VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

GRAF ŘÍZENÍ TOKU PROGRAMŮ V JAZYCE P4

CONTROL FLOW GRAPH FOR P4 PROGRAMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **TIMOTEJ PONEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. LUKÁŠ KEKELY, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Student:	**Ponek Timotej**
Program:	Informační technologie
Název:	**Graf řízení toku programů v jazyce P4** **Control Flow Graph for P4 Programs**
Kategorie:	Překladače

Zadání:

1. Seznamte se s jazykem P4 pro popis zpracování síťových paketů.
2. Nastudujte dostupný open-source překladač jazyka P4.
3. Navrhněte mid-end průchod překladače, který sestaví Control Flow graf pro daný P4 program.
4. Implementujte navržený průchod a sestavování grafu.
5. Konzultujte dosažené výsledky s vývojáři P4 kompilátorů a na základě zpětné vazby navrhněte možná zlepšení.

Literatura:

- Dle pokynů vedoucího a konzultanta.
- https://p4.org/
- https://github.com/p4lang/p4c

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

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

Vedoucí práce:	**Kekely Lukáš, Ing., Ph.D.**
Konzultant:	Puš Viktor, Ing., INTEL
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1. listopadu 2021
Datum odevzdání:	11. května 2022
Datum schválení:	29. října 2021

Abstrakt

Koncept SDN sa postupne stal jedným z najpopulárnejších riešení správy sietí. Umožňuje rýchlu rekonfigurovateľnosť sieťových zariadení tak, aby odrážala aktuálne požiadavky a taktiež umožňovala rýchle testovanie nových riešení. Týmto podporuje pokrok v sieťovej oblasti. Táto práca sa venuje jazyku P4, ktorý je jednou z implementácií konceptu SDN. Prínosom práce je zlepšenie časti existujúceho open-source prekladača jazyka P4, ktorá slúži na generovanie grafov riadenia toku programu. Nová implementácia zachytáva tok programu aj vo vnútri tabuliek a akcií, čo umožňuje ľahšie kontrolovať výstup prekladu a ďalej optimalizovať prekladač pre potreby redukcie mŕtveho kódu. Taktiež poskytuje možnosť generovať fullgraf vo formáte dot a preddefinovanom json formáte, ktorý zachytáva tok programu naprieč všetkými funkčnými blokmi daného P4 programu.

Abstract

Concept of SDN gradually became one of the most popular solutions for network management. It allows rapid reconfigurability of network devices, to reflect actual demands and to enable quick testing of new solutions, which supports overall advance in networking. This thesis focuses on P4 programming language, which is one of the implementations of SDN concept. The goal of this thesis is improvement of the existing open-source P4 compiler. More precisely, extension of a part used to generate control flow graphs. New implementation captures program flow even inside match-action tables and actions, which allows easier checking of compiler output and further optimization of compiler to reduce dead code. It also provides option to generate a fullgraph in dot format and a predefined json format that captures the flow of the program across all function blocks of the P4 program.

Kľúčové slová

SDN, OpenFlow, P4, p4c, prekladač, návštevník, graf riadenia toku, vyhľadávacia tabuľka, dot, fullgraf

Keywords

SDN, OpenFlow, P4, p4c, compiler, visitor, control flow graph, match-action table, dot, fullgraph

Citácia

PONEK, Timotej. *Graf řízení toku programů v jazyce P4*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Lukáš Kekely, Ph.D.

Graf řízení toku programů v jazyce P4

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Lukáša Kekelyho Ph.D. Ďalšie informácie mi poskytli Ing. Viktor Puš Ph.D. a Ing. Denis Matoušek. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Timotej Ponek
8. mája 2022

Poďakovanie

Chcel by som poďakovať vedúcemu mojej práce, Ing. Lukášovi Kekelymu Ph.D., za smerovanie a rady, ktoré mi pomohli pri písaní práce. Tiež by som chcel poďakovať Ing. Viktorovi Pušovi Ph.D a Ing. Denisovi Matouškovi, ktorí mi poskytli spätnú väzbu k návrhu vylepšenia riešenia a svojimi pripomienkami mi pomohli dotiahnuť implementáciu do výslednej podoby.

Obsah

Kapitola 1

Úvod

V dnešnej dobe internetu je dôležité, aby bolo pripojenie k sieti čo najdostupnejšie. Dostupnosť sieťového pripojenia môžu ovplyvňovať rôzne faktory, medzi nimi aj rýchlosť a efektivita spracovania paketov. Pre zlepšovanie týchto aspektov je dôležité stále posúvať technológie dopredu, zlepšovaním či už hardvéru alebo softvéru zariadení, skrz ktoré pakety putujú. Jedným zo spôsobov, ako zvýšiť efektivitu sieťových zariadení, je pomocou programovania ich dátovej vrstvy, skrz na tento účel špecializované jazyky. Medzi tieto jazyky patrí aj jazyk P4.

Programy vytvorené pre sieťové zariadenia pomocou jazyka P4, ako aj za pomoci iných jazykov, musia byť čo najviac optimalizované, aby zbytočne neplytvali obmedzenými hardvérovými prostriedkami zariadení. Kladie sa dôraz na nízku pamäťovú náročnosť a vysokú výpočetnú efektivitu programu. Prekladače jazykov, určených na programovanie sieťových zariadení, musia brať do úvahy tieto požiadavky, a preto sú neustále vyvíjané za účelom zefektívnenia zostavovaného programu a optimalizácie prekladu.

Na optimalizáciu prekladu sa používajú rôzne nástroje a techniky, ako eliminácia smyčiek, skladanie a propagácia konštánt alebo eliminácia mŕtveho kódu. Pre detekciu mŕtveho kódu je dôležité analyzovať tok programu, aby sa našli vetvy programu, z ktorých sa tok nedostane ďalej. V týchto bodoch sa program zasekne a nie je schopný ďalej pokračovať vo svojej činnosti, čo je nechcené. Za účelom analýzy toku programu sa vytvára graf toku programu.

Cieľom tejto práce je zlepšenie nástroja, ktorý generuje grafy toku programu pre programy napísané v jazyku P4. V kapitole 2 je vysvetlený koncept SDN, ktorého hlavnou myšlienkou je rozdelenie vrstiev v sieťových zariadeniach na riadiacu a dátovú. Ďalej je priblížený protokol OpenFlow, prvá reálna implementácia konceptu SDN. Kapitola 3 sa zaoberá ďalšou revolučnou technológiou v oblasti SDN, jazykom P4. Tento jazyk rozšíril koncept OpenFlow o plné programovanie dátovej vrstvy. V kapitole 4 je priblížený graf riadenia toku programu a jeho využitie v prekladačoch programovacích jazykov. V nasledujúcej kapitole 5 sa venujem aktuálnemu riešeniu prítomnom v repozitári open source prekladača p4c, ktoré generuje grafy riadenia toku programu v jazyku P4, vysvetľujem čo nezachytáva a ako ho je potrebné vylepšiť. Implementácia v kapitole 6 popisuje princípy ako funguje kód pridaný v mojom rozšírení prekladača p4c. V kapitole 7 porovnávam výstupy mnou implementovaného riešenia voči pôvodnému a popisujem úpravy vykonané pred pull-requestom do oficiálneho repozitára prekladača p4c. V závere (kapitola 8) sú zhrnuté celky, ktoré som musel naštudovať počas písania práce a zhodnotené jej prínosy. Popísané je tiež prijatie vytvorených výsledkov komunitou okolo prekladača P4.

Kapitola 2

Koncept SDN a OpenFlow protokol

2.1 Štandardný prístup

Bežne sú sieťové zariadenia určené len na špecifickú funkciu, ako prepínanie, smerovanie alebo analýza paketu [13]. Pracujú na základe predom poskytnutého softvéru od výrobcu. Riadiaca vrstva spolupracuje s dátovou vrstvou, pre efektívnosť výroby tieto vrstvy nie sú špecificky oddelené, a nie sú zvonka nijako prístupné/ovládateľné pre správcov sieťových zariadení.

Riadiaca vrstva (control plane) získava informácie o sieťovej topológií skrz riadiace sieťové protokoly a poskytuje potrebné informácie dátovej vrstve na vytvorenie vyhľadávacích tabuliek.

Úlohou **Dátovej vrstvy** (data plane / forwarding devices) je rýchle preposielanie paketov. Pakety preposiela na základe preddefinovaných pravidiel a tiež na základe dát získaných skrz riadiacu vrstvu. Z dát získaných od riadiacej vrstvy si dátová vrstva dopĺňa záznamy vo vyhľadávacích tabuľkách. Obsahuje špecifické integrované obvody (ASIC – Application specific integrated circuits), ktoré umožňujú dátovej vrstve rýchlo analyzovať hlavičky paketov a previesť ďalšie nutné operácie pre správne spracovanie paketu. Po spracovaní je paket presmerovaný na odpovedajúce zariadenie, prípadne zahodený.

Dáta sa v sieti neprenášajú ako jeden prúd bitov, ale rozdelené na menšie časti – do paketov. Paket je teda dátová jednotka prenáša sieťou. Na základe ISO/OSI[16] modelu sa sieť skladá zo siedmich vrstiev: **Fyzická** (L1), **Linková** (L2), **Sieťová** (L3), **Transportná** (L4), **Relačná** (L5), **Prezentačná** (L6) a **Aplikačná** vrstva (L7). ISO/OSI model je referenčným modelom, z ktorého vychádza TCP/IP model. Pre tvorbu a prenos paketov sú najdôležitejšie vrstvy L2, L3, L4 a L7. V každej z predošle spomínaných vrstiev sa paket skladá z dvoch častí: hlavička a telo. Hlavička obsahuje informácie relevantné pre aktuálnu vrstvu, zatiaľ čo telo pozostáva z dát, čo zväčša znamená celý paket z predošlej vrstvy. Každá vrstva berie informáciu, ktorú dostane z vrstvy nad ňou ako dáta, a pridáva k nej vlastnú hlavičku. Tento proces pridávania novej hlavičky k dátam z predošlej vrstvy sa nazýva zapuzdrenie (enkapsulácia).

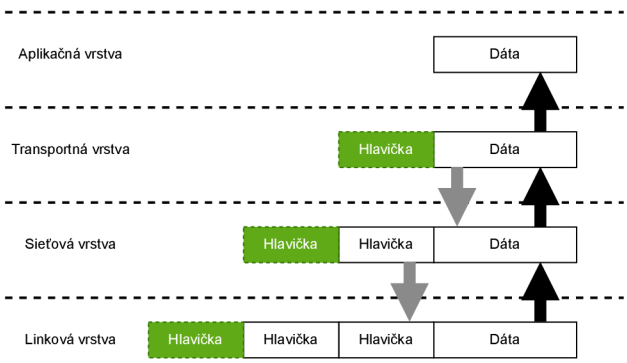
Skladanie paketu na jednotlivých vrstvách TCP/IP modelu [2] (obrázok 2.1):

1. **Aplikačná vrstva** (L7) – dáta určené pre konkrétnu aplikáciu

2. **Transportná vrstva** (L4) – pridáva k dátam TCP alebo UDP hlavičku, ktorá obsahuje zdrojový a cieľový port (UDP+TCP), pole s príznakmi (TCP), sekvenčné číslo (TCP) a ďalšie položky

3. **Sieťová vrstva** (L3) – zdrojová IP adresa, cieľová IP adresa, typ IP protokolu, IP voľby

4. **Linková vrstva** (L2) – obsahuje informácie o tom, o aký typ paketu ide, MAC adresa odosielateľa, MAC adresa príjemcu



Obr. 2.1: Skladanie paketu na jednotlivých vrstvách

Z laického pohľadu môžeme povedať, že pakety, ktorých adresátom je dané sieťové zariadenie, spracováva riadiaca vrstva. Pakety, ktorých adresátom nie je dané sieťové zariadenie, spracováva dátová vrstva, ktorá ich ďalej preposiela.

Keďže sieťové zariadenia nie sú nijako ovládateľné zvonka alebo programovateľné, možnosti meniť ich pravidlá na spracovanie paketov pre implementáciu nových riešení sú obmedzené. Nemožnosť meniť funkcionalitu sieťových zariadení po ich nasadení do siete viedla ku koncepcii myšlienky SDN, ktorej sa venujem v ďalšej sekcii.

2.2 SDN

Definícia SDN od ONF (Open Networking Foundation), čo je nezisková organizácia zameraná na vývoj, štandardizáciu a rozširovanie myšlienky SDN, znie:

> *Software-Defined Networking (SDN) je nová sieťová architektúra, kde je sieťové riadenie (riadiaca vrstva) oddelené od preposielania (dátová vrstva) a je priamo programovateľné.* [5]

Cieľom SDN je logické oddelenie riadiacej vrstvy od dátovej vrstvy, ako je možné vidieť na obrázku 2.3. Obrázok 2.2 zas načrtá rozdelenie jednotlivých vrstiev tak, ako to je v bežných sieťach. Výsledkom oddelenia riadiacej vrstvy od dátovej vrstvy je možnosť spravovať sieťové zariadenia centralizovane, skrz SDN kontrolér. Medzi hlavné benefity, ktoré SDN ponúka, patria [15]:

- **Zjednodušenie konfigurácie** – Bežne, vďaka rozdielnosti konfiguračných rozhraní sieťových zariadení a závislosti týchto rozhraní na výrobcovi, je potrebné zariadenia

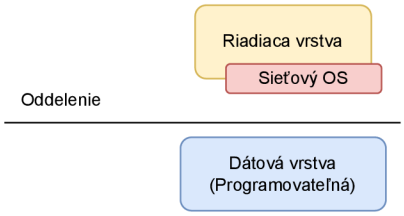
konfigurovať manuálne. Táto manuálna konfigurácia je prácna a náchylná na chyby. SDN ponúka riešenie. Skrz zjednotenie riadiacej vrstvy naprieč rôznymi sieťovými zariadeniami, je možné zariadenia konfigurovať naraz, pomocou SDN kontroléra. Vďaka zjednotenej konfigurácií cez SDN kontrolér je ďalej možné sieťové zariadenia dynamicky konfigurovať a optimalizovať na základe aktuálneho stavu/statusu siete.

