VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO VYHLEDÁVÁNÍ TESTOVACÍCH CEST

TOOL FOR SEARCHING FOR TEST PATHS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MICHAL STUPAR**
AUTHOR

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

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

145048

Ústav: Ústav inteligentních systémů (UITS)
Student: **Stupar Michal**
Program: Informační technologie
Specializace: Informační technologie

Název: **Nástroj pro vyhledávání testovacích cest**
Kategorie: Analýza a testování softwaru
Akademický rok: 2022/23

Zadání:

1. Nastudujte testování založené na modelech. Nastudujte kritéria pokrytí grafů.
2. Navrhněte nástroj pro vyhledávání testovacích cest splňující uživatelem vybrané kritérium pokrytí. Nástroj by měl zahrnovat i možnost doplnění existující testovací sady za účelem úplného pokrytí. Při návrhu předpokládejte existenci nástroje pro zjišťování sémantické splnitelnosti testovací cesty.
3. Implementujte nástroj jako knihovnu nebo modul a vytvořte k němu rozhraní příkazové řádky.
4. Vytvořte sadu automatických testů ověřující všechna podporovaná pokrytí.

Literatura:

- Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, 2008, 322 s. ISBN 978-0-511-39330-3.
- ISO/IEC/IEEE 29119-4:2015(E) Software and system engineering -- Software testing -- Test techniques.

Při obhajobě semestrální části projektu je požadováno:
První dva body zadání

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

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 10.5.2023
Datum schválení: 3.11.2022

Abstrakt

Práca obsahuje úvod do problematiky testovania založeného na modeloch, návrh frameworku pre vyhľadávanie testovacích ciest nad grafom toku riadenia, implementáciu tried a metód, ktorých realizácia bola overená sadou automatických testov. Vyhľadávanie ciest v grafe je vykonávané pomocou algoritmov prehľadávania do šírky a prehľadávania do hĺbky, ktoré sa snažia o nájdenie testovacích ciest pre splnenie štyroch implementovaných kritérií pokrytia grafu. Nástroj TRIP pri svojej činnosti využíva dva externé moduly, *GCC plugin* pre získanie grafu toku riadenia a *SMT solver* pre zistenie ohodnotenia cesty. Komunikácia medzi modulmi prebieha pomocou JSON formátu, ktorý sa využíva aj pre ukladanie nájdených ciest medzi jednotlivými behmi nástroja.

Abstract

The work contains an introduction into problematics model-based testing, framework design for searching test paths over a control-flow graph, the implementation of classes and methods, the implementation of which was verified by a set of automatic tests. Searching path in graphs using breadth-first search and depth-first search algorithms that try to find test paths for performance of the four implemented graph coating criterias. Tool TRIP uses two externals modules - GCC plugin for obtaining the control-flow graph and the SMT solver for determining the evaluation of the path. Communication between moduls is JSON format, which is also used for saving found paths between individual runs of the tool.

Kľúčové slová

testovanie založené na modeloch, kritéria pokrytia, graf toku riadenia, generovanie testovacej sady, nástroj TRIP

Keywords

model-based testing, coverage criteria, control-flow graph, test suite generation, tool TRIP

Citácia

STUPAR, Michal. *Nástroj pro vyhledávání testovacích cest.* Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Nástroj pro vyhledávání testovacích cest

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Aleša Smrčku, Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Michal Stupar
7. mája 2023

Poďakovanie

Rád by som sa poďakoval vedúcemu práce Ing. Alešovi Smrčkovi Ph.D. za cenné rady, spoločné konzultácie a celkový prínos pri tvorbe bakalárskej práce. Taktiež by som sa rád poďaloval členom tímu TRIP, ktorí sa podielali na celkovom vývoji nástroja a to konkrétne Františekovi Lukešovi a Jánovi Rybkovi.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

Úvod

Posledné roky môžeme pozorovať takmer nezastaviteľný pokrok v oblasti informačných technológií. Pred pár rokmi si ľudia medzi sebou vymieňali informácie pomocou listov, neskôr prišli telegramy, počítače, mobilné telefóny až po správy odosielané po celom svete v reálnom čase. Všetky tieto vymoženosti nám uľahčujú a spríjemňujú každodenný život, ale spolu s tým prichádza aj potenciálne riziko ich nedokonalosti a chybovosti. Testovanie programov sa práve preto stáva čoraz dôležitejšou súčasťou vývoja softvéru a spotrebováva sa pri tom nemalé množstvo zdrojov, energie a času.

Všetky procesy sa dajú optimalizovať a testovanie softvéru nie je žiadnou výnimkou. Automatizáciu testovania softvéru dokážeme znížiť potrebné množstvo energie a zároveň eliminovať tak potenciálne nedostatky softvéru ešte pred samotným vydaním.

Cieľom tejto práce bolo navrhnúť a implementovať framework, ktorý by bol vhodným prostredím na automatické generovanie testovacích sád na základe užívateľom zvoleného kritéria pokrytia. Avšak najväčšou motiváciou bolo vytvoriť jednoduchý a hlavne voľne prístupný nástroj, na ktorom by si mohli budúci študenti predmetu *Testovanie a dynamická analýza*[1] prakticky vyskúšať automatické generovanie testovacích sád.

Nástroj pre vyhľadávanie testovacích ciest pri svojej činnosti využíva graf toku riadenia programu, nad ktorým vytvára testovacie sady. Podporuje viacero druhou pokrytia grafu ako napríklad pokrytie všetkých uzlov, hrán, páru hrán alebo hlavých ciest. Pri vytváraní testovacej cesty je dôležitá aj jej sémantická splniteľnosť, ktorej popis a dôležitosť sa nachádza v časti 2.3.

Úvodná kapitola 2 opisuje úvod do testovania, testovanie založené na modeloch, graf toku riadenia, kritéria pokrytia ale aj prehľad súčasných riešení. Kapitolou 3 je samotný návrh nástroja, požiadavky kladené na nástroj, popis používaných JSON formátov, návrh architektúry a generátorov. V kapitole 4 sa nachádzajú všetky potrebné a zaujímavé implementačné detaily nástroja. Predposlednou kapitolou 5 je popis automatických testov vyhľadávacieho nástroja. Zhodnotenie návrhu, implementácie a využiteľnosti sa nachádza v poslednej kapitole 6.

[1] https://www.fit.vut.cz/study/course/231029/

Kapitola 2

Testovanie založené na modeloch

Testovanie založené na modeloch (z ang. model-based testing) je jedným z mnohých prístupov k testovaniu systémov. Princíp samotného testovanie založeného na modeloch zahŕňa vytváranie modelu systému, ktorý predstavuje rôzne stavy reálneho systému, prechody a udalosti medzi nimi. Model systému by mal čo najpresnejšie reprezentovať reálny systém, aby bolo možné čo najefektívnejšie vytvorenie testovacích prípadov, ktoré by odhalili potenciálne nedostatky systému. Vytvorené modely sa využívajú ako reprezentácia požadovaného správania testovaného systému (z ang. system under test) [8].

Spomínaný prístup k testovaniu systémov ponúka radu výhod. Medzi tie najväčšie výhody patrí možnosť jeho automatizácie. Testovacie prípady sa generujú automaticky na základe predloženého modelu a typu testovacieho kritéria. Jednoduchá škálovateľnosť, nakoľko generovanie testovacej sady a jej následne prevedenie dokáže byť plne automatické bez ohľadu na testovaný systém. Celkovo ľahká správa a priebeh samotného testovania. Z testovania je možné získať reporty o prevedení testovacej sady s jej následným vyhodnotením.

Pri testovaní založeného na modeloch sa pre generovanie testovacích prípadov využíva graf toku riadenia, ktorého podrobnejší opis sa nachádza v časti 2.1. Graf toku riadenia reprezentuje akýsi model reálneho testovaného systému. Nad grafom toku riadenia programu sa vytvárajú testovacie cesty, ktorých úlohou je dosiahnutie istej úrovne pokrytia grafu. Popis jednotlivých druhov pokrytia grafu je v časti 2.2. Prístup k automatickému získavaniu grafu toku riadenia sa nachádza v kapitole 2.4.

2.1 Základné pojmy graf toku riadenia

Graf toku riadenia (z ang. control-flow graph) je definovaný ako štvorica $G = (N, N_0, N_f, E)$, kde:

- N je konečná množina uzlov,
- $N_0 \subseteq N$, $N_0 \neq \emptyset$ je neprázdna množina vstupných uzlov,
- $N_f \subseteq N$ je množina koncových uzlov,
- $E \subseteq N \times N$ je množina hrán.

Každý uzol grafu predstavuje tzv. *základný blok* kódu (z anglického basic block). Pod pojmom *základný blok* rozumieme ucelenú časť kódu, ktorá priamočiaro nasledujú za sebou

a nie je tvorená žiadnymi skokmi alebo cieľmi iných skokov. Cieľmi skokov sú ohraničené jednotlivé základné bloky. Každý základný blok začína cieľom skoku a je ukončený skokom na ďalší blok. V grafe toku riadenia existujú dva špeciálne bloky. Prvým špeciálnym blokom je vstupný blok, ktorým celé riadenie vstupuje do grafu a druhým je výstupný blok, cez ktorý riadenie vystupuje z grafu.

- *Cesta v grafe* je neprázdna sekvencia uzlov $[n_0, n_1, \ldots, n_m]$ taká, pre ktorú platí, že každá za sebou idúca dvojica uzlov tvorí hranu v grafe toku riadenia.
- *Jednoduchou cestou* myslíme takú cestu, v ktorej sa žiaden uzol neopakuje. Výnimkou takého uzlu je uzol, ktorý by bol začiatočný a zároveň konečným uzlom danej cesty.
- *Testovacia cesta* je taký druh cesty, ktorý začína v jednom zo vstupných uzlov a končí v koncovom uzle grafu toku riadenia.
- *Testovacia sada* je množina testovacích ciest.
- *Dĺžka cesty* udáva počet hrán, ktoré obsahuje daná cesta. Cesta, ktorú tvorí iba samotný uzol má dĺžku 0.
- *Syntaktická dosiahnuteľnosť uzla* je pojem, ktorý sa využíva v oblasti grafov a rozhoduje o tom, či existuje cesta, resp. v prípade grafov hrana, medzi danými uzlami. Pri syntaktickej dosiahnuteľnosti využívame iba samotnú topológiu grafu, tj. uzly a hrany. Uzol X sa považuje za syntaktický dosiahnuteľný z uzla Y práve vtedy ak existuje hrana z uzla Y, ktorá vedie do uzla X.
- *Sémantická dosiahnuteľnosť uzla* znamená, že uzol X je sémantický dosiahnuteľný z uzla Y práve vtedy, ak existuje nejaká stopa z uzla Y do uzla X. Podrobnejší popis sémantickej dosiahnuteľnosti uzla sa nachádza v kapitole 2.3.

Všetky vyššie uvedený pojmy a definície boli naštudované z knihy [1].

2.2 Kritéria pokrytia

Kritérium pokrytia je pravidlo alebo súbor pravidiel na základe, ktorých sa generujú jednotlivé požiadavky na test. Ak testovacia sada spĺňa tieto všetky požiadavky na test, tak hovoríme, že testovacia sada spĺňa isté kritérium pokrytia. Požiadavkom na test rozumieme istú časť kódu, ktorú musí testovacia sada pokryť alebo splniť. Testovacia sada spĺňa kritérium pokrytia práve vtedy, ak každá požiadavka na test vyplývajúca z kritéria pokrytia je aspoň raz zahrnutá v testovacej sade.

Pokrytie udáva mieru ako veľmi testovacia sada skúma testovaný systém. Úroveň pokrytia sa typicky udáva v percentách vzhľadom na vybrané kritérium pokrytia a testovaciu sadu. Vhodne zvolená testovacia sada by mala dosiahnuť 100% pokrytia.

Softvérový artefakt je subjekt programátora pomocou, ktorých je tvorený výsledný produkt. Jedná sa napríklad o časti kódu ako sú funkcie, moduly, triedy alebo metódy. Následne to môžu byť riadky kódu, ucelené logické celky, práca s dátami, logika behu programu alebo akákoľvek kombinácia týchto artefaktov. Všetky tieto artefakty sa dajú využiť pri generovaní testovacích sád a využitie jednotlivých artefaktov zabezpečuje istý druh pokrytia.

Pokrytie riadkov kódu patrí medzi najznámejšie typy pokrytia. *Pokrytie každého riadku* kódu znamená, že každý riadok zdrojového kódu je pokrytý aspoň raz. Jedná sa o jedno

z najjednoduchších kritérií pokrytí ale ako sa ukázalo časom nezabezpečuje dostatočnú úroveň kontroly testovaného systému. Práve z toho dôvodu sa začína čoraz častejšie používať testovanie založené na pokrývaní grafu toku riadenia 2.1.

Kritéria pokrytia grafov

Pri využívaní jedného z *kritérií pokrytia grafu* sú požiadavky na test vytvárane nad uzlami, hranami a cestami daného grafu. Pri vytváraní požiadaviek na teste sa najčastejšie využíva graf riadenia toku, ktorý je definovaný v kapitole 2.1. Kritérium pokrytia grafu definuje výsledné požiadavky na test. Testovacia cesta môže zahrnúť naraz viacero testovacích požiadaviek. Podstatné je pri výslednom určovaní pokrytia daného kritéria, či bola každá požiadavka na test aspoň raz splnená. Samotné požiadavky sú typicky vo forme potreby prechodu danej hrany alebo potreby navštívenia daného uzla. Pojmy ohľadom kritérií pokrytia spolu s popisom jednotlivých typov kritérií pokrytia boli naštudované z knihy [5]:

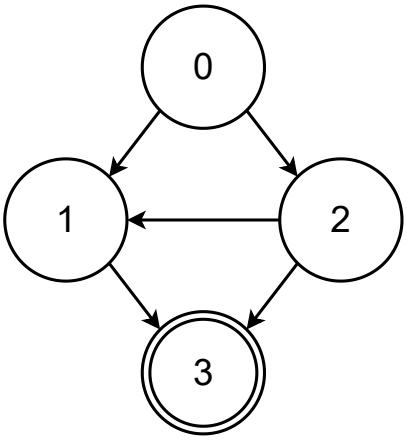
Kritérium pokrytia uzlov (z ang. node coverage) je základným typom pokrytia grafu. Kritérium pokrytia uzlov je splnené ak testovacie požiadavky obsahujú všetky syntaktický dosiahnuteľné uzly grafu riadenia toku.

