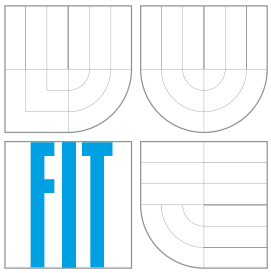
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATICKÁ TVORBA TESTOVACÍCH PŘÍPADŮ Z DATOVÝCH TOKŮ

AUTOMATIC GENERATION OF TEST CASES FROM DATA-FLOW

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DANIEL KRAUT

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2016

Abstrakt

Tato práce se zabývá automatickou tvorbou testovacích případů na úrovni jednotkového testování, pro zdrojové texty v jazyce C. K dosažení automatizace jsou použity metody přístupu k software jako k datovým tokům proměnných. Je zde nastíněn náhled na průběh tvorby testů a funkci CSP solverů, které jsou nezbytnou částí pro rozhodnutelnost těhto problémů. Statická analýza kódu je umožněna knihovnou LibTooling v projektu překladačového front-endu Clang. Jsou uvedeny výstupy z aplikace, jimiž jsou automaticky vytvořené vstupní hodnoty pro testovací případy.

Abstract

This thesis deals with automatic generation of test cases on Unit testing level for source codes in C language. In order to achieve automatization are used methods of approach to software as a Data-flow of variables. There is outlined a process of creating tests here as well as a function of CSP solvers which are necessary part of solving this problems. Static code analysis is accessed with LibTooling libraring as part of a compiler front-end project Clang. Output of a developed application are provided here which is automatically generated input values for test cases.

Klíčová slova

testování, statická analýza, automatizace, Clang, CSP, Gecode, jazyk C

Keywords

testing, static analysis, automatization, Clang, CSP, Gecode, C language

Citace

KRAUT, Daniel. *Automatická tvorba testovacích případů z datových toků*. Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Smrčka Aleš.

Automatická tvorba testovacích případů z datových toků

Prohlášení

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

.....
Daniel Kraut
1. srpna 2016

Poděkování

Rád bych poděkoval svému vedoucímu panu Ing. Aleši Smrčkovi, Ph.D. za inspiraci k zabývání se testováním software, vedení a rady, které mě nasměrovaly k řešení problému a jeho trpělivost.

Obsah

Kapitola 1

Úvod

Testování softwaru má obrovský význam v oblasti vývoje software. Svědčí o tom jak množství prostředků vynaložených společnostmi, k dosažení co nejvyšší úrovně odolnosti systémů proti chybám, tak i události z historie, kdy chyba lidská chyba zavinila ztrátu peněžní či dokonce životní. Testování je rozsáhlá oblast se širokou škálou nabízených nástrojů pro řešení obecných, či konkrétních problémů. Jedním z nich je problém automatizace. Testování, a pokusy o jeho automatizaci, se potýkají s řadou překážek, z nichž některé mohou být i obecně neřešitelné. Různé aplikace a frameworky k těmto problémům přistupují z různých úhlů pohledu a poskytují různé úrovně a oblasti úspěšnosti.

Programovací jazyk C, i přes svou nízko-úrovňovost a slabou úroveň abstrakce, je stále velmi rozšířený. Našel si mnohé zástupce v oblasti programování hardware a také v implementaci operačních systémů, kde je největším příkladem jádro Unixu, jehož naprostá většina je napsána v jazyce C. Jazyk C ovlivnil vývoj mnohých nástupců, a proto je možné, že tato práce bude inspirací k vytvoření nástrojů i pro vyšší programovací jazyky. Dalším důkazem o významu tohoto programovacího jazyka je i fakt, že je vyučován jako první jazyk na mnoha školách zaměřených na informační technologie.

Tento projekt si klade za cíl vytvořit nástroj, který statickou analýzou zdrojového kódu v jazyce C a s pomocí nástroje pro řešení logických výrazů, navrhne a vygeneruje vhodnou testovací sadu. Uživatelem aplikace je tester nebo programátor, který má zdrojový text k dispozici. Od této aplikace očekává, že mu nabídne takovou sadu testovacích případů, která dostatečně otestuje zdrojový kód. Otestovaná oblast se může zvětšit, pokud bude tento projekt rozšířen o další postupy při hledání cest (kritéria pokrytí, viz. kapitola 2.2). Tento projekt se zaměřuje jen na jednu úroveň testovacího cyklu a není všelék pro jakýkoliv problém.

Kapitola č. 2 uvede čtenáře do základů testování a kapitola č. 3 do statické analýzy jazyka C. Na ni navazuje část rozebírající návrh a implementaci aplikace (4). Poslední kapitola 5 prezentuje výstupní hodnoty generátoru.

Kapitola 2

Testování softwaru a jeho automatizace

Obsahem této části dokumentu je úvod do problematiky testování softwaru a jeho klasifikace 2.1. Navazuje specifičtější přístup v testování, který je využíván i k řešení problémů naší aplikace, včetně krátkého příkladu 2.2.1. Celou část uzavírá kapitola č. 2.3 o problematice automatizace testování, používaných metodách a automatizačních nástrojích.

2.1 Úvod do testování softwaru

Testování softwaru je v počítačovém inženýrství rozsáhlá oblast. V životním cyklu vývoje softwaru je samostatným odvětvím a hraje důležitou roli od testování dílčích modulů, až po akceptační testování dle specifikace požadavků zákazníkem. Jeho počátky sahají k počátkům rozvoje programování jako takového. Během desítek let se v tomto odvětví vyvinula řada metod a přístupů, jak k testovanému subjektu přistupovat a také množství členění, jakými různé testovací metody od sebe rozlišit. Široký výběr těchto přístupů a metod ukazuje, že žádný z nich nelze považovat za "*správný*" – toto téma bývá námětem mnoha diskuzí — ale každý z nich se hodí pro odlišný problém či dosahuje jiných výsledků z daného úhlu pohledu.

Testovaný subjekt označujeme jako *Testovaný systém*, zkr. **SUT** (z angl. - System under test). Jednou ze základních klasifikací přístupu k testování SUT je: [21]:

- testování černé skříňky (**Black-box** testing)
- testování bílé skříňky (**White-box** testing)
- testování šedé skříňky (**Grey-box** testing)

Černá skříňka označuje uzavřený systém, do kterého nemáme přístup. V takovém případě můžeme pouze předávat data na vstup a analyzovat výstup černé skříňky. Pro skutečně efektivní výsledky testování, tedy vyzkoušení co nejvíce různých stavů černé skříňky, jsme značně limitováni znalostmi o daném SUT. Nejvíce vycházíme ze specifikace požadavků a dokumentace produktu. Často se jako vstupy volí hraniční hodnoty dat, kdy vycházíme z binární podstaty datových struktur v počítači (minimum a maximum celých čísel, prázdný řetězec znaků, neplatný ukazatel, ...). Příkladem takového systému může být grafické rozhraní mobilní aplikace.

Bílá skříňka označuje úplný opak. U takového systému máme plný přístup ke zdrojovým textům. Znalost podkladové struktury kódu programu napomáhá při výběru testovacích případů a otevírá další možnosti způsobu testování, zejména pro více systematické (a efektivnější) pokrytí.

S pojmem šedá skříňka se v praxi nesetkáme tak často. Jde o přístup kombinující obě předešlé metody a příkladem může být program, ke kterému máme pouze API, ale nevidíme do vnitřních struktur.

Dalším důležitým členěním přístupu k systému je **statické** a **dynamické** testování. Statické testování je proces pečlivé a metodické revize návrhu, architektury a kódu programu, při němž hledáme chyby bez jeho spouštění. Někdy se tomuto testování říká také strukturální analýza. U dynamického testování program běží. Na odezvy SUT můžeme okamžitě reagovat, vyhodnotit správnost navržených testů a určit, do jaké míry produkt odpovídá specifikaci.

Vytvořená aplikace k této bakalářské práci analyzuje kód softwaru - pracuje tedy s bílou skříňkou. Slouží pro návrh dynamických testů, ale samotná testovaný software nepouští, jen využívá nástrojů statické analýzy.

2.2 Testování pro dosažení pokrytí

Kritérium pokrytí je pravidlo nebo předpis pro systematické vytváření požadavků na test. *Pokrytí* je míra, udávající jak moc daná testovací sada zkoumá SUT. Testovací sada je soubor testů. Nejjednodušším kritériem pokrytí může být pokrytí všech řádků kódu. Mezi další hledaná kritéria pokrytí patří:

- kritéria pokrytí grafu
- kritéria pokrytí logických výrazů
- kritéria pokrytí vstupních domén
- kritéria pokrytí datových toků

U krytéria pokrytí grafu nahlížíme na zdrojový kód jako na orientovaný graf, ve kterém můžeme nalézt cesty od vstupních bodů po výstupní. V takovém grafu pak hledáme cesty tak, abychom prošli například všechny uzly, všechny hrany nebo kombinace cest přes hrany. Zaměříme-li se na kritérium pokrytí logických výrazů, vybereme si v kódu všechny výrazy ovlivňující větvení programu. Poté hledáme takové hodnoty proměnných, které vyhodnotí vybrané výrazy alespoň jednou jako *true* a jednou jako *false*, či pro dosažení širšího pokrytí hledáme jejich různé kombinace. Vstupní doména je množina všech vstupů SUT (parametry metod, globální proměnné, ...). Myšlenkou kritéria pokrytí vstupních domén je rozdělit vstupní doménu na "kategorie", protože pro některé sémanticky podobné vstupy se program chová stejně. Cílem je otestovat všechny takové kategorie běhů.

2.2.1 Kritéria pokrytí datových toků

V počítačových programech provádíme operace nad daty. Zvolenou abstrakcí říkáme, co pro nás data představují, ale ve své podstatě jde vždy o binární hodnoty v paměti, reprezentované jako proměnné ve zdrojovém kódu. Zaměříme-li se na zajištění, že hodnoty jsou na jednom místě v programu vytvořeny a na jiném by měly být správně použity, sledujeme

datové toky. Právě tato místa jsou předmětem naší analýzy. V místě kde je proměnná vytvořena, je její hodnota uložena do paměti a takovému místu říkáme *definice*, zkráceně *def*. Použití této proměnné (např. ve výrazu, jako argument volání funkce) označuje místo, kde je její hodnota z paměti přečtena k dalšímu zpracování a je tedy místem *použití*, anglicky *use*. Pro určitou proměnnou x vytváříme *def-use* páry, zkráceně *du páry*. Značíme *du(x)*.