- **Zvýšenie výkonnosti** – Jedným z neustálych cieľov sieťových optimalizácií je maximálne využitie dostupného hardvéru (sieťovej infraštruktúry). Kvôli existencii rôznych technológií, použitých v sieťových zariadeniach, je optimalizácia výkonu siete ako celku náročná. Je možné optimalizovať iba určitú časť siete, so spoločným rozhraním. SDN prináša možnosť optimalizovať sieť globálne. Vďaka centralizovanej správe, SDN ponúka ucelený pohľad na aktuálny stav siete a skrz centralizované algoritmy bude ďalej možné riešiť radu problémov optimalizácie výkonu.

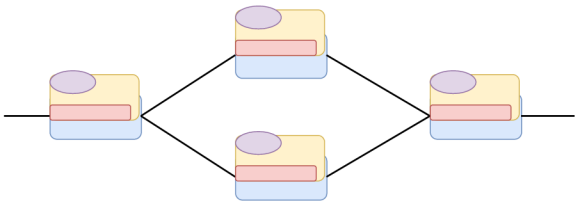
- **Podpora inovácií** – V dnešnom svete celkového technického pokroku, aj čo sa týka sieťových aplikácií, by mali byť budúce sieťové riešenia zamerané na inovatívnosť. Bohužiaľ, každá nová myšlienka sa potýka s problémami implementácie, testovania a nasadenia do existujúcich sieťových riešení. Hlavnou prekážkou je používanie proprietárneho hardvéru a bežných sieťových komponentov, ktoré neposkytujú možnosť modifikácie pre testovanie. SDN, narozdiel od bežného prístupu, ponúka programovateľnú sieťovú platformu, na ktorej je možné experimentovať s novými nápadmi pohodlne a flexibilne.

V SDN má dátová vrstva naďalej rovnakú funkciu ako v bežných sieťach, ale pravidlá na spracovanie paketov získava aj inak ako pomocou riadiacich protokolov. Je možné ich definovať skrz SDN kontrolér, ako je poukázané na obrázku 2.5. Pravidlá na spracovanie paketov sú v bežných sieťach pevne uložené na jednotlivých sieťových zariadeniach, čo zobrazuje obrázok 2.4. Prvou významnejšou implementáciou SDN bol OpenFlow.

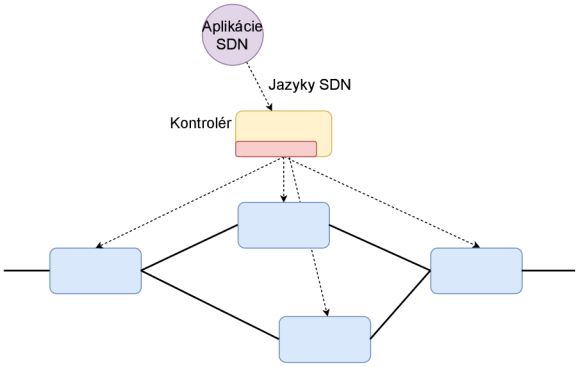
Obr. 2.2: Riadiaca a dátová vrstva sú tesne prepojené



Obr. 2.3: Oddelená riadiaca a dátová vrstva



Obr. 2.4: Tradičná sieť bez aplikácie konceptu SDN



Obr. 2.5: Sieť podľa konceptu SDN

2.3 OpenFlow

Koncept protokolu OpenFlow [7] vznikol ešte pred SDN, v roku 2008. Bol dobre prijatý výrobcami sieťových zariadení, pretože využíval práve vtedy dostupné funkcie zariadení, ako je monitorovanie tokov. Naviac nevyžadoval žiadne odhaľovanie vnútornej štruktúry práve vyrábaných zariadení.

Protokol OpenFlow vytvára, resp. definuje pravidlá pre komunikáciu medzi riadiacou vrstvou a dátovou vrstvou. Nevyžaduje prítomnosť nových špecifických blokov v sieťovom zariadení, iba prepája dátovú vrstvu s riadiacou vrstvou novým spôsobom, ktorý umožňuje spracovávať pakety na základe vlastných stanovených pravidiel. Využíva skutočnosť, že väčšina moderných prepínačov a smerovačov obsahuje tabuľku tokov (flow table). Tabuľka tokov beží súčasne s ostatnými funkciami dátovej vrstvy a je používaná na implementáciu firewallov, NAT, zaistenie kvality služieb (QoS - Quality of Service) a zbiera štatistiky. Hoci zariadenia od rôznych výrobcov majú rozdielne tabuľky tokov, našla sa skupina funkcií, ktoré sú spoločné pre viaceré z nich. OpenFlow využíva tieto spoločné funkcie.

Zariadenie schopné používať OpenFlow protokol sa skladá (aspoň) z 3 častí: **(1) tabuľka tokov**, ktorá má ku každému záznamu o toku priradenú akciu, **(2) zabezpečený kanál (Secure channel – SSL)**, ktorý vytvára spojenie medzi zariadením a riadiacim procesom, nazývaným kontrolér, ktorý umožňuje posielanie príkazov a paketov medzi zariadením a kontrolérom s využitím **(3) OpenFlow protokolu**, ktorý poskytuje štandardizovaný spôsob komunikácie medzi kontrolérom a prepínačom. Stanovením štandardného rozhrania (OpenFlow protokol), skrz ktoré môžu byť záznamy v tabuľke toku špecifikované externe, zariadenia podporujúce OpenFlow nepotrebujú byť inak špecificky programované za účelom zachytávania tokov.

Záznam v tabuľke tokov má 3 polia: **(1) hlavička paketu**, ktorá jednoznačne identifikuje tok, **(2) akcia**, ktorá definuje ako by mal byť paket spracovaný, **(3) štatistika**, ktorá zaznamenáva počet paketov a bitov pre každý tok, a čas, kedy bol naposledy prijatý paket daného toku (pomáha pri identifikácií neaktívnych tokov). Každý záznam v tabuľke toku má so sebou asociovanú nejakú akciu. Tri základné akcie, ktoré musí zariadenie podporovať pre fungovanie OpenFlow sú:

- Smerovanie paketov daného toku na zadaný port alebo porty. Toto umožňuje paketom putovať skrz sieť. Vo väčšine zariadení sa očakáva prebiehanie spracovania na plnej priepustnosti pripojeného rozhrania.

- Zapuzdrenie a smerovanie paketov daného toku do kontroléra. Paket sa dostane do zabezpečeného kanála, kde je zapuzdrený a poslaný kontroléru. Táto akcia sa typicky vykoná vždy pre prvý paket v novom toku, aby kontrolér rozhodol či by mal byť tok pridaný do tabuľky toku.
- Zahodiť pakety daného toku. Táto funkcia sa môže využiť na zvýšenie bezpečnosti, k zabráneniu DoS útokov, alebo obmedzeniu podozrivého zisťovania prevádzky skrz broadcast vysielanie od koncových staníc.

Pre vyššie popísané správanie sieťových prvkov je potrebný **OpenFlow kontrolér**. Prostredníctvom neho sa pridávajú a odstraňujú záznamy v tabuľkách tokov jednotlivých sieťových zariadení, podľa práve prebiehajúceho experimentu (programu). Môže mať podobu napríklad jednoduchej aplikácie bežiacej na počítači. Jedným z prvých OpenFlow kontrolérov bol NOX [9], čím sa začala tvorba viacerých nových kontrolných platforiem.

Kapitola 3

Jazyk P4

Protokol a koncept OpenFlow časom ukázal svoje nedostatky. OpenFlow vyžaduje prítomnosť špecifických funkcií dátovej vrstvy, ktoré nie sú formálne špecifikované. Na začiatku OpenFlow pracoval s jednou tabuľkou pravidiel, s ktorou sa porovnávali pakety na desiatke políčok v hlavičke. Rokmi sa toto rozhranie stávalo stále viac komplexnejšie, s viacerými podporovanými políčkami v hlavičke, s početne viac tabuľkami s pravidlami na porovnávanie len preto, aby prepínače mohli poskytnúť viac zo svojich technických možností kontroléru. Keďže toto nie je úplne praktické [1], vznikla potreba pre novú verziu protokolu OpenFlow, alebo nový programovací jazyk, ktorý by umožňoval naprogramovať sieťové zariadenia tak, aby boli nezávislé na protokole. Užívateľovi (programátorovi) by bolo umožnené si samovoľne hlavičky paketov definovať a meniť, a takto vytvárať nové protokoly pre práve meniace sa požiadavky trhu.

Rozšírením koncepcie SDN sa tak stal jazyk P4. Protokol OpenFlow umožňoval iba zachytávať toky skrz ich špecifikáciu v tabuľkách tokov. Oproti tomu jazyk P4 umožnil meniť funkciu sieťových zariadení aj skrz programovanie týchto zariadení pre dané účely. Ciele jazyka P4 sú:

- **Rekonfigurovateľnosť** – kontrolér by mal byť schopný meniť analyzovanie a spracovanie paketov za behu.
- **Nezávislosť na protokole** – prepínač by nemal byť závislý od špecifických formátov paketov. Namiesto toho by mal byť kontrolér schopný špecifikovať:
  - analyzátor paketov, ktorý by extrahoval polia z hlavičky s predom definovanými názvami a typmi,
  - kolekciu typovaných vyhľadávacích tabuliek, ktoré tieto hlavičky spracúvajú.
- **Nezávislosť od cieľa** – rovnako ako programátor v jazyku C nepotrebuje vedieť presnú architektúru procesora, od programátora kontroléra by tiež nemala byť vyžadovaná znalosť špecifikácie konkrétneho prepínača. Namiesto toho by mal prekladač vziať do úvahy možnosti prepínača pri preklade popisu nezávislého od cieľa (programu napísaného v P4) na program závislý od cieľa (používaný na konfiguráciu prepínača).

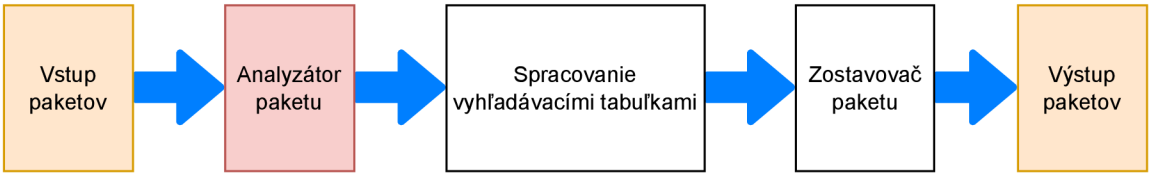
3.1 Čo je P4

P4 (Programming Protocol-Independent Packet Processors) [1] je open source jazyk na popis sieťových zariadení, doménovo špecifický, ktorý bol vytvorený s cieľom umožniť meniť

správanie dátovej vrstvy pre rôzne prípady využitia v sieti. Tento jazyk umožňuje určiť, ako majú zariadenia v sieti spracovávať pakety. Kladie dôraz na rekonfigurovateľnosť po nasadení zariadenia, nezávislosť na protokolu a cieli.

3.2 Koncepty jazyka P4

Program napísaný v jazyku P4 obsahuje definície nasledujúcich kľúčových komponentov:



Obr. 3.1: Obrázok zachytávajúci základnú architektúru programov v P4

- **Hlavička** umožňuje definovať formát hlavičiek protokolov, ktoré bude zariadenie rozpoznávať.
- **Analyzátor paketu** popisuje formou konečného automatu ako sú spracované a extrahované jednotlivé hlavičky paketu od začiatku až po koniec. Počiatočný stav je explicitný, ďalej sa definujú vlastné medzistavy a analýza končí buď stavom prijatia (accept) alebo odmietnutia (reject).
- **Vyhľadávacia tabuľka** *(Match-action table)* definuje porovnávací kľúč, ktorý hľadá zhodu na extrahovaných políčkach hlavičiek a k zhodám priraďuje akcie, ktoré sa majú vykonať.
- **Akcia** definuje komplexné úkony, ktoré sa majú vykonať pre pakety. Používajú sa vo vyhľadávacích tabuľkách a riadiacich blokoch.
- **Riadiaci blok** definuje poradie vyhľadávacích tabuliek, ktoré sú aplikované na paket a ostatné operácie prevedené nad paketom.

Hlavičky slúžia na definovanie políčok v pakete, ktoré sa bude analyzátor snažiť extrahovať. Po úspešnom extrahovaní všetkých hlavičiek v poradí v akom sú definované v analyzátore pokračuje tok programu do riadiacich blokov (čo je možné vidieť na obrázku 3.1). Riadiaci blok obsahuje definície tabuliek a akcií, ktoré sú lokálnymi prvkami daného bloku. To znamená, že rovnaký názov týchto prvkov je možné použiť v iných riadiacich blokoch. Akcie sú spúšťané skrz tabuľky, alebo volané explicitne ako funkcie z riadiacich blokov alebo tiež aj z iných akcií. Tabuľky vyberú na základe kľúča, ktorý môže byť zložený aj z viacero položiek, vždy len jednu akciu, ktorá je prevedená. Pri nenájdení zhody na základe kľúča alebo v prípade ak kľúč nie je zadaný, sa vykoná akcia označená ako *default_action*, prípadne ak nie je zadaná vykoná sa akcia *No_action*. Ako naznačuje názov tejto akcie, je to len prázdna akcia, ktorá nič nevykonáva. Výstupom tabuľky je štruktúra 1, v ktorej je nastavená premenná *hit*, ak bola nájdená zhoda v kľúči, a premenná *miss* ak nie. Tiež je nastavená premenná určujúca typ prevedenej akcie.

Po prechode všetkých v programe definovaných riadiacich blokov nasleduje prechod špecifickým riadiacim blokom, ktorého úlohou je znovu zostaviť paket a odoslať ho na príslušný port. Tento blok sa nazýva *deparser*.

```
enum action_list(T) {
// one field for each action in the actions list of table T
}
struct apply_result(T) {
    bool hit;
    bool miss;
    action_list(T) action_run;
}
```

Kód 1: Štruktúra automaticky syntetizovaná prekladačom, ktorá vzniká volaním tabuľky

3.3 Verzie jazyka P4

Jazyk P4 je doteraz známy v dvoch odlišných štandardizovaných verziách, $P4_{14}$ a $P4_{16}$[6]. $P4_{14}$ umožňuje popísať algoritmy konštruované pre dátovú vrstvu s využitím plejády jazykových konštrukcií, preddefinovaných funkcií a ďalších prvkov jazyka. Neposkytuje však možnosť práce s rôznymi architektúrami zariadení. Ďalšími nedostatkami je slabá typová kontrola, vo všeobecnosti voľnejšia sémantika, relatívne nízkoúrovňové konštrukcie a slabá podpora modularity.

