné rozšířit třídu `Queue`, je-li to možné, zmiňovanou výše. V opačném případě je vhodné se touto třídou inspirovat.

Implementace externí třídy a externí funkce

Externí třídy Implementace externí třídy musí splňovat několik podmínek, které jsou popsány na příkladu externí třídy `ExternCounter`[1]. Následující kód je deklarací externí třídy v hlavičkovém souboru.

```
class ExternCounter : public ExternType {
 public:
  BM_EXTERN_ATTRIBUTES {
```

[1]Inspirací pro tuto třídu byla externí třída `ExternCounter` nacházející se na adrese https://github.com/p4lang/behavioral-model/blob/master/tests/test_extern.cpp na řádku 33. Navštíveno dne 2.2.2020

```
    BM_EXTERN_ATTRIBUTE_ADD(init_count);
  }
    ExternCounter();
    void reset();
    void init() override;
    void increment();
    void increment_by(const Data &d);
    size_t get() const;

 private:
  Data init_count;

  size_t init_count_{0};
  size_t count{0};
};

int import_extern_example();
```

Následující odrážky popisují podmínky, které je nutné splnit při implementaci externí třídy.

- Externí třída musí dědit od třídy `ExternType`, která je součástí projektu BMv2.
- V tvořené třídě musí být přepsána metoda `init()`, která je součástí třídy `ExternType`, ve které je deklarována jako virtuální a která slouží k inicializaci externí třídy.
- Atributy, které je nutné nastavit mimo externí třídu, je nutné registrovat pomocí makra `BM_EXTERN_ATTRIBUTES` uvnitř kterého jsou jednotlivé argumenty registrovány pomocí makra `BM_EXTERN_ATTRIBUTE_ADD(Název atributu);`
- Implementace vlastního konstruktoru je možná, avšak konstruktor nesmí mít argumenty, jelikož při registraci externí třídy musí být dodržena podmínka `std::is_default_constructible`, která značí, že typ musí být vytvořen bez inicializačních hodnot, nebo argumentů.
- Za implementací třídy, je nutno externí třídu registrovat pomocí makra `BM_REGISTER_EXTERN(Název třídy)` a dále registrovat všechny metody makrem `BM_REGISTER_EXTERN_METHOD(Název externí třídy, název metody, první argument, ..., n-tý argument)`.
- V kódu simulačního modelu, ideálně v konstruktoru, musí být volána funkce `int import_extern(){ return 0;}`, jejíž implementace se nachází v souboru s implementací externí třídy, aby bylo při překladu vynuceno propojení tohoto souboru k simulačnímu modelu.

Následující kód je příkladem implementace metody `increment_by`, která navýší čítač o číslo zadané ve vstupním argumentu:

```
void ExternCounter::increment_by(const Data &d) {
  count += d.get<size_t>();
}
```

Argumenty externích metod mají povolenou omezenou sadu datových typů, z nichž nejdůležitější jsou[2]:

- const Data & - pro jakoukoliv P4 numerickou hodnotu.
- [const] Field & - pro odkaz na pole.
- [const] Header & - pro odkaz na hlavičku.
- const std::string & - pro řetězec.
- const std::vector<Data> - pro pole numerických hodnot.

Externí funkce Pro implementaci externí funkce platí stejná pravidla, jako pro implementaci externí metody. Pro vstupní argumenty a návratové hodnoty může tedy využívat pouze výše zmíněné datové typy. Registrace externí funkce probíhá pomocí makra `BM_REGISTER_EXTERN_FUNCTION(Název funkce);`

[2]Kompletní výčet datových typů lze nalézt na https://github.com/p4lang/behavioral-model/blob/master/include/bm/bm_sim/actions.h na řádku 50. Navštíveno dne 14.1.2020.

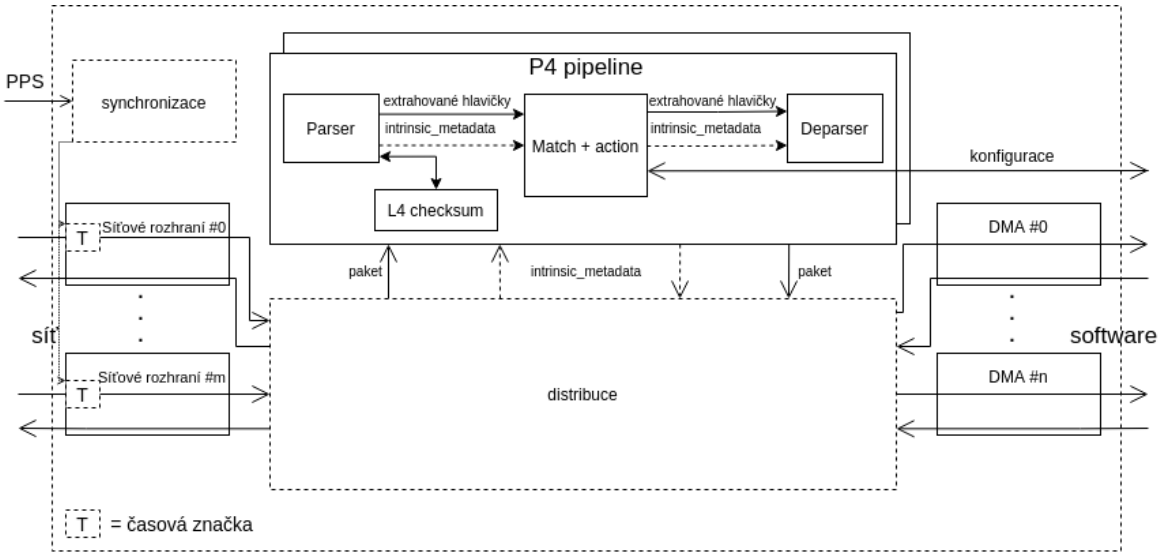
Kapitola 3

Návrh a implementace

V úvodu této kapitoly je popsána architektura COMBO-200G2QL, pro níž má být v této práci implementovaný simulační model. Dále jsou popsány požadavky na výsledný simulační model, které vycházejí z výše zmíněné architektury. Následuje návrh implementace simulačního modelu dle požadavků a jeho implementace.

Architektura COMBO-200G2QL

Jelikož má simulační model odpovídat architektuře P4 jádra, které je společné pro všechny karty rodiny COMBO, je zde popsána konkrétní architektura COMBO-200G2QL [8]. Obrázek 3.1 je schématem této architektury.



Obrázek 3.1: Architektura COMBO-200G2QL. Převzato z [8]

Architektura umožňuje přijímání a odesílání paketů z fyzických síťových rozhraní, nebo ze softwaru přes kanál přímého přístupu do paměti (DMA channel). Vnitřní hodiny jsou synchronizovány přes PPS (Pusle Per Second) vstup karty. Pomocí těchto vnitřních hodin jsou získávány časové značky přijetí paketu, které jsou ukládány do příslušné položky *instrinsic_metadat*. Blok distribuce se stará o mapování front jednotlivých síťových rozhraní a DMA kanálů na jednotlivá *P4 pipeline* jádra. *P4 pipeline* jádro se skládá ze všech prvků

provádějící funkcionalitu popsanou v P4 programu. P4 programovatelnými bloky této architektury jsou: *Parser*, *Match + action* a *Deparser*. Z bloku *Parser* je možné přejít na výpočet L4 kontrolního součtu, jehož výsledek je uložen do příslušné položky metadat. Architektura obsahuje dvě tyto jádra, která jsou napojena na blok distribuce, avšak z prostorových důvodů je znázorněno pouze jedno. Položky *intrinsic_metadat* jsou následující:

- **ingress_timestamp** - Časový údaj o příchodu paketu na vstup.
- **ingress_port** - Číslo vstupního portu.
- **egress_port** - Číslo výstupního portu.
- **packet_len** - Délka paketu.
- **hash, user16 a user4** - Umožňují odeslat uživatelská data do softwaru.

Požadavky na simulační model

Účelem práce je vytvořit simulační model, který bude splňovat zadané požadavky a který bude sloužit pro účely verifikace zařízení vyvíjeného sdružením CESNET. Těmito požadavky jsou: model P4 architektury odpovídající architektuře P4 jádra karet rodiny COMBO, podpora metadat P4 jádra karet rodiny COMBO, okamžité ukončení běhu modelu po zpracování paketů ze vstupních PCAP souborů a vytvoření flexibilního mechanismu pro přidávání funkcionality nových externích modulů.

Sdružení CESNET aktuálně pro účely verifikace využívá simulační model *Simple switch*. Tento model je využíván kvůli podobnosti jeho modelu architektury s požadovanou architekturou. To je důvod, proč implementace bude probíhat úpravami tohoto modelu. Úpravy budou také méně časově náročné oproti tvorbě nového modelu. Úpravy, které budou provedeny jsou následující:

- **Odstranění nepotřebného kódu a jeho zjednodušení** - *Simple switch* implementuje funkcionality, které architektura COMBO nevyužívá. Tyto funkce jsou: klonování paketů, opětovné zpracování paketu a multicast. Tyto funkcionality budou v metodě implementující funkční blok *Ingress pipeline* odstraněny. Jelikož požadovaný simulační model nemá implementovat ani prioritní fronty, bude možné zjednodušit implementaci vstupní fronty. Model dále nemá implementovat metodu pro funkční blok *Egresss pipeline*. Z toho důvodu bude upravena metoda pro funkční blok *Ingress pipeline* a metoda pro *Egress pipeline* bude odstraněna.
- **Metadata** - Požadovaný simulační model má oproti modelu *Simple switch* pracovat s odlišnou sadou metadat. Metadata musí být v simulačním modelu deklarována a model musí být přizpůsoben pro práci s těmito metadaty. *Simple switch* využívá metadata, která nemají být součástí požadovaného modelu. Jejich odstranění nemělo ovlivnit fungování výsledného simulačního modelu.
- **Model architektury** - Model architektury v1model musí deklarovat výše zmíněnou sadu metadat. Dále musí být odstraněna deklarace původních metadat a deklarace funkčního bloku *Egress pipeline*.
- **Ukončení simulace** - *Simple switch* nepodporuje ukončení programu po zpracování všech paketů z PCAP souboru. Místo toho simulační model cyklí v nekonečné smyčce, dokud není ukončen uživatelem. Pro účely verifikace je vhodné, aby se model po zpracování všech vstupních paketů z PCAP souboru korektně ukončil.

