Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Generování jednotkových testů na základě toku řízení programu

Plzeň 2019

Jan Albl

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Akademický rok: 2018/2019

Studijní program: Inženýrská informatika
Forma: Prezenční
Obor/komb.: Softwarové inženýrství (SWI)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. ALBL Jan	Štěnovický Borek 45, Štěnovický Borek	A16N0027P

TÉMA ČESKY:

Generování jednotkových testů na základě toku řízení programu

TÉMA ANGLICKY:

Unit test generator based on program control flow

VEDOUCÍ PRÁCE:

Ing. Richard Lipka, Ph.D. - KIV

ZÁSADY PRO VYPRACOVÁNÍ:

1. Seznamte se s metodami generování testových dat
2. Prozkoumejte metody a nástroje pro získání control flow programu v Javě
3. Navrhněte metodu pro získání parametrů jednotkových testů zajišťujících požadované pokrytí testovaného programu
4. Navržené řešení implementujte
5. Vytvořené řešení důkladně otestujte

SEZNAM DOPORUČENÉ LITERATURY:

Dodá vedoucí práce

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 15. května 2019

Jan Albl

Poděkování

Na tomto místě bych rád poděkoval panu Ing. Richardu Lipkovi, Ph.D. za odborné vedení práce, cenné rady, trpělivost a ochotu, které mi pomohly tuto práci zkompletovat. Dále bych rád poděkoval rodině a blízkým přátelům za pomoc a podporu během studia.

Abstrakt

Cílem této diplomové práce bylo navrhnout a implementovat postup pro generování testovacích dat, která se využijí při automatickém generování jednotkových testů pro Java kód. Navržený postup by měl být založen na analýze grafu toku řízení testované metody. V teoretické části práce jsou popsány způsoby pro generování testovacích dat a problematika, která se s tímto generováním pojí. Dále byly zkoumány nástroje pro extrakci grafu toku řízení z Java kódu. Jeden z těchto nástrojů byl vybrán a na jeho základě byla navržena metoda pro získání parametrů jednotkových testů. V praktické části byla vytvořena knihovna, která dokáže vygenerovat testovací data na základě analýzy grafu toku řízení pro některé metody objektů. Z těchto dat jsou poté generovány jednotkové testy.

Abstract

The aim of presented thesis was to design and implement a procedure for test data generation that will be used in the process of automatic generation of unit tests for Java code. The proposed procedure ought to be based on an analysis of the control flow graph of the test method. In the theoretical part there are described the methods for test data generation and the questions that are involved with this generation. Furthermore, the tools for extraction of control flow graph from Java code were researched. One of these tools was selected and on the base a method for gaining of unit test parameters was proposed. In the practical part the library which is able to generate test data by analyzing the control flow graph for some object methods was created. Then unit tests are generated from these data.

Obsah

Úvod

Proces testování je při vývoji kvalitního softwarového produktu velmi důležitý. Ověřuje spolehlivost, kvalitu a upozorňuje na chyby, které mohly během jeho vývoje vzniknout. Testování produktu by měla být věnována značná pozornost již od počáteční fáze vývoje, jeho podceňování totiž vede k větším nákladům při opravě později nalezených chyb a nespokojenosti zákazníka.

Existuje několik strategií testování softwaru. Během implementační fáze bychom měli testy provádět opakovaně. Důležitou položkou při testování jsou kvalitní testovací data. Proces získávání takovýchto testovacích dat může být u rozsáhlejších aplikací složitý.

Cílem této diplomové práce je usnadnit jednotkové testování vytvářením testovacích dat pro aplikace napsané v programovacím jazyce Java. Existuje několik způsobů, jak můžeme testovací data generovat. Ty nejzákladnější techniky jsou popsány ve druhé kapitole této práce. Dále je pak řešena problematika, se kterou se při generování dat setkáváme. Práce se zaměřuje především na využití toku řízení testovaného programu, jehož analýzou jsou data generována. V závěrečné části práce byla vytvořena knihovna, která implementuje navrženou techniku pro získání parametrů jednotkových testů.