Podpora rôznych cieľov a architektúr je hlavnou vlastnosťou $P4_{16}$. Táto verzia odstránila závislosť jazykových konštrukcií na danej architektúre. Štruktúra, možnosti a rozhrania špecifického cieľa sú teraz zapuzdrené do popisu architektúry, a skrz knižnicu danej architektúry sú dostupné funkcie špecifické pre cieľ. Jazyk sa celkovo zjednodušil, odobralo sa početné množstvo kľúčových slov a prvkov jazyka ako počítadlá a iné, ktoré predpokladali špecifické funkcionality zariadenia. Tieto prvky si užívateľ musí sám deklarovať a definovať, alebo použiť (ak sú deklarované) z knižnice architektúry zariadenia, pre ktoré je program určený. Oproti $P4_{14}$, $P4_{16}$ predstavila taktiež striktné typovanie, výrazy, vnorené dátové štruktúry a niekoľko mechanizmov modularity.

Vďaka značným výhodám $P4_{16}$, bol vývoj $P4_{14}$ ukončený, a dnes je $P4_{14}$ podporovaná len niekoľkými zariadeniami. Referenčný prekladač poskytuje nástroj na export programu napísaného v $P4_{14}$ do $P4_{16}$. Toto umožňuje správcom sietí jednoduchý upgrade ich programov do stále vyvíjanej verzie jazyka P4.

3.4 Architektúra $P4_{16}$

Špecifikácia $P4_{16}$ predstavuje model architektúry[12, 6]. Model architektúry P4 (v skratke: architektúra P4) identifikuje funkčné bloky, ktoré sú prítomné v danej dátovej vrstve cieľa, a taktiež špecifikuje rozhranie medzi nimi. Vo všeobecnosti, pre daný cieľ môžu existovať aj bloky s pevnou funkciou, aj programovateľné bloky. Správanie blokov s pevnou funkciou je definované výrobcom zariadenia, bez možnosti meniť ich správanie skrz P4.

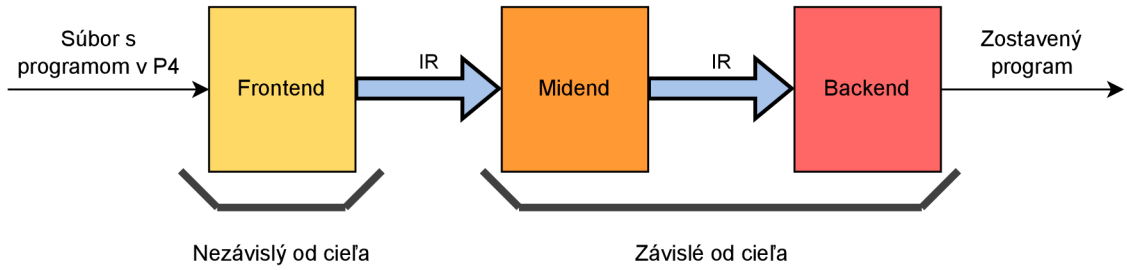
Funkčnosť programovateľných blokov je naopak možné meniť pomocou programu zapísanom v jazyku P4. Stojí za zmienku, že sa nepredpokladá, že programy napísané v P4 budú prenosné naprieč rôznymi modelmi architektúry. Programy vytvorené pre rovnakú architektúru by však mali byť prenosné naprieč všetkými cieľmi, ktoré splňujú uvažovaný model. Iba deklarácie programovateľných blokov sú zahrnuté v definícií modelu architektúry, pretože v princípe žiadny blok s pevnou funkciou nemôže byť manipulovaný programom v P4. Programovateľný blok musí byť označený ako analyzátor paketu alebo riadiaca funkcia. Funkcia analyzátoru paketu je správne identifikovať hlavičky, prítomné v každom prichádzajúcom pakete. Ako bolo už spomínané, typy hlavičiek, ich štruktúra, rovnako ako aj správanie ana-

lyzátora musia byť definované v P4 programe (táto zodpovednosť padá na plecia vývojára P4 programu, nie poskytovateľa modelu architektúry). Pre každý paket, analyzátor vytvorí spracovanú reprezentáciu relevantných hlavičiek, ktoré sú následne predané prvému riadiacemu bloku. Postupnosť riadiacich blokov následne spracováva paket. Toto zahŕňa exekúciu vyhľadávacích tabuliek, verifikáciu kontrolného súčtu a jeho prepočítanie, znovu zloženie paketu a podobne. Sem je zapísaná väčšina vysokoúrovňových funkčných deklarácií modelu architektúry, keďže sú tu zoskupené predošlé deklarované funkčné bloky.

Podrobnú špecifikáciu architektúry musí poskytnúť výrobca cieľa. Pre cieľ výrobca poskytuje knižnicový P4 súbor, obsahujúci všetky potrebné deklarácie funkčných blokov existujúcich v cieľovej zreťazenej linke ich typy ako aj iné dátové typy, konštanty, externé premenné a podobne.

3.5 Prekladač P4

Väčšina prekladačov, vytvorených pre preklad jazyka P4, používa 2-vrstvý model, pozostávajúci zo spoločného frontendu a backendu špecifického pre cieľ. Frontend je spoločný pre všetky ciele a je zodpovedný za spracovanie, syntaktickú a od cieľa nezávislú sémantickú analýzu programu. Program je vo finále transformovaný do internej reprezentácie (ďalej IR) určenej pre backend, ktorý vykonáva transformácie tejto IR závislé od cieľa.



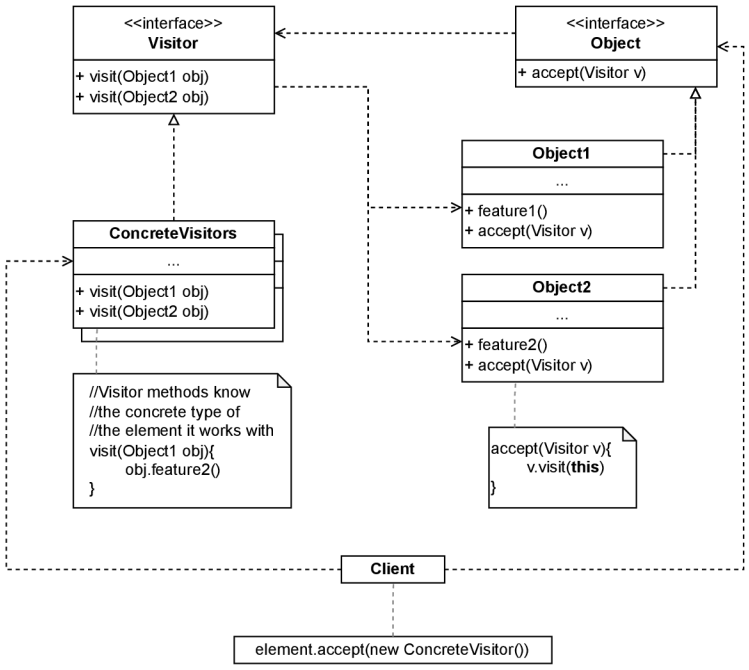
Obr. 3.2: Architektúra prekladača p4c

Priamo od tvorcov jazyka P4 je dostupný open source prekladač p4c [10]. Práve tomu sa podrobnejšie venujem v nasledujúcom texte a aj v celej práci. Uvedený p4c prekladač sa snaží čo najlepšie implementovať reštrikcie uvedené v špecifikácií jazyka P4 [3] a plne zodpovedať definovanej syntaxi a sémantike. Prekladač pracuje v 3 priechodoch, ako je možné vidieť na obrázku 3.2: frontend, midend a backend. V prvom priechode sa načíta kód zo súboru a prevedie sa do IR. Následne prebiehajú nad IR transformácie a optimalizácie nezávislé od cieľa, ktoré sú súčasťou syntaktickej a sémantickej analýzy, validácia programu a typová kontrola. V druhom priechode prebiehajú optimalizácie a transformácie nezávislé od cieľovej architektúry, ale založené na cieľovej architektúre. Používa sa tu rovnaká IR ako vo frontende. Posledný priechod je kompletne závislý od cieľa, generuje sa v ňom kód a alokujú zdroje. Môže využívať vlastnú IR.

IR je nemenná, silne typovaná a má štruktúru orientovaného acyklického grafu. Je manipulovateľná, až na pár výnimiek, len skrz vzor návštevník. Vo frontend a midend priechodoch je vždy serializovateľná naspäť do P4 programu. Toto zjednodušuje testovanie a ladenie programu. IR je možné v ktorejkoľvek fáze prekladu serializovať a zapísať vo for-

máte json do súboru. Toto pomáha naučiť sa a pochopiť štruktúru IR, a zistiť ako vyzerá v jednotlivých fázach prekladu.

Prekladač hojne využíva návrhový vzor návštevník (visitor pattern) [14]. V skratke ide o triedu, ktorá pridáva novú funkcionalitu do už existujúcej triedy, bez potreby toho aby bola pôvodná trieda rozšírená o nové metódy. Obrázok 3.3 zachytáva princíp tohto návrhového



Obr. 3.3: UML diagram návrhového vzoru návštevník

vzoru za pomoci UML diagramu.

1. Rozhranie **Visitor** (vľavo hore) deklaruje metódy, ktoré berú ako parameter navštevovaný objekt. Každá konkrétna implementácia metódy `visit` môže pracovať rozdielne s poskytnutou triedou, od vyvolania určitej metódy danej triedy, po zmenu hodnôt triednych premenných.

2. Rozhranie **Object** (vpravo hore) deklaruje metódu `accept`, ktorá berie argument typu **Visitor**. Každý zodpovedajúci objekt musí implementovať metódu `accept`, ktorej úlohou je presmerovať volanie na zodpovedajúcu metódu `visit`, ktorej argument má typ triedy, zodpovedajúci triede tohoto objektu.

3. **Client** (dole) reprezentuje nejaký komplexnejší objekt (napríklad strom zložený z objektov). Klienti zvyčajne nepoznajú zodpovedajúce dátové typy objektov, pretože s nimi pracujú skrz nejaké abstraktné rozhranie.

Výhodami tohto návrhového vzoru je možnosť pridania novej funkcionality do už existujúcej triedy, bez zmeny aktuálneho kódu danej triedy. Nevýhodou je, že návštevník môže pristupovať iba k verejným premenným a metódam danej triedy, a spracovávať ich. Nemôže pristupovať k privátnym metódam a premenným. Možným riešením je zmena privátnych metód a premenných na verejné, alebo vnorenie samotnej triedy návštevník do danej triedy, ktoré ale nie je možné vo všetkých programovacích jazykoch.

Prekladač používa viacero typov triedy návštevník, ktoré všetky dedia z bázovej triedy *Visitor*. Trieda *Inspector* sa používa na prechod objektom, ktorého cieľom je zber dát a možné vytvorenie nového objektu z nahromadených dát. Trieda *Modifier* môže meniť obsah navštevovaného objektu. Trieda *Transform* je používaná pre zmenu štruktúry navštevovaného objektu, zväčša ide o zmenu poradia uzlov v abstraktnom syntaktickom strome (ďalej len AST), spájanie viacerých uzlov alebo pridávanie nových. Ďalšími triedami, ktoré dedia z triedy *Visitor*, sú triedy *ControlFlowVisitor*, *Backtrack* a *P4WriteContext*.

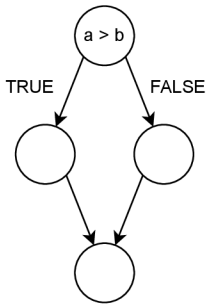
Kapitola 4

Graf riadenia toku

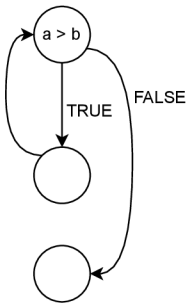
Teoretický popis a informácie v tejto kapitole boli čerpané z kníh [4, 8]. Graf riadenia toku programu (ďalej len CFG) je graf zostavený z **blokov**, ktoré predstavujú najdlhšiu súvislú sekvenciu kódu, nezávislú od nejakej podmienky, a **hrán**, ktoré predstavujú prechody medzi jednotlivými blokmi. Blok je sekvencia príkazov, ktoré sa vždy vykonajú v definovanom poradí, pokiaľ nie je vyvolaná výnimka. Začiatok bloku vždy tvorí príkaz, ktorý nasleduje po predchádzajúcej podmienke, a koniec bloku podmienka, skok alebo predpokladaný príkaz.

Vstupný bod (blok) CFG je uzol, skrz ktorý vstupuje tok riadenia do programu. **Výstupný bod** je uzol, v ktorom tok riadenia program opúšťa. Program môže mať viacero vstupných bodov, ktoré sa ale obvykle spájajú do jedného pre zjednodušenie. Je teda netypické aby mal program viacero vstupných bodov. Taktiež môže mať program viacero výstupných bodov, ktorých výstup je tiež možné spojiť do jedného uzlu.

CFG modeluje tok programu medzi jednotlivými blokmi. Je to orientovaný graf, $G = (N, E)$. Každý uzol $n \in N$ korešponduje bloku. Každá hrana $e = (n_i, n_j) \in E$ korešponduje možnému prechodu toku programu z bloku n_i do bloku n_j.



Obr. 4.1: CFG pre podmienku



Obr. 4.2: CFG pre cyklus

Pre zjednodušenie algoritmov na analýzu a optimalizáciu sa používajú bloky o jednom príkaze. Ich nevýhodou je ale väčšia pamäťová náročnosť, prechod takýmto grafom trvá značne dlhšie a majú viacero hrán a uzlov.

Existuje viacero spôsobov ako vykresľovať výsledné CFG a rôzne konštrukcie, obsiahnuté v nich. Hlavným kritériom pre vizualizáciu je, aby výsledný graf čo najlepšie zachytával princíp vykresľovanej konštrukcie. Dôležité je aby vykreslená konštrukcia nemusela byť slovne dovysvetlená, a jej princíp bol na prvý pohľad jasný. Na obrázkoch 4.1 a 4.2 sú zachytené spôsoby, akými sa bežne vykresľujú jazykové konštrukcie podmienky a cyklu.

