VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

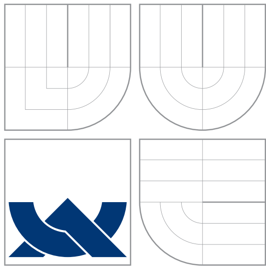
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

DYNAMICKÁ ANALÝZA POUŽITÍ KNIHOVNÍCH VOLÁNÍ

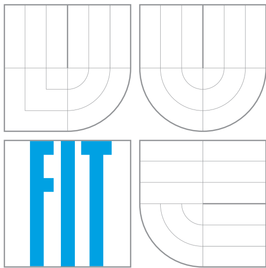
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE VIKTOR MALÍK
AUTHOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

DYNAMICKÁ ANALÝZA POUŽITÍ KNIHOVNÍCH VOLÁNÍ

DYNAMIC ANALYSIS OF PROGRAMS USING LIBRARY CALLS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — VIKTOR MALÍK
AUTHOR

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

BRNO 2014

Abstrakt
Tato bakalářská práce se zabývá vývojem dynamického analyzátoru, který sleduje používání knihovních volání analyzovaným programem. Analyzátor dále tato volání automaticky ovládá za účelem vytváření různých běhů programu, které pak agreguje do výsledného grafu toku řízení. Pro sledovaní a ovládání volání používá analyzátor vlastní sdílenou knihovnu pro operační systém GNU/Linux. Součástí práce je jak podrobný návrh celé aplikace, tak i její implementace v jazycích C/C++ zaměřující se na sledování standardních knihovních volání nad souborovým systémem.

Abstract
The objective of this bachelor's thesis is development of dynamic software analysis which monitors library calls of analysed program. The proposed analyser doubles library call routines in order to create different program runs. These runs are then aggregated into a single control flow graph which can be used for subsequent program analysis. Monitoring and controlling the calls is realised via stubs and wrappers encapsulated within a dynamic shared library for GNU/Linux operating system. The proof of concept is shown on dynamic analyser focused on file system library calls.

Klíčová slova
analýza software, dynamická analýza, knihovní volání, sdílená knihovna, dynamické zavádění/načítání

Keywords
software analysis, dynamic analysis, library calls, dynamic shared library, dynamic linker/loader

Citace
Viktor Malík: Dynamická analýza použití knihovních
volání, bakalářská práce, Brno, FIT VUT v Brně, 2014

Dynamická analýza použití knihovních volání

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Aleše Smrčky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Viktor Malík
20. mája 2014

Poděkování

Veľmi rád by som poďakoval vedúcemu mojej práce Ing. Alešovi Smrčkovi, Ph.D. za jeho skvelé nápady a rady pri riešení problémov počas tvorby práce.

© Viktor Malík, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technológií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Rozbor témy a opis existujúcich mechanizmov **4**
2.1 Metódy sledovania správania programov 4
2.1.1 Statická analýza 4
2.1.2 Dynamická analýza 5
2.1.3 Možnosti dynamickej analýzy 5
2.2 Monitorovanie a ovládanie knižnicových volaní 6
2.2.1 Knižnice v operačnom systéme Linux 6
2.2.2 Monitorovanie volaní pomocou vlastnej knižnice 6
2.3 Špecifikácia požiadavok pre dynamický analyzátor 7

3 Návrh dynamického analyzátora **9**
3.1 Štrukturálny opis 9
3.1.1 Štruktúra aplikácie 9
3.1.2 Štruktúra zdieľanej knižnice 10
3.1.3 Štruktúra riadiaceho programu 10
3.2 Opis interakcie 14
3.2.1 Komunikačný protokol 14
3.2.2 Interakcia medzi triedami riadiaceho programu 17
3.3 Opis správania analyzátora 19
3.3.1 Činnosť zdieľanej knižnice 19
3.3.2 Hlavný riadiaci algoritmus 19
3.3.3 Spúšťanie analyzovaného programu 20
3.3.4 Stopovanie analyzovaného programu 20
3.3.5 Algoritmus plánovania behov 20
3.3.6 Algoritmus vytvárania výsledného grafu 21
3.3.7 Detekcia skokov 22
3.3.8 Tvorba výstupu 23

4 Implementácia analyzátora **24**
4.1 Implementácia zdieľanej knižnice 24
4.1.1 Komunikačná časť 24
4.1.2 Zoznamy sledovaných volaní 24
4.1.3 Implementácia obaľovacích funkcií pre sledované volania 25
4.2 Implementácia riadiaceho programu 26
4.2.1 Nastavenie parametrov analýzy 26
4.2.2 Komunikačný modul 28

4.2.3 Spustenie analyzovaného programu 28
4.2.4 Riadiaca časť 28
4.2.5 Implementácia hashovacej tabuľky 29
4.2.6 Zachytávanie a riešenie chybových stavov 30
4.2.7 Správa pamäte 30

5 Opis experimentov a interpretácia výsledkov 31
5.1 Analýza jednoduchého programu 31
5.2 Analýza cyklu 31
5.2.1 Nastavenie správnej dĺžky podprogramu 32
5.2.2 Nastavenie menšej dĺžky podprogramu 32
5.2.3 Nastavenie väčšej dĺžky podprogramu 33
5.3 Porovnanie rôznych druhov agregácie 33
5.3.1 Program pracujúci s jedným súborom 33
5.3.2 Program obsahujúci cyklus 34
5.3.3 Program pracujúci s viacerými súbormi 34
5.4 Dĺžka analýzy v závislosti na konfigurácií 34

6 Záver 36

A Adresárová štruktúra projektu 38

B Preklad a spustenie programu 40

C Grafy získané počas experimentov 42

Kapitola 1

Úvod

Analýza softwaru je praktickým nástrojom pri skúmaní existujúcich programov, ako aj pri tvorbe nových. Na rozdiel od testovania, umožňuje nielen odhaliť chyby, ktoré program obsahuje, ale takisto poskytuje vývojárovi množstvo užitočných informácií, ktoré môžu byť ďalej použité pri ladení, či optimalizácií programu. Tématika analýzy software zahŕňa obrovské množstvo rôznych postupov a techník, z ktorých každá slúži na niečo iné. Jednou z týchto techník je aj dynamická analýza, ktorej hlavná výhoda spočíva v tom, že nepotrebuje zdrojové texty programu (ktoré sú často nedostupné, alebo chránené licenciou), ale analýzu vykonáva iba spúšťaním a sledovaním daného programu.

Cieľom tejto práce bolo navrhnúť a implementovať automatický dynamický analyzátor, ktorý sleduje používanie vybraných knižnicových volaní testovaným programom. Analyzátor ďalej ovplyvňuje tieto volania za účelom získania rôznych behov programu a nakoniec agreguje behy do výsledného grafu toku riadenia. Implementácia analyzátora je zameraná na sledovanie volaní týkajúcich sa práce so súbormi zo štandardnej knižnice jazyka C, používanej v operačnom systéme Linux.

Nasledujúci text obsahuje podrobný opis jednotlivých fází vývoja analyzátora. Na začiatku je uvedený teoretický rozbor problému a náčrt existujúcich možností analýzy. Nasleduje detailný návrh celého systému a jeho jednotlivých komponentov. Ten zahŕňa opis štruktúry, správania a komunikácie medzi týmito komponentmi. Ďalšou časťou je opis implementácie analyzátora v jazykoch C/C++, ktorý je zameraný prevažne na vývoj tých častí návrhu, ktoré sú pre dané jazyky špecifické. Na záver je uvedený rozbor rôznych experimentov s vytvoreným analyzátorom, vrátane interpretácie získaných výsledkov.

Kapitola 2

Rozbor témy a opis existujúcich mechanizmov

Táto kapitola sa venuje teoretickému rozboru zadania práce. Keďže sa má jednať o *dynamickú analýzu*, prvá časť kapitoly všeobecne opisuje možnosti sledovania správania softwaru s tým, že najväčší dôraz kladie práve na možnosti dynamickej analýzy. Druhou časťou zadania je, že ide o analýzu *použitia knižnicových volaní*, preto je ďalšia sekcia tejto kapitoly venovaná možnostiam monitorovania týchto volaní. Na konci kapitola obsahuje špecifikáciu požiadavok pre vytváraný analyzátor.

2.1 Metódy sledovania správania programov

Analýza software sa využíva na sledovanie správania programu, prípadne na získanie užitočných informácií o programe. V podstate sa jedná o proces automatického odvodzovania vlastností správania určitého programu [3]. Tieto vlastnosti môžu zahŕňať tok dát, využitie pamäte, volanie funkcií apod. V rámci analýzy sa používa množstvo rôznych techník, z ktorých každá používa iný prístup a vedie ku skúmaniu iných vlastností. Podľa základnej povahy týchto techník sa analýza programov delí na dve hlavné časti - *statická analýza* a *dynamická analýza*.

2.1.1 Statická analýza

Statická analýza je založená na skúmaní zdrojového textu programu a nedochádza pri nej k jeho spúšťaniu. Zahŕňa veľké množstvo techník, od jednoduchých, ako je získanie programových štatistík (napr. početnosť komentárov), až po rôzne zložitejšie, na sémantike založené metódy. Patrí medzi ne napríklad *analýza toku dát*, čo je proces získavania behových informácií o dátach daného programu, alebo *symbolická analýza*, ktorej cieľom je odvodenie presnej matematickej charakteristiky výpočtov v programe [5]. Všetky tieto techniky sú založené na princípe **dedukcie**, teda na odvodení konkrétnych behov od všeobecného zdrojového kódu [6].

Keďže statická analýza pracuje priamo so zdrojovým kódom (v niektorých prípadoch s objektovým kódom), na rozdiel od dynamickej, vždy obsiahne všetky možné behy programu. Často pri nej dochádza k vytvoreniu abstraktného modelu a ďalšej práci s týmto modelom [3]. To v niektorých prípadoch môže byť miernou nevýhodou statickej analýzy, keďže vtedy dôjde k určitej strate informácií [2].

Najčastejšie sa statická analýza využíva pri optimalizácií programov a pri dokazovaní ich správnosti [5].

2.1.2 Dynamická analýza

V prípade, že zdrojové súbory programu nie sú k dispozícií, (často sú napríklad chránené licenciou) využíva sa na analýzu daného programu **dynamická analýza**. Tá je na rozdiel od statickej založená na spúšťaní analyzovaného programu (väčšinou binárneho kódu). Môže sa jednať iba o jedno spustenie (napr. pri technikách určených na získanie štatistík o programe), alebo o opakovné spúšťanie analyzovaného programu. V takom prípade sú výsledné vlastnosti odvodené analýzou všetkých získaných behov [3].

Na rozdiel od statickej analýzy, ktorá obsiahne všetky možné behy programu, je tá dynamická limitovaná množinou reálne vykonaných behov. To môže znamenať v určitých prípadoch nevýhodu, keď je dynamická analýza chybná, pretože nepokryla všetky rôzne možnosti. Na druhej strane má táto skutočnosť aj svoju výhodu, keďže nikdy nedôjde k problému analýzy *falošných poplachov* (angl. *false positives*, niekedy aj *false alarms*), teda chybných varovaní analyzátoru o možnej chybe, ktoré sa môžu vyskytnúť v statickej analýze [2]. Ďalšou z výhod dynamickej analýzy je, že nie je nutné vytvárať nijakú abstrakciu, takže nemusí dôjsť ku strate informácií (hoci vo väčšine prípadov sa abstrakcia robí kvôli zjednodušeniu a urýchleniu) [2].

Keďže v tejto práci je použitá práve dynamická analýza, jej rôznym technikám sa venuje ďalšia kapitola.

2.1.3 Možnosti dynamickej analýzy

Hoci sa v rámci dynamickej analýzy používa veľké množstvo techník, je možné ich rozdeliť do troch základných skupín. Toto delenie uvažuje množstvo behov programu, ktoré daná technika používa pri analýze a prístup k vytváraniu týchto behov. Tieto tri skupiny spolu s **dedukciou**, používanou v statickej analýze, tvoria kompletné delenie analýzy software [6].

Analýza pozorovaním

Táto technika používa iba *jediný beh programu* a umožňuje preskúmať jeho ľubovoľné aspekty. Existuje veľké množstvo nástrojov využívajúcich danú techniku, väčšinou sa jedná o ladiace programy, tzv. „debuggery“. Ďalšími sú napríklad programy navrhnuté na kontrolu využívania pamäte, alebo kontrolu porušenia hraníc poľa [6].

Analýza indukciou

Indukcia je prechod od konkrétneho ku všeobecnému. V analýze sa využíva na *zhrnutie viacerých behov* do určitej formy abstrakcie (napríklad grafu). Táto technika pracuje s viacerými behmi programu získanými pomocou jeho opätovného spúšťania s rôznymi vstupmi. Využívajú ju napríklad nástroje na pokrytie kódu, ktoré spájajú príkazy a vetvy programu do výsledku, ktorý je možné vizualizovať [6].

Analýza experimentovaním

Hoci predchádzajúce metódy sú schopné sledovať správanie programu, ani jedna z nich nedokáže nájsť príčinu tohoto správania. Na to je nutné použiť posledný typ dynamickej analýzy, ktorý okrem toho, že používa viacero behov, tieto behy priamo *ovláda*. Vďaka

tomu dokáže vytvoriť sériu *experimentov* a pomocou nich *izolovať*, a tak nájsť príčiny daného správania [6].

Práve tento typ dynamickej analýzy je použitý v tejto práci. Všetky behy sú spúšťané s rovnakým vopred daným vstupom, ale existuje *ovládací program*, ktorý ich priamo ovplyvňuje. Tým je docielený vznik rôznych behov, ktoré sú ďalej spracovávané a spájané. Viac informácií o architektúre analyzátora sa nachádza v kapitole 3.

2.2 Monitorovanie a ovládanie knižnicových volaní

Po tom, čo sme zvolili vhodný typ analýzy programu, je nutné preštudovať, aké sú možnosti samotného sledovania zvolenej vlastnosti programu, v tomto prípade *knižnicových volaní*. V prvom rade je nutné stanoviť význam slova *knižnica* v operačnom systéme Linux a následne nájsť možnosť monitorovania a taktiež ovládania knižnicových volaní.

2.2.1 Knižnice v operačnom systéme Linux

Pod pojmom *knižnica* sa v programovaní rozumie samostatne preložiteľná jednotka obsahujúca implementáciu funkcií (väčšinou s podobnou funkcionalitou). Tieto knižnice môžu byť potom zdieľané medzi ostatných vývojárov a umožňujú tzv. *modulárne programovanie* (tj. výstavbu programu z modulov). Operačný systém Linux, pre ktorý je analyzátor v tejto práci tvorený, podporuje dva typy knižníc [4].

Statické knižnice sú také, ktoré sú ku programu, ktorý ich používa, pripojené počas prekladu. Využívajú sa hlavne pri malých programoch, kde obsahujú neveľké množstvo funkcií.

