VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ MODELŮ PRO TESTY ZE ZDROJOVÝCH KÓDŮ

GENERATING OF TESTING MODELS FROM SOURCE CODE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. DANIEL KRAUT**
AUTHOR

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

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Kraut Daniel, Bc.**
Program: Informační technologie Obor: Inteligentní systémy
Název: **Generování modelů pro testy ze zdrojových kódů**
Generating of Testing Models from Source Code
Kategorie: Analýza a testování softwaru

Zadání:

1. Nastudujte testování založené na modelech. Nastudujte kritéria pokrytí pro testy založené na modelech programů. Seznamte se s infrastrukturou překladače LLVM/Clang.
2. Navrhněte systém pro generování sady testovacích cest na základě uživatelem zadaného kritéria pokrytí. Kritéria pokrytí by měla zahrnout kritéria nad tokem řízení, tokem dat a logickými výrazy.
3. Implementujte nástroj pro automatické generování testovacích cest ze zdrojových souborů. Integrujte nástroj jako subsystém platformy Testos.
4. Automatickými testy podpořte ověření funkcionality programu.

Literatura:

- Ammann, P.; Offutt, J.; *Introduction to Software Testing*, Cambridge University Press, 2008. ISBN 978-0-511-39330-3.
- Gyori, A.; Lahiri, S. K.; Partush, N. Refining interprocedural change-impact analysis using equivalence relations. 2017. In Proc. of ISSTA'17. doi: 10.1145/3092703.3092719.

Při obhajobě semestrální části projektu je požadováno:

- První 2 body zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Cílem této diplomové práce je navrhnout a implementovat nástroj pro automatické generování cest ze zdrojových kódů. V rámci práce byla nastudována problematika testování založeného na modelech a navrženo možné řešení automatického generátoru, dle kritérií pokrytí definovaných nad modelem CFG. Stěžejním bodem diplomové práce je návrh a popisuje implementaci nástroje. Nástroj podporuje množství kritérií pokrytí, které umožňují uživateli navrhovaného nástroje zaměřit se na konkrétní aspekt zkoumaného systému, který chce pokrýt testy. Navíc je nástroj přizpůsoben pro další požadavky na velikost generované testovací sady, reflektující reálné požadavky s praktickým využitím. Generátor byl implementován v jazyce C++ a webové rozhraní k němu v jazyce Python, které zároveň slouží pro integraci do platformy Testos.

Abstract

The aim of the masters thesis is to design and implement a tool for automatic generation of paths in source code. Firstly was acquired a study of model based testing and possible design for the desired automatic generator based on coverage criteria defined on CFG model. The main point of the master theis is the tool design and description of its implementation. The tool supports many coverage criteria, which allows the user of such tool to focus on specific artefact of the system under test. Moreover, this tool is tuned to allow aditional requirements on the size of generated test suite, reflecting real world practical usage. The generator was implemented in C++ language and web interface for it in Python language, which at the same time is used to integrated the tool into Testos platform.

Klíčová slova

automatický generátor, testování, model checking, kritérium pokrytí, tok řízení, tok dat, logický výraz, testovací cesta, LLVM, IR, CFG

Keywords

automatic generator, testing, model checking, coverage criteria, control flow, data flow, logical expression, test path, LLVM, IR, CFG

Citace

KRAUT, Daniel. *Generování modelů pro testy ze zdrojových kódů.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Generování modelů pro testy ze zdrojových kódů

Prohlášení

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

.....
Daniel Kraut
22. května 2019

Poděkování

Děkuji vedoucímu práce panu Ing. Aleši Smrčkovi, Ph.D. za odborné vedení práce a cenné rady, které byly velkým přínosem při tvorbě této práce.

Obsah

Kapitola 1

Úvod

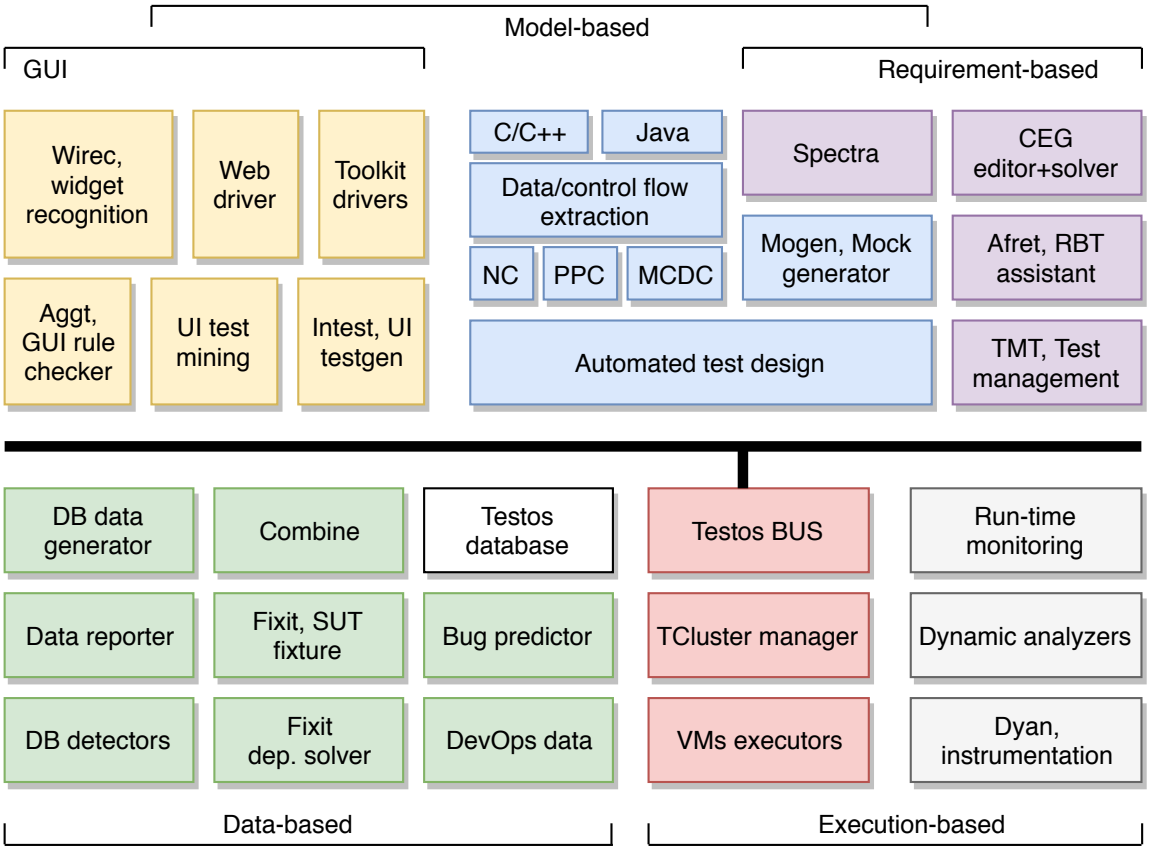
Tvorba softwaru je nelehký úkol. Tvorba softwaru, který funguje správně v různých podmínkách, je úkol daleko složitější, někdy až nedosažitelný. I proto stále více firem věnuje své úsilí na vytváření automatických kontrol vyvíjeného softwaru, které se nazývají softwarové testy. Cílem testování založeného na modelech je zobecnit testovaný systém a nabídnout uživateli způsob, jak zefektivnit tvorbu testů, tak aby pokryl co největší oblast chování zkoumaného systému. Tato práce nabízí způsob, jak tuto práci ještě více automatizovat.

V posledních letech lze pozorovat trend, kdy se většina procesů spojených s vývojem a nasazením softwaru automatizuje. Vznikají systémy propojených komponent, které automaticky přeloží, otestují, a dokonce vystaví software uživateli. Odhalil-li test chybu, je kladen důraz na co nejrychlejší uvědomění programátora s co nepřesnější lokalizací místa výskytu chyby.

Testování založené na modelech získalo největší zastoupení v oblastech kritických na bezpečnost lidí, majetku a prostředí. Jedním z důvodů, proč jiné oblasti využívající počítačový software nevyužívají modelové testování, jsou vyšší nároky na znalosti programátora implementujícího tyto testy. Ovšem s narůstajícím výpočetním výkonem a novými algoritmy vznikají technologie, které lze využít pro techniky automatické tvorby testů k redukci nebo eliminaci nároků na uživatele. Tato práce popisuje postup vývoje nástroje, který ze zdrojových kódů testovaného programu na vstupu vrátí uživateli sadu cest pro testy. Sada testů splňuje nároky definované teorií modelového testování a uživateli je nabídnuta možnost, jak množinu testů ovlivnit podle toho, na jaké kritérium modelu se zaměřuje. V rámci vývoje byl také kladen důraz na to, aby byl nástroj využitelný ve spojení s dalšími utilitami, které mohou vygenerované testy dále zpracovávat.

Diplomová práce a implementovaný nástroj byly vyvíjeny v rámci projektu Testos, který si klade za cíl vytvořit rozsáhlou platformu pro podporu softwarového testování. Testos je projekt vedený Alešem Smrčkou, Ph.D. Hlavní motivací projektu je vytvořit platformu umožňující širokou škálu přístupů k testování, nabídnout nástroje automatizující aktivity spojené s testováním a systém pro automatické hlášení výsledků testů a navedení programátora k možným chybám. Projekt s ambicemi zaštítit všechny tyto procesy okolo testování a verifikace softwaru zatím buď neexistuje, nebo o jeho existenci nejsou veřejně dostupné informace. Jedním z cílů této práce je připravit nástroj tak, aby byl integrovatelný do procesů platformy Testos.

Následující text seznámí čtenáře s vývojem navrženého nástroje. Úvodní kapitola 2 poskytuje teoretický úvod do testování založeného na modelech a přehled dostupných a použitých technologií. Na to navazuje kapitola 3 popisující návrh samotného nástroje, jeho architektury, algoritmů a webového rozhraní určeného pro integraci nástroje do platformy



Obrázek 1.1: Přehled oblastí zahrnutých v projektu Testos.

Testos. Kapitola 4 popisuje implementaci návrhu nástroje a webového rozhraní z kapitoly 3. Předposlední kapitola 5 ukazuje práci a výstupy implementovaného nástroje a nastiňuje případy, na které techniky použité v této práci nejsou schopny nalézt požadované cesty. Diplomovou práci zakončuje kapitola závěr 6, shrnující schopnosti nástroje a použitích technik analýzy.

Kapitola 2

Testování založené na modelech

Testování založené na modelech je rozvíjející se moderní přístup k testování softwaru. Zahrnuje tvorbu abstraktního modelu popisujícího chování zkoumaného systému a techniky, jak s modelem co nejlépe pracovat. Cílem je vytvářet softwarové testy efektivněji a zároveň vyzkoušet co nejvíce různých chování zkoumaného systému, dále jen *SUT* (z anglického System-Under-Test).

Kapitola 2.1 je teoretickým úvodem do oblasti přístupu ke generování testovacích případů v testování založeném na modelech. Navazující kapitola 2.2 popisuje základy práce s projektem LLVM, který byl použit pro vytvoření modelu. Poté je představeno několik současných projektů zaměřených na automatické testování, testování založeného na modelech a projekty používající LLVM IR 2.4.

2.1 Kritérium pokrytí

Kritérium pokrytí je pravidlo, nebo kolekce pravidel, které vynucují požadavky na testovací sadu. Kritérium pokrytí definuje požadavky úplně a jednoznačně. Testovací sada, *TS*, je množina testovacích případů [2].

Softwarový artefakt je subjekt programátora, pomocí něhož tvoří výsledný produkt, např. část kódu, řídící logika, práce s daty apod. *Test requirement* (požadavek na test) je specifický element softwarového artefaktu, který musí daný test splnit nebo pokrýt. Označme *TR* množinu požadavků na test.

Mějme množinu požadavků *TR* definující kritérium pokrytí *C*. Říkáme, že sada testů *TS* splňuje kritérium *C*, pokud pro každý požadavek *tr* z *TR*, existuje alespoň jeden test *t* z *TS*, který splňuje požadavek *tr*. V praxi se často setkáváme s procentuálním vyjádřením splnění *pokrytí*, které je dáno jako míra počtu splněných požadavků k celkovému počtu požadavků. Nejrozšířenějším měřítkem pokrytí je pokrytí všech řádků (*100% code coverage*), které znamená, že spuštění testů testovací sady projde programem tak, aby každý řádek zdrojového kódu byl proveden alespoň jednou. Kritéria vycházející z testování založené na modelech ukazují, že *100% code coverage* je často nedostatečné pro analýzu všech chování testovaného systému. V praxi může být pokrytí všech řádků někdy nedosažitelné a určitě není dostatečným ukazatelem dobré kvality programu [21].

Kritérium pokrytí grafů

Pro definování požadavků v rámci tohoto kritéria se předpokládá, že máme k dispozici model SUT ve formě grafu. Požadavky definujeme pro prvky grafu, tedy pro uzly, hrany a cesty.

Orientovaný graf je dvojice $G = (N, E)$, kde N je neprázdná množina uzlů a $E \subseteq N \times N$ je množina uspořádaných dvojic, tzv. orientovaných hran. *Cesta* v grafu je neprázdná sekvence uzlů $[n_1,\ n_2,\ ...,\ n_M]$ taková, že každá dvojice sousedících uzlů tvoří hranu, $(n_i, n_{n+1}) \in E,\ 1 \leq i < M$. *Délka cesty* je rovna počtu hran v dané cestě (délka cesty 0 je cesta o jediném uzlu). Uzel n je *syntakticky dosažitelný (reachable)*, z uzlu n_i, když existuje cesta z uzlu n do uzlu n_i. V teorii testování softwaru se často místo označení syntakticky dosažitelného uzlu používá *syntakticky dosažitelná cesta*, což je každá validní *cesta* v grafu reprezentující analyzovaný program. Označení syntakticky dosažitelné cesty se zavádí pro odlišení sémanticky dosažitelné cesty, definované v kapitole 3.6.

V kapitole 3.2 je dále definován *Graf toku řízení*, rozšiřující orientovaný graf pro účely modelování programu.

Jednoduchá cesta je taková cesta, ve které se žádný uzel neopakuje, s výjimkou cesty, ve které se počáteční uzel shoduje s koncovým. *Primární cesta* je taková jednoduchá cesta, která není podcestou žádné jiné cesty.

Node Coverage (NC): Kritérium pokrytí uzlů. Vyžaduje, aby *TR* obsahovaly každý syntakticky dosažitelný uzel. Kritérium je splněno, pokud pro každý syntakticky dosažitelný uzel n existuje cesta p, obsahující uzel n. Odpovídá pokrytí všech řádků kódu.

Edge Coverage (EC): Kritérium pokrytí hran. Požaduje, aby každá syntakticky dosažitelná cesta o délce 0 nebo 1 byla součástí *TR*.

Edge-Pair Coverage (EPC): Kritérium pokrytí párů. Vyžaduje, aby *TR* obsahovaly každou syntakticky dosažitelnou cestu o délce nejvýše 2.

Prime-Path Coverage (PPC): Kritérium pokrytí primárních cest. Vyžaduje, aby TR obsahovaly všechny primární cesty.

Z definic vyplývá, že z uvedených kritérií pokrytí je nejsilnější *PPC*, tzn. že zahrnuje požadavky minimálně pro pokrytí ostatních kritérií.

Kritérium pokrytí datových toků

Předmětem pozornosti kritéria pokrytí datových toků jsou místa v programu, kde se čtou a zapisují data. Sledováním těchto míst lze určit datové toky. Ve většině programovacích jazyků, jsou tato místa vyznačena zápisem nebo přístupem k datům skrze proměnné.

Pracujeme-li s modelem ve formě grafu, musíme si uzly označit podle toho, jak pracují s proměnnými v SUT. Funkce $def(n) = (v_1, v_2, ..., v_m)$ říká, že uzel n definuje, neboli zapisuje do proměnných $v_1, v_2, ..., v_m$. Funkce $use(n) = (v_1, v_2, ..., v_m)$ říká, které proměnné $v_1, v_2, ..., v_m$, uzel n používá, neboli z nich čte.

Dvojice *du-pair* z místa (uzlu) definice proměnné *def*, do místa (uzlu) jejího užití *use* určují datové toky. Cesta *du-path* pro proměnnou v je *jednoduchá* cesta $[n_1, n_2, ..., n_j]$, ve které první uzel n_1 definuje hodnotu proměnné: $v \in def(n1)$, poslední uzel n_j, proměnnou používá: $v \in use(n2)$, a žádný uzel mezi prvním a posledním uzlem proměnnou nedefinuje.

Kritéria pokrytí datových toků jsou definována nad seskupením množin cest du-path, tzv. *def-pair* množin a *def-path* množin. def-pair $du(n_i, n_j, v)$ je množina všech cest du-path pro proměnnou v, které začínají v uzlu n_i a končí v uzlu n_j. def-path $du(n_i, n_j)$ je množina všech cest du-path pro proměnnou v, které začínají v uzlu n_j.

All-Defs Coverage (ADC): Kritérium pokrytí definic. Vyžaduje, aby TR obsahovaly alespoň jednu cestu z každé množiny cest $du(n_i, v)$.

