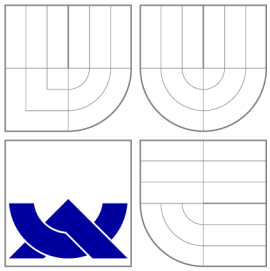
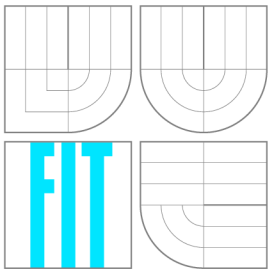


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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
18. května 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 má velmi velký význam v oblasti vývoje software. Svědčí o tom jednak množství prostředků vynaložené 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 s širokou škálou nabízených nástrojů pro řešení obecných či konkrétních problémů. Jedním z nich je i problém automatizace. Testování, a pokusy o jeho automatizaci, se potýká s mnoha překážkami, z nichž některé mohou být i neřešitelné, či o jejich řešitelnosti zatím nemáme formální důkaz. Různé aplikace a frameworky k těmto problémům přistupují z různých pohledů na věc a poskytují různé úrovně i oblasti úspěšnosti.

Programovací jazyk C, i přes svou nízko-úrovňovost a slabou úroveň abstrakce, je stále velmi rozšířený. Díky své nízké úrovni si našel 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 důležitosti tohoto programovacího jazyka je i fakt, že je vyučován jako první jazyk na mnoha školách se zaměřením na informační technologie.

Tento projekt si klade za cíl vytvořit nástroj, který statickou analýzou zdrojového kódu v jazyce C vygeneruje návrhy vstupů pro testovací případy. Uživatelem aplikace je tester, nebo dokonce autor programu, který má zdrojový text k dispozici. Od této aplikace očekává, že mu nabídne sadu vstupních dat takovou, která dostatečně otestuje co největší část zdrojového kódu. 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 základů jazyka C. Na to navazuje část 4 rozebírající návrh a implementaci aplikace a hlavní část končí 5 částí prezentující generované hodnoty nad testovacími ukázkami.

Kapitola 2

Testování softwaru a jeho automatizace

Obsahem této části dokumentu je rychlý 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. Celou část uzavírá kapitola 2.3 o problematice automatizace testování a krátký přehled dostupných řešení.

2.1 Úvod do testování softwaru

Oblast testování softwaru je v počítačovém inženýrství, pro jeho velikost, samostatným odvětvím v životním cyklu vývoje softwaru. Jeho počátky sahají mnoho let zpátky k počátkům rozvoje programování jako takového. Během těchto let se v tomto odvětví vyvinulo spoustu metod a přístupů, jak k testovanému subjektu, označovaném jako *Testovaný systém*, zkr. **SUT** (z angl. - System under test), přistupovat a také množství členění, jak různé testovací metody od sebe rozlišit. Široký výběr těchto metod a přístupů ukazuje, že žádný z nich nelze považovat za "*správný*" (i když toto je námětem mnoha diskuzí), ale každý z nich se hodí pro odlišný problém či dosahuje jiných výsledků z daného úhlu pohledu.

Jedním ze základním rozdělení testování je dle toho, jaký máme na SUT náhled [10]:

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

Černá skříňka označuje naprosto uzavřený systém, do kterého nemáme přístup. V takovém případě můžeme pouze posílat data na vstup a analyzovat výstup černé skříňky. Příkladem může být vizuální testování například GUI mobilní aplikace. Naproti tomu bílá skříňka je úplný opak. U takového systému máme plný přístup ke zdrojovým textům, které využíváme ke tvorbě jednotlivých testů. Tento přístup může poskytnou lepší pokrytí všech potenciálních problémů. S pojmem šedá skříňka se 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.

Tato práce analyzuje zdrojový kód, naším systémem je tedy bílá skříňka. Dále můžeme náš přístup klasifikovat, jakým způsobem kód analyzuje, na statickou a dynamickou analýzu kódu. Jak z názvu vypovídá, u dynamické analýzy běží samotný program a testy reagují,

námi specifikovaným způsobem, na jeho výstupy. Tento projekt se ovšem specifikuje na statickou analýzu kódu a tedy konkrétní řešení úzce souvisí se zvoleným programovacím jazykem.

2.2 Analýza datových toků

V programování vždy provádíme operace nad daty. Zvolenou abstrakcí si 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 vytvořeny a na jiném by měly být správně použity, sledujeme *datové toky*. Právě tato místa jsou podstatou naší analýzy. Místo, kde je proměnná vytvořena je její hodnota uložena do paměti a takovému místu říkáme *definice*, zkráceně *def*. Použití této proměnné označuje místo, kde je její hodnoty z paměti přečtena k dalšímu zpracování a je tedy místem *použití*, anglicky *use*. Pro určitou proměnnou x vytváříme *def-use* páry nebo jen *du páry* a 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 zůstanou nám pouze *čisté def cesty*.

Tester ke hledání cest nepřistupuje náhodným výběrem. Za roky vývoje testovacích technik byly vytvořeny metody, jak j vytváření testů přistupovat systematicky, aby se zvýšila jejich účinnost. *Kritérium pokrytí (z angl. - coverage criterion)* je pravidlo nebo předpis pro systematické generování požadavků na test. *Pokrytí (z angl. - coverage)* je míra udávající, jak moc daná testovací sada zkoumá SUT. Testovací sada je soubor testů. Pro datové toky byly definovány 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 čtením

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 jsou napravo u příslušných řádků vyznačeny *def* a *use* pro všechny proměnné. Programátor analyzující tuto funkci hledá def-use cesty pro všechny možné proměnné dle daného *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. 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 příliš obtížný úkol.

2.3 Automatizace testování

Důležitost testování softwaru není potřeba zvlášť vyzdvihovat. Chyby v programech mohou stát jejich vydavatele značný obnos peněz, či v horších případech někoho stojí život. Testování je, stejně jako programování, mentální disciplína a lidé dělají chyby. O automatizaci testování je velký zájem a aplikacím se daří v této disciplíně dosáhnout různých výsledků na rozlišných úrovních. Tato práce se zaměřuje na automatizaci tvorby testovacích případů. Nejprve si definujme, z čeho se skládá **testovací případ (test case)**:

- 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

Předpokládejme že uživatel naší aplikace si zajistí prefixové a postfixové hodnoty sám. Rozdělíme-li si zdrojový text na funkce, můžeme je samostatně otestovat jejich voláním s patřičnými hodnotami testovacího případu a kontrolou jejich očekávaného výsledku. Kdybychom tuto část dělali manuálně, vybrali bychom si kritérium, analyzovali kód a vytvořili testovací sadu tak aby cesty programem dané kritérium splňovaly. Z obecného pohledu - pohledu automatického nástroje - nejsme schopni určit jaký má funkce sémantický účel a tedy nevíme očekávaný výsledek. Tento dokument popisuje analýzu a implementaci nástroje, na generování hodnot testovacích případů pro splnění daného kritéria. K jeho řešení byl použit princip analýzy datových toků, kdy sledováním vytváření, použití a omezení proměnných pomocí CSP solveru určíme jejich možné hodnoty. Tento přístup může odhalit nejen chyby programátora vedoucí k pádu programu, ale i ty, které pouze způsobují nekorektní stavy, které se mohou projevit různými způsoby. V testování se tyto pojmy definují jako [10]:

- **vada (fault)** - statický defekt v software
- **chyba (error)** - špatný/nekorektní vnitřní stav systému, který je projevem nějakého defektu
- **selhání (failure)** - externí projev nesprávného chování

Existuje mnoho nástrojů, které směřují stejným směrem jako tato práce či přistupují k automatizaci testování jiným způsobem. Toto jsou některé z nich.