Dynamické knižnice sú načítané a pripojené až pri spúšťaní programu. Navyše sú tieto knižnice *zdieľané*, teda viacero programov ich môže využívať súčasne. Vďaka týmto vlastnostiam je urýchlený preklad programov a takisto sa výrazne zníži ich pamäťová náročnosť [4].

V systéme Linux existuje veľké množstvo zdieľaných knižníc, poskytujúcich rôznu funkcionalitu (napr. matematická knižnica). Najdôležitejšia a najpoužívanejšia z nich je **štandardná knižnica jazyka C**, vo väčšine systémov implementovaná pomocou *GNU C knižnice*, známej ako *glibc*. Opisu tejto knižnice sa venuje nasledujúca sekcia.

Štandardná knižnica jazyka C

Štandardná knižnica jazyka C obsahuje definíciu funkcií obaľujúcich *systémové volania* jadra Linuxu [1]. Poskytuje teda akési rozhranie pre komunikáciu s jadrom a zabezpečuje, že užívateľské procesy nikdy nevolajú systémové funkcie priamo z jadra, ale používajú volania definované práve v tejto knižnici. V implementácií analyzátora sú použité vybrané volania týkajúce sa práce so súbormi, ktoré obaľujú rovnomenné systémové volania.

2.2.2 Monitorovanie volaní pomocou vlastnej knižnice

V prípade, že chceme monitorovať knižnicové volania (napríklad z *glibc*), môžeme vytvoriť vlastnú zdieľanú knižnicu obsahujúcu práve tie funkcie, ktoré je záujem sledovať (prípadne ovládať). Následne je potrebné zabezpečiť, aby sa knižnica načítala a pripojila pred tou pôvodnou.

Vytvorenie vlastnej knižnice

Nová zdieľaná knižnica musí obsahovať funkcie s rovnakým rozhraním (hlavičkou), ako majú tie sledované. To zaručí, že ak sa daná knižnica načíta skôr ako pôvodná, zavolajú sa práve nové verzie funkcií.

V tele každej funkcie je potom možné zaznamenať jej volanie a zavolať pôvodnú funkciu, prípadne úplne nahradiť jej vykonanie (napr. simulovať vznik chyby). V prípade, že je požadované volať pôvodnú funkciu, je na to možné použiť jednu z funkcií dynamického načítavania, konkrétne `dlsym`.

Použitie `dlsym`

Funkcia `dlsym` umožňuje získať adresu symbolu (funkcie) nachádzajúceho sa v zdieľanom objekte. Jej prvým parametrom je deskriptor objektu a druhým meno daného symbolu. Namiesto prvého parametru je možné zadať hodnotu `RTLD_NEXT`, ktorá zaručí, že bude nájdený ďalší výskyt daného symbolu (tzn. ten z pôvodnej knižnice)[1]. Potom stačí zavolať túto získanú funkciu s požadovanými parametrami.

Načítanie a prilinkovanie knižnice

Pri spustení programu v Linuxe sa spusí dynamický zostavovací program (linker), ktorý načíta požadované zdieľané knižnice. Ich umiestnenie v systéme je uložené v systémovej premennej `LD_LIBRARY_PATH`. V našom prípade je však nutné, aby bola vytvorená knižnica načítaná skôr ako tá pôvodná. Na to je možné využiť ďalšiu premennú prostredia `LD_PRELOAD`. Objekty uvedené v tejto premennej sú načítané a prilinkované ešte pred tými z `LD_LIBRARY_PATH`. To znamená, že keď do `LD_PRELOAD` vložíme cestu k novej zdieľanej knižnici, funkcie z nej budú použité namiesto tých pôvodných[2].

2.3 Špecifikácia požiadavok pre dynamický analyzátor

Po tom, ako boli teoreticky vysvetlené mechanizmy, ktoré budú používané počas tvorby analyzátora, je ešte nutné špecifikovať, aké sú požiadavky na tento analyzátor (tj. čo analyzátor má a čo nemá robiť).

Jednotlivé požiadavky a ich časti sú uvedené v nasledujúcom zozname:

- dynamická analýza, teda analýza pomocou **spúšťania** testovaného programu
- **sledovanie** volania vybraných **knižnicových volaní** analyzovaným programom
  - jednoduché úpravy zoznamu sledovaných volaní
- **automatická** tvorba grafu toku riadenia analyzovaného programu
  - **opakované** spúšťanie analyzovaného programu za účelom získania jeho rôznych behov
    - automatické ovládanie zachytených knižnicových volaní tak, aby to viedlo k priechodu rôznymi vetvami programu

[1] `dlsym(3)`, http://linux.die.net/man/3/dlsym
[2] `ld.so(8)`, http://linux.die.net/man/8/ld.so

    * analyzátor nemení vstupy ani parametre spúšťania testovaného programu
  - agregácia získaných behov do **približného** grafu toku riadenia, ako kompaktnej štruktúry spájajúcej čiastočne abstraktné informácie z množstva behov programu
    * postupné spájanie behov do grafu počas priebehu analýzy
    * vyhľadávanie opakujúcich sa postupností volaní (cyklov a podprogramov) v grafe
  - možnosť nastavenia stupňa abstrakcie informácií použitého pre agregáciu
- analyzátor je určený pre operačný systém **GNU/Linux**

Kapitola 3

Návrh dynamického analyzátora

Pred začatím samotného vývoja software je veľmi dôležitou fázou vytvorenie kvalitného návrhu. Čím je návrh detailnejší a prepracovanejší, tým menej problémov bude nutné riešiť počas samotnej implementácie. Vďaka tomu nebude nutné často prerábať už vytvorený kód, čo programátorovi ušetrí množstvo času.

Táto kapitola obsahuje kompletný a podrobný návrh dynamického analyzátora. Je rozdelená do troch častí, keď každá z nich reprezentuje iný pohľad na program a opisuje ho z tohoto pohľadu.

3.1 Štrukturálny opis

Prvá časť sa zameriava na opis *statickej štruktúry* celej aplikácie. Pod štruktúrou sa rozumie rozdelenie aplikácie na časti, opis týchto častí a vzťahov medzi nimi. Sekcia ďalej opisuje štruktúru jednotlivých komponentov, ich podčastí, a tak ďalej, až ku základným, ďalej nedeliteľným častiam.

3.1.1 Štruktúra aplikácie

Ako bolo naznačené v sekcii 2.2.2, na to, aby bolo možné sledovať a ovládať knižnicové volania, je nutné vytvoriť vlastnú verziu zdieľanej knižnice obsahujúcu požadované funkcie. Preto prvou časťou analyzátora je práve táto **zdieľaná knižnica**.

Keďže analyzátor má pracovať na princípe *dynamickej analýzy experimentovaním*, je nutné, aby existoval proces, ktorý bude zodpovedný za vytváranie potrebných experimentov a takisto ukladanie a spracovanie získaných výsledkov. Na toto nie je možné využiť zdieľanú knižnicu, nakoľko jej činnosť je vždy po skončení behu analyzovaného programu takisto ukončená. Kvôli tomu musí existovať ďalší komponent analyzátora, konkrétne **riadiaci program**. Ten je zodpovedný za riadenie celého priebehu analýzy, ovládanie činnosti zdieľanej knižnice a takisto za zber výsledkov a ich spracovanie, prípadne vizualizáciu.

Komunikácia medzi riadiacim programom a zdieľanou knižnicou je realizovaná výmenou správ pomocou mechanizmu *lokálnych soketov systému Unix*[1]. Vždy keď je volaná jedna zo sledovaných funkcií nachádzajúcich sa v zdieľanej knižnici, táto zašle riadiacemu programu správu, ktorý dané volanie zaznamená a vygeneruje odpoveď, ktorú zašle naspäť. Detaily komunikačného protokolu opisujúceho jednotlivé správy sú opísané s sekcii 3.2.1.

[1] `unix(7)`, http://man7.org/linux/man-pages/man7/unix.7.html

Celková štruktúra analyzátora je zobrazená na obrázku 3.1. Ten zároveň ukazuje aj vzťah analyzátora a analyzovaného programu, keď je tento spúšťaný riadiacim programom s úpravou premennej prostredia `LD_PRELOAD` a vďaka tomu volá funkcie zo zdieľanej knižnice (viac v časti 2.2.2).

Obr. 3.1: Štruktúra analyzátora

3.1.2 Štruktúra zdieľanej knižnice

Zdieľaná knižnica sa skladá z nasledujúcich dvoch sekcií:

1. **Sledované funkcie** – obsahuje implementáciu sledovaných funkcií. Tie sú rozdelené do dvoch skupín, podľa toho, či je požadované ich vykonávanie priamo ovládať, alebo ho len sledovať. Tieto dve skupiny sa líšia svojou činnosťou, ich opis je uvedený v sekcií 3.3.1.

2. **Komunikačná časť** – obsahuje funkcie na vytvorenie spojenia s riadiacim programom cez soket, ďalej funkcie na vytváranie a odosielanie správ, rovnako ako funkcie na ich prijímanie a spracovávanie.

3.1.3 Štruktúra riadiaceho programu

Riadiaci program je tou najzložitejšou a najkomplexnejšou časťou celej aplikácie. Z toho dôvodu je pri jeho vývoji využitý objektovo orientovaný prístup, ktorý mimoriadne zvýši prehľadnosť, udržovateľnosť a takisto aj rozšíriteľnosť programu. Práve preto je aj návrh riadiaceho programu objektovo orientovaný.

Riadiaci program je zodpovedný za ovládanie celej dynamickej analýzy programu. To zahŕňa spúšťanie analyzovaného programu, sledovanie a ovládanie knižnicových volaní (s využitím zdieľanej knižnice), vďaka ich ovládaniu vytváranie rôznych behov analyzovaného programu, ukladanie získaných behov a ich spájanie do výsledného grafu toku riadenia.

Kvôli tomu, že riadiaci program musí vykonávať veľké množstvo úloh, je rozdelený do troch základných častí:

1. **Dátová časť** – slúži na uchovávanie získaných informácií o behoch programu, jednotlivých volaniach apod.

2. **Sledovacia časť** – je určená na sledovanie analyzovaného programu. Keďže sledovanie je realizované s využitím zdieľanej knižnice, zahŕňa aj mechanizmy na komunikáciu s touto knižnicou (cez sokety).

3. **Riadiaca časť** – obsahuje triedy určené na riadenie analýzy, plánovanie behov analyzovaného programu a ich následné ukladanie. Takisto je zodpovedná za spájanie behov a vytváranie grafu toku riadenia.

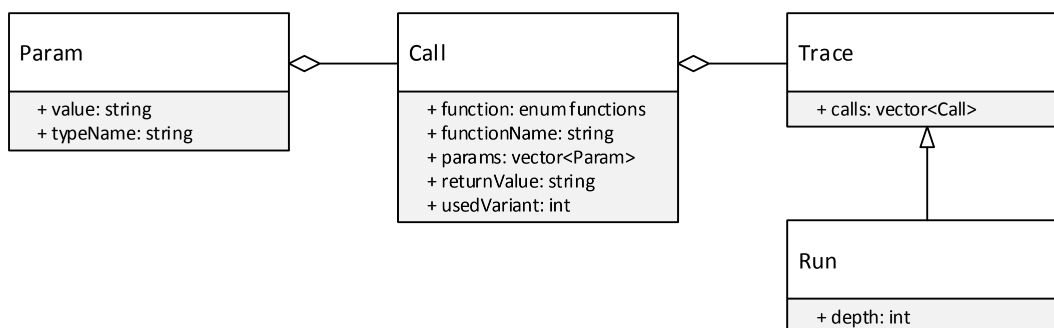
Štruktúra dát

Ešte pred tým, ako bude vysvetlená štruktúra dát, je nutné definovať názvoslovie s tým súvisiace:

- *Stopa* (angl. *trace*) je ľubovoľná postupnosť volaní, vyskytujúcich sa bezprostredne za sebou v analyzovanom programe.

- *Beh* (angl. *run*) je celá postupnosť volaní v analyzovanom programe od začiatku programu až do jeho ukončenia.

Celá štruktúra dát aplikácie je zobrazená na obrázku 3.2.

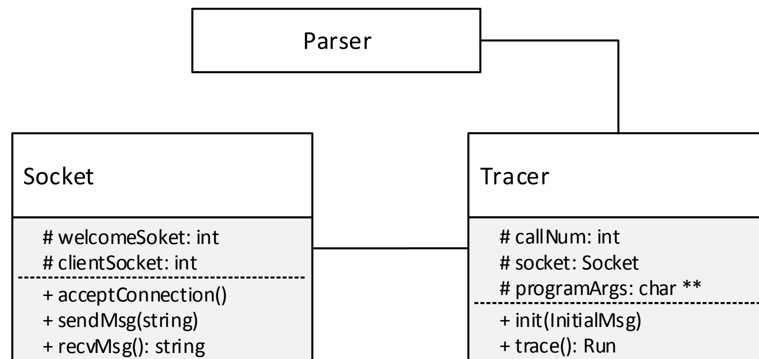


Obr. 3.2: Štruktúra dát analyzátora

Call reprezentuje jedno konkrétne volanie funkcie v analyzovanom programe. Okrem toho, že obsahuje jej názov a návratovú hodnotu volania, uchováva aj zoznam parametrov, s ktorými bola daná funkcia volaná. Tie sú reprezentované triedou *Param*, ktorá obsahuje názov dátového typu a hodnotu daného parametru. Postupnosť (vektor) volaní je stopa (*Trace*). Beh (*Run*) rozširuje *Trace* a reprezentuje postupnosť volaní od začiatku do konca analyzovaného programu. *Run* obsahuje oproti *Trace* informáciu o hĺbke behu, ktorá bude využitá počas analýzy.

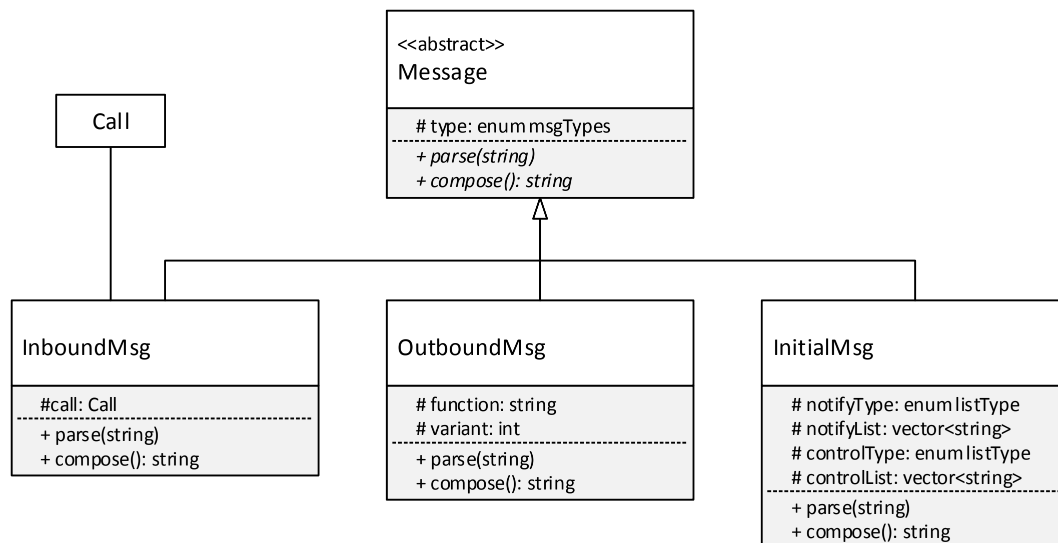
Sledovacia časť

Úlohou tejto časti riadiaceho programu je sledovanie a riadenie analyzovaného programu a získavanie jeho behov. Základom je trieda *Tracer*, ktorá spustí analyzovaný program s úpravou premennej prostredia `LD_PRELOAD`. Potom s využitím triedy *Socket* a komponentu *Parser* komunikuje so zdieľanou knižnicou a tak získa celý beh programu. Štruktúru tejto časti zobrazuje obrázok 3.3



Obr. 3.3: Štruktúra sledovacej časti

Časť *Parser* má na starosti vytváranie a spracovávanie správ v rámci komunikácie so zdieľanou knižnicou. Obsahuje abstraktnú triedu *Message* ktorú následne rozširujú triedy reprezentujúce jednotlivé druhy správ (obr. 3.4). Detailný opis štruktúry a významu jednotlivých správ je uvedený v sekcii 3.2.1.