Vo všeobecnosti, veľa častí prekladačov programovacích jazykov závisí od CFG. Analýza ktorá pomáha optimalizovať kód zvyčajne začína analýzou CFG a jeho konštrukciou. Prekladače typicky používajú CFG v spojení s inou IR. CFG reprezentuje vzťahy medzi blokmi, zatiaľ čo príkazy v blokoch sú reprezentované inou štruktúrou, ako abstraktný syntaktický strom alebo iné. Kombináciu týchto reprezentácií vzniká hybridná IR.

CFG pomáha odhaliť mŕtvy kód v programe a cykly. Mŕtvy kód v programe je blok, z ktorého sa nedá dostať do žiadneho výstupného bodu. Začína sa prvým uzlom, z ktorého neexistuje cesta do výstupného bodu. Takúto časť programu je možné odstrániť, čím sa síce stráca istá funkcionalita, ktorej prítomnosť v programe by nám ale spôsobila nedobehnutie programu. Preto je chcené bloky mŕtveho kódu z programu eliminovať.

Syntax P4 nepodporuje explicitnú tvorbu cyklov, môžu však vzniknúť vo vyhľadávacích tabuľkách, napríklad ak nejaká akcia vyvolá znovu samú seba na základe nejakej podmienky. Cykly v programe sú nežiadané, pretože hardvér na sieťových zariadeniach často dokáže cykliť iba do určitej hĺbky. Pre zabránenie tvorby cyklov v syntaktických analyzátoroch sa používa technika *loops unrolling*[11], kde sa prekladač snaží odstrániť cykly prepisom kódu tak, aby prebiehal sekvenčne.

Kapitola 5

Návrh generovania CFG v P4

5.1 Súčasný stav

Prekladač p4c poskytuje backend *p4c-graphs* na generovanie CFG vo formáte dot. Tento backend v súčasnej dobe podporuje generovanie grafov pre prvky P4 programu nachádzajúce sa na najvyššej úrovni programu. Backend generuje grafy v dot formáte pre vizualizáciu pomocou nástroja GraphViz, pre analyzátory paketov a riadiace bloky prítomné v $P4_{14}$ alebo v $P4_{16}$ zdrojových súboroch. V nasledujúcom odstavci popisujem tvorbu grafov pre riadiace bloky v súčasnom riešení backendu *p4c-graphs*. Spomínané funkcie sa nachádzajú v súbore *controls.cpp*.

Generovanie grafov prebieha prechodom abstraktného syntaktického stromu (ďalej len AST), v ktorom je IR uložená. Aj pri prechode AST sa využíva návrhový vzor návštevník. Vyhľadávajú sa uzly, ktorých typ je `IR::ControlBlock` čo značí, že ide o riadiace funkcie. Po úspešnom nájdení sa vytvorí telo CFG, s uzlom `__START__` a prechodom do prázdneho uzlu. Pre tento blok sa získa uzol `IR::P4Control` cez funkciu `getNode()`. Zo získaného uzlu sa následne prejde do uzlu typu `IR::BlockStatement`, obsahujúceho telo riadiacej funkcie. V ňom sa nájde atribút `components`, čo je pole s jednotlivými blokmi programu v riadiacej funkcii. Každý prvok v poli `components` sa ďalej navštívi, podľa jeho typu v IR sa spustí príslušná funkcia, ktorá berie daný prvok ako argument. Vo vyvolaných funkciách sa vytvárajú a pridávajú nové uzly do CFG. Navštívením všetkých prvkov v poli `components` sa končí analýza daného uzlu AST s typom `IR::ControlBlock` a do CFG sa pridá hrana `__END__`, značiaca koniec riadiacej funkcie. Následne je vygenerovaný CFG pre danú riadiacu funkciu, a zapísaný do dot súboru s názvom rovnakým aký má spracovaná riadiaca funkcia. Toto prebieha až pokiaľ nie sú prejdené všetky uzly v AST typu `IR::ControlBlock`, čiže vygenerované CFG pre všetky riadiace bloky v danom P4 programe.

V súčasnom riešení je pri tabuľkách problém, že ich prechod končí ich nájdením. Nevyhľadáva sa uzol s kľúčom, od ktorého závisí spustená akcia, a taktiež sa nevyhľadávajú uzly s akciami, ktoré môže vyvolať tabuľka. Neprejdením týchto uzlov sa stráca informácia o závislosti toku programu na príkazoch, vykonávaných akciami a závislosť vykonanej akcie na kľúči danej tabuľky.

Hlavný blok implementácie súčasného riešenia sa nachádza v súbore *p4c-graphs.cpp*. Prijíma vstupné argumenty:

- --graphs-dir (dir) – Cesta ku zložke, do ktorej sa uložia vytvorené CFG jednotlivých funkčných blokov programu vo formáte dot.

- --fromJSON (file) – Cesta ku súboru s IR programu zapísanou vo formáte json, ktorá sa použije na generovanie CFG. Ak je tento argument zadaný, nemusí sa zadávať cesta k programu v jazyku P4.

Ak nie je zadaný argument --fromJSON, použije sa súbor s programom v P4. Súbor s programom v P4 sa spracuje frontend priechodom a uloží do premennej `program`. Následne sa spracuje IR v premennej `program` midend priechodmi, a uloží sa do premennej `top`. V prípade, že bol zadaný argument --fromJSON, načíta sa IR programu, ktorá je uložená v zadanom súbore. Táto reprezentácia zodpovedá IR programu v P4 po spracovaní frontend priechodmi, a preto je uložená do premennej `program` bez nejakých úprav. Pre vytvorenie inštancie tejto triedy sa použije konštruktor, čo berie ako argument reprezentáciu vo formáte json. Ďalej sa IR uložená v premennej `program` spracuje rovnako ako v predošlom prípade, a výsledok midend prechodov sa uloží do premennej `top`. Po týchto prechodoch máme uložené všetko potrebné na tvorbu CFG riadiacich funkcií a analyzátorov paketov. Nasleduje vytvorenie CFG riadiacich blokov. Volaním `top->getMain()->apply(cgen)` sa navštívi daná hodnota `top->getMain()` triedou `controls`, ktorá vytvára CFG riadacich blokov. Volanie `top->getMain()` získa všetky uzly typu `IR::Block::constantValue`. Po tomto spracovaní nasleduje tvorba CFG analyzátorov paketu. Volaním `program->apply(pgg)` sa navštívi premenná `program` triedou `parsers`, ktorá vytvára CFG analyzátorov paketu. Dosiahnutím tohto bodu program končí. Tvorby CFG riadiacich blokov a analyzátorov paketu nie sú od seba závislé, čo znamená, že jednotlivé volania je možné prípadne prehodiť. Jednotlivé triedy, ktoré riešenie obsahuje, sú popísané nižšie. Celé riešenie sa nachádza v mennom priestore *graphs*.

Trieda *Graphs* dedí z triedy *Inspector*, ktorá je typu návštevník. Z triedy *Graphs* ďalej dedí trieda *Controls*. Obsahuje potrebné funkcie a premenné, ktoré umožňujú plnú funkcionalitu štandardnej boost knižnice na tvorbu grafov vo formáte dot. Obsahuje definície štruktúr, ktorými sa popisujú uzly a hrany boost grafu. Tieto štruktúry sa ukladajú ako atribúty uzla alebo hrany, a na základe nich sa nastavujú atribúty pri zápise grafu do súboru vo formáte dot. Ďalej táto trieda obsahuje funkcie pre pridávanie uzlov a hrán do boost grafu s tým, že tieto funkcie zároveň nastavujú potrebné atribúty pre práve pridané prvky.

Pre tvorbu CFG riadiacich blokov je špecializovaná trieda *Controls*. Táto trieda funguje na princípe návrhového vzoru návštevník, a dedí od triedy *Graphs*. Vytváranie jednotlivých CFG pre riadiace bloky v tejto triede začína vo funkcii `preorder(const IR::PackageBlock *block)`. V tejto funkcii sa postupne prechádzajú všetky uzly AST, ktoré predstavujú konštantné hodnoty. Ako prvé sa v nej skontroluje či spracovávaný uzol obsahuje nenulovú hodnotu. Ak nie, preskakuje sa na ďalší uzol predstavujúci konštantnú hodnotu. Inak sa skontroluje či uzol je typu `IR::ControlBlock` a ak je, pokračuje sa spracovaním, ktoré bude popísané nižšie. Ak nie je, skontroluje sa či je uzol typu `IR::PackageBlock`, a navštívi sa tento uzol. To znamená, že sa rekurzívne opäť spustí funkcia *preorder*.

Spracovanie uzlu typu `IR::ControlBlock` pokračuje vytvorením boost grafu, ktorý sa uloží do premennej `g_`. V tomto grafe sa vytvorí podgraf, a následne sa uloží do triednej premennej `g`. Táto premenná je dôležitá, pretože práve skrz ňu sa pridávajú uzly a hrany, pri volaní zodpovedajúcich funkcií. Následne je nastavený atribút mena pre graf `g_`. Pridané sú uzly `__START__` a `__EXIT__`. V triednej premennej `parents` sa nachádzajú uzly, z ktorých bude viesť nasledujúca hrana pri vkladaní nového uzla do grafu.

Následne sa zavolá funkcia `visit` s argumentom typu `IR::ControlBlock`. Tá spôsobí spustenie nasledujúcej funkcie (výčet zachytáva poradie spustených funkcií):

1. `preorder(const IR::ControlBlock *block)` – Tu sa zavolá funkcia `visit`, s argumentom typu `IR::P4Control`.

2. `preorder(const IR::P4Control *cont)` – V tejto funkcii ako prvé prebehnú operácie, ktoré vynulujú potrebné premenné, ak aktuálny graf obsahuje viac ako jeden podgraf. S takýmto prípadom som sa ale nestretol, tento kód teda považujem za istú kontrolu správnosti (safety-check). Po týchto operáciách sa zavolá funkcia `visit` s argumentom typu `IR::BlockStatement *statement)`.

3. `preorder(const IR::BlockStatement *statement)` – V tejto funkcii sa prechádza (volá sa na ne funkcia `visit`) všetkými uzlami, ktoré môžu byť rôzneho typu, a na ktorých referencie sú uložené v poli `statement->components`.

V nasledujúcich odsekoch sú spomenuté jednotlivé funkcie, ktoré môže vyvolať funkcia `visit` volaná s argumentom typu uzol z poľa spomínaného vyššie (`statement->components`). Uvedené funkcie spracúvajú uzol daného typu, pridávajú potrebné hrany a uzly do boost grafu, a prípadne volajú funkciu `visit`, ak sú to uzly, ktoré spôsobujú vetvenie.

- `preorder(const IR::IfStatement *statement)` – Volá funkciu `visit` na obidva uzly, `TRUE` aj `FALSE`, ktoré môže podmienka vyvolať.

- `preorder(const IR::SwitchStatement *statement)` – Zavolá funkciu `visit` na všetky uzly s jednotlivými stavmi danej prepínacej konštrukcie.

- `preorder(const IR::MethodCallStatement *statement)` – Najskôr zavolá na argument funkciu, ktorá ho zanalyzuje a pretypuje na správny typ, alebo vytvorí uzol zodpovedajúceho typu. Podľa typu tohto argumentu sa buď navštívi uzol typu `IR::P4Table` alebo `IR::P4Control`.

- `preorder(const IR::AssignmentStatement *)` – Vloží argument `statement` na zásobník `statementStack`.

- `preorder(const IR::ReturnStatement *)` – Vytvorí uzol zo všetkých príkazov nachádzajúcich sa v zásobníku `statementStack` a do premennej `return_parents` vloží potrebné uzly tak, aby sa mohli vytvoriť zodpovedajúce hrany.

- `preorder(const IR::ExitStatement *)` – Má funkcionalitu rovnakú ako predchádzajúca funkcia.

- `preorder(const IR::P4Table *table)` – získa z argumentu názov tabuľky, a vytvorí v grafe uzol typu `TABLE`. Aktualizuje premennú `parents`.

Po prechode všetkými uzlami v poli `statement->components` sa program postupne vynorí späť, a vráti do funkcie `preorder(const IR::PackageBlock *block)`. Tu sa vytvoria chýbajúce hrany, ktoré získame iteráciou cez premennú `parents`. Potom sa nastavia atribúty daného grafu, čo má na starosti trieda *GraphAttributeSetter*, ktorá je súčasťou už vyššie spomenutej triedy *Graphs*. Graf sa následne uloží do súboru, a pokračuje sa v iterácií skrz uzly AST predstavujúce konštantné hodnoty.

Na tvorbu grafov analyzátorov paketu je špecializovaná trieda *Parsers*. Táto trieda tiež funguje na návrhovom vzore návštevník, dedí od triedy *Inspector*. Navštevuje uzly v AST typu `IR::ParserState*`, `IR::PathExpression*` a `IR::SelectExpression*`. Vo funkciách `postorder`, ktoré berú ako argument uzly s typmi vymenovanými v predchádzajúcej vete, sa

spracúvajú uzly predstavujúce jednotlivé stavy analyzátora paketu, a uzly ktoré predstavujú prechody medzi týmito stavmi. Jednotlivé stavy a prechody medzi stavmi sú uložené v mapách:

- `transitions` – uchováva objekty štruktúrovaného typu `TransitionEdge`.
- `states` – uchováva objekty typu `IR::ParserState`.

Obidve mapy mapujú jednotlivé objekty, ktoré skladujú, na objekt typu `IR::P4Parser`, ktorý predstavuje uzol s daným analyzátorom paketu.

Po prechode všetkými uzlami, ktoré obsahujú stavy a prechody daného analyzátora paketu sa zavolá funkcia `postorder` s argumentom typu `IR::P4Parser* parser`. V nej sa z mapy `states` vyberú stavy, ktoré sú namapované na analyzátor paketu, určený argumentom funkcie. Tieto stavy sa zapíšu ako uzly CFG vo formáte dot do súboru. Následne sa z mapy `transitions` vyberú prechody prislúchajúce práve spracovávanému analyzátoru paketu, a zapíšu sa ako prechody medzi uzlami CFG vo formáte dot. Takto sa vytvárajú CFG pre všetky analyzátory paketu, ktoré sú definované v danom P4 programe. Oproti triede *Controls* sa tu na tvorbu CFG nepoužíva boost knižnica, ale graf sa rovno zapisuje vo formáte dot do súboru s názvom daného analyzátora paketu, bez použitia nejakej internej štruktúry, ktorá by ho reprezentovala.