2.3.1 KLEE

KLEE LLVM Execution Engine [7] je open source nástroj pro automatické generování testů bez další asistence. Nápad je podobný - také sleduje proměnné a uchovává si informace o jejich aktuálním stavu a omezení a používá CSP solver, ale provedení je jiné. Namísto generování pouze hodnot vstupů, KLEE převezme celý proces testování. Pro daný program vyzkouší co nejvíce vstupních hodnot a následně program s nimi spouští, dokud nedojde k pádu programu, což je vyhodnoceno jako neúspěšný test. Takový přístup vykazuje velmi vysoké hodnoty pokrytí a KLEE není na poli těchto nástrojů nováčkem, ale naopak silný a optimalizovaný nástroj. Na druhou stranu pravděpodobně není příliš vhodný pro rozsáhlé či rychle se rozvíjející projekty, kde by vývoj zpomaloval, kvůli nárůstu počtu testů.

2.3.2 DART

DART: Directed Automated Random Testing [3] využívá opět podobný princip. Statickým analyzátorem nejdříve zjistí vstupní hodnoty, poté pro ně zvolí náhodné hodnoty s kterými začne testovat a v omezujících tvrzeních programu generuje další nové hodnoty pro další test, dokud neprojde všechny možné cesty programu, či neskončí test chybou.

2.3.3 *x*Unit

Nástroj nemá přímo název **xUnit**, ale prefixové *x* lze nahradit mnoha zkratkami pro programovací jazyky, např.: **JUnit** pro Javu [12], **CPPUnit** pro C++[11], **PyUnit** pro Python [4]. Jde tedy o označení rodiny frameworků, určených pro vytváření Unit (překládáno jako Jednotkových) testů. Jde o rozdílné nástroje oproti předchozím dvěma či mnou vytvořena aplikace, protože negenerují části testů. Pouze nabízejí API pro velmi jednoduchou přípravu daného prostředí a definici jednotlivých testovacích případů. I přesto určitě stojí za zmínku, protože šetří práci testerů z jiného úhlu pohledu a často jsou referovány jako automatické nástroje pro "tvorbu" testů, i když negenerují samotné hodnoty. Integrace některého z frameworků s generátorem testovacích případů je rozhodně dobrý námět pro další rozvoj mé práce.

Kapitola 3

Jazyk C a použité nástroje

Tato část bakalářské práce popisuje zvolený jazyk a dostupné překladače (3.2). Kapitola 3.2.1 seznamuje čtenáře s pojmem AST a postupovat při jeho vytváření či analýze. Poslední částí je představení konstrukcí jazyka C 3.3 a knihovny pro jejich analýzu 3.2.2 a knihovnu pro zpracování jeho datových typů (3.5).

3.1 Výběr jazyka

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. Dalšími důvody, proč byl vybrán právě jazyk C je, jsou mé osobní zkušenosti s ním a nabídka nástrojů a knihoven pro jeho analýzu. V průběhu práce s analyzátorem jazyka C jsem nabyl spoustu nových poznatků o jeho syntaxi. Jedním z příkladů může být například možný zápis indexace pole, kde *a[4]* i *4[a]* je výraz pro výběr prvku na indexu 4 z proměnné *a* typu pole.

3.2 LLVM a Clang

LLVM [13] 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 byla zkratkou pro Low Level Virtual Machine (nízko-úrovňový virtuální stroj - volně přeloženo), ale postupem času se z něj stal projekt zastřešující mnoho překladačových technologií, takže pouze o virtuální stroj se již nejedná. 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 vlastně 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 právě pro rozhraní LLVM, které jsou vyvíjeny pod záštitou projektu LLVM. Pro jazyk C jde o front-end Clang.

Clang [5] 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 (např.: vstupní argumenty). Projekt Clang také zahrnuje statický analyzátor a poskytuje rozhraní pro používání a vytváření nových analytických nástrojů. Díky tomu byl nejlepším kandidátem pro použití v mém projektu a na svou stranu přispěl i dobrou dokumentací.

3.2.1 AST

Abstraktní syntaktický strom (z angl. - Abstract syntax tree) [8], zkráceně **AST** je stromová reprezentace abstraktní syntaktické struktury zdrojového kódu. Syntaxe je abstraktní v tom smyslu, že ne všechny detaily musí 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 a jak vypadá jeho grafická reprezentace ve stromové struktuře a jak vypadá jeho textová podoba vypsaná pomocí Clang:

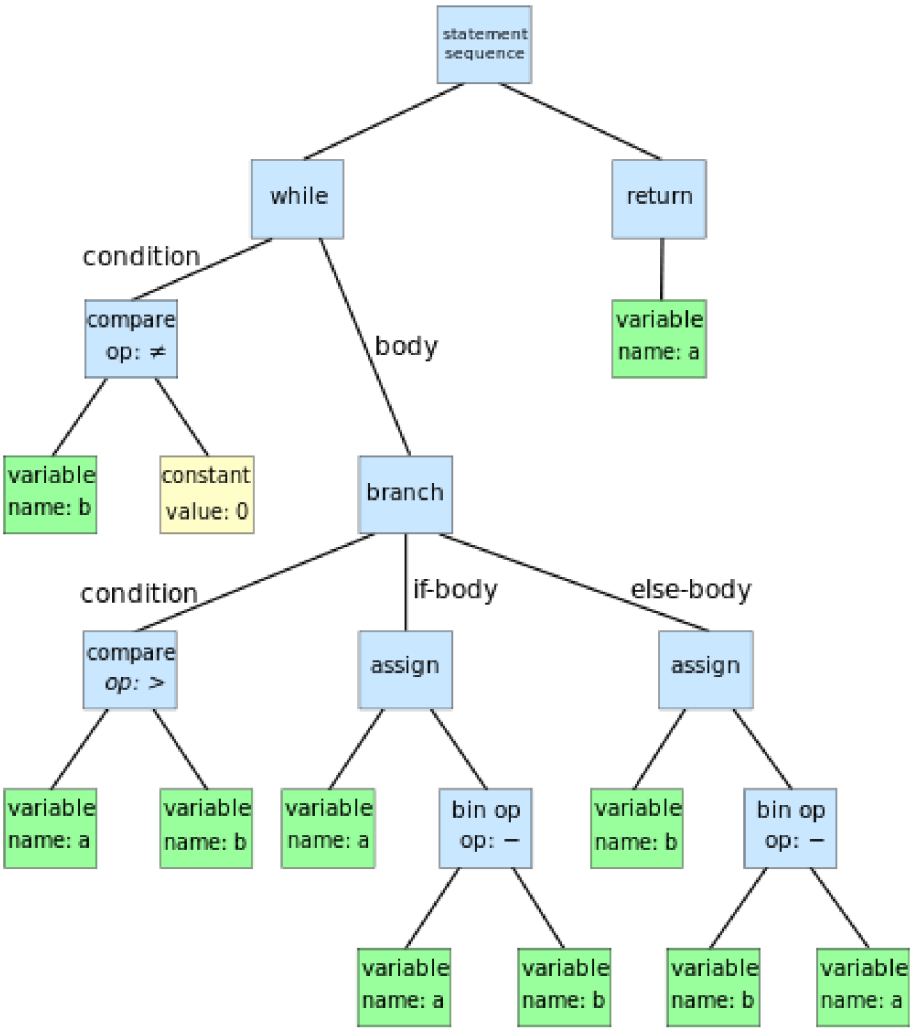
```
1 int a = 5, b = 7;
2 while (b != 0) {
3   if (a > b)
4     a = a - b;
5   else
6     b = b - a;
7 }
8 return a
```