Protože *def-use páry* se mohou nacházet na odlišných místech v programu, můžeme říct, že mezi nimi existuje konkrétní cesta, tedy *def-use cesta*, začínající vždy v *def* a končící v *use*. Konkrétní proměnná x může mít v programu více *def(x)* a více *use(x)*, které dokonce mohou vést jinou cestou v programu a stále tvořit jeden konkrétní *def-use pár*. Vede-li ovšem *def-use cesta* pro x přes jiný *def(x)*, porušuje se tím základní myšlenka datových toků a to konzistence hodnoty mezi místem vytvoření a použití. Takové cesty při návrhu testů zásadně vylučujeme a zůstávají nám pouze *čisté du cesty*.

Pro datové toky jsou definována 3 kritéria pokrytí [1]:

- **ADC** (All-Defs Coverage) - testovací sada musí obsahovat alespoň jednu cestu pro každou proměnnou a každou její definici
- **AUC** (All-Uses Coverage) - testovací sada musí obsahovat alespoň jednu cestu od každého def-use páru
- **ADUPC** (All-du-Paths Coverage) - testovací sada musí obsahovat všechny možné cesty ze všech definic proměnné ke všem možným použitím+

Protože data-flow testování je v podstatě sledování stavu proměnných, napomáhá tento přístup k identifikaci problémů s proměnnou [2]:

- která nikdy nebyla použita
- která nikdy nebyla deklarována
- která byla deklarována více než jednou, před jejím prvním použitím
- jejíž paměť byla uvolněna ještě před použitím

Příklad kódu a návrhu testovací sady

```
1 function(x, y) {          | def { x, y }, use {}
2   while x < 5 {           | def {}, use { x }
3     x++                   | def { x }, use { x }
4     y++                   | def { y }, use { y }
5     if y == 2             | def {}, use { y }
6       foo(x,y)            | def {}, use { x, y }
7   }
8 }
9 return y                  | def {}, use { y }
```

Zdrojový text 2.1: Krátký příklad kódu k testování

V příkladu 2.2.1 jsou napravo u příslušných řádků vyznačeny *def* a *use* pro všechny proměnné. Programátor analyzující tuto funkci hledá def-use cesty dle zvoleného kritéria. Zvolíme-li pro jeden test hodnoty x = 10, y = 10, cyklus while se neprovede ani jednou, a i když každá proměnná byla jednou definována i použita, nesplňujeme ani jedno kritérium. Pro "nejnižší" pokrytí musíme projít **každou** definicí. Jiný vstup, např.: x = 4, y = 0,

po průchodu funkcí "navštíví" všechny možné definice a k nim alespoň jedno použití, takže jsme splnili pokrytí ADC. Abychom zvýšili šanci na nalezení chyby, můžeme požadovat ještě širší pokrytí. Přidáme-li test se vstupem x = 4, y = 1, navštívíme i všechny použití a dosáhneme pokrytí AUC. I na tomto jednoduchém příkladu vidíme, že pro naše požadavky nestačí jen jeden *testovací případ*, proto vždy vytváříme testovací sadu. Vyhledat všechny cesty pro splnění ADUPC by znamenalo vytvořit ještě větší sadu, ale se systematickým přístupem i to není obtížný úkol.

2.2.2 Testovací případ – test case

Pro správné generování testů naší aplikací je nezbytné definovat, z čeho se testovací případ zkládá:

- hodnoty testovacího případu - vstupní hodnoty nutné pro dokončení běhu
- očekávaný výsledek - hodnoty produkované programem z hodnot testovacího případu, funguje-li program dle očekávání
- prefixové hodnoty - vstupy nutné pro uvedení SUT do předpokládaného stavu před testem
- postfixové hodnoty - vstupy, které je nutné poslat SUT, aby bylo možné zjistit jeho stav

Vytvořená aplikace automatizuje tvorbu těchto testovacích případů, tedy alespoň těch částí, které jsou zvoleným přístupem automatizovatelné, viz kapitola č. 2.3 o automatizaci. Aplikace hledá takovou množinu testovacích případů, aby bylo splněno zvolené kritérium pokrytí datových toků. Množina testovacích případů se označuje jako *test set* nebo *test suite*. Testovací množina se vkládá do spustitelných testovacích skriptů. Požadavky na testy, do nichž patří i splnění vybraného pokrytí, se obecně označují jako *test requirements* [3].

2.2.3 Úrovně provádění testů

Různé úrovně testování doprovází každou etapu vývoje software:

- Akceptační testování - posouzení softwaru s ohledem na požadavky zákazníka
- Systémové testování - posouzení softwaru s ohledem na celkový návrh architektury
- Integrační testování - posouzení softwaru s ohledem na návrh jednotlivých podsystémů
- Testování modulů - posouzení softwaru s ohledem na podrobnější návrh
- Jednotkové testování - posouzení softwaru s ohledem na implementaci

Vytvořený nástroj se zaměřuje na tvorbu jednotkových testů. *Jednotka* je fragment kódu popisující chování systému a související datové struktury, zaobalený vadále nedělitelném celku, například jednoduché třídy, metody funkce, klauzule (Prolog) nebo triggery (SQL). Konkrétně tato práce se zaměřuje na tvorbu jednotkových testů pro funkce v jazyce C.

2.3 Automatizace testování

Důležitost testování softwaru není třeba zdůrazňovat, jeho největším ukazatelem jsou historické chyby od počátků vývoje softwaru. Chyby v programech mohou stát jejich vydavatele značnou finanční ztrátu či v horších případech mohou ohrozit lidský život. Testování je, stejně jako programování, mentální disciplína a lidé dělají chyby. O automatizaci kteréhokoliv procesu testování je stále větší zájem a aplikacím se daří v této disciplíně dosáhnout různých výsledků na rozlišných úrovních.

2.3.1 Generování testů s hledáním vstupních dat

Techniky založené na vyhledávání využívají statickou analýzu a hledací algoritmy, pro nalezení optimálních vstupních dat s efektivním pokrytím. Hledacími algoritmy mohou být například genetické algoritmy, vedené fitness funkcemi. Náš nástroj, který je blíže popsán v kapitole 4, využívá hledání datových toků a omezujících výrazů, pro které hledá vhodná řešení pomocí CSP solveru.

Základní **myšlenka našeho přístupu**: Tester/programátor má k dispozici zdrojový kód programu a tvoří jednotkové testy pro jednotlivé funkce. V nich si vyhledá všechna *def* a *use*, zvolí si požadované krytérium pokrytí datových toků a pro jeho splnění vybere vhodné cesty. Pro každou cestu najde vhodné argumenty funkcí právě tak, aby program funkcí prošel danou cestou. Na základě nalezených dat vytvoří jednotlivé testy a doplní *asserts*, dle požadovaného chování funkce. My pomocí statické analýzy nalezneme *def-use cesty*. Poté ze všech možných cest vybereme takové, které dohromady splní kritérium pokrytí. V každé cestě postupně zaznamenáváme údaje o všech proměnných tak, jak se objevují ve výrazech a podmínkách, ze kterých dostaneme sadu omezujících výrazů pro hledané vstupní proměnné – argumenty funkce. Tuto sadu předáme CSP solveru, který nalezne řešení, tzn. hodnoty vstupních proměnných (pokud řešení existuje). Z tohoto řešení jsou vytvořeny testovací případy pro každou vybranou cestu. Poslední krok je neautomatizovatelný problém. Neznáme sémantických význam funkcí, a proto nemůžeme doplnit *asserts*, čili tvrzení o chování funkce. To je ponecháno testerovi.

Existují podobné nástroje založené na technikách vyhledávání:

DOTgEAr

Automatický generátor testů DOTgEAr [5] se také zaměřuje na datové toky, ale u objektově orientovaných softwarových systémů. Pro hledání optimálních vstupních dat používá evoluční algoritmy. Pro optimalizaci hledání využívá, kromě statické analýzy, analýzu dynamickou společně s mutačním testováním. Původně byl navržen pro vyhledávání testů pro programy v Javě, má ovšem za cíl vytvořit podporu i pro jazyky C++ a C#.

EvoSuite

EvoSuite [27] je dalším z řady nástrojů generující jednotkové testy pro jazyk Java. Umožňuje výběr z několika různých kritérií pokrytí a spouštět testy v chráněném sandboxu s virtuálním souborovým systémem a virtuálním síťovým připojením. Je založen na metaheuristickém vyhledávání, zejména na technice zvané *seeding* [13].

2.3.2 Náhodné testování

Náhodné testování je založeno na jednoduchém principu — předkládat SUT opravdu náhodné vstupy. Výhodou je dobrý poměr počtu generovaných testů za čas strávený jejich vytvářením, kde pokročilé nástroje dosahují opravdu dobrých výsledků. Tento přístup ovšem není vhodný u programů, které vyžadují garanci širokého pokrytí, což jsou zejména zdravotně kritické aplikace. Náhodně generované testy jsou vyhodnocovány buď pouze ohodnocením prošel/neprošel, pokud test způsobí pád programu, nebo pomocí specifikace či poloautomaticky se zpětnou vazbou od uživatele.

Metody výběru vstupních dat:

- generování čistě náhodných vstupů
- náhodné posloupnosti dat (někdy nazýváno stochastické testování), může odhalit chyby, u kterých záleží na pořadí prováděných operací
- náhodný výběr z existující databáze dat

Jako *ART* (Adaptive Random Testing) se označují algoritmy, které se různými technikami snaží náhodný výběr vylepšit k dosažení efektivnějších výsledků při stejném počtu testů. Vstupní data pak již nejsou čistě náhodná. Jednou z technik je vyloučení okolí již dříve vybrané hodnoty, pro kterou se předpokládá, že bude mít sémanticky stejný vliv na chování programu.