All-Uses Coverage (AUC): Kritérium pokrytí užití. Vyžaduje, aby TR obsahovaly alespoň jednu cestu z každé množiny cest $du(n_i, n_j, v)$.

All-du-Paths Coverage (ADUPC): Kritérium pokrytí všech du-cest. Vyžaduje, aby TR obsahovaly každou cestu z každé množiny cest $du(n_i, n_j, v)$.

Kritérium pokrytí logických výrazů

Logické výrazy slouží k určení vývoje průběhu programu. Typicky jsou programovacích jazycích reprezentovány výrazy typu *boolean*, používající formule výrokové logiky. Při abstrakci zdrojového kódu do grafu řízení instrukce s vyhodnocením logických výrazů znamenají větvení grafu. V modelu konečných automatů nebo UML stavového automatu se vyskytují jako *stráže (guards)* na přechodech mezi stavy automatu.

Predikát je výraz, který se vyhodnocuje na booleovskou hodnotu. Může obsahovat proměnné, relační operátory, volání funkcí a logické operátory booleovské logiky. *Klauzule* je predikát, který neobsahuje žádné logické operátory. Predikát se může skládat z libovolného počtu klauzulí spojených logickými operátory, ale sám se nesmí skládat z predikátů.

Nechť P je množina všech predikátů a C je množina všech klauzulí obsažených v predikátech z P. Pro každý predikát $p \in P$, nechť C_p je množina klauzulí v p, tedy $C_p = \{c | c \in p\}$. C lze definovat jako sjednocení $C = \bigcup_{p \in P} C_p$.

Predicate Coverage (PC): Kritérium pokrytí predikátů. Pro každý predikát $p \in P$, TR obsahuje dva požadavky: p musí být vyhodnoceno na *true* a p musí být vyhodnoceno na *false*.

Clause Coveraget (CC): Kritérium pokrytí klauzulí. Pro každou klauzuli $c \in C$, TR obsahuje dva požadavky: c musí být vyhodnoceno na *true* a c musí být vyhodnoceno na *false*.

Combinatorial Coverage (CoC): Kritérium pokrytí všech kombinací. Vyžaduje aby pro každý predikát $p \in P$, TR obsahovaly požadavky na vyhodnocení všech kombinací všech klauzulí z C_p.

Obecně platí, že pokrytí PC nezahrnuje CC a CC nezahrnuje PC. CoC zahrnuje obě, protože obsahuje všechny možné kombinace. Z praktického hlediska je CoC nepoužitelné, protože zahrnuje pro každý predikát s n klauzulemi 2^n možných pravdivostních ohodnocení klauzulí. Ovšem nedostatečná síla pokrytí PC a CC dala za vznik silnějším kritériím, jako kritérium pokrytí MCDC.

Majoritní klauzule c_i predikátu p určuje jeho hodnotu, pokud všechny *minoritní klauzule* $c_i \in p, j \neq i$ mají hodnoty takové, že změna pravdivostní hodnoty c_i změní pravdivostní hodnotu p.

Modified Condition/Decision Coverage (MCDC): Kritérium pokrytí změn podmínky a rozhodnutí. Vyžaduje aby pro každý predikát $p \in P$ a každou majoritní klauzuli $c_i \in C_p$, TR obsahovaly dva požadavky: c_i se vyhodnotí na *true* a c_i se vyhodnotí na *false*.

2.2 Překladač Clang a framework LLVM

LLVM kompilační framework je soubor modulárních a znovupoužitelných kompilátorů a nástrojů. LLVM začal jako výzkumný projekt [13] s cílem vytvořit moderní kompilační strategii založenou na SSA (Static single assignment) [22] podporující statickou i dynamickou

kompilaci libovolného programovacího jazyka. Od svého počátku se rozrostl na kolekci nejen kompilačních ale i analytických nástrojů. Celá infrastruktura je postavena na LLVM IR (LLVM Intermediate representation), což je nízkoúrovňový programovací jazyk, podobný asembleru. Optimalizační proces překladu se uplatňuje nejvíce během kompilace LLVM IR, ale i při linkování a za běhu (např. u JIT). LLVM nabízí back-end pro statickou nebo JIT (Just-in-Time) kompilaci pro širokou škálu architektur od x86-32 a x86-64, přes ARM a PowerPC, až po moderní RISC-V a WebAssembly nebo spíše ezoterické jako SPIR.

Jsou dostupné front-end kompilátory generující LLVM IR pro velké množství programovacích jazyků. Nejznámější je *Clang*, který umí přeložit C, C++ a Objective-C. *DragonEgg* je plugin do GCC pro překlad do LLVM IR, čímž přináší podporu pro jazyky FORTRAN, Ada a Go. Projekt *Mono* je framework pro překlad .NET jazyků jako C#, F# a Visual Basic. GHC obsahuje back-end pro překlad do LLVM. Existují méně známe projekty pro kompilaci dynamických jazyků jako Python, Ruby, Lua, a další.

Omezíme-li se na jazyky C, C++ a Objective-C, je překladač Clang implementován ve stejné filozofii jako LLVM, tedy s cílem maximalizace modularity a znovupoužitelnosti jeho částí. Clang podporuje 3 možnosti napojení se na proces kompilace. *LibClang* je vysokoúrovňové rozhraní v jazyce C pro použití částí překladače. Je vhodné, pokud chceme komunikovat s překladačem Clang z různých programovacích jazyků, např. Pythonu, a postačuje nám vyšší úroveň abstrakce nad procesem kompilace. *Clang Plugins* umožňuje přidávat dodatečné akce během interakce překladače s AST. Pluginy se načtou jako dynamické knihovny za běhu překladače a předávají se jako parametry při spuštění Clangu z příkazové řádky. *LibTooling* je rozhraní cílené na tvorbu vlastních nástrojů nebo při integraci služeb používajících Clang nástroje. Toto rozhraní nabízí plnou kontrolu nad AST v různých částech překladu. Skrze toto rozhraní již vznikla řada v praxi velice používaných nástrojů, mezi které patří nástroj pro automatické formátování kódu (clang-format), automatické opravy chyb překladu (clang-fixit), statické analyzátory (clang-tidy, clang-check), aj. V kontextu této semestrální práce přináší použití rozhraní překladače Clangu několik nevýhod:

- Omezení množiny cílených programovacích jazyků na C, C++ a Objective-C.
- Práce na úrovni AST znesnadňuje interoperabilitu s jinými nástroji.
- Potřeba spouštět vlastní nástroj vždy nad zdrojovým kódem.
- Většina uživatelem požadovaných optimalizací probíhá až při kompilaci LLVM IR.

Architektua LLVM

LLVM byl vyvinut jako kolekce knihoven, které implementují části kompilátoru. Tyto jednotlivé části mohou být buď vestavěny do existujících kompilátorů pro přidružení funkcionality LLVM, nebo vybraná skupina LLVM nástrojů pro příkazovou řádku může být použita pro přímý přístup k funkcím knihovny. Všechny knihovny pracují nad společnou přechodnou reprezentací nazvanou LLVM IR. Je možné exportovat LLVM IR mezi různými částmi kompilace a předat jej dalším nástrojům k analýze a úpravě.

2.3 Jazyk LLVM IR

LLVM IR je jazyk pro přechodnou reprezentaci (Intermediate Representation) ve formě SSA. Nabízí bezpečnost datových typů, nízkoúrovňové operace, flexibilitu a schopnost reprezentovat libovolné konstrukce vyšších programovacích jazyků. Je společnou reprezentací

zdrojového kódu přes všechny fáze LLVM kompilační strategie. LLVM IR je navržen tak, aby byl použitelný ve 3 formách: jako zkompilované IR v paměti, jako bytekód uložený na disku (ideální pro JIT), nebo v lidsky čitelné reprezentaci jazyka na úrovni asembleru. Všechny 3 formy jsou významově ekvivalentní, takže můžeme definovat operace nad textovou podobou LLVM IR a převést je do programu pracujícího nad modelem v paměti.

Struktura zdrojových souborů

Soubory obsahující program v bytekódu LLVM IR mají standardně příponu `.ll`. Počáteční řádky souboru obsahují metainformace o zdroji, ze kterého byly LLVM IR instrukce vygenerovány a informace o cílové architektuře. Další sekcí je deklarace globálních proměnných. Za nimi následuje definice samotného kódu.

Z pohledu abstrakce je nejvyšší strukturou *translation modul*, které se při linkování skládají do spustitelného programu. Překládáme-li například soubor se zdrojovým kódem jazyka C, obsahuje odpovídající modul všechny definice z překládaného souboru a všechny definice z připojených hlavičkových souborů direktivou `#include`. Modul se skládá, z kromě již zmíněných metainformací a globálních proměnných, z libovolného počet funkcí. Funkce jsou pojmenovány identifikátorem odpovídajícím identifikátoru funkce vyššího programovacího jazyka, ze kterého byly vygenerovány. Funkce se skládají ze *základních bloků* (basic block), zkráceně *BB*. Každému BB je přiřazeno jméno – návěstí (label) a je složen z instrukcí jazyka LLVM IR. Základní blok je největší posloupnost instrukcí, které se vždy vykonávají po sobě. Poslední instrukcí BB bývá instrukce skoku na jiný blok, nebo instrukce návratu z funkce.

Identifikátory

Identifikátory v LLVM IR jsou dvojího typu: globální nebo lokální. Globální začínají znakem `@` a lokální znakem `%`. Další vlastnost identifikátoru lze odvodit od formátu zápisu, určující 3 rozdílné účely:

1. Pojmenované hodnoty (proměnné) mají za prefixem řetězec znaků.
2. Hodnoty bez jména mají pouze prefix a číselnou hodnotu. Většinou existuje jeden čítač pro překládaný modul, přiřazující postupně čísla od nuly pro každou novou nepojmenovanou hodnotu.
3. Konstanty jsou psány přímo ve formátu daném odpovídajícímu datovému typu.

Datové typy

LLVM IR je silně typovaný, což znamená, že každý registr, funkční parametr, návratová hodnota funkce a instrukce má přidružený typ. Neexistuje implicitní přetypování. Jediný způsob, jak změnit datový typ je pomocí přetypování explicitního. Operandy předané do volání funkce nebo instrukce musí vždy odpovídat požadovanému datovému typu.

V kontextu této práce se zaměříme na proměnné s těmito datovými typy:

- Typ *integer*. Jednoduchý celočíselný typ, který definuje počet bitů, které zabírá. Zapisuje se ve formátu `iN`, kde N značí počet bitů. Povolený počet bitů je od 1 až do $2^{23} - 1$.

- Typ *pointer*. Typ pro ukazatel do specifického místa v paměti. Zapisuje se ve formátu `<type> *`, kde `<type>` je nutné nahradit za datový typ, na který ukazatel odkazuje. Není povoleno vytvářet ukazatele na typ *void*(typ nepředstavující žádnou hodnotu s nulovou velikostí) a na typ *label*.
- Typ *label*. Tento typ reprezentuje návěstí v kódu, většinou základní blok.
- Typ *array*. Složený typ reprezentující pole prvků stejného datového typu jdoucích po sobě v paměti. Zapisuje se pomocí hranatých závorek ve formátu `N x <type>`, kde N je konstantní celé číslo udávající velikost pole a `<type>` je datový typ prvků pole.
- Typ *structure*. Typ pro reprezentaci kolekce prvků společně v paměti. Prvkem může být libovolný typ, který má známou velikost. Zapisuje se výčtem prvků oddělených čárkou ve složených závorkách.

Datový typ pro práci s desetinnými čísly není nutný pro oblast této práce. Jeho přidání je možným námětem pro navazující projekty.

Příklady:

```
1  i1 ; integer o velikosti 1 bit, obdoba bool
2  i32 ; integer o velikost 32 bitu
3
4
5  [4 x i8] ; pole 4 integeru velikosti 8-bitu
6  [3 x [4 x i32]] ; pole 3x4 integeru i32
7
8  [4 x i32]* ; ukazatel na na pole obsahujici 4 hodnoty i32
9
10 { i8, i32 (i32) * } ; struktura obsahujici 8-bitove cislo a ukazatel na funkci
```

Výpis 2.1: Příklady datových typů v jazyce LLVM IR.

Metadata a ladící symboly

Překladače vyšších programovacích jazyků umožňují *překlad s ladícími symboly*. Výsledný binární soubor obsahuje důležité informace pro zpětné přiřazení přeložených instrukcí na místa ve zdrojovém kódu odkud vznikly. Patří mezi ně například původní názvy funkcí a proměnných. V jazyce LLVM IR se označují jako metadata.

Metadata obsahují dodatečné informace o původním zdrojovém kódu, včetně odkazů na původní řádek a sloupec, ze kterého daná instrukce vznikla. To lze využít pro zpětné informování uživatele o místě výskytu chyby.

Každá informace metadat je identifikovaná svým číslem a zapsána s vykřičníkem na začátku. V instrukcích se pak na příslušná metadata pomocí těchto identifikátorů odkazuje. Definice, co informace znamená, se zapisuje jako seznam na konec `*.ll` souborů.

Příklad metadat relevantních vzhledem k analýze pro testování:

```
1  !10 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
2  !15 = !DILocalVariable(name: "a", arg: 1, scope: !7, file: !1, line: 3, type: !10)
3  !22 = !DILocation(line: 5, column: 2, scope: !7)
```

Výpis 2.2: Ukázka metadat. 1. řádek je informace o původním datovém typu, 2. řádek jsou informace o lokální proměnné a na 3. řádku je odkaz na místo v původním zdrojovém souboru.

Mimo metadata vložená na konec souboru, obsahují základní bloky LLVM IR také deklarace lokálních proměnných. Ty jsou určeny použitím instrukce `call`, kde volaná procedura nese prefix `llvm.dbg` a její argumenty jsou metadata.

Instrukce

LLVM IR má poměrně limitovanou sadu instrukcí pro popis programu. Jsou reprezentovány pouze obyčejné operace. Překlad na specifičtější instrukce pro danou architekturu je ponechán na back-endu při překladu do strojového kódu.

Řídící instrukce určují tok běhu programu. Každý BB končí takovou instrukcí. Řídící instrukce hrají nejdůležitější roli při tvorbě grafu řízení programu. Mezi ty nejpodstatnější patří:

- `ret` - návrat řízení z funkce zpět k místu jejího volání. Tato instrukce určuje terminální BB.
- `br` - instrukce skoku, použita pro reprezentaci konstrukcí `if`, `while` a `for` vyšších programovacích jazyků. Pokud obsahuje pouze argument cíle skoku, jde o skok nepodmíněný. V případě podmíněného skoku obsahuje první hodnotu nebo identifikátor proměnné datového typu `i1` a další dva argumenty jsou cíle, pokud je podmínka rovna `true` nebo `false`. Cíle jsou datového typu `label`.
- `switch` - instrukce větvění pro reprezentaci konstrukce `switch` z vyšších programovacích jazyků. První argument je podmínka celočíselného typu, druhý je `label` výchozího cíle a třetí je pole dvojic (`integer` hodnota, `label` cíl) a skok na daný cíl je vybrán, pokud se jeho hodnota rovná hodnotě v prvním argumentu.

Instrukce s aritmetickými, logickými a bitovými operátory ve formě SSA vždy produkují nové hodnoty, nemodifikují hodnoty uložené v proměnných. Mezi jejich zástupce patří `div` (dělení), `fneq` (negace), `or` (logický OR), `shl` (bitový posun), aj. Většinou odpovídají operátorům z vyšších programovacích jazyků, před použitím optimalizací. Nejpodstatnější jsou zejména logické operátory, které ovlivňují tvorbu cest u kritéria pokrytí logických výrazů.

Instrukce pro přístup do paměti jsou klíčové pro tvorbu testovacích sad pro kritéria pokrytí datových toků. Mezi nejpodstatnější patří `load` (načtení dat), `store` (uložení dat) a `getelementptr` (získání adresy prvku ve složeném datovém typu).

Mezi další relevantní instrukce patří **komparační instrukce** `icmp` a **instrukce výběru výsledku** `phi` a `select`. Dokumentace LLVM IR tyto instrukce kategorizuje jako "ostatní" [15].

2.4 Existující nástroje

Podkapitola nabízí přehled v současnosti dostupných nástrojů zabývajících se stejnou problematikou. Mezi programy, které jsou volně dostupné, nebyl nalezen nástroj, který by přesně odpovídal nástroji navrhovaném v rámci této práce. Mezi často používané techniky statické analýzy patří symbolická exekuce a varianty techniky model-checking. Mezi níže uvedené nástroje patří: KLEE, NModel a PyModel, MoMuT, CMBC, Diffblue. Také byla popsán význam řešičů SAT a SMT, které jsou často využívány jinými nástroji.

Řešiče SAT a SMT

Boolean Satisfiability problém, zkráceně SAT, je definovaná jako test splnitelnosti logické formule prvního řádu. Zajímá nás, jestli je existuje ohodnocení všech proměnných ve formuli, aby formule byla splněná (výsledkem formule je *true*). Problém *SAT Modulo Theories*, zkráceně SMT, je rozšířením SAT o rozhodování formulí prvního řádu s ohledem na teorie popisující výrazy uvnitř proměnných formule. Nejčastěji jde o teorie nad aritmetikou s celými čísly, reálnými čísly nebo poli. Řešiče SAT nebo SMT jsou programy, které hledají konkrétní ohodnocení proměnných formule, aby byla splněná.