Kritérium pokrytia hrán (z ang. edge coverage) zahŕňa kritérium pokrytia hrán. Pre dosiahnutie kritéria pokrytia hrán je potrebné aby každá syntakticky dosiahnuteľná hrana grafu toku riadenia bola aspoň raz zahrnutá v testovacích požiadavkách.

Kritérium pokrytia páru hrán (z ang. edge-pair coverage) vyžaduje aby testovacie požiadavky obsahovali všetky syntakticky dosiahnuteľné cesty o maximálnej dĺžke 2. Splnením daného kritéria pokrytia je zabezpečené aj kritérium pokrytia hrán, nakoľko spomínané kritérium pokrytia obsahuje iba syntakticky dosiahnuteľné hrany o dĺžke 0 a 1.

Kritérium pokrytia hlavný ciest (z ang. prime-path coverage) je najsilnejšie kritérium pokrytia vrámci kritérií pokrytia grafov. Požiadavky na test musia obsahovať každú syntakticky dosiahnuteľnú hlavnú cestu. Hlavá cesta je taká jednoduchá cesta, ktorá nie je podcestou žiadnej inej jednoduchej cesty.

Príklad grafu toku riadenia sa nachádza na obrázku 2.1 a testovacie požiadavky pre splnenie jednotlivých kritérií pokrytia sú v tabuľke 2.1.

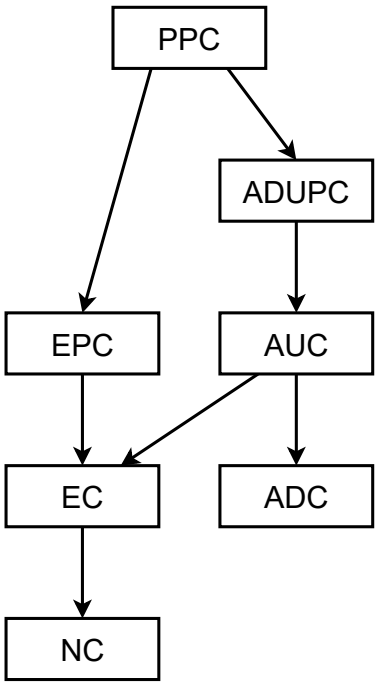


Obr. 2.1: Graf toku riadenia

Kritérium pokrytia grafu	**Testovacie požiadavky**
Kritérium pokrytia uzlov	{[0], [1], [2], [3]}
Kritérium pokrytia hrán	{[0, 1], [0, 2], [1, 3], [2, 1], [2, 3]}
Kritérium pokrytia páru hrán	{[0, 1, 3] [0, 2, 1], [0, 2, 3], [2, 1, 3]}
Kritérium pokrytia uzlov	{[0, 1, 3], [0, 2, 1, 3], [0, 2, 3]}

Tabuľka 2.1: Kritériá pokrytia grafu a ich testovacie požiadavky ku grafu na obrázku 2.1

Medzi všetkými vyššie uvedenými kritériami pokrytia existuje vzťah. Kritérium A zahŕňa kritérium B, práve vtedy ak testovacia sada kritéria A spĺňa všetky požiadavky kritéria B, tým sa určuje aj tzv. sila kritéria. Kritérium A je silnejšie ako kritérium B. Jednotlivé kritéria pokrytia grafu a ich vzťahy medzi sebou sú zobrazené v obrázku 2.2.



Obr. 2.2: Vzťah kritérií pokrytia grafu (sila kritéria stúpa zdola nahor), vysvetlivky ku skratkám v tabuľke 2.2

Skratka	Celý názov kritéria pokrytia
PPC	Kritérium pokrytia hlavných ciest
ADUPC	Kritérium pokrytia všetkých párov definícií a užití
EPC	Kritérium pokrytia páru hrán
AUC	Kritérium pokrytia všetkých užití
EC	Kritérium pokrytia hrán
ADC	Kritérium pokrytia všetkých definícií
NC	Kritérium pokrytia uzlov

Tabuľka 2.2: Vysvetlivky k skratkám kritérií pokrytí k obrázku 2.2

2.3 Sémantická dosiahnuteľnosť cesty

Pri vytváraní testovacích ciest je dôležitá okrem syntaktickej dosiahnuteľnosti cesty aj jej sémantická dosiahnuteľnosť. Zisťovanie sémantickej dosiahnuteľnosti cesty patrí medzi je veľmi náročné a zložité činnosti. Avšak pre generovanie testovacích ciest veľmi podstatná, nakoľko bez vstupného ohodnotenia funkcií je takmer nemožné docieliť presný prechod zdrojovým kódom. Jedným z existujúcich riešení problému sémantickej dosiahnuteľnosti je využívanie tzv. SMT nástrojov (z ang. satisfiability modulo theories). Tieto nástroje nám dokážu na základe matematicko-logický formúl vrátiť správne ohodnotenie, tj. vstupné hodnoty funkcie aby sa docielil jej požadovaný prechod [3].

Jednotlivé základné bloky grafu toku riadenia obsahujú *GIMPLE* inštrukcie (*GIMPLE* je zjednodušený jazyk prekladača *GCC, the GNU Compiler Collection*[1]), ktoré predstavujú trojadresnú reprezentáciu odvodenú od reprezentácie *GENERIC* (jedná sa o reprezentáciu funkcie uloženú v poli príslušného uzla stromu, ktoré sú následne konvertované do jazyka GIMPLE).

Pred samotným ohodnocovaním jednotlivých ciest je potrebné *GIMPLE* inštrukcie vhodne upraviť a vytvoriť z nich formule pre dostupné SMT nástroje. Bližší popis ohľadom ohodnocovania ciest je popísaný v kapitole 3.1.

2.4 Získavanie grafu toku riadenia

Využívanie kritérií pokrytia grafov vyžaduje graf toku riadenia programu. Pre získanie grafu toku riadenia existuje viacero možností. Jednou z možností je využitie prekladaču *Clang* a frameworku LLVM[2] alebo využitie prekladača *GCC, the GNU Compiler Collection* a jeho GIMPLE inštrukcie [13]. Pre nástroj TRIP sme sa rozhodli využiť druhú zo spomínaných možnosti nakoľko vrámci vydávania nových verzií prekladača sa nemenia interné štruktúry potrebné pre získavanie grafu toku riadenia zo zdrojových kódov. Celkové porovnanie dostupné na webe [6].

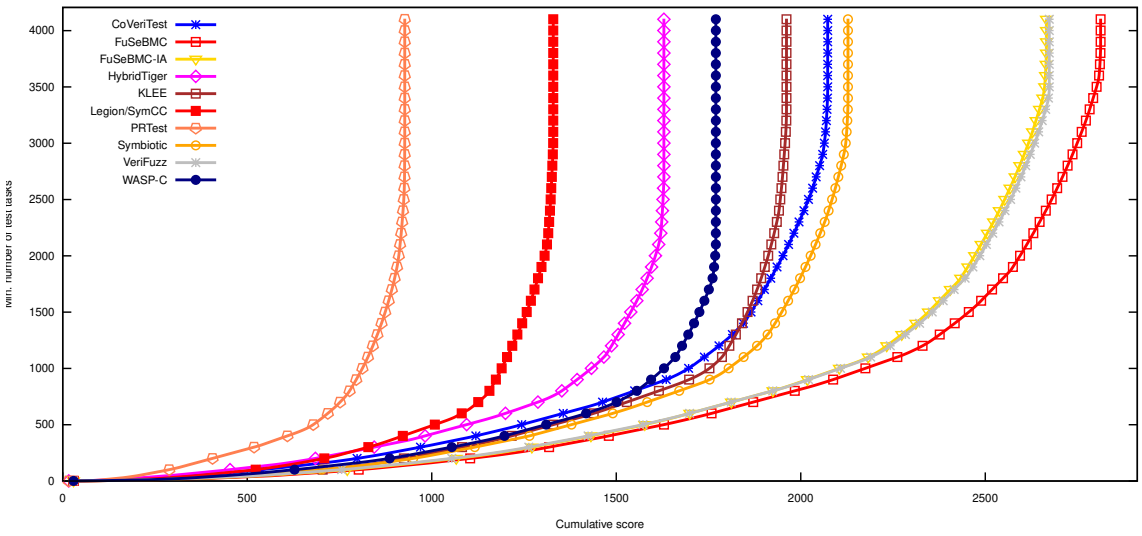
Pre vytváranie testovacích požiadaviek vrámci kritéria pokrytia grafu síce nie sú potrebné inštrukcie, ktoré sa nachádzajú v základných blokoch, ale pre neskoršie zisťovanie sémantickej splniteľnosti sú nevyhnutnou súčasťou. Práve preto je dôležité zabezpečenie získavanie všetkých informácii v rámci grafu toku riadenia naprieč jednotlivými verziami

[1] https://gcc.gnu.org/
[2] https://clang.llvm.org/

prekladača. Podrobnejší popis samotného získavania a využívania grafu toku riadenia sa nachádza v kapitole 3.1.

2.5 Existujúce nástroje

Automatizáciu testovania systémov sa zaoberá nemalé množstvo ľudí, organizácii, vedeckých skupín ale aj firiem. V dnešnej dobe existuje množstvo nástrojov, ktoré sa snažia o čo najpresnejšie ale zároveň čo najefektívnejšie testovanie systémov. Medzi aktuálne najlepšie nástroje patrí *FuSeBMC*, ktorý je momentálne najefektívnejším dostupným nástrojom. Následne sú ešte dostupné nástroje ako *VeriFuzz*, *KLEE* alebo *CMBC* [2]. Všetky tieto nástroje nie sú typickými nástrojmi pre vyhľadávanie testovacích ciest ale skôr sa zameriavajú na formálnu verifikáciu.



Obr. 2.3: Porovnanie existujúcich nástrojov, obrázok prevzatý z [2]

FuSeBMC

Verifikačný nástroj *FuSeBMC* (z ang. Functional Sequential Equivalence Based Model Checker) je komplexný nástroj, ktorý spracováva zdrojové kódy programovacieho jazyka C [12]. Celý priebeh začína analýzou zdrojových kódov. Následne sú medzi jednotlivé časti kódu vkladané návesti (z ang. labels). Zo zdrojových kódov spolu s vloženými návesťami vznikne interná reprezentácia systému uložená vo formáte *GraphML*[3]. Pomocou vytvoreného grafu sa začnú vytvárať jednotlivé testovacie prípady. Ako posledná časť samotného nástroja je selektívny fuzzer[4], ktorý sa učí z doteraz vytvorených testovacích prípadov.

KLEE

Nástroj *KLEE* je dynamický symbolický nástroj využívajúci štruktúru *LLVM*[5], ktorú využíva napríklad aj prekladač *Clang*. Pomocou nástroja sme schopní generovať a prevádzať

[3] GraphML je formát na kreslenie grafov založený na formáte XML
[4] technika poskytovania neplatných, neočakávané alebo náhodne vygenerovaných údajov
[5] https://llvm.org/

testy. *KLEE* dokáže pracovať s programami napísanými v programovacích jazykoch C alebo C++. Medzi hlavné výhody nástroja *KLEE* patrí, že dokáže nahradiť konkrétne hodnoty za symbolické. Následne využíva techniku nazývanú „concolic testing“, ktorá kombinuje symbolické a konkrétne vykonávanie. Testovanie je možné vykonávať opakovane bez nutnosti spúšťania nástroja *KLEE*, nakoľko *KLEE* vygeneruje testovacie výstupy, ktoré sa dajú znova využiť. Všetky informácie boli získane z oficiálnej stránky nástroja [15].

Kapitola 3

Návrh nástroja pre vyhľadávanie testovacích ciest

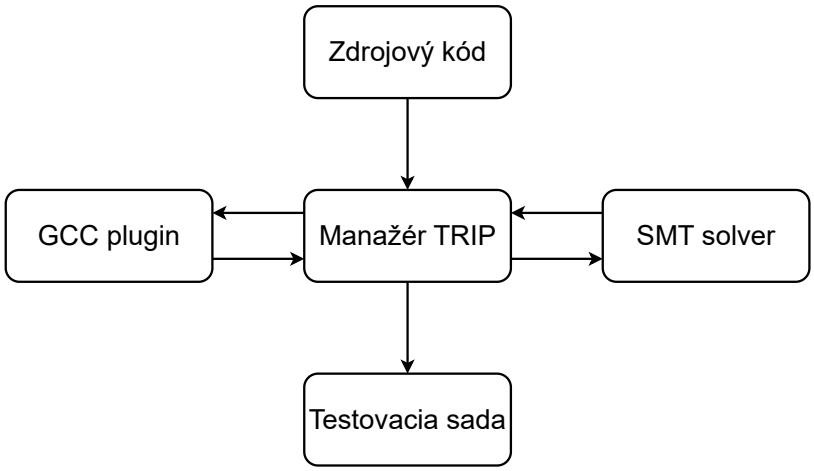
Vytvorenie čo najlepšieho návrhu nástroja bolo kľúčovou súčasťou projektu. Nakoľko na celom vývoji nástroja TRIP sme sa podieľali traja študenti, konkrétne František Lukeš, Ján Rybka a Michal Stupar. Vývoj nástroja TRIP sme si rozdelili do trochu častí, ktoré nie sú priamo na sebe závislé, avšak aktívne medzi sebou komunikujú. Práve z toho dôvodu museli byť využité stubs[1], ktoré zabezpečujú možnosť osobitného testovania jednotlivých troch častí.