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 [2].

```
-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>
|-<<<NULL>>>
|-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>
    |-<<<NULL>>>
    |-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'
    |-BinaryOperator 0xa8de1b0 <line:10:3, col:11> 'int' '='
    | |-DeclRefExpr 0xa8de0e0 <col:3> 'int' lvalue Var 0xa8dde70 'a' 'int'
    | `-BinaryOperator 0xa8de188 <col:7, col:11> 'int' '-'
    |   |-ImplicitCastExpr 0xa8de158 <col:7> 'int' <LValueToRValue>
    |   | `-DeclRefExpr 0xa8de108 <col:7> 'int' lvalue Var 0xa8dde70 'a' 'int
    |   `-ImplicitCastExpr 0xa8de170 <col:11> 'int' <LValueToRValue>
    |     `-DeclRefExpr 0xa8de130 <col:11> 'int' lvalue Var 0xa8ddf00 'b' 'int
    `-BinaryOperator 0xa8de2a8 <line:12:3, col:11> 'int' '='
      |-DeclRefExpr 0xa8de1d8 <col:3> 'int' lvalue Var 0xa8ddf00 'b' 'int'
      `-BinaryOperator 0xa8de280 <col:7, col:11> 'int' '-'
        |-ImplicitCastExpr 0xa8de250 <col:7> 'int' <LValueToRValue>
        | `-DeclRefExpr 0xa8de200 <col:7> 'int' lvalue Var 0xa8ddf00 'b' 'int
        `-ImplicitCastExpr 0xa8de268 <col:11> 'int' <LValueToRValue>
          `-DeclRefExpr 0xa8de228 <col:11> 'int' lvalue Var 0xa8dde70 'a' '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 [8]

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 **Clang Plugin**, která 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 [6] je C++ rozhraní určené pro programování samostatných nástrojů, využívajících technologie Clangu a nabízí plnou a nezávislou kontrolu nad AST.

LibTooling využívá modulu RecursiveASTVisitor, který projde AST před či po (dle našich požadavků) překladu a optimalizacích a dovolí nám se "zastavit" nad námi specifikovanými entitami, projít do potřebných uzlů AST a dělat analýzu. Dva základní prvky Clang AST jsou Stmt (tvrzení) a Decl (deklarace). Expr (výrazy) jsou ve Clangu považo-

vány za Sodnož Stmt.

3.3 Konstrukce jazyka C

V této kapitole si ukážeme konstrukce a výrazy, které tvoří syntaxi jazyka C a jejich reprezentaci ve formě abstraktního syntaktického stromu.

3.3.1 If a if-else

If je rozhodovací řídící konstrukce s povinnou rozhodovací podmínkou a *else* je její nepovinná část. Podmínka určuje, výlučně která z větví, buď *if* nebo *else*, bude provedena a to je-li podmínka pravdivá či nepravdivá. Ve Clang AST je tato konstrukce pojmenována *IfStmt* a obsahuje odkaz na tělo podmínky, větev těla if a větev těla else. Navíc Clang AST poskytuje odkaz na tzv. *Condition Variable (podmínkovou proměnnou)*, která může být deklarována v těle podmínky a slouží k její vyhodnocení. Tento odkaz, stejně jako odkaz na else, může být nulový (NULL), není-li tato část v kódu přítomna.

3.3.2 While a do-while

While i *do-while* jsou obě konstrukce cyklu, který se provádí tolikrát, dokud je pokaždé splněna rozhodovací podmínka, tedy i s možností nekonečného cyklu. Jediný rozdíl mezi nimi je ten, že u klasického *while* se podmínka kontroluje na začátku, oproti *do-while* kde se testuje na konci. Do AST se transformují každá svojí reprezentací, *WhileStmt* a *DoStmt*, obě obsahující odkazy na podmínku, tělo cyklu a opět *condition variable*.

3.3.3 For cyklus

For je opět konstrukce cyklu a každý *for* je převeditelný na *while* i obráceně. For vznikl s důrazem právě na onu *condition variable*, která explicitně vyjadřuje, že cyklus bude proveden určitý počet krát. Tento účel se odboural s vývojem jazyka, protože do podmínek všech cyklů, již můžeme umístit libovolné podmínky. *For* cyklus uzavírá navíc extra výraz, kde se obvykle zvyšuje počítadlo vyjádřené podmínkovou proměnnou. *For* v AST obsahuje odkazy na všechny zmíněné části, včetně uzavíracího výrazu.

3.3.4 Switch a case

Switch je rozhodovací konstrukce představující přepínač s proměnným počtem možných větví. Podmínkou nyní již není výraz, ale pouze jedna proměnná celočíselného datového typu. Každá větev je definována jako *case* s konkrétní hodnotou rozhodovací proměnné, která určuje požadovaný celočíselný stav pro výběr právě této větve. Clang definuje tyto tvrzení jako *SwitchStmt* a *CaseStmt*. *SwitchStmt* obsahuje odkaz na podmínku, *condition variable* a tělo či pole *CaseStmt*. Každý *CaseStmt* pak obsahuje odkaz na podmínkovou hodnotu, na tělo a na další *CaseStmt* v pořadí. *DefaultStmt* je speciální případ *CaseStmt* označující případ, kdy podmínková proměnná nenabyla ani jeden ze stavů definovaných v *case*-ech.

3.3.5 Break a continue

Break je konstrukce používaná k okamžitému opuštění právě prováděného cyklu nebo přepínače. Pochopitelně tedy nelze použít mimo jakýkoliv cyklus či přepínač, například k ukončení funkce. *Break* zaručuje skok programu právě za tělo konstrukce, ve které *break* použijeme, bez vyhodnocení jakékoliv podmínky. Oproti tomu *continue* je tvrzení uvnitř cyklu, zaručující odskočení na příkaz vykonání podmínky, kde jsou akce prováděny dle aktuálního stavu. *Continue* nelze použít uvnitř přepínače. Příslušne jsou definovány *BreakStmt* a *ContinueStmt*, které neobsahují žádné odkazy na cykly, ke kterým se vztahují.

3.3.6 Deklarace

Po *Stmt* je *Decl* druhou základní jednotou v Clang AST, představující deklarace, nebo spíše rodinu deklarací, které dědí vlastnosti z matčinné *Decl* (stejný princip jako u výše uvedených *Stmt*). Pro jazyk C jsou nejdůležitější *VarDecl* označující jakoukoliv deklaraci proměnnné, či proměnných jsou-li na jednom řádku ve zdrojovém kódu, a obsahující odkaz na její typ. Dalším příkladem je *TypedefDecl* pro vytváření nových typů klíčovým slovem *typedef*, opět obsahující funkce na získání původního typu. Dále se často setkáme s *RecordDecl* určující deklaraci či definici nových struktur odkazující odkazy na *RecordDecl*, což jsou deklarace zastupující jednotlivé elementy.

3.4 CSP a datové typy

Jak bylo naznačeno v kapitole 2, 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. 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 nějakou modifikací těchto technik. Dnešní CSP solvery používají formu Constraint modelování pro vytváření modelů popisujících daný CSP. Při určitého solveru byl kladen důraz na možnost spolupráce s knihovnou LibTooling, která je napsána v jazyce C++ a na kvalitu dokumentace, jelikož jsem s tímto problémem neměl zkušenosti.

3.5 Gecode

Gecode [14] 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 vyhrál ve všech kategoriích na MiniZinc Challenge, což je každoroční soutěž CSP solverů v různých testech. V posledních soutěžích byl poražen konkurencí. 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.

3.5.1 Celočíselné datové typy

V jazyce C mezi ty to 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, příslušná IntVar proměnná je zdola omezena nerovností 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 výt 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 IntVary. 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 *rel()* (relace), očekávající na vstup dvě proměnné typu IntVar a *enum* hodnotu určující typ relace - IRT_EQ (=), IRT_NQ (≠) IRT_LE (<), IRT_LQ (≤), IRT_GR (>) nebo IRT_GQ (≥). 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.

3.5.2 Datové typy s plovoucí desetinnou čárkou