5.2 Návrh vylepšenia

Analýzou funkcionality súčasného riešenia a serializovanej IR vybraného P4 programu[1] som dospel k nasledujúcim záverom. Je potrebné:

- Zachytiť tok programu v tabuľkách. Využije sa súčasné riešenie, v ktorom sa rozšíri prejdenie uzlu s tabuľkou o:
  1. Získanie a prechod uzlom s porovnávacím kľúčom danej tabuľky.
  2. Získanie a prejdenie uzlami s jednotlivými akciami, ktoré je možné skrz tabuľku vyvolať.
- Spojenie CFG pre jednotlivé prvky programu do jedného CFG, čo sprehľadní analýzu toku programu. Pre túto funkcionalitu sa grafy v súčasnom dot formáte prevedú do json formátu, čo zjednoduší ďalšiu manipuláciu s nimi. Následné spojenie grafov bude prebiehať tak, že sa odstránia nepotrebné `__START__` a `__END__` uzly, a pridajú sa hrany, ktoré spoja grafy jednotlivých prvkov programu v poradí v akom sú definované v `main`.

Pre zachytenie toku programu v tabuľkách, bude potrebné rozšíriť funkciu `bool ControlGraphs::preorder(const IR::P4Table *table)` o vyhľadanie kľúča v tabuľke a prejdenie tabuľke príslušiacich akcií. Signatúra funkcií, ktoré je potrebné pridať:

- `bool ControlGraphs::preorder(const IR::TableProperties *properties)`
  V tejto funkcií bude potrebné nájsť uzol typu `IR::Property` s názvom `key`, a dostať sa až k uzlu typu `IR::StringLiteral` ktorého hodnota predstavuje vyhľadávací kľúč v tabuľke.

[1] program dostupný na adrese https://hikingandcoding.wordpress.com/2019/09/17/getting-started-with-p4/

- `bool ControlGraphs::preorder(const IR::Key *key)`
  Funkcia ktorá získa vyhľadávací kľúč v tabuľke a pridá ho ako uzol do CFG. Tento uzol bude koreňom pre všetky akcie, ktoré môže tabuľka vyvolať.

- `bool ControlGraphs::preorder(const IR::P4Action *action)`
  Funkcia ktorá prejde jednotlivé akcie a príkazy v nich a pridá zodpovedajúce uzly do CFG.

Tieto prvky sa nachádzajú v IR v poli `controlLocals`, čo je atribút uzlu `IR::P4Control`.

Pre vytváranie celkového grafu toku programu bude potrebné vytvoriť novú triedu, ktorá bude obsahovať funkcie na spájanie jednotlivých CFG. Predbežná signatúra premenných a funkcií ktoré bude trieda obsahovať:

- `merged_cfg` – Premenná do ktorej sa bude postupne ukladať spájaný CFG.
- `current_cfg` – Pomocná premenná obsahujúca práve načítaný CFG, ktorý sa pridá k `merged_cfg`.
- `fnc getMain()` – Funkcia získa názvy riadiacich blokov a analyzátorov a vráti ich v poradí, v akom sa nachádzajú pri definícií `main` v spracovávanom P4 programe.
- `fnc loadCFG(name)` – Funkcia načíta CFG s názvom daný parametrom `name`.
- `fnc mergeCFG()` – Funkcia pripojí aktuálne načítaný graf v premennej `current_cfg` ku vytváranému grafu v premennej `merged_cfg`.
- `fnc Write_to_file(fileName)` – Funkcia zapíše vytvorený graf v premennej `merged_cfg` do súboru vo formáte json.
- `fnc CreateMergedCFG()` – Hlavná funkcia, ktorá:
  1. Pomocou `getMain()` načíta názvy prvkov programu do premennej `top_level_names`.
  2. Ďalej bude iterovať skrz mená v premennej `top_level_names`, pre každé meno načíta graf pomocou `loadCFG(name)` a pridá ho ku generovanému grafu v `merged_cfg`.
  3. Po dokončení cyklu sa graf zapíše do súboru pomocou funkcie `Write_to_file(fileName)`.

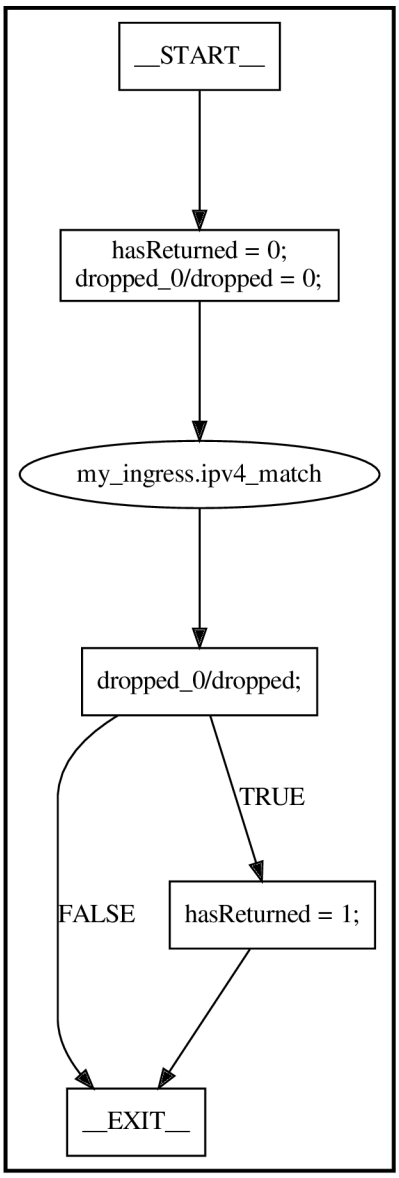
V ďalšej sekcii je možné nájsť mockup toho, ako bude CFG vyzerať po pridaní prechádzania skrz tabuľky a akcie v AST.

5.3 Mock-up

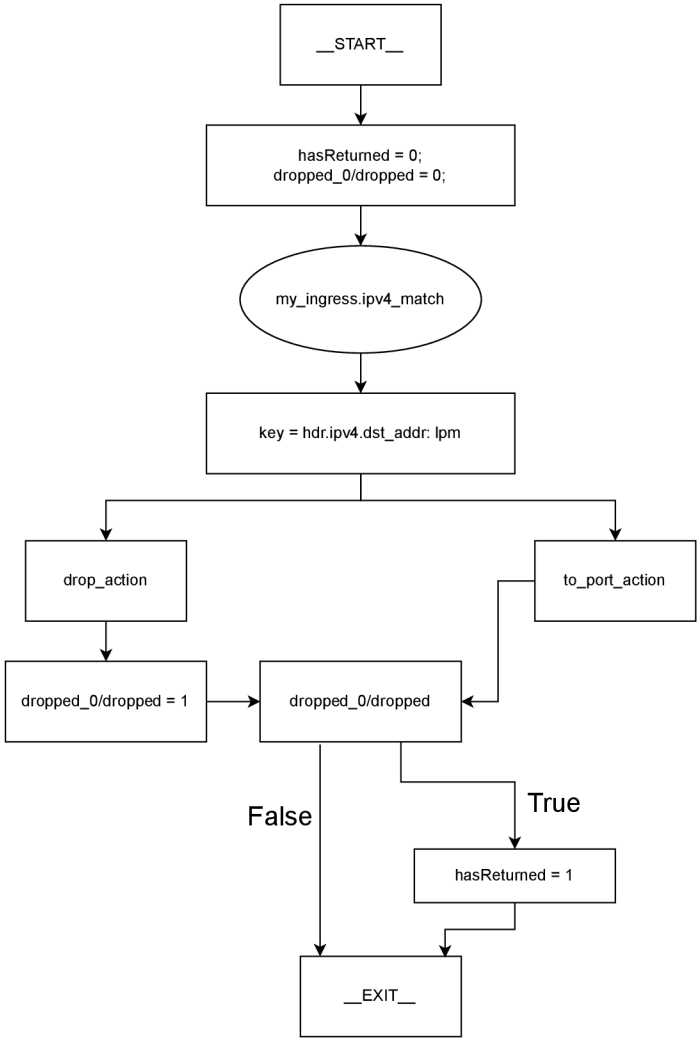
Z kódu jednoduchého P4 programu, dostupného na odkaze 1, je vyňatá riadiaca funkcia `my_ingress` ako kód 2. Táto funkcia obsahuje tabuľku, na ktorej je možné znázorniť, čo súčasné riešenie generujúce CFG nezachytáva a je treba zachytiť v navrhovanom riešení. Obrázok 5.1 ukazuje CFG generovaný súčasným riešením, kde je zaznamenaný iba prechod toku programu do tabuľky, ale nie sú zaznamenané akcie v tabuľkách. Navrhované riešenie 5.2 zachytáva prechod akciami v tabuľke, kde sa akcia vyberie na základe kľúča, deklarovaného v tabuľke a nastaveného skrz kontrolér.

```
control my_ingress(inout headers_t hdr,
                   inout metadata_t meta,
                   inout standard_metadata_t standard_metadata)
{
    bool dropped = false;

    action drop_action() {
        mark_to_drop(standard_metadata);
        dropped = true;
    }

    action to_port_action(bit<9> port) {
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
        standard_metadata.egress_spec = port;
    }

    table ipv4_match {
        key = {
            hdr.ipv4.dst_addr: lpm;
        }
        actions = {
            drop_action;
            to_port_action;
        }
        size = 1024;
        default_action = drop_action;
    }

    apply {
        ipv4_match.apply();
        if (dropped) return;
    }
}
```

Kód 2: Riadiaca funkcia obsahujúca `my_ingress` tabuľku



Obr. 5.1: Súčasný vygenerovaný CFG



Obr. 5.2: Mockup novonavrhovaného CFG, ktorý zachytáva tok programu v tabuľkách

Kapitola 6

Implementácia

6.1 Formát json výstupu

Pre vytvorenie výstupu vo formáte json boli dve možnosti:

1. Konvertovať graf vytvorený skrz boost knižnicu do json. Graf by mal podobnú štruktúru ako v dot, a bol by spätne serializovateľný do dot skrz štandardný nástroj príkazovej riadky, ktorý sa používa na generovanie grafov vo formáte pdf alebo png, z už existujúceho dot súboru.

2. Vytvoriť vlastnú štruktúru json súboru, ktorý by reprezentoval CFG.

Zvolil som druhú variantu. Mojím cieľom bolo vytvoriť json formát, ktorý by bol ľahko čitateľný a pochopiteľný aj pre človeka nepoznajúceho jeho kompletnú štruktúru. Prvá varianta takýto json neponúkala. Preto som sa rozhodol navrhnúť vlastnú, prehľadnejšiu štruktúru json súboru, nad ktorou by sa dali ďalej vytvoriť jednoduché nástroje na jej spracovanie.

Formát mnou navrhnutého json výstupu je ilustrovaný v kóde 3. Objekt *info* obsahuje názov programu, ktorý korešponduje s názvom vstupného súboru s P4 programom. Pole *nodes* na riadku 5 obsahuje objekty, ktoré predstavujú jednotlivé prvky programu v P4. Jednotlivé prvky sú objekty, s položkami:

- *type* – Predstavuje typ prvku programu. Nadobúda hodnoty `"parser"` alebo `"control"`.
- *name* – Predstavuje názov prvku v programe.
- *nodes* (riadok 9) – Pole, ktoré obsahuje jednotlivé uzly CFG. Tieto uzly sú reprezentované objektmi s položkami *node_nmb*, *name*, *type*, *type_enum*.
- *transitions* (riadok 24) – Pole, ktoré obsahuje jednotlivé prechody medzi uzlami CFG. Jednotlivé prechody obsahujú položky *from*, *to*, *cond*.

Objekty v poli *nodes* majú štruktúru:

- *node_nmb* – Reprezentuje index na ktorom sa uzol nachádza v poli *nodes*. Je to pomocný údaj, ktorý môže uľahčiť budúce spracovanie.
- *name* – Obsahuje reťazec, ktorý bude vypísaný v strede uzla.
- *type* – Obsahuje informáciu o aký typ uzla sa jedná.

```
{
    "info": {
        "name": "program"
    },
    "nodes": [
        {
            "type": "parser",
            "name": "MyParser",
            "nodes": [
                {
                    "node_nmb": 0,
                    "name": "start",
                    "type": "state",
                    "type_enum": 8
                },
                {
                    "node_nmb": 1,
                    "name": "accept",
                    "type": "state",
                    "type_enum": 8
                },
                ...
            ],
            "transitions": [
                {
                    "from": 0,
                    "to": 1,
                    "cond": "allways"
                },
                ...
            ]
        },
        ...
    ]
}
```

Kód 3: Formát json výstupu

- *type_enum* – Typ uzla je v boost grafe reprezentovaný výčtom. Táto položka obsahuje číslo, ktoré zodpovedá danému typu v tomto výčte. Uloženie tejto informácie môže znovu uľahčiť budúce spracovanie, najmä ak v programe ktorý spracuje výstupný json používame rovnaký výčet, ako sa nachádza v triede *Graphs*.

Položky v poli *transitions* sú zasa objekty so štruktúrou:

- *from* – Reprezentuje index uzla z poľa *nodes*, z ktorého hrana začína.
- *to* – Reprezentuje index uzla z poľa *nodes*, ku ktorému hrana smeruje.
- *cond* – Obsahuje reťazec, označujúci podmienku, ktorá musí byť splnená, aby tok programu šiel v smere danej hrany. Prípadne obsahuje iný slovný popis, ktorý nám upresňuje kedy alebo prečo sa tok programu vyberie danou hranou (`always`, `default`).

V mojej prvej implementácií sa json výstup vytvára zároveň s vytváraním boost grafov. Po zistení, že všetky potrebné informácie na vytvorenie json výstupu má v sebe štruktúra

reprezentujúca boost graf, sa v druhej implementácií vytvára json výstup pomocou získavania uzlov a hrán z grafov vytvorených za pomoci boost knižnice. Presný popis generovania json výstupu bude vysvetlený v nasledujúcich sekciách.