Popis jednotlivých častí a ich vzájomná komunikácie sa nachádza v kapitole 3.1. Požiadavky kladené na nástroj TRIP sa nachádzajú v kapitole 3.2. Pre vzájomnú komunikáciu medzi časťami nástroja TRIP sme sa rozhodli využiť formát JSON (JavaScript Object Notation)[2], ktorý umožňuje jednoduchú výmenu informácii medzi aplikáciami. Ďalšie výhody aj spôsob využívania JSON formátu sa nachádza v kapitole 3.3. Pre prácu s JSON formátom bolo navrhnuté využitie JSON knižnice, ktorej podrobnejší popis sa nachádza v kapitole 3.4. Podrobný návrh architektúry nástroja TRIP je v kapitole 3.5. Návrh jednotlivých generátorov testovacích požiadaviek podľa zvoleného kritéria pokrytia je opísaný v kapitole 3.6. Kapitola 3.7 opisuje návrh využitých vyhľadávacích algoritmov a prístup k vytváraniu ďalších algoritmov pre vyhľadávanie testovacích ciest.

3.1 Jednotlivé časti nástroja

Vývoj nástroja TRIP pre vyhľadávanie testovacích ciest sme rozdelili do trochu separátnych častí. Celý nástroj TRIP sme rozdelili do tzv. manažéra, ktorý je samotným jadrom systému a zároveň akýmsi frameworkom pre vyhľadávanie testovacích ciest. Ďalšou častou nástroja TRIP je tzv. *GCC plugin*, ktorý je zodpovedný za vytvorenie grafu toku riadenia. Na vývoji *GCC plugin* sa podieľa František Lukeš. Poslednou časťou nástroja TRIP je tzv. *SMT solver*, ktorého hlavnou úlohou je zisťovanie sémantickej dosiahnuteľnosti ciest. Vývoju *SMT solvera* sa venuje Ján Rybka. Grafické znázornenie rozdelenia nástroja TRIP je na obrázku 3.1.

[1] funkcia, ktorá simuluje originálnu funkciu a bez ohľadu na parametre vykonáva tú istú úlohu
[2] https://www.json.org/



Obr. 3.1: Rozdelenie nástroja TRIP na jednotlivé časti

Pre implementáciu nástroja TRIP sme zvolili programovací jazyk C++, konkrétne jednotlivé časti podporujú stabilnú verziu C++11. Hlavným dôvodom pre využitie programovacieho jazyka C++ bola integrácia modulu *GCC plugin* do prekladača *GCC, the GNU Compiler Collection*, ktorého potenciálne možnosti boli programovací jazyk C, prípadne C++. Plánom do budúcna je vytvorenie *Docker*[3] kontajnera pre ešte jednoduchšie spustenie nástroja.

Komunikácia medzi časťami nástroja

Komunikácia medzi jednotlivými časťami nástroja TRIP mohla prebiehať pomocou viacerých variant. Napríklad vo vopred dohodnutom formáte výmeny dát, ktorý by bol z hľadiska rýchlosti určite najlepšou voľbou ale zároveň by sa stratila akákoľvek možnosť výmeny jednotlivých časti nástroja TRIP. Ako jedna z ďalších možností sa ponúka využitie už definovaného formátu, ktorý by zabezpečil jednoduchú možnosť výmeny časti nástroja TRIP za inú časť. Vznikla by tak jediná potreba úpravy dát do vybraného formátu. Z týchto dôvodov sme sa rozhodli pre využitie JSON formátu ako jedného z najpopulárnejších formátov pre výmenu dát. JSON formát vyniká jednoduchosťou, intuitívnosťou, nakoľko sa ľahko číta a upravuje pri prípadných potrebách úpravy grafu toku riadenia. Okrem nespočetného množstva výhod je taktiež dostupná JSON knižnica 3.4 pre C++.

GCC plugin

Jednou z častí nástroja TRIP je *GCC plugin*. Zo zdrojových kódov zadaných užívateľom sa pomocou *GCC pluginu* získa graf toku riadenia, ktorý bude následne predaný vo formáte JSON naspäť manažérovi nástroja TRIP. Získavanie grafu toku riadenia prebieha počas prekladu zdrojového kódu pomocou vhodného pozastavenia prekladu a vloženiu vlastnej časti programu, ktorá je zodpovedná za zozbieranie všetkých potrebných údajov pre graf toku riadenia. Medzi tieto potrebné dáta patria: názov funkcie, parametre prekladanej fun-

[3] https://www.docker.com/

kcie, deklarácie, ktoré si vytvára prekladač na začiatku prekladu, základné bloky, vstupný blok a výstupný blok. Každý základný blok obsahuje viacero inštrukcií a potenciálny skok na ďalší základný blok. Tieto inštrukcie môžu byť typu porovnávanie hodnôt, napríklad *GIMPLE* inštrukcia *gimple_cond* s následným parametrom typu porovnávania napríklad *le_expr*, ktorý reprezentuje znak $\leq$ (z ang. less than or equal to). Tieto inštrukcie obsahujú aj skoky na ďalšie základne bloky na základne vyhodnotenia výrazu. Všetky tieto výrazy neskôr spracováva časť *SMT solver*. Príklad získaného JSON formátu, ktorý spĺňa všetky náležitosti pre spracovanie nástrojom TRIP je v prílohe C.

Pre získanie grafu toku riadenia z prekladača *GCC, the GNU Compiler Collection* sa dá využiť nasledujúci príkaz, ktorý okrem samotného grafu poskytne radu ďalších užitočných grafov:

```
gcc [názov_zdrojového_súboru] -fdump-tree-all-graph -c -g
```

Väčšina údajov potrebný pre nástroj TRIP sa nachádza v súbore:

```
[názov_zdrojového_súboru].015t.cfg
```

SMT solver

Pre získanie ohodnotenia jednotlivých ciest a zistenie ich sémantickej dosiahnuteľnosti je nevyhnutná časť *SMT solver*, ktorá je súčasťou nástroja TRIP. *SMT solver* je zodpovedný za prevedenie jednotlivých *GIMPLE* inštrukcií do matematicko-logických formúl, ktoré sú následne predávané dostupným riešiteľom daných formúl. Medzi aktuálne najpoužívanejšie nástroje na riešenie splniteľnosti týchto formúl prvého radu patrí *Z3 Theorem prover*[4], *SMT-RAT*[5] alebo *MathSAT 5*[6]. Všetky tieto riešitele logických formúl prvého radu nám okrem boolovskej hodnoty pravda alebo nepravda dokážu v prípadne nájdenia riešenia vrátiť aj ohodnotenie za akých podmienok daná formula platí. Následne toto ohodnotenie sa dá využiť ako vstupná hodnota na prevedenie danej testovacej cesty alebo ako podstatná informácia pri vyhľadaní a vytváraní ďalších potenciálnych testovacích ciest. Tieto údaje sú veľmi podstatné pri využívaní zložitých vyhľadávacích algoritmov, ktoré dokážu efektívne tieto ohodnotenia ciest využiť.

3.2 Požiadavky kladené na nástroj

Nástroj TRIP nepodporuje generovanie testovacích sád pre *rekurzívne funkcie*. *Rekurzívna funkcia* je taká funkcia, ktorá počas behu volá samú seba. Takáto funkcia by nebola problematická pre zostrojenie grafu toku riadenia, nakoľko samotné volanie tej istej funkcie by sa zmenilo na inštrukciu *gimple_call* (v prípade používania prekladača GCC, the GNU Compiler Collection). Takisto by *rekurzívna funkcia* nebola problémom pre *manažéra* nástroja TRIP, nakoľko ten spracováva základne bloky grafu a neprihliada na inštrukcie pri tvorení cest. Problém by nastal až pri zisťovaní sémantickej splniteľnosti danej časti. Takisto nástroj podporuje iba *konečné funkcie*. Pod pojmom *konečné funkcie* sú myslené také funkcie, ktoré majú konečný počet možný vstupov a výstupov. Tzv. nekonečné funkcie by boli tiež problematické vrámci zisťovania sémantickej dosiahnuteľnosti daných uzlov, nakoľko by nebolo možné určiť všetky premenné, ktoré sa v danej funkcii nachádzajú. Pre správne fungovanie

[4] https://github.com/Z3Prover/z3
[5] https://ths-rwth.github.io/smtrat/
[6] https://mathsat.fbk.eu/

je vhodné používanie prekladača, ktorý podporuje revíziu programovacieho jazyka C++11, ktorý je označený pod identifikátorom `ISO/IEC 9899:2011`.

Programovací jazyk C nie je nutnosťou. Nástroj TRIP sa dá využiť aj ako vyhľadávač testovacích ciest nad grafom toku riadenia bez využitia *GCC pluginu* a *SMT solvera*. Tieto moduly je možné jednoducho vynechať z celého procesu a využiť predpripravené náhrady modulov, ktoré sú podrobnejšie rozpísané v kapitole 4.7. Pri vynechaní modelu pre získavanie grafu toku riadenia je potrebné tento graf vytvoriť iným spôsobom pre zabezpečenie funkčnosti nástroja. Konkrétny popis grafu toku riadenia sa nachádza v kapitole 3.3. Takisto sa v danej kapitole nachádza jednoduchý popis JSON formátu, ktorý slúži na výmenu medzi *manažérom* nástroja TRIP a samotným *SMT solverom*. Dodržanie týchto formátov hraje kľúčovú požiadavku na spustenie nástroja TRIP. Okrem týchto dvoch JSON formátov sa využíva aj posledný tretí JSON formát, ktorý slúži na import a export ciest.

Nevyhnutnou súčasťou prekladu nástroja je JSON knižnica s názvom *JSON for Modern C++* od *Niels Lohmanna* [11]. Pri navrhovaní a implementácii bola využitá posledná stabilná verzia 3.11.2, ktorá bola vydaná 12.8.2022. Využívanie JSON formátu sa nachádza hneď na začiatku samotného nástroja a je využívaný na väčšinu komunikácie medzi modulmi nástroja. Práve z toho dôvodu nie je možné danú knižnicu vynechať.

3.3 Popis používaných JSON formátov

JSON (JavaScript Object Notation) patrí medzi najrozšírenejšie formáty pre výmenu dát medzi aplikáciami. Nástroj TRIP využíva JSON formát pre komunikáciu medzi *manažérom* nástroja TRIP a *GCC pluginom*, *manažérom* nástroja TRIP a *SMT solverom* a pri ukladaní a načítavaní testovacej sady za účelom dosiahnutia vyššieho pokrytia.

JSON formát využíva pre ukladanie dát systému „kľúč : hodnota“ (z ang. key : value). Kľúče sa ukladajú ako slovo medzi dvojité úvodzovky na ľavú stranu dvojbodky (napr. "mesto": hodnota). Hodnota sa píše na pravú stranu dvojbodky a môže byť jedným z týchto šiestich typov: číslo, slovo, objekt, pole, booleovská hodnota (`true` alebo `false`) alebo `null`. Celý príklad správneho JSON formátu grafu toku riadenia sa nachádza v prílohe C. Príklad JSON formátu pre import a export sa nachádza v prílohe D.

Povinnosť jednotlivých kľúčov a hodnôt v nižšie uvedených tabuľkách je myslená vrámci správneho fungovania nástroja, avšak v prípade ak je povinnosť označená ako „kritická“, tak je daný kľúč a hodnota nevyhnutná pre spustenie vyhľadávania. Absencia kľúčov a hodnôt označených ako „povinná“ nepovedie k ukončeniu nástroja akurát ostanú uložené základné hodnoty (z ang. default value), prípadne predošlé hodnoty. Poslednou kategóriou je „voliteľná“. Hodnoty týchto kľúčov majú častokrát informatívny, resp. ladiaci význam.

Pri navrhovaní JSON formátu pre komunikáciu medzi *GCC pluginom* a nástrojom TRIP sme využívali oficiálnu stránku prekladača *GCC, the GNU Compiler Collection* pre vytvorenie predstavy vnútornej štruktúry grafu toku riadenia [14].