Obr. 3.4: Štruktúra komponentu *Parser*

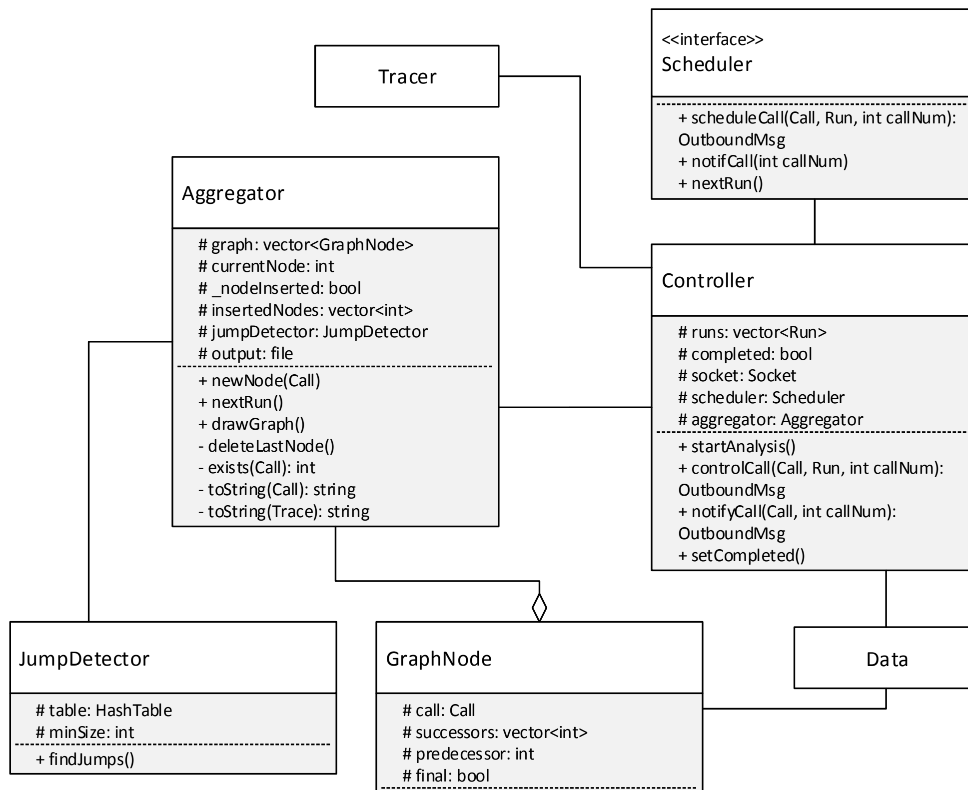
Riadiaca časť

Jedná sa o centrálnu časť celého riadiaceho programu. Jej základom je trieda *Controller*, ktorá ovláda celý priebeh analýzy.

Ďalšou súčasťou je rozhranie *Scheduler*, zodpovedné za plánovanie vytvárania jednotli-

vých behov. Trieda implementujúca toto rozhranie plánuje akým spôsobom budú ovládané behy programu, tak aby došlo k preskúmaniu čo najväčšieho množstva jeho vetiev. Každá takáto trieda môže použiť inú techniku plánovania, jedinou podmienkou je aby implementovala dané rozhranie. Metódy tohoto rozhrania sú volané vždy, keď zdieľaná knižnica zachytí vykonanie nejakého zo sledovaných volaní (viď. sekcia 3.2.2).

Na koniec sa tu nachádza agregačná časť, ktorej základom je trieda *Aggregator*. Tá je zodpovedná za spájanie získaných behov do výsledného grafu toku riadenia (agregáciu). Tento graf je reprezentovaný vektorom uzlov, ktoré sú inštanciami triedy *GraphNode*. Každý uzol obsahuje informáciu o volaní v ňom uloženom, ďalej zoznam indexov svojich nasledovníkov, rovnako ako index svojho predchodcu. Takisto sa v ňom nachádza informácia o tom, či je daný uzol grafu konečný (tj. či dané volanie bolo posledné v nejakom behu). Agregácia môže byť realizovaná podľa viacerých kritérií, tie sú bližšie opísané v časti 3.3.6. Okrem triedy *Aggregator* táto časť obsahuje ešte triedu *JumpDetector*, ktorej úlohou je vyhľadávanie cyklov a opakujúcich sa postupností volaní (stôp) v grafe a následné vytváranie skokov v grafe. Presný opis algoritmu vyhľadávania skokov sa nachádza v sekcii 3.3.7.



Obr. 3.5: Štruktúra riadiacej časti

3.2 Opis interakcie

Ďalšia časť návrhu analyzátora sa zameriava na interakciu, teda *vzájomné pôsobenie* medzi jeho jednotlivými komponentmi. V prvom rade je to opis komunikácie medzi dvoma základnými časťami – zdieľanou knižnicou a riadiacim programom. Ako bolo vysvetlené v sekcii 3.1.1, táto komunikácia prebieha s využitím soketov. Prvá podsekcia tejto časti návrhu je venovaná detailnému opisu protokolu použitého v jej priebehu.

Ďalej bude taktiež znázornená a vysvetlená interakcia medzi jednotlivými triedami riadiaceho programu pri vykonávaní jeho hlavných činností.

3.2.1 Komunikačný protokol

Komunikácia medzi zdieľanou knižnicou a riadiacim programom prebieha prostredníctvom výmeny správ pomocou soketov. Táto komunikácia je typu *klient-server*, keď zdieľaná knižnica vystupuje v roli klienta (tj. odosiela správy) a riadiaci program zasa v roli serveru (prijíma správy a reaguje na ne). Nasledujúce odseky opisujú protokol použitý počas tejto komunikácie, to znamená presnú štruktúru a poradie použitých správ.

Jedná sa o jednoduchý textový, riadkovo založený protokol. To znamená, že správy sú zasielané a prijímané ako text a jednotlivé informácie, ktoré nesie jedna správa sú vždy oddelené novým riadkom (konkrétne znakmi `CRLF`). Na konci celej správy sa nachádza ešte jeden nový riadok naviac, takže celá správa končí dvoma `CRLF` za sebou. Komunikácia vždy prebieha v tvare *správa – odpoveď*, pričom prvú správu zasiela zdieľaná knižnica riadiacemu programu pri vzniku nejakej udalosti a odpoveď na ňu je vytvorená riadiacim programom a zaslaná späť.

Celú komunikáciu je možné rozdeliť do dvoch fáz. V prvom rade je to **ustanovenie spojenia**, ktoré prebieha na začiatku (vždy po novom spustení analyzovaného programu). Po tom ako je spojenie vytvorené, nastáva fáza **hlavnej komunikácie**, počas ktorej zdieľaná knižnica zasiela riadiacemu programu informácie o zachytených volaniach.

Ustanovenie spojenia

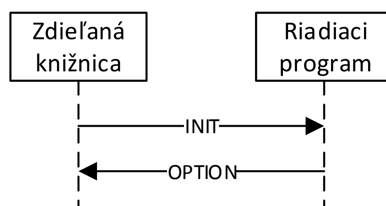
Vytvorenie spojenia je úvodnou fázou celej komunikácie. Keďže po skončení behu jednej inštancie analyzovaného programu je ukončená aj činnosť zdieľanej knižnice, ustanovenie spojenia musí prebiehať vždy pri spustení nového behu analyzovaného programu. Kód zdieľanej knižnice sa začne vykonávať pri volaní prvej zo sledovaných funkcií, preto vtedy prebehne aj táto fáza komunikácie.

Celý jej priebeh obsahuje len dve správy, jedná sa teda o dvojcestné nadviazanie spojenia (angl. two-way handshake). Slúži nielen na vytvorenie spojenia medzi zdieľanou knižnicou a riadiacim programom, ale taktiež aj na nastavenie parametrov analýzy v zdieľanej knižnici. Postupnosť správ je zobrazená na obrázku 3.6.

Prvou zaslanou správou je správa typu `INIT`. Neobsahuje žiadne doplnkové informácie, slúži len na oznámenie, že zdieľaná knižnica zahajuje komunikáciu. Správa má nasledujúci tvar:

```
INIT
```

Odpoveďou je správa typu `OPTION`. Tá slúži okrem informovania zdieľanej knižnice, že riadiaci program je pripravený na komunikáciu aj na špecifikovanie parametrov sledovania analyzovaného programu. Konkrétne sa jedná o určenie zoznamu funkcií, ktoré má riadiaci



Obr. 3.6: Ustanovenie spojenia medzi zdieľanou knižnicou a riadiacim programom

program záujem priamo ovládať (ďalej len *zoznam ovládaných funkcií*) a zoznamu tých, ktorých vykonávanie chce iba sledovať (ďalej len *zoznam sledovaných funkcií*). Štruktúra správy `OPTION`:

```
OPTION
NOTIFICATION
(ALL|NONE|INCLUDE)
[<zoznam-funkcii>]
CONTROL
(ALL|NONE|INCLUDE)
[<zoznam-funkcii>]
```

Za riadkom `NOTIFICATION` nasleduje zoznam sledovaných funkcií a za riadkom `CONTROL` zasa zoznam ovládaných funkcií. Pre každý z nich je na nasledujúcom riadku nutné určiť, či bude zahŕňať všetky dostupné funkcie (`ALL`), žiadnu z nich (`NONE`), alebo iba vybrané funkcie (`INCLUDE`). V prípade poslednej možnosti je potrebné na ďalších riadkoch uviesť zoznam názvov týchto vybraných funkcií, napísaných malými písmenami a oddelených medzi sebou pomocou `CRLF`.

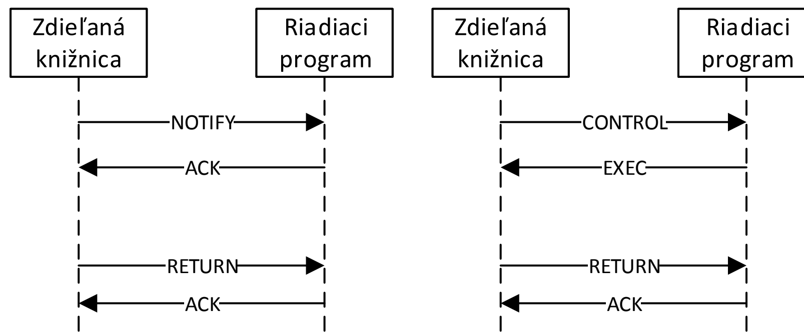
Hlavná komunikácia

Po tom ako je vytvorené spojenie, nastáva fáza hlavnej komunikácie. Vždy keď analyzovaný program zavolá zo zdieľanej knižnice jednu z funkcií, tá zašle riadiacemu programu informáciu o jej vykonaní. Podľa toho, či má riadiaci program záujem volanie tejto funkcie ovládať, alebo len sledovať, existujú dve možnosti výmeny správ ohľadom daného volania zobrazené na obrázku 3.7.

Prvá možnosť (naľavo) zobrazuje situáciu, kedy riadiaci program chce volanie iba sledovať. Po zaznamenaní volania zašle zdieľaná knižnica správu typu `NOTIFY`. Táto správa má tvar:

```
NOTIFY
<nazov-funkcie>
<zoznam-parametrov>
```

Správa informuje o názve vykonávanej funkcie a ďalej obsahuje hodnoty parametrov, s ktorými je daná funckia volaná. Jednotlivé parametre sú medzi sebou oddelené znakom `CRLF`. Odpoveďou na túto správu je správa typu `ACK`, čo je len jednoduché potvrdenie prijatia, v tvare:



Obr. 3.7: Možnosti výmeny správ pri volaní sledovanej funkcie analyzovaným programom

```
ACK
```

Po tom, čo zdieľaná knižnica obdrží správu `ACK`, dokončí volanie funkcie, uloží si jeho návratovú hodnotu a tú následne zašle riadiacemu programu v správe typu `RETURN`, ktorej štruktúra je nasledujúca:

```
RETURN
<navratova-hodnota>
```

Odpoveďou na správu `RETURN` je opäť správa `ACK`.

Ak však príde volanie, ktorého vykonanie má riadiaci program záujem priamo ovládať, komunikácia prebieha podľa diagramu napravo. Na začiatku pošle zdieľaná knižnica správu `CONTROL`, ktorá má podobnú funkciu ako správa `NOTIFY`, teda informovať, že sa daná funkcia bude vykonávať. Takisto štruktúra týchto správ je takmer rovnaká, okrem úvodného riadku znamenajúceho typ správy:

```
CONTROL
<nazov-funkcie>
<zoznam-parametrov>
```

Najväčší rozdiel oproti správe `NOTIFY` je v reakcii riadiaceho programu. Tu nestačí iba zaslanie potvrdenia prijatia, je nutné špecifikovať, akým spôsobom bude pozmenené vykonanie tejto funkcie. Na toto slúži správa `EXEC`, ktorá má tvar:

```
EXEC
<nazov-funkcie>
<varianta-priebehu>
```

Najdôležitejšou informáciou v tejto správe je *varianta priebehu*. Jedná sa o celé číslo určujúce, ktorá z variant volania funkcie implementovaných v zdieľanej knižnici sa má vykonať. Viac informácií sa nachádza v kapitole 3.3.1.

Po prijatí správy `EXEC` zdieľaná knižnica vykoná požadovanú variantu a takisto ako pri prvom type komunikácie pošle riadiacemu programu informáciu o návratovej hodnote volania. Na to je použitá správa `RETURN` s odpoveďou `ACK`, ktorých štruktúra a funkcia je popísaná vyššie.