6.2 Rozšírenie o prechod uzlami s tabuľkou, kľúčom a akciami

Pre pridanie tejto funkcionality bolo nutné zistiť, ktorými volaniami sa dostanem k potrebným uzlom AST. Rozširovala sa už existujúca funkcia `preorder(const IR::P4Table *table)`, ktorú je možné pozrieť v kóde 4. V texte sa ďalej budem odkazovať na riadky v tomto kóde. V nej bolo pridané získanie uzla s kľúčom danej tabuľky, pomocou volania `table->getKey()` (riadok 8). Po získaní uzla s kľúčom mohla nasledovať implementácia metódy `preorder(const IR::Key *key)`, v ktorej sa prechádza pole `key->keyElements` a do premennej `sstream`, ktorá slúži na budovanie textového reťazca, sa postupne zapisujú všetky prvky, ktoré tvoria daný kľúč. Po prechode poľom nasleduje pridanie uzlu do boost grafu a aktualizovanie premennej `parents`.

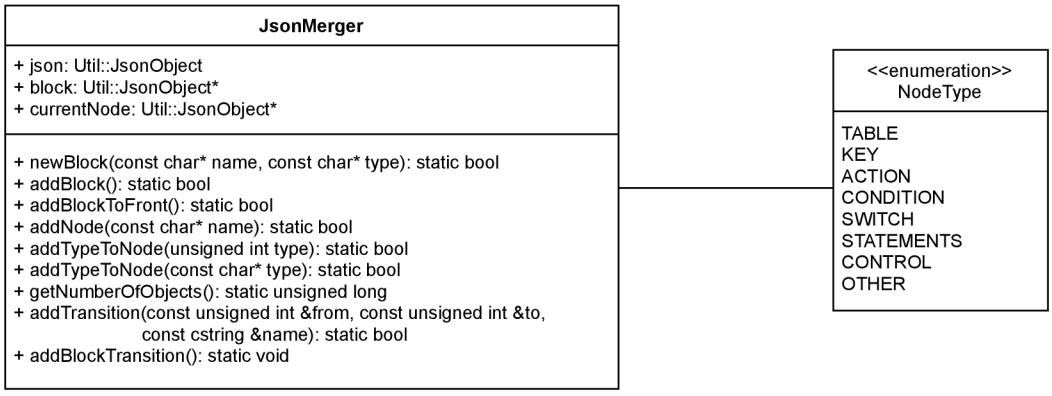
Vo funkcii spracovávajúcej tabuľku sa ďalej pokračuje získaním všetkých akcií skrz volanie `table->getActionList()->actionList`, ktoré sa uložia do premennej `actions` (riadok 16). Ešte predtým sa ale uloží do premennej `keyNode` typu `Parents` hodnota `parents.back()` (riadok 11 a 12). Hodnota tejto premennej je dôležitá pre správne vytvorenie hrany k uzlu s akciou tak, aby vždy vychádzala od uzla s kľúčom. Pri iterácií skrz všetky akcie sa práve z tohto dôvodu najprv nastaví premenná `parents` na hodnotu premennej `keyNode` (riadok 18). Pre každú akciu sa získa jej názov a v CFG sa vytvorí uzol, s označením zodpovedajúcim názvu akcie (riadok 20). Skrz následné volania sa operáciami nad IR reprezentáciou získa uzol s telom akcie (riadky 24-32). Toto telo sa vždy navštívi, teda zavolá sa funkcia `preorder(const IR::P4Action *action)` (riadok 33). V nej sa iba zavolá funkcia `visit` nad telom akcie, ktoré predstavuje blok s príkazmi a je uložené v `action->body`. Telo akcie sa spracováva rovnako, akoby šlo o telo riadiacej funkcie. Po spracovaní tela akcie nasleduje volanie funkcie `merge_other_statements_into_vertex`, ktorá ak sú ešte nejaké príkazy na zásobníku `statementStack`, pridá ich ako uzol do CFG (riadok 38). Po nej sa do premennej `new_parents` pridajú hodnoty, ktoré budú zodpovedať pozícií posledného uzla danej akcie (riadok 40). Hodnota uložená v tejto premennej je dôležitá pre správne spojenie posledných uzlov všetkých akcií s nasledujúcim uzlom v CFG. Pokračuje sa vymazaním hodnoty v premennej `parents` (riadok 41). Po takomto spracovaní všetkých akcií prítomných v danej tabuľke sa práve do premennej `parents` uloží hodnota z premennej `new_parents` (riadok 44).

6.3 Prvá implementácia

Začal som implementáciou triedy *JsonMerger*. Táto trieda v sebe zapuzdruje funkcie a premenné potrebné pre vytvorenie json výstupu (súbor *fullGraph.json*), ktorý je generovaný počas už existujúcich prechodov AST. K už existujúcemu kódu na vytvorenie uzlov a hrán boost grafu sa priradili volania funkcií, ktoré pridali dané uzly a hrany do premennej obsahujúcej generovaný json v triede *JsonMerger*. Na obrázku 6.2 je možné vidieť, že generovanie tohto json formátu prebieha zároveň s prechodmi tried *Controls* a *Parsers*. V nasledujúcom texte sa budem odkazovať na funkcie a premenné triedy *JsonMerger*, ktorej diagram je na obrázku 6.1.

```
bool ControlGraphs::preorder(const IR::P4Table *table) {
    auto name = table->getName();

    auto v = add_and_connect_vertex(name, VertexType::TABLE);

    parents = {{v, new EdgeUnconditional()}};

    auto key = table->getKey();
    visit(key);

    Parents keyNode;
    keyNode.emplace_back(parents.back());

    Parents new_parents;

    auto actions = table->getActionList()->actionList;
    for (auto action : actions){
        parents = keyNode;

        auto v = add_and_connect_vertex(action->getName(), VertexType::ACTION);

        parents = {{v, new EdgeUnconditional()}};

        if (action->expression->is<IR::MethodCallExpression>()) {
            auto mce = action->expression->to<IR::MethodCallExpression>();

            // needed for visiting body of P4Action
            auto resolved = P4::MethodInstance::resolve(mce, refMap, typeMap);

            if (resolved->is<P4::ActionCall>()) {
                auto ac = resolved->to<P4::ActionCall>();
                if (ac->action->is<IR::P4Action>()) {
                    visit(ac->action->to<IR::P4Action>());
                }
            }
        }

        merge_other_statements_into_vertex();

        new_parents.insert(new_parents.end(), parents.begin(), parents.end());
        parents.clear();
    }

    parents = new_parents;

    return false;
}
```

Kód 4: Rozšírený prechod uzlom obsahujúcim tabuľku



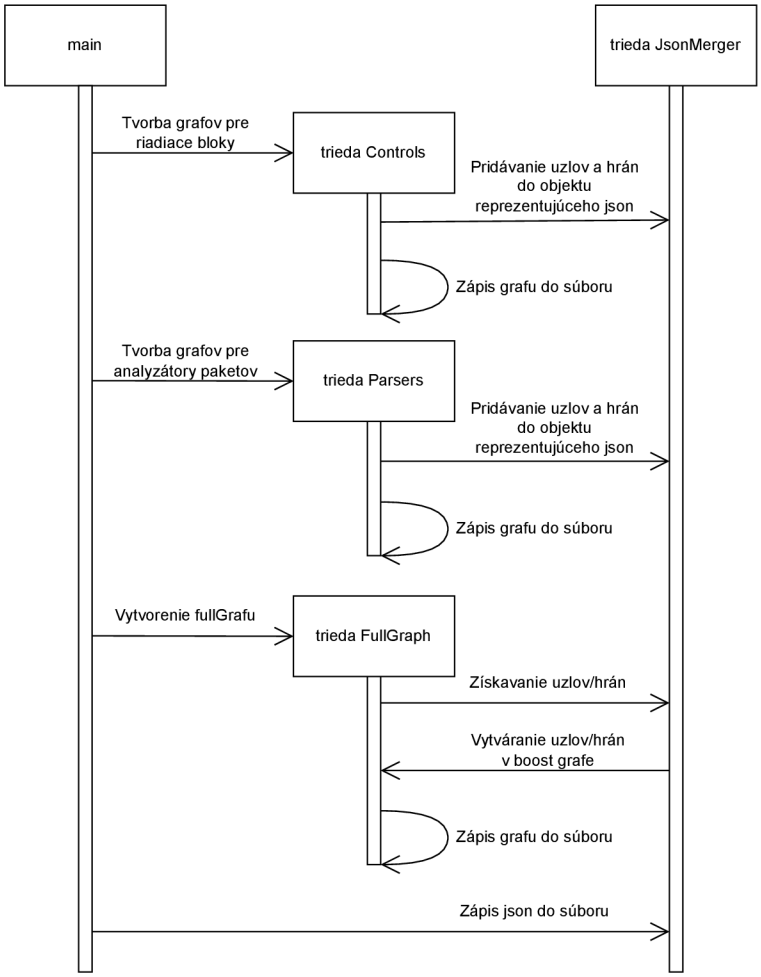
Obr. 6.1: Diagram triedy JsonMerger

V programe sa najskôr prechádzajú uzly v AST, ktoré obsahujú riadiace bloky. Tieto prechody sa nachádzajú v triede *Controls*. Na začiatku prechodu uzlom (vo funkcii `preorder(const IR::PackageBlock *block)`), obsahujúcim riadiaci blok, sa vždy vytvorí volaním funkcie `newBlock()` nový json objekt so zodpovedajúcim menom a typom `"control"`. V tejto funkcii sa novovytvorený objekt uloží na koniec poľa `nodes` v premennej `json` (ktorá je pomenovaná takto stroho, pretože obsahuje výsledný json, ktorý sa zapíše do súboru), a vytvoria sa zatiaľ prázdne json polia `nodes` a `transitions`. Pole `nodes` v novovytvorenom objekte obsahuje uzly CFG, predstavujúce samostatné bloky kódu obsiahnuté v danej riadiacej funkcii, ako aj podmienky (if, table-key, switch-expr), alebo názvy tabuliek a akcií. V poli `transitions` sú zase uložené všetky prechody medzi uzlami v poli `nodes`. Ďalej pracujem už iba s novovzniknutým objektom, ktorý je dostupný skrz premennú `block`.

Nasleduje vloženie uzlov `__START__` a `__EXIT__` do poľa `nodes`, volaním funkcie `addNode()`. Uzol `__START__` predstavuje vstupný bod toku programu do riadiaceho bloku, a uzol `__EXIT__` predstavuje bod, ktorým tok programu vždy riadiaci blok opúšťa. Čiže ak je telo riadiaceho bloku prázdne, čo sa zistí až po prechode uzla obsahujúcim telo daného riadiaceho bloku, boost graf aj objekt `block` obsahujú vždy iba tieto dva uzly.

Po vytvorení týchto uzlov v objekte `block` a boost podgrafe sa pokračuje prechodom tela riadiaceho bloku. Jednotlivé prechádzané uzly z AST, ktoré predstavujú samostatné príkazy, nasledujúce za sebou a nespôsobujúce vetvenie programu sú spájané do jedného uzla v CFG. Nové uzly sa pridávajú do CFG až keď sa narazí prechodmi AST na uzol, ktorý spôsobuje vetvenie programu. Takýto uzol predstavuje podmienku, prepínač alebo tabuľku. Nové uzly sa automaticky spájajú s už existujúcimi predchádzajúcimi uzlami v CFG pridaním zodpovedajúcich hrán. Pre správne prepojenie týchto uzlov sa používa premenná typu `Parent`, ktorá obsahuje id (číslo) predchádzajúceho uzlu a text predstavujúci podmienku alebo iný predpoklad, pri ktorého platnosti sa prechádza danou hranou. Po prejdení všetkých príkazov nachádzajúcich sa v tele riadiaceho bloku sa pokračuje na prechod uzlov AST, ktoré predstavujú analyzátory paketu.

Grafy vo formáte dot pre analyzátory paketov sa nevytvárajú za pomoci knižnice boost. Pridávanie uzlov do `json` v triede *Parsers* prebieha vo funkcii `postorder(const IR::P4Parser *parser)`, kde sa na začiatku znovu vytvorí nový json objekt so zodpovedajúcim názvom a typom, odkaz na neho sa uloží do `block`, a pridá na začiatok poľa `nodes` v premennej `json` volaním funkcie `newBlock()`. Potom sa iteruje stavmi analyzátora paketu, uloženými v mape `states`. Volaniami funkcií `addNode()` a `addTypeToNode()` sa stavy pri-



Obr. 6.2: Sekvenčný diagram zachytávajúci prechod jednotlivými triedami v prvej implementácií.

dajú do objektu v premennej `block`. Zároveň sa stavy pridávajú do pomocnej mapy **nodes**, kde sa mapuje názov stavu na jeho index v poli `nodes` v premennej `block`. Táto mapa je potrebná pre vytvorenie hrán v očakávanom formáte v objekte `block`, kde sa na uzly odkazuje číslom, reprezentujúcim ich index v poli `nodes`, a nie skrz názvy jednotlivých uzlov. Následne sa iteruje hranami analyzátora paketu, uloženými v mape `transitions`. Každá hrana v sebe nesie informáciu o názve uzla, v ktorom začína a v ktorom končí. Tu sa využije mapa **nodes**, z ktorej sa získa index uzla z ktorého hrana začína, a v ktorom končí, skrz názvy jednotlivých uzlov. Hrana sa pridá do objektu `block` volaním `addTransition()`.

Pre vytvorenie fullgrafu vo formáte dot sa používajú funkcie z boost knižnice a objekt `json` z triedy *JsonMerger*. Z objektu `json` sa získajú všetky prvky v poli `nodes`, predstavujúce grafy jednotlivých funkčných blokov v programe. Týmito objektmi sa následne iteruje:

1. Vytvorí sa nový podgraf s názvom zodpovedajúcim názvu funkčného bloku.
2. Následne sa iteruje uzlami objektu z poľa `nodes`, a tie sa postupne pridávajú do podgrafu.

3. Potom sa iteruje hranami objektu z poľa `transitions`, ktoré sa postupne pridávajú do podgrafu.

4. Na konci sa vytvorí prepojenie medzi aktuálnym a predchádzajúcim podgrafom, kde sa spojí posledný (najnižší) uzol z predchádzajúceho podgrafu s prvým (najvyšším) uzlom z aktuálneho podgrafu (bude vysvetlené nižšie).