Standardně se pro tyto účely používají proměnné typu float, double a long double s vzrůstající přesností reprezentace desetinných čísel. Gecode nabízí opět jediný typ *FloatVar*, který je ovšem oproti omzení pro velká celá čísla ukládán tak, aby dosáhl co největší přesnosti. Gecode používá intervalovou aritmetiku - hodnota není uložena jako jediné číslo, ale jako dvojice, omezující hodnotu shora i zdola. Tím se snaží zvýšit přesnost při častém zaokrouhlování v aritmetických operacích. FloatVar musí zase být inicializován než bude použit ve výpočtech a jeho rozmezí je definováno ve Float/limits.hpp jako Float::Limits::min a Float::Limits:max, jejich přesná hodnota není známa. Opět je k dispozici možnost vytvoření polí FloatVarArray a operací nad nimi. Zkoušením modelování 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 standardně limit nastaven od -100.0 do 100.0. Pokud uživatel chce zvýšit tento limit, musí spustit nástroj s argumentem *-float <limit>*, kde *limit* představuje číselnou hodnotu.

3.5.3 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, tak nám nezbývá než zařadit položky, které jsou jednoduchého datového typu mezi ostatní proměnné, a uchovávat si informace o jejich umístění v modelovaném systému.

3.5.4 Pole a ukazatele

V jazyce C mají tyto dva typy, složený - pole a adresový - ukazatel, velmi úzký vztah. Rozdíl nalezneme například v nemožnosti zjistit velikost adresované paměti v bytech přes ukazatel s pomocí operátoru *sizeof*, na rozdíl od zjištění velikosti pole. Tento vztah nás ovšem v modelování nijak nezajímá. Příkladem zaměnitelnosti ukazatele za pole je například syntaxe pro přístup k prvku pole přes operátor “hranaté závorky” *-x[i]*, který je pouze syntaktický cukr pro výraz v ukazatelové aritmetice **(x + i)*. Ukazatele obecně představují, obzvlášť pro začínající programátory, Achillovu patu, protože jazyk C je poměrně benevolentní k akcím, které ho donutíme dělat. Bez automatické kontroly přístupu do paměti, která vlastně není naše, je velmi snadné přepsat místo, kde například byl uložen program běžící aplikace a napáchat tím značné škody. Takové chyby se v rozsáhlých projektech hůře hledají, ale nelze vinit programovací jazyk, který byl k těmto účelům navržen.

Kapitola 4

Návrh a implementace

4.1 Tvorba CFG

Control Flow Graph, zkratka **CFG** [9], je jednou z možných reprezentací zdrojového kódu. V kapitole 3.2.1 jsme si již představili podobu AST a v kapitole 2.2 ukázali, jak nalézt definice a užití proměnných. Velice rychle bychom zjistili, že psát *def(x)* a *use(x)* ke každému řádku kódu je zbytečné a reprezentace AST se k tomuto účelu také příliš nehodí. 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 proveden po sobě ve stejném pořadí. V jazyce C tok programu ovlivňují řídící konstrukce představené v kapitole 3.3. Všechny ostatní výrazy, společně s deklaracemi můžeme tedy sdružit do základních bloků.

Při analýze AST, který vygeneruje Clang, procházíme jeho stromovou strukturu a vytváříme vlastní CFG. Narazíme-li na řídící konstrukci, vytvoříme rozvětvení a pokračujeme do příslušných uzlů. Jak je AST strukturován je naznačeno v kapitole 3.2.1, pro jeho detailnější analýzu je nejlepším zdrojem informací dokumentace. Každý uzel v sobě nese tyto informace:

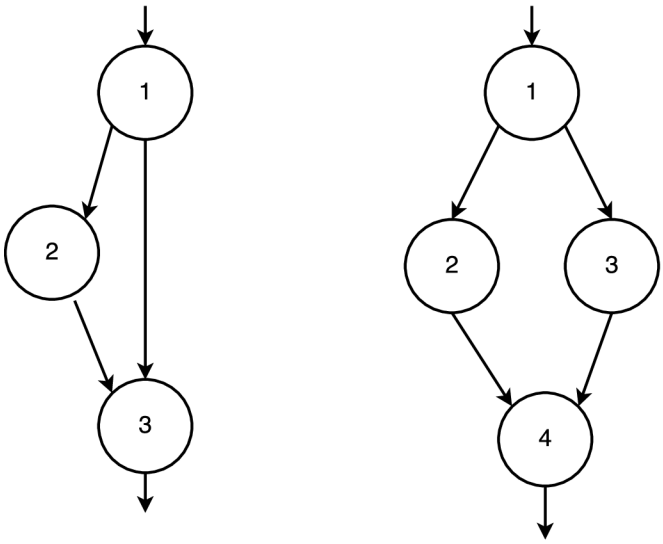
- pole ukazatelů na navazující uzly
- výraz podmínky, či pole výrazů (pro přepínač)
- pole všech navštívených výrazů, v prováděném pořadí
- pole všech def() a use() pro daný základní blok, v prováděném pořadí
- jednoznačný identifikátor uzlu

Identifikátorem může být unikátní číselná hodnota, nejlépe začínající od 0 v kořenovém uzlu, aby se pak tato hodnota mohla použít pro vykreslení očíslovaného grafu. Každý výraz, na který narazíme přidáme na konec pole výrazů. Využijeme toho, že již procházíme AST a jeho výrazy a každý z nich analyzujeme pro algoritmem pro vyhledání všech def() a use(). Takový algoritmus může být implementován rekurzivním způsobem, kde každá operace znamená nové volání funkce pro všechny strany výrazu, ať už unární, binární či ternární a nalezení proměnné znamená přidání use pro danou proměnou. Výjimkami jsou výrazy přiřazení, kdy nejdříve projdeme pravou stranu výrazu pro use a poté levou pro def a unární výraz inkrementace a dekrementace, který pro danou proměnou znamená přidání 1. def a

2. use pro prefixový operand, nebo 1. use a 2. def pro postfixový operand. Když dojdeme na řídící konstrukci, vytvoříme nové uzly, jejichž ukazatele uložíme do pole ukazatelů a rekurzivním způsobem analyzujeme těla řídících konstrukcí. Implementujeme-li vytvoření CFG rekurzivním způsobem, mohou vzniknout prázdné uzly, například při *while(true) {}* a také redundantní přechody u konstrukcí jako *break* a *continue*. Těm můžeme předejít neustálou kontrolou při vytváření nových uzlů, nebo je vymazat po vytvoření stromu. Celá funkce je tedy sada *if*-výrazů nebo jeden velký *switch* kontrolující, v jakém typu *Stmt* právě jsme. U podmíněných řídících konstrukcí přidáme i podmínku mimo pole všech výrazů.

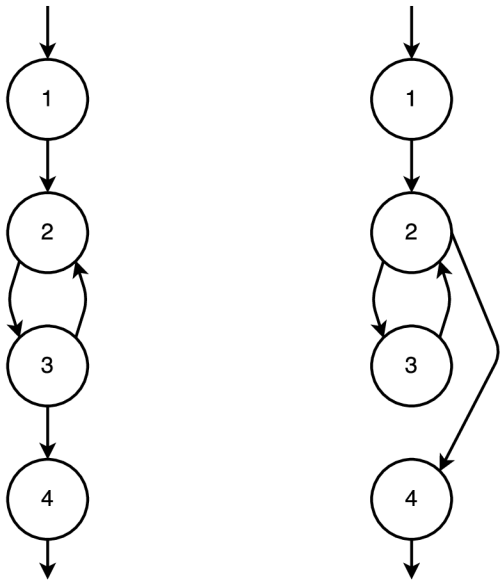
4.1.1 Reprezentace řídících konstrukcí jazyka C v CFG

Podmíněná konstrukce If-than a If-than-else



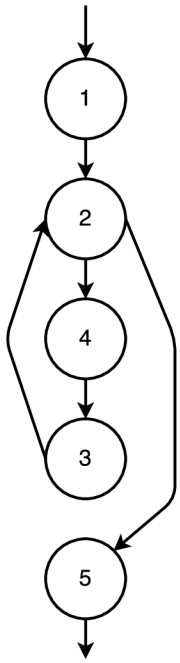
Obrázek 4.1: Vlevo if-then a napravo if-then-else

Cykly While a Do-while



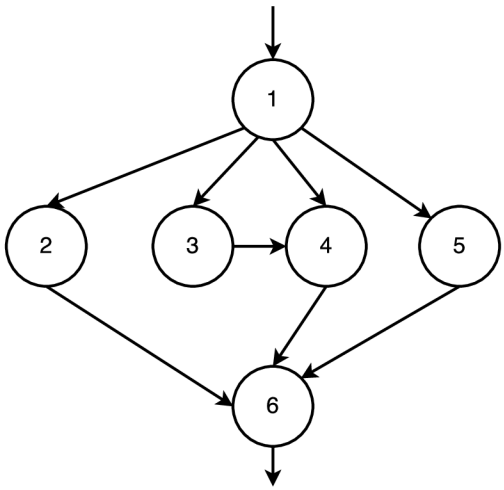
Obrázek 4.2: Vlevo do cyklus a napravo cyklus do-while

For cyklus



Obrázek 4.3: For cyklus

Přepínač switch



Obrázek 4.4: Jedna z možných variant přepínače Switch

4.2 Výběr cest k testování

V kapitole **??** jsme si představili 3 různá kritéria pokrytí. Které z nich si vybereme nijak neovlivní další vývoj aplikace, pouze vytvoří více testovacích případů. Já osobně jsem zvolil ADC, tedy pokrytí všech definic. Nalezení vhodných cest můžeme vyřešit algoritmem, který nalezne všechny možné cesty, z nichž každou se pokusí přidat do seznamu všech cest. Ta bude přidána jenom v tom případě, že dosáhla takové def-use cesty, kde pro proměnou x v místě definice *def(x, m)*, ještě nemá v seznamu def-use zastoupení. Hledání všech cest opět můžeme implementovat rekurzivním algoritmem:

- 1) začneme v kořenovém uzlu
- 2) uložíme si identifikátor uzlu
- 3) projdeme pole definic a použití:
- 3.a) najdeme-li def, zařadíme jej jako aktuální def do mapy, kde klíčem je unikátní identifikátor proměnné, společně s identifikátorem uzlu
- 3.b) najdeme-li use, vyhledáme dle identifikátoru proměnné aktuální def, ze kterého použijeme identifikátor uzlu a identifikátor proměnné do pole def-use cest
- 4) projdeme pole odkazů na navazující uzly
- 4.a) je-li pole prázdné, přidáme aktuální cestu společně s polem def-use cest do seznamu nalezených cest
- 4.b) je-li pole neprázdné, rozgenerujeme každý z uzlů jako následující v cestě (společně s polem aktuální cesty, nalezených def-use cest a mapy proměnná-def) a skočíme na krok 2), ale pouze v případě, že přechod z aktuálního uzlu na následující uzel, se objevil v aktuální cestě maximálně jednou

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. Jak bylo zmíněno výše, přidání do seznamu vybraných cest pro pokrytí ADC, je podmíněno nutností cesty projít alespoň jednu novou definici.

4.3 Modelování CSP modelu

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 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 3.5.1 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:

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

Zdrojový text 4.1: Jednoduchý výraz

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