1 Softwarové testování

Při tvorbě kvalitního softwarového produktu je jeho testování zásadním aspektem. Dokáže odhalit chyby a problémy, které se mohou projevit až při provozu u koncového zákazníka. Neošetřená chyba v programu může vést i k samotnému selhání celého systému. Na rozdíl od testování hmotného produktu, kde chyby mohou být zřetelné na první pohled, je testování softwaru složitější, neboť chyba se může projevit pouze za specifické kombinace vstupních parametrů a vnitřního stavu programu. Nedosáhneme-li takového stavu, chyba se neprojeví a vývojář tudíž její existenci nezjistí. Některé statistiky [20] udávají, že až 50 % z celkových nákladů na vývoj softwaru je vynaloženo pouze na jeho testování.

Historie testování je stará jako jeho vývoj. [32] Při tvorbě prvních jednoduchých programů, které vznikaly přibližně v první polovině dvacátého století, bylo jejich ověřování velmi pracné. Výpočetní výkon počítačů byl v této době omezený a drahý. Samotné ověřování programů bylo prováděno manuálním zkoumáním programů včetně jejich výstupů. S postupným časem rostla komplexita programů, a tak nebylo uskutečnitelné jejich kompletní ověření vzhledem k počtu možných kombinací vstupů a průchodů kódem. Nové způsoby testování se zaměřovaly na hledání chyb v programu, zvyšování kvality produktu a první automatizaci v testovacích případech. I v dnešní době je testování softwaru zásadní, časově náročná a finančně nákladná aktivita.

1.1 Manuální a automatické testování

S přibývajícím počtem testovacích případů rostly výdaje a taktéž doba nezbytná pro testování. Vznikala potřeba **automatizace** testovacích postupů, díky kterým by se testování výrazně urychlilo. Automatizace testů přinesla několik výhod, jako například rychlost, opakovatelnost a úsporu prostředků. [19, 25] Existují aplikace, které nelze dostatečně testovat pomocí automatických testů, proto je zapotřebí využití zejména metody manuálního testování. [28] Příkladem mohou být aplikace silně závislé na uživatelském rozhraní (například počítačové hry). Manuální a automatické testování jsou navzájem komplementární a je tedy výhodné spolu tyto přístupy při vývoji kombinovat. V této práci se zaměřím na generování testovacích dat, která mohou být využity jak pro automatické, tak manuální testování.

1.2 Unit testing

Softwarové testování můžeme rozdělit do několika typů, z nichž každý lze použít k odlišným účelům v různých fázích projektu. [23] Integrační testy se například zaměřují na správnou komunikaci jednotlivých komponent uvnitř aplikace, zatímco funkční testy ověřují, zda hlavní funkce systému splňují své požadavky. Testování výkonu na druhé straně zjišťuje schopnost softwaru pracovat v různých zátěžových podmínkách.

Účelem testování jednotky (unit testing) je ověřit, zda každá jednotka softwaru funguje tak, jak byla navržena a popsána v dokumentaci. [6] Jednotka je nejmenší užitečně testovatelná část jakéhokoliv softwaru. Nejčastěji má jeden nebo několik vstupů a jeden výstup. Například v objektově orientovaném programování je nejmenší jednotkou metoda, v procedurálním programování může být jednotkou například funkce, individuální program, postup atd.

K tvorbě unit testu potřebujeme znát i strukturu kódu (white box testing), díky tomu mohou být provedeny všechny možné testy, které zajistí, že veškeré součásti kódu byly náležitě vykonány a jsou v souladu se specifikací. [31] Jednotkové testování může být považováno za základní aspekt pro řadu dalších forem testování. V programovacím jazyce Java je například praktické psát unit testy, které validují jednotlivé metody třídy vystavené ostatním třídám. Bonusy, které unit testing přináší, jsou třeba rychlejší vývoj, ladění, zpětná vazba, lepší dokumentace, návrh apod. [23]