U náhodného testování se někdy můžeme setkat s pojmem *fuzzing*, či *fuzz testing*. Tato technika je založena na vkládání neplatných nebo pozměněných dat a velkého množství dat. Takové testy se mohou označovat i jako "útoky", které zkoumají, za jakých podmínek dojde k pádu aplikace. Tento typ testování je vhodný pro nalezení problémů spojených s chybou typu přetečení zásobníku, DoS útoky, SQL injection, formátem souborů, atp. [9]

Jako *monkey testing* (opičí testování) se obvykle nazývají techniky a nástroje, pracující s SUT jako s černou skříňkou a generující náhodná stochastická data. Často se tyto nástroje používají k testování GUI webových a mobilních aplikací, kde jednotlivé testy představují opici, která slepě kliká. Pokud "opici" nastavíme tak, aby měla základní znalosti o SUT, pamatovala si místa, kde již byla a mohla automaticky hlásit nalezené chyby, mluvíme o chytré opici. V opačném případě jde o hloupou či ignorantskou opici [18].

QuickCheck

QuickCheck [4] je nástroj pro automatické testování programů v jazyce Haskell. Závisí na poskytnutné specifikaci programátorem. Ten specifikuje vlastnosti, které mají být pro jednotlivé funkce splněny. QuickTest poté automaticky otestuje, zda jsou dané vlastnosti dodrženy na velké množině náhodně generovaných testovacích případů. Nástroj poskytuje knihovnu pro definici vlastností, generátorů dat a náhledy na distribuci testovacích dat. Projekt začal v roce 1999 a díky oblíbenosti byly již přepsány jeho verze pro více než 20 jiných programovacích jazyků.

GRT

Guided Random Testing [8] je nástroj pro automatickou generaci testů pro jazyk Java. GRT nejdříve statickou analýzou získá informace o daném SUT, například vhodné konstanty, vedlejší efekty metod a závislosti mezi metodami. Začíná u metod s co nejméně závislostmi, vygeneruje sekvenci vstupů, spustí testy s touto sekvencí a poté dynamickou analýzou

rozhoduje, jestli výsledky použije jako možná vstupní data pro další testy. Je tedy založen na řízení zpětnou vazbou.

Randoop

Randoop [30] je generátor jednotkových testů pro třídy jazyka Java. Principiálně je podobný nástroji GRT. Využívá řízení zpětnou vazbou pro generování testů pro metody a konstruktory jednotlivých tříd. Výsledky získané spuštěním sekvence testů použije pro vytvoření sady tvrzení (assert), které zachycují chování programu. Z těchto tvrzení a sekvencí kódu generuje testovací případy. Tento přístup lze použít jak pro hledání chyb v programu, tak i pro tvorbu regresních testů[1].

american fuzzy lop

American fuzzy lop je fuzzer, který pro efektivnější navyšování pokrytí generovanými testy používá genetické algoritmy. Na začátku vyžaduje spouštění SUT s alespoň jedním souborem vstupních dat. Po zjištění reakcí programu na tento soubor a pokusu o redukci jeho velikosti, je vstupní soubor modifikován ve snaze objevit nečekaná chování. Nabízí také nástroje pro zkoumání vyvolané chyby a syntaktický analyzér pro co nejlepší identifikaci problému.

SAGE

Scalable Automated Guided Execution (SAGE) je white-box fuzzing nástroj. Po spuštění analyzovaného programu s prvními vstupy, získává SAGE omezení na vstupy v rozhodovacích podmínkách a pomocí CSP solveru generuje vstupy pro další testy. Pro tyto nové testy se celý proces opakuje, teoreticky donekonečna. Jde o proprietární software používaný firmou Microsoft [10].

2.3.3 Symbolická exekuce

Symbolická exekuce je analýza kódu, za účelem zjištění, jaké vstupy způsobují exekuci jednotlivých částí. Během symbolické exekuce jsou argumenty funkce nahrazeny symbolickými hodnotami. K těmto symbolům jsou přidány určující výrazy, tak jak se postupně objevují ve výrazech daného programu. V podmínkách se poté exekuce rozděluje na dvě nezávislé větve, jedna pro splnění a druhá pro nesplnění dané podmínky. Pro vyhodnocení konkrétních hodnot symbolů je použit SMT/CSP solver.

KLEE

Jedním z nejlepších nástrojů založených na symbolické exekuci je **KLEE LLVM Execution Engine** [16]. Tento open source projekt si zakládá na dosažení vysokého pokrytí u škály programů rozdílných druhů.

DART

DART: Directed Automated Random Testing [7] byl jedním z prvních nástrojů využívajících symbolickou exekuci. Pro automatické testování software kombinuje 3 hlavní

[1] Regresní testy se využívají při opětovném testování funkcí a vlastností aplikace pro ověření, že provedené změny či implementace nových vlastností v aplikaci nemělo žádný vliv na stávající funkce a vlastnosti [17].

techniky: automatickou extrakci rozhraní programu pomocí statické analýzy, automatickou generaci náhodných testů pro simulaci nejobecnějšího prostředí daného programu a dynamickou analýzu náhodných testů pro řízení systematické symbolické exekuce alternativními cestami v programu.

CREST

CREST [26] je automatický generátor testů pro jazyk C. Pracuje na principu vkládání příkazů do analyzovaného kódu, pro provedení kombinace symbolické exekuce souběžně s konkrétní exekucí. Tento princip je označován jako *concolic testing*. Pro hledání konkrétních hodnot využívá SMT solver Yices, ale nejnovější verze byla rozšířena o další SMT solver (Z3), který podporuje nelineární aritmetiku.

2.3.4 Pomocné nástroje pro automatizaci

Nástroje této kategorie nejsou plně automatickými generátory testů, ale pouze podpůrnými nástroji pro jejich tvorbu. I přesto ovšem stojí za zmínku, protože jsou v praxi velmi používány a často i integrovány do komplexních nástrojů.

*x*Unit

Nástroj nenese přímo název **xUnit**, ale prefixové *x* lze nahradit mnoha zkratkami pro programovací jazyky, např.: **JUnit** pro Javu [23], **CPPUnit** pro C++[22], **PyUnit** pro Python [12]. Jde tedy o označení rodiny frameworků, určených pro vytváření jednotkových testů. Nabízejí API pro velmi jednoduchou přípravu daného prostředí a definici jednotlivých testovacích případů. Právě pomocí těchto frameworků bývají tvořeny jednotkové testy v nástrojích zmíněných výše.

Selenium

Selenium [31] je testovací framework pro webové aplikace. Nabízí jednoduchý nástroj pro nahrávání a opakování akcí v prohlížeči, bez nutnosti učit se skriptovací jazyk. Také nabízí specifický testovací jazyk, určený ke tvorbě testů pro mnoho populárních programovacích jazyků, které mohou být spuštěny ve většině moderních webových prohlížečů.

Kapitola 3

Statická analýza

Tato část bakalářské práce popisuje zvolený jazyk a dostupné nástroje k jeho analýze. Začíná představením několika forem data-flow analýzy (3.1), na což navazuje představení překladače zvoleného jazyka (3.2) a ukázka AST (3.2.1), který analyzujeme. Kapitola č. 3.3 krátce představuje jazyk C a podporované konstrukce v naší aplikaci. Celou část uzavírá kapitola o řešení logických výrazů pomocí CSP solveru (3.4).

3.1 Formy data-flow analýz

Data-flow analýza je tradiční technika analýzy programů. Používá se pro sběr informací o toku datových hodnot napříč základními bloky, které se mohou využít k [11]:

- aplikaci optimalizací překládaného kódu
- symbolickému debugování, exekuci
- statickému hledání chyb
- odvození datových typů

3.1.1 Available expression Analysis

Analýza dostupných výrazů. Pro každý bod v programu určíme, které výrazy **musí** již být vypočteny, a později nemodifikovány, u všech cest k bodu v programu. Proces můžeme rozdělit do třech akcí:

1. výraz je v bodě *generován*, pokud je v tomto bodě spočítán
2. výraz je *zabit* definováním operandů výrazu
3. výraz A+B je *dostupný* v bodě, pokud každá cesta od začátku do tohoto bodu vyhodnocuje A+B a po posledním vyhodnocení nenásleduje definice operandu A ani B

3.1.2 Reaching definitions analysis

Analýza dosažení definic. Pro každý bod v programu určíme, která přiřazení **mohla** být provedena a nepřepsána, když spuštěný program běžel tímto bodem po některé z cest. Tento přístup odpovídá hledáním *def-use* cest popsaných v kapitole č 2.2.1.

3.1.3 Live variables analysis

Analýza živých proměnných. Udává pro každý bod programu, které proměnné **by mohly** být naživu na výstupu z bodu programu. Proměnná je živá, pokud obsahuje hodnotu, která může být později použita. Určení živých proměnných můžeme provést ve dvou krocích v bodech programu:

1. cesta je *X-clear* pokud neobsahuje žádné definice proměnné X
2. proměnná X je živá v bodě p, pokud existuje *X-clear* cesta od bodu p k místu použití proměnné X; jinak X je mrtvá v bodě p

3.1.4 Very busy expressions analysis

Analýza velmi vytížených výrazů. Cílem Very Busy Expressions Analysis je určit pro každý bod programu, které výrazy **musí** být *very busy* na výstupu z tohoto bodu. Výraz je *very busy* pokud je zaručeno, že bude někdy v budoucnu výraz vypočítán. Například výraz A+B bude *very busy* v základním bloku, který se jako if výraz větví do obou základních bloků takových, ve kterých bude výraz A+B vypočítán.

3.2 LLVM a Clang

LLVM [25] je projekt překladačové infrastruktury, zahrnující kolekce modulárních a znovupoužitelných technologií z oblasti překladačů, používaných k vývoji front-end a back-end částí kompilátorů. Původně název LLVM byl zkratkou pro Low Level Virtual Machine (nízko-úrovňový virtuální stroj), ale postupem času se z něj stal projekt zastřešující mnoho překladačových technologií. Je napsán v C++ a navržen pro optimalizace v různých fázích překladu programu (překlad, linkování, běh), původně pro jazyky C a C++, ale díky své obecnosti a rozhraní, umožňuje použití pro širokou škálu jazyků zahrnující funkcionální i imperativní a interpretované i překládané (i částečně překládané jako Java). Samotný LLVM je tedy back-end pro širokou škálu procesorových architektur, čímž se rozumí ta část překladače zajišťující překlad do strojového kódu. Jeho vstupem je mezikód vytvořený některým z front-endů, navržených pro rozhraní LLVM, které mohou být vyvíjeny pod záštitou projektu LLVM. Pro jazyk C jde o front-end Clang.