```
IntVar tmp1 = expr(*this, i - 42);     // mezivysledek (i - 42)
IntVar tmp2 = expr(*this, 42 * tmp1); // mezivysledek (42 * (i - 42))
IntVar x = expr(*this, i + tmp2);    // vysledek cele rovnice
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, ale 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. Speciálním typem jsou proměnné typu ukazatel a pole. Je nutné si uchovávat speciální mapu s klíčem indexu daného prvku, a hodnotou jako odkaz na odpovídající mapovanou proměnnou. Zde ale dochází k neurčitosti v modelování - je-li index do pole (respektive ukazatel přes aritmetiku ukazatelů) vypočítán za běhu programu ze zvolených proměnných, nemůžeme v momentě rozhodnout, který index zvolit. Nezbývá než vytvořit **předpoklad**. Předpoklad označuje konkrétní výběr neurčitého indexu. Při výběru se vychází z definičního oboru vypočítaného indexu - hodnota musí být větší než nula a musí patřit do definičního oboru výsledku výrazu, určující hodnotu indexu. Při takové akci explicitně vybíráme konkrétní řešení a znemožňujeme obecné řešení. Proto si musíme někde kopii modelu s jeho aktuálním stavem před výběrem indexu. Dojde-li k situaci, kdy model s námi vybraným indexem nemá žádné řešení, je nutné vrátit všechny změny, tedy nahrát uložený model a vybrat hodnotu jinou. Obvykle se začíná nejmenší možnou hodnotou indexu, která se postupně zvyšuje, při neúspěšných řešeních. Při přílišném používání neurčitých indexů, může metoda vést k velké procesorové i paměťové náročnosti.