Ak aktuálny objekt s funkčný blokom obsahuje iba 2 uzly (uzly `__START__` a `__EXIT__`), je preskočená iterácia pre uzly a hrany v objekte, a pridá sa do podgrafu iba jeden uzol s názvom `Empty body`. Tento uzol bude slúžiť iba na prepojenie po sebe idúcich podgrafov a vo výslednom grafe nebude vykreslený. Je to z dôvodu sprehľadnenia výsledného fullgrafu – podgrafy, v ktorých sa nevykoná žiadny kód, sú prázdne.

Prepojenie jednotlivých podgrafov funguje tak, že sa pridá hrana, ktorá akoby spája jednotlivé podgrafy (v skutočnosti ale spája uzly z týchto podgrafov). Používajú sa tu pomocné premenné:

- `t_adder` – Predstavuje súčet uzlov vo všetkých predošlých podgrafoch (bez uzlov v aktuálnom podgrafe). Dá sa povedať, že reprezentuje index uzla `__START__` v aktuálnom podgrafe.

- `t_prev_adder` – Predstavuje index posledného uzla v predchádzajúcom podgrafe (teda uzla `__EXIT__` alebo `accept`). Táto premenná je potrebná, pretože posledný uzol v predchádzajúcom podgrafe neleží na poslednom indexe v tomto podgrafe, nie je možné ho teda získať výpočtom `t_adder - 1`.

- `i` – Používa sa na indexáciu podgrafov a je tiež potrebná pre správne vytvorenie hrán medzi podgrafmi. Za jej pomoci sa nastavujú atribúty týchto hrán.

Pri vytváraní tohto prepojenia môže nastať viacero situácií:

- Spája sa predchádzajúci podgraf analyzátora paketu a súčasný podgraf riadiaceho bloku. Pridá sa hrana začínajúca v uzle `accept` predchádzajúceho podgrafu, ktorého index získame výpočtom `t_adder - 2`, a koncom v uzle `__START__` súčasného podgrafu, ktorého index predstavuje premenná `t_adder`.

- Spájajú sa dva podgrafy riadiacej funkcie. Pridá sa hrana so začiatkom v uzle `__EXIT__` predchádzajúceho podgrafu, ktorého index máme uložený v premennej `t_prev_adder`, a koncom v uzle `__START__` súčasného podgrafu, ktorého index predstavuje premenná `t_adder`.

Hrana, ktorá je pridaná medzi dva po sebe idúce podgrafy musí mať naviac nastavené atribúty `lhead` a `ltail`. Atribút `lhead` určuje kde sa vykreslí koniec danej hrany, a nastavuje sa na rám aktuálneho podgrafu a atribút `ltail` určuje kde sa vykreslí začiatok danej hrany, a nastavuje sa na rám predchádzajúceho podgrafu.

Tento spôsob generovania json a fullgrafu v dot formáte interne generoval json aj ak nebol zvolený ako výstup skrz argumenty. Preto som sa snažil nájsť iný spôsob generovania fullgrafu. Ponúkalo sa ukladanie boost podgrafov z prechodu uzlami AST obsahujúcimi riadiace funkcie a aj analyzátory paketu, a ich následné spracovanie podľa argumentov z príkazovej riadky – buď sa vytvorí iba fullgraf v dot formáte, alebo iba v json formáte, alebo sa vytvoria grafy pre všetky funkčné bloky rovnako ako tomu bolo doteraz.

Taktiež, keď sa na to pozriem spätne, tak sa ponúka riešenie vytvárať iba interne json, a na základe argumentov z neho vytvoriť boost grafy pre každý funkčný blok (default

behavior), alebo fullgraf. Túto cestu som si nezvolil, pretože som chcel meniť čo najmenej pôvodného kódu, a nie len zachovať pôvodnú funkcionalitu, ale aj čo najviac pôvodný kód.

6.4 Druhá implementácia

Z dôvodov spomenutých na konci predchádzajúcej sekcie som sa rozhodol pre prerobenie prvej implementácie. Zmenil som triedu *Controls* tak, aby ukladala boost grafy pre riadiace bloky programu do vektoru `controlGraphsArray`. Triedu *Parsers* som upravil tak, aby na generovanie grafov pre analyzátory paketu používala boost knižnicu. To znamenalo len zmenu smyčiek, v ktorej sa uzly a hrany vypisovali priamo do súboru na smyčky, v ktorých sa pridávajú uzly a hrany do boost grafu. Využila sa tu aj mapa **nodes** z prvej implementácie, pretože uzly v boost grafe sú bez zadania mena ako argumentu pomenované ich indexom – číslo, ktoré vyjadruje koľký v poradí je daný uzol vytvorený. Pre správne pridanie hrán sa teda pracuje s mapou na získanie zodpovedajúceho indexu hrany. Vytvorený boost graf sa znovu uložil do vektoru `parserGraphsArray`.

V prvej implementácií ešte neboli spomínané a ani pridané nové argumenty programu najmä z dôvodu, že bola zavrhnutá. Avšak, ak by v prvej implementácií boli, ich funkcionalita by bola rovnaká ako tu. Pridané argumenty, sú popísané nižšie.

- --graphs – Vypíše CFG pre každý funkčný blok do súboru s názvom zodpovedajúcim názvu funkčného bloku. Zodpovedá pôvodnému správaniu riešenia generujúceho CFG.
- --fullGraph – Pospája CFG jednotlivých funkčných blokov do jedného CFG. Zapíše ho vo formáte dot do súboru s názvom *fullgraph.dot*.
- --jsonOut – Prevedie CFG jednotlivých funkčných blokov do formátu json. Presná schéma tohoto formátu bola popísaná v sekcii 6.1.

Ak pri spustení programu nie je zadaný žiadny z týchto argumentov, program sa správa akoby bol spustený s argumentom --graphs.

Grafy uložené vo vektoroch `controlGraphsArray` a `parserGraphsArray` sa ďalej spracúvajú vo funkcii vo funkcii `process` triedy *Graph_visitor*. V tejto funkcii sa vykoná kód podľa zadaných vstupných argumentov programu.

V prípade argumentu --graphs sa iba zapíšu jednotlivé grafy uložené vo vektoroch do súborov vo formáte dot, s názvami zodpovedajúcimi názvom grafov.

Pri zadaní argumentu --fullGraph sa vytvára fullgraf vo formáte dot. V tejto časti používam štruktúru `fullGraphOpts` (kód 5) s položkami:

- `fg` – Predstavuje miesto kde sa ukladá výsledný graf. V ňom sa pri každej iterácií vytvorí nový podgraf, a cez funkciu `boost::copy_graph` sa do neho skopíruje už existujúci graf.
- `node_i` – Indexuje uzly vo výslednom grafe.
- `cluster_i` – Indexuje jednotlivé podgrafy výsledného grafu.

Táto štruktúra slúži na predanie dát, medzi dvoma volaniami funkcie `forLoopFullGraph`, prvým pre grafy analyzátorov paketu, a druhým pre grafy riadiacich funkcií.

Vytváranie boost fullgrafu vo formáte dot prebieha vo funkcii `forLoopFullGraph`. V nej sa iteruje vektorom s už vytvorenými grafmi. V každej iterácií sa najskôr vytvorí nový

```
struct fullGraphOpts{
    Graph fg;  // fullGraph

    /* variables needed for subgraph connection*/
    unsigned long node_i = 0;  // node indexer
    unsigned cluster_i = 0;  // cluster indexer
};
```

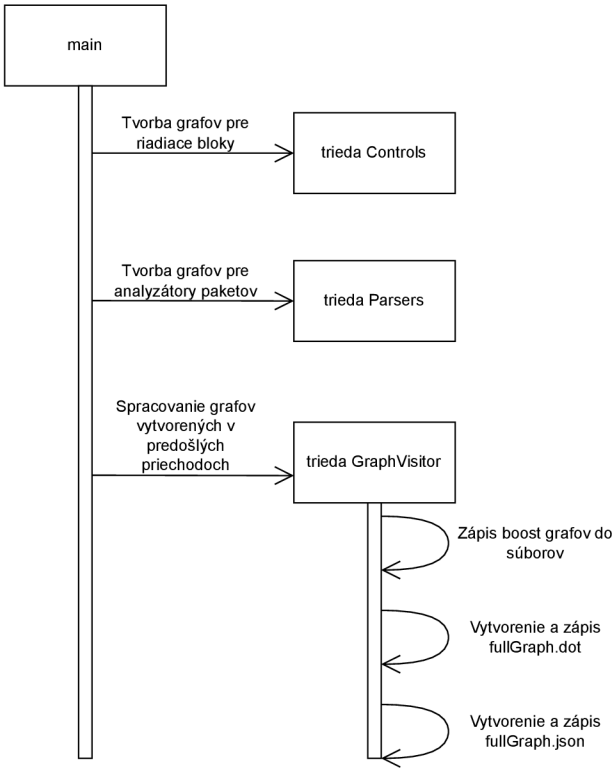
Kód 5: Štruktúra fullGraphOpts

podgraf v grafe `fg` a nastavia sa mu potrebné atribúty a názov, pre ktorého vytvorenie sa využije pomocná premenná na indexáciu podgrafov `cluster_i`. Následne sa vykoná kód podľa počtu uzlov v kopírovanom grafe Ak kopírovaný graf obsahuje iba dva uzly, je do podgrafu pridaný iba jeden uzol – tak ako pri prvej implementácií, na sprehľadnenie výsledného fullgrafu. Inak sa skopíruje celý graf do podgrafu pomocou funkcie `boost::copy_graph`. Kvôli nefungujúcemu kopírovaniu názvov hrán bola vytvorená štruktúra `edge_name_copier`, ktorej úlohou je kopírovať názvy hrán z pôvodného grafu do podgrafu. Táto štruktúra sa predáva ako jeden z argumentov funkcie `boost::copy_graph`. Následné spájanie jednotlivých podgrafov je realizované rovnako ako v prvej implementácií s tým, že namiesto pomocnej premennej `t_adder` sa používa premenná `node_i`, a pracuje sa grafmi riadiacich blokov, namiesto json objektov.

Po zadaní argumentu --jsonOut sa vytvorí fullgraf v preddefinovanom json formáte. Jeho tvorba prebieha vo funkcii `forLoopJson`, v ktorej sa znovu iteruje cez grafy uložené vo vektoroch. Pre každý graf sa vytvorí nový objekt, do ktorého sa pridá názov a typ spracovávaného grafu. Ďalej sa iteruje skrz uzly v grafe, ktoré sú postupne pridávané ako json objekty do poľa `nodes` so zodpovedajúcim názvom a typom. Okrem iného sa do týchto objektov ukladá aj index daného uzla v poli, do ktorého bude pridaný, a číslo, ktoré zodpovedá typu uzla (enum). Následne sa iteruje skrz hrany v grafe, získavajú sa informácie z ktorého uzla hrana ide, a do ktorého ide, a taktiež popis hrany (ktorý zväčša vyjadruje podmienku, preto sa položka aj volá `cond`). Tieto informácie sú postupne pridané ako json objekty do poľa `transitions`.

Táto implementácia je jednoduchšia na čítanie a zaberá menej riadkov kódu. Z jej sekvenčného diagramu 6.3 je možné vidieť, že zároveň pracuje priamočiarejšie ako prvá implementácia. Medzi jej nepochybnú výhodu patrí prepísanie triedy *Parsers* tak, aby používala funkcie boost knižnice na tvorbu grafov. Predtým sa graf generoval skrz textový výstup priamo do súboru, čo bolo síce možné, ale kód vďaka tomu nepôsobil ucelene, v jednej triede sa na generovanie grafov používal boost, a v druhej nie. Alternatívne by šlo aj triedu *Controls* prepísať tak, aby generovala graf rovno do súboru bez použitia boostu, ak bolo účelom nepoužívania boost v triede *Parsers* zjednodušenie pamäťovej náročnosti. Výhoda znovupoužitia grafov pre jednotlivé funkčné bloky je v tom, že ich môžme skopírovať jednou funkciou do podgrafu výsledného fullgrafu. Pokúšal som sa aj nejako tieto grafy pospájať, čím by sme nemuseli generovať nový graf pre fullgraf a ušetriť na pamäťovej náročnosti, ale neúspešne. Nie je možné len pridať hranu, ktorá by spájala tieto grafy, alebo jeden celý graf premiestniť do druhého.

Vo výsledku táto implementácia vytvorila menej tried – iba jednu (*Graph_visitor*), ktorá pracuje s boostovskými grafmi, namiesto 2 v predošlej implementácií, kde sa vytvorila trieda *JsonMerger*, ktorá slúžila na vytváranie json výstupu a trieda *Full_graphs*, ktorej úlohou bolo vytvoriť fullgraf. Toto považujem za výhodu, kód je viac kompaktný.



Obr. 6.3: Sekvenčný diagram zachytávajúci prechod jednotlivými triedami v druhej implementácií.

6.5 Problémy s vytváraním fullgrafu

Pri prvotnej snahe vytvárať fullgraf spájaním jednotlivých podgrafov som narazil na nasledujúci problém. Pri použití základného **dot** engine nie je možné spojovať 2 podgrafy. Je to možné iba pri použití vykresľovacieho enginu **fdp**, ktorý je ale určený pre neorientované grafy. Jeho použitie pre vykreslenie orientovaného grafu viedlo k vzniku neprehľadného grafu, v ktorom neboli uzly na rovnakej úrovni usporiadané vedľa alebo pod sebou. Vykresľovací engine **fdp** jednoducho ignoruje, že ide o orientovaný graf, a vykreslí ho ako neorientovaný. Preto som tento problém vyriešil spájaním posledného uzla a prvého uzla v dvoch po sebe nasledujúcich podgrafoch. Princíp tohto spájania podgrafov bol už popísaný vyššie, v sekcii 6.3.

Ďalej som pre tvorbu fullgrafu zvažoval ako ho urobiť čo najčitateľnejším. Práve z tohto dôvodu vzniklo rozhodnutie nevykresľovať v grafoch prázdnych riadiacich blokov uzly `__START__` a `__EXIT__`. Vhodnejšie je nahradiť ich jedným uzlom, ktorý sa vykreslí ako neviditeľný použitím atribútu *invis*. Zvažoval som tiež pre každú tabuľku vytvoriť nový podgraf, čím by bol jasne vymedzený tok programu, ktorý prebieha v tabuľke. Túto ideu som ale zavrhol z dôvodu, že pri použití funkcie `boost::copy_graph` sa nekopírujú podgrafy zanorené v podgrafoch.