1.3 Statická a dynamická analýza kódu

Statická analýza kódu se provádí bez nutnosti běhu programu. Může rozpoznat možné kódovací chyby, nedostatky či zadní vrátka již v počáteční fázi životního cyklu vývoje softwaru. Klasickým příkladem nástroje, který uskutečňuje statickou analýzu, je kompilátor. **Dynamická analýza** se využívá v době, kdy je testovaný program již spuštěn. Dokáže odhalit chyby, které jsou statickou analýzou neodhalitelné, jako je například doba odezvy programu, systémová a paměťová náročnost, funkční chování atd. Nalezne však defekty pouze v části kódu, který je skutečně prováděn. To může být nevýhodou u složitých programů, protože s přibývajícím počtem všech možných cest napříč programem roste i větší pravděpodobnost, že při dynamické analýze neprozkoumáme všechny větve. [12, 16]

1.4 Testovací data

Při provádění funkčních testů jsou ve většině případů potřebná testovací data. Ta jsou průchodem kódu ovlivňována či ona sama ovlivňují chování softwaru. [5] Můžeme je rozdělit do dvou kategorií, a to na **pozitivní** a **negativní testovací data**. [2] Pozitivní testovací data jsou použita typicky jako standardní vstup. Tato data by měla být aplikací správně zpracována a vedou k očekávanému výsledku. Negativní testovací data mohou být chybná, nedostatečná či upravená a v některých případech mohou vést k vyvolání chybového stavu. Současně testují schopnost programu zvládat extrémní, neočekávaný či výjimečný vstup. Při špatném navržení testovacích dat se nemusí projevit všechny možné scénáře vykonávání programu, a to může zapříčinit snížení kvality celého testování softwaru. [4]

Existuje několik způsobů, jak lze vytvořit testovací data, a to buď manuálně, kopií dat pocházejících ze starších klientských systémů, nebo generováním pomocí speciálních nástrojů. [5] S ohledem na téma diplomové práce dále přiblížím vytváření testovacích dat generováním.

2 Generování testovacích dat

V rámci testování softwaru je často žádoucí nalézt testovací data, která jsou zpracovávána testovaným programem. Ruční vyhledávání těchto testovacích dat je ve složitějším softwaru časově velmi náročné a pracné. Vznikla tedy potřeba pro automatizaci, která by zkrátila dobu tohoto procesu. Během vývoje softwaru bylo vyvinuto několik způsobů, jak můžeme testovací data generovat. V této kapitole bude popsáno několik základních technik generování a současně se pokusím přiblížit problematiku s tím související.

2.1 Náhodné generování

Nahodilé generování testovacích dat je nejjednodušší generační technikou. Vstup je generován zcela nahodile a může být použit pro jakýkoliv datový typ. Vytvoření dat je velmi rychlé, ale nevýhodou tohoto postupu je nedostatečné pokrytí kódu či stavová exploze, viz sekce - 2.5.1. S ohledem na fakt, že se generování spoléhá pouze na pravděpodobnost, je velmi nízká šance na průchod větví, která je dosažitelná jen malým procentuálním podílem vstupních dat programu.

Jako příklad zvažme následující algoritmus 1. Při náhodném generování je zřejmé, že je mnohem pravděpodobnější volání funkce *write* s parametrem b než s parametrem a. [3, 14, 20]

Algoritmus 1

```
function doSomething($a, b$)
    if $a$ ==b then
        write(a)
    else
        write(b)
    end if
end function
```

2.2 Cílově orientovaný přístup

Cílově orientovaný přístup (Goal-Oriented) pro generování dat poskytuje vodítko k určitému souboru cest. Místo generování vstupu, který prochází od vstupu do výstupu bloku kódu, generuje vstup, který prochází danou

nespecifickou cestou $\boldsymbol{u}$. Generátor může tímto způsobem najít jakýkoliv vstup pro jakoukoliv cestu p, kde $p \in u^*$. [14] Tento postup snižuje riziko nevytvoření dat pro relativně nedostupné cesty. Obecně jsou známé dvě metody používající tuto techniku, tzv. řetězový přístup a přístup orientovaný na tvrzení. [3, 20]