- **Načítání externích modulů** - *Simple switch* umožňuje vytváření externích tříd a funkcí, avšak výsledné zařízení by mělo umožňovat jejich dynamické načítání, obecné volání metod a jednoduchou implementaci nových modulů bez nutnosti většího zásahu do kódu zařízení.

3.1 Návrh a implementace vytvoření nového simulačního modelu

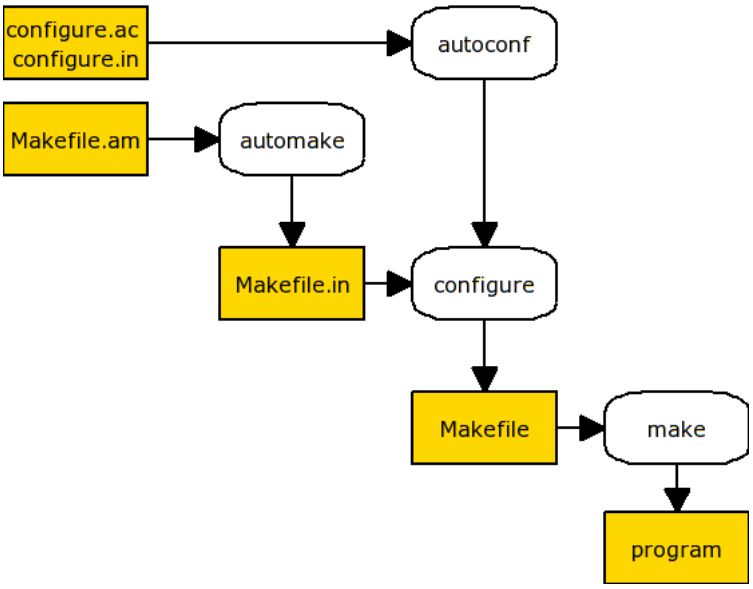
Návrh

Za účelem vytvoření nového simulačního modelu musí být vytvořena kopie simulačního modelu *Simple switch*. Všechny soubory této kopie, které je nutné překládat musí být vloženy do již existujícího souboru *Makefile*, který překládá celý BMv2 projekt. Dále musí být upraven soubor *Makefile* zkopírovaného simulačního modelu. Jelikož bude nutné měnit i model architektury `v1model`, musí být vytvořena i kopie tohoto souboru.

Implementace

Pro práci na bakalářské práci byl poskytnut virtuální stroj, na kterém již byly všechny potřebné balíčky nainstalované [1]. Nejprve musel být stažen celý projekt BMv2 [9]. Ve složce `/behavioral-model/targets` byla vytvořena kopie složky `simple_switch` obsahující zdrojové kódy simulačního modelu *Simple switch*. Kopie složky dostala název `smart_NIC` a simulační model v této složce bude v práci dále nazýván stejným jménem. V této fázi bylo nutné zajistit korektní překlad simulačního modelu *Smart NIC*. Projekt BMv2 využívá pro překlad balíček Autotools [5], který funguje tak, že programátor vytvoří soubor *configure.ac* a *Makefile.am*. Z těchto souborů je pomocí nástrojů balíčku Autotools automaticky generovaný soubor *Makefile* používaný pro překlad simulačního modelu. Obrázek 3.2 zobrazuje postup generování souboru *Makefile*.

[1] Kompletní výčet potřebných balíčků lze nalézt na https://github.com/p4lang/behavioral-model/blob/master/README.md



Obrázek 3.2: Postup překladu s balíčkem Autotools. Převzato z [14]

Za účelem překladu modelu *Smart NIC* bylo nutné upravit 3 soubory:

- **/behavioral-model/targets/smart_NIC/Makefile.am -** Tento soubor mimo jiné obsahuje informace o knihovnách pro daný simulační model. *Makefile.am* pro původní simulační model generuje knihovny `libsimpleswitch.la` a `libsimpleswitch_runner.la`. Tyto knihovny byly přejmenovány na `libsmartnic.la` a `libsmartnic_runner.la`. Do tohoto souboru budou také později přidávané cesty k nově vytvořeným souborům, které bude nutné překládat.
- **/behavioral-model/targets/Makefile.am -** Tento soubor obsahuje názvy složek, ve kterých jsou uloženy simulační modely. Do proměnné `SUBDIRS` v tomto souboru je přidán název složky `smart_NIC`.
- **/behavioral-model/configure.ac -** Tento soubor obsahuje mimo jiné cestu ke všem souborům *Makefile*, které má vygenerovat. Tyto cesty začínají na řádku 277. Mezi tyto cesty musí být přidán *Makefile* pro simulační model *Smart NIC*.

Nyní mohou být pomocí následující sekvence příkazů provedených ze zdrojové složky `behavioral-model` vytvořeny všechny soubory *Makefile*. Úprava souborů zajistila korektní vytvoření souboru *Makefile* i pro model *Smart NIC*. Příkaz `make` překládá všechny soubory projektu BMv2.

1. ./autogen.sh
2. ./configure
3. make

První dva příkazy je nutné provádět pouze v případě, kdy byl upraven buď soubor *configure.ac* a nebo kterýkoliv ze souborů *Makefile.am*. Stačí-li přeložit pouze simulační model, lze provést pouze příkaz `make`.

3.2 Návrh a implementace odstranění a úpravy kódu simulačního modelu

Návrh

Simple switch obsahuje vlastní implementaci vstupní fronty zvanou `InputBuffer`, která zajišťuje rozdílné priority pro různé druhy paketů. Pro požadovaný simulační model bude pro vstupní frontu postačovat třída `Queue`, která je součástí projektu BMv2. Bude vytvořena instance této třídy, a metody `transmit_thread` a `ingress_thread` pracující se vstupní frontou budou upraveny pro práci s frontou třídy `Queue`.

V modelu může být odstraněna metoda `egress_thread` implementující výstupní zpracování, ve kterém se nachází zpracování P4 programovatelným blokem *Egress pipeline*. Z tohoto důvodu však musí být upravena metoda `ingress_thread` implementující vstupní zpracování. Na konci této metody musí být paket místo vložení do fronty mezi vstupním a výstupním zpracováním vložen do výstupní fronty. Dále musí být hned po dokončení zpracování funkčním blokem *Ingress pipeline* nastaven výstupní port paketu, jelikož o tento krok se v modelu *Simple switch* stará implementace fronty mezi vstupním a výstupním zpracováním. Implementace této fronty může být z simulačním modelu odstraněna.

Pro odstranění funkcionality klonování paketů, zaslání paketu k opětovnému zpracování a multicastu by mělo být postačující odstranit metody a části kódu v metodě `ingress_thread`, které tuto funkcionalitu implementují.

Implementace

Speciální akce pro paket V metodě `ingress_thread` simulačního modelu *Simple switch* probíhá po zpracování programovatelným P4 blokem *ingress* dotazovaní na akce, které se mají provést s paketem. Kostra tohoto dotazování vypadá takto:

```
if (clone){
    //... kod klonovani paketu ...
}
if (learning) {
    //... kod uceni ...
}
if (resubmit) {
    //... kod opetovneho zaslani ...
} else if (mcast_grp != 0) {
    //... kod multicastu ...
} else if (egress_spec == DROP_PORT) {
    //... kod zahozeni ...
} else {
    //... kod unicastu ...
}
```

V simulačním modelu byla tato kostra upravena na následující tvar při čemž všechny větve nevyskytující se v této kostře byly vymazány:

```
if (egress_spec == DROP_PORT) {
    //... kod zahozeni ...
} else {
    //... kod unicastu ...
}
```

Simulační model obsahoval metody využívané při klonování paketů. Jsou jimi `mirroring_get_session`, `mirroring_add_session` a `mirroring_delete_session`. Dále pro tyto účely obsahuje třídu `MirroringSessions`. Tato třída mohla být odstraněna, avšak pro odstranění výše zmíněných metod by musely být upraveny soubory generované Thrift překladačem, nebo by musely být vygenerovány znovu. Z tohoto důvodu tyto metody zatím vypisují informaci o jejich neimplementování.

Vstupní fronta V simulačním modelu bylo možné provést zjednodušení implementace vstupní fronty. Pro tu je v simulačním modelu *Simple switch* implementovaná třída `InputBuffer`. Implementace této třídy mohla být odstraněna. Avšak vstupní fronta musela být nahrazena frontou třídy `Queue`. Do deklarace třídy byla přidána deklarace *private* proměnné `Queue<std::unique_ptr<Packet>> input_buffer`. Dále byla v konstruktoru třídy provedena inicializace pomocí příkazu `input_buffer(1024)`. Číslo 1024 udává kapacitu vstupní fonty. V metodě `receive_` je původní kód vložení paketu do fronty nahrazen příkazem `input_buffer.push_front(std::move(packet))` a v metodě `ingress_thread` je původní kód získání paketu z fronty nahrazen kódem `input_buffer.pop_back(&packet)`.