Clang [14] je překladačový front-end pro programovací jazyky C, C++, Objective-C, Objective-C++, OpenMP, OpenCL a CUDA. Stejně jako LLVM jde o software s otevřeným zdrojovým kódem. Je navržen aby mohl nahradit nejvíce používanou kolekci překladačů - GCC, díky čemuž s tímto projektem sdílí skoro stejné rozhraní. Projekt Clang také zahrnuje statický analyzátor a poskytuje jednoduché rozhraní, pro používání a vytváření nových analytických nástrojů. Díky tomu byl nejlepším kandidátem pro použití v tomto projektu.

3.2.1 AST

Abstraktní syntaktický strom (z angl. - Abstract Syntax Tree) [19], zkráceně **AST** je stromová reprezentace abstraktní syntaktické struktury zdrojového kódu. Syntaxe je abstraktní v tom smyslu, že ne všechny detaily musí být reprezentovány v této reprezentaci, jako například uzavírací závorky. AST je produktem syntaktické analýzy front-endu překladačů a jeho určitá podoba je závislá na návrhu konkrétního překladače. Všechny reprezentace ale AST mají povinné společné rysy:

- typy proměnných musí být zachovány, stejně jako místo každé deklarace
- pořadí spustitelných výrazů musí být reprezentováno a dobře definováno
- levé a pravé komponenty binárních operací musí být uchovány a správně identifikovány
- identifikátory a jejich přiřazené hodnoty musí být uchovány pro výrazy přiřazení
- mělo by být možné přeložit AST zpět do formy zdrojového kódu

Kořenem AST může být funkce či celý modul zdrojového kódu a dále se větví k příslušným výrazům. Výrazy stejného typu mají vždy stejnou reprezentaci, a tedy můžeme říct, že existuje konečná množina výrazů, ze kterých je AST složen. Jelikož budeme využívat AST, který vytváří Clang, je nutné této reprezentaci porozumět. Uveďme si příklad krátkého kusu kódu, jak vypadá jeho grafická reprezentace ve stromové struktuře a jak vypadá jeho textová podoba vypsaná pomocí Clang:

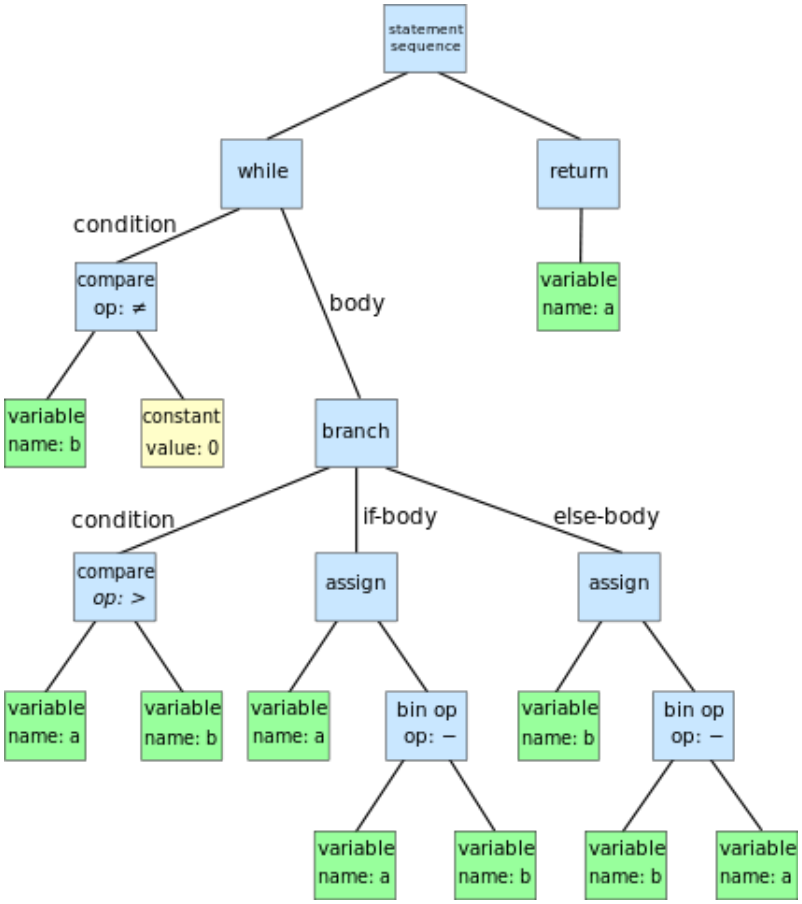
```
int a = 5, b = 7;
while (b != 0) {
  if (a > b)
    a = a - b;
  else
    b = b - a;
}
return a
```

Zdrojový text 3.1: Krátký algoritmus v jazyce C

Pro výpis textové reprezentace lze použít příkaz `clang -cc1 -ast-dump test.c`. Pro barevné odlišení doporučuji přidat přepínač `-fcolor-diagnostics` a pro lepší porozumnění jednotlivých částí nastudovat referenci Clangu [6].

```
-DeclStmt 0xa8ddf98 <line:7:2, col:18>
| |-VarDecl 0xa8dde70 <col:2, col:10> col:6 used a 'int' cinit
| | `-IntegerLiteral 0xa8dded0 <col:10> 'int' 5
| `-VarDecl 0xa8ddf00 <col:2, col:17> col:13 used b 'int' cinit
|   `-IntegerLiteral 0xa8ddf60 <col:17> 'int' 7
`-WhileStmt 0xa8de320 <line:8:2, line:13:2>
|-BinaryOperator 0xa8de010 <line:8:9, col:14> 'int' '!='
| |-ImplicitCastExpr 0xa8ddff8 <col:9> 'int' <LValueToRValue>
| | `-DeclRefExpr 0xa8ddfb0 <col:9> 'int' lvalue Var 0xa8ddf00 'b' 'int'
| `-IntegerLiteral 0xa8ddfd8 <col:14> 'int' 0
`-CompoundStmt 0xa8de300 <col:17, line:13:2>
  `-IfStmt 0xa8de2d0 <line:9:2, line:12:11>
    |-BinaryOperator 0xa8de0b8 <line:9:6, col:10> 'int' '>'
    | |-ImplicitCastExpr 0xa8de088 <col:6> 'int' <LValueToRValue>
    | | `-DeclRefExpr 0xa8de038 <col:6> 'int' lvalue Var 0xa8dde70 'a' 'int'
    | `-ImplicitCastExpr 0xa8de0a0 <col:10> 'int' <LValueToRValue>
    |   `-DeclRefExpr 0xa8de060 <col:10> 'int' lvalue Var 0xa8ddf00 'b' 'int'
    |   .
        .
        .
`-ReturnStmt 0xa8de380 <line:15:2, col:9>
  `-ImplicitCastExpr 0xa8de368 <col:9> 'int' <LValueToRValue>
    `-DeclRefExpr 0xa8de340 <col:9> 'int' lvalue Var 0xa8dde70 'a' 'int'
```

Zdrojový text 3.2: Textová reprezentace AST



Obrázek 3.1: Grafická reprezentace AST. Převzato z [19]

3.2.2 LibTooling

Clang nabízí 3 možnosti napojení na proces překladu a jeho AST. Prvním je **LibClang**, což je základní rozhraní v jazyce C, nabízející vyšší úroveň abstrakce nad AST. Druhou možností je rozhraní **Clang Plugin**, které umožňuje programování zásuvných modulů. Zásuvné moduly provádět dodatečné akce jako část překladu a spouštějí se zároveň s překladačem příslušným argumentem. Nabízí například možnost dodatečných informačních zpráv o varováních a chybách při překladu, ale neumožňují plnou kontrolu nad AST. Třetí knihovna, což je ta kterou používáme, je **LibTooling**. LibTooling [15] je C++ rozhraní určené pro programování samostatných nástrojů, využívajících technologie Clangu a nabízí plnou a nezávislou kontrolu nad AST.

3.3 Výběr jazyka k analýze

C je procedurální programovaný jazyk, vyvinutý počátkem 70. let pro potřeby operačního systému Unix, pány Kenem Thopsonem a Dennisem Ritchiem. Je kompilovaný a poměrně nízko-úrovňový, takže programátor si akce na vyšší úrovni abstrakce, které nabízejí jiné programovací jazyky, musí zaručit sám. Tento jazyk je velmi rozšířený a proto věřím, že

nástroj pro automatické generování testovacích případů bude mít velké uplatnění. Široká nabídka knihoven a nástrojů pro analýzu jazyka C byla další důvodem jeho výběru.

3.3.1 Podporované konstrukce jazyka C

3.3.2 Rozhodovací konstrukce

- if a if-else podmínky
- do-while cykly
- while cykly
- for cykly
- switch přepínače
- goto skoky
- break
- continue

3.3.3 Aritmetické operátory

- =
- + - * /
- += -= *= /=
- ,
- výběrový operátor .

3.3.4 Operátory rovnosi a nerovnosti

- == !=
- < >
- <= >=
- && ||

3.4 CSP a datové typy

Jak bylo naznačeno ve 2. kapitole, k řešení hodnot proměnných byl použit CSP solver. CSP je zkratka pro *Constraint satisfaction problem* (Problém splnitelnosti omezení - volně přeloženo). CSP jsou obecně matematické problémy definovány jako množina objektů, jejichž stav musí splňovat konečný počet omezení - *constraint*. Formálně je CSP definován jako trojice $\{X, D, C\}$, kde:

- $X = \{X_1, X_2, ..., X_n\}$ je množina proměnných

- $D = \{D_1, D_2, ..., D_n\}$ je množina příslušných definičních oborů
- $C = \{C_1, C_2, ..., C_n\}$ je množina omezení

Každá proměnná X_i může nabývat hodnot z neprázdné D_i. Každé omezení $C_j \subset C$ je dvojice $\{t_j, R_j\}$, kde $t_j \subset X$ je podmnožina k proměných a R_j je k-rozměrná relace ke korespondujícím podmnožinám definičních oborů D_j. Ohodnocení proměnných je funkce z podmnožiny proměnných k určité množině hodnot k patřičným podmnožinám definičních oborů. Ohodnocení v splňuje omezení $\{t_j, R_j\}$, pokud hodnoty přiřazené k proměným t_j splňují relaci Rj [24]. Aby ohodnocení bylo řešením musí splňovat dvě podmínky - konzistentnost a úplnost. Ohodnocení je konzistentní pokud neporušuje žádné z omezení a je úplné, pokud zahrnuje všechny proměnné. O takovém ohodnocení můžeme prohlásit že řeší daný CSP.