Graf toku riadenia v JSON formáte:

Graf toku riadenia:

Názov kľúča	Povinnosť	Očakávaná hodnota	Popis
basicBlocks	Kritický	Kľúče k objektom s názvom *základných blokov* 3.2	pre správne fungovanie nástroja je nutné aby kľúče reprezentovali jedinečné označenie (*ID*) základného bloku začínajúce hodnotou „0“, ktoré sa inkrementuje o 1
entryBlock	Kritický	Číslo	Označuje vstupný *základný blok* grafu toku riadenia
exitBlock	Kritický	Číslo	Označuje výstupný *základný blok* grafu toku riadenia
declarations	Povinný	Pole	Obsahuje objekty typu *parameter*
functionName	Voliteľný	Slovo	Názov prehľadávanej funkcie
function-Parameters	Povinný	Pole	Obsahuje objekty typu *parameter*

Tabuľka 3.1: Graf toku riadenia v JSON formáte

Základný blok:

Názov kľúča	Povinnosť	Očakávaná hodnota	Popis
statements	Povinný	Pole	Obsahuje objekty *špecifikácia*

Tabuľka 3.2: Základný blok v JSON formáte

Špecifikácia:

Názov kľúča	Povinnosť	Očakávaná hodnota	Popis
jumps	Povinný	Objekt s názvom *skoky* 3.4	objekt je povinný v prípade ak *špecifikácia* obsahuje rozhodovaciu inštrukciu ako napr. *gimple_cond*, inak je nežiadaný
localization	Voliteľný	Objekt s názvom *lokalizácia* 3.5	lokalizácia v zdrojovom kóde
statement	Povinný	Objekt s názvom *konkrétna špecifikácia* 3.6	samotná inštrukcia a jej parametre

Tabuľka 3.3: Špecifikácia v JSON formáte

Skok:

Názov kľúča	Povinnosť	Očakávaná hodnota	Popis
false	Povinný	Číslo	Jedinečné identifikačné číslo (*ID*) základného bloku, ktorý nasleduje v prípade neplatnej podmienky
true	Povinný	Číslo	Jedinečné identifikačné číslo (*ID*) základného bloku, ktorý nasleduje v prípade platnej podmienky

Tabuľka 3.4: Skok v JSON formáte

Lokalizácia:

Názov kľúča	Povinnosť	Očakávaná hodnota	Popis
file	Voliteľný	Slovo	Názov zdrojového súboru
line	Voliteľný	Slovo	Riadok *konkrétnej špecifikácie* v zdrojovom kóde

Tabuľka 3.5: Lokalizácia v JSON formáte

Konkrétna špecifikácia:

Názov kľúča	Povinnosť	Očakávaná hodnota	Popis
comparison	Povinný	Slovo	Konkrétny typ inštrukcie je povinný, v prípade ak sa jedná o porovnávaciu *inštrukciu* ako napr. *gimple_cond*, inak je daná hodnota nežiadúca
instruction	Kritický	Slovo	Názov inštrukcie
leftSide	Povinný	Slovo	Hodnota na ľavej strane inštrukcie, jedná sa o povinný atribút v prípade ak konkrétna inštrukcia využíva hodnotu na ľavej strane, inak je hodnota nežiadúca
rightSide	Povinný	Slovo	Hodnota na pravej strane inštrukcie, jedná sa o povinný atribút v prípade ak konkrétna inštrukcia využíva hodnotu na pravej strane, inak je hodnota nežiadúca

Tabuľka 3.6: Konkrétna špecifikácia v JSON formáte

Parameter:

Názov kľúča	Povinnosť	Očakávaná hodnota	Popis
name	Povinný	Slovo	Názov parametra
type	Povinný	Slovo	Dátový typ parametra

Tabuľka 3.7: Parameter v JSON formáte

JSON formát pre predávanie výsledku ohodnotenia cesty od SMT solvera:

Predávanie ohodnotenia cesty:

Názov kľúča	Povinnosť	Očakávaná hodnota	Popis
context	Voliteľný	Slovo	Voľná premenná, ktorá môže slúžiť na optimalizáciu vyhľadávacieho algoritmu, resp. na predávanie ľubovoľných dát medzi vyhľadávacím algoritmom a SMT solverom
reachability	Povinný	Číslo	Ohodnotenie cesty na základe vopred definovaných hodnôt z `graphs.hpp`, ak by JSON neobsahoval ohodnotenie cesty ostane tam pôvodné ohodnotenie cesty, ohodnotenie cesty pri vzniku je 0

Tabuľka 3.8: Predávanie ohodnotenia cesty v JSON formáte

Ukladanie testovacích sád:

Testovacia sada:

Názov kľúča	Povinnosť	Očakávaná hodnota	Popis
paths	Povinný	Pole	Obsahuje objekty typu cesta, v prípade ak sa ukladajú iba testovacie cesty môže byť pole prázdne, prípadne kľúč vynechaný
testPaths	Povinný	Pole	Obsahuje objekty typu cesta, v prípade ak sa ukladajú iba všeobecne nájdené cesty môže byť pole prázdne, prípadne kľúč vynechaný

Tabuľka 3.9: Testovacia sada v JSON formáte

Cesta:

Názov kľúča	Povinnosť	Očakávaná hodnota	Popis
basicBlocks	Povinný	Pole	Obsahujúce čísla, ktoré reprezentujú jedinečné identifikátory (*ID*) základných blokov, ktoré sa nachádzajú v grafe toku riadenie, v prípade nevalidného čísla sa cesta preskakuje
context	Voliteľný	Slovo	Voľná premenná na ukladanie ľubovoľných dát medzi jednotlivými spúšťaniami nástroja
reachability	Povinný	Číslo	Ohodnotenie cesty na základe vopred definovaných hodnôt z `graphs.hpp`, ak by JSON neobsahoval ohodnotenie cesty ostane základná hodnota 0

Tabuľka 3.10: Cesta v JSON formáte

3.4 JSON knižnica

Pri návrhu nástroja TRIP z dôvodu častého využívania JSON formátu na komunikáciu modulov sme rozhodli pre využitie už jednej z existujúcich JSON knižníc. Nakoľko žiadna štandardná knižnica neobsahuje prácu s JSON formátom, museli sme využiť jednú z verejne dostupných knižníc. Pri výbere hrala jednu z hlavných rolí kvalita prevedenia knižnice, jej použiteľnosť a programátorská dokumentácia aby sme vedeli knižnicu pohodlne implementovať do nástroja TRIP. Ak by bola najpodstatnejšia rýchlosť parsovania JSON formátu, tak by sme sa s vysokou pravdepodobnosťou nerozhodli pre tento formát a navrhli by sme si internú štruktúru na predávanie dát medzi modulmi. Na výber sa ponúkali knižnice: *JSON for Modern C++*[7], *RapidJSON*[8] alebo *C++ library for interacting with JSON*[9]. Nakoniec sme sa rozhodli pre zahrnutie knižnice *JSON for Modern C++* od *Niels Lohmanna* [11], nakoľko spĺňala všetky požiadavky a je aktuálne jednou z najpopulárnejších.

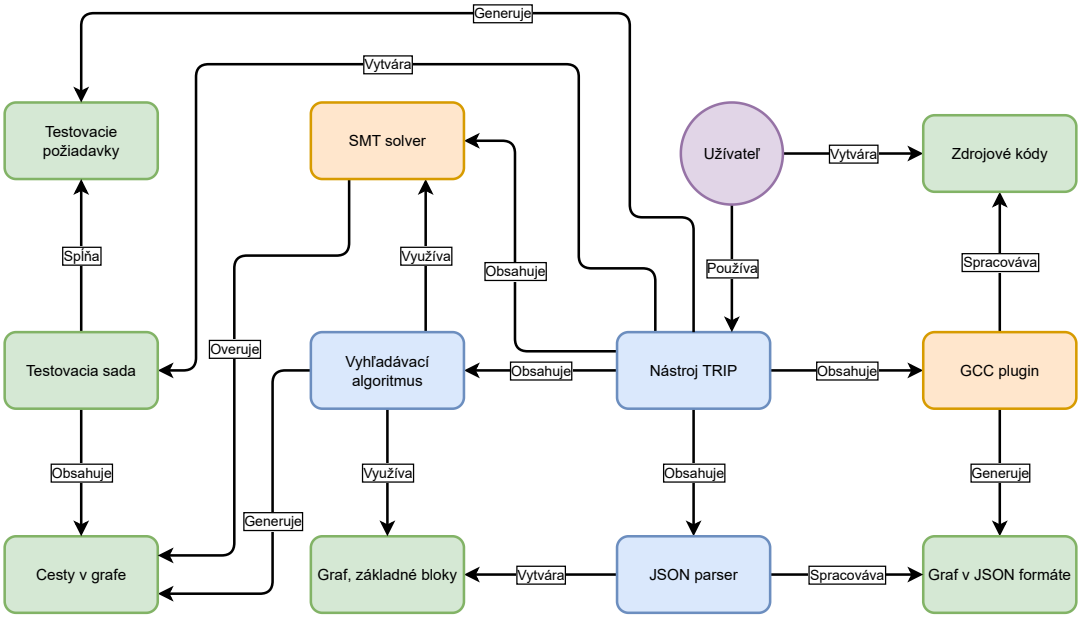
3.5 Návrh architektúry

Celkový návrh architektúry nástroja TRIP sa nachádza na obrázku 3.2. Nástroj TRIP sa skladá z dvoch externých častí *SMT solver* a *GCC plugin*, ktoré sú podfarbené oranžovou farbou. Tieto modely sú nahradené náhradnými modulmi, ktorých implementácia sa nachádza v kapitole 4.7. Následne nástroj TRIP okrem samotného nástroja obsahuje dve interné časti *Vyhľadávací algoritmus* a *JSON parser*, všetky tieto časti sú označené modrou farbou. Zelenou farbou sú označené dátové celky, ktoré sa pri fungovaní nástroja TRIP využívajú.

Užívateľ vytvára a poskytuje zdrojové kódy, následne využíva samotný nástroj TRIP. Pred spustením nástroja TRIP si užívateľ pomocou rozhrania príkazovej riadky 4.1 vyberie

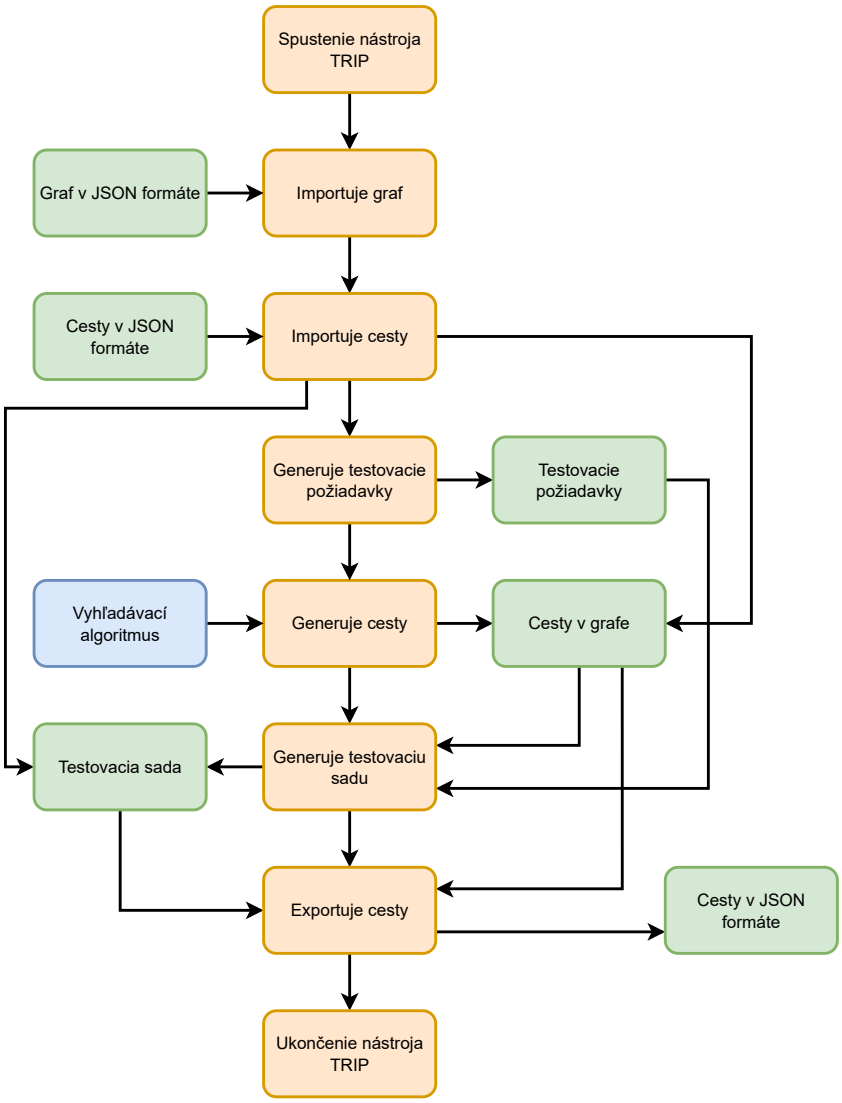
[7] https://github.com/nlohmann/json
[8] https://rapidjson.org/
[9] https://github.com/open-source-parsers/jsoncpp

kritérium pokrytia grafu. Jedným z prvých krokov nástroja TRIP je spracovanie zdrojových kódov pomocou *GCC plugina*, ten z nich vygeneruje reprezentáciu grafu toku riadenia v JSON formáte. Následne *JSON parser*, ktorý je interným modulom nástroj TRIP tento graf v JSON formáte parsuje a vytvára internú reprezentáciu grafu toku riadenia, základných blokov, deklarácií a vstupných parametrov. Podrobnejší popis jednotlivých tried sa nachádza v kapitole 4.2. Po úspešnom načítaní dát z JSON formátu do interných štruktúr prichádza na rad generátor kritérií pokrytia, ktorý je internou súčasťou nástroja TRIP. Následne je spustený vyhľadávací algoritmus, ktorý s pomocou externej časti *SMT solver* vytvára cesty v grafe, ktoré sa neskôr môžu stať súčasťou testovacej sady. Ukončením vyhľadávania prichádza na rad finálne generovanie testovacej sady, ktoré prebieha pomocou samotného manažéra nástroja TRIP. Testovacia sada sa skladá z ciest v grafe, ktoré boli nájdené pomocou vyhľadávacieho algoritmu, zároveň spĺňajú vytvorené testovacie požiadavky a definíciu testovacej cesty.



Obr. 3.2: Návrh architektúry

Priebeh spustenia nástroja TRIP môžeme rozdeliť do šiestich častí. Prvou časťou je import grafu z JSON formátu. Následne prebieha import ciest, ak bola daná možnosť zvolená pri spúšťaní. Treťou časťou je generovanie testovacích požiadavok. Pri ďalšej časti sa využíva interná časť nástroja TRIP, ktorou je *vyhľadávací algoritmus* a generujú sa cesty v grafe. Po ukončení vyhľadávania nastáva generovanie testovacej sady. Generovanie testovacej sady pozostáva zo spĺňania testovacích požiadaviek využitím ciest v grafe, ktoré spĺňajú definíciu testovacej cesty. Poslednou časťou je export ciest v prípade výberu exportu ciest pri spúšťaní. Celý priebeh je vizuálne zobrazený na obrázku 3.3.



Obr. 3.3: Graf priebehu spustenia nástroja TRIP

3.6 Generovanie testovacích požiadaviek podľa kritéria pokrytia

Návrh generátora testovacích požiadaviek na základe vybraného kritéria pokrytia bol rozdelení do štyroch oddelených generátorov podľa kritéria pokrytia. Všetky tieto generátory fungujú nad množinou základných blokov grafu toku riadenia. Využívanie jedného z generátorov nebráni k využitiu ďalšieho generátora, avšak výsledný efekt bude rovnaký ako použitie generátora testovacích požiadaviek pre splnenie najsilnejšieho z vyberaných generátorov. Zobrazenie sily jednotlivých kritérií pokrytia je znázornené v grafe 2.2. Podrobnejší popis implementácie jednotlivých kritérií pokrytí a ich generátorov je v kapitole 4.3.