Výstupní zpracování Dále bylo možné odstranit metodu `egress_thread`, avšak pro korektní funkcionalitu musela být upravena metoda `ingress_thread`. V původní metodě je na jejím konci paket vložen do fronty mezi vlákny `ingress_thread` a `egress_thread` pomocí metody `enqueue`. V simulačním modelu *Smart NIC* však bylo nutné konec metody `ingress_thread` upravit následovně:

```
//...
int egress_spec = phv->get_field("intrinsic_metadata.egress_port")
                  .get_int();
packet->set_egress_port(egress_spec);
if (egress_spec == 511) {
   // ... kod zahozeni paketu ...
   continue;
}
deparser->deparse(packet.get());
output_buffer.push_front(std::move(packet));
```

Tučně zvýrazněné řádky byly do metody přidány, nebo byly upraveny. V metodě je z vnitřních metadat získán výstupní port, který byl nastaven v P4 kódu. Výstupní port musí být dále nastaven v instanci třídy `packet`. Toto nastavení v původním modelu probíhá v metodě `enqueue`. Ta však bude odstraněna, a proto musí být příkaz nastavení portu přidán zde. Následně probíhá možné zahození paketu, za kterým je uskutečněn kód P4 bloku *deparser*, který se původně prováděl v metodě `egress_thread`. Nakonec je paket vložen do

výstupní fronty, místo fronty mezi vlákny `ingress_thread` a `egress_thread`. Nyní mohla být odstraněna fronta `egress_buffers` mezi vlákny. Byla odstraněna její deklarace i její inicializace v konstruktoru. Pro tuto frontu byly implementovány čtyři metody:

- `set_egress_queue_depth`
- `set_all_egress_queue_depths`
- `set_egress_queue_rate`
- `set_all_egress_queue_rates`

Stejně jako u metod pro klonování paketu by musely pro jejich odstranění být upraveny automaticky vygenerované soubory. Z toho důvodu metody prozatím vypisují chybovou hlášku o tom, že nejsou implementovány.

Návrh a implementace metadat a modelu architektury

Návrh

Požadovaná architektura musí pracovat s výše zmíněnými *Intrinsic_metadaty*, která obsahují některé položky *Standard_metadat* a *Intrinsic_metadat* simulačního modelu *Simple switch*. Dále jsou rozšířena o položky *hash*, *user16* a *user4*. Metadata v původním modelu architektury *v1model* musí být odstraněna a nová metadata musí být přidána. Následně musí být v kódu simulačního modelu upraveny nebo odstraněny ty části kódu, které pracují s původními metadaty. Z modelu architektury musí být odstraněna deklarace p4 programovatelného bloku `egress_pipeline`.

Implementace

Za účelem úpravy metadat bylo nutné upravit model architektury *v1model.p4*, v němž se nachází definice metadat pro simulační model. Původní deklarace byla nahrazena následující strukturou metadat.

```
@metadata @name("standard_metadata")
    struct standard_metadata_t {
    @alias("intrinsic_metadata.ingress_timestamp")bit<48> ingress_timestamp;
    @alias("intrinsic_metadata.ingress_port") bit<9> ingress_port;
    @alias("intrinsic_metadata.egress_port")bit<9> egress_port;
    @alias("intrinsic_metadata.packet_len")bit<32> packet_len;
    @alias("intrinsic_metadata.hash")bit<4> hash;
    @alias("intrinsic_metadata.user16")bit<16> user16;
    @alias("intrinsic_metadata.user4")bit<4> user4;
}
```

Metadata musí být definována jako *Standard_metadata*, aby byla přijata P4 překladačem. U těchto metadat je však možné vytvořit alias, takže v simulačním modelu a externích třídách k těmto metadatům lze přistupovat pomocí vytvořeného aliasu.

V konstruktoru simulačního modelu musela být upravena deklarace metadat, se kterými model pracuje. Původní deklarace byla nahrazena tímto kódem:

```
add_required_field("intrinsic_metadata", "ingress_timestamp");
add_required_field("intrinsic_metadata", "ingress_port");
add_required_field("intrinsic_metadata", "egress_port");
add_required_field("intrinsic_metadata", "packet_len");
add_required_field("intrinsic_metadata", "hash");
add_required_field("intrinsic_metadata", "user16");
add_required_field("intrinsic_metadata", "user4");
force_arith_header("intrinsic_metadata");
```

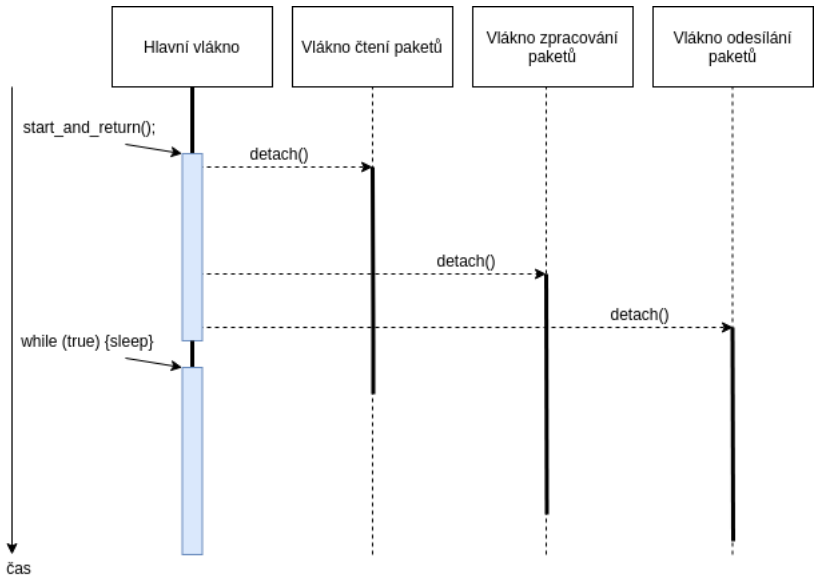
V kódu simulačního modelu *Simple switch* se pracuje s metadaty pro vstupní časovou značku, vstupní port, výstupní port i délku paketu. Ve všech částech kódu, které s těmito metadaty pracují byl pouze upraven jejich název tak, aby odpovídal nové deklaraci. Pro všechna ostatní metadata, byla práce s nimi odstraněna. Pro nová metadata *hash*, *user16* a *user4* nemusely být v simulačním modelu provedeny žádné změny.

Pro překlad je využíván backend P4 překladače pro simulační model *Simple switch*. Tento překladač vyžaduje deklaraci funkčního bloku *Egress pipeline* v modelu architektury *v1model* a jeho definici v P4 programu pro simulační model. Odstranění nutnosti deklarace funkčního bloku by znamenalo vytvořit nový backend P4 překladače pro simulační model *Smart NIC*. I když v1model a P4 programy pro simulační model obsahují funkční blok *Egress pipeline*, tak se simulační model chová jako by tento funkční blok neexistoval. Vytvoření backendu je možností budoucího rozšíření projektu.

3.3 Návrh a implementace ukončování programu po zpracování všech paketů

Návrh

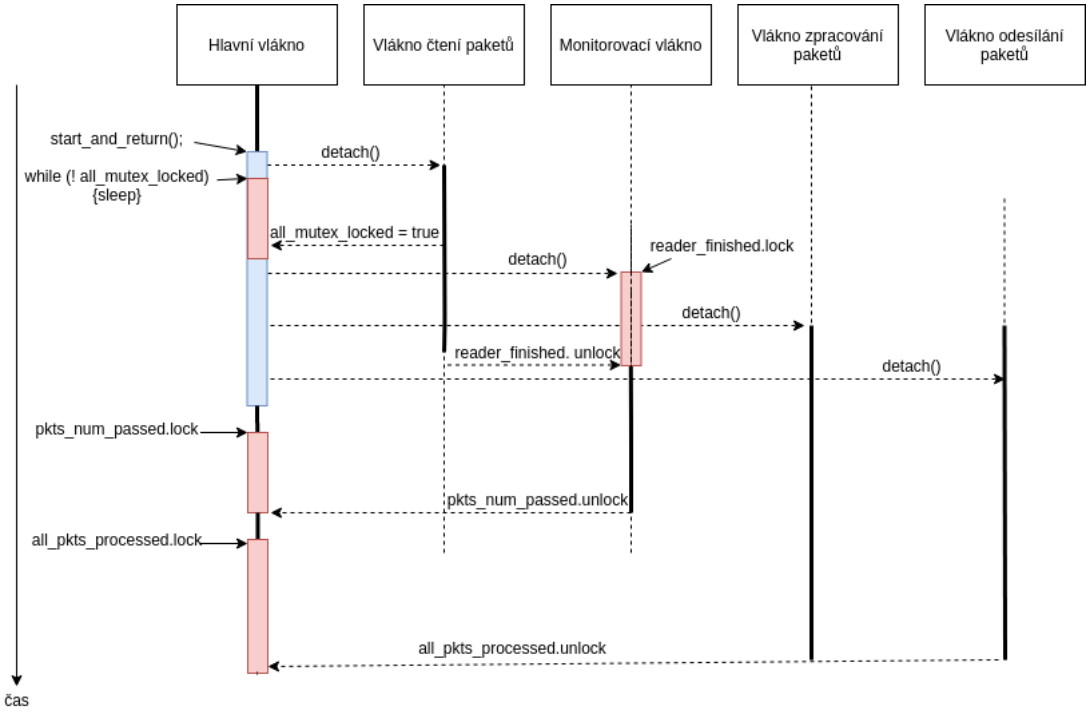
Současná implementace simulačního modelu *Simple switch* nepodporuje okamžité ukončení programu. Místo toho je na konci programu hlavního vlákna příkaz `while(true) std::this_thread::sleep_for(std::chrono::seconds(100))`, což je nekonečná smyčka, ve které je hlavní vlákno vždy uspáno na sto sekund. Toto řešení není vhodné, jelikož je nutné program ukončit manuálně například pomocí signálu `SIGINT` (Ctrl+C). Pro účely verifikace je tedy vhodnější automatické ukončení programu po zpracování všech paketů ze vstupních PCAP souborů. Obrázek 3.3 je diagramem znázorňujícím postupné spouštění vláken v původním simulačním modelu. Tučné čáry v tomto diagramu znázorňují činnost vlákna. Modré bloky znázorňují metody, nebo cykly a dobu jejich trvání.