Velký pokrok ve schopnosti vyřešit ohodnocení formulí v relativně rozumném čase, který nastal v posledních dvou desetiletích, umožnil vzniku mnoha nástrojů pro statickou analýzu použitelných nad reálným softwarem. Většina z dostupných nástrojů používá ve svém jádru řešiče SAT nebo SMT. Mezi nejvíce používané řešiče SMT patří Z3, MathSAT, veriT, STP a další.

KLEE

KLEE je nástroj schopný automaticky generovat a provádět testy nad zkoumaným systémem. Je založen na symbolické exekuci a postaven na architektuře LLVM. Obsahuje interpreter instrukcí LLVM IR a udržuje si stav registrů a paměti jako symbolické proměnné. Ve svém jádru používá rešič SMT zvaný *STP constraint solver* [9] pro nalezení mrtvého kódu nebo nastavení symbolických proměnných tak, aby mohl spustit SUT pro danou cestu.

KLEE dosahuje velice dobré úrovně pokrytí. Již v jeho počátcích byla jeho síla demonstrována při použití na sadě nástrojů *GNU COREUTILS* a *BUSYBOX*, kde pokrytí všech řádků kódů bylo od 90% výše, pro některé z nástrojů až 100%. KLEE v těchto nástrojích odhalil 56 kritických chyb a řadu nekonzistentností. Tyto výsledky jsou z roku 2008, od kterého byl nástroj dále zdokonalován, ale jeho použití nemusí být vhodné u rozsáhlých projektů. Program KLEE je vydáván pod UIUC Open Source licencí.

NModel a PyModel

Oba tyto nástroje byly vytvořeny dle stejného přístupu ke tvorbě testů z modelu programu. Rozdíl je jen v použitém jazyce – NModel je napsán a přijímá model v C#, PyModel v Pythonu. Po uživateli nástroje je vyžadováno, aby vytvořil model chování sledovaného systému manuálně [12]. Pro modelování konečného chování, lze model specifikovat zápisem FSM. Pokud se požaduje zkoumání potencionálně nekonečného chování, je nutné model naprogramovat v odpovídajícím jazyce (C# nebo Pythonu) a pomocí nástrojů převést do požadované reprezentace, se specifikací omezení velikosti generovaného grafu. Na základě vytvořeného modelu jsou generovány testovací případy. Pro automatické spuštění je vyžadováno manuální napojení modelu na metody zkoumaného programu pomocí adaptéru. Výhodou manuálního napojení je, že zkoumaný program nemusí být nutně naprogramován v jazycích C# nebo Python.

MoMuT

MoMuT, vyvíjený na AIT ve Vídni, je sada nástrojů pro generování testů z manuálně připraveného modelu. Mezi modelovací jazyky patří *UML, Časované automaty, Rozhraní požadavků* a *Akční systémy*, buď v textové nebo i v grafické formě. MoMuT používá mutační strategii při generování testů [1]. Mutační testování je založeno na pozměňování validních

dat (vstupních nebo samotného programu) tak, aby se stala neplatnými. Pokud program produkuje stejné výsledky pro validní i nevalidní data, pravděpodobně obsahuje chybu. Přístup je založen na důkazu, že pokud jsou splněny podmínky mutačního testování, lze splnit pokrytí logických výrazů [20].

CMBC

CMBC je nástroj na automatickou tvorbu testů pro jazyky C a C++. Vstupní kód si nejdříve převede do modelu CFG, což odpovídá přístupu aplikovaného v rámci této práce. Nad tímto modelem provede "rozvinutí" cest, včetně cyklů, až do určité hloubky – omezení (bound), nad kterými provádí kontroly systému. Do vygenerované posloupnosti stavů systému vkládá aserce a interně si vytváří formuli podmínek spojených logickým operátorem AND. Tuto formuli dá na vstup řešiči SAT, aby zjistil, jestli rozvinutá cesta je sémanticky dosažitelná. Tato technika se jmenuje Bounded Model Checking [3]. Nástroj CBMC, implementující tuto techniku, je v síle porovnatelný s nástrojem KLEE, v některých případech i lepší [24].

Diffblue

Diffblue je proprietární program pro automatickou tvorbu testů. V jeho jádru je *"Diffblue AI"*, které slibuje lepší analýzu zdrojových programů v jazycích C, C++ a Java. Využívá jiných Open Source nástrojů, jako již zmíněného CMBC, nebo JBMC (pro Javu) [8], Fastsynth a další. Z oblasti *AI* využívá přizpůsobené algoritmy dynamického programování a odloženého Q-learningu pro systémů reprezentovaných pomocí modelu Markovovských rozhodovacích procesů [5].

V letošním roce spustila firma Diffblue online prostředí [18] pro demonstrační vyzkoušení schopností nástroje. Z praktického hlediska lze usoudit, že cíle nástroje Diffblue se velice podobají cílům této diplomové práce. Bohužel kvůli uzavřenosti jeho implementace nelze jednačně rozpoznat, které konkrétní algoritmy a techniky Diffblue interně využívá.

Kapitola 3

Návrh generátoru cest a webového rozhraní

V této kapitole je popsán návrh nástroje pro generování cest a jeho napojení do platformy Testos pomocí webového rozhraní. Nejprve jsou formálně definovány požadavky na výsledný nástroj. Poté je vysvětleno, jakým modelem je reprezentován zkoumaný program. Navazuje podkapitola 3.3, nabízející velmi abstraktní pohled na návrh jednotlivých částí nástroje. Generování cest je rozděleno na dvě části. Nejdříve je v podkapitole 3.4 vysvětleno co jsou v kontextu této práce *cíle* a jak je lze vygenerovat pro vybrané kritérium pokrytí. V následující podkapitole 3.5 jsou navrženy dva algoritmy, jak z *cílů* vygenerovat cesty. Předposlední část návrhu, podkapitola 3.6, upravuje předchozí algoritmy pro získání sémanticky splnitelných cest, které mohou již být využity pro reálné aplikace. Poslední podkapitola 3.7 se zabývá návrhem REST API, pro začlenění navrhované nástroje mezi další utility platformy Testos.

3.1 Formální definice požadavků

Tabulka 3.1 nabízí přehled požadavků na navrhovaný nástroj, které je nutné splnit. Řešení definovaných požadavků je popsáno buď v následujících podkapitolách návrhu, nebo v kapitole 4 zabývající se implementací navrhovaného nástroje. Pro lepší přehled, ve kterých kapitolách je řešen daný požadavek je k dispozici tabulka 3.2.

3.2 Reprezentace modelu programu

Mezi zástupce používaných modelů pro testování patří modely založené na jazycích UML a SysML, konečné automaty, časované automaty a další. V rámci této práce bude použita reprezentace pomocí **grafu toku řízení**, dále jen **CFG** (z anglického Control Flow Graph). CFG je orientovaný, neohodnocený graf, který potencionálně může obsahovat cyklické podgrafy. Každý uzel představuje jeden základní blok a hrany jsou přechody mezi nimi. Obecně je CFG definován jako čtveřice $G = (N, N_0, N_f, E)$, kde:

- N je konečná množina uzlů
- $N_0 \subseteq N, N_0 \neq \emptyset$ je neprázdná množina počátečních uzlů
- $N_f \subseteq N$ je množina koncových uzlů
- $E \subseteq N \times N$ je množina hran

ID	Popis	Kategorie	Závislost
REQ.1	Program generuje množinu cest pro zadaný vstupní zdrojový kód.	funkcionální	
REQ.2	Program podporuje C++14	nefunkcionální	
REQ.3	Zdrojový kód na vstupu je v jazyce LLVM IR.	funkcionální	
REQ.4	Uživateli programu je umožněno specifikovat kritérium pokrytí.	funkcionální	
REQ.5	Vygenerovaná množina cest musí splňovat zvolené kritérium pokrytí.	funkcionální	REQ.4
REQ.6	Mezi podporovaná kritéria pokrytí patří: NC, EC, EPC, PPC, ADC, AUC, ADUPC, PC a MCDC	funkcionální	REQ.4
REQ.7	Uživateli programu je umožněno ovlivnit velikost výsledné sady cest.	funkcionální	
REQ.8	Uživateli programu je umožněno specifikovat maximální délku vygenerované cesty.	funkcionální	
REQ.9	Program je možné spustit v kombinaci s dalšími nástroji z projektu Testos.	funkcionální	
REQ.10	Vstupní LLVM IR je v nezměněné podobě předán k další analýze.	funkcionální	REQ.3, REQ.9
REQ.11	Výstupní množina cest je předána ve standardním textovém formátu.	funkcionální	
REQ.12	Výstup programu je možné získat ve formátu JSON.	funkcionální	REQ.12
REQ.13	K programu je k dispozici skript pro nasazení webové služby na server a API pro ovládání funkcionality.	funkcionální	REQ.9
REQ.14	API webové služby je zdokumentováno	nefunkcionální	REQ.13
REQ.15	Program využívá externí nástroj pro ověření sémantické dosažitelnosti cesty.	funkcionální	
REQ.16	Program je schopen nahradit sémanticky nedosažitelnou cestu, pokud jiná dosažitelná cesta existuje.	funkcionální	REQ.16
REQ.17	Náhradní cesta neporušuje splnění vybraného kritéria pokrytí.	funkcionální	REQ.16

Tabulka 3.1: Požadavky na navrhovaný nástroj. Sloupce zleva: jednoznačný identifikátor, slovní popis požadavku, typ a závislosti na jiné požadavky.

ID požadavku	Kapitola	ID požadavku	Kapitola	ID požadavku	Kapitola
REQ.1	3.5	REQ.7	4.2	REQ.13	4.5
REQ.2	4.1	REQ.8	4.2	REQ.14	3.7
REQ.3	3.3	REQ.9	3.7	REQ.15	3.6
REQ.4	4.2	REQ.10	3.3	REQ.16	3.6
REQ.5	3.4	REQ.11	4.6	REQ.17	3.6
REQ.6	4.2	REQ.12	4.6		

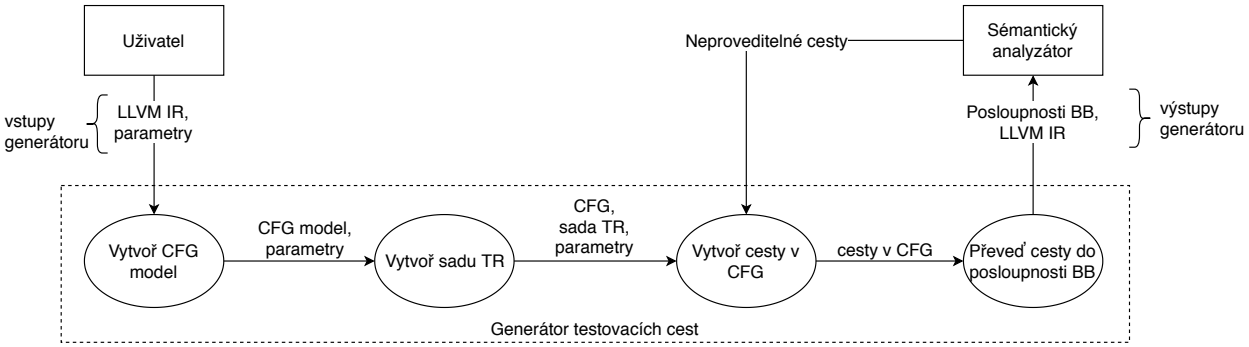
Tabulka 3.2: Odkazy na kapitoly, kde je daný požadavek vyřešen.

Pro řešení průchodu grafem obsahujícím cykly se zavádí pojem *silně vázaná komponenta*, zkráceně *SCC* (z anglického Strongly Connected Component). *SCC* v orientovaném grafu G je největší podgraf grafu G takový, že každý uzel uvnitř C je dosažitelný cestou složenou pouze z uzlů uvnitř C [6]. Silně vázané komponenty každého konečného orientovaného grafu (V, E) lze vyhledat pomocí Tarjanova algoritmu v čase $O(|E| + |V|)$.

Zaměříme se pouze na intraprocedurální analýzu, kde vstupním blokem je vždy pouze jediný bod – počátek analyzované funkce. Ačkoliv ve vyšších programovacích jazycích funkce obecně obsahuje více výstupních bodů, v reprezentaci LLVM IR je obvyklé mít jeden blok obsahující instrukci `ret` pro návrat z funkce. Pro jednoduchost proto předpokládejme, že množiny N_0 i N_f obsahují pouze jeden uzel. Není však obtížné rozšířit navrhované postupy pro definici N_f jako množinu s více uzly.

3.3 Architektura nástroje

Generátor potřebuje na vstupu kód ve formě LLVM IR (REQ.2) a parametry (kritérium pokrytí a velikost testovací sady). Výstupem generátoru je množina cest reprezentována posloupnostmi základních bloků v LLVM IR, včetně původního LLVM IR pro navazující analýzu (REQ.10).



Obrázek 3.1: Data flow diagram navrženého generátoru testovacích cest.

3.4 Generování cílů dle kritéria pokrytí

Generátor cílů je část navrhovaného nástroje, která analyzuje zadanou funkci a vytvoří množinu cest, které pokud budou obsaženy v testovací sadě, splní zvolené kritérium pokrytí. Označením **cíl** je myšlena cesta *cesta* složená z uzlů grafu CFG (čili ze základních bloků LLVM IR). V následujících podkapitolách je navrženo, jak vytvořit množinu cílů pro jednotlivá kritéria pokrytí. Množinu cílů pro danou funkci označujeme $TARGETS$. Správné vytvoření množiny cílů je první polovina splnění požadavku REQ.5.

Kritérium pokrytí grafu

Tato kritéria pokrytí pracují pouze s elementy grafu (uzly a hrany), proto se dá říct, že jsou "nejjednodušší" ke splnění, vzhledem k reprezentaci k představeným algoritmům. Splnění kritéria **NC** vyžaduje pokrytí všech uzlů grafu. Do množiny $TARGETS$ vložíme všechny uzly grafu, čili $TARGETS = N$, kde N je množina uzlů z grafu $CFG = (N, N_0, N_f, E)$. Algoritmus 3.2, který se snaží pokrýt co nejvíce cílů každou generovanou cestou, lze pro

kritérium NC nahradit optimálnější variantou. Využijeme algoritmu 3.1 pro krátké cesty, ve kterém po 5. kroku *rozšíříme* nalezenou cestu p_3 o průchod všemi takovými uzly grafu, které nejsou ještě součástí p_3 a zároveň SCC, do které náleží má neprázdný průnik s uzly p_3.

Pro splnění kritéria **EC** sledujeme hrany. Množinu *TARGETS* naplníme množinou hran E z *CFG*. Hrana je uspořádaná dvojice uzlů, tvořící cestu o délce 1.

Kritérium **EPC** vyžaduje každou cestu délky 2. Pro každý uzel grafu vygenerujeme všechny možné cesty o délce 2, vycházející z tohoto uzlu. Tyto cesty vložíme do množiny *TARGETS*.

Splnění kritéria **PPC** je nejsložitější. Je nutné v grafu vyhledat všechny primární cesty. Vytvoříme si tabulku všech jednoduchých cest o všech délkách. Začneme s množinou všech uzlů, tedy množinou všech cest o délce 0. Postupně a přidáváme do tabulky **jednoduché** cesty o 1 delší rozvětvením cest z předchozího kroku. Nejde-li již dále vytvářet cesty nové (každý cesta skončila v koncovém uzlu nebo v cyklu ve svém startovacím uzlu), označíme si primární cesty. Začneme od cest s největší délkou. Vybranou cestu označíme jako *primární* a z tabulky vyřadíme všechny její podcesty. Takto pokračujeme, dokud v tabulce nezůstanou jen primární *cesty*. Ty vložíme do množiny *TARGETS*.

Kritérium pokrytí datových toků

Pro splnění kritéria pokrytí datových toků musíme sledovat zápis a čtení z proměnných. V modelu CFG zatím tyto informace nemáme. Rozšiřme CFG na graf **DFG** (Data Flow Graph), $DFG = (N, N_0, N_f, E, V, def, use)$, kde:

- N, N_0, N_f, E jsou shodné jako v CFG
- $def : N \to V$ je funkce, která pro daný uzel vrací množinu všech proměnných, které se v tomto uzlu definují
- $use : N \to V$ je funkce, která pro daný uzel vrací množinu všech proměnných, které se v tomto uzlu používají

Výsledky funkcí *def* a *use* je vhodné předpřipravit, protože musíme procházet jednotlivé instrukce v uzlech – *základních blocích*. Sledujeme aritmetické instrukce, instrukce přístupu do paměti, komparační instrukce a instrukce výběru výsledku. Pomocí metadat určíme, jestli jde o přístup/zápis do skutečných proměnných v původním programu a ty pak přidáme do výsledné množiny pro daný BB.

Další přípravný krok je nalezení cest *du-path*. Pro každou proměnnou $v \in V$ a každý uzel $n \in N$, který proměnnou definuje ($def(n) \cap \{s\} \neq \{\}$), vytvoříme všechny *jednoduché* cesty začínající v uzlu n a končící v uzlech u, pro $\forall u \in N \wedge use(u) \cap \{s\} \neq \{\}$). Použijeme například algoritmus BFS, upravená tak, aby zahazoval cesty, ve kterých se vyskytne cyklus a cesta není jednoduchá. Poté vytvoříme seskupení *def-pair* a *def-path*, dle jejich definic (2.1).