Generátor pre kritérium pokrytia uzlov

Pre vytvorenie všetkých testovacích požiadaviek pre kritérium pokrytia všetkých uzlov je potrebné vybrať z grafu každý jeden základný blok bez ohľadu na jeho nasledovníkov alebo predchodcov.

Generátor pre kritérium pokrytia hrán

Splnenie kritéria pokrytia všetkých hrán vyžaduje, aby testovacie požiadavky obsahovali každú jednu hranu medzi dvoma základnými blokmi.

Generátor pre kritérium pokrytia páru hrán

Testovacie požiadavky musia obsahovať všetky cesty o maximálne dĺžke 2. Pre vygenerovanie všetkých párov hrán je potrebné z každého základného bloku vygenerovať všetky dostupné cesty o maximálnej dĺžke 2.

Generátor pre kritérium pokrytia hlavných ciest

Kritérium pokrytia hlavných ciest si vyžaduje, aby testovacie požiadavky obsahovali všetky hlavné cesty. Pod pojmom hlavná cesta rozumieme takú cestu, ktorá nie je podcestou žiadnej inej cesty. Pri generovaní všetkých hlavných ciest je použitá jednoduchá logika vygenerovania všetkých dostupných ciest v grafe a následne sa ukladajú do testovacích požiadaviek iba tie cesty, ktoré nie sú podcestou žiadnej inej cesty. Pre zvýšenie efektivity sa postupuje od najdlhších ciest k tým najkratším. Nástroj TRIP je predovšetkým frameworkom pre výučbu vyhľadávania testovacích ciest, práve z toho dôvodu nie je potrebné dbať na absenciu vysokej efektivity.

3.7 Vyhľadávacie algoritmy

Vyhľadávací algoritmus je najdôležitejšou časťou nástroja TRIP. Bez vyhľadávacieho algoritmu by nebolo možné vytvárať žiadne cesty v grafe. Existuje viacero prístupov a metód pre riešenie úloh prehľadávania stavového priestoru, medzi ktoré patrí aj graf toku riadenia. Tieto prístupy a metódy delím do troch základných častí: *slepé metódy* (z ang. blind search methods), *informované metódy* (z ang. informed search methods) a metódy *lokálneho prehľadávania* (z ang. local search methods) [4]. Vyberatie vhodného algoritmu a neskôr jeho implementácia patrí k obecne najťažším úlohám pri prehľadávaní stavového priestoru.

Pre demonštráciu nástroja TRIP sme sa rozhodli vybrať dve najznámejšie slepé metódy. *Prehľadávanie do šírky* (z ang. breadth-first search) a *prehľadávanie do hĺbky* (z ang. depth-first search). Pri oboch vyhľadávacích algoritmoch boli využité ich základné princípy, ale museli byť mierne upravené z dôvodu zamedzenia potenciálneho zacyklenia. Vrámci pridávania ďalšieho základného bloku do cesty bola pridaná kontrola, či sa už daný blok nenachádza v ceste. V prípade ak sa daný uzol nenachádzal v ceste, tak bol uzol expandovaný v opačnom prípade bolo expandovanie uzla ukončené. Podrobnejší popis implementácie vyhľadávacích algoritmov sa nachádza v kapitole 4.4.

Kapitola 4

Implementačné detaily

Táto kapitola opisuje implementačné detaily navrhnutého nástroja TRIP. Jednotlivé kapitoly obsahujú konkrétnu implementáciu jednotlivých časti nástroja. V prvej časti 4.1 je opis rozhrania príkazovej riadky, jednotlivé podporované prepínače. Druhá kapitola 4.2 obsahuje popis všetkých implementovaných tried a metód, ktoré tvoria samotné jadro nástroja TRIP. Všetky podporované kritéria pokrytia a ich generovanie je opísané v kapitole 4.3. Nasledujúca kapitola 4.4 obsahuje popis implementovaných vyhľadávacích algoritmov. Opakované spúšťanie nástroja TRIP sa nezaobíde bez ukladania a načítavania ciest. Popis ukladaného súboru sa nachádza v kapitole 4.5. Beh programu by sa nezaobišiel bez regulácie na základe vypršania času (z ang. timeout), ktorý opisuje kapitola 4.6. Nakoľko nástroj TRIP spolupracuje s dvomi externými časťami, ktoré nie sú aktuálne ešte dostupné je potrebné ich nahradiť pomocou tzv. stubs[1]. Popis náhradných modulov sa nachádza v kapitole 4.7. Posledná kapitola 4.8 obsahuje návratové kódy nástroja TRIP s ich popisom.

4.1 Rozhranie príkazovej riadky

Nástroj TRIP na vstupe očakáva viacero argumentov, ktoré určujú jeho správanie. Niektoré kombinácie prepínačov nie sú prípustné nakoľko ich význam je v rozpore. Neprípustné kombinácie k danému prepínaču sú v stĺpci „neprípustné kombinácie“. Jednotlivé prepínače sa nachádzajú v tabuľke 4.1.

Miesto prvého zadávaného argumentu patrí názvu zdrojového kódu pre vyhľadávanie testovacích ciest. Nástroj TRIP nie je v súčastnosti schopný bez zadania zdrojového kódu fungovať, nakoľko zatiaľ nepodporuje ukladanie grafu toku riadenia, ktorý je pre vyhľadávanie testovacích ciest nevyhnutný.

Každý prepínač je možné použiť iba raz, inak sa program ukončí s chybovou hláškou „Invalid arguments.“ v preklade neplatné argumenty, návratovou hodnotou 1 a vypíše sa nápoveda na štandardný výstup. Využívanie kombinácie viacerých generátorov testovacích požiadaviek nevedie ku chybe, ale ani k zvýšeniu sily pokrytia, podrobnejšie vysvetlenie v kapitole 3.6.

Používanie prepínačov `-ip` alebo `-itp` vyžaduje zadanie vstupného súboru ciest pomocou prepínača `-in`. Obdobne to platí aj pre prepínače so súborom pre ukladanie ciest. Pri použití jedného z prepínačov `-op` alebo `-otp` je povinný prepínač `-out` s názvom súboru. V prípade ak budú použité prepínače bez zadania súborov je program ukončený chybovou hláškou „Invalid arguments.“ s návratovou hodnotou 1.

[1] funkcia, ktorá simuluje originálnu funkciu a bez ohľadu na parametre vykonáva tú istú úlohu

Výpis pomocou prepínačov `-pbs` alebo `-pas` je štruktúrovaný výpis dát, ktoré môže slúžiť ako ladiaci výpis pri implementácii nových algoritmov nástroja TRIP. Ukážka takého výpisu je v prílohe E.

Prepínač	**Popis**	**Neprípustné kombinácie**
-h	Na štandardný výstup sa vypíše nápoveda	
-nc	Vytvoria sa testovacie požiadavky pre kritérium pokrytia uzlov	
-ec	Vytvoria sa testovacie požiadavky pre kritérium pokrytia hrán	
-epc	Vytvoria sa testovacie požiadavky pre kritérium pokrytia páru hrán	
-ppc	Vytvoria sa testovacie požiadavky pre kritérium pokrytia hlavných ciest	
-pbs	Pred vyhľadávaním ciest sa vypíšu na štandardný výstup všetky údaje grafu	
-pas	Po vyhľadávaním ciest sa vypíšu na štandardný výstup všetky údaje grafu	
-ip	Zo vstupného súboru zadaného spolu s prepínačom `-in` sa načítajú iba nájdené cesty	-itp
-itp	Zo vstupného súboru zadaného spolu s prepínačom `-in` sa načítajú iba nájdené testovacie cesty	-ip
-op	Do výstupného súboru zadaného spolu s prepínačom `-out` sa uložia iba nájdené cesty	-otp
-otp	Do výstupného súboru zadaného spolu s prepínačom `-out` sa uložia iba nájdené testovacie cesty	-op
-in [súbor]	Súbore pre načítanie ciest	
-out [súbor]	Súbore pre uloženie ciest	
-per [číslo]	Celé nezáporné číslo udávajúce počet minimálny percent pre ukončenie generovania testovacích ciest	

Tabuľka 4.1: Prepínače s popisom

4.2 Implementované triedy a metódy

Hlavnou náplňou tejto bakalárskej práce bolo vytvoriť nástroj, resp. framework, ktorý by umožňoval praktickú výučbu v oblasti vyhľadávania testovacích ciest. Práve navrhnutie a implementácia tried a metód tvoria jadro celého frameworku. Jednotlivé požiadavky na framework boli rozdelené na drobnejšie čiastky a implementované ako triedy. Konkrétne sa jedná o triedy: `Graph` (v preklade graf), `BasicBlock` (v preklade základný blok), `Statement` (v preklade špecifikácia) a `Path` (v preklade cesta). Tieto triedy nie sú priamo prepojené,

takže boli implementované oddelene. Všetky triedy majú verejný atribút s názvom `context` typu `string`, ktorý slúži pre uloženie akýchkoľvek dát, ktoré si môže uložiť *SMT solver* alebo vyhľadávací algoritmus pre optimalizáciu vyhľadávania.

Okrem metód boli implementované aj dve menšie štruktúry, ktoré sú súčasťou tried: `localization_t` (v preklade lokalizácia), ktorej UML diagram sa nachádza na obrázku 4.5 a `parameter_t` (v preklade parameter), ktorej UML diagram je na obrázku 4.6.

Implementácia všetkých metód a štruktúr sa nachádza v súbore `graphs.cpp`. Všetky metódy majú popis vo formáte *Doxygen*[2].

4.2.1 Trieda `Graph`

Trieda `Graph` obsahuje predovšetkým potrebné atribúty a metódy potrebné pre prácu s grafom. Medzi základné atribúty, triedy `Graph` patrí: zoznam základných blokov, vstupný blok do grafu, výstupný blok z grafu, deklarácie premenných, vstupné parametre, testovacie požiadavky, nájdené cesty, testovacie cesty alebo názov funkcie. Na obrázku 4.1 je zobrazená reprezentácia triedy `Graph` pomocou UML diagramu.

Metódy:

Metódy obsahujúce slovo `print`

Metódy, ktoré vypíšu na štandardný výstup aktuálny stav grafu, zoznamu ciest alebo zoznamu testovacích ciest.

Metóda `createTestPaths(unsigned coveragePercentage)`

Metóda, ktorá po ukončení vyhľadávania začne vytvárať testovacie cesty dokiaľ nie je dosiahnutá prahová hodnota zadaná pomocou parametra `coveragePercentage`. Podľa vopred nastaveného typu testovacích ciest sa v prípade krátkych ciest začína tvorba testovacej sady výberom ciest s najkratšou dĺžkou. Naopak pri výbere najdlhších ciest sa generovanie testovacej sady začína najdlhšími cestami. Možnosť výberu typu ciest sa nastavuje v súbore `settings.hpp`.

Metóda `addBB(BasicBlock *basicBlock) : bool`

Pridávanie ďalšieho základného bloku do grafu je realizované pomocou tejto metódy. Pri vkladaní prvého základného je nevyhnutné, aby jeho identifikačné číslo (*ID*) bolo 0. Pri vkladaní ďalšie základného bloku je nutné dbať na to, aby jeho identifikačné číslo (*ID*) priamo nasledovalo za posledným. V prípade nedodržania týchto pravidiel je návratová hodnota *false*, v opačnom prípade *true*. Dôvodom takého prístup k overovaniu identifikačného čísla (*ID*) je, že základné bloky sú ukladané zaradom vektore a následná manipulácia s nimi by mohla spôsobiť nekonzistentnosť.

Metóda `canBeTestPath(Path *path) : bool`

Cesta môže byť testovacou cestou práve vtedy ak začína vo vstupnom základnom bloku grafu a končí vo výstupnom základnom bloku grafu. Táto metóda okrem tejto kontroly skontroluje aj poradie všetkých základných blokov, či existuje hrana z predchádzajúceho základného bloku do kontrolovaného základného bloku. V prípade ak cesta splní tieto všetky

[2] https://www.doxygen.nl/

Obr. 4.1: Trieda `Graph`

podmienky môže sa stať testovacou cestou a návratová hodnota je *true*. Akékoľvek porušenie pravidla znamená návratovou hodnotu *false*.

Metóda `coverage() : unsigned`

Metóda slúži na výpočet pokrytia testovacích požiadaviek. Pokrytie sa udáva v percentách ako zaokrúhlené celé číslo pomeru splnených testovacích požiadaviek voči všetkým testovacím požiadavkám. Splnenou testovacou požiadavkou sa rozumie taká testovacia požiadavka, ktorá bola splnená aspoň jednou testovacou cestou. Výpočet pomocou vzorca 4.1.

$$\text{pokrytie} = \frac{\text{splnené testovacie požiadavky}}{\text{všetky testovacie požiadavky}} * 100 \tag{4.1}$$

Metóda `allEdges() : vector<Path*>`

Metóda, ktorá vygeneruje všetky hrany v grafe. Generovanie prebieha spôsobom, že ku každému základnému bloku vygeneruje cestu o dĺžke nula, jednu alebo dva (v prípade, ak je základný blok ukončený rozhodovacou inštrukciou). Hlavné využite metódy je pri generovaní kritéria pokrytia hrán.