CSP solver je program, který řeší dané CSP problémy (z angličtiny: solver - řešitel). Řešící algoritmy používají techniky jako backtracking, propagace omezení a lokání vyhledávání. Z nich nejpoužívanější, backtracking, je rekurzivní algoritmus založený na vytváření stromové struktury s ukládáním informací o každém kroku, kde každý uzel stromu představuje výběr hodnoty pro jednu z proměnných. Pokud je postupným výběrem porušeno některé omezení, vrací se algoritmus po větvích stromu směrem ke kořeni, kde v každém uzlu zruší vybranou hodnotu a vybere novou, dokud nemá již z čeho vybírat. Touto metodou jsou postupně prohledána všechna řešení. Dnešní solvery používají modifikaci některé z těchto technik. Dnešní CSP solvery používají formu Constraint modelování pro vytváření modelů popisujících daný CSP. Při výběru určitého solveru byl kladen důraz na možnost spolupráce s knihovnou LibTooling, která je napsána v jazyce C++ a na kvalitu dokumentace.

3.4.1 Gecode

Gecode [28] je nástroj pro vytváření systému a aplikací založených na constraint modelování. Nabízí velmi dobrou rychlost, efektivnost a jednoduché rozhraní pro snadnou integraci do vyvíjených aplikací. V letech 2008-2012 dokonce zvítězil ve všech kategoriích na MiniZinc Challenge[1]. Jedná se o svobodný software s otevřeným zdrojovým kódem, distribuovaný pod MIT licencí. Součástí projektu Gecode je i terminálový nástroj pro rychlé vyzkoušení Constraint modelování a vizualizační nástroj GIST (Grafický interaktivní vyhledávací nástroj), poskytující jak začátečníkům v CSP modelování grafický náhled na problém, tak pokročilým zasahovat do procesu vyhledávání a tím experimentovat s různými strategiemi vyhledávání, což může být vhodné k urychlení či pochopení řešení daného problému.

3.4.2 Podporované datové typy

Celočíselné datové typy

V jazyce C mezi tyto datové typy patří *char, short, int, long int a long long int*, včetně jejich znaménkové (*signed*) i bezznaménkové (*unsigned*) formy. V Gecode pro modelování číselného typu musíme použít vestavěný typ *IntVar*. Každá proměnná typu IntVar musí být inicializována a to buď rozmezím možných hodnot od-do, vestavěnou množinou hodnot - *IntSet* nebo kopírovacím konstruktorem z jiné, již vytvořené proměnné. Obsahuje-li zdrojový kód bezznaménkovou proměnnou, příslušnou IntVar proměnnou zdola omezíme nerovností

[1] MiniZinc je každoroční soutěž CSP solverů na různorodých benchmarcích [29].

pro hodnoty větší či rovny nule. Dále je příslušný typ (char, short, ...) omezen dle velikosti alokované paměti na nejvyšší možnou hodnotu. Tady ovšem přichází omezení v modelování v systému Gecode pro tzv. "velká čísla", což jsou čísla uložena na více než 4 bytech pro dosažení většího rozsahu. Gecode definuje minimum a maximum v hlavičkovém souboru `Int/Limits.hpp`, jako konstanty `Int::Limits::min` a `Int::Limits::max`, které mohou být závislé na architektuře stroje, na kterém byl Gecode přeložen. Tato velikost je vždy ale maximálně v rozmezí 32-bitového integeru. Podpora pro 64-bitová čísla v Gecode není možná, tyto velká čísla používá pro vnitřní výpočty, například lineárních závislostí mezi modelovanými IntVar. Každá proměnná větší nebo rovna klasickému integeru, je tedy tímto limitem omezena a proto není tento nástroj vhodný pro testování systému, u kterých je nutností modelovat 64-bitová čísla.

V průběhu modelování je nutné k proměnným přidávat další omezení oboru hodnot. K tomu je poskytnuta funkce relace – `rel()`, očekávající na vstup dvě proměnné typu IntVar a numerickou hodnotu určující typ relace - IRT_ EQ (=), IRT_ NQ ($\neq$) IRT_ LE (<), IRT_ LQ ($\leq$), IRT_ GR (>) nebo IRT_ GQ ($\geq$). Dále Gecode poskytuje vytváření polí IntVarArray a dodatečné omezení mezi nimi, například *distinct* - všechny proměnné pole musí mít různou hodnotu, *linear* pro lineární rovnice všech členů, atp.

Čísla s plovoucí řadovou čárkou

Standardně se pro tyto účely používají proměnné typu float, double a long double s vzrůstající přesnosti reprezentace desetinných čísel. Gecode nabízí opět jediný typ *FloatVar*, který je ukládán tak, aby dosáhl co největší přesnosti. Gecode používá intervalovou aritmetiku – hodnota není uložena jako jediné číslo, ale jako dvojice, omezující hodnotu shora a zdola. Tím se snaží zvýšit přesnost při častém zaokrouhlování v aritmetických operacích. FloatVar musí zase být inicializován než je možné jej použít ve výpočtech a jeho rozmezí je definováno ve *Float/limits.hpp* jako `Float::Limits::min` a `Float::Limits:max`, jejichž přesná hodnota není známa. Opět je k dispozici možnost vytvoření polí FloatVarArray a operací nad nimi. Experimentálním modelováním s typem FloatVar bylo zjištěno, že inicializace na jeho limitní hodnoty definované v hlavičkových souborech, značně zpomaluje délku vyhodnocení, již při použití malého počtu proměnných. Proto je v naší aplikaci limit nastaven od -50.0 do 50.0. Pokud uživatel chce zvýšit tento limit, musí spustit nástroj s argumentem *-float <limit>*, kde *limit* představuje číselnou hodnotu.

Struktury

Struktury jsou heterogenní datové typy, složené z různých jednoduchých a složených datových typů i různého počtu. Gecode nenabízí žádný vestavěný datový typ na jejich vytváření, nezbývá tedy než zařadit položky, které jsou jednoduchého datového typu mezi ostatní proměnné, a uchovat si informace o jejich umístění v modelovaném systému.

Kapitola 4

Návrh a implementace

Funkční proces vytvořené aplikace lze rozdělit do 4 částí. V první fázi (4.1) je analyzován vstup čili zdrojový kód, ze kterého je vytvořen CFG. Při aplikaci kritéria pokrytí All-Defs (4.2) jsou nalezeny *def-use cesty* a vyhledány cesty programem pro splnění onoho pokrytí. Následně je nad každou cestou znovu prováděna statická analýza, u které se vytváří model pro CSP solver (4.3). V poslední fázi (4.4) je pro každou cestu vytvořen testovací případ, dle nalezených vstupních argumentů. Postup je znázorněn na obrázku č. 4.4.

4.1 1. fáze - tvorba CFG ze zdrojového kódu

- **Vstup:** zdrojový kód ve formě AST
- **Výstup:** CFG

Control Flow Graph, zkratka **CFG** [20], je jednou z možných reprezentací zdrojového kódu. V kapitole 3.2.1 jsme si představili podobu AST, ovšem tato reprezentace je nevhodná pro hledání cest průchodů programem. Standardně se používá CFG, který má stromovou strukturu, i když větve mohou vézt do zpět do vyšších uzlů. Jeden uzel v CFG představuje základní blok.

Základní blok je základní stavební kámen CFG a představuje největší možnou skupinu příkazů, o kterých můžeme s určitostí prohlásit, že budou vždy provedeny po sobě ve stejném pořadí. V jazyce C tok programu ovlivňují řídící konstrukce, představené v kapitole 3.3.1. Všechny ostatní výrazy jsou sdruženy do posloupnosti příkazů v příslušných základních blocích.

Ze všeho nejdříve je nastaven Clang. Je vytvořen `Visitor`, reagující na deklaraci funkce a předán Clangu pro analýzu vstupního souboru, který je ve formě čistého zdrojového kódu. Clang vytvoří AST a při každé nalezené definici funkce, je volána callback funkce našeho `Visitoru`. Zde začíná naše statická analýza, pro každou funkci zvlášť. `Clang::FunctionDecl` je třída z knihovny LibTooling, obsahující objekty související s deklarací funkce. Prvním krokem je uložení funkčních parametrů, pro které budeme hledat konkrétní hodnoty a následuje analýza těla funkce.

Jednotlivé konstrukce jazyka jsou v AST reprezentovány specifickými třídami, které všechny dědí z `clang::stmt`. U každé položky AST je dynamickým přetypováním zjištěn její typ, podle kterého jsou provedeny odpovídající akce. Při nalezení řídící konstrukce je CFG odpovídajícím způsobem "rozvěten" a proveden přesun do patřičného základního bloku. Při nalezení výrazu, je výraz zařazen na konec vektoru výrazů do základního

bloku. Procházení AST je umožněno skrze odkazy v třídách reprezentujících řídící konstrukce. Například `clang::IfStmt` obsahuje metody `getCond, getThen` a `getElse`, které vrací odkaz na *statement* podmínky, větve *if-then* a větve *else*. Každá větev pak může být `clang::CompoundStmt`, což je třída reprezentující blok příkazů (v jazyce C ohraničen složenými závorkami), který obsahuje pouze pole všech *statement* uvnitř sebe.