Nyní můžeme vytvářet TR dle kritérií pokrytí. Pro splnění **ADC** vybereme libovolnou cestu *du-path* z množin *def-path* pro každou proměnnou a každý počáteční uzel. Tyto cesty vložíme do množiny cílů *TARGETS* a použijeme algoritmy jako v u pokrytí datových toků

Pro splnění **AUC** a **ADUPC** postupujeme stejně, ale vybíráme z množin *def-pair*. Pro **AUC** libovolnou cestu *du-path* pro každou proměnnou, každý počáteční a každý koncový uzel. Pro **ADUPC** vybíráme všechny, ne libovolnou.

Výběr "libovolné" cesty pro ADC a AUC lze korigovat externím nástrojem, který je schopen určit, zda lze nalézt takové vstupní parametry programu, aby spuštěný program

tuto cestu prošel. Předpokládá se, že takový nástroj bude vytvořen v platformě Testos. Pokud nástroj vyhodnotí, že cesta není dosažitelná, vybereme jinou cestu z nabízené množiny.

Kritérium pokrytí logických výrazů

Vzhledem k nízké úrovní a omezené sadě instrukcí LLVM IR, se podmínky ve vyšších programovacích jazycích generují jako posloupnost BB, ve které jeden BB odpovídá jedné klauzuli. Navíc front-endy generující LLVM IR využívají zkráceného vyhodnocování při překladu podmínek. Nejtěžší částí analýzy je nalezení celých výrazů a jejich reprezentace v grafu. V CFG je nutné nejdříve zrekonstruovat původní predikát, ze kterého klauzule vznikly. Víme, že vstupním bodem predikátu je nejlevější klauzule predikátu. Z tohoto uzlu můžeme algoritmem BFS projít všechny klauzule tak, že kontrolujeme, zda jsme stále v rámci jednoho predikátu. Rámec predikátu určíme z metadat v LLVM IR souboru, z definice `scope`. Všechny takto navštívené uzly podgraf v CFG. Podgraf označíme identifikátorem společným pro jeden predikát.

Z každého nalezeného predikátového podgrafu vedou hrany do 2 různých uzlů – jeden pro výsledek výrazu `true` a jeden pro `false`. Tyto dva uzly přidáme do podgrafu a označíme je jako koncové. Pro splnění kritéria **PC** stačí nalézt cestu z počátečního uzlu podgrafu obou koncových uzlů. Tyto 2 cesty vygenerujeme pro každý predikátový podgraf a výslednou množinu přidáme do *TARGETS*.

V praxi nejvíce vyžadované kritérium pokrytí z hlediska logických výrazů je MCDC. K jeho splnění musí TS obsahovat cesty, kde každá klauzule je vyhodnocena na `true` a `false` a celý predikát je alespoň jednou `true` a `false`. Využijeme nalezené predikátové podgrafy. Do *TARGETS* vložíme všechny hrany uvnitř tohoto podgrafu – hrany do koncových uzlů zajistí splnění změny výsledku predikátu a společně s ostatními hranami splnění výsledků klauzulí, viz obrázky 3.2 a 3.3.

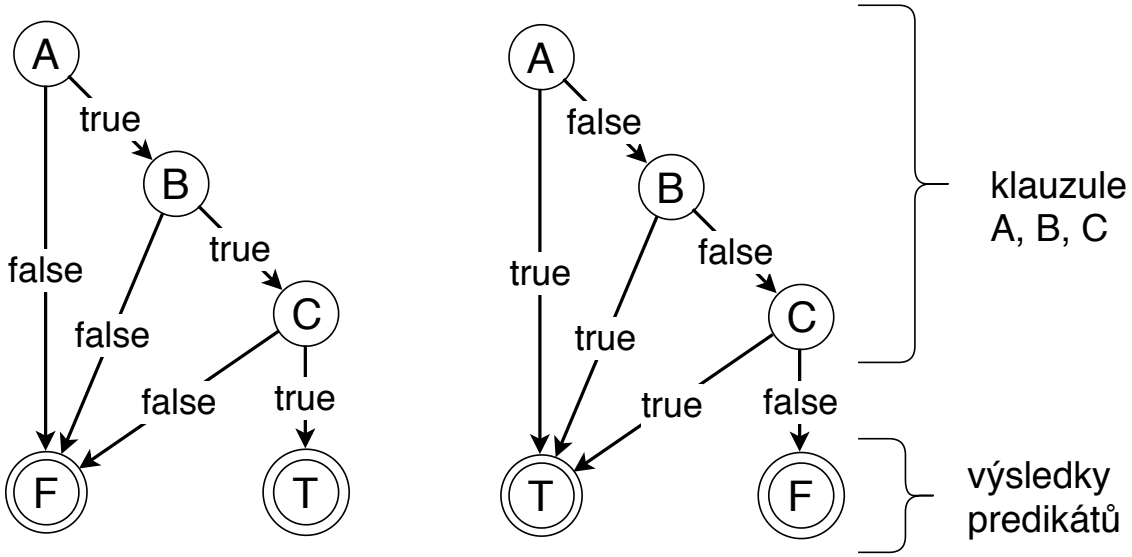
3.5 Generování cest programem z cílů

Každé z kritérií, zkoumaných v rámci této práce, se zaměřuje na jiný softwarový artefakt. Díky tomu, že modelujeme SUT pomocí CFG, lze dosáhnout dostatečné míry abstrakce na získání obecně aplikovatelných algoritmů, přes všechna kritéria. Tyto algoritmy se liší podle toho, co se v grafu vyhledává.

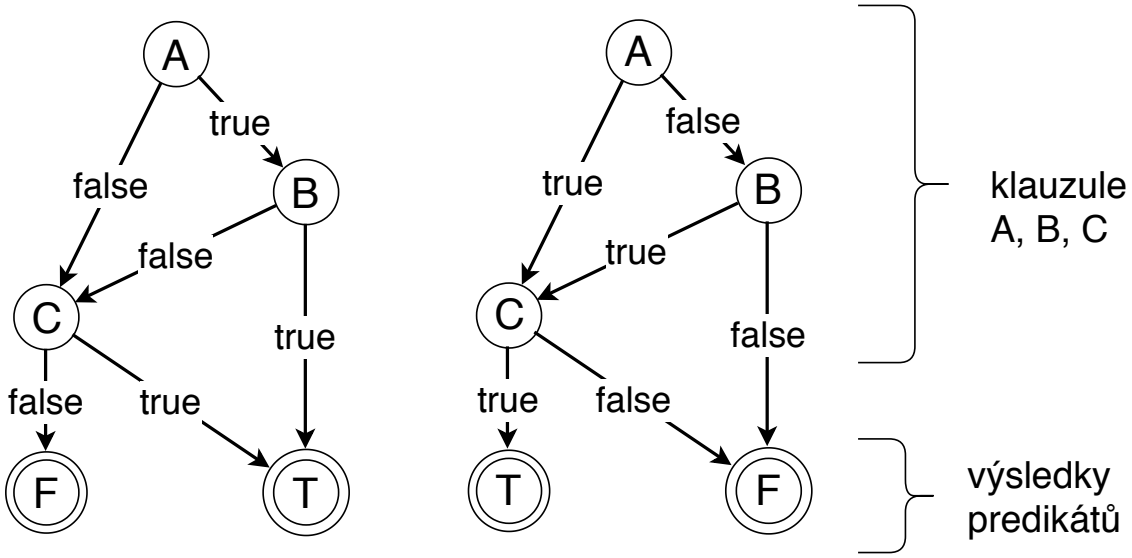
Z uživatelského hlediska jde přistupovat ke generování cest dvojím způsobem. Někdy je vyžadováno, aby množina testů TS obsahovala co nejmenší počet dlouhých cest. Tím se snižuje čas potřebných k jejich provedení, obzvláště u systémů, u kterých přípravné a úklidové práce spojené s testy (nazývané *setup* a *teardown*) zabírají velké množství prostředků. Nevýhodou takového přístupu je obtížné nalezení přesného místa, které chybu vyvolalo. Druhý způsob generování je přesně opačný – nalézt co největší testovací sadu s krátkými cestami. Tento přístup je naopak náročnější na spotřebované prostředky ke spuštění testů, ale je snazší nalézt epicentrum chyby.

Navržený nástroj podporuje obě varianty přístupu dle parametru od uživatele. Algoritmy pro obě varianty mají na vstupu graf *CFG* $G = (N, N_0, N_f, E)$ a množinu cílů *TARGETS* a na výstupu testovací sadu *TS*. Uzel v grafu lze definovat jako cestu o délce 0 a hranu jako cestu o délce 1. Množinu *TARGETS* tak můžeme definovat jako množinu cest, i pro kritérium pokrytí uzlů nebo hran.

Definujme si pomocné funkce:



Obrázek 3.2: Příklady predikátového podgrafu. Vlevo pro podmínku if (A && B && C) a vpravo pro podmínku if (A || B || C).



Obrázek 3.3: Příklady predikátového podgrafu. Vlevo pro podmínku if ((A && B) || C) a vpravo pro podmínku if ((A || B) && C).

- `shortest_path(`s`,` e`)` je funkce, která najde v grafu nejkratší cestu z uzlu s do uzlu e. V neohodnoceném grafu lze implementovat algoritmem BFS. Neexistuje-li taková cesta, funkce vrací prázdnou cestu.
- `first(`p`)` je funkce, která pro zadanou cestu $p = [n_1, n_2, ..., n_k]$ vrátí její první uzel n_1.
- `last(`p`)` je funkce, která pro zadanou cestu $p = [n_1, n_2, ..., n_k]$ vrátí její poslední uzel n_k.
- spojení cest $p_1 = [n_1, ..., n_i]$ a $p_2 = [n_i, ..., n_j]$, zapsané binární operací $p_1 + p_2$, vznikne nová cesta $p_3 = [n_1, ..., n_i, ..., n_j]$.

- `reachable(`n`)` je funkce, která pro zadaný uzel n vrátí množinu všech uzlů syntakticky dosažitelných z n

Algoritmus generující krátké cesty

Z dvou navržených tento jednodušší algoritmus postupuje tak, že vždy nalezne nejkratší cestu pro každý cíl z množiny *TARGETS* a odebere všechny cíle, které touto cestou pokryl. Končí, když jsou pokryty všechny cíle.

Vstup: množina cílů `TARGETS`
Výstup: množina cest `TS`

```
1  do
2    take t from TARGETS
3    p1 := shortest_path(N0, first(t))
4    p2 := shortest_path(last(t), Nf)
5    p3 := p1 + t + p2
6    add p3 to TS
7    for each subpath s in p3
8      TARGETS = TARGETS \ {s}
9  while TARGETS ≠ {}
```

Výpis 3.1: Algoritmus generující velkou testovací sadu s krátkými cestami

Algoritmus generující dlouhé cesty

Základní myšlenkou tohoto algoritmu je snaha poskládat co nejvíce cest z *TARGETS* "za sebe". Algoritmus si vytváří seznam *CONNECTIONS* (uspořádanou množinu) takových cest, které jdou na sebe navázat. Podmínkou dvojice navázatelných cest je, aby první uzel druhé cesty byl syntakticky dosažitelný z posledního uzlu cesty první.

Vstup: množina cílů `TARGETS`
Výstup: množina cest `TS`

```
1  do
2    take t from TARGETS
3    CONNECTIONS = {t}
4    for each path t in TARGETS do
5      let S be size of CONNECTIONS
6      for each path pi in CONNECTIONS, 1 ≤ i ≤ S do
7        if i == 1 && first(pi) ∈ reachable(last(t))
8          TARGETS = TARGETS \ {t}
9          insert t at the beginning of CONNECTIONS
10         GOTO 4
11       else if i == S && first(t) ∈ reachable(last(pi))
12         TARGETS = TARGETS \ {t}
13         insert t at the end of CONNECTIONS
14         GOTO 4
15       else if i < S && first(t) ∈ reachable(last(pi)) && first(pi+1) ∈ reachable(last(t))
16         TARGETS = TARGETS \ {t}
17         insert t in CONNECTIONS after pi
18         GOTO 4
19
20   take p from CONNECTIONS
21   for each path pi in CONNECTIONS do
22     p = p + shortest_path(last(p), first(pi)) + pi
```

```
23    add p to TS
24
25  while TARGETS ≠ {}
```

Výpis 3.2: Algoritmus generující malou testovací sadu s dlouhými cestami.

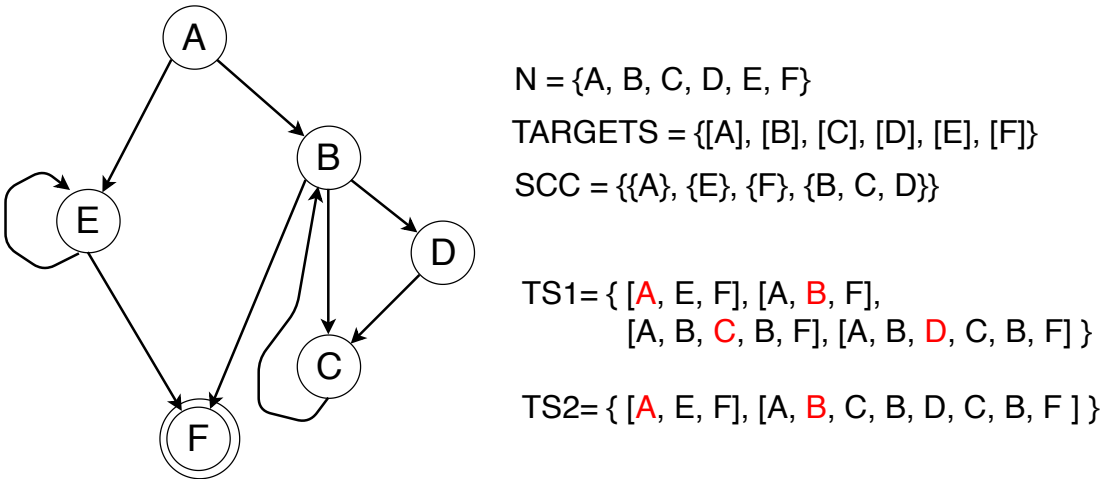
Pro zrychlení algoritmu si lze výsledky funkce `reachable(n)` předpočítat, pro všechny uzly grafu CFG. Definujme pomocnou funkci RTO(G), která pro zadaný orientovaný graf vrátí seznam silně vázaných komponent grafu, seřazený v opačném topologickém uspořádání (Reverse topological order). Tato funkce lze implementovat pomocí Tarjanova algoritmu používajícího DFS [16]. Návrh hledání `reachable(n)` pro všechna $n \in N$ popisuje algoritmus 3.3:

Vstup: CFG graf $G = (N, N_0, N_f, E)$
Výstup: výsledky funkce $reachable(n)$ pro všechna $n \in N$

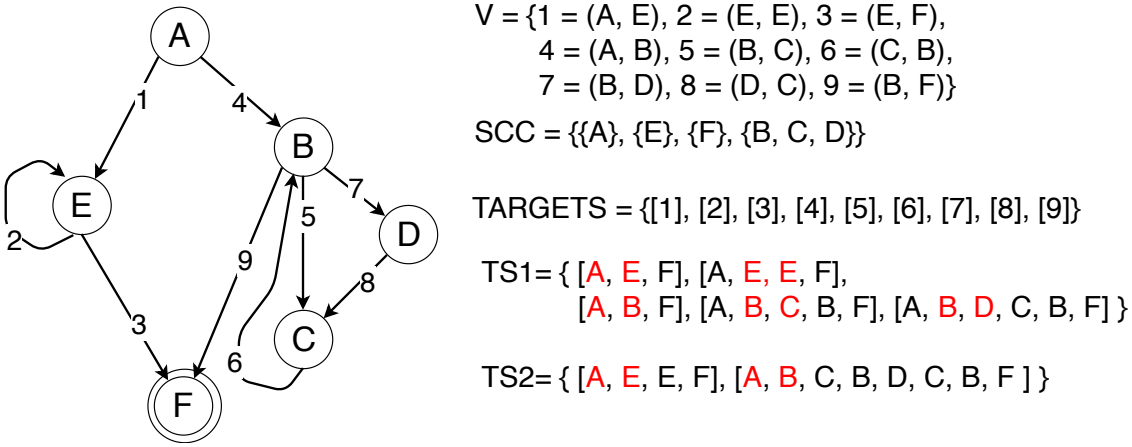
```
1  for each SCC s in RTO(G):
2     reach_s := {}
3     for each node n in SCC:
4        for each node m in {u|u ∈ N ∧ (n,u) ∈ E}:
5           if m ∩ s = ∅:
6              reach_s := reach_s ∪ m
7
8     for each node n in SCC:
9        reachable(n) = reach_s
```

Výpis 3.3: Algoritmus pro nalezení výsledků reachable(n), kde n je libovolný uzel grafu CFG

Úspěšná implementace alespoň jednoho ze dvou algoritmů 3.1 a 3.2 splňuje požadavek REQ.1 a požadavek REQ.5. Výsledky algoritmů lze vidět na obrázcích 3.4 a 3.5. *TS1* je testovací sada pro 1. variantu algoritmu *TS2* pro druhou variantu. Červenou barvou je označen aktuálně hledaný cíl z množiny *TARGETS*.