4.4 Moduly a běh programu

Následuje popis jednotlivých modulů vytvořené aplikace.

AutoTestGen.cpp

Modul s main() funkcí, který pouze plní funkci vytvoření instance typu Clang nástroj, a předání zdrojových souborů ke statické analýze.

WalkThroughAST.cpp a WalkThroughAST.h

Modul obsahující třídy dědící z knihovny LibTooling, která zprostředkovává statickou analýzu. Z nich nejdůležitější je ASTVisitor, který nám umožňuje se zastavit v různých částech

AST, dle našich podmínek. My se zastavíme při každém nálezu definice funkce, proo kterou vytvoříme instanci objektu FunctionUnderTest, ze kterého získáme informace pro nastartování modelovacího jádra.

FunctionUnderTest.cpp a FunctionUnderTest.h

Účelem tohoto modulu je poskytnou rozhraní pro vytvoření CFG k dané testované funkci a ten poskytnout objektům, zajišťujícím vytvoření cest, dle potřebného kritéria pokrytí.

DefUseCFG.cpp, DefUseCFG.h a DefUseBlock.h

Modul obsahuje definici jediné třídy DefUseCFG, která představuje CFG graf, s uzly popisujícími def-use páry, jak bylo nastíněno v kapitole 2.2. Metody třídy implementují navržené algoritmy a jedna z metod je odpovědná za výpis mé reprezentace CFG do souboru typu DOT, který je možné různými nástroji převést na grafickou reprezentaci tohoto grafu. Jednotlivé uzly stromu jsou instancemi třídy DefUseBlock definované ve stejnojmenném hlavičkovém souboru. Uzly obsahují členy popsané v návrhu implementace a def() či use() jsou specifikovány jako instance třídy DefOrUse ze stejného hlavičkového souboru.

AllDefsTestSuite.cpp a AllDefsTestSuite.h

Modul pro definici testovací sady s kritériem pokrytí All-defs. Implementuje algoritmy navržené v kapitole 4.2 pro hledání všech cest a následnou eliminaci těch redundantních. Jeho vstupem je DefUseCFG a poskytuje metodu, vracející jednu cestu po druhé ze seznamu použitých cest.