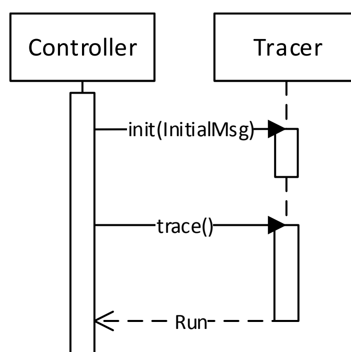
3.2.2 Interakcia medzi triedami riadiaceho programu

Riadiaci program obsahuje množstvo tried, ktoré medzi sebou navzájom spolupracujú a volajú svoje metódy. Táto časť zobrazuje to, v akom poradí sú jednotlivé metódy volané počas vykonávania základných činností riadiaceho programu, konkrétne **spustenia stopovania behu analyzovaného programu**, **výmeny správ so zdieľanou knižnicou** a **reakcie na správu o volaní**. Vysvetlenie je zamerané prevažne na triedy sledovacej a riadiacej časti riadiaceho programu.

Úvodnou metódou celej analýzy je metóda triedy *Controller* s názvom `startAnalysis` (ďalej bude používané značenie `Controller.startAnalysis`).

Spustenie stopovania behu

Analýza pozostáva zo sledovania a získavania viacerých behov analyzovaného programu. Ich vytváranie má na starosti trieda *Tracer*, ktorej činnosť je ovládaná triedou *Controller*. Spustenie stopovania analyzovaného programu je zobrazené na obrázku 3.8.



Obr. 3.8: Spustenie sledovania analyzovaného programu

`Tracer.init` spustí analyzovaný program, vytvorí soket a nadviaže spojenie so zdieľanou knižnicou. Bližší opis spúšťania programu touto metódou sa nachádza v sekcii 3.3.3. O vytvorenie celého behu sa potom stará metóda `Tracer.trace`.

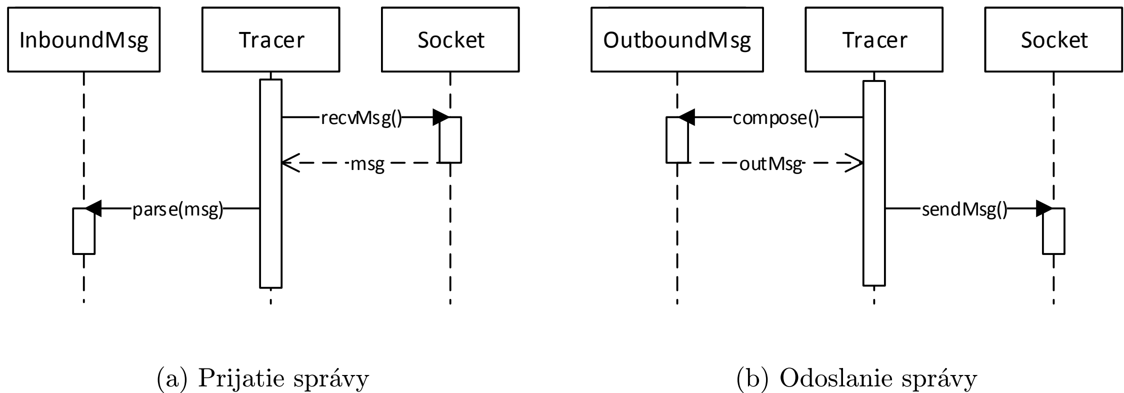
Výmena správ so zdieľanou knižnicou

Metóda `Tracer.trace`, ktorá má na starosti stopovanie analyzovaného programu, čaká na správy od zdieľanej knižnice. Tieto správy potom spracováva a vytvára vhodné reakcie na ne. Obrázok 3.9 zobrazuje proces prijatia a spracovania správy, rovnako ako proces vytvorenia a odoslania.

V rámci komunikácie o jednom konkrétnom volaní sú obe činnosti vykonané dva krát z toho dôvodu, že najskôr je prijatá samotná správa o volaní (buď `CONTROL`, alebo `NOTIFY`) a po zaslaní odpovede na ňu je ešte prijatá správa o návratovej hodnote volania (`RETURN`).

Reakcia na správu o volaní

Po tom, čo je prijatá správa o volaní zo zdieľanej knižnice, je nutné vytvoriť na túto správu správnu odpoveď. Na obrázku 3.10 je zobrazená reakcia na prijatie správy typu `CONTROL`

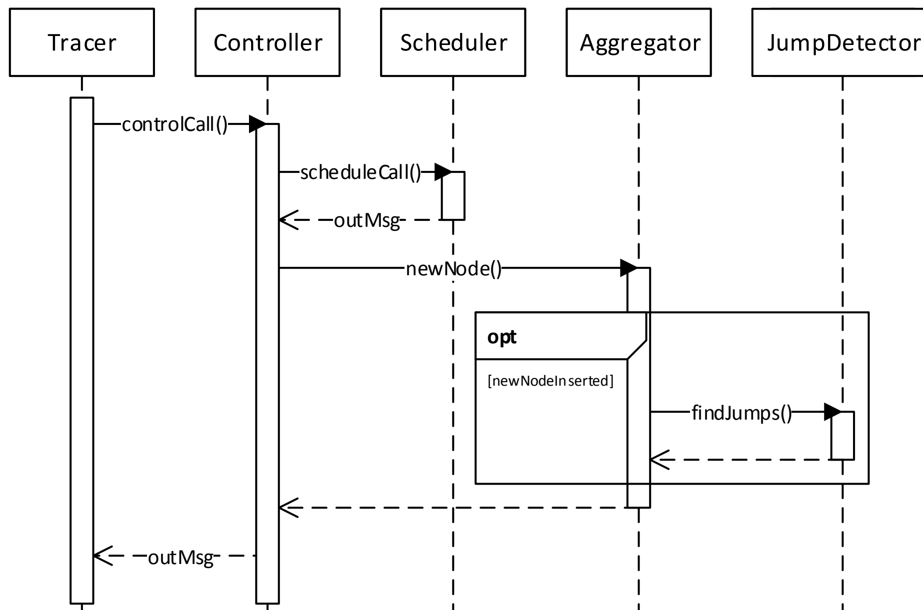


(a) Prijatie správy

(b) Odoslanie správy

Obr. 3.9: Komunikácia so zdieľanou knižnicou

informujúcej o ovládateľnom volaní zachytenom zdieľanou knižnicou.



Obr. 3.10: Reakcie na prijatie správy o volaní funkcie

Celý proces začína volaním triedy *Controller*. Z nej je následne volaná metóda rozhrania *Scheduler*, ktorá vytvorí a vráti správu obsahujúcu odpoveď (správa je typu `EXEC`). Po získaní odpovede je volaná trieda *Aggregator*, ktorá agreguje nové volanie do existujúceho grafu toku riadenia a v prípade, že došlo k úprave grafu (vloženiu nového uzlu), zavolá detekciu skokov v triede *JumpDetector*. Nakoniec trieda *Controller* vráti triede *Tracer* odpoveď získanú od triedy *Scheduler*, ktorú *Tracer* zašle späť zdieľanej knižnici.

Reakcia na správu typu `NOTIFY`, ktorá informuje o volaní, ktoré nie je možné ovládať je podobná ako tá predchádzajúca. Jediným rozdielom je, že na začiatku sa volá metóda triedy *Controller* `notifyCall` namiesto `controlCall`. Tá potom nevolá z rozhrania *Scheduler* metódu `scheduleCall`, ale metódu `notifCall`. Táto nevracia správu obsahujúcu odpoveď,

nakoľko tá je vždy rovnaká, konkrétne správa typu `ACK`. Agregácia a vyhľadávanie skokov prebieha rovnako, ako v prvom prípade.

3.3 Opis správania analyzátora

Poslednou súčasťou návrhu systému je *opis správania analyzátora*. Obsahuje podrobné vysvetlenie činnosti jednotlivých komponentov a algoritmov v nich použitých. Prvá časť sa zameria na fungovanie zdieľanej knižnice, ostatné potom na činnosť jednotlivých tried riadiaceho programu (konkrétne tried *Controller*, *Tracer*, *Scheduler*, *Aggregator* a *JumpDetector*).

3.3.1 Činnosť zdieľanej knižnice

Ako už bolo vysvetlené v časti 3.1.2, zdieľaná knižnica sa skladá z komunikačnej časti a implementácie sledovaných funkcií.

Komunikačná časť obsahuje funkcie, ktoré zabezpečujú vytvorenie spojenia cez soket, spracovávanie správ prijatých od riadiaceho programu a vytváranie správ na odoslanie. Tieto funkcie sú volané z druhej časti, zo **sledovaných funkcií**. Činnosť každej z týchto funkcií sa skladá z rovnakých krokov:

1. V prípade, že sa jedná o prvú funkciu od spustenia analyzovaného programu, prvým krokom je **vytvorenie spojenia** s riadiacim programom cez soket. Tento krok zahŕňa odoslanie správy `INIT`, prijatie správy `OPTION` a na základe informácií v nej nastavenie zoznamov ovládaných a sledovaných funkcií.

2. Vytvorenie a odoslanie **správy o vykonávaní** daného volania. Pri ovládaných funkciách ide o správu `CONTROL`, pri sledovaných zasa o správu `NOTIFY`.

3. **Prijatie a spracovanie odpovede** na predchádzajúcu správu (správa `EXEC`, resp. `ACK`).

4. Podľa prijatého čísla varianty, **vykonanie príslušnej varianty**. V prípade, že sa jedná o variantu s číslom 0, alebo je funkcia iba sledovaná, jedná sa o zavolanie originálnej funkcie. Pred samotným vykonaním sa uloží návratová hodnota.

5. Odoslanie **správy s návratovou hondotou**, konkrétne správy typu `RETURN`. Takisto tento krok obsahuje aj prijatie odpovede (správy `ACK`).

3.3.2 Hlavný riadiaci algoritmus

Činnosť celého riadiaceho programu ovláda trieda *Controller*. V nej sa nachádza metóda `startAnalysis`, v ktorej sa celá analýza odohráva. Ako už bolo vysvetlené, analýza programu prebieha jeho opakovaným spúšťaním a ďalej sledovaním a ovládaním jeho behu, konkrétne knižnicových volaní, ktoré analyzovaný program volá.

Celá analýza prebieha nasledujúcim spôsobom. Na začiatku je získaný **prvý beh** programu iba jednoduchým sledovaním volaní (tzn. v úvodnej správe `OPTION` sú všetky volania označené ako sledované a nedochádza k ich ovládaniu). Následne sú v **hlavnom riadiacom cykle** získavané ďalšie behy, pri ktorých už dochádza k ovládaniu jednotlivých volaní. Plánovanie tohoto ovládania, rovnako ako ukončenie cyklu má na starosti plánovač, teda trieda implementujúca rozhranie *Scheduler*.

Získanie behov prebieha s využitím triedy *Tracer* tak, ako to bolo popísané v sekcii 3.2.2. Počas hlavného riadiaceho cyklu je získaný beh uložený iba v prípade, že je rozdielny od tých ostatných (dovtedy získaných). To, či je rozdielny, sa určí na základe toho, či počas agregácie došlo ku vzniku aspoň jedného nového uzlu (tj. došlo k volaniu, ktoré neobsahoval ani jeden z doterajších behov).

3.3.3 Spúšťanie analyzovaného programu

O správne spustenie analyzovaného programu sa stará trieda *Tracer*, konkrétne jej metóda `init`. Spustenie programu prebieha v novo vytvorenom procese. Najskôr je v ňom nastavená cesta ku zdieľanej knižnici do premennej prostredia `LD_PRELOAD`. Následne je spustený samotný analyzovaný program. V prvom rade je automaticky spustený dynamický linker, ktorého úlohou je načítanie zdieľaných knižníc vyžadovaných programom. Vďaka nastaveniu `LD_PRELOAD` dôjde k tomu, že je nájdená a pripojená vytvorená zdieľaná knižnica a funkcie v nej obsiahnuté.

V rodičovskom procese sa nadviaže prichádzajúce spojenie cez socket od zdieľanej knižnice (riadiaci program vystupuje v roli serveru) a prijíme sa správa `INIT`. Následne je poslaná správa `OPTION` daná ako argument metódy `init`.

3.3.4 Stopovanie analyzovaného programu

Samotné sledovanie (a ovládanie) analyzovaného programu zabezpečuje metóda `Tracer.trace`, ktorá v cykle prijíma a odosiela správy do zdieľanej knižnice. Jedna iterácia cyklu slúži na získanie informácií o jednom konkrétnom volaní a obsahuje nasledujúce kroky:

1. **Prijatie správy o volaní**. Po tom, ako je táto správa prijatá, sú volané ďalšie triedy ktoré zabezpečia zaznamenanie volania a vytvorenie odpovede (postupnosť volania týchto tried je znázornená v časti 3.2.2).

2. **Odoslanie odpovede** vytvorenej v predchádzajúcom kroku.

3. **Prijatie správy o návratovej hodnote** a jej následné uloženie ku danému volaniu.

4. **Potvrdenie prijatia** predchádzajúcej správy (`ACK`).

Tento cyklus prebieha, kým prichádzajú správy od zdieľanej knižnice.

3.3.5 Algoritmus plánovania behov

To, akým spôsobom budú jednotlivé volania v behoch ovládané (tj. aké varianty volaní v zdieľanej knižnici budú vybrané) ovplyvní vznik rôznych behov a teda aj preskúmanie rôznych vetiev programu. Čím viac týchto vetiev (čiže rôznych behov) sa podarí získať, tým väčšia je presnosť dynamickej analýzy, pretože je menšia pravdepodobnosť, že nejaká možnosť nebola pokrytá. O plánovanie ovládania volaní sa stará *plánovač*, reprezentovaný rozhraním *Scheduler* a triedami toto rozhranie implementujúcimi.

Každá z nich môže použiť inú techniku plánovania, cieľom je vždy vytvoriť odpoveď na správu o aktuálne prebiehajúcom volaní. To znamená určiť, aká varianta priebehu tohoto volania bude zdieľanou knižnicou vykonaná. Jednou z možností je systematické používanie variant tak, aby došlo ku vytvoreniu všetkých možných behov, ktoré použitie dostupných variant umožňuje. V nasledujúcej časti bude opísaný algoritmus založený na algoritme prehľadávania stavového priestoru do šírky (BFS) známeho z oblasti umelej inteligencie.

Algoritmus plánovania BFS

Prehľadávanie do šírky (angl. *breadth-first search*) je založené na tom, že pre každý uzol grafu sú najprv preskúmaní všetci jeho nasledovníci a až následne sa pokračuje ďalším uzlom na danej úrovni, prípadne na úrovni nižšej.