Obrázek 3.4: Příklad vygenerovaných cest pro kritérium NC.



Obrázek 3.5: Příklad nalezení cest pro kritérium EC.

3.6 Ověření sémantické dosažitelnosti cest

Mějme CFG graf G, modelující program P. **Sémanticky dosažitelná cesta** v grafu G je taková cesta t, pro kterou existuje reálné ohodnocení H vstupních parametrů takových, že program P spuštěný s parametry H navštíví cestu t (jinými slovy projde všemi uzly cesty t v posloupnosti definovaných cestou t).

Navrhovaný nástroj generuje cesty, u kterých je zajištěna syntaktická dosažitelnost (viz kapitola 2.1). Pro reálné použití nástroje na praktických příkladech je nutné zajistit sémantickou dosažitelnost. Generátor cest využije externího nástroje, který umožňuje pro zadanou cestu ověřit její sémantickou dosažitelnost. Paralelně k této diplomové práci je vyvíjen nástroj TINDGER na platformě Testos sloužící k tomuto účelu. TINDGER je blíže popsán v kapitole 4.1 a vzájemná komunikace v kapitole 4.4. Vyhodnocení, zda je cesta sémanticky dosažitelná splňuje požadavek REQ.15.

Postup ověřování sémantické dosažitelnosti

Syntakticky dosažitelná cesta, pro kterou chceme ověřit její sémantickou dosažitelnost, obsahuje části, které chceme zachovat (podcesty odpovídající cílům, které pokrývá) a části, které můžeme měnit (podcesty mimo cíle). Je nutné pouze zajistit, že výsledná cesta zůstane syntakticky dosažitelná, první uzel cesty zůstane vstupním uzlem funkce a poslední uzel cesty zůstane koncovým uzlem funkce. Z hlediska tvorby dotazů na externí nástroj pro ověření sémantické dosažitelnosti lze postupovat několika způsoby:

1. iterovat ověření jedné cesty, kterou v každé iteraci částečně změníme a zkusíme předat k ověření, dokud nenalezneme sémanticky dosažitelnou variantu
2. vygenerovat dávku alternativních cest, všechny je předat k ověření a nechat nástroj vybrat první sémanticky dosažitelnou cestu
3. kombinace prvních dvou metod

První metoda je snazší na implementaci, protože není třeba zpětně mapovat výsledek externího nástroje na vybranou alternativu a také není nutné hledat správný limit na počet vygenerovaných alternativ. Mezi nevýhody patří režie spojená s neustálým navazováním komunikace. Druhý způsob zefektivňuje vzájemnou komunikaci a může výrazně snížit rychlost

nalezení sémanticky dosažitelné cesty pomocí optimalizací nad dávkou cest (jádra nástrojů často umožňují ověření části, která je společná a uložení stavu pro navazující ověřování). Sémantická dosažitelnost patří mezi výpočetně velmi náročné operace. Nevýhodou je již zmíněný limit počtu alternativ, a také že vygenerovaná sada alternativních cest může obsahovat velkou množinu cest, jejichž počáteční část není sémanticky dosažitelná (například průchod přes vybranou větev konstrukce *if*) a nástroj musí složitě tyto alternativy eliminovat. Třetí způsob dotazování na externí nástroj má za snahu kombinovat výhodou obou předchozích přístupů, pro zvýšení rychlosti nalezení sémanticky dosažitelné cesty.

Algoritmus generování alternativních cest

Není-li cesta sémanticky dosažitelná, potřebujeme změnit pouze její vybranou část aby vedla jinými uzly grafu a zbytek cesty zůstal nezměněn. Algoritmus má na vstupu počáteční uzel n a koncový uzel m, kde uzel m je dosažitelný z uzlu n, a hledá všechny možné cesty mezi nimi. Algoritmus lze implementovat 2 způsoby:

1. přidávat do cesty cykly uvnitř SCC, kterými cesta prochází
2. pomocí algoritmu BFS

Druhá varianta, algoritmus BFS, je jednodušší z hlediska systematického procházení, protože nalezne cesty vždy od nejkratší po postupně delší. Interně používá k ukládání neprozkoumaných cest fronu. BFS lze implementovat v *lazy* variantě, kdy si objekt představující generátor interně uloží frontu neprozkoumaných cest a na požádání vygeneruje novou alternativu.

Hledání alternativ cesty s jedním cílem

Mějme syntakticky dosažitelnou cestu p, která začíná ve vstupním a končí v koncovém základním bloku analyzované funkce a obsahuje pokrývá cíl t (t je podcestou cesty p). Označme t_{pref} nejdelší podcestu cesty p končící počátečním uzlem t,. Označme t_{post} nejdelší podcestu cesty p začínající koncovým uzlem t. Nazvěme t_{pref} *prefix cíle* t a t_{post} nazvěme *postfix cíle* t.

Generujeme-li cestu pro jediný cíl, navazujeme na algoritmus pro hledání krátkých cest 3.1. Použijeme 2 generátory alternativ 3.6 – jeden pro prefix a druhý pro postfix hledaného cíle. Nejdříve generujeme alternativy prefixu a hledáme takovou cestu, kterou když spojíme s cílem, je sémanticky dosažitelná. Hledání lze dále rozdělit na hledání sémanticky dosažitelného samotného prefixu, poté spojeného s cílem a změnu prefixu, pokud spojení neprodukuje sémanticky dosažitelnou cestu. Po nalezené první části se snažíme generovat alternativy postfixu a opět najít cestu, jejíž spojení s již nalezeným prefixem a cílem, produkuje dosažitelnou úplnou cestu. Nejsme-li schopni nalézt takový postfix, změníme prefix a proces opakujeme. Je nutné nastavit limit délky maximální délky generované cesty, protože jinak algoritmus nemusí nikdy skončit, nebo můžeme produkovat tak velké cesty, že jejich ověření neúměrně velké množství času.

Hledání alternativ cesty s dvěma a více cíli

V tomto případě navazujeme na algoritmus pro generování dlouhých cest 3.2, který spojuje vícero cílů. Můžeme navázat na předchozí definice prefixu, který více omezíme. Mějme cestu p složenou z posloupnosti cílů $T = [t_1, t_2, ..., t_n]$. Prefix prvního cíle t_1 a postfix posledního

cíle t_n odpovídají definicím prefixu a postfixu z předchozí kapitoly. Pro ostatní cíle je jeho prefix t nejdelší cesta začínající v koncovém uzlu cíle t_{i-1} a končící v prvním uzlu cíle t_i, pro $1 < i \leq n$.

Vytvoříme generátory alternativ pro prefix každého cíle a postfix posledního cíle. Stejně jako v předchozí kapitole generujeme různé alternativy od začátku cesty, spojujeme s cíli až se dostaneme do cesty včetně postfixu. Nejsme-li schopni nalézt sémanticky dosažitelnou cestu obsahující všechny cíle, odebereme jeden z cílů a celý proces opakujeme.

3.7 Webové rozhraní

K vytvořenému nástroji pro generování cest by měl být k dispozici způsob, jak jej napojit do platformy Testos, po splnění požadavku REQ.9. Výhodami webového rozhraní je nezávislost na programovacím jazyce a možnost spuštění nástroje samostatně na jiném stroji se vzdáleným přístupem. Rozhraní bude využívat architekturu REST. Webová služba a její rozhraní pro obsluhu vytvořeného nástroje má 2 hlavní úkoly:

1. obsluhu zdrojových kódů programů, nad kterými chceme nástroj spouštět
2. správu procesů nástroje a jejich výsledků

Definici rozhraní a význam jednotlivých požadavků popisuje tabulka 3.3. Tabulka dokumentuje API webové služby a splňuje požadavek REQ.14.

Cesta	HTTP metoda	Popis
/api/llvm-files	GET	stažení seznamu dostupných LLVM IR souborů
/api/llvm-files/<soubor>	GET	stažení souboru s názvem <soubor>
/api/llvm-files/<soubor>	POST	nahrání souboru s názvem <soubor>
/api/llvm-files/<soubor>	DELETE	smazání souboru s názvem <soubor>
/api/jobs	GET	stažení informací o běžících i dokončených úlohách nástroje
/api/jobs	POST	vytvoření a spuštění nové úlohy nástroje
/api/jobs/<job-id>	GET	stažení výsledku úlohy s ID <job-id>
/api/jobs/<job-id>	DELETE	odebrání úlohy s ID <job-id>

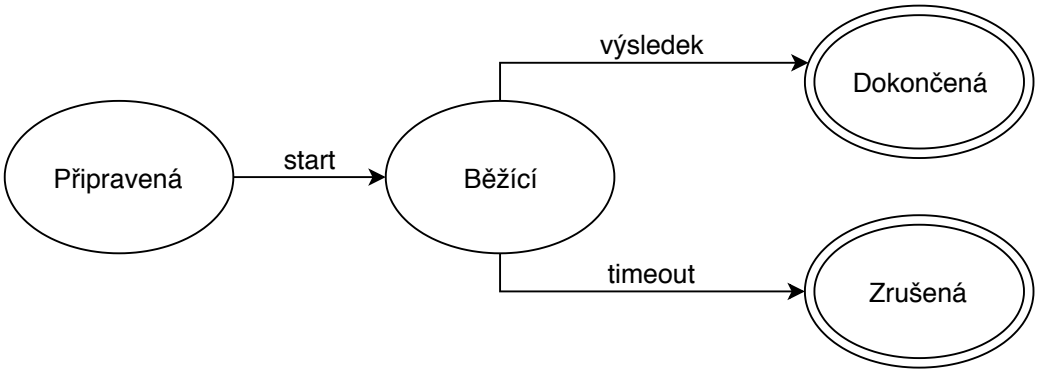
Tabulka 3.3: Definice REST API webové služby.

Správa souborů LLVM IR je přímočará. Uživatel chce soubor nahrát, smazat, stáhnout nebo získat informaci o všech dostupných souborech. Soubory se identifikují pomocí názvu souboru, tudíž nahrání souboru s názvem, který již existuje, vede k přepsání starého souboru nově nahrávaným. Správa procesů je blíže popsána v následující podkapitole.

Obsluha úloh generátoru cest

Dle API z tabulky 3.3 lze vyčíst, že uživatel webové služby dá požadavek na vytvoření úlohy o spuštění generátoru a jako odpověď dostane ID úlohy. Služba neposílá informace o skončení úlohy, takže je úkolem uživatele si zjistit, v jakém je úloha stavu. Buď se může periodicky dotazovat, nebo poslat 1 požadavek po vypršení maximální doby běhu úlohy. Je nutné definovat maximální délku běhu úlohy, aby webová služba mohla zajistit, že každý

proces se spuštěným generátorem vždy skončí a nezahltí tak serverové prostředky. Stavy úlohy jsou znázorněny na grafu 3.6.



Obrázek 3.6: Stavový automat popisující stavy úloh webové služby obsluhující generátor cest.

Jak se budou informace o spuštěných procesech ukládat je implementační detail popsaný v kapitole 4.

Kapitola 4

Implementace generátoru a webového rozhraní

Nadcházející kapitoly popisují implementaci generátoru cest pro testy, nazvaného **COBAP (COverage BAsed Paths) Generator**. Nejdříve jsou krátce popsány použité technologie a knihovny a objasnění důvodu jejich použití. V dalších kapitolách je navázáno na algoirtmy a protokoly navržené v kapitole 3 popisem implementace nástroje pro generování cest ze zdrojových kódů. Popis implementace nástroje je rozdělen do dvou částí, implementace generátoru *cílů* pro splnění kritéria pokrytí a implementace generátoru cest obsahujících tyto cíle. Poslední podkapitola obsahuje popis REST služba a webového rozhraní, pro spuštění nástroje COBAP v rámci platformy Testos.

4.1 Technologie a knihovny

Podkapitoly se věnují popisu a odůvodnění použitých knihoven a technologií.

LLVM

Forma reprezentace zdrojového kódu překladačem LLVM byla popsána v kapitole 2.2. Použití knihovny a nástrojů LLVM je aktuálně nejsnazší způsob, jak využít schopnosti překladače pro analýzu zdrojových souborů pro jazyky C, C++, Rust a další. Z knihovny LLVM byly použity datové typy a algoritmy ze skupin:

- *IR* - nástroje pro procházení a analýzu instrukcí LLVM IR
- *Graph, CFG* - navigace ve grafové struktuře základních bloků zdrojového kódu a vyhledávání SCC
- *Debug Info Metadata* - získání informací z debugových informací přiložených při překladu
- *Optional* - typově bezpečný wrapper pro objekt, který nemusí obsahovat požadovanou hodnotu (známý také jako typ *Maybe* z jiných programovacích jazyků), je součástí standardu C++17
- *CommandLine* - knihovna pro definici typů a zpracování argumentů příkazové řádky programů [14]

COBAP byl vyvíjen a testován s LLVM ve verzi 7.0.

C++14

Použití jazyka C++ je nejjednodušší způsob jak využít API knihoven LLVM. Jeho další výhodou je rychlost exekuce. C++ je kompilovaný do binární podoby a díky dlouholetému vývoji optimalizací kompilátorů patří programy vyvýjené v C++ mezi nejrychlejší. Analýza zdrojových kódů je obecně náročná operace na výpočetní výkon, a proto je na místě využít rychlý, kompilovaný jazyk.

C++14 přináší opravy a doplnění mnohem zásadnějšího standardu C++11. Použitím jazyka C++14 je splněn požadavek REQ.2. Překlad LLVM 7.0 vyžaduje minimálně C++11, ale dle mailových konverzací [4] bude u vývojářů snaha pro příští verze použít konstrukce novějších standardů. V nástroji COBAP bylo z C++14 využito:

- *member aggregate initialization* [10]
- *make_unique* - vytvoření objektů na haldě s automatickým uvolněním paměti po zániku objektu vlastníka

Pro potencionální navazující práce na nástroji by se mělo uvažovat o přechodu na standard C++17. Ve stávající implementaci jsou místa, kde by tento nový standard zvýšil přehlednost kódu, například pomocí *structured bindings* [11].

Generátor testovacích vstupů

TINDGER - Test INput Data GEneratoR je nástroj vyvíjený na platformě Testos a slouží pro sémantické ověření dosažitelnosti cesty programem (repositář projektu je k dispozici na privátním školním serveru VUT). Pokud je cesta sémanticky dosažitelná, nástroj vygeneruje testovací vstupy tak, aby program spuštěný s těmito vstupy přesně prošel zvolenou cestou.

TINDGER je vyvíjen paralelně k této práci a je důležitým prvkem pro dosažení výsledků použitelných nad reálnými programy. Bez jeho přítomnosti (pokud není nahrazen jiným nástrojem produkujícím stejné výsledky) lze získat z našeho generátoru pouze syntakticky dosažitelné cesty, které nemusí být reálně dosažitelné. Diplomová práce k nástroji TINDGER [23] bude dostupná v průběhu roku 2019.

Kromě ověření sémantické dosažitelnosti obsahuje projekt TINDGER také implementaci pojmenování základních bloků a instrukcí LLVM IR. Soubor s LLVM IR vygenerovaný pomocí překladače Clang obsahuje očíslování bloků a instrukcí, ale když se poté načte do paměti skrze knihovny LLVM, jsou tyto informace ztraceny. Proto byl po společné domluvě implementován algoritmus v rámci projektu TINDGER, který kopíruje algoritmus pojmenování bloků a instrukcí z překladače Clang/LLVM. Nástroj COBAP při překladu referencuje tento algoritmus. Tato jména bloků a instrukcí byla sice později odstraněna z výstupu pro komunikaci mezi nástroji COBAP a TINDGER, ale jsou použity pro konečný výstup z nástroje COBAP pro snazší lidsky čitelnou kontrolu výsledků. Implementace se nachází v projektu TINDGER v adresáři `src/llvmextractor` a souborech `BasicBlockLabeler.h` a `BasicBlockLabeler.cpp`.

JSON knihovna a výstup

JSON je použit jako výstupní formát jak pro výsledek nástroje COBAP, tak pro komunikaci mezi nástroji COBAP a TINDGER. Pro serializaci a deserializaci byla vybrána knihovna s titulem *JSON for Modern C++* od Nielse Lohmanna [17]. Mezi její výhody patří:

- Snadná definice požadovaného výstupu - syntaxe definic JSON objektů je velice podobná výstupnímu JSON formátu
- Jednoduchost přidání do projektu - úplná implementace knihovny je dostupná jako jediný hlavičkový soubor
- Permisivní licence - projekt je distribuován pod licencí MIT

Jedinou nevýhodou je průměrná rychlost serializace a deserialiazace, jak ukazují nezávislé testy [25]. Očekává se, že výstup a načítání JSON objektů není úzké hrdlo nástroje a není obětovat nutné jednoduchost a čitelnost za minimální zisk v rychlosti nástroje.