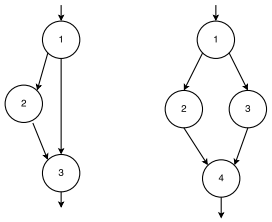
Vytvořené základní bloky – uzly CFG – obsahují, mimo jiné, tyto informace:

- typ uzlu - např. if, while, switch, obyčejná posloupnost výrazů
- podmíněný výraz (jde-li o podmíněnou řídící konstrukci)
- pole ukazatelů na navazující uzly

Obsahuje-li některý z výrazů ternární operátor, je objeven v této fázi programu. Je nutné projít AST všech výrazů a je-li tento ternární operátor nalezen, je CFG rozvětven stejně jako při konstrukci *if-else*, kde výrazen podmínky je podmínka ternárního operátoru a obě větve obsahují pouze právě analyzovaný výraz, kde je ternární operátor nahrazen *true* či *false* stranou.

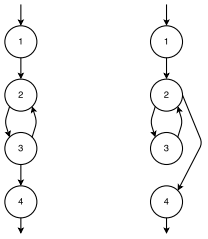
4.1.1 Reprezentace řídících konstrukcí v CFG

Podmíněná konstrukce If-than a If-than-else



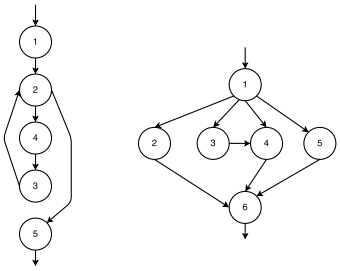
Obrázek 4.1: Vlevo if-then a napravo if-then-else

Cykly While a Do-while



Obrázek 4.2: Vlevo do cyklus a napravo cyklus do-while

For cyklus a přepínač switch



Obrázek 4.3: Vlevo for cyklus a vpravo jedna z možných variant přepínače Switch

4.1.2 Související moduly

AutoTestGen.cpp

Modul s `main()` funkcí. Vytváří objekt třídy `WalkThroughAST`, kterému předává vstupní argumenty programu.

WalkThroughAST.cpp a WalkThroughAST.h

Modul obsahuje vytvoření `Visitoru` nástroje Clang a definici jeho callback funkce. Zpracovává vstupní parametry a s němi spouští `ClangTool`. Třída `WalkThroughAST` také obsahuje vektor tříd `FunctionUnderTest` a veřejnou metodu pro jeho získání.

FunctionUnderTest.cpp a FunctionUnderTest.h

Modul reprezentující jednu testovanou funkci. V této fázi programu je stěžejní částí vytvoření objektu `DefUseCFG` a metody k získání odkazu na něj.

DefUseCFG.cpp, DefUseCFG.h a DefUseBlock.h

DefUseBlock.h obsahuje definici základních bloků, ze kterých je tvořen CFG. Modul DefUseCFG, vedle konstruktoru a několika jednoduchých metod, obsahuje důležité metody, viz výše. Většina práce je prováděna ve funkci `appendTree`, což je rekurzivní metoda procházející prvky AST a připojujícíc jednotlivé části CFG k sobě. Po vytvoření CFG je zavolána metoda `cleanRedundantBlocks`, která odstraní prázdné základní bloky, které mohly být vytvořeny jako vedlejší produkt u neúplných řídících konstrukcí.

4.2 2. fáze - aplikace kritéria pokrytí

- **Vstup:** CFG funkce
- **Výstup:** množina cest, která splňuje zvolené kritérium pokrytí

V kapitole 2.2 byly představeny 3 různá kritéria pokrytí datových toků. Z nich bylo konkrétně vybráno pokrytí All-Defs. Výběr určitého kritéria nemá větší vliv na chování aplikace či výsledky generátoru. All-Defs je pro demonstaci funkčnosti generátoru dostačující kritérium pokrytí.

V první části druhé fáze jsou statickou analýzou nalezeny všechny definice a použití všech proměnných. Princip je podobný analýze řídících konstrukcí z 1. fáze. Nyní již analyzujeme pouze výrazy, pro které je rodičovskou třídou `clang::Expr`. Typ části výrazu zjistíme dynamickým přetypováním na určitou dceřinou třídu. Postupně se zanořujeme do podvýrazů přes odkazy v binárních, unárních, výběrových a závorkovacích operátorech, až narazíme na konečnou část výrazu, což může být konkrétní hodnota (celočíselná, znak, atd.) nebo odkaz na proměnnou. V této části nás zajímají pouze proměnné. *Def* proměnné je vyvolán pro proměnné stojící na levé straně operátoru přiřazení a pro proměnné s unárním inkrementací, či dekrementací. Je dbán důraz na pořadí provádění těchto operací.

V druhé části druhé fáze jsou vyhledány cesty průchodu programem, které tvoří základ pro tvorbu testovacích případů. Vyhledávací algoritmus byl implementován rekurzivním způsobem. První volání algoritmu je s kořenovým uzlem:

1. uložíme si identifikátor uzlu
2. projdeme pole definic a použití:
   (a) najdeme-li def, zařadíme jej jako aktuální def do mapy, kde klíčem je unikátní identifikátor proměnné, společně s identifikátorem uzlu
   (b) najdeme-li use, vyhledáme dle identifikátoru proměnné aktuální def, ze kterého použijeme identifikátor uzlu a identifikátor proměnné do pole def-use cest
3. projdeme pole odkazů na navazující uzly CFG
   (a) je-li pole prázdné, přidáme aktuální cestu společně s polem def-use cest do seznamu nalezených cest, algoritmus končí
   (b) je-li pole neprázdné, zavoláme rekurzivně algoritmus s pro každý následující uzel a každému volání předáme aktuální informace (posloupnost cesty, mapu proměnná-def, pole dosažených def-use cest)

Mapou se rozumí kontejner typu slovník. Omezení generování při maximálně jednom výskytu přechodu (a tedy se přechod může vyskytnout maximálně 2krát) je dostatečně, pro nalezení všech def-use párů. Přidání aktuální cesty do seznamu nalezených cest, je podmíněnou nutností cesty projít alespoň jednu ještě nedosaženou definici.

4.2.1 Související moduly

AutoTestGen.cpp

Modul s `main()` funkcí. U každé nalezené funkce z 1. fáze zavolá metodu na aplikaci All-Defs kritéria pokrytí.

AllDefsCoverage.cpp a AllDefsCoverage.h

Modul pro definici testovací sady s kritériem pokrytí All-defs. Konstruktor požaduje odkaz na CFG aktuální funkce. Privátní metoda *followPath* implementuje rekurzivní algoritmus hledání všech cest a metoda `addTest` nalezené cesty přidává do seznamu vybraných cest, pokud nová cesta prochází ještě nenavštívenou definicí. Veřejná metoda `getNextPath` je důležitá pro třetí fázi, kde vrací jednotlivé cesty ze seznamu vybraných cest.

DefUseCFG.cpp, DefUseCFG.h a DefUseBlock.h

Pro tuto fázi modul DefUseCFG obsahuje metodu pro analýzu výrazů a hledání definic a použití proměnných.

4.3 3. fáze - hledání vstupních hodnot pomocí CSP solveru

- **Vstup:** Cesty průchodu programem tvořené posloupností základních bloků
- **Výstup:** Nalezené řešení pro danou cestu

Třída modelu pro použití knihovny Gecode, musí splňovat následující pravidla:

- dědit ze třídy Space z knihovny Gecody
- implementovat kopírovací konstruktor
- implementovat virtuální funkci copy(), vracející aktuální ukazatel na používaný Space
- všechny inicializované proměnné, relace a výrazy musí patřit do aktuálního Space (požadují ukazatel na Space jako argument)
- označit proměnné a metodu k *branchingu*

Entitou typu *Space* se rozumí modelovací "prostor", ve kterém se automaticky propagují všechny constraint (omezení), k příslušným proměnným. Branching je strategie větvení, pro nalezení vhodných hodnot proměnných. Branchingem se není třeba zabývat, pokud se nezaobíráme optimalizacemi.

Členskými proměnnými třídy našeho modelu jsou pole Gecode proměnných pro celá čísla a čísla s plovoucí desetinou čárkou, zvlášť pro argumenty a zvlášť pro lokální proměnné. Předmětem našeho úsilí je nalézt hodnoty argumentů, takže na tato pole musí být po dokončení modelování aplikován branching. Gecode nabízí 2 typy polí IntVarArgs a IntVarArray (zaměnitelné s Float), z nichž branching jde napojit pouze na IntVarArgs, které však oproti IntVarArray má statickou velikost a nepodporuje dynamické přidávání proměnných. Vzhledem k neznámému počtu argumentů (kvůli polím a ukazatelům), pro nás není inicializace na určitý počet prvků pole vhodné řešení. Tento problém lze ovšem jednoduše vyřešit. Pokaždé když přidáváme dynamicky novou proměnnou, vyrobíme z aktuálního pole argumentů dynamické pole proměnných, přidáme novou proměnnou a pole argumentů znovu inicializujeme na námi rozšířené pole proměnných. Tento přístup vyžaduje, aby byl branching aplikován až po dokončení modelování.

Nyní již umíme vytvářet proměnné a v kapitole 3.4.2 jsme si představili i modelování relací. Dalším krokem je navázat aritmetickými operacemi, mezi proměnnými. K tomu slouží funkce `expr()`, kde první argument je opět Space, ve kterém modelujeme a druhým parametrem je výraz. Jsou podporovány operace sčítání, odečítání, dělení, násobení a modulo s odpovídajícími symboly z matematiky. Gecode toho ovšem umí víc, jako jsou odmocniny, mocniny, maximum, atp., které ovšem nevyužijeme. Funkce expr je přetížená, a argumenty mohou být buď typu celého čísla nebo čísla s plovoucí desetinnou čárkou a návratová hodnota je výsledná proměnná v odpovídajícího typu. Chceme-li provést matematickou operaci mezi celým číslem a desetinným číslem, musíme vytvořit pomocnou proměnou a funkcí *channel()* ji nastavit rovnost na proměnnou rozdílného typu. Jak tedy

probíhá samotné modelování. Pro každou cestu vytvoříme nový model - instanci třídy dědící z `Gecode::Space` a popořadě procházíme navštívené základní bloky. Každý blok obsahuje výrazy, které převedeme na operace v modelu. Například výraz:

```
1 int x = i + 42 * (i - 42);
```

Zdrojový text 4.1: Jednoduchý výraz

lze převést na trojici výrazů a vložení do pole lokáních proměnných:

```
1 IntVar tmp1 = expr(*this, i - 42);     // mezivysledek (i - 42)
2 IntVar tmp2 = expr(*this, 42 * tmp1); // mezivysledek (42 * (i - 42))
3 IntVar x = expr(*this, i + tmp2);    // vysledek cele rovnice
4 local_int_variables << x;      // vlozeni promenne x do pole typu IntVarArray
```

Zdrojový text 4.2: Model jednoduchého výrazu v Gecode

My ovšem tvoříme výrazy dynamicky a díky stromové struktuře AST, lze každou operace převést na rekurzivní volání levé a pravé strany listových uzlů, vracející proměnou příslušného typu, představující buď mezivýsledky operací nebo přiřazenou hodnotu z objevené lokální proměnné.