Algoritmus plánovania behov funguje na podobnom princípe. Postupne berie získané behy, z nich vyberá jedno za druhým ich volania a pre každé z týchto volaní použije v ďalších behoch všetky jeho dostupné varianty (*expanduje* tieto volania). Týmto spôsobom dochádza k vzniku nových behov, v ktorých potom expanzia vždy začína až volaním, ktorého expanzia viedla k vzniku daného behu (hĺbka tohoto volania je označovaná ako *hĺbka behu*). Celý algoritmus je popísaný pomocou nasledujúcich krokov:

1. Vezmi prvý beh a označ ho ako aktívny. Prvý beh má hĺbku 0.
2. Z aktívneho behu vyber volanie s poradovým číslom o jedna väčším ako je hĺbka behu a označ ho ako expandované volanie (platí, že prvé volanie má poradové číslo 1).
3. Práve prebiehajúci beh expanduj v expandovanom volaní prvou nepoužitou variantou a nastav hĺbku tohoto behu na hĺbku expandovaného volania. To znamená, že v danom behu budú pre všetky volania pred expandovaným použité rovnaké varianty ako boli použité v aktívnom behu, pre expandované volanie bude použitá prvá možná varianta a pre ostatné bude použitá varianta 0.
4. Ak sú pre expandované volanie ešte k dispozícií ďalšie nepoužité varianty, pokračuj krokom 3, inak pokračuj krokom 5.
5. Ak expandované volanie nie je posledným v aktívnom behu, označ ďalšie volanie ako expandované, označ všetky jeho varianty ako nepoužité a pokračuj krokom 3. V opačnom prípade pokračuj krokom 6.
6. Ak za aktívnym behom v zozname behov ešte nejaký existuje, označ ho ako aktívny a pokračuj krokom 2. Inak ukonči analýzu zastavením prevádzania *hlavného riadiaceho cyklu* (3.3.2).

Ukončenie činnosti algoritmu (a tým pádom aj celej analýzy) je podmienené tým, že nie všetky v ňom vytvorené behy sú rozdielne od už získaných a vďaka tomu by nemalo dôjsť k nekonečnému pridávaniu nových behov (čo sa však môže stať a preto algoritmus nie je vždy konečný).

3.3.6 Algoritmus vytvárania výsledného grafu

Cieľom činnosti analyzátoru je vytvorenie rámcu (angl. *framework*) pre všeobecnú analýzu programu. Jednou z jeho podčastí je aj tvorba grafu toku riadenia (tzv. *control-flow graph*). Vzhľadom na to, že je využívaná dynamická analýza, ktorá má k dispozícií iba obmedzený počet behov programu, je možné získať iba približnú verziu tohoto grafu (jej presnosť potom závisí od množstva a rôznorodosti získaných behov). O jeho vytváranie sa stará trieda *Aggregator*, ktorá je volaná vždy pri príchode nového volania (viď. sekcia 3.2.2).

Na začiatku analýzy je do grafu vložený nultý uzol, ktorý neobsahuje žiadne volanie. Tento uzol je zároveň nastavený ako *aktívny*. Potom vždy po príchode nového volania prebiehajú nasledujúce kroky:

1. Porovnanie, či nejaký z nasledovníkov aktívneho uzlu neobsahuje rovnaké volanie.
2. Ak nejaký z nasledovníkov takéto volanie obsahuje, je nastavený ako aktívny a činnosť triedy *Aggregator* končí. V opačnom prípade sa pokračuje ďalším krokom.
3. Je vytvorený nový uzol obsahujúci príchodzie volanie a zaradený do grafu ako nasledovník aktívneho uzlu. Zároveň je tento nový uzol označený za aktívny.

Vždy po spustení nového behu analyzovaného programu je ako aktívny nastavený nultý uzol.

Porovnávanie dvoch volaní a určenie toho, či sú rovnaké, záleží na použitom stupni abstrakcie. Tá určí, ktoré z informácií o volaní sú dôležité pre porovnanie a ktoré je možné zanedbať. Vďaka tomu môže existovať viacero tried implementujúcich agregáciu, keď každá z nových tried dedí od základnej triedy *Aggregator*. Keďže porovnávanie dvoch volaní prebieha tak, že najskôr sú obidve reprezentované pomocou reťazca (angl. *string*) a potom sú tieto dva reťazce porovnané, stačí aby nové triedy redefinovali metódu prevádzajúcu volanie na reťazec. V princípe existujú tri základné druhy abstrakcie popísané ďalej.

Agregácia podľa mena volania

Prvý druh je najvyššou formou abstrakcie a na to, aby dve volania boli považované za rovnaké stačí, aby sa zhodovali názvy volaných funkcií. Tento typ agregácie je využiteľný napríklad pre získanie jednoduchého odtlačku správania programu (angl. *fingerprint*).

Agregácia podľa bázového parametru

Druhou možnosťou je agregácia nielen podľa mena funkcie, ale aj podľa hodnoty bázového parametru. Pri každom volaní sa jedná o ten parameter, ktorý najpresnejšie vyjadruje rozdielnosť volania od ostatných volaní tej istej funkcie. Tu už je výsledný graf možné použiť pre zložitejšie činnosti, ako je overenie splniteľnosti základných pravidiel správania programu (napr. zistenie, či každý otvorený súbor je vždy aj uzatvorený).

Agregácia podľa hodnoty všetkých parametrov

Posledná možnosť neobsahuje žiadnu abstrakciu a rovnosť volaní je určená na základe mena funkcie a takisto na základe hodnôt všetkých parametrov volania. Tu môže vzniknúť problém pri parametroch dátového typu ukazovateľ (angl. *pointer*), pretože ich hodnota sa môže medzi jednotlivými behmi meniť, hoci hodnota v odkazovanej časti pamäte môže byť rovnaká. Tento typ agregácie by bolo možné (avšak nie isté, práve pre problém s ukazovateľmi) využiť napríklad pre kompaktné uloženie množiny behov programu.

3.3.7 Detekcia skokov

Jednoduchá agregácia, ako bola popísaná v predchádzajúcej sekcií vytvorí graf, ktorý má tvar stromu. To znamená, že každý uzol má nejaký počet nasledovníkov, ale vždy iba jedného predchodcu. To by znamenalo, že do určitého bodu v programe sa je možné dostať iba z jedného jediného miesta. V reálnom programe však takáto situácia neplatí a to z dôvodu existencie dvoch riadiacich štruktúr, konkrétne *cyklov* a *podprogramov*. Do začiatku cyklu je možné sa dostať z jeho konca a do začiatku podprogramu zasa z miesta jeho volania. Preto

aby vytvorený graf čo najpresnejšie zodpovedal reálnemu toku riadenia, je nutné nájsť tieto skoky. Na to slúži trieda *JumpDetector*, ktorá ich hľadá a zaznamenáva do grafu.

Použitý algoritmus detekcie skokov je založený na hľadaní **rovnakých postupnosti volaní** (stôp). Funguje tak, že všetky vzniknuté stopy ukladá a vždy pri vzniku novej zisťuje, či sa takáto stopa v programe už nenachádza. V prvom rade je nutné určiť minimálnu dĺžku stopy, ktorú je už možné považovať za podprogram (prípadne za telo cyklu). Algoritmus je navrhnutý tak, že všetky dlhšie podprogramy budú takisto zachytené.

Na ukladanie jednotlivých stôp sa používa **hashovacia tabuľka**, ktorá mapuje reťazcovú reprezentáciu jednotlivých stôp na indexy v grafe, na ktorých je daná stopa uložená. Na začiatku je táto tabuľka prázdna. Vždy pri vložení nového uzlu dôjde k nasledujúcej postupnosti krokov:

1. Získanie stopy, ktorá končí práve vloženým uzlom. V prípade, že takáto stopa neexistuje (medzi koreňom stromu a vloženým uzlom je menší počet uzlov ako je minimálna dĺžka stopy), algoritmus končí.

2. Zistenie, či daná stopa už bola do grafu vložená. Ak taká stopa ešte neexistuje, sú na jej index v hashovacej tabuľke vložené indexy stopy získanej v kroku 1 a algoritmus končí. V opačnom prípade je vektor indexov získaný z hashovacej tabuľky označený ako *cieľ skoku*.

3. Stopa získaná v kroku 1 musí spĺňať podmienku, že bola celá vytvorená v jednom behu programu. V prípade, že nebola, algoritmus končí, inak pokračuje ďalším krokom.

4. Ešte pred vytvorením samotného skoku je nutné stopu z grafu odstrániť a upraviť hashovaciu tabuľku tak, aby zodpovedala novo vytvorenému skoku. Pre každý uzol zo stopy je najskôr nájdená stopa končiaca v tomto uzle. Ak sa práve táto stopa nachádza v hashovacej tabuľke, je upravená jej časť, ktorá sa prekrýva s práve odstraňovanou stopou tak, aby smerovala do cieľa skoku. Následne je tento uzol odstránený z grafu.

5. Z uzlu pôvodne sa nachádzajúceho pred stopu získanou v kroku 1 je vytvorený skok do prvého uzlu v cieli skoku. Zároveň je ako aktívny nastavený posledný uzol v cieli skoku (toto nastavenie zaručí, že aj v prípade, ak je podprogram dlhší, sa bude pokračovať jeho ďalším volaním).

3.3.8 Tvorba výstupu

Výstupom analyzátora je graf riadenia toku programu. Tento je možné zobraziť v rôznych formátoch, napríklad v nejakom štandardnom (ako je XML, alebo JSON), prípadne vo formáte, ktorý umožňuje vizualizáciu (napríklad zdrojový súbor pre program `graphviz`). O vytváranie výstupu sa stará trieda *Aggregator*, keďže vytvorený graf sa nachádza práve v nej. Hoci v tomto grafe sú volania uložené v jednotlivých uzloch, vo výstupnom grafe by mali byť priradené k prechodovým hranám. Preto každý uzol grafu v triede *Aggregator* musí byť vo výstupe vizualizovaný ako hrana prichádzajúca do tohoto uzlu.

Okrem toho je vytvorený zvláštny uzol označený `F`, ktorý značí *koncový bod* programu. Zo všetkých uzlov označených, ako konečné (majú príznak `final` nastavený na `true`) je vytvorená ε-hrana (prázdna hrana) do tohoto koncového uzlu. Vďaka tomu je analyzovaný program vo výstupe reprezentovaný, ako *SESE program (single-entry/single-exit)*, čo znamená, že má práve jeden vstupný (nultý uzol) a jeden výstupný bod (koncový uzol).

Kapitola 4

Implementácia analyzátora

Po tom, ako je vytvorený dobrý návrh, je možné pristúpiť ku samotnej implementácií analyzátora. V nej sa v podstate postupuje podľa návrhu s tým, že je nutné tento návrh prispôsobiť možnostiam zvoleného implementačného jazyka.

Táto kapitola opisuje implementačné detaily jednotlivých častí analyzátora. Zameriava sa prevažne na tie detaily, ktoré sú špecifické pre jazyky C/C++, v ktorých je tento analyzátor naprogramovaný, prípadne na také, v ktorých sa implementácia odlišuje od návrhu.

4.1 Implementácia zdieľanej knižnice

Zdieľaná knižnica je implementovaná v jazyku C. Je to výhodné hlavne z toho hľadiska, že analyzátor je zameraný na skúmanie volaní *štandardnej knižnice jazyka C*. Jedná sa o jeden zdrojový súbor obsahujúci ako komunikačnú časť, tak aj implementáciu všetkých analyzovaných funkcií.

4.1.1 Komunikačná časť

Vytvorenie spojenia a komunikácia (odosielanie a prjímanie správ) cez soket je implementované s využitím funkcií z knižnice `sys/socket.h`, konkrétne `socket`, `connect`, `send` a `recv`.

Keďže správy použité v protokole sú textové, na ich spracovávanie a vytváranie sú použité funkcie manipulujúce s textom z knižnice `string.h`.

4.1.2 Zoznamy sledovaných volaní

V časti analyzovaných funkcií sa nachádzajú dva druhy funkcií. Jedny umožňujú ako sledovanie, tak aj ich ovládanie (podľa toho, v ktorom zo zoznamov v správe `OPTION` sa daná funkcia nachádza) a tie druhé umožňujú iba ovládanie (tie v správe `OPTION` nie je možné zaradiť do zoznamu ovládaných funkcií).

Zoznam implementovaných funkcií, ktoré je možné ovládať, sa nachádza v nasledujúcej tabuľke:

`read`	`link`	`fchmod`
`write`	`symlink`	`flock`
`open`	`unlink`	`opendir`
`open64`	`stat`	`readdir`
`close`	`lstat`	`closedir`
`lseek`	`fstat`	`mkdir`
`creat`	`access`	`rmdir`
`creat64`	`chmod`	`fsync`

Zoznam implementovaných funkcií, ktoré je možné iba sledovať, sa nachádza v nasledujúcej tabuľke:

`mmap`	`select`	`lchown`
`munmap`	`poll`	`mount`
`mlock`	`dup`	`umount`
`munlock`	`dup2`	`umount2`
`mlockall`	`shmget`	`umask`
`munlockall`	`chown`	`rewinddir`
`brk`	`fchown`	`sync`

Technicky nie je problém tieto volania ovládať, avšak ich riadenie nebolo implementované, pretože to je nad rámec bakalárskej práce.

4.1.3 Implementácia obaľovacích funkcií pre sledované volania

Definície všetkých analyzovaných funkcií majú rovnakú štruktúru. Keďže je knižnica implementovaná v jazyku C, rovnako ako originálne funkcie v štandardnej knižnici jazyka C, hlavičky funkcií v zdieľanej knižnici sú zhodné s tými, ktoré sú uvedené v manuálových stránkach daných volaní.

Ďalej sú volané funkcie komunikačnej časti tak, aby bola zaistená správna výmena správ s riadiacim programom tak, ako je to uvedené v návrhu. Ovládateľné funkcie naviac obsahujú riadiacu štruktúru `switch`, v ktorej je vybraná správna varianta. V prípade, že sa má vykonať originálna funkcia, je na to využitá funkcia `dlsym` tak, ako je to popísané v sekcií 2.2.2.

Funkcie stat, lstat, fstat

Jediný problém pri implementácií obaľovacích funkcií vznikol pri funkciách `stat`, `lstat` a `fstat`. Pri volaní týchto funkcií z programu sa v skutočnosti nevolajú rovnomenné funkcie zo štandardnej knižnice jazyka C, ale pomocou ďalších staticky prilinkovaných funkcií sú volané funkcie `__xstat`, `__lxstat` a `__fxstat`. Preto v zdieľanej knižnici sú definované funkcie práve s týmito hlavičkami, ale v komunikácií s riadiacim programom sú použité názvy správnych funkcií (`stat`, `lstat` a `fstat`).

4.2 Implementácia riadiaceho programu

Vzhľadom na to, že už návrh riadiaceho programu je objektovo orientovaný, je nutné vybrať jazyk, ktorý OO prístup umožňuje. Preto bol zvolený jazyk C++, konkrétne v jeho novej norme C++11, ktorá prináša množstvo nových programátorských možností oproti tej starej.

Jednotlivé triedy sú implementované v súlade s návrhom, doplnené sú niektoré ďalšie pomocné atribúty a metódy (prevažne metódy typu `get` a `set` používané pre získavanie a nastavovanie hodnôt atribútov, ktoré sú väčšinou označené ako *chránené*, čiže z iných tried neprístupné). Rovnako je doplnených niekoľko nových tried, ktorých význam bude popísaný ďalej v tejto sekcii.

Úvodným bodom programu je funkcia `main` nachádzajúca sa v súbore `main.cpp`. Z nej je volaná hlavná metóda analýzy `Controller.startAnalysis`.