Python a Flask

Python je interpretovaný programovací jazyk, který nachází široké využití. Důvodem je jeho univerzálnost a skutečnost, že kombinuje procedurální, funkční a objektově orientovaná programovací paradigmata. Jeho interpreter je dostupný na mnoha platformách a často i předinstalovaný na mnoha linuxových distribucích. Z těchto důvodů byl vybrán jako jazyk pro implementaci jednoduché aplikace a API pro integraci do platformy Testos.

Flask je framework pro tvorbu webů a webových rozhraní vyvýjen v jazyce Python. Jeho filozofií je poskytnout velice jednoduchý způsob jak rychle vytvořit webovou službu na základech malého webového jádra. Byl použit ve verzi 1.0.2 s Pythonem 3.6.

GNU Timeout

GNU Timeout je nástroj pro spuštění jiných programů s omezením časové délky běhu. Je součástí balíku *GNU Coreutils* a tedy přítomen na všech linuxových distribucích. Je použit při spouštění nástroje TINDGER pro případy, kdy by sémantické ověření trvalo příliš dlouho a pro spuštění nástroje COBAP z webové služby, aby nedošlo k zahlcení serveru. Výhodou je, že lze jednoduše spůstit z obou programovacích jazyků. Malou nevýhodou je nutná přítomnost GNU Coreutils na spouštěné platformě, který je jednoduše přístupný přes *Homebrew* na Mac OS a přes *Windows Subsystem for Linux* na platformách Winodws 10 a Windows Server 2019. Díky tomu, že je GNU Timeout poměrně široce dostupný, byl vybrán namísto nutnosti implementace manuální správy spouštěných externích procesů v obou jazycích C++ a Python.

4.2 Argumenty spuštění nástroje COBAP

Konfigurace spuštění nástroje COBAP se řídí pomocí těchto přepínačů:

- `-h`, `--help` - výpis nápovědy a ukončení programu
- `-c`, `--coverage` - výběr kritéria pokrytí
- `-s`, `--set-size` - výběr velikosti vygenerované sady cest
- `-m`, `--max-length` - maximální délka každé z vygenerovaných cest
- `-o`, `--output` - název výstupního souboru
- `-w`, `--workspace` - cesta ke složce pro uložení dočasných souborů

Argumenty jsou po dvojicích, protože lze použít buď krátkou (`-c`) nebo dlouhou (`--coverage`) formu zápisu.

`big` je očekávaná hodnota pro argument `--set-size`, pokud chce uživatel vygenerovat velkou testovací sadu s krátkými cestami. V opačném případě nastaví hodnotu argumentu na `small`. Nastavení maximální délky cesty se udává celočíselnou hodnotou, která značí počet základních bloků, které generovanou cestu tvoří. Tato nastavení splňují požadavky REQ.7 a REQ.8.

Nastavení *--workspace* se používá, pokud chce uživatel přesměrovat výstup souborů používaných jako vstup do nástroje TINDGER. Toto nastavení bylo přidáno hlavně pro oddělení vytvářených souborů v implementaci webového rozhraní nástroje.

Argument *--coverage* očekává jednu ze zkratek kritéria pokrytí: NC, EC, EPC, PPC, ADC, AUC, ADUPC, PC, CC, MCDC, jejichž definice je popsána v kapitole 2.1. Tento přepínač je povinný. Volba tohoto nastavení splňuje požadavek REQ.4 a implementace všech vypsaných kritérií pokrytí splňuje požadavek REQ.6.

Další povinný argument je cesta k souboru LLVM IR. Tento argument se udává bez přepínače. Příklad spuštění generátoru nad souborem snippet1.ll s podmínkou splnění kritéria MCDC a vygenerování velké sady cest:

```
./cobap -c MCDC -s big snippet1.ll
```

4.3 Generátor cílů

Byla vytvořena velice jednoduchá implementace rozhraní generátoru. V konstruktoru je předán objekt analyzované funkce (`llvm::function`) a generátor nabízí jednu veřejnou metodu `GetTargets` vracející objekt `CoverageTargets` obsahující hledané cíle. Generátor pro jednotlivé kritérium pokrytí je vlastní třída dědící z tohoto rozhraní implementovaného ve třídě `CoverageCriteriaBase`. Takto jednoduché rozhraní přináší zapouzdření veškeré funkcionality do jednotlivých tříd generátorů, snadné předání dat k dalším částem nástroje (výsledek `GetTargets()` je předán na vstup generátoru cest) a snadnou rozšiřitelnost, pokud by byl požadavek na podporu dalších kritérií pokrytí či jiných způsobů generování cílů. Také lze takto jednoduše vygenerovat více objektů `CoverageTargets` současně a ty předat několika generátorům, pro vytvoření rovnou několika sad cest pro několik různých pokrytí.

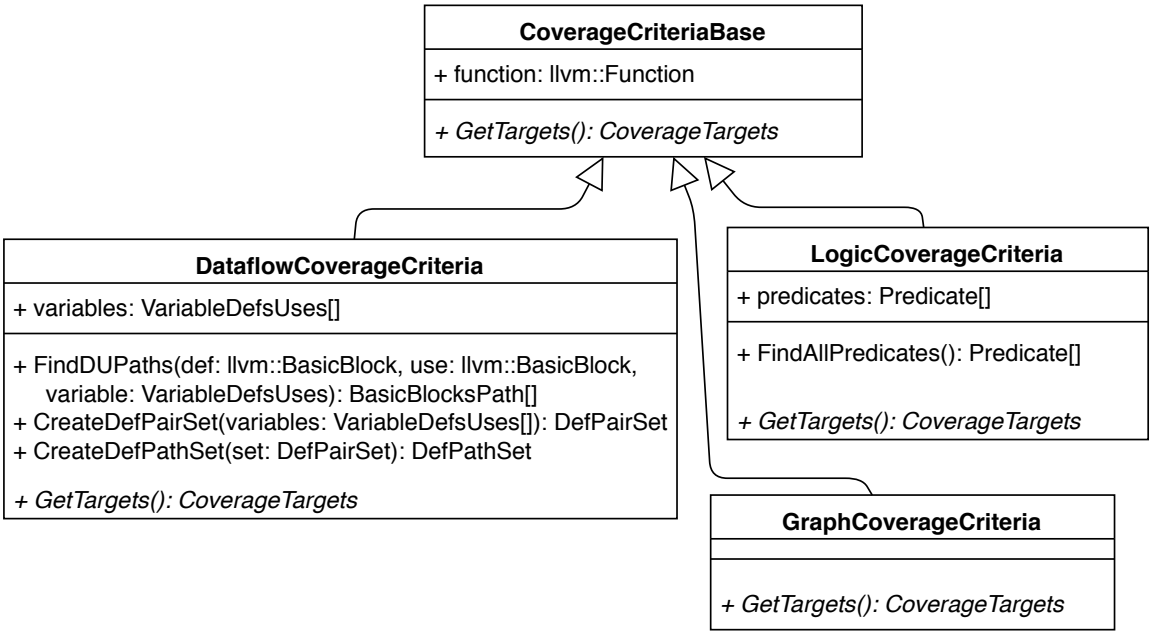
Obrázek 4.1 ukazuje, že byly vytvořeny další 3 rozhraní dědící z `CoverageCriteriaBase`, každé pro jednu oblast kritéria pokrytí. Do nich se zapouzdřila společná funkcionalita.

Cíle pro kritéria pokrytí grafů

Oproti zbylým oblastem kritérií pokrytí byly nejjednodušší na implementaci, protože pracuje pouze s grafem CFG bez zkoumání samotných instrukcí, což dělá implementaci generátoru velice přímočarou. Rodičovská třída `GraphCoverageCriteria` neobsahuje žádnou společnou funkcionalitu 4.2, pouze typově odděluje skupinu kritérií pokrytí zaměřenou na analýzu grafu. Třídy pro jednotlivé kritéria pokrytí grafu jsou na diagramu 4.2.

Cíle pro kritéria pokrytí datových toků

Datové toky sledují operace s proměnnými, proto je v rodičovské třídě `DataflowCoverageCriteria` implementována funkcionalita nalezení proměnných a nalezení všech jejich definic a užití. Konkrétní kritéria pokrytí pak mohou vytvořit množinu *def-path* nebo *def-path* (popsány v kapitole 2.1), podle toho, ze které vybírají datové toky pro vytvoření cílů. Z definic pokrytí



Obrázek 4.1: Diagram tříd základních rohraní pro generátory cílů dle kritéria pokrytí.

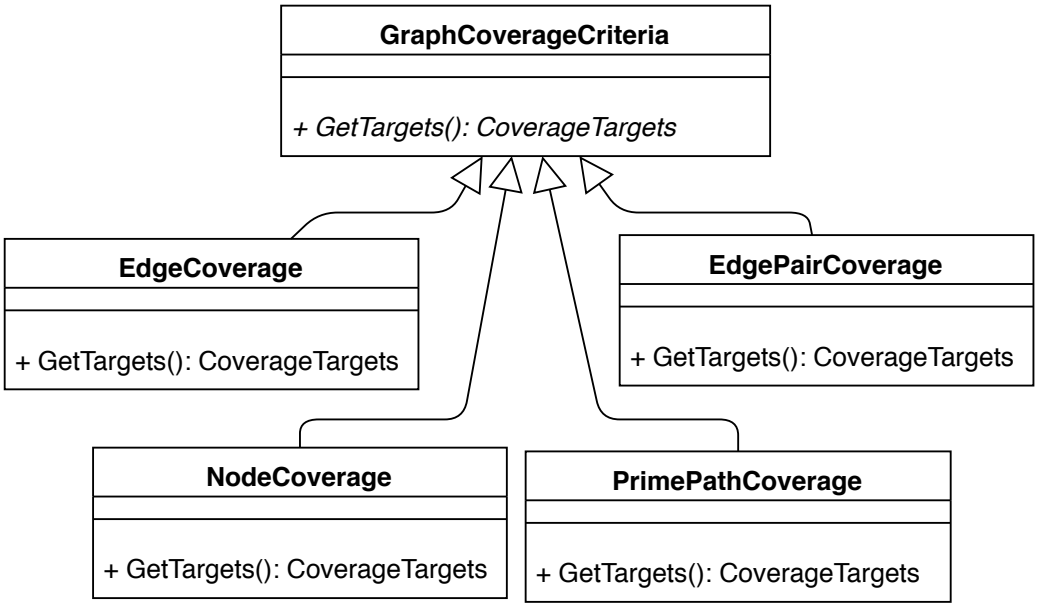
ADC a AUC stačí pro každou proměnnou, definici a v případě AUC pro každé užití přidat do množiny cílů jednu cestu *du-path*. V implementaci jsou cesty du-path přidány všechny a sloučeny do skupin, ze kterých si pak generátor může vybrat jednu a pokud je sémanticky dosažitelná, prohlásit celou skupinu za splněnou. Diagram tříd 4.3 popisuje vytvořené třídy a metody pro společnou funkcionalitu.

Pokud uživatel vybere kritérium pokrytí z oblasti pokrytí datových toků, musí být na vstupu LLVM IR soubor obsahující ladící symboly. Bez nich nelze získat požadované informace o tom, který operand LLVM IR instrukcí je pro uložení mezivýsledku, a který naopak odpovídá uživatelem definované proměnné.

Cíle pro kritéria pokrytí logických výrazů

Diagram 4.4 naznačuje, že společná třída `LogicCoverageCriteria` vytváří pole objektů nazvaných `Predicate`. Třída `Predicate` odpovídá predikátům a je složen ze stromu klauzulí (třída `Clause`), definovaných v kapitole 2.1. Objekt `Clause` obsahuje pouze odpovídající základní blok a ukazatele na další klauzuli/základní blok v sérii podmínek. Protože jeho obsah není tak důležitý, nebyl zahrnut do diagramu 4.4. Diagram ale naznačuje metody pro objekt `Predicate`, který umožňuje mimo jiné také vyhledávat cesty složené výhradně ze základních bloků náležících danému predikátu, tedy ze základních bloků klauzulí a navazujících základních bloků dle výsledku *true* nebo *false* celého predikátu.

Stejně jako u datových toků, je nutné pro kritéria pokrytí logických výrazů překládat zdrojové soubory včetně ladících symbolů, aby bylo možné složit dohromady všechny klauzule daného predikátu.



Obrázek 4.2: Diagramy tříd pro jednotlivé generátory cílů pro kritéria pokrytí grafu.

4.4 Generátor cest

Generátor cest byl opět navržen s jednoduchým rozhraním. V konstruktoru dostane celočíselnou hodnotu nastavení maximální délky cesty, funkci, pro kterou cesty generuje a objekt verifikátoru cest 4.4. Dále obsahuje jen dvě metody, jednu pro generování krátkých a jednu pro generování dlouhých cest. Obě metody berou jako argument objekt s hledanými cíli, prakticky tedy výstup z generátoru cílů. Návratovou hodnotou metod je seznam cest. V projektu COBAP je generátor implementován ve třídě `PathGenerator`.

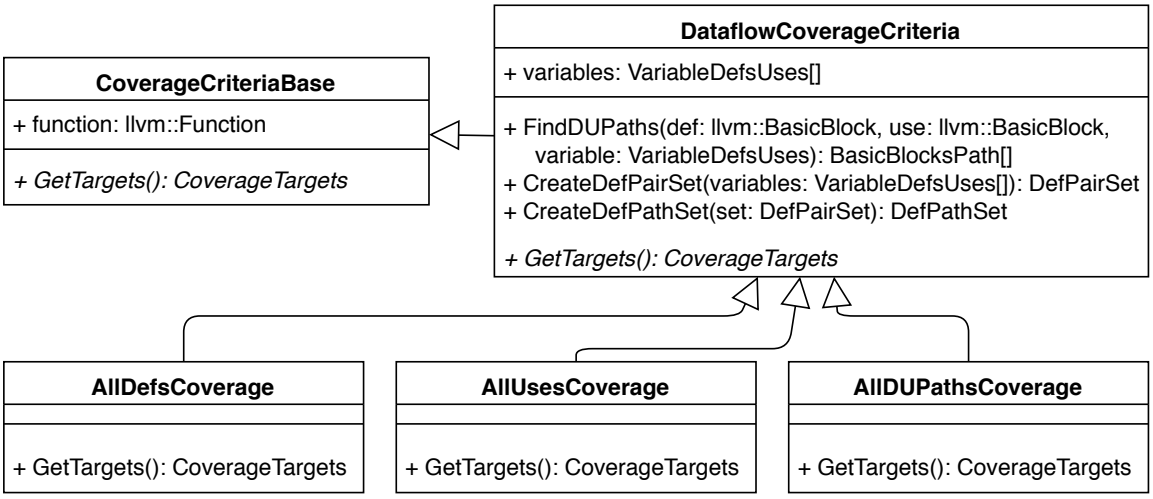
Metoda `GenerateShortPaths` implementuje algoritmus 3.1 pro hledání krátkých cest a metoda `GenerateLongPaths` algoritmus 3.2 pro hledání dlouhých cest. Obě kombinují jejich odpovídající algoritmus s dotazem na sémantickou platnost cesty pomocí verifikátoru. Pokud je cesta sémanticky dosažitelná přidá se do výsledku, do seznamu vygenerovaných cest. Pokud není, postupně se rozšiřuje dle způsobu popsaném v kapitole 3.6. Rozšiřování se děje pomocí algoritmu BFS, implementovaném v třídě `PathAlternativesIterator`. Její implementace je popsána v kapitole 4.4.

Verifikátor cest

Verifikátor je objekt s veřejnou metodou `IsFeasible`, která pro argument cesty programem vrací hodnotu typu boolean. Výsledkem je *true*, pokud byl verifikátor schopen ověřit, že cesta je sémanticky dosažitelná. *False* v opačném případě.

V nástroji byl implementován jediný verifikátor, který pro ověření sémantické dosažitelnosti dané cesty volá nástroj TINDGER. Pro vzájemnou komunikaci jsou důležité 3 soubory:

1. analyzovaný program v LLVM IR, vstup obou nástrojů
2. soubor s cestou k ověření ve formátu JSON, vstup nástroje TINDGER
3. výsledek nástroje TINDGER, ve formátu JSON



Obrázek 4.3: Diagramy tříd pro jednotlivé generátory cílů pro kritéria pokrytí datových toků.

Cesta je sémanticky dosažitelná, pokud výsledný JSON nástroje TINDGER, neobsahuje element `error`, nebo je jeho hodnota prázdná. TINDGER produkuje navíc soubor s logovacími informacemi a soubor s SMT formulí, která reprezentuje dotaz na sémantickou dosažitelnost. Oba jsou nepodstatné pro účely našeho generátoru, mohou sloužit pouze pro analýzu komunikace a chyb v implementaci.

Verifikátor používá GNU Timeout 4.1 pro spuštění nástroje TINDGER, s nastavenou maximální dobou běhu na 1 minutu.