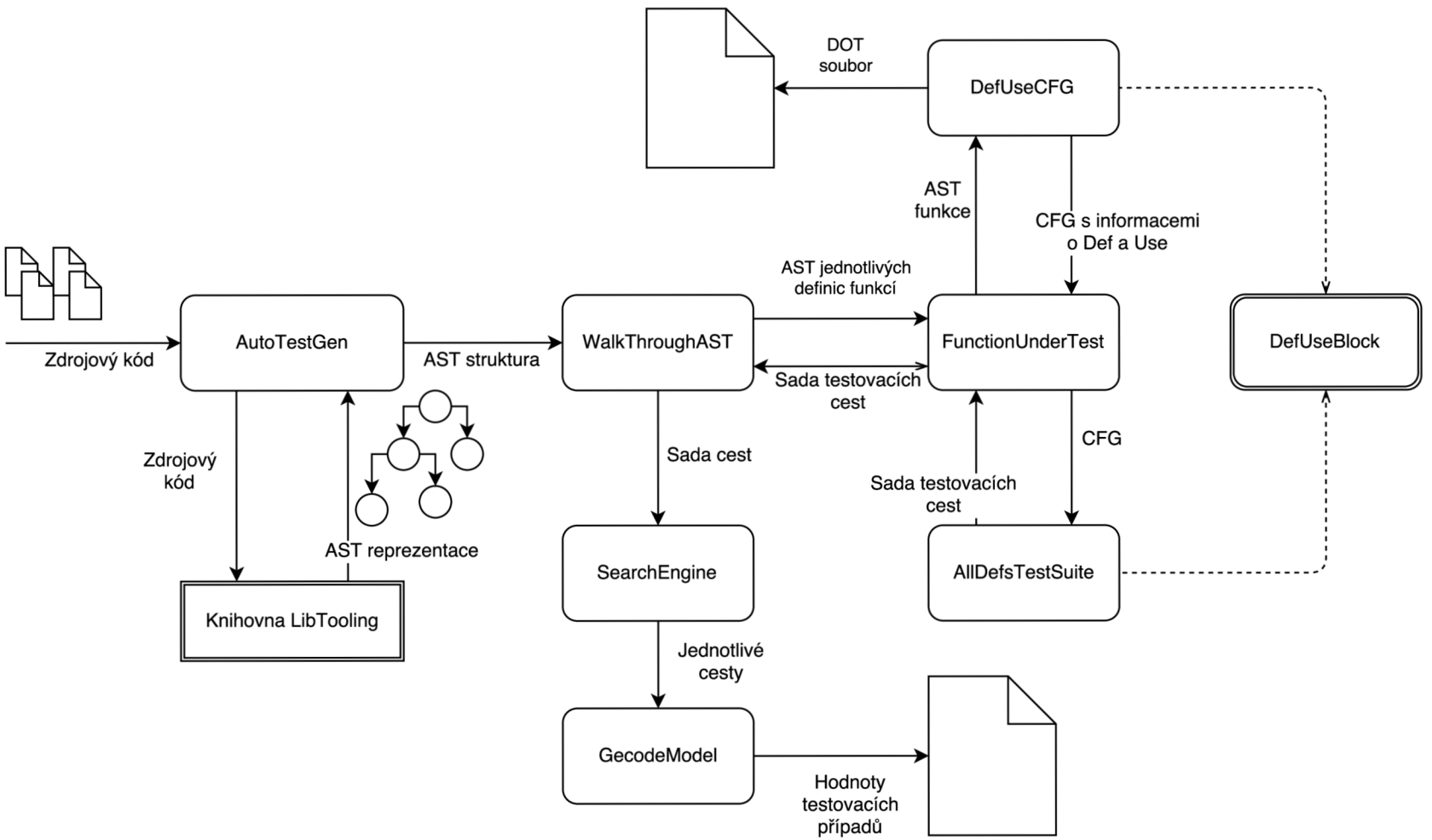
MyEngine.cpp a MyEngine.h

MyEngine je prostředník pro práci s CSP modelem pro generování vstupních argumentů. Obsahuje pouze jednoduché metody pro získání jednotlivých cest a předání jednotlivých výrazů do jádra generátoru.

GecodeTest.cpp a GecodeTest.h

Jádro samotného generátoru. Implementuje navržené algoritmy a metodiky z kapitoly 4.3. Jeho výstupem jsou jednotlivá řešení pro danou cestu.

Grafická reprezentace běhu programu



Obrázek 4.5: Grafická reprezentace běhu programu

Kapitola 5

Generování testů

Následující testy byly náhodně vytvořeny pouze pro demonstraci, nemají vyšší význam.

5.1 Test průchodu

Nesmyslná podmínka která nemá řešení:

Kód

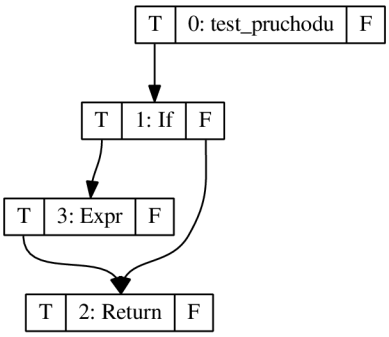
```
1 int test_pruchodu(int x)
2 {
3   if (x % 9 > 9)
4     x++;
5
6   return x;
7 }
```

Zdrojový text 5.1: Nesmyslná podmínka

Výstup aplikace: řešení a CFG

```
1 Funkce ``test_pruchodu'':
2 Cesta c. 1:
3 0 -> 1 -> 3 -> 2 ->
4 Pro danou cestu nebylo nalezeno vhodne reseni
```

Zdrojový text 5.2: Nesmyslná podmínka



Obrázek 5.1: CFG testu průchodu

5.2 Test celých čísel

Kód

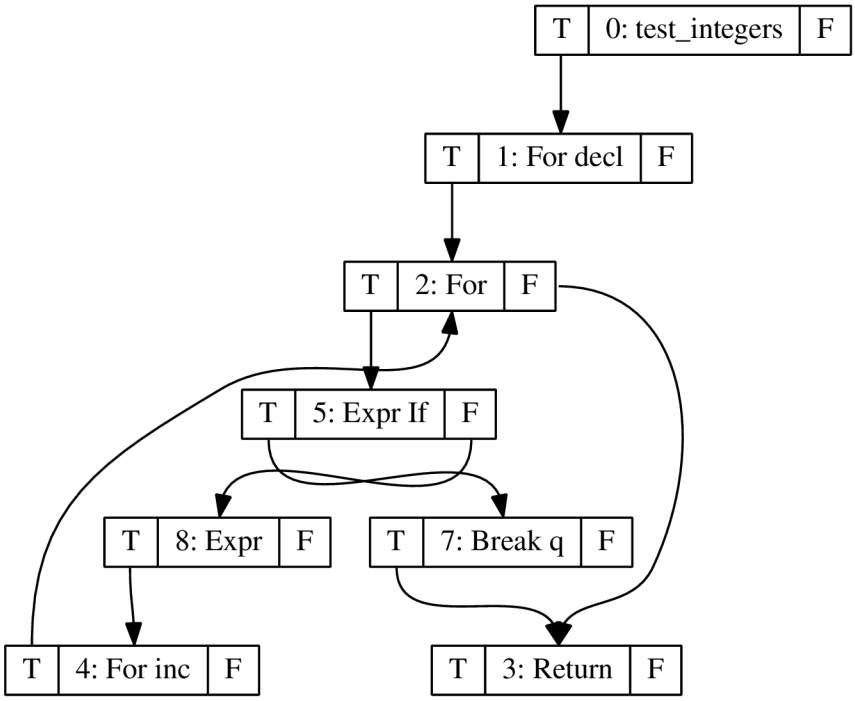
```
1
2  int test_integers(char x, unsigned y)
3  {
4    for (int i = x; i < 10; i++)
5    {
6      y += y;
7
8      if (y < x && y > 2)
9        break;
10     else
11       y++;
12   }
13
14   return y;
15 }
```

Zdrojový text 5.3: Nesmyslná podmínka

Výstup aplikace: řešení a CFG

```
1  Funkce "test_integers":
2  Cesta c. 1:
3  0 -> 1 -> 2 -> 5 -> 8 -> 4 -> 2 -> 3 ->
4  Reseni:
5  x = 9
6  y = 5
7
8  Cesta c. 2:
9  0 -> 1 -> 2 -> 5 -> 7 -> 3 ->
10 Reseni:
11 x = 5
12 y = 2
```

Zdrojový text 5.4: Nesmyslná podmínka



Obrázek 5.2: CFG testu průchodu

5.3 Test čísel s plovoucí desetinnou čárkou

Kód

```
1  double test_floats(double d, float f, char c)
2  {
3    if (c == 'V' || c == 'U' || c == 'T')
4    {
5      return f;
6    }
7    else
8    {
9      long double ld1 = d * f;
10     d *= 3.14;
11     f += 3.14;
12
13     if (ld1 <= 42.0)
14       return  f;
15     else
16       return d;
17   }
18
19   return f;
20 }
```