Obrázek 3.3: Diagram spouštění vláken

Ukončování programu po zpracování všech paketů bude probíhat porovnáním celkového počtu paketů ke zpracování s aktuálním počtem zpracovaných paketů. Počet aktuálně zpracovaných paketů bude získáván ve vláknech pro zpracování a odeslání paketů. Celkový počet paketů ke zpracování bude získán ve vlákně pro čtení paketů. Tato hodnota musí být předána přes hlavní vlákno do vláken pro zpracování a odeslání paketů. Aby bylo zaručeno předání správné hodnoty mezi vlákny, musí být vyřešena synchronizace. Té bude dosaženo pomocí semaforů. Všechny semafory budou při inicializaci uzamčeny. První semafor bude znázorňovat dokončení čtení všech vstupních paketů. Ve chvíli, kdy vlákno čtení paketů dokončí svoji činnost, bude tento semafor odemčen a hodnota počtu paketů na vstupu může být předána. Druhý semafor bude znázorňovat, že může být předána hodnota počtu paketů na vstupu hlavnímu vláknu. Tento semafor bude zamčen v hlavním vlákně ihned po volání metody `start_and_return()`. Jeho odemčení znamená, že hlavní vlákno může získat počet paketů na vstupu. Tato hodnota je předána vláknům pro zpracování a odeslání paketů. Poslední semafor značí, že byly zpracovány všechny pakety. Ten je uzamčen v hlavním vlákně

a odemčen je po zpracování všech paketů. To značí, že může být volán destruktor třídy simulačního modelu a program může být ukončen. Návrhem tohoto řešení je obrázek 3.4



Obrázek 3.4: Synchronizace vláken

Implementace

V rámci implementace musely být upravovány třídy projektu BMV2. Za tímto účelem byly vytvořeny nové třídy pro všechny třídy, ve kterých bylo nutné přidat kód pro zjištění počtu paketů ke zpracování. Pro tyto nové třídy bylo využito dědění od těch původních. V těchto nových třídách byly přepsány všechny potřebné metody. Dále do těchto nových tříd musely být přidány všechny *private* a *protected* prvky z tříd původních. Jména těchto upravených tříd byla vytvořena přidáním slova Mod za původní název. Například pro třídu `DevMgrIface` byla třída s úpravami pojmenována `DevMgrIfaceMod`. Pro tyto třídy byly vytvořeny nové soubory, jejichž jména se vytvářela stejně, jako názvy tříd. Tyto nové soubory musely být přidány do souboru `/behavioral-model/targets/smart_NIC/Makefile.am` do proměnné `libsmartnic_la_SOURCES` kvůli jejich následnému překladu.

Pro ukončování programu musí třída simulačního modelu implementovat destruktor ukončující činnost vláken pro zpracování a odesílání paketů. Dále musí mít program přehled o celkovém počtu paketů ve vstupních PCAP souborech a o aktuálním počtu již zpracovaných paketů. Aktuální počet již zpracovaných paketů je zjišťován ve třídě `SimpleSwitch`, do které pro tento účel byla přidána proměnná `out_pkts_num`. Zpracováním je myšleno zahození paketu, nebo jeho odeslání. Tyto události probíhají ve dvou částech kódu. Zahození je možné na konci vlákna `ingress_thread` při splnění podmínky `egress_port == 511`, což je číslo portu rezervované pro pakety, které mají být zahozeny. Výše zmíněná proměnná je v

této podmínce inkrementována. Odesílání probíhá ve vlákně `transmit_thread`, do kterého je také přidána inkrementace proměnné.

Celkový počet paketů ve vstupních PCAP souborech je zjišťován v třídě `PcapFilesReaderMod`, která se nachází v souboru `pcap_file_mod.cpp`. Za tímto účelem je zde deklarována proměnná `in_pkts_num`. Počet paketů je získáván v metodě `scan()`. Tato metoda prochází všechny vstupní PCAP soubory a postupně odesílá pakety simulačnímu modelu.

Předání celkového počtu vstupních paketů třídě SimpleSwitch

Pro předání je nutné nejprve spustit vlákno pro čtení paketů. Toto vlákno nejprve uzamkne všechny semafory zmíněné v návrhu. Těmi jsou:

- `reader_finished_mutex`
- `pkts_num_passed_mutex`
- `all_pkts_processed_mutex`

Tyto semafory jsou součástí třídy `DevMgrIfaceMod`, jejíž instance existuje po celou dobu existence simulačního modelu. Nemá-li nějaká z tříd přístup k těmto semaforům, tak jim je předán ukazatel na tyto semafory. Po uzamknutí semaforů může hlavní vlákno spustit monitorovací vlákno, vlákno zpracování paketů a vlákno odesílání paketů. Po spuštění vláken hlavní vlákno uzamyká semafor `pkts_num_passed_mutex` a čeká, až bude moci získat počet paketů na vstupu. Mezitím monitorovací vlákno,které je součástí třídy `FilesDevMgrImp`, uzamyká semafor `reader_finished_mutex` a čeká na ukončení činnosti vlákna čtení paketů. Po dokončení čtení monitorovací vlákno předá hodnotu počtu paketů na vstupu třídě `DevMgrIfaceMod` a odemyká semafor `pkts_num_passed_mutex`. To umožní hlavnímu vláknu získat počet paketů na vstupu a předat ji třídě `SimpleSwitch`. V této fázi je uzamknut v hlavním vlákně semafor `all_pkts_processed_mutex`. Který je odemčen po zpracování všech paketů. Hned po odemčení je volán destruktor třídy `SimpleSwitch`, který ukončuje činnost simulačního modelu a následně je ukončen celý program.

Mimo výše zmíněných změn ve třídách `DevMgrIfaceMod`, `PcapFilesReaderMod` a `FilesDevMgrImpMod` musela být upravena i funkce `main` za účelem vytvoření vlastní instance třídy `DevMgrIfaceMod`.

```
bm::OptionsParser parser;
parser.parse(argc, argv, &simple_switch_parser);
//.....
simple_switch = new SimpleSwitch(drop_port, enable_swap_flag);
bool use_files_arg = parser.use_files;
DevMgrIfaceMod *ptr_to_my_dev_mgr = nullptr;
if(use_files_arg){
   ptr_to_my_dev_mgr = new FilesDevMgrImpMod(false,parser.wait_time);
}
int status;
if(use_files_arg) {
   status = simple_switch->init_from_options_parser(parser,nullptr,
            std::unique_ptr<DevMgrIfaceMod>(ptr_to_my_dev_mgr));
} else {
   status = simple_switch->init_from_options_parser(parser);
```

```
}
if (status != 0) std::exit(status);
// ... thrift ...
simple_switch->start_and_return();
if(use_files_arg){
   ptr_to_my_dev_mgr->pkts_num_passed_mutex.lock();
   simple_switch->set_in_pkts_num(ptr_to_my_dev_mgr->get_in_pkts_num());
   simple_switch->all_pkts_processed_mutex =
         &(ptr_to_my_dev_mgr->all_pkts_processed_mutex) ;
   simple_switch->~SimpleSwitch();
} else {

   while(true) std::this_thread::sleep_for(std::chrono::seconds(100));
}
```

Změny začínají získáním hodnoty parametru `--use-files` a deklarací ukazatele na vlastní implementaci třídy `DevMgrIface`. Poté, pokud je využit výše zmíněný parametr, je provedena instanciace výše zmíněné třídy `DevMgrIfaceMod` a inicializace simulačního modelu pomocí metody `init_from_options_parser()`, ve které je předán ukazatel na výše zmíněnou instanci. Pokud nebyl využit argument `--use-files`, probíhá inicializace a ukončení programu stejně, jako v modelu *Simple switch*. Následně je provedeno připojení na Thrift port a spuštění zpracování paketů. Pokud byl využit parametr `--use-files` je z instance třídy `DevMgrIfaceMod` získán počet paketů ke zpracování a následně předán simulačnímu modelu spolu s ukazatelem značícím zpracování všech paketů. Nakonec je volán destruktor, ve kterém je výše zmíněný semafor uzamknut. Po odemknutí semaforu je program ukončen.

3.4 Návrh a implementace externích tříd

Tvorba externích tříd musí splňovat pravidla popsaná v sekci 2.3. Byla-li vytvořena v novém souboru, tak cesta k tomuto souboru musí být přidána do souboru `/behavioral-model/-targets/smart_NIC/Makefile.am` a to konkrétně do proměnné `libsmartnic_la_SOURCES`, aby mohl být zdrojový kód třídy přeložen. Deklarace rozhraní tvořené třídy musí být dále vložena do modelu architektury *v1model.p4*. Vzor deklarace v modelu architektury je popsán v sekci 2.2. V této fázi je možné v P4 programu externí třídu využívat. Instanciace probíhá příkazem `KonstruktorTridy() promenna;`. Tato instanciace probíhá mimo jakýkoliv programovatelný blok. Následně lze v kontrolním bloku `Ingress` volat její metody příkazem `promnna.nazevMetody();`. Všechny výše popsané kroky lze provést i v P4 programech pro simulační model *Simple switch*. P4 program využívající externí třídy musí být překládán s argumentem `--emit-externs`.