2.2.1 Řetězový přístup

Tento přístup se pokouší identifikovat posloupnost uzlů (příkazů), která je nezbytná pro realizaci cílového uzlu. Posloupnost uzlů je sestavena iterativně během provádění. Během vykonávání vyhledávacího procesu program rozhodne, zda bude pokračovat v provádění této větve, nebo zda bude vykonána větev alternativní, vzhledem k tomu, že aktuálně zvolená větev nevede k cílovému uzlu. Je-li tok spouštění nežádoucí, vyhledávací algoritmy se pokusí automaticky nalézt nový vstup pro změnu provedení toku. [14, 21]

2.2.2 Přístup orientovaný na tvrzení

Přístup orientovaný na tvrzení (assertion-oriented approach) využívá určité podmínky (nazývané tvrzení), které jsou ručně nebo automaticky vloženy do kódu. Jestliže je tvrzení splněno, postupuje se dále v provádění, jinak se vyskytne chyba v programu či tvrzení. V následujícím kódu je tvrzení, které říká, že b nesmí být nulové. Cílem generace orientované na tvrzení je najít jakoukoliv cestu k tvrzení. [14, 21]

Algoritmus 2

```
function FIE($a, b$)
    b = (a-a/2)*(a+1);
    assert(b != 0);
    return a/b;
end function
```

2.3 Generování podle cest

Generování zaměřené na cestu (path-oriented generation) je považováno za nejlepší z přístupů ke generování testovacích dat. [2, 14] Neposkytuje generátoru možnost výběru mezi souborem cest, ale je vybrána pouze jedna specifická cesta, pro kterou jsou data generována. Tento přístup je podobný cílově orientovanému (viz sekce - 2.2) s výjimkou použití specifických cest.

[14] Díky tomu dosahuje lepších výsledků pokrytí, ale je těžší najít testovací data. Pokud je technika založena výlučně na grafu toku řízení (control flow graph, viz kapitola - 3), vede často k výběru nedosažitelných cest relativně i absolutně. [14] Proto se při generování pomocí této techniky přidávají různá omezení.

2.4 Generování genetickými algoritmy

Slibným směrem pro generování testovacích dat pro komplexní software se stává využití genetických algoritmů (GA). Jedná se o heuristický postup, který se snaží za pomoci evoluční biologie nalézt řešení složitých problémů. [30] Využívá principů dědičnosti, křížení, mutace a přirozeného výběru prostřednictvím hodnotící funkce.

Genetické algoritmy jsou založeny na postupné tvorbě **populací** (generací) různých řešení daného problému. Vstupní populace je v mnoha případech generována náhodně ze vzorků vstupního prostoru testovaného programu. [30] Tento způsob přináší určitou slepotu při generování testovacích dat. [34] Byly však navrženy i zdokonalené metody pro zvýšení jejich kvality.

Při přechodu do nové generace jsou jedinci ve stávající populaci ohodnoceni pomocí **ohodnocující funkce**, která spočte podle určitých kritérií kvalitu (fitness) daného prvku. Po ohodnocení následuje výběr jedinců z populace, kteří se mohou stát rodiči pro nové prvky nové populace. Výběr je většinou řízen podle kvality jedince. Vybraní jsou pak modifikováni pomocí mutací a křížení s ostatními, čímž vznikne nová populace. Tento postup se iterativně opakuje, dokud nedostaneme postačující kvalitu řešení.

V mnoha případech je genetický algoritmus spuštěn pro každou větev samostatně. Pro pokrytí všech větví testovaného programu je nutné genetický algoritmus několikrát spustit, což způsobuje, že proces generování je zdlouhavý. Některé experimentální výsledky uvádí, že testovací sada získaná pomocí genetických algoritmů dokáže pokrýt či je velmi blízko pokrytí dané větve. [34]

2.5 Problémy při generování testovacích dat

Během generování dat se můžeme setkat s určitými problémy, které proces zkomplikují. Mnoho programovacích jazyků dovoluje vytvářet abstraktní datové typy, ukazatele a další programové konstrukce, které ztěžují generování testovacích dat. Kromě konstrukčních problémů je zde i problém stavové exploze, viz sekce - 2.5.1.