Metóda `allGraphPaths() : vector<Path*>`

Metóda, ktorá dokáže vytvoriť všetky existujúce cesty v grafe. Pri vytváraní všetkých ciest sa najprv vytvoria cesty s dĺžkou nula obsahujúce jednotlivé základné bloky grafu, následne sa tieto cesty sa vložia do fronty. Cesta sa vyberie z fronty a uloží do zoznamu všetkých ciest. Ak má cesta možné pokračovanie, tak sa vytvorí jej kópia, pridá sa ďalší základný blok a vloží sa naspäť do fronty. V prípade ak posledný uzol je rozhodovací, tak sa uložia do fronty dve nové kópie ciest s oboma možnosťami rozhodnutia. Zoznam všetkých ciest po úspešnom vyprázdnení fronty obsahuje všetky existujúce cesty grafu. Najväčšie využitie tejto metódy sa aktuálne používa pri generátore kritéria pokrytia hlavných ciest.

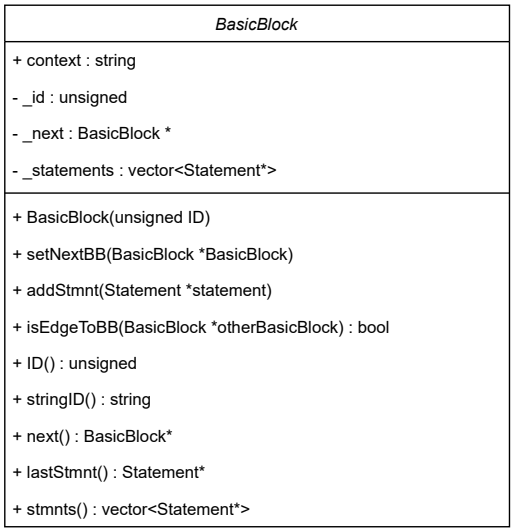
4.2.2 Trieda `BasicBlock`

`BasicBlock` v preklade základný blok ako už názov naznačuje, predstavuje najdôležitejšiu jednotku samotného grafu toku riadenia. `Základný blok` je priamočiara sekvencia príkazov idúcich bezprostredne za sebou bez akéhokoľvek vetvenia, skokov alebo cieľov skokov. Základnými atribútmi triedy `základný blok` sú: identifikačné číslo (*ID*), nasledujúci základný blok a zoznam špecifikácií. UML diagram triedy `základný blok` je na obrázku 4.2.

Metódy:

Konštruktor `BasicBlock(unsigned ID)`

`Základný blok` je jediná trieda, ktorá má konštruktor obsahujúci parameter. Okrem nastavenia základný hodnôt atribútov je u základného bloku dôležité aj jeho identifikačné číslo (*ID*), ktoré sa zadáva ako parameter konštruktora. Identifikačné číslo (*ID*) je uložené v atribúte `_id`, ktoré je počas existencie inštancie triedy `základný blok` nemenné.



Obr. 4.2: Trieda `BasicBlock`

Metóda `isEdgeToBB(BasicBlock *otherBasicBlock) : bool`

Metóda, ktorá overuje, či z daného základného bloku existuje hrana do iného základného bloku zadaného pomocou parametra `otherBasicBlock`. Hrana môže existovať priamo ako nasledovník základného bloku, v tom prípade je uložený daný skok blok ako atribút `_next` alebo hrana môže byť súčasťou podmieneného skoku, ktorý je súčasťou poslednej špecifikácie základného bloku. V prípade, ak sa podarí nájsť hranu do zadaného základného bloku je návratová hodnota *true*, inak *false*.

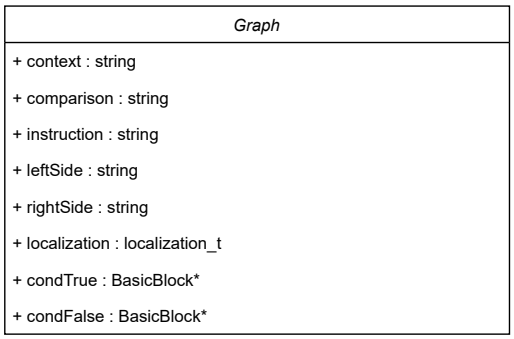
Metóda `stringID() : string`

Využívanie jedinečného identifikátora triedy `základný blok` je kľúčovou súčasťou zisťovania podciest. Metóda vracia identifikačné číslo (*ID*) ako slovo, ktoré pred identifikátorom ale aj za ním obsahuje špeciálny znak „|“. K implementácii špeciálnych znakov sa pristúpilo z dôvodu nesprávneho zisťovania podciest. Zisťovanie podciest prebieha na základe vyhľadávania názvu cesty ako podreťazca v názve druhej cesty, ktorý sa najprv skladal iba z identifikačného čísla (*ID*) a znaku „-“, ktorý reprezentoval hranu. Pri tomto spôsobe ukladania názvu ciest sa mohlo stať, že napr. cesta „5-4“ sa tvárila ako podcesta cesty „15-4“. Nakoľko posunutie názvu cesty „5-4“ o jeden znak doprava presne spárovalo s cestou „15-4“ a prehlásilo ju za podcestu. Pridanie špeciálneho znaku oddeľujúceho čísla zabezpečuje, že k žiadnemu podobnému omylu nemôže nastať. Z názvu cesty „5-4“ sa stala cesta „|5|-|4|“ a z cesty „15-4“ sa stala „|15|-|4|“. Tieto nové názvy ciest už nedovoľujú podobnú chybu.

4.2.3 Trieda `Statement`

Trieda `Statement` obsahuje atribúty potrebné na ukladanie jednotlivých špecifikácii inštrukcie. Namiesto implementácia triedy `Statement` mohla byť vytvorená iba štruktúra s rovnakým názvom, nakoľko aktuálne nemá žiadne metódy. Z dôvodu potenciálu pridá-

vania metód v budúcnosti sme sa rozhodli pre implementáciu triedy. Všetky atribúty sú uvedené na obrázku 4.3.



Obr. 4.3: Trieda `Statement`

4.2.4 Trieda `Path`

Trieda `Path` v preklade cesta ponúka viacero užitočných metód, ktoré uľahčujú prácu *manažérovi* nástroja TRIP ale aj algoritmom pri generovaní ciest. Trieda `Path` obsahuje tieto základné atribúty: názov cesty, dosiahnuteľnosť cesty a jednotlivé základné bloky, z ktorých sa daná cesta skladá. Všetky atribúty a metódy obsahuje UML diagram na obrázku 4.4.

Metódy:

Metóda `isBBInPath(BasicBlock *wantedBasicBlock) : bool`

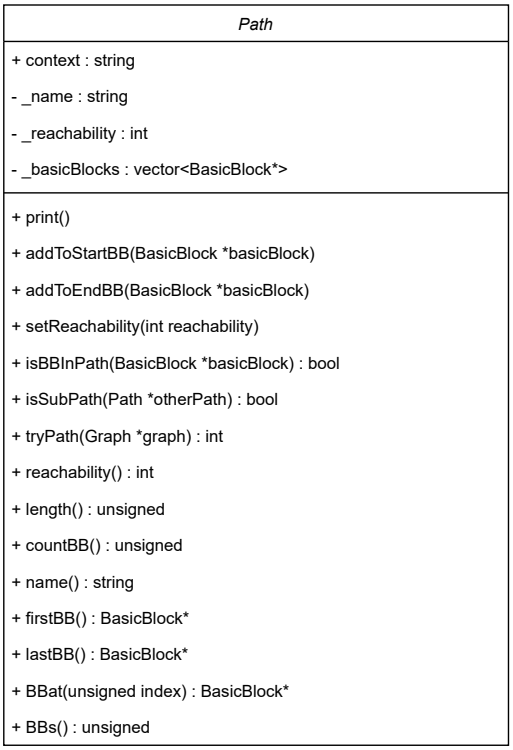
Metóda, ktorá overuje, či v danej ceste sa nachádza hľadaný základný blok. V prípade nájdenia je návratová hodnota *true*, inak *false*. Hlavné využitie funkcie je pri dvoch implementovaných vyhľadávacích algoritmoch.

Metóda `isSubPath(Path *otherPath) : bool`

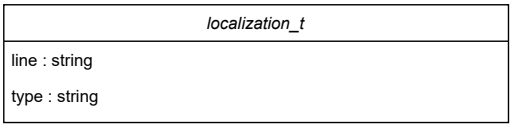
Jedná sa o najdôležitejšiu funkciu triedy `Path`. Zisťovanie, či je daná cesta podcestou inej cesty bude kľúčovou funkciu pri implementácii zložitých vyhľadávacích algoritmov. Práve z toho dôvodu sa pri tejto funkcii využíva vyhľadávanie podcesty pomocou hľadania podreťazca v názve cesty namiesto klasického porovnávania jedného základného bloku za druhým. Kvôli tejto metóde bolo nutné pridať špeciálne znaky „|“ k identifikačným číslam (*ID*) základných blokov, ktoré tak zabraňujú potenciálnemu nesprávnemu spárovaniu podreťazca.

Metóda `tryPath(Graph *graph) : int`

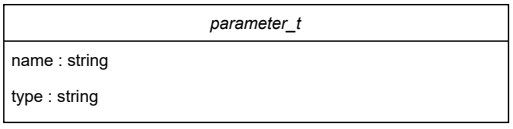
Metóda zabezpečujúca overenie sémantickej dosiahnuteľnosti cesty. Nakoľko na zisťovanie sémantickej dosiahnuteľnosti cesty sa využíva externý modul *SMT solver*, ktorý nie je momentálne súčasťou nástroja bolo potrebné ho nahradiť jedným z náhrady modulov 4.7. Následne návratová hodnota z *SMT solvera* vo formáte JSON je parsovaná a spracovávaná v časti *JSON parser*.



Obr. 4.4: Trieda `Path`



Obr. 4.5: Štruktúra `localization_t`



Obr. 4.6: Štruktúra `parameter_t`

4.3 Podporované kritéria pokrytia

Nástroj TRIP podporuje štyri kritéria pokrytia. Kritérium pre *pokrytia uzlov* (z ang. node coverage), kritérium pre *pokrytie hrán* (z ang. edge coverage), kritérium pre *pokrytie páru hrán* (z ang. edge-pair coverage) a kritériu pre *pokrytie hlavných ciest* (z ang. prime path coverage). Pre každé jedno kritérium pokrytia bol implementovaný samostatný generátor testovacích požiadaviek. Všetky štyri generátory sú implementované ako samostatné funkcie v zdrojovom súbore `algorithm.cpp`.

Pri implementácií algoritmov vytvárajúcich testovacie požiadavky na základe vybraného kritéria sa nedbalo na jeho vysokú efektivitu. Hlavnou požiadavkou bola jednoduchosť a priamočiarosť implementácie. Nástroj má predovšetkým slúžiť ako framework, ktorý sa bude využívať pri výučbe testovanie založeného na modeloch.

Generátor generateNodeCoverage(Graph *graph)

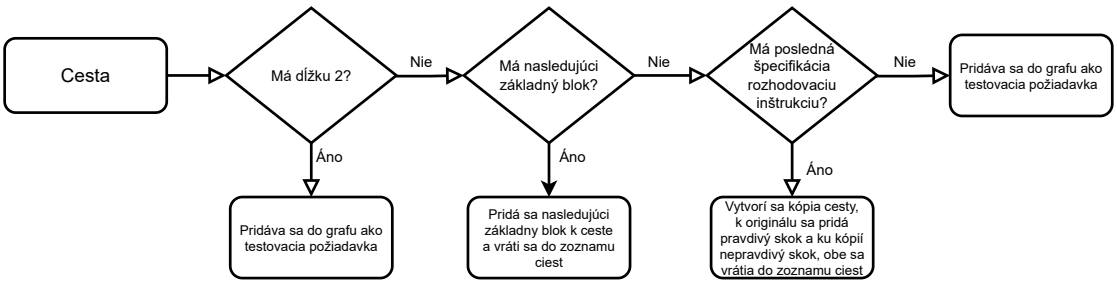
Generátor pre vytvorenie testovacích požiadavok pre kritérium *pokrytia uzlov*. Pri generovaní testovacích požiadaviek sa využíva metóda `BBs() : vector<BasicBlock*>` triedy `Graph`. Volána metóda vráti vektor všetkých základných blokov, ktoré následne samostatne jeden po druhom vytvoria testovaciu požiadavku, ktorá je pridaná do grafu.

Generátor generateEdgeCoverage(Graph *graph)

Generátor, ktorý vytvára testovacie požiadavky pre kritérium *pokrytia hrán*. Pri generovaní je využitá metóda `allEdges() : vector<Path*>` triedy `Graph`, ktorej návratová hodnota je vektor všetkých ciest o dĺžke 1. Tieto cesty sú následne pridané do grafu ako testovacie požiadavky.

Generátor generateEdgePairCoverage(Graph *graph)

Generátor pre vytváranie testovacích požiadavok pre kritérium *pokrytia páru hrán*. Najprv sa zo všetkých uzlov grafu získaných metódou `allEdges() : vector<Path*>` triedy `Graph` vytvoria cesty o veľkosti jedného základného bloku a uložia do zoznamu ciest. Následne sú tieto cesty spracovávané jedna po druhej. Graf spracovania jednotlivej cesty sa nachádza na obrázku 4.7.



Obr. 4.7: Graf toku spracovania cesty

Generátor generatePrimePathCoverage(Graph *graph)

Generátor slúžiaci na vytvorenie testovacích požiadaviek pre kritérium *pokrytia hlavných ciest*. Pomocou metódy `allGraphPaths() : vector<Path*>` triedy `Graph` sa získa vektor všetkých existujúcich ciest grafu, ktoré sú vo vektore zoradené vzostupne podľa dĺžky. Pred samotným vytváraním testovacích požiadaviek je vhodné daný vektor otočiť, nakoľko hlavné testovacie cesty nie sú žiadnou podcestou inej cesty. Práve otočené poradie vektora umožňuje zrýchlenie vytvárania testovacích požiadaviek, keďže potenciálna existujúca podcesta je odhalená skôr pri prechode vektorom ciest.