Jak navázat proměnné ze zdrojového kódu na proměnné v modelu? Základem je opět mapa (kontejner typu slovník), kde klíčem je jednoznačný identifikátor proměnné a hodnota na pozici klíče je odkaz do pole argumentů/lokálních proměnných. Argumenty funkce jsou na začátku inicializovány v poli argumentů, ale pokud s v kódu vyskytuje jejich přiřazení na jinou hodnotu (jak výsledek výrazu, tak i unární inkrementace/dekrementace), stávají se jejich aktuální stavy lokální proměnnou (vazby v modelech CSP zůstávají). Jde-li o členskou proměnnou struktury, jejím klíčem je složenina identifikátoru struktury a odpovídajících členských proměnných tvořící řetězec vedoucí ke konkrétnímu členu.

Posledním problémem je volání funkcí. Tato práce se nezaměřuje na zanoření analýzy i do volaných funkcí. Jednak proto, že ne ke všem definicím lze získat přístup (např. funkce z extérních knihoven) a také analýza možného rekurzivního volání funkcí je nad rámec bakalářské práce. Místo toho tento nástroj považuje návratovou hodnotu z funkcí jako dodatečnou hledanou hodnotu, se kterou zachází stejně, jako při hledání argumentu funkce. Taková hodnota se pak stává dalším požadavkem na test (test requirement), kdy dané volání musí vracet vybranou hodnotu. Proto je nutné ukládat i argumenty, se kterými byly funkce volány.

4.3.1 Související moduly

SearchEngine.cpp a SearchEngine.h

`SearchEngine` je prostředník pro práci s CSP modelem pro generování vstupních argumentů. Po inicializaci objektu je metodou `printSolution` započat proces modelování. Metoda prochází základní bloky jednotlivých cest a předává modelu `GecodeTest` jednotlivé výrazy.

GecodeTest.cpp a GecodeTest.h

Jádro samotného generátoru. Implementuje navržené algoritmy a metodiky z kapitoly 4.4. Jeho výstupem jsou jednotlivá řešení pro danou cestu. Výrazy jsou analyzovány stejně jako v kapitole 4.2, ve kterých jsou přidávány nové proměnné do modelu a získávány již vytvořené proměnné pro modelování aritmetických výrazů a nerovnic.

4.4 4. fáze - Generování jednotkových testovacích případů

- **Vstup:** Vstupní argumenty jednotlivých cest a požadavky na test
- **Výstup:** Jednotkové testovací případy

Čtvrtá a poslední fáze je nejjednodušší. Cílem je vygenerovat jednotkové případy pro analyzované funkce. Testy jsou generovány do souboru s názvem ve formátu:
`dfta(číslo testu)_(název zdrojového souboru)_(idetifikátor testované funkce).c`.

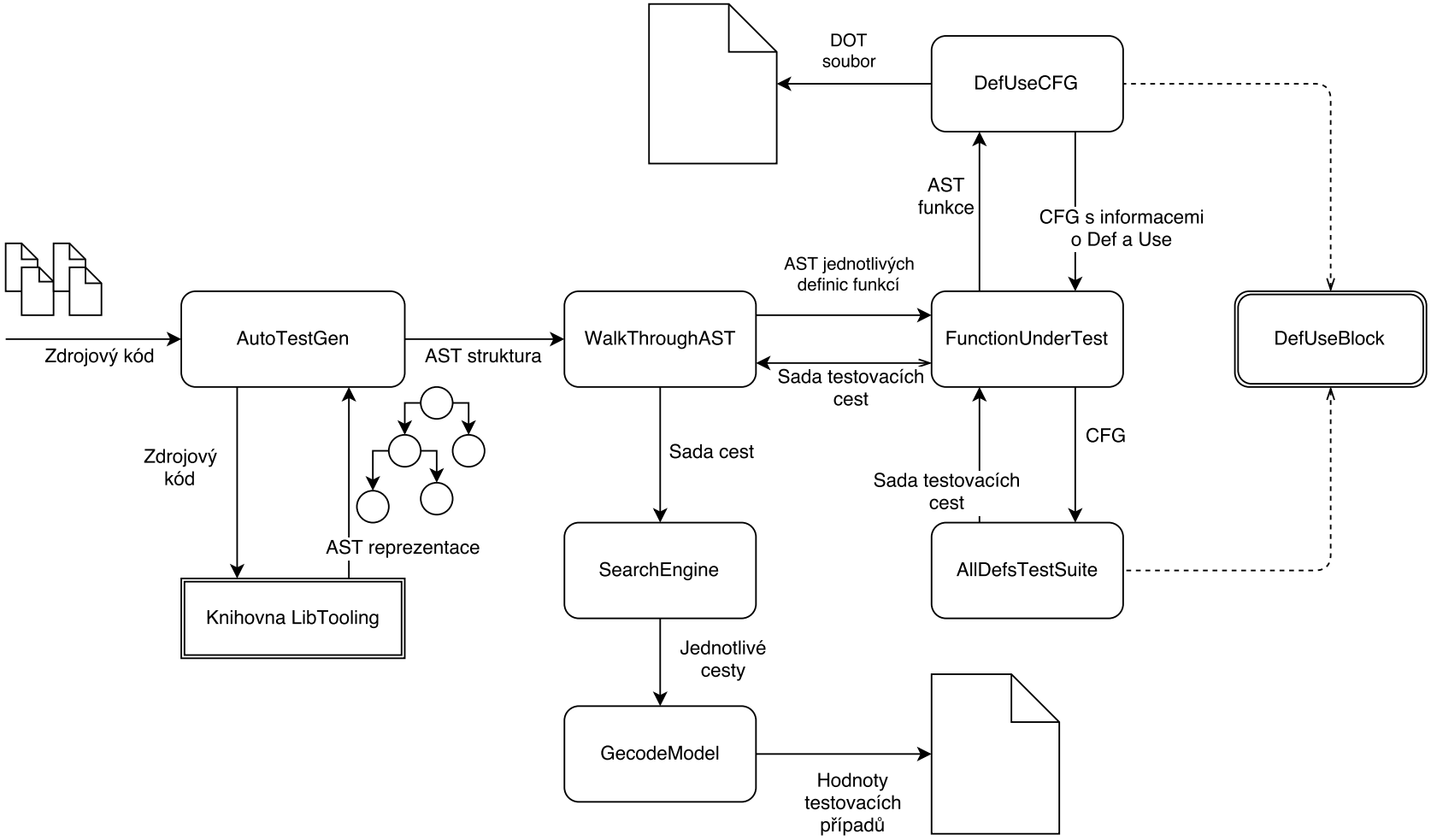
Do cílového souboru s testem mohou být vepsány až 4 různé položky. Nejdříve, pokud testovaná funkce pracuje se strukturami, jsou jejich definice zkopírovány na začátek souboru s testem. Informace kde tyto definice najít, se ukládají při analýze vytvářených proměnných. Následují požadavky na volání funkcí. Tento projekt nezachází do hloubkové analýzy zdrojového kódu, pouze vytváří dvojníky volaných funkcí, tzv. *stubs*. Každé volané funkci je vytvořena její replika s požadovanými návratovými hodnotami. Tyto repliky jsou vloženy do testu za definice heterogeních struktur. Následuje prosté zkopírování testované funkce, aby v testu mohla být patřičně zavolána. Následující místo v testu je vyplněno zapoznámkovanými informacemi ohledně procházených definic daným testem. Tyto informace slouží ke kontrole vytvořených testů pro splnění kritéria All-Defs. Nezbývá než vytvořit testovací funkci. Identifikátor funkce jednoho testu je ve tvaru `test(číslo testu)_(identifikátor funkce)()`. Uvnitř testu je zavolána testovací funkce a pod její volání je uživatel aplikace vyzván krátkou poznámkou ke vložení očekávaných výsledků testu.

4.4.1 Související moduly

SearchEngine.cpp a SearchEngine.h

Veškerá práce je prováděna ve funkci `printSolution` v modulu `SearchEngine`. Interaguje pouze s modulem `GecodeModel`, ze kterého si získá informace o nalezeném řešení a požadavích na teset. Tvoří test do souboru pomocí metod pro práci se soubory (pomocí streamů) ze standartní knihovny jazyka C++.

Grafická reprezentace běhu programu



Obrázek 4.4: Vstupem aplikace je zdrojový kód (vlevo). Aplikace se napojí na Clang pomocí Visitoru, získá AST reprezentaci, kterou dále analyzuje třídami z knihovny LibTooling. Jednotlivé funkce jsou reprezentovány třídou FunctionUnderTest, u které je nejdříve vytvořen CFG, poté aplikováno kritérium pokrytí. Sada cest této funkce jde do GecodeModel modulu, ze kterého získáme hodnoty pro jednotlivé testovací případy, ze kterých jsou následně vytvořeny testovací případy.

Kapitola 5

Generování testů

Následující testy byly vytvořeny pro základní demonstraci práce nástroje DFTA (Data Flow Test Automatization). Nejdříve jsou zaměřeny na konstrukce používané v jazyce C (5.1), poté na aritmetické operátory (5.2). Navazuje kapitola č. 5.3 demonstrující volání funkcí uvnitř testované funkce. Kapitola č. 5.4 analyzuje vytvořené cesty a posledním příklad ukazuje rychlost hledání a tvorb testů aplikací (5.5).