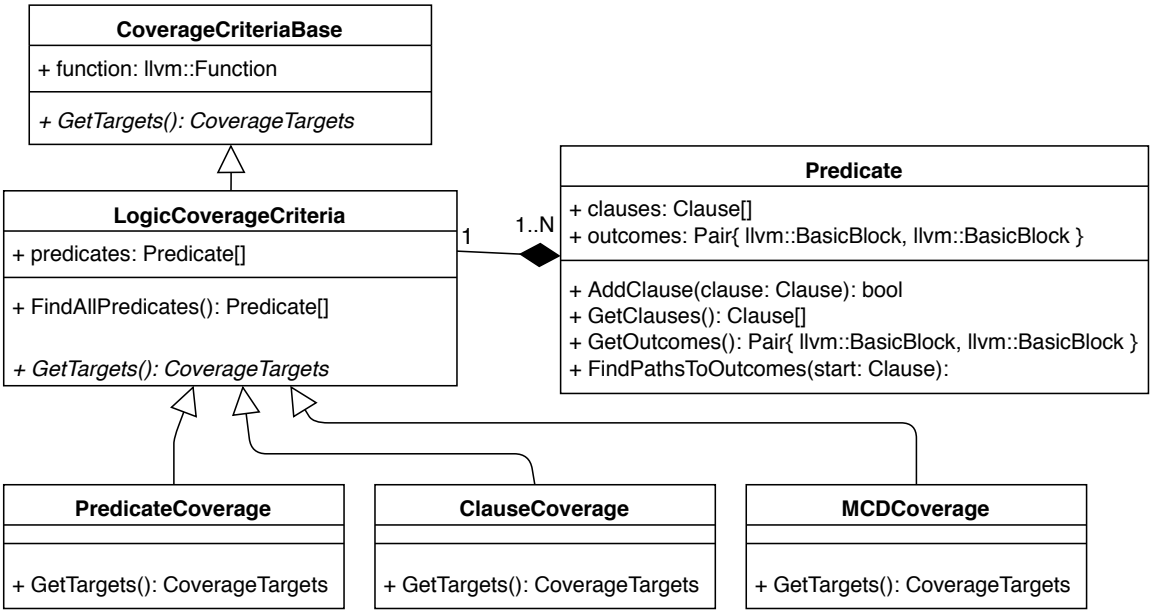
Hledání alternativ cesty

V nástroji potřebujeme rozšiřovat cestu, která není sémanticky dosažitelná. Za tímto účelem vznikla třída `PathAlternativesIterator`, která dostane v konstruktoru dva pevné body – základní bloky označující začátek a cíl cesty. Je požadováno, aby třída našla potencionálně nekonečnou množinu cest, mezi těmito dvěma body. `PathAlternativesIterator` uvnitř implementuje variantu BFS, takže pokud mezi startem a cílem existuje syntaktická cesta, algoritmus ji nalezne.

Třída byla implementována jako iterátor, což je koncept velice využívaný v jazyce C++, a v určité formě i v jiných jazycích. Uvnitř si uchovává seznam alternativ hledané cesty, které ještě nebyly uživateli třídy poskytnuty. Protože je jádrem algoritmu algoritmus BFS, je každá následující poskytnuta cesta minimálně tak dlouhá, jako ta předchozí. Uživateli objektu třídy jsou poskytnuty 3 operátory:

1. operátor `++`, pro přesun na další alternativu cesty
2. operátor `->`, pro přístup k datům současné alternativy
3. operátor `bool()`, pro ověření, že je k dispozici validní cesta

Uživatel generuje cesty mezi 2 poskytnutými základními bloky pomocí operátoru `++`, operátorem `bool()` zjistí, že cesta byla nalezena a pokud ano, přistoupí k jejím datum přes operátor `->`.



Obrázek 4.4: Diagramy tříd pro jednotlivé generátory cílů pro kritéria pokrytí logických výrazů.

4.5 Webové rozhraní

Tato kapitola popisuje implementaci webového API navrženého v kapitole 3.7 a jednoduché uživatelské aplikace.

Spuštění služby s webovým rozhraním

Webová služba byla implementována v jazyce python a frameworku Flask 4.1. Implementace se nachází v souboru `app.py` v adresáři `api`. Spouští se z terminálu v adresáři `api` pomocí příkazů:

```
export FLASK_APP=app.py
flask run
```

Pokud není flask nakonfigurován jinak, spustí tyto příkazy službu na lokální síti s adresou `http://localhost:5000`. Na této adrese je ve webovém prohlížeči k dispozici uživatelské rozhraní, nebo lze posílat požadavky na REST API na URL adresu s prefixem cesty k požadovanému rozhraní 4.5 nebo 4.5.

Správa LLVM IR souborů

Soubory LLVM IR, které uživatel nahrál pro analýzu, jsou uloženy ve složce `llvm-files`. Používá se přístup čistě pomocí souborového systému. Pokud uživatel nahraje soubor se stejným názvem, dojde k přepsání již existujícího souboru.

Po spuštění úlohy generátoru cest, dojde k překopírování souboru LLVM IR do složky se spuštěnou úlohou. To proto, že může v průběhu běhu nástroje dojít ke smazání nebo přepsání LLVM IR souboru.

REST API pro LLVM IR soubory

Cesta v URL adrese pro posílání požadavků na akce s LLVM soubory byla nastavena na:

`<adresa-serveru>/api/llvm-files`

Implementace podporuje dvojí způsob předání dat v požadavku. Buď se zpracuje textové tělo ve formátu JSON, nebo jsou data předána jako data formuláře.

Požadavek s metodou HTTP DELETE vyžaduje klíč `file-name` zadaném v těle JSON nebo ve formulářových datech, nesoucí název souboru určeného ke smazání.

Požadavek s HTTP metodou POST vyžaduje buď JSON tělo s klíči `file-name` (název souboru) a `file-data` (obsah souboru v kódování base64), nebo formulářový klíč `llvm-file` obsahující posílaný soubor.

Správa úloh generátoru

Každé úloze je při jejím vzniku přiřazen identifikátor ve formátu UUID. Je vytvořena podsložka ve složce `jobs`, nesoucí jednoznačný identifikátor vytvářené úlohy. Do složky úlohy je nakopírován soubor LLVM IR, který chce uživatel analyzován. Dále je v ní vytvořen soubor `job-info.txt`, obsahující metainformace o prováděné úloze, zejména datum a času jejího vytvoření. Poté dojde ke spuštění samotného generátoru.

Úloha je ve stavu *finished* (dokončená), pokud složka s úlohou obsahuje výsledek generátoru, který je zapsán do souboru `paths.json`. Pokud daný soubor není přítomen, zkontroluje se čas vytvoření úlohy v souboru s metainformacemi. Pokud je čas delší než 10 minut, je stav úlohy *cancelled (zrušena)*, v opačném případě je úloha ve stavu *running (spuštěna)*. Stavy odpovídají navrženému životnímu cyklu úloh dle automatu na diagramu 3.6.

Spuštění generátoru

Předpoklad pro vytvoření úloh generátoru je přítomnost přeložených nástrojů COBAP a TINDGER v adresáři, kde se nachází skript s webovou službou. V našem projektu je webová služba implementována v jazyce Python a frameworku Flask, zapsána v souboru `app.py` v adresáři `api`. Po přijetí validního dotazu se spustí nástroj COBAP skrze GNU Timeout 4.1 takto:

```
timeout 10m ./cobap -c
```

Pro vytvoření procesu je použita metoda `Popen` z balíku subprocess.

REST API pro úlohy spuštění generátoru

REST API pro přístup k akcím s LLVM IR soubory byla zvolena:

Cesta v URL adrese pro posílání požadavků na akce s LLVM soubory byla nastavena na:

`<adresa-serveru>/api/jobs`

kde `<adresa-serveru>` je URL adresa serveru, například `localhost:5000` při lokálním spuštění pomocí `flask run`. Implementace podporuje dvojí způsob předání dat v požadavku. Buď se zpracuje textové tělo ve formátu JSON, nebo jsou data předána jako data formuláře.

Požadavek s metodou HTTP DELETE vyžaduje klíč `file-name` zadaném v těle JSON nebo ve formulářových datech, nesoucí název souboru určeného ke smazání.

Požadavek s HTTP metodou POST vyžaduje buď JSON tělo s klíči `file-name` (název souboru) a `file-data` (obsah souboru v kódování base64), nebo formulářový klíč `llvm-file` obsahující posílaný soubor.

4.6 Výstupy nástroje

Následující podkapitoly popisují formát a obsah výstupů, které aplikace produkuje. Všechny výstupy jsou v textovém formátu.

Výsledek generátoru

Výstup generátoru je uložen do souboru ve formátu JSON. JSON obsahuje jen nejdůležitější informace: jméno funkce a seznam vygenerovaných cest. U každé cesty je možné vložit dodatečné informace pro uživatele do elementu metadata. Formát je shodný jak pro konečný výstup generátoru, tak pro dotazy při komunikaci do nástroje TINDGER. Příklad výstupu je ve výpisu 4.1.

```
{
  "function": {
    "name": "TestAlternatives",
    "paths": [
      {
        "metadata": [],
        "path": [
          0,
          1,
          11,
          12,
          15
        ],
        "tag": "long path"
      }
    ]
  }
}
```

Výpis 4.1: Příklad výsledku generátoru.

Log s dodatečnými informacemi

Přes všechny fáze generátoru jsou do souboru `cobap.log` ukládány dodatečné informace, usnadňující diagnostiku nástroje. U kritéria pokrytí datových toků jsou vypsány informace o nalezených proměnných a místech definic a použití proměnných. U kritéria pokrytí logických výrazů jsou vypsány informace o nalezených predikátech a klauzulí, z kterých se skládají.

Pro jednotnou formu logovacích zpráv, byla převzána a upravena implementace logovací knihovny z blogu Dr.Dobb's [19]. Byla vybrána tato implementace, protože knihovan je jediný hlavičkový soubor s pouhými 230 řádky kódu. Byly definovány 4 úrovně významu

logované iformace: varování (warn), chybová (error), informační (info) a ladící (debug). Bylo upraveno API knihovny tak, aby usnadnilo výpis dle určité úrovně. Příklad ukládaných dat:

```
- 07:48:16.635 INFO: Local variables:
variable name:' a', function parameter, definitions: 4, uses: 7
variable name:' i', local variable, definitions: 2, uses: 4
variable name:' k', local variable, definitions: 2, uses: 3
variable name:' j', local variable, definitions: 2, uses: 3
```

Kapitola 5

Demonstrační příklad

V této kapitole je demonstrována funkčnost implementovaného nástroje na praktickém příkladě. Nejprve je vytvořen umělý příklad zdrojového kódu, na kterém bude nástroj demonstrován. V další kapitole 5.2 je zkoumáno generování cest pro splnění kritéria pokrytí, tzv. cílů. Další kapitola 5.3 je demonstrováno generování výsledných cest, včetně ověření sémantické dosažitelnosti. Na závěr je uveden příklad, pro generátor není v rozumném čase nalézt sémanticky dosažitelnou cestu.

5.1 Příklad zdrojového kódu

Vytvořme si příklad zdrojového kódu tak, aby splňoval tyto podmínky:

1. príklad je relativně malý
2. obsahuje alespoň jeden cyklus a jednu podmínku if-else
3. obsahuje alespoň 3 proměnné a užití proměnných v různých větvích podmínky
4. výraz alespoň jedné podmínky je složen ze 2 a více podmínek spojených logickými operátory

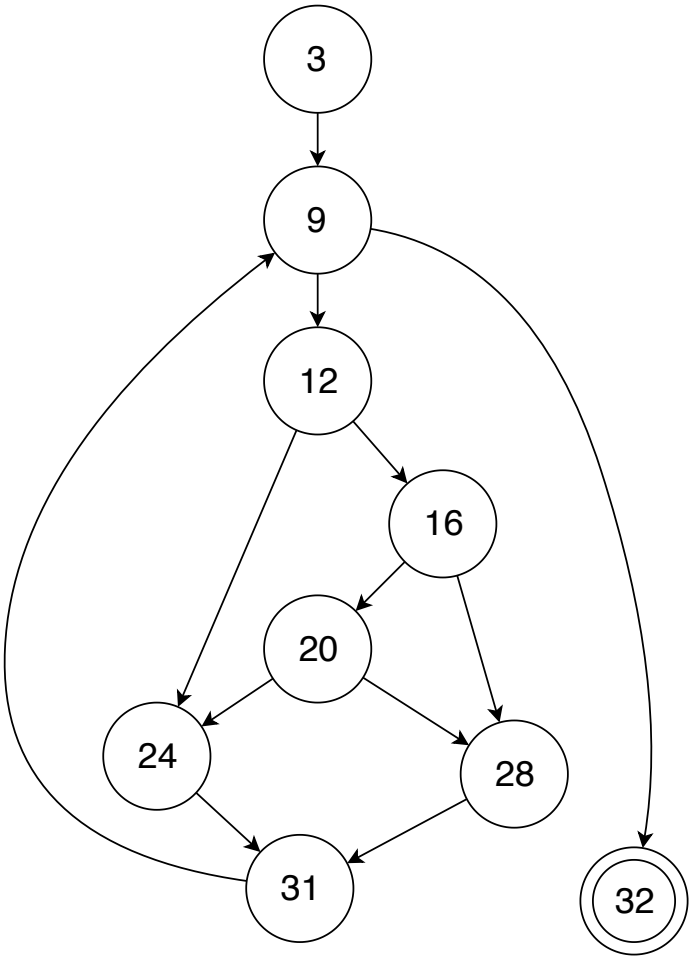
```
1  int foo(int min, int max, int delta)
2  {
3      int result = 0;
4      int counter = 1;
5      while (counter < 100)
6      {
7          if (counter % 2 == 0 || (counter > min && counter < max))
8              result += counter;
9          else
10             counter += 10;
11     }
12
13     return result;
14 }
```

Výpis 5.1: Umělý příklad zdrojového kódu v jazyce C.

Příklad 5.1 je vybraný kandidát, který splňuje všechny podmínky. Pro přeložení do jazyka LLVM IR použijeme překladač Clang. Příklad uložíme do souboru `demo.c` a použijeme příkaz:

`clang -g -S -emit-llvm demo.c`

Clang vygeneruje soubor `demo.ll` obsahující instrukce LLVM IR uspořádané do základních bloků, včetně ladících symbolů. Odpovídající CFG graf je na obrázku 5.1.



Obrázek 5.1: CFG graf odpovídající kódu z příkladu 5.1 přeloženého do LLVM IR.

Uzly grafu na obrázku 5.1 jsou číslovány podle čísla, jaké jim přiřadil Clang při generování LLVM IR. Tato čísla se nepoužívají k identifikaci uzlů v rámci knihoven LLVM, ale jsou v obrázku ponechány pro lepší kontrolu.

5.2 Generování cílů

Kritéria pokrytí grafu

Správnost vygenerování cílů pro pokrytí grafu se kontroluje poměrně snadno. Node Coverage přidává každý uzel, Edge Coverage každou navazující dvojici uzlu a Edge-Pair coverage všechny trojice. U Prime Path Coverage lze ověřit, že každá cest je primární: 1. cesta může obsahovat stejný uzel 2x pouze v případě, že je to začínající a koncový uzel a za 2. žádná primární cesta není podcestou jiné primární cesty. Vygenerované cíle pro příklad 5.1 jsou vypsány v tabulce 5.1.

Kritérium pokrytí	Vygenerované cesty
NC	[3], [9], [12], [16], [20], [24], [28], [31], [32]
EC	[3, 9], [9, 12], [9, 32], [12, 24], [12, 16], [16, 20], [16, 28], [20, 24], [20, 28], [24, 31], [28, 31], [31, 9]
EPC	[3, 9, 12], [3, 9, 32], [9, 12, 24], [9, 12, 16], [12, 24, 31], [12, 16, 20], [12, 16, 28], [16, 20, 24], [16, 20, 28], [16, 28, 31], [20, 24, 31], [20, 28, 31], [24, 31, 9], [28, 31, 9], [31, 9, 12], [31, 9, 32]
PPC	[3, 9, 32], [3, 9, 12, 24, 31], [12, 24, 31, 9, 32], [3, 9, 12, 16, 28, 31], [12, 16, 28, 31, 9, 32], [16, 20, 24, 31, 9, 12], [16, 28, 31, 9, 12, 24], [20, 28, 31, 9, 12, 16], [3, 9, 12, 16, 20, 24, 31], [3, 9, 12, 16, 20, 28, 31], [12, 16, 20, 24, 31, 9, 32], [12, 16, 20, 28, 31, 9, 32], [16, 20, 28, 31, 9, 12, 24], [20, 24, 31, 9, 12, 16, 28], [24, 31, 9, 12, 16, 20, 28], [28, 31, 9, 12, 16, 20, 24]

Tabulka 5.1: Příklad nalezených cílů pro kritéria pokrytí grafů.

Kritéria pokrytí datových toků

Pro ověření správnosti generování cílů si nejdřív vypišme pro každou proměnnou bloky, ve který se definuje a používá do tabulky 5.2:

Proměnná	Uzly definice	Uzly užití
min	3	16
max	3	20
result	3, 24	24, 32
counter	3, 28	12, 16, 20, 24, 28

Tabulka 5.2: Tabulka s definicemi a užitím promměných z demonstračního příkladu.

Výsledky generování cílů jsou v tabulce 5.3. U kritérií ADC a AUC jsou cesty odděleny složenými závorkami, což značí, že jde o množinu cest, z nichž musí alespoň jedna být ve výsledku generátoru cest. Cílů je vygenerováno poměrně dost, to proto že množiny mohou obsahovat redundanci mezi sebou. Ověření jednotlivých cílů je nutné provést ručně, procházením od definic do užití pro každou proměnnou. Automaticky lze pouze ověřit, že AUC a ADUPC generují stejné cesty, a rozdíl je jen v tom, že AUC je shlukuje do množin pro výběr. Vygenerované cíle souhlasí s návrhem a úspěšně pokrývají kritéria pokrytí datových toků.