4.4 Implementácie vyhľadávacích algoritmov

Vyhľadávací algoritmus je kľúčovou súčasťou pre vytváranie ciest v grafe. Nástroj TRIP obsahuje dve slepé vyhľadávacie metódy. Konkrétne metóda *prehľadávania do šírky* (z ang. breadth-first search) a *prehľadávania do hĺbky* (z ang. depth-first search). Obe metódy boli mierne upravené pre potreby vyhľadávania ciest v grafe. Expandovaný základný blok sa môže pridať do cesty iba za predpokladu, že sa už nenachádza v expandovanej ceste. Dôvodom zavedenia tohto pravidla bolo, že prehľadávanie cyklov, ktoré boli upravené na základné bloky a skoky viedlo z zacykleniu. Zavedením tohto pravidla sa stratila možnosť nájdenia cesty, ktorá by obsahoval prejdenie cyklu ale ako demonštratívny príklad vyhľadávacieho algoritmu to je postačujúce. Pre doplnenie chýbajúcich testovacích ciest je možné pridať cesty pomocou importu a exportu ciest, ktorý sa nachádza v kapitole 4.5.

Pre správne fungovanie je nevyhnutné, aby algoritmy počas vyhľadávanie alebo expanzie jednotlivých uzlov kontrolovali inštanciu triedy `RunManagement` pomocou volania metódy `canRun() : bool`, ktorá vracia booleovskú hodnota na základe vypršania času pre vyhľadávanie ciest. V prípade návratu zápornej hodnoty z metódy je potrebné prehľadávanie ukončiť nakoľko prichádza na radu vytváranie testovacích ciest a mohlo by dôjsť k potenciálnym nekonzistenciám medzi dátami.

Samotná implementácia bola navrhnutá s ohľadom na prípadnú modifikáciu algoritmov alebo doplnenie vlastných vyhľadávacích algoritmov. Práve preto boli implementované do oddelených zdrojových súborov. Oba spomínané algoritmy sú implementované v samostatnom súbore `algorithm.cpp`. Výber používaného algoritmu sa realizuje v súbore `settings.cpp`.

Popis algoritmu pre vyhľadávanie do šírky:

1. Do fronty sa vloží vstupný základný blok grafu toku riadenia.
2. Pokiaľ je fronta prázdna alebo vypršal čas je vyhľadávanie ciest ukončené.
3. Vyberie sa prvá cesta z fronty.
4. Ak posledný základný blok vybratej cesty má nasledujúci základný blok, ktorý sa nenachádza v expandovanej ceste je tento základný blok pridaný k ceste a cesta je vložená naspäť do fronty, pokračuje sa bodom 2.
5. Ak posledný základný blok vybratej cesty má nasledujúci základný blok, ktorý sa už nachádza v exandovanej ceste, tak je cesta vložená do vektora nájdených ciest a pokračuje sa bodom 2.

6. V prípade ak neexistuje posledná špecifikácia posledného základného bloku vybratej cesty (zvyčajne sa jedná o základný blok bez špecifikácie inštrukcií, napr. koncový základný blok) je cesta vložená do vektora nájdených ciest a pokračuje sa bodom 2.

7. Ak existuje posledná špecifikácia posledného základného bloku ale jej posledná inštrukcia nie je rozhodovacia je cesta vložená do vektora nájdených ciest a pokračuje sa bodom 2.

8. Ak posledná špecifikácia posledného základného bloku vybratej cesty obsahuje rozhodovaciu inštrukciu, tak sa vytvorí kópia takejto cesty a k originálnej ceste sa pridá skok na základný blok v prípade kladného rozhodnutia inštrukcie a ku kópií cesty sa pridá skok na základný blok v prípade záporného rozhodnutia inštrukcie. Originál aj kópia cesty sa vložia do fronty a pokračuje sa bodom 2.

9. Jedná sa o chybný typ cesty, ktorá nie je uložená a pokračuje sa bodom 2.

Popis algoritmu pre vyhľadávanie do hĺbky:

Metóda *vyhľadávania do hĺbky* funguje na rovnakom princípe ako metóda pre *vyhľadávanie do šírky*, ktorá je vyššie podrobne popísaná akurát namiesto fronty sa používa `zásobník`. Využitie zásobníka zabezpečuje, že sa najprv expanduje jedná a tá istá cesta dokiaľ nedôjde k ukončeniu jej expanzie napr. bodom 5, 6 alebo 7, prípadne bodom 9 ak sa jedná o chybnú cestu.

4.5 Import a export ciest

Dáta sú medzi jednotlivými behmi nástroja TRIP ukladané pomocou JSON formátu v súbore. Popis JSON formátu, ktorý sa využíva na import a export ciest a testovacích ciest sa nachádza v kapitole 3.3 konkrétne v tabuľkách 3.9 a 3.10. Import a export je realizovaný pomocou zadania príslušných prepínačov pri spúšťaní nástroja TRIP. Popis správneho spúšťania importu a exportu sa nachádza v kapitole 4.1. Implementácia importu a exportu sa nachádza v súbore `jsonParser.cpp`. Import ciest zo súboru prebieha ihneď po vytvorení internej reprezentácie grafu toku riadenia a export prebieha a posledná časť behu nástroja TRIP. Príklad súboru importu alebo exportu v JSON formáte sa nachádza v prílohe D.

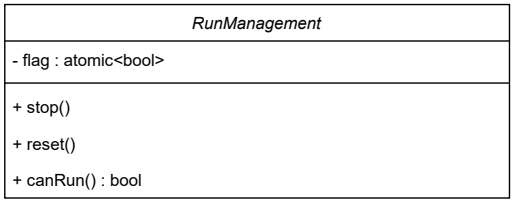
Import ciest zo súboru prebieha ihneď po vytvorení internej reprezentácie grafu toku riadenia. Počas importu prebieha validácia správnosti ciest. Kontroluje sa existencia základného bloku s identifikačným číslom (*ID*), z ktorého sa skladá vkladaná cesta. Následne prebieha kontrola, či existujú hrany medzi základnými blokmi, ktoré nasledujú za sebou v poli `basicBlocks` v JSON súbore. V prípade ak zlyhá niektorá z validácii je vypísaná chybová hláška na štandardný chybový výstup. Zlyhanie vkladania cesty nevedie k okamžitému ukončeniu programu.

Hlavné využitie importu a exportu je možnosť ukladať si už nájdené cesty v grafe medzi jednotlivými behmi nástroja TRIP. Medzi jednotlivými behmi je možné zmeniť vyhľadávací algoritmus ale zachovať si už nájdené cesty v grafe. Okrem toho je možné využiť import pre dosiahnutie úplného pokrytia kritéria, nakoľko je možné vložiť už otestované cesty, ktoré povedú k zvýšeniu celkového pokrytia.

4.6 Správa behu nástroja

Počas behu programu je nutné zabezpečiť ukončenie na základe vypršania času, nakoľko vyhľadávanie ciest môže v istých prípadoch zabrať nemalé množstvo času. Ukončenia vyhľadávania a reguláciu behu programu v čase zabezpečuje trieda `RunManagement`. Popis atribútov a metód pomocou UML diagramu sa nachádza na obrázku 4.8. Trieda je implementovaná v samostatnom súbore `manager.cpp`. Nastavovanie času pre vyhľadávanie ciest sa nachádza v súbore `settings.hpp`.

Manažér behu programu obsahuje dôležitý *atomický booleovský atribút* s názvom `flag`. Dôvodom pre použitie atomickej premennej je zachovanie konzistentnosti dát nakoľko program počas vyhľadávania beží paralelne [7].



Obr. 4.8: Trieda `RunManagement`

Princíp fungovania manažéra behu

Pred spustením samotného vyhľadávania sa beh programu rozdelí na dve jadrá. Jedno jadro prevádza samotné vyhľadávanie ciest a druhé reprezentuje manažéra behu. Jadro manažéra behu po spustení je uspané po dobu určenú pre vyhľadávanie ciest. Následne po prebudení je vykonaná zmena *atomického booleovského atribútu* na hodnotu *false*. Vyhľadávací algoritmus počas svojho behu kontroluje hodnotu atribútu manažéra behu programu pomocou metódy `canRun() : bool`. Akonáhle je nastavená hodnota na *false* je nutné vyhľadávanie ukončiť a prichádza na rad ďalšie spracovanie dát.

4.7 Náhrada externých modulov

Spustiteľnosť nástroja TRIP je závislá na tzv. *stubs*. *Stub* je funkcia, ktorá simuluje originálnu funkciu a bez ohľadu na parametre vykonáva tú istú úlohu [9]. Nástroj TRIP obsahuje dve externé časti *GCC plugin* a *SMT solver*. Z dôvodu, že tieto časti ešte nie sú plnohodnotne implementované je potrebné ich funkcionalitu nahradiť.

Príklad *stub* v reálnom živote pre priblíženie fungovania. Zákazník príde do reštaurácie a u čašníka si objedná halušky. Po chvíli mu čašník donesie pizzu. Následne si zákazník objedná lososa. Čašník mu znova donesie pizzu. Zákazník si objedná cestoviny so syrovou omáčkou ale po čase mu čašník do tretice znova donesie pizzu. Bez ohľadu na objednávku čašník stále prinesie pizzu. Obdobne aj funkcia bez ohľadu na parametre a aktuálny stav behu programu vráti stále tu istú hodnotu.

Stub GCC plugina

GCC plugin je zodpovedný za vytvorenie reprezentácie grafu toku riadenia vo formáte JSON. Funkcia, ktorá je implementovaná na simulovanie originálnej funkcie pridá k názvu zdrojového súboru príponu `.json` a následne sa pokúsi daný súbor otvoriť a načítať z neho dáta. Obsahom súboru by mala byť reprezentácia grafu toku riadenia vo formáte JSON. Príklad takéhoto validného obsahu súboru je v prílohe C. Prípadne popis pre vytvorenie grafu toku riadenia v JSON formáte sa nachádza v prílohe 3.3.

Stub SMT solver

Komunikácia medzi *manažérom* nástroja TRIP a *SMT solverom* prebieha pomocou JSON formátu. Implementovaná funkcia simuluje dosiahnuteľnosť cesty bez akýchkoľvek obmedzení. Funkcia vracia hodnotu `{"reachability" : 1}`. Hodnota `1` znamená, že cesta je dosiahnuteľná. Všetky typy dosiahnuteľnosti sú definované v súbore `graphs.hpp`, takisto aj stub používa hodnotu definovanú zo súboru. Číslo `1` je použité ako demonštratívna hodnota.

4.8 Návratové kódy

Riadne ukončenie programu vedie k návratovej hodnote `0`. V prípade neočakávaných dát alebo chovania je program ukončený s príslušným návratovým kódom a výpisom chybového hlásenia na štandardný chybový výstup. Zoznam všetkých návratových kódov s popisom chyby je v tabuľke 4.2.

Návratový kód	**Popis**
0	Riadne ukončenie programu
1	Neplatné vstupné argumenty programu
2	Zlyhalo otváranie súboru
3	Zlyhalo parsovanie JSON formátu
4	Nesprávne poradie identifikačných čísel (*ID*) základných blokov
5	Graf nemá vstupný základný blok
6	Graf nemá výstupný základný blok
7	Percentá zadané mimo rozsah 0 až 100
8	Nenájdené základné bloky vo vstupnom JSON formáte
9	Konkrétna špecifikácie bez inštrukcie

Tabuľka 4.2: Návratové kódy s popisom

Kapitola 5

Testovanie nástroja

Predposledná kapitola popisuje testovanie testovacieho nástroja. Môže to znieť paradoxne ale testovanie akéhokoľvek programu je veľmi dôležitou súčasťou vývoja. Pri testovaní boli implementované jednotkové testovanie (z ang. unit testing) a integračné testovanie (z ang. integration testing). Okrem týchto dvoch typov testovania bolo implementované aj testovanie nesprávnych vstupných argumentov pomocou piatich rôznych variant.

Testovanie prebiehalo pomocou vytvárania kontrol (`assert`) zo štandardnej knižnice `<assert.h>`, ktoré overovali správnosť hodnôt. Každá jedna kontrola (`assert`) obsahuje logický výraz, ktorý pre splnenie musí byť pravdivej booleovskej hodnoty. Celkovo ich bolo implementovaných 108.

Testy boli implementované pomocou nástroja *ctest*, ktorý je súčasťou komplexného nástroja *CMake* [10].

5.1 Spustenie testov

1. `cmake -S. -Bbuild`
2. `cd build`
3. `cmake -build .`
4. `ctest` alebo `make runTests`

5.2 Jednotkové testy nástroja

Pri jednotkovom testovaní bola overovaná správna implementácia hlavných tried a metód, ktoré sú popísané v kapitole 4.2. Každá metóda bola overená aspoň raz. V nasledujúcich tabuľkách je ku každej metóde napísaný počet implementovaných kontrol.