4.2.1 Nastavenie parametrov analýzy

Už v návrhu bolo uvedených niekoľko vecí, ktoré môže užívateľ spúšťajúci analýzu nastaviť. Keďže ich je dosť veľký počet (a pri prípadnom rozširovaní programu môže tento počet ešte narastať), nastavovanie pomocou parametrov príkazového riadku by bolo nepraktické. Preto je činnosť analyzátora nastavená v konfiguračnom súbore.

Jedná sa o riadkovo založený konfiguračný súbor, kde každý riadok obsahuje jedno nastavenie v tvare:

```
<nazov-nastavenia> = <hodnota>
```

Riadky začínajúce znakom `#` sú považované za komentáre a ich obsah je ignorovaný. Zoznam podporovaných nastavení (musia byť uvedené všetky v ľubovoľnom poradí):

- `program` – názov a parametre spúšťania analyzovaného programu; v podstate sa jedná o príkaz, ktorým bude analyzovaný program spúšťaný
- `control` – zoznam názvov funkcií, ktorých volania budú riadiacim programom ovládané (tj. tieto volania budú spúšťané s rôznymi variantami); možnosti hodnoty tohoto nastavenia sú:
  - `all` – zahrnuté budú všetky podporované funkcie
  - `none` – zahrnuté nebudú nijaké funkcie
  - zoznam názvov funkcií oddelených čiarkou – budú zahrnuté všetky uvedené funkcie
- `notify` – zoznam názvov funkcií, ktorých volania budú riadiacim programom iba sledované; možnosti hodnoty tohoto nastavenia sú rovnaké, ako v prípade nastavenia `control`
- `variants` – zoznam skupín variant, ktoré budú použité pri ovládaní volaní funkcií uvedených v nastavení `control`; jednotlivé názvy skupín sú medzi sebou oddelené čiarkou a ich možné hodnoty sú:
  - `inval` – simulácia chýb neplatnej hodnoty, konkrétne `EBADF` (zlý popisovač súboru) a `EINVAL` (neplatná hodnota)

  - `io` – simulácia vstupno-výstupnej chyby (`EIO`)
  - `access` – simulácia chybných prístupových práv k súboru (`EACCES`)
  - `memory` – simulácia chýb súvisiacich s pamäťou, konkrétne `EFAULT` (chybný ukazovateľ) a `ENOMEM` (nedostatok pamäte)
  - `interrupt` – simulácia prerušenia volania funkcie externým signálom (`EINT`)
  - `path` – simulácia chýb súvisiacich s cestou k súboru zadanou ako parameter, konkrétne `ENAMETOOLONG`, `ENOENT` a `ENOTDIR`
  - `limits` – simulácia chýb súvisiacich so systémovými limitmi, konkrétne `EDQUOT`, `EFBIG`, `ENOSPC`, `EMFILE`, `ENFILE` a `EMLINK`
  - `permissions` – simulácia chýb súvisiacich s možnosťami a povoleniami súborového systému, konkrétne `EPERM` (súborový systém nepodporuje požadovanú činnosť) a `EROFS` (súborový systém je iba na čítanie)
  - `file` – simulácia chýb súvisiacich so súborom uvedenom ako parameter volania, konkrétne `EISDIR`, `EEXIST`, `ELOOP` a `EBUSY`

- `scheduler` – typ použitého plánovača (scheduler); v implementácií je jedinou podporovanou možnosťou hodnota `bfs`, ktorá nastaví použitie BFS plánovania popísaného v sekcií 3.3.5

- `aggregator` – typ použitej agregácie, jednotlivé typy sú popísané v sekcií 3.3.6; v implementácií sú podporované možnosti:

  - `name` – agregácia podľa mena funkcie
  - `base_param` – agregácia podľa mena funkcie a bázového parametru

- `subroutine` – minimálna veľkosť stopy, ktorá je považovaná za podprogram, resp. telo cyklu; táto hodnota je použitá pri detekcií skokov v grafe toku riadenia popísanej v sekcii 3.3.7; pri uvedení hodnoty 1 sa nijaká detekcia skokov nevykonáva

- `output` – typ výstupu; aktuálne sú použiteľné nasledujúce hodnoty:

  - `dot` – výstupom je zdrojový súbor pre Linuxový program `dot` (z balíčku `graphviz`)
  - `json` – výstupom je reprezentácia grafu pomocou jazyka JSON

- `destination` – cesta k výstupnému súboru (buď absolútna cesta, alebo relatívna ku zložke z ktorej je analyzátor spúšťaný)

K rozboru tohoto konfiguračného súboru dochádza ešte pred spustením samotnej analýzy (pred zavolaním metódy `Controller.startAnalysis`). Má to na starosti trieda *Configuration*, ktorá jednak spracuje konfiguračný súbor a takisto ukladá informácie z neho získané. Zároveň je táto trieda zodpovedná za vypísanie správy `help`. Jej štruktúra sa nachádza na obrázku 4.1.

Trieda obsahuje jednotlivé nastavenia analyzátora. Metóda `parse` dostáva ako argumenty parametre funkcie `main`, spracováva ich, následne číta konfiguračný súbor riadok po riadku a volá metódu `setOption` pre uloženie jednotlivých nastavení.

Configuration

testedProgram: vector<string>
initMsg: InitialMsg
variants: vector<string>
scheduler: string
aggregator: string
subroutine: int
output: string
destination: string
help: boolean
- helpMsg: const string

+ parse(int, char **)
+ printHelp()
- setOption(string, string)

Obr. 4.1: Trieda *Configuration*

4.2.2 Komunikačný modul

Komunikácia je rovnako ako na strane zdieľanej knižnice riešená pomocou *pomenovaných soketov* z knižnice `sys/socket.h`. Pomenovaný soket je umiestnený v súbore `/tmp/analyserSocket`.

Spracovávanie a vytváranie správ sa deje s využitím vstavanej triedy `std::string`, ktorá poskytuje potrebné metódy.

Dôležitou časťou je správne určenie momentu, kedy došlo k ukončeniu analyzovaného programu a už neprídu nijaké správy od zdieľanej knižnice. Keďže ani zdieľaná knižnica nedokáže určiť, ktoré volanie je posledné v analyzovanom programe, musí to určiť riadiaci program na základe toho, že došlo k uzavretiu soketu z druhej strany. Pri prijímaní správy pomocou `recv` to je možné zistiť podľa toho, že volanie vráti hodnotu 0, alebo, že sa v sokete nenachádza žiadna správa. Na druhej strane, pri odosielaní pomocou `send` je použitý príznak `MSG_NOSIGNAL`, aby pri uzavretí druhej strany soketu nebol odoslaný signál `SIGPIPE` a súčasne je zachytený chybový kód `EPIPE`, ktorý znamená práve uzavretie soketu.

4.2.3 Spustenie analyzovaného programu

Pred samotným spustením analyzovaného programu musí byť vytvorený nový proces. Na to je použitá funkcia `fork`. Samotné spustenie zabezpečuje funkcia `execv`, ktorá berie ako argumenty názov spúšťaného súboru a zoznam parametrov v tvare poľa reťazcov jazyka C (dátový typ `char *[]`). Preto je atribút triedy *Tracer*, ktorý ukladá spúšťaný program tohoto dátového typu.

4.2.4 Riadiaca časť

V riadiacej časti je trieda *Controller* implementovaná presne podľa návrhu. Za zmienku tu stojí vytváranie nastavenia prvého behu, ktorý má obsahovať iba sledované funkcie. Preto je zoznam sledovaných funkcií pre tento beh vytvorený ako zjednotenie zoznamov ovládaných a sledovaných funkcií z nastavenia analýzy. Zoznam ovládaných volaní je pre prvý beh prázdny.

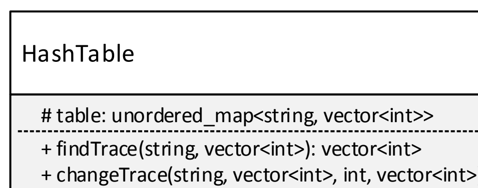
V časti plánovača je rozhranie *Scheduler* implementované ako abstraktná trieda (tj. trieda obsahujúca čisto virtuálne metódy), keďže jazyk C++ neposkytuje rozhrania. Jedinou implementovanou stratégiou plánovania je stratégia založená na algoritme BFS. Zabezpečuje ju trieda *BFSScheduler*, ktorá rozširuje abstraktnú triedu *Scheduler*.

Trieda *Aggregator* je rovnako implementovaná podľa návrhu s tým rozdielom, že metóda `toString`, ktorá má ako argument objekt triedy *Call* a slúži na prevod volania na jeho reťazcovú reprezentáciu, je virtuálna, aby bolo možné doimplementovať ďalšie typy agregácie. Základná trieda *Aggregator* používa agregáciu podľa mena funkcie a zároveň je vytvorená trieda *BaseParamAggregator*, ktorá od tejto triedy dedí a používa agregáciu podľa bázového parametru. Pri použitých funkciách pracujúcich so súbormi je ako bázový parameter väčšinou zvolený popisovač súboru (angl. *file descriptor*). Funkcie, ktoré nepracujú s popisovačom, používajú cestu k súboru, preto je pri nich zvolená táto cesta ako bázový parameter.

Detekcia skokov je implementovaná pomocou hashovacej tabuľky tak, ako je to popísané v návrhu. Keďže trieda *JumpDetector* má na starosti aj úpravy grafu, často musí pristupovať ku chráneným atribútom triedy *Aggregator*. Preto je v triede *Aggregator* označená kľúčovým slovom `friend`, ktoré umožní triede *JumpDetector* priamy prístup ku všetkým členom triedy *Aggregator*. Ďalšou, z implementačného hľadiska dôležitou časťou je samotná hashovacia tabuľka, ktorej sa venuje nasledujúca sekcia.

4.2.5 Implementácia hashovacej tabuľky

Detekcia skokov prebieha s využitím hashovacej tabuľky. Prístup k nej zaisťuje trieda *HashTable*, ktorá obsahuje samotnú tabuľku a takisto metódy potrebné pre prístup k nej. Tabuľka je implementovaná ako kontajner z STL knižnice jazyka C++ (konkrétne normy C++11) `std::unordered_map`. Tento kontajner je indexovaný pomocou reťazcovej reprezetácie stôp (reťazec je prepočítaný vstavanou hashovacou funkciou a výsledné číslo je indexom do tabuľky). Jednotlivé prvky obsahujú vektory celých čísel, ktoré určujú kde v grafe sa daná stopa nachádza (keďže graf je reprezentovaný, ako vektor uzlov, jedná sa o indexy do tohoto vektoru). Štruktúra triedy *HashTable* je znázornená na obrázku 4.2.



Obr. 4.2: Trieda *HashTable*

Trieda obsahuje metódu `findTrace` pre vyhľadanie stopy v tabuľke (reťazcová reprezentácia stopy je prvým argumentom metódy). Ak na indexe stopy neexistuje nijaký záznam je na tento index vložený vektor indexov daný ako druhý argument metódy.

Ďalšou metódou pre prístup k tabuľke je `changeTrace`, ktorá nájde vektor na indexe danej stopy (prvý argument) a ak je zhodný s vektorom indexov daných ako druhý argument metódy, je stopa upravená. Táto úprava sa robí tak, že hodnoty vo vektore indexov, nachádzajucom sa v tabuľke, začínajúce od poradového čísla daného tretím argumentom sú

nahradené vektorom nových hodnôt (štvrtý argument metódy). Táto metóda je používaná pri úprave tabuľky v kroku č. 4 algoritmu detekcie skokov (3.3.7).

4.2.6 Zachytávanie a riešenie chybových stavov

Chybové stavy, ktoré môžu nastať počas analýzy sú ošetrené pomocou mechanizmu výnimiek, ktoré jazyk C++ ponúka. Tieto výnimky sú odchytávané priamo vo funkcií `main`, kde je aj vytvorená reakcia na ne (vačšinou sa jedná o vypísanie chybovej hlášky a ukončenie analýzy).

Aplikácia môže vyvolať niekoľko základných druhov výnimiek:

- **ConfigurationException** – výnimka je vyvolaná v prípade chybnej syntaxe konfiguračného súboru
- **ProtocolException** – výnimka je vyvolaná v prípade chyby protokolu (prijatá nesprávny typ správy apod.); tento typ výnimky by pri správnom chode aplikácie nemal nastať
- **SocketException** – výnimka je vyvolaná pri chybe jednej z funkcií zaisťujúcich komunikáciu cez soket
- **SocketClosedException** – výnimka je vyvolaná v prípade, že došlo k uzavretiu soketu zo strany zdieľanej knižnice (tj. k ukončeniu behu analyzovaného programu)

4.2.7 Správa pamäte

Keďže jazyk C++ neobsahuje mechanizmy na automatické uvoľňovanie pamäte, je nutné aby toto bolo zaistené ručne programátorom. Na to sa využívajú špeciálne metódy tried známe ako *deštruktory*. Takáto metóda je volaná vždy, keď má daný objekt zanikúť. Staticky vytvorené objekty sú rušené automaticky, ale tie dynamicky vytvorené musia byť rušené explicitným volaním operácie `delete`.

Nasledujúci zoznam obsahuje informácie o tom, ktoré triedy sú zodpovedné za rušenie ktorých objektov:

- Trieda *Call* je zodpovedná za rušenie všetkých svojich objektov typu *Param*
- Trieda *Run* je zodpovedná za rušenie objektov *Call*, ktoré obsahuje. Platí, že každé volanie patrí práve do jednoho behu, vďaka tomu sú všetky volania zrušené.
- Trieda *Controller* ruší všetky behy uložené ako objekty triedy *Run*. Takisto je zodpovedná za rušenie objektov tried *Socket*, *Scheduler* a *Aggregator*.
- Trieda *Aggregator* ruší všetky uzly grafu toku riadenia, ktorý obsahuje. Taktiež ruší objekt triedy *JumpDetector*.
- Trieda *Tracer* ruší uložený názov a parametre spúšťania analyzovaného programu, keďže sú uložené v tvare poľa reťazcov jazyka C (`char *[]`).

Kapitola 5

Opis experimentov a interpretácia výsledkov

Táto kapitola obsahuje opis niekoľkých jednoduchých experimentov s vytvoreným analyzátorom. Ich cieľom bolo zistiť, či analyzátor správne zachytáva jednotlivé volania a či vytvorený graf toku riadenia programu zodpovedá tomu reálnemu (vytvorenému ručne zo zdrojového kódu). Ďalšie experimenty sú vytvorené na preskúmanie rozdielneho správania analyzátora pri zmenách v konfigurácií.