Kritéria pokrytí logických výrazů

Analyzovaná funkce foo obsahuje 2 predikáty. Jeden je uvnitř cyklu `while` a druhý uvnitř podmínky `if`. *While predikát* obsahuje pouze jedinou klauzuli, která v grafu odpovídá uzlu 9 a dle výsledku predikátu vede tok řízení do uzlu 12 nebo 32. *If predikát* obsahuje 3 klauzule s odpovídajícími uzly 12, 16 a 20. Výsledek *if predikátu* větví řízení do uzlu 24 nebo 28.

Výsledky lze poměrně snadno systematicky ověřit. Kritérium PC je splněno, protože cíle obsahují výsledné uzly všech predikátů. Kritérium CC také, protože cíle obsahují všechny výsledky klauzulí. U MCDC ověříme, že obsahuje takové množiny cest, aby každá klauzule vedla do obou výsledků predikátu, ke kterému patří. Správné výsledky jsou v tabulce 5.4.

Kritérium pokrytí	Vygenerované cesty
ADC	{[3, 9], [3, 9, 12], [3, 9, 12, 24], [3, 9, 12, 16], [3, 9, 12, 16, 20], [3, 9, 12, 16, 20, 24], [3, 9, 12, 16, 20, 28], [3, 9, 12, 16, 28]}, {[3, 9, 12, 16]}, {[3, 9, 12, 16, 20]}, {[3, 9, 12, 24], [3, 9, 12, 16, 20, 24], [3, 9, 32]}, {[24, 31, 9, 12, 24], [24, 31, 9, 12, 16, 20, 24], [24, 31, 9, 32]}, {[28, 31, 9], [28, 31, 9, 12], [28, 31, 9, 12, 24], [28, 31, 9, 12, 16], [28, 31, 9, 12, 16, 20], [28, 31, 9, 12, 16, 20, 24], [28, 31, 9, 12, 16, 20, 28], [28, 31, 9, 12, 16, 28]}
AUC	{[3, 9]}, {[3, 9, 12]}, {[3, 9, 32]}, {[3, 9, 12, 24], [3, 9, 12, 16, 20, 24]}, {[3, 9, 12, 24], [3, 9, 12, 16, 20, 24]}, {[3, 9, 12, 16]}, {[3, 9, 12, 16]}, {[3, 9, 12, 16, 20]}, {[3, 9, 12, 16, 20]}, {[3, 9, 12, 16, 20, 28], [3, 9, 12, 16, 28]}, {[24, 31, 9, 32]}, {[24, 31, 9, 12, 24], [24, 31, 9, 12, 16, 20, 24]}, {[28, 31, 9]}, {[28, 31, 9, 12]}, {[28, 31, 9, 12, 24], [28, 31, 9, 12, 16, 20, 24]}, {[28, 31, 9, 12, 16]}, {[28, 31, 9, 12, 16, 20]}, {[28, 31, 9, 12, 16, 20, 28], [28, 31, 9, 12, 16, 28]}
ADUPC	[3, 9], [3, 9, 12], [3, 9, 32], [3, 9, 12, 24], [3, 9, 12, 16, 20, 24], [3, 9, 12, 24], [3, 9, 12, 16, 20, 24], [3, 9, 12, 16], [3, 9, 12, 16], [3, 9, 12, 16, 20], [3, 9, 12, 16, 20], [3, 9, 12, 16, 20, 28], [3, 9, 12, 16, 28], [24, 31, 9, 32], [24, 31, 9, 12, 24], [24, 31, 9, 12, 16, 20, 24], [28, 31, 9], [28, 31, 9, 12], [28, 31, 9, 12, 24], [28, 31, 9, 12, 16, 20, 24], [28, 31, 9, 12, 16], [28, 31, 9, 12, 16, 20], [28, 31, 9, 12, 16, 20, 28], [28, 31, 9, 12, 16, 28]

Tabulka 5.3: Příklad nalazených cílů pro kritéria pokrytí datových toků.

Kritérium pokrytí	Vygenerované cesty
PC	[24], [28], [12], [32]
CC	[12, 24], [12, 16], [16, 20], [16, 28], [20, 24], [20, 28], [9, 12], [9, 32]
MCDC	[12, 24], [12, 16, 20, 24], [12, 16, 28], [12, 16, 20, 28], [16, 20, 24], [16, 28], [16, 20, 28], [20, 24], [20, 28], [9, 12], [9, 32]

Tabulka 5.4: Příklad nalazených cílů pro kritéria pokrytí datových toků.

5.3 Generování cest

Příklad zdrojového kódu 5.1 je vhodný pro demonstraci hledání cílů, bohužel generuje příliš mnoho sémanticky nedosažitelných cest. Vytvořme nový příklad 5.2, který obsahuje cyklus o dvou iteracích, obsahující podmínku, která se provede v druhé iteraci.

```
1  int foo(int a, int b)
2  {
3      int sum = 1;
4      for (int i = 0; i < 2; ++i)
5      {
6          if (i == 1)
7          {
8              sum += a * b;
9          }
10     }
11
12     return sum;
13 }
```

Výpis 5.2: Umělý příklad zdrojového kódu v jazyce C.

Překladač Clang vyrobí z tohoto příkladu LLVM IR se 7 základními bloky: 2, 7, 10, 13, 19, 20, 23. Základní blok, který obsahuje přičtení operace `sum + a * b` do proměnné `sum`, je blok č. 13. Spuštění generátoru cest s kritériem pokrytí NC produkuje tyto logovací zprávy (upraveno pro kompaktnější zobrazení):

```
- Target is: [%2]
- Trying to verify first found shortest path: [%2 -> %7 -> %23]
- First found path is not feasible. Now trying just its parts.
- Prefix: [%2] itself is feasible
- Trying prefix + target: [%2]
- Feasible prefix found: [%2]
- Current postfix candidate: [%2 -> %7 -> %23]
- Postfix candidate is not feasible.
- Current postfix candidate: [%2 -> %7 -> %10 -> %19 -> %20 -> %7 -> %23]
- Postfix candidate is not feasible.
- Current postfix candidate: [%2 -> %7 -> %10 -> %13 -> %19 -> %20 -> %7
    -> %23]
- Postfix candidate is not feasible.
- Current postfix candidate: [%2 -> %7 -> %10 -> %19 -> %20 -> %7 -> %10
    -> %19 -> %20 -> %7 -> %23]
- Postfix candidate is not feasible.
- Current postfix candidate: [%2 -> %7 -> %10 -> %13 -> %19 -> %20 -> %7
    -> %10 -> %19 -> %20 -> %7 -> %23]
- Postfix candidate is not feasible.
- Current postfix candidate: [%2 -> %7 -> %10 -> %19 -> %20 -> %7 -> %10
    -> %13 -> %19 -> %20 -> %7 -> %23]
- Postfix: [%2 -> %7 -> %10 -> %19 -> %20 -> %7 -> %10 -> %13 -> %19
    -> %20 -> %7 -> %23] itself is feasible
- Trying postfix + target + postfix: [%2 -> %7 -> %10 -> %19 -> %20 -> %7
    -> %10 -> %13 -> %19 -> %20 -> %7 -> %23]
- Moving on with another target.
- Printing JSON of 1 paths for function 'foo'
```

Z výstupu je vidět, že nástroj nejdříve zkusil cestu [2, 7, 23], ale TINDGER mu oznámil, že není splnitelná. Nástroj tedy rozbíjí cestu na prefix a postfix pro nalezení splnění cíle [2]. Prefix je nalezen ihned, protože cesta [2] je prvním blokem funkce. Během hledání postfixu je vyzkoušeno 6 sémanticky nedosažitelných kandidátů. Až 7. kandidát je dosažitelný a posledním krokem je ověření celé cesty [2, 7, 10, 19, 20, 7, 10, 13, 19, 20, 7, 23], která je prohlášena za sémanticky dosažitelnou a je přidána mezi výsledky. Generátor chce přejít na další cíl, ale zjišťuje, že jedinou cestou pokryl všechny cíle, proto vypisuje JSON s 1 cestou na výstup. Výstupem je JSON 5.3.

```
1 {
2   "function": {
3     "name": "foo",
4     "paths": [
```

```
5       {
6         "metadata": [],
7         "path": [0,1,2,4,5,1,2,3,4,5,1,6],
8         "tag": "long path"
9       }
10    ]
11  }
12 }
```

Výpis 5.3: Výsledek generátoru cest. Upraveno formátování pro kompaktnější výpis.

5.4 Limit generátoru sémanticky dosažitelných cest

Dle způsobu, jakým se cesty generují, není příliš náročné nalézt příklady kódů, které budou pro navržený generátor problematické. Jedním takovým příkladem je i funkce `sum1000` ve výpisu 5.4.

```
1  int sum1000(int a)
2  {
3          int sum = 0;
4      for (int i = 0; i < 1000; ++1)
5          sum += a;
6
7      return sum;
8  }
```

Výpis 5.4: Příklad kódu, pro který generátor není schopen nalézt sémanticky dosažitelnou cestu.

Tato velice jednoduchá funkce obsahuje cyklus, který se vždy provede 1000-krát. To je zřejmé na první pohled pr0ogramátorovi, ale ne stroji. Protože nástroj není schopen najít cykly s předem definovaným počtem průchodů, bude se snažit ověřit nejkratší cestu a tu pak postupně rozšiřovat. Celkově by musel požádat nástroj o ověření sémantické dosažitelnosti minimálně 1000-krát, ale dříve dojde na nastavený limit délky cesty. Přitom každý průchod cyklem se bude skládat minimálně ze dvou bloků (pokud neuvažujeme optimalizace). Generátor v tomto případě buď vrátí prázdný výsledek, nebo bude předčasně ukončen na timeout.

Řešení takových případů existují. Řešení zahrnuje rozšíření paměťového modelu pro nástroj TINDGER, který potom pomocí zvoleného řešiče SMT bude hledat nejen splnitelnost zadaných požadavků, ale také možné cesty v grafu [7], tedy rozvinutí smyčky. Toto je ovšem cílem následujícího výzkumu a návrh i řešení je nad rámec této diplomové práce.

Kapitola 6

Závěr

V této diplomové práci byl navržen postup a popsána implementace generátoru testovacích případů na základě technik z testování založeného na modelu. Po teoretickém úvodu, ve kterém je čtenář uveden do problematiky testování založeného na modelech, možností knihovny LLVM a zhodnocení existujících nástrojů zabývajících se touto oblastí, je čtenáři představen návrh nástroje. Hned na začátku návrhu, jsou formálně definovány požadavky, aby v několika bodech popsali cíle tohoto projektu. Návrh představuje jazykově nezávisle algoritmy, které mohou vést k úspěšné implementaci nástroje. Také je navrženo webové rozhraní, pro umožnění napojení generátoru cest na další nástroje, které mohou jeho znalosti převést na tvorbu skutečných testovacích případů na reálných platformách. Ve 4. kapitole je čtenáři popsána úspěšná implementace, na níž navazuje kapitola demonstrující rozsah jeho schopností.

Jedním ze stěžejních přínosů této práce je podpora široké škály kritérií pokrytí. Většina dostupných nástrojů se věnuje jen pokrytí všech řádků kódu a větvení a některé pokročilé nástroje se věnují kritériu pokrytí MCDC, které se stalo průmyslovým standardem. Představený nástroj nabízí nejen tato kritéria, ale i řadu dalších, které sice nejsou příliš rozšířené mezi běžnými vývojáři, mohou ale vést ke zlepšení testovaného softwaru. Dále je uživateli poskytnuta možnost specifikovat, jestli chce vygenerovat sadu cest spíše menší, a ušetřit strojový čas během testování, nebo sadu větší s kratšími cestami u nichž je větší šance lokalizace problému v testovaném softwaru.

Zvolený přístup tvorby cest se také neobešel bez určitých omezení. V 5. kapitole je ukázka triviálního příkladu, jehož varianty se zcela určitě vyskytují v reálném softwaru a implementovaný nástroj není schopen v něm cesty najít. Řešení tohoto problému bylo krátce nastíněno a je vhodným námětem pro navazující studijní práce.

Literatura

[1] Aichernig, B.; Brandl, H.; Jöbstl, E.; aj.: MoMuT::UML Model-Based Mutation Testing for UML. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, April 2015, s. 1–8, doi:10.1109/ICST.2015.7102627.

[2] Ammann, P.; Offutt, J.: *Introduction to software testing*. Cambridge University Press, 2008, ISBN 1139468677.

[3] Anielak, G.; Jakacki, G.; Lasota, S.: Incremental test case generation using bounded model checking: an application to automatic rating. *International Journal on Software Tools for Technology Transfer*, ročník 17, 06 2014, doi:10.1007/s10009-014-0317-2.

[4] Bastien, J. F.: *Migrating past C++11*. [Online; navštíveno 09.03.2019]. URL http://lists.llvm.org/pipermail/llvm-dev/2019-January/129452.html

[5] Brázdil, T.; Chatterjee, K.; Chmelík, M.; aj.: Verification of Markov Decision Processes Using Learning Algorithms. In *Automated Technology for Verification and Analysis*, editace F. Cassez; J.-F. Raskin, Cham: Springer International Publishing, 2014, ISBN 978-3-319-11936-6, s. 98–114.

[6] of California, U.: *Design and Analysis of Algorithms*. [Online; navštíveno 20.05.2019]. URL https://www.ics.uci.edu/~eppstein/161/960220.html#sca

[7] Charvát, L.; Smrčka, A.; Vojnar, T.: An Abstraction of Multi-port Memories with Arbitrary Addressable Units. In *Computer Aided Systems Theory - EUROCAST 2013*, editace R. Moreno-Díaz; F. Pichler; A. Quesada-Arencibia, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 978-3-642-53856-8, s. 460–468.

[8] Cordeiro, L.; Kesseli, P.; Kroening, D.; aj.: JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *30th International Conference on Computer Aided Verification*, 2018, doi:10.1007/978-3-319-96145-3_10.

[9] Ganesh, V.: *Decision Procedures for Bit-vectors, Arrays, and Integers*. Dizertační práce, 09 2007, doi:10.13140/RG.2.1.3591.8245.

[10] cppreference group: *Aggregate initialization*. [Online; navštíveno 14.02.2019]. URL https://en.cppreference.com/w/cpp/language/aggregate_initialization

[11] cppreference group: *Structured binding declaration*. [Online; navštíveno 15.02.2019]. URL https://en.cppreference.com/w/cpp/language/structured_binding

[12] Jacky, J.: PyModel: Model-based testing in Python. Austin, Texas, July 2010.

[13] Lattner, C.; Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[14] LLVM-Project: *CommandLine 2.0 Library Manual*. [Online; navštíveno 10.01.2019].
URL https://www.llvm.org/docs/CommandLine.html

[15] LLVM-Project: *LLVM Language Reference Manual*. [Online; navštíveno 10.01.2019].
URL https://llvm.org/docs/LangRef.html

[16] LLVM-Project: *Tarjan's Algorithm to find Strongly Connected Components*. [Online; navštíveno 20.05.2019].
URL https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/

[17] Lohmann, N.: *JSON for Modern C++*. [Online; navštíveno 10.01.2019].
URL https://github.com/nlohmann/json

[18] Ltd., D.: *Diffblue - Playground*. [Online; navštíveno 15.05.2019].
URL https://playground.diffblue.com

[19] Marginean, P.: *Logging In C++*. [Online; navštíveno 11.05.2019].
URL http://www.drdobbs.com/cpp/logging-in-c/201804215

[20] Offutt, A. J.; Voas, J. M.: *Subsumption of condition coverage techniques by mutation testing*. Technická zpráva, Dept. of Information and Software Systems Eng., George Mason Univ., 1996.

[21] Prause, C.; Werner, J.; Hornig, K.; aj.: *Is 100% Test Coverage a Reasonable Requirement? Lessons Learned from a Space Software Project*. 11 2017, doi:10.1007/978-3-319-69926-4_25.

[22] Roy, S.; Srikant, Y. N.: The Hot Path SSA Form: Extending the Static Single Assignment Form for Speculative Optimizations. In *Compiler Construction*, editace R. Gupta, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN 978-3-642-11970-5, s. 304–323.

[23] Sušovský, T.: *Decision Procedures for Bit-vectors, Arrays, and Integers*. Diplomová práce, Vysoké učení technické v Brně, 06 2019, k nalezení na:.
URL http://www.fit.vutbr.cz/study/DP/DP.php

[24] Vorobyov, K.; Krishnan, P.: Combining Static Analysis and Constraint Solving for Automatic Test Case Generation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, ISSN 2159-4848, s. 915–920, doi:10.1109/ICST.2012.196.

[25] Yip, M.: *C/C++ JSON parser/generator benchmark*. [Online; navštíveno 11.04.2019].
URL https://rawgit.com/miloyip/nativejson-benchmark/master/sample/performance_Corei7-4980HQ@2.80GHz_mac64_clang7.0.html