Zdrojový text 5.5: Test čísel s plovoucí desetinnou čárkou

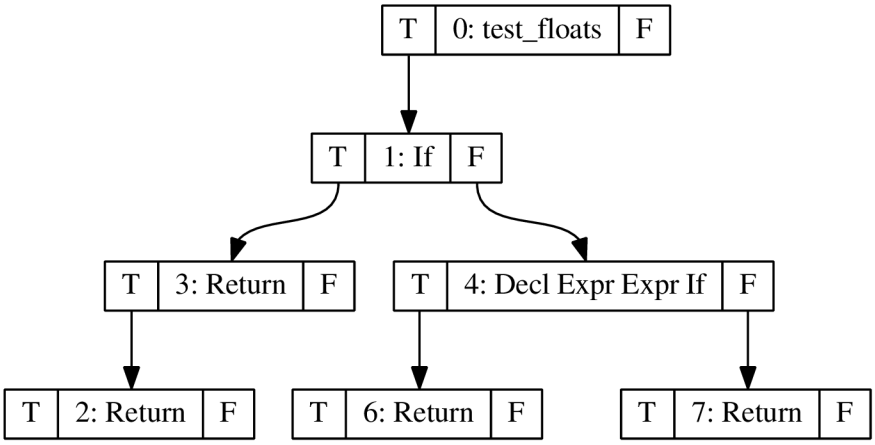
Výstup aplikace: řešení a CFG

```
Funkce "test_floats":
Cesta c. 1:
0 -> 1 -> 4 -> 7 ->
Reseni:
d = -50
f = -2
c = -128

Cesta c. 2:
0 -> 1 -> 4 -> 6 ->
Reseni:
d = -50
f = -0.84
c = -128

Cesta c. 3:
0 -> 1 -> 3 ->
Reseni:
d = -100
f = -100
c = 86
```

Zdrojový text 5.6: Test plovoucí desetinné čárky



Obrázek 5.3: CFG testu průchodu

5.4 Test struktur

Kód

```
int test_struktur(struct str A, struct str B, int x)
{
  int i_left = A.i;
  int i_right = B.i;
  while (i_left == i_right && i_left > 0 && i_right > 20)
  {
    if (A.d > B.d)
      break;
    A.i++;
  }

  double d_left = A.d;
  double d_right = B.d;
  d_right++;

  if (d_left >= 3.14 && d_right > 1.73)
    B.i++;
  else
    x++;

  return x;
}
```

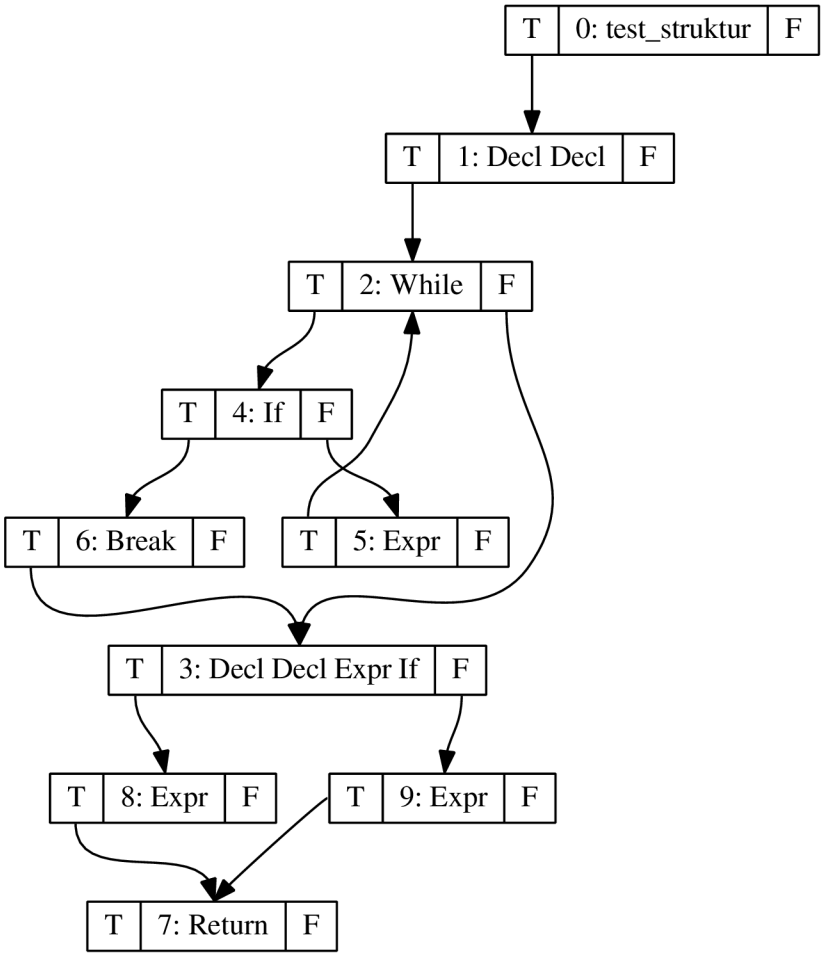
Zdrojový text 5.7: Test struktur

Výstup aplikace: řešení a CFG

```
Funkce "test_struktur":
Cesta c. 1:
0 -> 1 -> 2 -> 4 -> 6 -> 3 -> 9 -> 7 ->
Reseni:
A = { d = -99.9999, i = 21, }
B = { d = -100, i = 21, }
x = -2147483646

Cesta c. 2:
0 -> 1 -> 2 -> 4 -> 6 -> 3 -> 8 -> 7 ->
Reseni:
A = { d = 3.14001, i = 21, }
B = { d = 0.730001, i = 21, }
x = -2147483646
```

Zdrojový text 5.8: Test struktur



Obrázek 5.4: CFG testu průchodu

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.

Rád bych uvedl, že tento projekt měl, a doufám že nadále bude mít, pro mě velký osobní přínos. Získal jsem cenné zkušenosti s automatizačními nástroji a z oblasti testování obecně. Snaha o spojení dvou velkých knihoven a studium jejich dokumentací a rozsáhlých referencí pro mne bylo poměrně výzvou a jsem rád, že jsem došel k pozitivním výsledkům. Velmi rád budu pokračovat na dalším rozvoji tohoto zajímavého projektu, například integrace s jinými nástroji zmíněnými v kapitole 2.3.3, nebo rozvoj pro jiné programovací jazyky. Clang nabízí stejný výstup z dalších jazyků, které jsou většinou podobné jazyku C a proto věřím, že je tato možnost reálná.

Na druhou stranu vím, že má implementace obsahuje nedostatky a plně nepodporuje některé z konstrukcí jazyka C, které vychází z jeho nízké úrovně (například přetypování). Osobně jsem poměrně nespokojen se stavem zdrojových kódu, určených k odevzdání. Rád bych na tomto problému zapracoval a aplikaci vylepšil, aby byla skutečně využitelná a přínosná pro testery.

Literatura

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

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

[3] 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

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

[5] 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/

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

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

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

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

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

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

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

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

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

Přílohy

Seznam příloh

Příloha A

Obsah CD

- **/usr** - zdrojové soubory
- **/doc** - elektronická verze bakalářské práce
- **README.txt** - uživatelská příručka