5.1 Analýza jednoduchého programu

Prvým experimentom je spustenie analyzátora na testovacom programe. Cieľom experimentu je zistiť, či analyzátor správne zachytí všetky volania a vetvenia v grafe toku riadenia. Analyzovaný program obsahuje pre prehľadnosť iba niekoľko základných volaní. Hlavná časť programu (bez deklarácie premenných a hlavičky funkcie `main`) je nasledujúca:

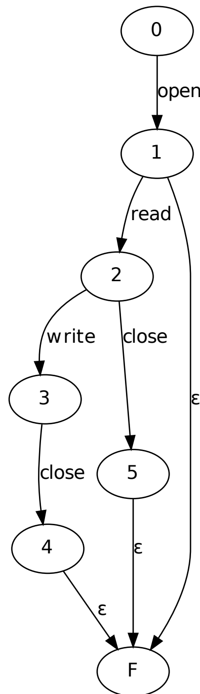
```
fd = open("tst/01/file.txt", O_RDONLY);
if (fd >= 0){
   size = read(fd, buf, 20);
   if (size >= 0)
      write(1, buf, size);
   close(fd);
}
```

Po spustení analýzy tohoto programu s jednoduchou konfiguráciou (tj. sledovanie a ovládanie všetkých funkcií, agregácia podľa mena funkcie) je výsledný získaný graf na obrázku 5.1.

Porovnaním obrázku a zdrojového textu zistíme, že graf presne zachytáva jednotlivé volania a vetvenia, ktoré sa v programe vyskytujú.

5.2 Analýza cyklu

V druhej časti sa bude jednať o sériu experimentov zameraných na zistenie toho, či je analyzátor schopný správne nájsť cykly nachádzajúce sa v analyzovanom programe. Ten bude podobný, ako program v prvom experimente s tým rozdielom, že uvedený blok kódu



Obr. 5.1: Graf toku riadenia jednoduchého programu

bude volaný dva krát za sebou. V každom volaní tohoto bloku bude otváraný a čítaný iný súbor, ale keďže používame agregáciu iba podľa mena volania, analyzátor by mal považovať bloky za rovnaké.

Nastavenie konfigurácie zostáva rovnaké, ako v prvom prípade, teraz je však dôležité nastaviť minimálnu dĺžku podprogramu. Toto nastavenie sa bude medzi jednotlivými experimentami v sérií meniť, keď v prvom bude nastavená presná dĺžka tela cyklu, v druhom nižšia hodnota a v treťom zasa vyššia hodnota.

5.2.1 Nastavenie správnej dĺžky podprogramu

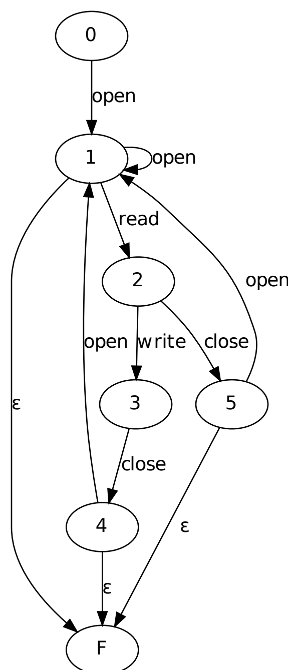
Cieľom tohoto experimentu je zistiť, či je analyzátor schopný zachytiť cyklus nachádzajúci sa v analyzovanom programe. Uvedený blok kódu sa skladá zo štyroch volaní (`open`, `read`, `write` a `close`), preto nastavíme minimálnu dĺžku podprogramu na hodnotu 4. Získaný graf sa nachádza na obrázku 5.2.

Porovnaním získaného grafu s grafom z experimentu 1 je vidieť, že analyzátoru sa podarilo nájsť opakujúcu sa postupnosť volaní a vytvoriť zodpovedajúce skoky v grafe, konkrétne z uzlov `1`, `4` a `5`, späť na začiatok bloku, tj do uzlu `1`.

5.2.2 Nastavenie menšej dĺžky podprogramu

V ďalšom experimente nastavíme menšiu dĺžku podprogramu, ako je počet volaní v cykle. Keďže sa však má jednať o **minimálnu** dĺžku podprogramu, analyzátor by sa mal správať rovnako, ako v predchádzajúcom experimente a nemalo by dôjsť ku zmene výsledného grafu. Cieľom experimentu bolo zistiť, či to tak v skutočnosti je.

Po spustení experimentu s minimálnou dĺžkou podprogramu rovnou 2 sa zistilo, že získaný graf je zhodný s grafom na obrázku 5.2. To znamená, že analyzátor dokázal detekovať



Obr. 5.2: Graf toku riadenia programu obsahujúceho cyklus

aj cyklus, ktorého dĺžka tela je väčšia, ako zadaná minimálna dĺžka podprogramu.

5.2.3 Nastavenie väčšej dĺžky podprogramu

V poslednom experimente v sérií bude nastavená väčšia minimálna dĺžka podprogramu, ako je tá správna. Cieľom bude zistiť, či analyzátor skutočne nezachytí existujúci cyklus.

Experiment bol spustený s nastavením minimálnej dĺžky podprogramu 10. Získaný graf sa nachádza v prílohe C na obrázku C.1. Tento graf má tvar stromu, kde každý uzol má práve jednoho predchodcu. To znamená, že nedošlo k detekcii nijakých skokov.

5.3 Porovnanie rôznych druhov agregácie

V tejto časti sa nachádza ďalšia séria experimentov, tentokrát zameraná na porovnávanie výstupov analyzátora pri rôznych nastaveniach typu agregácie. V implementácií analyzátora sú k dispozícií dva druhy agregácie, konkrétne *podľa názvu funkcie* a *podľa bázového parametru*, takže porovnávanie bude realizované práve medzi nimi. V každom z experimentov bude analýza spustená dva krát na rovnakom programe iba so zmenou nastavenia typu agregácie.

5.3.1 Program pracujúci s jedným súborom

Prvý experiment bude realizovaný na programe, ktorý pracuje iba s jedným súborom a navyše neobsahuje nijaký cyklus ani skoky. Cieľom experimentu je zistiť základný rozdiel medzi dvoma typmi agregácie. Ako analyzovaný program bol zvolený Linuxový program `cat`, ktorý otvorí súbor a vypíše jeho obsah. Najskôr bol tento program spustený s nastavením agregácie podľa názvu funkcie a potom podľa bázového parametru. Získané grafy sa

nachádzajú v prílohe C na obrázku C.2.

Pri porovnaní získaných grafov zistíme, že ich štruktúra je rovnaká, jediným rozdielom je množstvo informácií priradených ku hrane. V tomto prípade teda agregácia podľa bázového parametru nemení tvar grafu, iba poskytuje väčšie množstvo informácií o volaniach (okrem názvu funkcie aj hodnotu bázového parametru).

5.3.2 Program obsahujúci cyklus

V druhom experimente bude analyzovaný program používaný v druhej sade experimentov. Ako sme už videli, pri správnom nastavení minimálnej dĺžky podprogramu je analyzátor schopný v tomto programe nájsť skoky, keďže daná postupnosť volaní sa v ňom nachádza dva krát za sebou. Keďže sú však otvárané súbory s rozdielnym názvom, pri agregácií podľa bázového parametru by malo dôjsť k oddeleniu niektorých volaní. Cieľom tohoto experimentu bude zistiť, či sa tak naozaj stane. Grafy získané počas experimentu sa nachádzajú v príohe C na obrázku C.3.

Môžeme vidieť, že kým v prvom prípade graf obsahuje cyklus, v druhom tomu tak nie je. V tomto grafe došlo k oddeleniu dvoch blokov kódu, keďže jednotlivé volania `open` prebehli nad rôznymi súbormi.

V druhom grafe navyše stojí za povšimnutie rozdiel medzi dvom časťami grafu, ktoré by mali mať rovnaký tvar. V prvej existuje hrana z uzlu č. `2` do uzlu č. `4`, ale v tej druhej zodpovedajúca hrana nevedie z uzlu `6` do uzlu `8`, ale do uzlu `9`. V prvom prípade došlo k vytvoreniu skoku, kým v druhom nie. Je to z toho dôvodu, že za uzlom č. `4` existuje dosť dlhá postupnosť volaní, ktorá bola vložená do hashovacej tabuľky (konkrétne postupnosť uzlov `4,5,6,7`). Potom keď za uzlom `2` vznikla rovnaká stopa, bol vytvorený skok. Za uzlom č. `8` program končí, preto nemohla byť zachytená dosť dlhá stopa.

5.3.3 Program pracujúci s viacerými súbormi

Posledným experimentom v tejto sérií je analýza programu, ktorý pracuje s viacerými súbormi. Cieľom bude opäť zistiť rozdiel medzi agregáciou podľa mena a podľa bázového parametru. Ako analyzovaný program bol zvolený Linuxový program `cp`, ktorý slúži na kopírovanie súborov.

Získané grafy sú dosť rozsiahle, ale časť v ktorej sa líšia je zobrazená na obrázku C.4. Tu si môžeme všimnúť, že analyzátor v prvom prípade nesprávne agregoval dve volania `open` z uzlu `3` nad rôznymi súbormi do jedného volania. Prejavil sa teda rozdiel medzi dvoma druhmi agregácie, keď agregácia podľa mena bola chybná.

5.4 Dĺžka analýzy v závislosti na konfigurácií

V poslednej časti experimentov sa zameriame na meranie trvania analýzy a veľkosti výstupného grafu v závislosti na rôznych nastaveniach analýzy (prevažne na minimálnej dĺžke podprogramu a na type agregácie). Cieľom experimentov bude zistiť, pri akej konfigurácií trvá analýza najkratší a pri akej najdlhší čas.

Ako analyzovaný program je zvolený Linuxový program `find` na vyhľadávanie súborov. Tento program je dostatočne rozsiahly na to, aby bolo možné reálne merať rozdiely v dĺžke jeho behu. Analýza je postupne spúšťaná s nastavením minimálnej dĺžky podprogramu `2`, `6` a `10` pre agregáciu podľa názvu funkcie a potom s rovnakými dĺžkami pre agregáciu podľa bázového parametru. Pri každej analýze je meraný čas jej behu pomocou nástroja `time` a

takisto zistený počet uzlov grafu toku riadenia. Získané výsledky sú zobrazené v tabuľke 5.1.

Minimálna dĺžka podprogramu	**Typ/čas/veľkosť agregácie**	
	Názov funkcie	**Bázový parameter**
10	5m0.529s, 70 uzlov	10m0.176s, 376 uzlov
6	1m2.903, 24 uzlov	8m55.564s, 175 uzlov
2	0m58.708, 13 uzlov	7m57.846s, 71 uzlov

Tabuľka 5.1: Čas a veľkosť agregácie v závislosti na rôznych nastaveniach analýzy

Preskúmaním tabuľky si môžeme všimnúť niekoľko skutočností. Čím je väčšia minimálna dĺžka podprogramu, tým je generovaný väčší počet uzlov. Je to spôsobené tým, že pri menších dĺžkach je nájdených viac podprogramov, tým pádom viac skokov a to znamená, že sa nevytvára také množstvo uzlov, ako pri väčších dĺžkach.

Pri porovnaní typov agreagácií je jasné, že pri agregácií podľa bázového parametru vzniká oveľa viac uzlov. Toto správanie je samozrejme vysvetliteľné faktom, že pri tomto type agregácie je menej dvojíc uzlov považovaných za zhodné a teda ich celkovo vzniká väčší počet.

Ďalej je možné pozorovať skutočnosť, že s rastúcim počtom uzlov rastie aj celková dĺžka analýzy, hoci nie priamo úmerne. Napríklad pri agregácií podľa bázového parametru s minimálnou dĺžkou podprogramu 2 vznikol takmer rovnaký počet uzlov, ako pri agregácií podľa názvu funckie a minimálnej dĺžke podprogramu 10, ale analýza trvala v prvom prípade o 60% viac času. Z toho je jasné, že dĺžka analýzy závisí aj od celkovej zložitosti grafu a počtu vytváraných skokov (ktorých muselo byť v prvom prípade viac).

Kapitola 6

Záver

Cieľom práce bolo vytvoriť automatický dynamický analyzátor. Ten má za úlohu sledovať volania vybraných knižnicových volaní analyzovaným programom. Ďalej tieto volania ovláda za účelom získania rôznych behov programu, ktoré nakoniec agreguje do výsledného grafu toku riadenia. Analyzátor je určený pre operačný systém Linux. Sleduje a ovláda volania s využitím vlastnej zdieľanej knižnice a nastavním premennej prostredia `LD_PRELOAD`. Okrem zdieľanej knižnice obsahuje analyzátor riadiaci program, ktorý ovláda priebeh celej analýzy.

Práca v prvom rade obsahuje podrobný návrh zdieľanej knižnice a riadiaceho programu tak, aby ich bolo možné implementovať v ľubovoľnom programovacom jazyku, ktorý vyhovuje podmienkam návrhu (pre zdieľanú knižnicu možnosť preložiť zdrojový kód, ako zdieľaný objekt a pre riadiaci program objektový prístup). Celý návrh je vytvorený tak, aby bolo možné analyzátor kedykoľvek rozširovať bez nutnosti veľkých zmien v existujúcej implementácií. Toto rozširovanie môže zahŕňať pridávanie a odoberanie sledovaných volaní, či implementáciu nových stratégií riadenia analýzy a agregácie behov.

Súčasťou práce je tiež implementácia analyzátora v jazykoch C/C++. Táto implementácia sa zameriava na sledovanie volaní nad súborovým systémom. Ich ovládanie je väčšinou realizované simuláciou chybových stavov týchto volaní. Takisto je urobených niekoľko jednoduchých experimentov s vytvoreným analyzátorom, ktoré demonštrujú jeho funkcionalitu.

Celý projekt ponúka množstvo možností rozšírenia, ktoré môže zahŕňať implementáciu pokročilých algoritmov pre riadenie analýzy, agregácie volaní, alebo detekcie skokov, ďalej rôzne optimalizácie behu analýzy, či nové prístupy pri ovládaní volaní.

Literatúra

[1] The GNU C Library (glibc). http://www.gnu.org/software/libc/libc.html, 2013-08-12 [cit. 2014-04-14].

[2] Ball, T.: The Concept of dynamic analysis. http://research.microsoft.com/en-us/um/people/tball/papers/fse-concept.pdf.

[3] Fleury, E.; Point, G.; Vincent, A.: Binary Program Analysis: Theory and Practice. http://www-verimag.imag.fr/ async/CCIS/talk_13/Fleury.pdf, 2013-06-13.

[4] Jones, M. T.: Anatomy of Linux dynamic libraries. http://www.ibm.com/developerworks/library/l-dynamic-libraries/l-dynamic-libraries-pdf.pdf, 2008-08-20.

[5] Wögerer, W.: A Survey of Static Program Analysis Techniques. http://www.ics.uci.edu/ lopes/teaching/inf212W12/readings/Woegerer-progr-analysis.pdf, 2005-10-18.

[6] Zeller, A.: Program Analysis: A Hierarchy. http://www.eecs.yorku.ca/course_archive/2004-05/F/6431/ZellerErnst.pdf.