Chce-li programátor využít externí třídu přímo v kódu simulačního modelu, neexistuje jednoduché řešení instanciace externí třídy. Nejprve musí být vytvořena externí instance, která je uložena do proměnné `std::unordered_map<std::string, ExternFactoryFn> factory_map`. Následně musí být registrovány její atributy pomocí metody `_register-_attributes()`. Tato metoda přidá všechny registrované atributy do proměnné `std::unordered_map<std::string, void *> attributes`. Poté tyto registrované atributy musí být nastavené na požadované hodnoty. Následně je volána inicializační funkce vytvořená v implementaci externí třídy a nakonec je externí instance převedena na instanci konkrétní externí třídy. Za tímto účelem byla vytvořena následující metoda.

```
template<typename T>
T ExternRunner::externInit(string externClass,
                  map<string, ArgumentVal> atributy){

    auto externInstance = ExternFactoryMap::get_instance()
                      ->get_extern_instance(string(externClass));

    externInstance->_register_attributes();

    for (std::map<string, ArgumentVal>::iterator it = atributy.begin();
                      it != atributy.end(); ++it){

        if(!it->second.oneValueSet()){
            std::cout << "Nastaveno vice hodnot \n";
            return nullptr;
        }
        if(!externInstance->_has_attribute(it->first)){
            std::cout << "Atribut v externi tride neexistuje \n";
            return nullptr;
        }
        if(it->second.getDataSet()){
            externInstance->_set_attribute<Data>(it->first,
                                        it->second.getDataVal());
        } else if (it->second.getStringSet()){
            externInstance->_set_attribute<std::string>(it->first,
                                std::string(it->second.getStringVal()));
        }
    }
    externInstance->init();
    T specificInstance = dynamic_cast <T> (externInstance.get());
    return specificInstance;
}
```

Pro registrování atributů různých datových typů byla vytvořena třída `ArgumentVal`, ve které uživatel nastaví proměnnou požadovaného datového typu. V metodě, jejíž kód se nachází výše je určeno, jaký datový typ byl použit a podle toho je volán příslušný příkaz `_set_attribute`. Třída `ArgumentVal` a inicializační metoda prozatím umožňují registrování atributu datových typů `Data` a `string` avšak umožňují jednoduché rozšíření o další atributy.

Pro přidávání nových externích tříd bude vytvořena nová složka. Volání metod sloužících k připojení souboru s externí metodou k simulačnímu modelu bude vloženo do jedné metody, která bude volána v konstruktoru simulačního modelu. Jako vzorové externí třídy budou vytvořeny čítač a třída pro výpočet kontrolního součtu TCP protokolu.

Externí třída pro výpočet kontrolního součtu TCP protokolu

Jako příklad externí třídy třídy je implementována třída pro výpočet kontrolního součtu TCP protokolu [11, 7]. Kontrolní součet je počítán z TCP paketu, což je paket obsahující hlavičku TCP protokolu, hlavičku aplikační vrstvy TCP/IP a data paketu. Pro výpočet tohoto součtu je vytvořena takzvaná pseudo hlavička, která je vložena před TCP paket a

z tohoto nového paketu je počítán kontrolní součet. Jelikož TCP paket obsahuje i hodnotu kontrolního součtu, tak je tato hodnota vynulována. Samotný výpočet probíhá tak, že je paket rozdělen na šestnácti bitové části. Tyto části jsou postupně sčítány pomocí bitového součtu. Jsou-li všechny části paketu sečteny a výsledná hodnota má více než 16 bitů, je spodních 16 bitů této hodnoty přičteno k vrchním šestnácti bitům této hodnoty. V této šestnáctibitové hodnotě je každý bit znegován a výsledkem je konečný kontrolní součet. Pro ověření kontrolního součtu je vypočítán nový kontrolní součet výše popsaným způsobem a k němu je přičtena hodnota předtím získaného kontrolního součtu. Je-li výsledkem tohoto součtu číslo 0, tak paket došel nepoškozený.

Následující kód je deklarací rozhraní externí třídy pro výpočet TCP kontrolního součtu v modelu architektury.

```
extern L4ChecksumExtern {
       L4ChecksumExtern();
       void computeChecksum<T,H,N>(in T pseudoHdr, in H payloadPart,
                                   out N checksum);
       void verifyChecksum<T,H,N>(in T pseudoHdr, in H payloadPart,
                                   out N checksum);
}
```

Deklarace metody pro výpočet TCP kontrolního součtu v externí třídě je následující.

```
void computeChecksum(const Field &pseudoHdr, const Field &payloadPartP4,
                     Data &csum);
```

Metoda pro výpočet kontrolního součtu i metoda pro ověření kontrolního součtu mají stejné argumenty, kterými jsou:

- **const Field &pseudoHdr -** Ukazatel na pseudo hlavičku, reprezentovanou sekvencí bitů.
- **const Field &payloadPartP4 -** Ukazatel na sekvenci bitů reprezentující potřebné hlavičky, které byly extrahované v P4 programu. Tento argument je nutný. Třída `ExternType` umožňuje přístup k extrahovaným hlavičkám a jejím polím, avšak k jejich získání je nutné znát buď jejich pozici ve struktuře hlaviček a nebo jméno hlavičky. Jméno hlavičky a její pozice ve struktuře hlaviček však nejsou pevně dány, ale odvíjí se od jména a pozice definované v P4 souboru. Neexistuje tedy jednoznačný způsob, jak v kódu externí třídy získat potřebné extrahované hlavičky a z toho důvodu jsou předávány jako argument.
- **Data &csum -** - Ukazatel na proměnnou, do které bude uložen kontrolní součet.

Následující text popisuje postup výpočtu kontrolního součtu s důležitými částmi kódu. Pro výpočet kontrolního součtu jsou nejprve bitové sekvence ze vstupů převedeny na datový typ `std::string`. Následně je pomocí příkazu `packet = &get_packet()` získán ukazatel na aktuálně zpracovávaný paket. Tuto funkcionalitu poskytuje třída `ExternType`, od které externí třída dědí. Dalším krokem je získání neextrahované části paketu. To probíhá příkazy:

```
const char* payloadPartExternChar = packet->get_packet_buffer().start();
const std::string payloadPartExternStr(payloadPartExternChar,
                 packet->get_packet_buffer().get_data_size());
```

Tyto příkazy nejprve získají neextrahovanou část paketu, kterou následně převedou na datový typ `std::string`. Dalším krokem je výpočet délky TCP paketu, která není součástí žádné hlavičky paketu, ale musí být vypočtena. Toho je dosaženo sečtením délek proměnné obsahující extrahované hlavičky a proměnné obsahující neextrahované hlavičky a data paketu. Tato hodnota je následně převedena na číslo ve dvojkové soustavě a přidána k pseudo hlavičce. Následně jsou pseudo hlavička, extrahované hlavičky a neextrahovaná část paketu spojeny do jedné sekvence bitů uložené v proměnné `tcpPacket`. S touto sekvencí bitů je proveden samotný výpočet kontrolního součtu. K tomu slouží následující cyklus:

```
int checksum = 0;
while(!tcpPacket.empty()){
  std::bitset<16> csumPart(tcpPacket.substr(0,16));
  tcpPacket.erase(0,16);
  checksum+= csumPart.to_ulong();
}
```

Tento cyklus získá prvních 16 bitů ze složeného TCP paketu, ze kterého je zároveň vymaže. Hodnotu této šestnácti bitové části přičte k proměnné `checksum`, ve které bude po dokončení cyklu součet všech šestnácti bitových částí paketu. Cyklus končí ve chvíli, kdy je proměnná `tcpPacket` prázdná.

Posledním krokem je sečtení spodních 16 bitů a vrchních 16 bitů. Za tímto účelem je hodnota v proměnné `checksum` převedena na bitovou reprezentaci. Následně je získáno spodních 16 bitů a horních 16 bitů, které jsou k sobě přičteny. Ve výsledné hodnotě jsou všechny bity znegovány a tato hodnota je vložena do výstupního argumentu `csum`.

Kapitola 4

Testování

Ukončení po zpracování paketu

Pro snadnější testování korektnosti fungování simulačního modelu bylo nejprve implementováno a testováno ukončení programu po zpracování všech vstupních paketů. Byl spuštěn simulační model příkazem: `../simple_switch -i 0@lo -use-files 1 ../jsonFolder/-accept_all_test.json`. Funkcionalita P4 programu *accept_all_test* je taková, že každý vstupní paket přijme a odešle ho na stejné rozhraní, na kterém byl přijat. Výstupní PCAP soubor pro dané rozhraní tedy musel obsahovat stejný počet paketů jako vstupní PCAP soubor pro stejné rozhraní. Test byl úspěšný v případě, kdy se program ukončil sám a zároveň výstupní PCAP soubory obsahovaly stejný počet paketů, jako vstupní PCAP soubory. V tomto případě, byl obsah paketů irelevantní. Tento test byl prováděn nejprve s jedním rozhraním a následně s dvěma rozhraními, aby byla zaručena funkčnost pro více rozhraní. Vstupní PCAP soubory obsahovaly pět paketů nebo žádný paket. Tím byla otestována funkčnost i v případě prázdného PCAP souboru.