Mezi problémové konstrukce patří **pole** a **ukazatele**, které vytváří problémy při symbolickém provádění, protože jejich hodnoty jsou známy až za běhu programu. [14] Dalším problémem může být *indexace* nebo **problém tvaru** (struktura vstupu, která může být dána ukazateli).

Objekty jsou stejně jako ukazatele často dynamicky přiděleny. Navíc se s nimi pojí i další problémové oblasti při generování, jako abstraktní třídy, dědičnost či polymorfismus. Tyto konstrukce komplikují analýzu předcházející generování testovacích dat především tím, že v době kompilace programu nelze určit, jaká konkrétní data jsou přidělována nebo jaký implementovaný kód je popřípadě volán. Z toho vyplývá, že jakékoli řešení těchto problémů musí být řešeno dynamicky po spuštění testovaného programu.

Další potencionálně problémovou konstrukcí jsou **smyčky** (viz sekce - 3.1.2). Pokud jsou smyčky závislé například na vstupních proměnných, nemají konstantní počet iterací. V případě symbolického vyhodnocení tak nevíme, kolikrát daná smyčka proběhne.

Software se většinou skládá z funkcí a různých přidaných **modulů**. Při symbolickém provedení nemusí být zdrojový kód funkce či modulu dostupný, například překompilované knihovny, a proto není možno provést úplnou statickou analýzu. [14]

2.5.1 Stavová exploze

Objekty a datové typy mohou v každém okamžiku nabývat různých stavů a hodnot. Množina všech těchto stavů pak vytváří stavový prostor systému. S přibývajícím počtem stavových proměnných v systému roste velikost stavového prostoru exponenciálně. Tento jev se nazývá **state explosion problem** neboli problém stavové exploze. [22]

Při testování většiny softwaru nelze pokrýt celý stavový prostor, neboť se i při malém počtu proměnných potýkáme s problémem stavové exploze. Počet všech možných stavů, kterých může nabývat například primitivní parametr pro číslo s pohyblivou řádovou čárkou (float či double), je velmi velký a v reálném prostředí by otestování všech stavů bylo nemožné.

Pro ověření správnosti programu však nemusíme generovat celou škálu hodnot, kterou může proměnná nabývat, ale stačí nám vygenerovat pouze hodnoty, které mají vliv na průchod programem. Výběr těchto hodnot však nemusí být triviální a proměnné na sobě mohou být například závislé.

3 Řízení toku programu

Mnoho programovacích jazyků disponuje příkazy pro řízení toku (control flow statement), které rozhodují o dalším provádění programu tím, že jej větví, cyklí nebo jinak mění jeho běh. Příkladem příkazu pro řízení toku jsou příkazy *if* nebo *loop*. [1] Pro reprezentaci sekvencí, které by mohly být procházeny programem během jeho vykonávání, se běžně používá graf toku řízení (control flow graph). V této kapitole detailně popíši graf toku řízení a nástroje, které se používají pro jeho získání z programů napsaných v programovacím jazyce Java.

3.1 Graf toku řízení

Graf toku řízení programu (CFG) je směrový graf, kde uzly reprezentují základní bloky v programu a hrany představují tok řízení mezi nimi. [17]

Základní blok je posloupnost po sobě jdoucích instrukcí, má jeden vstupní a jeden výstupní bod. Vstoupí-li řízení programu do základního bloku, provede všechny jeho instrukce bez zastavení nebo možnosti větvení s výjimkou konce. [17] Celý kód aplikace může být jednoznačně rozdělen do skupin základních bloků, které se nepřekrývají.

Základní bloky grafu toku řízení můžeme rozdělit do různých typů podle poslední instrukce základního bloku. [10]

- **Jednocestný základní blok** - jeho poslední instrukce je nepodmíněný skok a vede z něj pouze jedna hrana.