Testované metódy z testovacieho súbor `basicBlocks.cpp`

Názov metódy	Počet kontrol
`setNextBB(BasicBlock *BasicBlock)`	1
`addStmnt(Statement *statement)`	3
`isEdgeToBB(BasicBlock *otherBasicBlock)`	2
`ID()`	2
`stringID()`	2
`next()`	3
`lastStmnt()`	1
`stmnts()`	1

Tabuľka 5.1: Testované metódy z testovacieho súbor `basicBlocks.cpp`

Testované metódy z testovacieho súbor `paths.cpp`

Názov metódy	Počet kontrol
`print()`	1
`addToStartBB(BasicBlock *basicBlock)`	2
`addToEndBB(BasicBlock *basicBlock)`	5
`setReachability(int reachability)`	1
`isBBInPath(BasicBlock *basicBlock)`	4
`isSubPath(Path *otherPath)`	6
`tryPath(Graph *graph)`	1
`reachability()`	1
`length()`	1
`countBB()`	1
`name()`	1
`firstBB()`	1
`lastBB()`	1
`BBat(unsigned index)`	3
`BBs()`	3

Tabuľka 5.2: Testované metódy z testovacieho súbor `paths.cpp`

Testované metódy z testovacieho súbor `graphs.cpp`

Názov metódy	Počet kontrol
`print()`	2
`printPaths()`	2
`printTestPaths()`	2
`setEntryB(BasicBlock *entryBlock)`	1
`setExitB(BasicBlock *exitBlock)`	1
`addPath(Path *path)`	4
`addTestReq(Path *testRequirement)`	2
`addParam(parameter_t parameter)`	2
`addDeclar(parameter_t declaration)`	2
`createTestPaths(unsigned percentageCoverage = 100)`	1
`addBB(BasicBlock *basicBlock)`	6
`addTestPath(Path *path)`	8
`canBeTestPath(Path *path)`	6
`countNodes()`	1
`countEdges()`	1
`coverage()`	3
`params()`	5
`declars()`	5
`entryB()`	3
`exitB()`	3
`BBat(unsigned index)`	2
`BBs()`	2
`paths()`	1
`testPaths()`	5
`pathsLength(unsigned length)`	3
`allEdges()`	1
`testReq()`	1
`allGraphPaths()`	1
`pathsNames()`	1

Tabuľka 5.3: Testované metódy z testovacieho súbor `graphs.cpp`

5.3 Integračné testy nástroja

Integračné testovanie overuje fungovanie jednotlivých celkov nástroja. V rámci implementovaných integračných testov boli overené správnosti všetkých štyroch generátorov požiadaviek na test. Následne boli overené oba vyhľadávacie algoritmy a funkčnosť importu a exportu ciest. Taktiež bola overená funkčnosť metód triedy `RunManagement`, nastavením časovača na jednu sekundu, ktorá po jej uplynutí úspešne ukončí testovanie. Integračné testy sú implementované v súbore `coverages.cpp`, nakoľko všetky časti sa testujú popri testovaní pokrytí grafu toku riadenia.

Kapitola 6

Záver

Hlavným cieľom bolo navrhnúť a implementovať nástroj pre vyhľadávanie testovacích ciest, ktorý prakticky demonštruje testovanie založené na modeloch pri výučbe predmetu *Testovanie a dynamická analýza*[1]. Taktiež jedným z hlavných cieľov bolo nástroj navrhnúť tak, aby bolo možné jednotlivé externé časti vymieňať nezávisle na sebe.

Výsledkom práce je plne funkčné jadro nástroja TRIP, ktoré spĺňa všetky predpoklady a ciele naň kladené. Obsahuje štyri kľúčové triedy potrebné pre vyhľadávanie testovacích ciest. Nástroj TRIP aktuálne podporuje štyri rôzne kritéria pokrytia, ktorých pokrývanie zabezpečujú dva základné vyhľadávacie algoritmy.

Všetky štyri body zadania sa podarilo úspešne splniť. Od naštudovania problematiky, cez návrh, implementáciu až po vytvorenie sady automatických testov. Jadro nástroja TRIP je pripravené na integráciu *GCC plagina* a *SMT solvera*, ktoré dotvoria celý nástroj. Veríme, že výsledný nástroj bude vhodným frameworkom pre výučbu automatického generovania testovacích sád, ktoré v budúcnosti odhalia nemalé množstvo chýb.

V budúcnosti plánujeme nástroj vylepšiť o ďalšie funkcionality, ako napríklad implementácia ďalších kritérií pokrytia alebo zdokonalenie vyhľadávacích algoritmov. Taktiež plánujeme vytvoriť *Docker*[2] kontajner, ktorý pridá jednoduchosť používania celého nástroja.

[1] https://www.fit.vut.cz/study/course/231029/
[2] https://www.docker.com/

Literatúra

[1] Ammann, P. *Introduction to software testing*. New York: Cambridge University Press, 2008. ISBN 978-0-521-88038-1.

[2] Association, E. *5th Competition on Software Testing (Test-Comp 2023)* [online]. 2023. Dostupné z: https://test-comp.sosy-lab.org/2023/results/results-verified/.

[3] Cimatti, A., Griggio, A., Schaafsma, B. J. a Sebastiani, R. The MathSAT5 SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg. Lecture Notes in Computer Science. ISBN 9783642367410.

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L. a Stein, C. *Introduction to algorithms*. 3rd. MIT press, 2009. ISBN 9780262033848.

[5] Dwarakanath, A. a Jankiti, A. Minimum Number of Test Paths for Prime Path and Other Structural Coverage Criteria. In: *Testing Software and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. Lecture Notes in Computer Science. ISBN 3662448564.

[6] Exterman, D. *GCC vs Clang: Battle of the Behemoths* [online]. 2021. Dostupné z: https://www.incredibuild.com/blog/gcc-vs-clang-battle-of-the-behemoths.

[7] group cppreference. *Std::atomic* [online]. 2023. Dostupné z: https://en.cppreference.com/w/cpp/atomic/atomic.

[8] *IEEE/ISO/IEC International Standard - Software and systems engineering–Software testing–Part 4: Test techniques*. IEEE, 2021.

[9] Jung, J. *How to test software, part I: mocking, stubbing, and contract testing* [online]. 2019. Dostupné z: https://circleci.com/blog/how-to-test-software-part-i-mocking-stubbing-and-contract-testing/.

[10] Kitware, I. a Contributors. *CMake Reference Documentation* [online]. 2023. Dostupné z: https://cmake.org/cmake/help/latest/index.html.

[11] Lohmann, N. *JSON for Modern C++* [online]. 2023. Dostupné z: https://github.com/nlohmann/json.

[12] Loulergue, F. a Wotawa, F. FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs. In:. Switzerland: Springer International Publishing AG, 2021, sv. 12740. ISBN 3030793788.

[13] Pearson Education, I. *How the LLVM Compiler Infrastructure Works* [online]. 2008. Dostupné z: https://www.informit.com/articles/article.aspx?p=1215438.

[14] team, G. *GCC, the GNU Compiler Collection* [online]. 2023. Dostupné z: https://gcc.gnu.org/.

[15] Teamn, T. K. *KLEE Symbolic Execution Engine* [online]. 2023. Dostupné z: http://klee.github.io/.

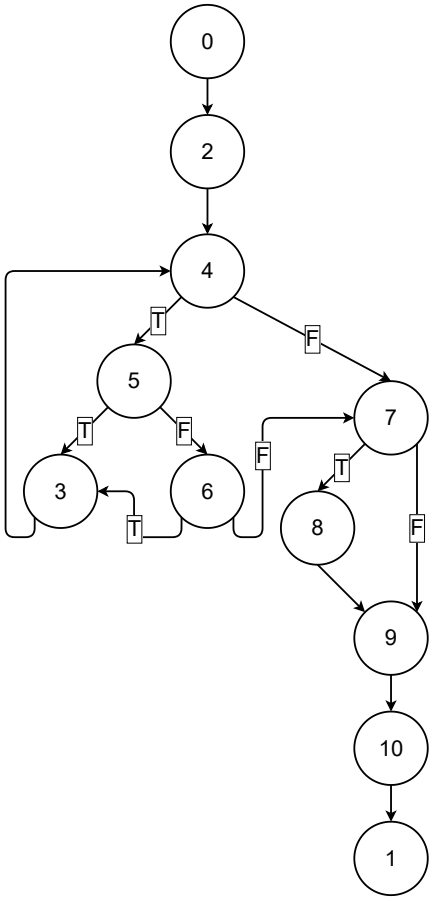
Príloha A

Zdrojový kód vzorového príkladu

```
int foo(int a, int b){
    a += bar(b);
    while(a-- < 0 && (b < 0 || b > 100))
        a = -a;
    if(a)
        b += a;
    b = b + 10;
    return b;
}
```

Príloha B

Graf tokuriadenia vzorového príkladu



Obr. B.1: Graf toku riadenia vzorového príkladu z prílohy A

Príloha C

Graf toku riadenia vzorového príkladu v JSON formáte


```
{
    "functionName" : "foo",
    "basicBlocks" : {
        "0" : {
            "statements" : [],
            "next" : 2
        },
        "1" : {
            "statements" : []
        },
        "2" : {
            "statements" : [{
                "localization" : {
                    "file" : "example.c",
                    "line" : 2
                },
                "statement" : {
                    "instruction" : "gimple_call",
                    "leftSide" : "D.1988",
                    "rightSide" : "bar (b)"
                }
            }, {
                "localization" : {
                    "file" : "example.c",
                    "line" : 2
                },
                "statement" : {
                    "instruction" : "gimple_assign",
                    "leftSide" : "a",
                    "rightSide" : "D.1988 + a"
                }
            }],
```

```
    "next" : 4
},
"3" : {
    "statements" : [{
        "localization" : {
            "file" : "example.c",
            "line" : 4
        },
        "statement" : {
            "instruction" : "gimple_assign",
            "leftSide" : "a",
            "rightSide" : "-a"
        }
    }],
    "next" : 4
},
"4" : {
    "statements" : [{
        "statement" : {
            "instruction" : "gimple_assign",
            "leftSide" : "a.0_1",
            "rightSide" : "a"
        }
    }, {
        "localization" : {
            "file" : "example.c",
            "line" : 3
        },
        "statement" : {
            "instruction" : "gimple_assign",
            "leftSide" : "a",
            "rightSide" : "a.0_1 + -1"
        }
    }, {
        "localization" : {
            "file" : "example.c",
            "line" : 3
        },
        "statement" : {
            "comparison" : "le_expr",
            "instruction" : "gimple_cond",
            "leftSide" : "a.0_1",
            "rightSide" : "0"
        },
        "jumps" : {
            "true" : 5,
            "false" : 7
        }
```

```
    }]
},
"5" : {
    "statements" : [{
        "localization" : {
            "file" : "example.c",
            "line" : 3
        },
        "statement" : {
            "comparison" : "le_expr",
            "instruction" : "gimple_cond",
            "leftSide" : "b",
            "rightSide" : "0"
        },
        "jumps" : {
            "true" : 3,
            "false" : 6
        }
    }]
},
"6" : {
    "statements" : [{
        "localization" : {
            "file" : "example.c",
            "line" : 3
        },
        "statement" : {
            "comparison" : "ge_expr",
            "instruction" : "gimple_cond",
            "leftSide" : "b",
            "rightSide" : "100"
        },
        "jumps" : {
            "true" : 3,
            "false" : 7
        }
    }]
},
"7" : {
    "statements" : [{
        "localization" : {
            "file" : "example.c",
            "line" : 5
        },
        "statement" : {
            "comparison" : "ne_expr",
            "instruction" : "gimple_cond",
            "leftSide" : "a",
```

```
            "rightSide" : "0"
        },
        "jumps" : {
            "true" : 8,
            "false" : 9
        }
    }]
},
"8" : {
    "statements" : [{
        "localization" : {
            "file" : "example.c",
            "line" : 6
        },
        "statement" : {
            "instruction" : "gimple_assign",
            "leftSide" : "b",
            "rightSide" : "b + a"
        }
    }],
    "next" : 9
},
"9" : {
    "statements" : [{
        "localization" : {
            "file" : "example.c",
            "line" : 7
        },
        "statement" : {
            "instruction" : "gimple_assign",
            "leftSide" : "b",
            "rightSide" : "b + 10"
        }
    }, {
        "statement" : {
            "instruction" : "gimple_assign",
            "leftSide" : "D.1993",
            "rightSide" : "b"
        }
    }],
    "next" : 10
},
"10" : {
    "statements" : [{
        "statement" : {
            "instruction" : "gimple_label",
            "leftSide" : "<L7>"
        }
```

```
            }, {
                "localization" : {
                    "file" : "example.c",
                    "line" : 8
                },
                "statement" : {
                    "instruction" : "gimple_return",
                    "rightSide" : "D.1993"
                }
            }],
            "next" : 1
        }
    },
    "entryBlock" : 0,
    "functionParameters" : [{
        "name" : "a",
        "type" : "int"
    }, {
        "name" : "b",
        "type" : "int"
    }],
    "declarations" : [{
        "name" : "D.1993",
        "type" : "int"
    }, {
        "name" : "D.1988",
        "type" : "int"
    }],
    "exitBlock" : 1
}
```

Príloha D

Ukážka importu a exportu v JSON formáte


```
{
    "paths": [
        {
            "basicBlocks": [
                0,
                1
            ],
            "context": "Edge",
            "reachability": 1
        },
        {
            "basicBlocks": [
                0,
                2
            ],
            "context": "Edge",
            "reachability": 2
        }
    ],
    "testPaths": [
        {
            "basicBlocks": [
                0,
                2,
                3
            ],
            "context": "Shortest path",
            "reachability": 1
        }
    ]
}
```

Príloha E

Ukážka výpisu grafu na rozhranie príkazovej riadky

```
Function name: foo

Entry basic block: 0

Entry basic block: 1

Parameters:
       Parameter 0:
              Type: int
              Name: a

Declarations:
       Declaration 0:
              Type: int
              Name: D.1993
Basic blocks:
       Basic block 0:
              Next basic block: 2

       Basic block 1:

       Basic block 2:
              Statements:
                     Statement 0:
                            Instruction: gimple_assign
                            Left side: time
                            Right side: 20
                            Localization:
                                   File: basicIf.c
                                   Line: 4

                     Statement 1:
```

```
                            Instruction: gimple_cond
                                .
                                .
                                .

Test requirements:
       Test requirement 0:
              Basic blocks:
                     Basic block: 0
                     Basic block: 2
                     Basic block: 4
                     Basic block: 5
                     Basic block: 6
                     Basic block: 1
              Name: |0|-|2|-|4|-|5|-|6|-|1|
              reachability: 0

       Test requirement 1:
              .
              .
              .

Test paths:
       Test path 0:
              Basic blocks:
                     Basic block: 0
                     Basic block: 2
                     Basic block: 3
                     Basic block: 5
                     Basic block: 6
                     Basic block: 1
              Name: |0|-|2|-|3|-|5|-|6|-|1|
              reachability: 1

       Test path 1:
              Basic blocks:
                     Basic block: 0
                         .
                         .
                         .
```