Dodatok A

Adresárová štruktúra projektu

Na priloženom CD sa nachádzajú zdrojové kódy analyzátoru. Adresárová štruktúra tohoto CD je nasledujúca:

```
/
├── bin/
├── doc/
│   ├── doxygen/
│   └── thesis/
├── obj/
├── out/
│   ├── graph/
│   └── json/
├── src/
│   ├── Analyzer/
│   │   ├── Aggregator/
│   │   ├── Exceptions/
│   │   ├── Parser/
│   │   └── Scheduler/
│   └── SharedLib/
├── Makefile
└── settings.conf
```

Význam a obsah jednotlivých adresárov:

bin spustiteľné súbory, konkrétne samotný analyzátor a zdieľaná knižnica

doc dokumentácia ku projektu

> **doxygen** zložka pre dokumentáciu vytvorenú programom `doxygen`; obsahuje konfiguračný súbor a dokumentáciu vo formáte HTML;
>
> **thesis** elektronická podoba tejto práce vo forme `pdf` a zdrojových súborov pre program LaTeX

obj objektové súbory vytvárané počas prekladu aplikácie

out výstupné súbory rozdelené do adresárov podľa typu

src zdrojové súbory projektu

- **Analyzer** zdrojové súbory riadiaceho programu; táto zložka obsahuje triedy *Call*, *Configuration*, *Controller*, *Param*, *Run*, *Socket*, *Trace*, *Tracer* a súbor s úvodnou funkciou `main.cpp`
  - **Aggregator** triedy *Aggregator*, *BaseParamAggregator*, *GraphNode*, *HashTable*, *JumpDetector*
  - **Exceptions** triedy výnimiek *ConfigurationException*, *ProtocolException*, *ProtocolClosedException*, *SocketException*
  - **Parser** triedy *InboundMsg*, *InitialMsg*, *Message*, *OutboundMsg*
  - **Scheduler** triedy *BFSScheduler*, *Scheduler*
- **SharedLib** zdrojové súbory zdieľanej knižnice

Makefile súbor pre preklad projektu pomocou nástroja `make`

settings.conf súbor s nastaveniami analýzy

Každá trieda je reprezentovaná dvojicou zdojových súborov – hlavičkového súboru `<nazov-triedy>.h`, ktorý obsahuje definíciu triedy a súboru `<nazov-triedy>.cpp`, ktorý obsahuje implementáciu metód triedy.

Dodatok B

Preklad a spustenie programu

Na preklad aplikácie je využitý nástroj `make`. Súbor `Makefile` je priložený v koreňovom adresári projektu. Pre spustenie prekladu stačí zadať príkaz:

```
make all
```

Samotný preklad sa deje prekladačom `gcc` s využitím automatickej tvorby závislostí, ktorú tento prekladač ponúka pri použití prepínačov `-MMD` a `-MP`. Pre každú triedu sú jej závislosti uložené do súboru `<nazov-triedy>.d` uloženého do adresára `obj`.

Zdieľaná knižnica je prekladaná ako zdieľaný objekt použitím prepínačov `-shared` a `-fPIC`.

Ďalšie možnosti prekladu pomocou `make`:

- `make analyzer` (alebo iba `make`) – preloží iba riadiaci program
- `make lib` – preloží iba zdieľanú knižnicu
- `make clean` – odstráni objektové a binárne súbory
- `make doc` – vytvorí dokumentáciu vo formáte HTML pomocou programu `doxygen` (program musí byť nainštalovaný na danom počítači)
- `make graph` – vytvorí graf pomocou programu `dot` (v systéme musí byť nainštalovaný balíček `graphviz`); podmienkou je, že zdrojový súbor sa nachádza v priečinku `out/graph` a má názov `graph.gv`; výstup je uložený do rovnakého priečinku pod názvom `graph.pdf`

Po úspešnom preklade je program možné spustiť príkazom:

```
bin/analyzer CONFIG-FILE
```

Ako parameter je nutné uviesť umiestnenie konfiguračného súboru s nastaveniami analýzy. Príklad obsahu konfiguračného súboru:

```
# program name
program = tst/01/main
# function lists
```

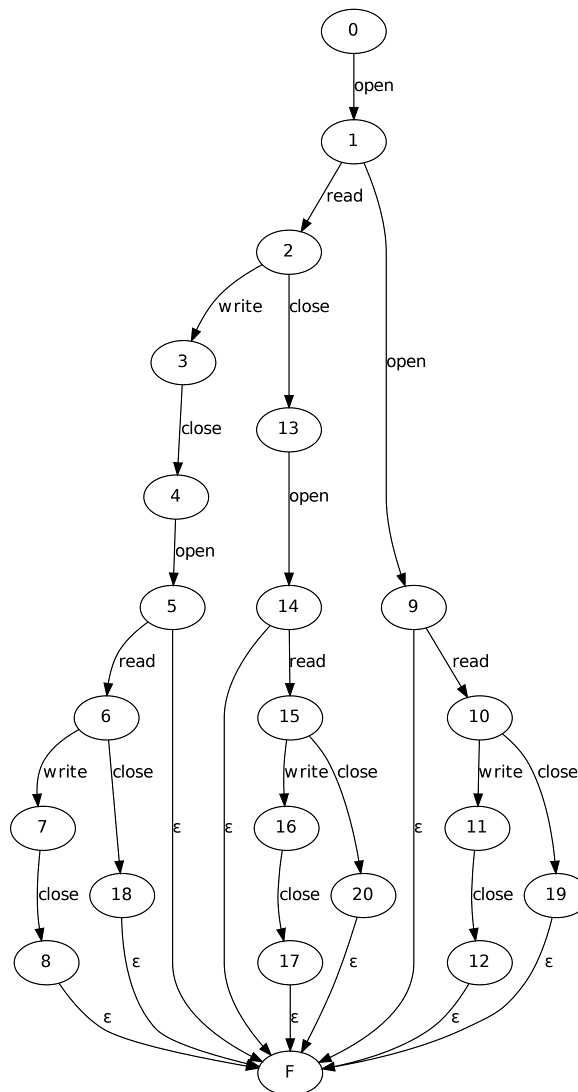
```
control = read,write,open,close
notify = all
# variants
variants = access,memory,path,limits
# scheduler type
scheduler = bfs
# aggregator type
aggregator = name
# minimal subroutine size
subroutine = 4
# output
output = dot
destination = out/graph/graph.gv
```

Pri spustení programu s parametrom `--help` sa vypíše nápoveda.

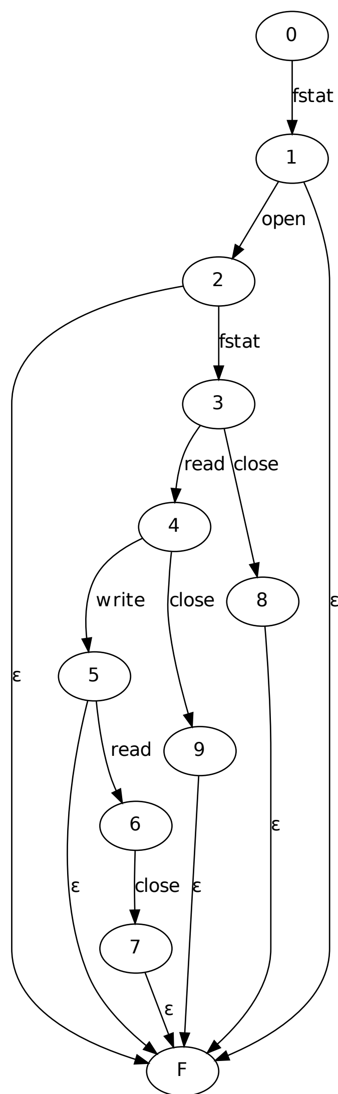
Dodatok C

Grafy získané počas experimentov

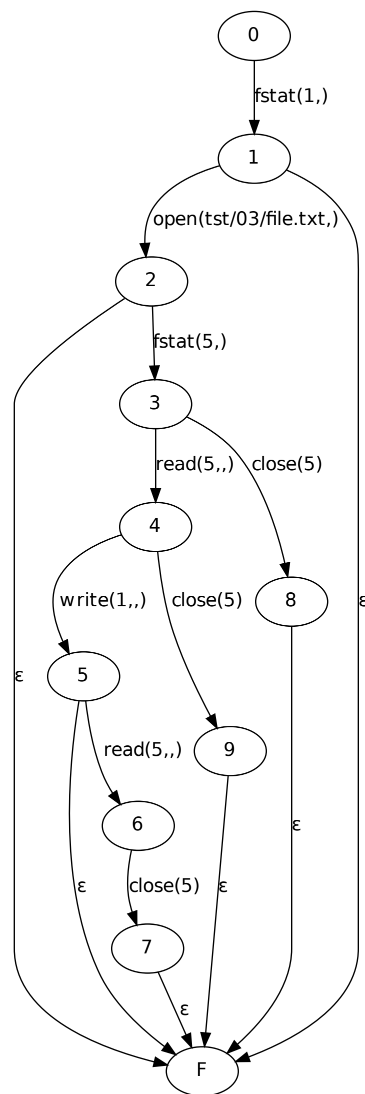
Táto príloha obsahuje jednotlivé grafy získané počas experimentov realizovaných v kapitole 5. Jednotlivé obrázky sú odkazované z tejto kapitoly.



Obr. C.1: Detekcia cyklov s nastavením veľkej minimálnej dĺžky podprogramu

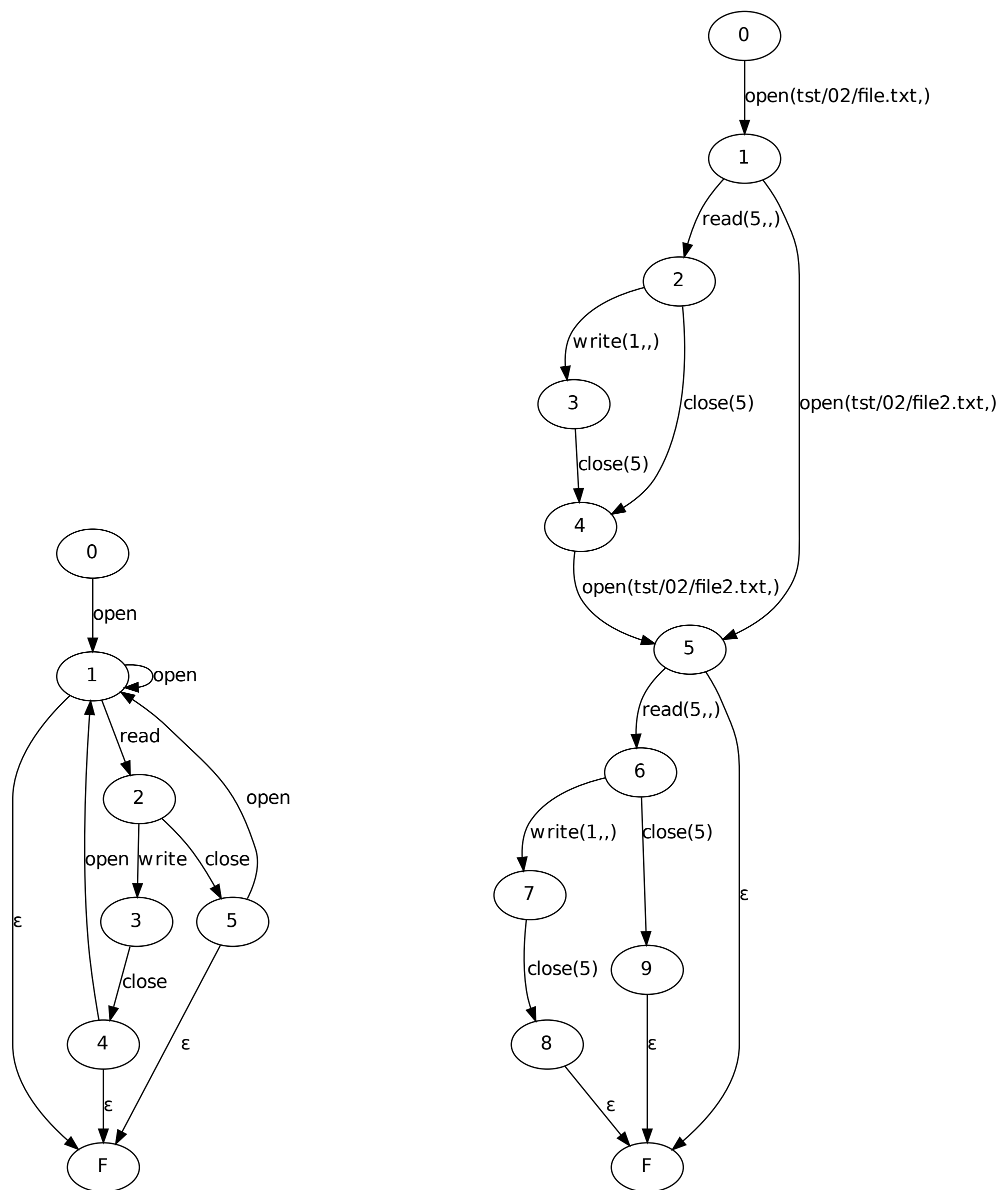


(a) Agregácia podľa názvu funkcie



(b) Agregácia podľa bázového parametru

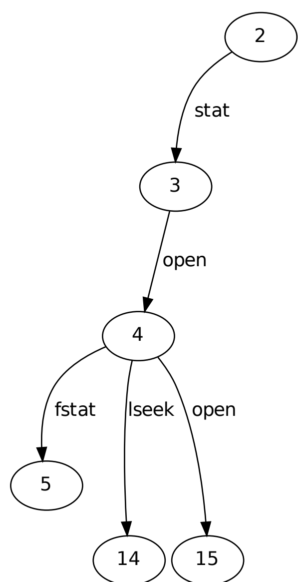
Obr. C.2: Rôzne druhy agregácie nad jednoduchým programom



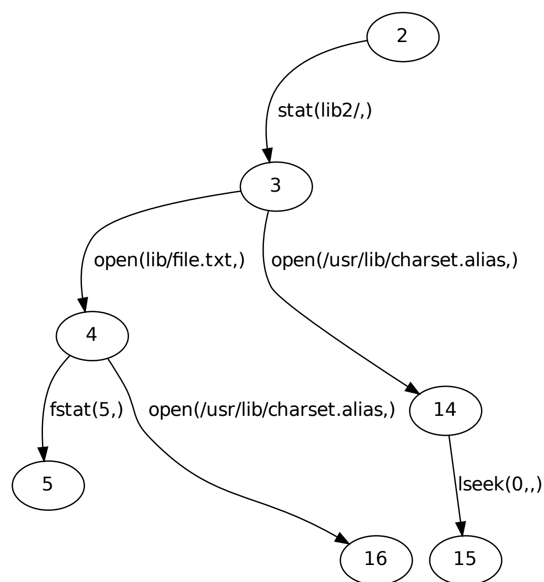
(a) Agregácia podľa názvu funkcie

(b) Agregácia podľa bázového parametru

Obr. C.3: Rôzne druhy agregácie nad programom obsahujúcim skoky



(a) Agregácia podľa názvu funkcie



(b) Agregácia podľa bázového parametru

Obr. C.4: Rôzne druhy agregácie nad programom pracujúcim s viacerými súbormi