Testování probíhalo manuálně, jelikož skript na testování by musel ošetřit případ, kdy se simulační model vůbec neukončí. Manuální kontrola počtu paketů navíc není časově náročná, jelikož testování bylo prováděno s malým počtem paketů. Pro kontrolu počtu paketů byl využit program Wireshark.

Funkcionalita simulačního modelu

Pro testování funkcionality simulačního modelu byl vytvořen skript napsaný v jazyce Python 2.7. Tento skript obsahuje testy kontrolující, zda simulační model zpracovává pakety dle funkcionality testovacích P4 programů. Výstupní PCAP soubory každého z testů jsou ukládány do složky `/smart_NIC/pcapFolder/PCAP_test_results`. Dále byl vytvořen skript pro automatické generování testovacích PCAP souborů.

Generování PCAP souborů

Pro účely generování vstupních PCAP souborů byl vytvořen modul generator.py, naprogramovaný v jazyce Python 2.7. Tento modul využívá Scapy, což je program napsaný v jazyce Python, který umožňuje generování a odchytávání paketů, ale také umožňuje extrakci hlaviček a polí z těchto hlaviček. Pro generování jednoho paketu byla vytvořena funkce `generate_packet(srcaddr, dstaddr, ipVersion, out_pcap)`, která má jako ar-

gumenty zdrojovou IP adresu, cílovou IP adresu, verzi IP hlavičky a výstupní PCAP soubor. V této funkci je nejprve pomocí následujícího kódu vygenerován náhodný obsah paketu.

```
payload = ""
for i in xrange(PAYLOAD_LEN):
   tmp = chr(random.randint(33,126))
   payload += tmp
```

Ke generování slouží cyklus, který má počet opakování nastavený proměnnou `PAYLOAD_LEN`. V cyklu je příkazem `tmp = chr(random.randint(33,126))` vygenerován náhodný znak s ASCII hodnotou v daném rozsahu a ten je přidán do proměnné pro obsah paketu. Dalším krokem je nakonfigurování polí hlaviček paketu. Příkazem `eth = scapy.Ether()` je vytvořená proměnná reprezentující ethernetovou hlavičku a některá pole této hlavičky jsou nastavena. Obdobně je nastavena IP hlavička. Poté je nastaven buď TCP protokol, nebo UDP protokol. Který z protokolů bude nastaven je vybráno náhodně. Nakonec je nastaven DNS protokol. Celý paket je následně složen a zapsán do souboru pomocí příkazu `scapy.wrpcap(out_pcap, eth/ip/tcp_or_udp/payload ,append=True)`. Paket je přidán na PCAP konec souboru.

Další funkcí je `packet_generator(packet_count, version, out_pcap)`, jejíž argumenty jsou celkový počet paketů ke generování, verze IP protokolu a výstupní PCAP soubor. Tato funkce nejprve vymaže soubor se zadaným názvem, pokud již existuje. Tím je zaručeno, že se v souboru budou nacházet pouze pakety generované touto funkcí. Následně se v cyklu generují pakety pomocí výše popsané funkce.

Pro jednoduchost ověřování PCAP souborů je zdrojová IPv4 adresa vybrána z hodnot "10.0.2.15"a "10.0.2.20". Cílová IPv4 adresa je vždy ta z těchto hodnot, která nebyla vybrána jako zdrojová IPv4 adresa. Je-li místo IPv4 protokolu nastavován IPv6 protokol, je zdrojová IP adresa nastavena na hodnotu "2001::42"a cílová IP adresa na "2001::43". Zdrojová MAC adresa je generována náhodně a cílová MAC adresa je vždy '12:23:34:45:56:67'. Pole *Time to live* IP hlavičky je vždy nastaveno na hodnotu 42.

Testování funkcionality

Základní myšlenkou za testováním funkcionality simulačního modelu je porovnání výstupního PCAP souboru s očekávaným výsledkem, který je ovlivněn P4 programem a vstupním PCAP souborem. Spuštění simulačního modelu ve skriptu probíhá pomocí funkce `os.system('příkaz')`, která daný příkaz provede. Má-li být provedeno více příkazů paralelně je použit objekt `Pool`, který paralelní spouštění příkazů umožňuje. Pro čtení paketů z PCAP souborů a pro získávání informací z těchto paketů byl použit modul dpkt.

Skript obsahuje několik testů:

- Testy spuštění modelu s chybnými, nebo chybějícími parametry - U těchto testů je kontrolována návratová hodnota simulačního modelu. V každém z těchto testů musí být návratová hodnota 1.
- Test zahození všech paketů - V P4 programu je definováno, že má model všechny pakety zahodit. Po skončení běhu simulačního modelu je otevřen výstupní soubor a kontroluje se, zda je prázdný.

- Test přijmutí všech paketů - P4 program definuje, že každý paket bude přijat. Ve výstupním PCAP souboru se kontroluje, že je počet vstupních paketů roven počtu paketů ve vstupním PCAP souboru.

- Test přijmutí paketů s cílovou IP adresou 10.0.2.15 - P4 program a pravidlo v tabulce definuje, že jsou přijaty pouze pakety s cílovou IP adresou 10.0.2.15. Jejich zdrojová MAC adresa je nastavena na původní cílovou MAC adresu, cílová MAC adresa je nastavena na 42:42:42:42:42:42 a jejich Time to live je snížen na hodnotu 41. Po skončení běhu simulačního modelu jsou ve vstupním souboru spočítány pakety s cílovou IP adresou 10.0.2.15. Následně je u paketů ve výstupním souboru zkontrolováno, zda se v něm nacházejí pouze pakety s cílovou IP adresou 10.0.2.15 a jejich počet se rovná počtu paketů s touto IP adresou ve vstupním PCAP souboru. Dále jsou kontrolovány hodnoty všech změněných polí hlaviček.

- Test přijímání paketů na dvou rozhraních - Podobný případ jako v předchozím testu. Pakety s cílovou IP adresou 10.0.2.15 jsou odesílány přes rozhraní 0 a pakety s cílovou IP adresou 10.0.2.20 jsou odeslány přes rozhraní 1. Je zjištěn počet paketů pro každou z těchto IP adres v obou vstupních souborech. Následně jsou zkontrolovány výstupní soubory stejně jako v předchozím testu.

- Test externího čítače - V P4 kódu jsou instanciovány dva externí čítače. Jeden počítá TCP pakety a druhý počítá UDP pakety. Aktuální hodnota čítače je uložena do pole *Total Length* IPv4 hlavičky aktuálně zpracovávaného paketu. Po skončení běhu simulačního modelu je otevřen výstupní PCAP soubor. Dále jsou vytvořena počítadla TCP a UDP paketů. Pakety výstupního PCAP souboru jsou čteny a podle toho, zda se jedná o TCP nebo UDP paket, je inkrementováno příslušné počítadlo. Aktuální hodnota počítadla musí být totožná s hodnotou uloženou do pole *Total Length*. Stejný test je proveden znovu s nastavením inicializační hodnoty pro každý čítač.

- Test externí třídy pro výpočet TCP a UDP kontrolního součtu - Nejprve je testován samotný výpočet kontrolního součtu. Za tímto účelem je vytvořen P4 program `compute_chcecksum.p4`, ve kterém je kontrolní součet vypočítán a uložen do pole *checksum* hlavičky TCP nebo UDP. Následně jsou otevřeny vstupní PCAP soubor, ve kterém byly kontrolní součty vypočítány pomocí Scapy a výstupní PCAP soubor, ve kterém byly kontrolní součty vypočítány externí třídou. Kontrolní součet paketu ve vstupním PCAP souboru musí být roven kontrolnímu součtu paketu ve výstupním souboru.

  Dalším testem je výpočet kontrolního součtu s pakety obsahujícími IPv6 hlavičku a extrahovaný DNS protokol. Tento test probíhá stejně jako předchozí.

  Dále bylo testováno ověření kontrolního součtu. Výsledek hodnoty `verify_checksum` byl uložen do pole `checksum`. Hodnota metody `verify_checksum` je rovna nule v případě, kdy byl paket neporušen. ve výstupním PCAP souboru je tedy kontrolováno, zda je v každém paketu hodnota tohoto pole rovna 0.

  Dalším testem kontrolního součtu je případ, kdy byl paket porušen. Porušení paketu bylo simulováno změno pole výstupního portu v TCP hlavičce v P4 kódu. Ověření probíhá stejně jako v předchozím testu s tím rozdílem, že hodnota pole `checksum` musí být rozdílná od nuly.

Výsledkem testování bylo zjištění, že simulační model zpracovává pakety přesně podle funkcionality ve vytvořených P4 souborech. Testování probíhá nad jednoduchými P4 soubory. Pro kompletní představu o funkcionalitě simulačního modelu by bylo vhodné vytvořit rozšířenou sadu složitějších P4 souborů, nad kterými by byl simulační model testován.

4.1 Dosažené výsledky

V rámci bakalářské práce byl popsán projekt BMv2 a tvorba nových simulačních modelů v tomto projektu. Dále byl vytvořen nový simulační model implementující architekturu P4 jádra karet COMBO. Tento model byl vytvořen úpravami již existujícího modelu *Simple switch* a jeho modelu architektury *v1model*. Programovatelnými bloky architektury tohoto modelu jsou: *Parser*, *Ingress pipeline* a *Deparser*. Model pracuje s *instrinsic_metadaty* architektury P4 jádra karet COMBO. Model ukončí svoji činnost ihned po zpracování všech paketů. Pro model je implementován vzor externí třídy `ExternCounter`, která slouží jako čítač paketů. Další vzorovou externí třídou je třída pro výpočet TCP kontrolního součtu. Architektura simulačního modelu *Simple switch* umožňovala jednoduché volání uživatelem vytvořených externích objektů v P4 kódu. V případě, kdy by chtěl programátor využít externí třídu v kódu simulačního modelu, byla implementována funkce usnadňující jeho volání.