5.1 Testy konstrukcí jazyka C

Těhto 18 jednoduchých testů demonstruje zpracování základních konstrukcí vytvořeným nástrojem. Konkrétně se jednotlivé testy zaměřují na:

- test01.c – jednoduché spouštění nástroje bez konstrukcí
- test02.c–test04.c – if konstrukce
- test05.c–test06.c – if konstrukce, včetně větve else
- test07.c–test10.c – while konstrukce
- test11.c–test14.c – do–while konstrukce
- test15.c–test18.c – for konstrukce

Každá konstrukce je otestována vícekrát, pokaždé vytvořena lehce odlišně. Jsou postaveny samostatně (bez těla), poté s jednořádkovým tělem, složeným tělem a také vnořeny navzájem do sebe.

5.2 Testy aritmetických a porovnávacích operátorů

Testy test19.c–test35.c postupně zkoušejí podporované operátory. Nejprve jednoduché aritmetické operátory: sčítání, odečítání, násobení a dělení. Poté jsou ověřeny různé přiřazovací operátory. Nakonec jsou v podmíněných konstrukcích vyzkoušeny relační operátory.

5.3 Testy s vytvořením replik funkcí

K těmto testům patří soubory test36.c a test37.c, ve kterých je demonstrován princip vytváření zástupných replik volaných funkcí pro jednotkové testy. V prvním z nich je vytvořena a volána jedna funkce, ve druhém jsou vytvořeny dvě pomocné funkce, které se i volají navzájem.

5.4 Testy ověřující analýzu data-flow

Test 38 prochází kaskádou if konstrukcí pro dosažení všech jednotlivých definic. Test39.c krátce ukazuje nevyužitou definici – i když je proměnná *b* definovaná na 6. řádku, nelze ji spojit s žádným použitím. Test č. 40 mění post-inkrementaci na pre-inkrementaci a otáčí pořadí *def* a *use*. Díky této malé změně byly vytvořeny dva testy místo jednoho, které prochází obě větve konstrukce if. Test č. 41 končí hlášením: "Pro danou cestu nebylo nalezeno vhodné řešení", protože if obsahuje nesmyslnou podmínku.

5.5 Rychlost implementace

Test42.c obsahuje 26 aritmetických operátorů, 4 porovnání v blocích if, if-else, while i do-while a také jedno volání pomocné funkce. Byly vytvořeny 3 soubory s testy: jeden pro pomocnou funkci a dva testy pro stežejní funkci. Podle */usr/bin/time -v* aplikace běžela 0.03 vteřiny a použila 17kB paměti.

Kapitola 6

Závěr

Cílem mé bakalářské práce bylo nastudovat statickou analýzu kódu a testování datových toků a posléze navrhnout a implementovat nástroj pro automatické generování testovacích případů. Věřím, že jsem dostatečně vyjádřil získané poznatky z této problematiky. Aplikace byla naprogramována v jazyce C++ s využitím statické analýzy překladačového front-endu Clang a knihovny Gecode pro modelování CSP (Problémy splnitelnosti omezení), která tvoří jádro vyhledávacího algoritmu pro vhodné kandidáty hodnot.

Aplikace splňuje pouze základní požadavky a ještě není hotová pro nasazení do praxe. I přes nedostatky se podařilo propojit 2 složité knihovny a s jejich pomocí generovat celé jednotkové testy. Experimentálně s malými testy pracuej generátor rychle, ale jeho funkčnost nebyla vyzkoušena na velké sadě testovacích vstupů.

Pro mne měl projetk velký osobní přínos. Určitě bych rád pokračoval v rozvíjení projektu a opravení nedostatků, které sráží kvalitu nástroje.

Literatura

[1] Ammann, P.; Offutt, J.: *Introduction to Software Testing*. Cambridge University Press, 2008, ISBN 978-0-511-39330-3.

[2] Badlaney, J.; Ghatol, R.; Jadhwani, R.: *An Introduction to Data-Flow Testing*. [Online; navštíveno 20.7.2016].
URL ftp://ftp.ncsu.edu/pub/tech/2006/TR-2006-22.pdf

[3] Brian Marick: *The Craft Of Software Testing, Subsystem Testing*. Prentice Hall PTR, 1995, ISBN 0-13-177411-5.

[4] Claessen, K.; Hughes, J.: *QuickCheck: An automatic testing tool for Haskell*. [Online; navštíveno 20.7.2016].
URL http://www.cse.chalmers.se/~rjmh/QuickCheck/

[5] Dr., Ing. Norbert Oster: *Automatic Dataflow-oriented Test Case Generation for Object-oriented Software Systems by Evolutionary Algorithms*. [Online; navštíveno 20.7.2016].
URL https://www2.informatik.uni-erlangen.de/EN/staff/oster/DOTgEAr/index.html

[6] Generovaná reference nástrojem Doxygen: *clang Namespace reference*. [Online; navštíveno 17.5.2016].
URL http://clang.llvm.org/doxygen/namespaceclang.html

[7] Godefroid, P.; Klarlund, N.; Sen, K.: *DART: Directed Automated Random Testing*. Technická zpráva, Bell Laboratories and University of Illinois, 2005.
URL https://en.wikipedia.org/wiki/Software_testing

[8] Ma, L.; Artho†, C.; Zhang, C.; aj.: *GRT at the SBST 2015 Tool Competition*. [Online; navštíveno 20.7.2016].
URL https://staff.aist.go.jp/c.artho/papers/sbst-2015.pdf

[9] Margaret Rouse: *fuzz testing (fuzzing)*. [Online; navštíveno 20.7.2016].
URL http://searchsecurity.techtarget.com/definition/fuzz-testing

[10] Microsoft research: *SAGE: Whitebox Fuzzing for Security Testing*. [Online; navštíveno 20.7.2016].
URL http://research.microsoft.com/en-us/um/people/pg/public_psfiles/sage-in-one-slide.pdf

[11] Nielson, F.; ands Chris Hankin, H. R. N.: *Principles of Program Analysis*. Springer Science and Business Media, 2005, ISBN 3-540-65410-0.

[12] Putcell, S.: *PyUnit - the standard unit testing framework for Python.* [Online; navštíveno 17.5.2016].
URL http://pyunit.sourceforge.net/

[13] Rojas, J. M.; Fraser, G.; Arcuri, A.: *Seeding strategies in search-based unit test generation.* Technická zpráva, Department of Computer Science, University of Sheffield and Scienta, Norway, and SnT Centre, University of Luxembourg, [Online; navštíveno 20.7.2016].
URL
http://www.evosuite.org/wp-content/papercite-data/pdf/stvr_seeding.pdf

[14] Schulte, C.; Tack, G.; Lagerkvist, M. Z.: *clang: a C language family frontend for LLVM.* [online; navštíveno 17.5.2016] vydání.
URL http://clang.llvm.org/

[15] The Clang team: *LibTooling - Clang 3.9 documentation.* [Online; navštíveno 17.5.2016].
URL http://clang.llvm.org/docs/LibTooling.html

[16] The KLEE team: *KLEE LLVM Execution Engine.* [Online; navštíveno 17.5.2016].
URL http://klee.github.io/

[17] Tomáš hlava: *Progresní a regresní testy.* [Online; navštíveno 20.7.2016].
URL http://testovanisoftwaru.cz/tag/regresni-testy/

[18] tým eXtremeSoftwareTesting.com: *Random Software Testing Techniques.* [Online; navštíveno 20.7.2016].
URL http://extremesoftwaretesting.com/Techniques/RandomTesting.html

[19] Wikipedia: *Abstract Syntax Tree.* [Online; navštíveno 17.5.2016].
URL https://en.wikipedia.org/wiki/Abstract_syntax_tree

[20] Wikipedia: *Control flow graph.* [Online; navštíveno 17.5.2016].
URL https://en.wikipedia.org/wiki/Control_flow_graph

[21] Wikipedia: *Software testing.* [Online; navštíveno 17.5.2016].
URL https://en.wikipedia.org/wiki/Software_testing

[22] WWW stránka: *cppunit test framework.* [Online; navštíveno 17.5.2016].
URL https://freedesktop.org/wiki/Software/cppunit/

[23] WWW stránka: *JUnit - About.* [Online; navštíveno 17.5.2016].
URL http://junit.org/junit4/

[24] WWW stránky: *Constraint Satisfaction Problem.* [Online; navštíveno 17.5.2016].
URL https://en.wikipedia.org/wiki/Constraint_satisfaction_problem

[25] WWW stránky: *The LLVM Compiler Infrastructure.* [online; navštíveno 17.5.2016] vydání.
URL http://www.llvm.org/

[26] WWW stránky projektu: *CREST - Concolic test generation tool for C.* [Online; navštíveno 20.7.2016].
URL http://www.burn.im/crest/

[27] WWW stránky projektu: *EvoSuite - Automatic Test Suite Generation for Java*. [Online; navštíveno 20.7.2016].
URL http://www.evosuite.org/evosuite/

[28] WWW stránky projektu: *GECODE - An open, free, efficient constraint solving toolkit*. [Online; navštíveno 17.5.2016].
URL http://www.gecode.org/

[29] WWW stránky projektu: *MiniZinc: Challenge*. [Online; navštíveno 20.7.2016].
URL http://www.minizinc.org/challenge.html

[30] WWW stránky projektu: *Randoop*. [Online; navštíveno 20.7.2016].
URL https://randoop.github.io/randoop/

[31] WWW stránky projektu: *Selenium*. [Online; navštíveno 20.7.2016].
URL http://docs.seleniumhq.org/

Přílohy

Seznam příloh

Příloha A

Obsah CD

- **/usr** - zdrojové soubory aplikace
- **/doc** - elektronická verze bakalářské práce
- **/lib** - extérní knihovny nutné pro chod aplikace
- **/tests** - demonstrační testy
- **README.txt** - uživatelská příručka
- **Makefile** - skript pro přeložení aplikace
- **LICENSE.txt** - textový soubor s licencemi použitých knihoven