- **Dvoucestný základní blok** - jeho poslední instrukcí je podmíněný skok. Vedou z něj tedy dvě hrany.

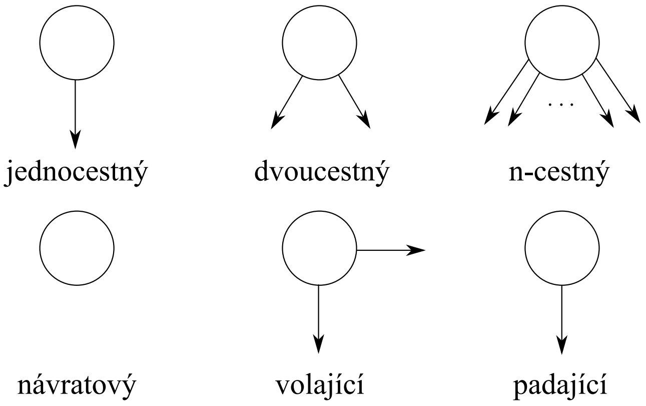
- **N-cestný základní blok** - poslední instrukce tohoto bloku je indexový skok. Z tohoto uzlu vede až N hran.

- **Volající základní blok** - poslední instrukcí je volání podprogramu. Existují dvě hrany vedoucí z tohoto uzlu. První k podprogramu, který se volá, a druhá k instrukci po volání programu, pokud tedy pokračujeme ve vykonávání volajícího.

- **Návratový základní blok** - jeho poslední instrukcí je ukončení programu nebo návrat. Z tohoto bloku nevedou žádné hrany.

- **Padající základní blok** - speciální typ jednocestného základního bloku, jehož následující instrukce (blok) je cílová adresa instrukce větvení. V programovacích jazycích bývá tato následující instrukce často unikátně označena.

Všechny typy základních bloků reprezentovaných v grafu toku řízení můžeme vidět na obrázku č. 3.1.



Obrázek 3.1: Základní bloky grafu toku řízení

Graf toku řízení *G = (N, E)* programu P je tedy orientovaný graf, kde N je množina základních bloků (uzlů), E reprezentuje množinu hran. V každém grafu existují dva speciální bloky n_1 a n_n, kde n_1 reprezentuje **vstupní blok** (entry block) a n_n **blok výstupní** (exit block). Vstupním blokem vstupujeme do programu P a výstupním jej opouštíme. Tyto bloky jsou velmi důležité při následné analýze toku řízení, protože při vykonávání programu P se tyto bloky vždy zpracují. Graf toku řízení splňuje následující podmínky:

1. $\forall n \in N$, n reprezentuje základní bloky programu P
2. $\forall e = (n_i, n_j) \in E$, e představuje tok řízení od jednoho základního bloku k druhému a $n_i, n_j \in N$
3. $\exists f : \boldsymbol{\beta} \rightarrow N \bullet \forall b_i \in \boldsymbol{\beta}, f(b_i) = n_k$ pro $n_k \in N \wedge \nexists b_j \in \boldsymbol{\beta} \bullet f(b_j) = n_k$

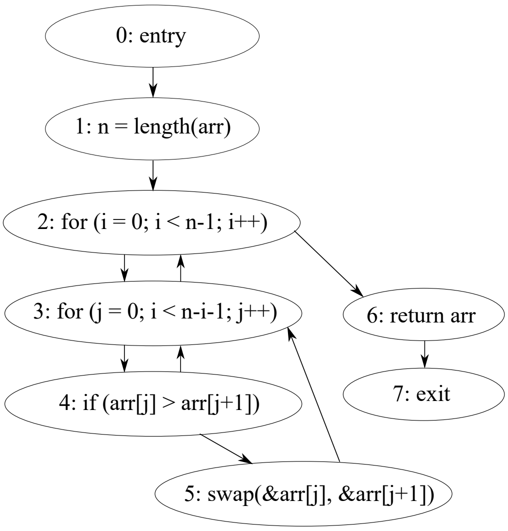
Cesta v grafu toku řízení *G = (N, E)* ze základního bloku n_1 do bloku n_m je reprezentována jako sekvence hran (n_1, n_2), (n_2, n_3), ..., (n_{m-1}, n_m) $\in E, m \geq 1$. [10]

Při sestavování grafu toku řízení nejdříve vytvoříme základní bloky a poté přidáme hrany. Například algoritmus bubble sort 3 obsahuje tři dvoucestné základní bloky.