Práce by mohla pokračovat rozvojem podpory externích objektů v simulačním modelu a v projektu BMv2. V současné době architektura tohoto modelu neumožňuje volání metod externích objektů v programovatelných blocích *Parser* a *Deparser*. Dále projekt BMv2 neumožňuje registrovat externí třídu, jejíž konstruktor obsahuje argumenty, nebo inicializační hodnoty. Za účelem implementace těchto funkcionalit by bylo nutné upravit některé třídy projektu BMv2 a pravděpodobně i překladače pro tento model. S tím by souviselo hlubší prozkoumání částí kódu souvisejících s registrací externích tříd, zápisem externích instancí do JSON souboru a voláním metod za běhu programu. Další možnou úpravou je rozšíření podporovaných datových typů ve funkci pro instanciaci externí třídy v simulačním modelu. V současné době musí model architektury *v1model* a P4 program pro simulační model obsahovat P4 programovatelný blok *Egress pipeline*. Možným rozšířením práce je vytvoření backendu P4 překladače pro simulační model *Smart NIC*, tak aby model architektury a P4 program nemusel obsahovat tento blok.

Projekt BMv2 je P4 konsorciem upravován a jsou přidávány další funkcionality. Je tedy možné, že některé z těchto funkcionalit budou časem přidány.

Kapitola 5

Závěr

Cílem této práce bylo vytvoření simulačního modelu v projektu BMv2, který bude využívaný ve verifikačním prostředí pro vývoj digitálního obvodu, na kterém lze spouštět P4 programy. Tímto modelem bude ve verifikačním prostředí nahrazen dosavadní simulační model *Simple switch*.

Byla popsána teorie jazyka P4, projekt BMv2, vytváření nových simulačních modelů v tomto projektu a vytváření vlastních externích tříd pro simulační model. Jako výchozí simulační model pro implementaci nového modelu byl zvolen *Simple switch* a jeho model architektury *v1model*. Model architektury byl změněn tak, aby podporoval požadovaná *intrinsic_metadata*. Simulačnímu modelu byly odebrány funkcionality, které výsledný model nebude využívat. Dále byl přizpůsoben práci s novými metadaty. Některé třídy projektu BMv2 byly upraveny tak, aby umožňovaly ukončení simulačního modelu po zpracování všech paketů. Za tímto účelem byla změněna i funkce *main* vytvořeného simulačního modelu. Byly vytvořeny vzorová externí třídy čítače a TCP kontrolního součtu.

Testování probíhalo porovnáváním výstupních PCAP souborů s očekávanými výsledky. Za tímto účelem byl vytvořen skript provádějící toto testování. Tento skript testuje i funkcionalitu vzorové externí třídy pro čítač.

Práce by mohla pokračovat rozšířením podpory externích objektů v simulačním modelu a v projektu BMv2.

Literatura

[1] Bas, A. a Fingerhut, A. *Simple switch documentation* [online]. Revidováno 6.2.2020 [cit. 26.3.2020]. Dostupné z: https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md.

[2] Benáček, P. *Generation of High-Speed Network Device from High-Level Description* [online]. Praha, CZ, 2016. [cit. 24.12.2019]. Disertační práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Dostupné z: https://github.com/benycze/PhD-Thesis/blob/master/text/phd-benacek.pdf.

[3] Benáček, P. a Puš, V. *Jazyk P4 jako budoucnost sdn (dokončení)* [online]. root.cz, 27. leden 2016 [cit. 15.12.2020]. Dostupné z: https://www.root.cz/clanky/jazyk-p4-jako-budoucnost-sdn-dokonceni/.

[4] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N. et al. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*. červenec 2014, sv. 44, č. 3, s. 88–95. DOI: 10.1145/2656877.2656890.

[5] Free Software Foundation. *Automake manual* [online]. Verze 1.16.2. únor 2020 [cit. 13.3.2020]. Dostupné z: https://www.gnu.org/software/automake/manual/automake.

[6] Haleplidis, E., Pentikousis, K., Denazis, S., Salim, J. H., Meyer, D. et al. *Software-Defined Networking (SDN): Layers and Architecture Terminology* [online]. Leden 2015 [cit. 15.11.2019]. DOI: 10.17487/RFC7426. ISSN 2070-1721. Dostupné z: https://tools.ietf.org/html/rfc7426.

[7] Kozierok, C. M. *TCP Checksum Calculation and the TCP Pseudo Header* [online]. verze 3.0. Září 2005. Dostupné z: http://www.tcpipguide.com/free/t_TCPChecksumCalculationandtheTCPPseudoHeader-2.htm.

[8] Netcope technologies, a.s.. *Netcope P4 User Guide*. Verze 4.6. Netcope technologies, a.s., únor 2020 [cit. 25.4.2020].

[9] P4 Language Consortium. *Behavioral model v2* [online]. [cit. 11.11.2019]. Dostupné z: https://github.com/p4lang/behavioral-model.

[10] P4 Language Consortium. *P4 Language Specification* [online]. Verze 1.2.0. říjen 2019 [cit. 24.12.2019]. Dostupné z: https://p4.org/p4-spec/docs/P4-16-v1.2.0.html.

[11] Postel, J. *TRANSMISSION CONTROL PROTOCOL* [online]. Září 1981 [cit. 14.5.2020]. DOI: 10.17487/RFC0793. Dostupné z: https://tools.ietf.org/html/rfc793.

[12] SDxCentral. *What is OpenFlow* [online]. SDxCentral, 26. srpen 2013 [cit. 15.12.2020]. Dostupné z: https://www.sdxcentral.com/networking/sdn/definitions/what-is-openflow/.

[13] The P4.org Architecture Working Group. *Portable Switch Architecture* [online]. Verze 1.1.0. Listopad 2018 [cit. 3.3.2020]. Dostupné z: https://p4.org/p4-spec/docs/PSA-v1.1.0.html.

[14] Vaněk, P. *Cmake: zjednoduš si život* [online]. AbcLinuxu, 8. červen 2006 [cit. 20.4.2020]. Dostupné z: https://www.abclinuxu.cz/clanky/programovani/cmake-zjednodus-si-zivot.

Příloha A

Implementace modelu architektury

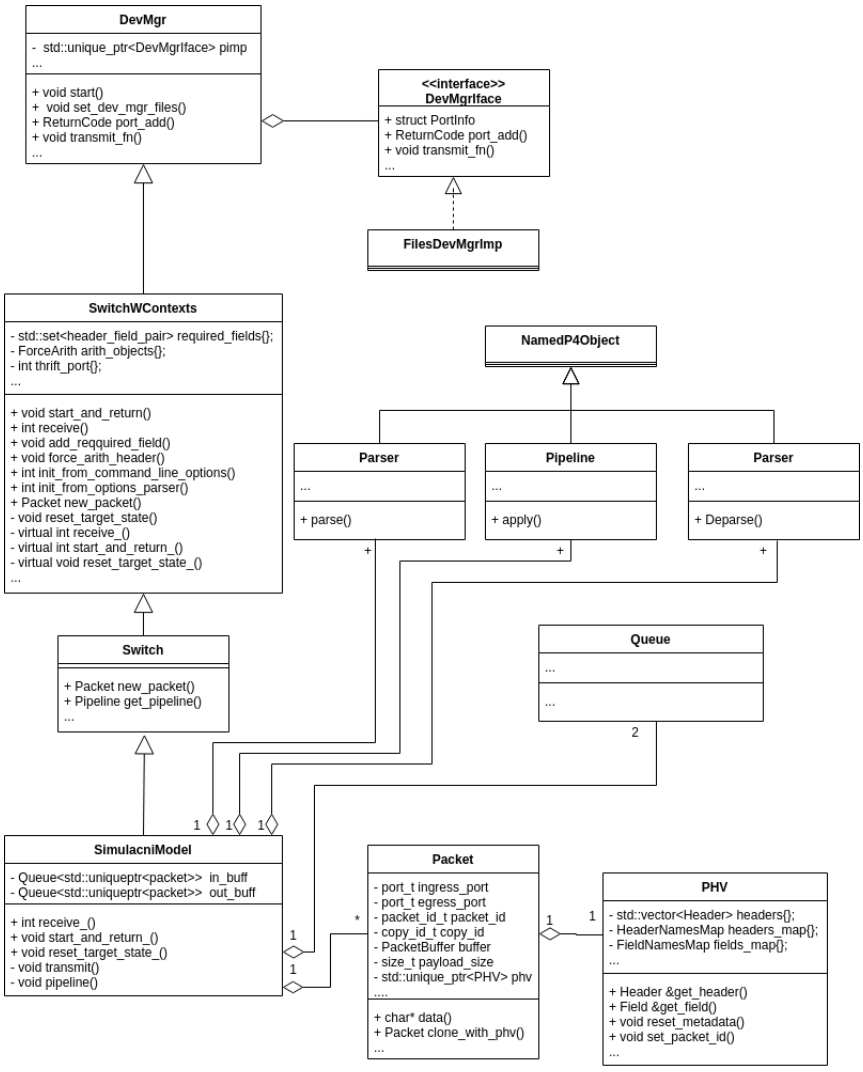
Následující kód je implementací modelu architektury 2.2. Kód je převzat z [10]

```
# include <core.p4> // Zakladni P4 knihovna
# include "very_simple_switch_model.p4" // Model architektury

typedef bit<48> EthernetAddress; // Deklarace typu pro MAC adresu
typedef bit<32> IPv4Address; // Deklarace typu pro IPv4 adresu

header Ethernet_h { // Deklarace ethernetove hlavicky a jejich poli
    EthernetAddress dstAddr;
    EthernetAddress srcAddr;
    bit<16> etherType;
}

header IPv4_h { // Deklarace Ipv4 hlavicky
    bit<4> version;
    //...
    IPv4Address dstAddr;
}


struct Parsed_packet { // Struktura extrahovanych hlavicek
    Ethernet_h ethernet;
    IPv4_h ip;
}

error { // Parse error
    IPv4OptionsNotSupported,
    IPv4IncorrectVersion,
    IPv4ChecksumError
}

parser TopParser(packet_in b, out Parsed_packet p) {
    state start {
        b.extract(p.ethernet);
```

```
        transition select(p.ethernet.etherType) {
            0x0800: parse_ipv4;
            // bez defaultniho pravidla -> jine pakety zahazuje
        }
    }
    state parse_ipv4 {
        b.extract(p.ip);
        // ...
        transition accept;
    }
}

control TopPipe(inout Parsed_packet headers,
                in error parseError,
                in InControl inCtrl,
                out OutControl outCtrl) {

      action Drop_action() {
        // kod akce
      }
      action Set_nhop(IPv4Address ipv4_dest, PortId port) {
        // kod akce
      }
     table ipv4_match {
         key = { // klic k vyhledavani }
         actions = {
            //seznam akci
         }
         size = 1024;
         default_action = Drop_action;
     }
     apply {
          // ...
          ipv4_match.apply()
          if (outCtrl.outputPort == DROP_PORT) return;
    }
}

control TopDeparser(inout Parsed_packet p, packet_out b) {
    apply {
        b.emit(p.ethernet);
        b.emit(p.ip);
    }
}

VSS(TopParser(), // instanciace balicku VSS
    TopPipe(),
    TopDeparser()) main;
```