6.6 Použité knižnice

Na tvorbu výstupu vo formáte json som použil knižnicu prítomnú v p4c repozitári. Na tvorbu CFG som použil boost knižnicu, pretože tá sa už používala na tvorbu grafov v súčasnom riešení, a bola pre ňu vytvorená v triede *Graphs* sada funkcií, ktoré uľahčovali prácu s ňou.

Kapitola 7

Výsledky

Implementované riešenie bolo otestované na sade testovacích programov. Vygenerované výstupy pre tieto programy, ako aj programy samotné sa nachádzajú na priloženom médiu. Tieto výstupy boli ručne kontrolované, čo znamená, že každý graf v dot formáte bol ručne prevedený nástrojom dot do pdf/png formátu. Po tomto prevedení mohli byť výsledné grafy skontrolované, či sú validné a odpovedajú očakávaniu. Preukázala sa funkčnosť riešenia, a všetky grafy odpovedali očakávaniu. Výstupy vo formáte json nebolo potrebné nejako špecificky kontrolovať.

Na obrázku 7.1 je možné vidieť vygenerovaný graf pre rovnaký príkladový P4 program ako bol použitý už v kapitole 5 (obrázok 5.1). V tomto obrázku je možné si všimnúť hlavne pridaný uzol s kľúčom tabuľky (*lpm: "hdr.ipv4.dst_addr" exact: "hdr.ipv4.src_addr"*) a uzly s akciami *drop_action* a *to_port_action*. Z uzlov obsahujúcich jednotlivé akcie vedú hrany do uzlov, ktoré reprezentujú telo akcie. V tomto prípade ide o jeden uzol, ale týchto uzlov môže byť bežne viac. Tok programu sa po vynorení z jednotlivých akcií musí spojiť vždy v jednom uzle (v zobrazenom príklade ide o uzol s podmienkou *dropped_0/dropped;*).

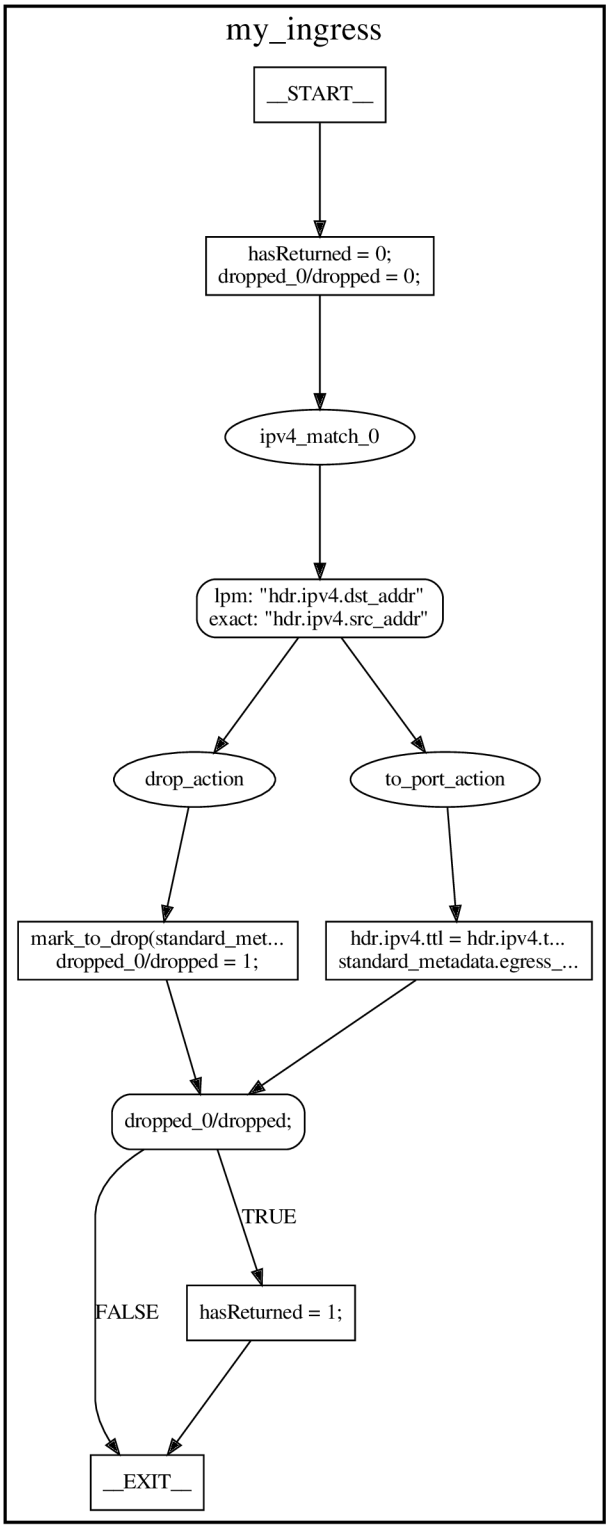
Pre demonštráciu práce s json výstupom som vytvoril krátky skript v jazyku Python, ktorý vytvára z jsonu fullgraf v dot formáte. Má viacero vstupných argumentov, ktorých popis je možné získať zadaním argumentu --help. Poskytujú kozmetické úpravy fullgrafu, ktoré menia spôsob, akým sú niektoré prvky vykresľované. Asi najzaujímavejším argumentom je --LR, ktorý mení vykresľovanie jednotlivých uzlov a hrán v grafe zo štandardného zhora dole, na zľava doprava. Tento skript je dostupný v súboroch priložených na médiu, a nebol pridaný do oficiálneho repozitára p4c.

Za účelom vytvorenia pull-requestu bola vytvorený kópia oficiálneho repozitára prekladača p4c. Sem bolo pridané moje riešenie s vhodnými commit správami. V commitoch som tiež aktualizoval Readme daného backendu, kde som pridal popis nového json výstupu a popísal, ako používať nové argumenty programu, aktualizoval ukážkové obrázky grafov a pridal fullgraf vo formáte png a json pre program *firewall* a *flowlet_switching-bmv2*. Ďalej som pridal vhodné komentáre k mnou vytvorenému kódu a upravil kód tak, aby vyhovoval štýlu písania kódu používaného v p4c repozitári, kde sa používa *Google coding style*[1]. Pre kontrolu dodržania pravidiel, ktoré *Google coding style* vyžaduje je v repozitári p4c dostupný nástroj *cpplint*. Ním som svoj kód skontroloval a nevyhovujúce riadky kódu upravil. Po finálnych úpravách nasledoval pull-request[2]. Recenzent odhalil drobné chyby v mojom

[1] https://google.github.io/styleguide/cppguide.html
[2] https://github.com/p4lang/p4c/pull/3268

riešení, ktoré som opravil. Taktiež som musel upraviť zostavovacie súbory tak, aby fungovalo automatizované zostavovanie. Musel som upraviť najmä zostavovací súbor pre *bazel*, ten som lokálne u seba na zostavovanie nepoužíval, a tiež pridať niektoré hlavičkové súbory, bez ktorých mi síce preklad fungoval lokálne, ale pre automatizovaný preklad chýbali. Riešenie bolo po týchto úpravách schválené recenzentom a pridané do oficiálneho repozitára[3].

[3] https://github.com/p4lang/p4c/commit/c6bd86ffe8d1d4d9fb3d8c34caa1c0291c11f1f7



Obr. 7.1: Graf pre riadiaci blok *my_ingress* vygenerovaný novým riešením

Kapitola 8

Záver

Cieľom tejto práce bolo rozšíriť existujúci referenčný open source prekladač jazyka P4 (p4c), konkrétne jeho backend na generovanie CFG (*p4c-graphs*). Pre pochopenie k čomu jazyk P4 slúži, bolo potrebné naštudovať si technológie a myšlienky, ktoré viedli k jeho vzniku. Preto som začal získavaním informácií o myšlienke SDN, jej prvej implementácií, protokolom OpenFlow, a ďalšom rozvoji tejto myšlienky, vedúcemu k vzniku jazyka P4. Práca ďalej obsahuje stručný rozbor špecifikácie jazyka P4 a spôsobu, ako sa pomocou neho programujú sieťové zariadenia. Moja práca sa bližšie týka dostupného open source prekladača p4c, ktorého popisu sa preto venujem podrobnejšie. Ďalej bolo potrebné naštudovať si, čo je to graf riadenia toku a zistiť jeho význam pri optimalizácií prekladačov.

Pre vytvorenie návrhu rozšírenia existujúceho riešenia generujúceho CFG bolo nutné naštudovať si IR programu používanú p4c prekladačom, ktorá má formu AST. Následne som sa zameral na pochopenie súčasného riešenia a zisťoval, ako prebieha a čím končí analýza tabuľky v triede *Controls*. Na základe znalosti IR som navrhol, ktoré prvky je nutné ďalej analyzovať. Pre tieto prvky bolo nutné implementovať vyhľadanie a prechod uzlami v AST, v ktorých sa nachádzajú. Ďalej bola navrhnutá funkcionalita spojiť jednotlivé grafy riadiacich prvkov programu do jedného grafu, ktorý by zachytával celý tok programu. Tiež bolo navrhnuté vytvorenie výstupu vo formáte json, ktorý bude reprezentovať novovytvorený graf. Pre túto funkcionalitu bola vytvorená predbežná signatúra tried a funkcií, ktorá ale bola počas implementácie značne zmenená.

Výstupom práce je rozšírenie funkcionality existujúceho backendu *p4c-graphs*. Bol rozšírený o pridanie prechodov uzlami, ktoré obsahujú kľúč a akcie danej tabuľky. Pridaná bola tiež možnosť generovať graf vo formáte dot, ktorý zachytáva celý tok programu, a json reprezentácie tohto grafu. Nové riešenie teda zachováva funkcionalitu pôvodného riešenia, a rozširuje ju o všetku navrhnutú funkčnosť.

Výsledné riešenie bolo otestované na viacerých programoch v jazyku P4. Bol vytvorený pull-request do oficiálneho repozitára prekladača p4c, ktorý bol schválený, a riešenie bolo teda prijaté.

Pokračovaním tejto práce by mohla byť úpráva vykresľovania tabuliek bez kľúčov, kde sa vždy spúšťa iba akcia označená ako *default_action*. Sem by bolo dobré zistiť, či výsledný zostavený program maže akcie, ktoré sú kvôli neprítomnosti kľúča nedostupné, alebo či sa dá cez kontrolér nejako meniť *default_action*. Podľa výsledkov tohto bádania by sa mohol optimalizovať samotný preklad, ako aj tvorba CFG. V ďalších smeroch by sa mohla upraviť tvorba uzlov pre prepínacie konštrukcie a zvážiť nové úpravy, ktoré by sprehľadnili výsledný fullgraf.

Literatúra

[1] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N. et al. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* New York, NY, USA: Association for Computing Machinery. jul 2014, zv. 44, č. 3, s. 87–95. DOI: 10.1145/2656877.2656890. ISSN 0146-4833. Dostupné z: https://doi.org/10.1145/2656877.2656890.

[2] Chapman, D. B., Zwicky, E. D. a Russell, D. *Building Internet Firewalls*. 1st. USA: O'Reilly & Associates, Inc., 1995. ISBN 1565921240.

[3] Consortium, T. P. L. *P4_16 Language Specification* [online]. [cit. 2021-12-30]. Dostupné z: https://p4.org/p4-spec/docs/P4-16-v1.2.2.html.

[4] Cooper, K. D. a Torczon, L. Chapter 5 - Intermediate Representations. In: Cooper, K. D. a Torczon, L., ed. *Engineering a Compiler (Second Edition)*. Second Edition. Boston: Morgan Kaufmann, 2012, s. 221–268. DOI: https://doi.org/10.1016/B978-0-12-088478-0.00005-0. ISBN 978-0-12-088478-0. Dostupné z: https://www.sciencedirect.com/science/article/pii/B9780120884780000050.

[5] Foundation, O. N. Software-defined networking: the new norm for networks. *ONF White Paper*. 2012, zv. 2, s. 2–6.

[6] Hauser, F., Häberle, M., Merling, D., Lindner, S., Gurevich, V. et al. A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. *CoRR*. 2021, abs/2101.10632. Dostupné z: https://arxiv.org/abs/2101.10632.

[7] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L. et al. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* New York, NY, USA: Association for Computing Machinery. mar 2008, zv. 38, č. 2, s. 69–74. DOI: 10.1145/1355734.1355746. ISSN 0146-4833. Dostupné z: https://doi.org/10.1145/1355734.1355746.

[8] Mikhailov, A., Hmelnov, A., Cherkashin, E. a Bychkov, I. Control flow graph visualization in compiled software engineering. In:. Máj 2016, s. 1313–1317. DOI: 10.1109/MIPRO.2016.7522343.

[9] NOX contributors collective of. *NOX OpenFlow controller* [https://github.com/noxrepo/nox]. GitHub, 2021.

[10] P4 contributors collective of. *P4 reference compiler* [https://github.com/p4lang/p4c]. GitHub, 2021.

[11] P4 CONTRIBUTORS collective of. *P4 reference compiler* [https://github.com/p4lang/p4c/blob/main/docs/parsersUnroll-readme.md]. GitHub, 2021.

[12] PAROL, P. *P4 Network Programming Language—what is it all about?* [online]. codilime, 2021 [cit. 2021-12-30]. Dostupné z: https://codilime.com/blog/p4-network-programming-language-what-is-it-all-about/.

[13] SALISBURY, B. *The Control Plane, Data Plane and Forwarding Plane in Networks* [online], 27. septembra 2012. Dostupné z: http://networkstatic.net/the-control-plane-data-plane-and-forwarding-plane-in-networks/.

[14] SHVETS, A. *Visitor design pattern* [online]. refactoring.guru, 2022 [cit. 2022-4-25]. Dostupné z: https://refactoring.guru/design-patterns/visitor.

[15] XIA, W., WEN, Y., FOH, C. H., NIYATO, D. a XIE, H. A Survey on Software-Defined Networking. *IEEE Communications Surveys Tutorials*. 2015, zv. 17, č. 1, s. 27–51. DOI: 10.1109/COMST.2014.2330903.

[16] ZIMMERMANN, H. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*. 1980, zv. 28, č. 4, s. 425–432. DOI: 10.1109/TCOM.1980.1094702.