Příloha B

Diagram tříd



Obrázek B.1: Diagram důležitých tříd

Příloha C

Příklad užití externí třídy pro výpočet TCP kontrolního součtu v P4 programu

Výpočet kontrolního součtu

Následující kód je příkladem vypočítání TCP kontrolního součtu v P4 programu.

```
//vytvoreni osmibitoveho nuloveho pole predavaneho v pseudo hlavicce
bit<8 > reserved = 00000000;

//vynulovani hodnoty kontrolniho souctu
hdr.tcp.checksum = 0000000000000000;

//vytvoreni sekvence bitu reprezentujicich pseudo hlavicku
bit<80> pseudoHdr = hdr.ipv4.srcAddr ++ hdr.ipv4.dstAddr
                  ++ reserved ++ hdr.ipv4.protocol;

//vytvoreni sekvence bitu reprezentujicich extrahovane hlavicky
bit<160> payload = hdr.tcp.srcPort++hdr.tcp.dstPort++hdr.tcp.seqNo
++hdr.tcp.ackNo++hdr.tcp.dataOffset++hdr.tcp.res++hdr.tcp.cwr
++hdr.tcp.ece++hdr.tcp.urg++hdr.tcp.ack++hdr.tcp.psh++hdr.tcp.rst
++hdr.tcp.syn++hdr.tcp.fin++hdr.tcp.window++hdr.tcp.checksum
++hdr.tcp.urgentPtr;

//promenna pro ulozeni kontrolniho souctu
bit<16> csum;

//vypocet kontrolniho souctu
L4csum.computeChecksum(pseudoHdr, payload, csum);

//vlozeni vypocitaneho kontrolniho souctu do pole v TCP hlavicce
hdr.tcp.checksum = csum;
```

Ověření kontrolního součtu

Následující kód je příkladem ověření TCP kontrolního součtu v P4 programu. V proměnné `csum` se bude nacházet číslo 0 v případě, kdy bylo ověření kontrolního součtu úspěšné. V opačném případě se v ní bude nacházet číslo rozdílné od nuly.

```
//vytvoreni osmibitoveho nuloveho pole predavaneho v pseudo hlavicce
bit<8 > reserved = 00000000;

//vytvoreni sekvence bitu reprezentujicich pseudo hlavicku
bit<80> pseudoHdr = hdr.ipv4.srcAddr ++ hdr.ipv4.dstAddr
                    ++ reserved ++ hdr.ipv4.protocol;

//vytvoreni sekvence bitu reprezentujicich extrahovane hlavicky
bit<160> payload = hdr.tcp.srcPort++hdr.tcp.dstPort++hdr.tcp.seqNo
++hdr.tcp.ackNo++hdr.tcp.dataOffset++hdr.tcp.res++hdr.tcp.cwr
++hdr.tcp.ece++hdr.tcp.urg++hdr.tcp.ack++hdr.tcp.psh++hdr.tcp.rst
++hdr.tcp.syn++hdr.tcp.fin++hdr.tcp.window++hdr.tcp.checksum
++hdr.tcp.urgentPtr;

//promenna pro vysledku overeni kontrolniho souctu
bit<16> csum;

//overeni kontrolniho souctu
L4csum.verifyChecksum(pseudoHdr, payloadVerify, csum);
```

Příloha D

Spuštění simulačního modelu, překlad P4 souborů a spuštění testů

Spuštění simulačního modelu

Pro práci se simulačním modelem je nejprve nutné nainstalovat potřebné balíčky.[1] Dalším krokem je stažení repozitáře projektu BMv2.[2]. Za účelem spuštění simulačního modelu *Smart NIC* je nutné zkopírovat vše z přiložené složky `behavioral-model` do složky `behavioral_model` stažené z repozitáře. Všechny dotazy na přepsání existujících souborů je nutné potvrdit. Nyní lze celý projekt přeložit pomocí následujících příkazů volaných ze složky `behavioral-model`.

1. ./autogen.sh
2. ./configure
3. make

V této fázi je projekt přeložen i se simulačním modelem *Smart NIC* a je možné využívat všechny simulační modely nacházející se ve složce `/behavioral-model/targets`. Simulační model *Smart NIC* se nachází ve složce `/behavioral-model/targets/smart_NIC`. Tato složka obsahuje několik důležitých složek a to `./p4FilesFolder`, která je určena pro vkládání P4 programů, pro tento simulační model. Další složkou je `./jsonFolder`, která je určena pro ukládání přeložených P4 souborů. Poslední složkou je `./pcapFolder`, která je určena pro vkládání vstupních a výstupních PCAP souborů a ze které je simulační model spouštěn, jelikož neumožňuje nastavení cesty k PCAP souborům.

Model je spouštěn ze složky `./pcapFolder` pomocí příkazu:

```
../simple_switch -i 0@<iface0> --use-files <time>
   ../jsonFolder/<název_json_souboru>
```

Místo *<iface0>* je nutné doplnit název rozhraní a je důležité, aby ve složce `./pcapFolder`

[1] Seznam potřebných balíčků lze nalézt na adrese https://github.com/p4lang/behavioral-model/blob/master/README.md

[2] Repozitář projektu BMV2 je dostupný na adrese https://github.com/p4lang/behavioral-model

byly umístěny soubory *<iface0>_in.pcap* a <iface0>_out.pcap. Rozhraní může být přidán libovolný počet. Místo argumentu *<time>* musí být vloženo celé číslo určující počet sekund, po které simulační model čeká, než začne zpracovávat pakety. Během této čekací lze z jiného terminálu ze složky `/behavioral-model/targets/smart_NIC` pomocí příkazu `./runtime_CLI` naplnit tabulky simulačního modelu. Úplný postup plnění tabulek je uveden v sekci 2.3. Nakonec je místo *<nazev_json_souboru>* zvolen název JSON souboru, kterým bude simulační model nakonfigurován. Lze použít příznak `--log-console` pro vypisování všech akcí, které se aktuálně provádí s paketem, do terminálu.

Překlad P4 souborů

Pro překlad P4 souborů je nutné nainstalovat potřebné balíčky[3]. Dalším krokem je stažení repozitáře[4]. Dále je nutné přeložit a nainstalovat překladač. Ze složky `/p4c` je nutné provést následující příkazy:

1. mkdir build
2. cd build
3. cmake ..
4. make
5. make check
6. sudo make -install

Nyní je možné překládat P4 soubory. Pro simulační model *smart NIC* je využit překladač `p4c-bm2-ss`. Příkladem jeho použití je následující příkaz spuštěný ze složky simulačního modelu `smart_NIC`:

```
p4c-bm2-ss -o ./jsonFolder/<název_json_souboru>
        ./p4FilesFolder/<název_p4_souboru> -emit-externs
```

Spuštění testů

Testovací skript se nazývá `test.py` a nachází se ve slože `./smart_NIC/pcapFiles`. Lze ho spustit z příkazové řádky příkazem `sudo ./test.py` ve výše zmíněné složce. Tato složka obsahuje složku `PCAP_test_results`, ve které se nachází výstupní PCAP soubory ze všech testů.

[3] Potřebné balíčky lze nalézt na adrese https://github.com/p4lang/p4c/blob/master/README.md v sekci *Dependencies*

[4] Repozitář lze nalézt na adrese https://github.com/p4lang/p4c

Příloha E

Obsah přiloženého CD

Přiložené CD obsahuje následující složky:

- **/behavioral-model** - Složka obsahující zdrojové kódy programu. Návod na instalaci a spuštění se nachází v sekci D.
- **/latex_src** - Složka obsahující zdrojové kódy pro vytvoření pdf souboru.
- **/pdf** - Složka obsahující pdf soubor s textem bakalářské práce.