Algoritmus 3

```
function BubbleSort(arr)
    $n$ = length(arr);
    for $i = 0; i < n - 1; i + +$ do
        for $j = 0; j < n - i - 1; j + +$ do
            if $arr[j] > arr[j + 1]$ then
                swap(&$arr[j]$, &$arr[j + 1]$);
            end if
        end for
    end for
    return $arr$;
end function
```

Graf toku řízení tohoto příkladu je zobrazen na obrázku č. 3.2.



Obrázek 3.2: Graf toku řízení pro algoritmus č. 3

Pro implementaci grafu toku řízení v paměti počítače se ve většině případů používá spojový seznam. Implementace pomocí matice je neefektivní, neboť graf toku řízení je řídký. Je používán různými softwarovými nástroji, které analyzují tok řízení programu. Zejména pak kompilátory, které jsou jeho prohledáváním schopny optimalizovat kompilovaný kód. [10, 17] Při analýze grafu toku řízení je užitečné znát některé koncepty, které budou vysvětleny v sekcích níže (viz sekce - 3.1.1, 3.1.2, 3.1.3)).

3.1.1 Dominance uzlu

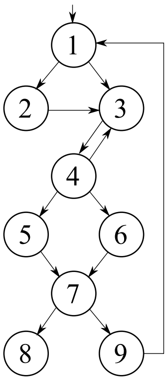
Uzel *d* grafu toku řízení **dominuje** uzlu *n*, pokud každá cesta ze vstupního uzlu grafu do uzlu *n* vede přes uzel *d*. Pro vyjádření tohoto vztahu můžeme použít zápis *(d dom n)* a říkáme, že uzel *d* je **dominátorem** uzlu *n*. [7, 26] Uzel v grafu toku řízení může mít z principu více dominátorů. Soubor všech dominátorů pro uzel *n* značíme *Dom(n)*. Vlastnosti dominance mezi uzly:

- Každý uzel dominuje sám sobě $n \in Dom(n)$.
- Uzel *d* je **striktně dominantní** pro uzel *n*, pokud $d \in Dom(n)$ a $d \neq n$ značíme *(d sdom n)* .
- Vstupní uzel smyčky je dominantní pro všechny uzly uvnitř cyklu.
- Všechny uzly, kromě vstupního uzlu grafu, mají unikátní **bezprostřední dominátor** *m*, který je posledním dominátorem pro *n* v libovolné cestě od vstupního uzlu do *n*. Tento vztah se značí zápisem *(m idom n)* a musí platit podmínka $m \neq n$.

Hranice dominance (dominance frontier) uzlu *n* (zkráceně *DF(n)*) je množina všech uzlů *d*, kde *n* je dominátor bezprostředního následníka *d*, ale *n* není striktně dominantní vůči *d*. Pro usnadnění můžeme říct, že se jedná o sadu uzlů, kde dominance uzlu *n* končí.

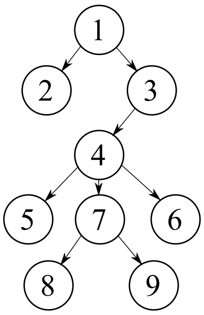
Dalším pojmem, se kterým se můžeme při analýze grafu toku řízení setkat, je **strom dominance**. Strom se skládá ze základních bloků grafu a jeho hierarchie určuje dominanci mezi uzly. Rodič ve stromu je vždy dominátorem potomka. Kořenem stromu je vstupní uzel, který je dominantní pro všechny uzly v grafu.

Na obrázku č. 3.3 je znázorněný graf toku řízení, který použijeme pro snazší pochopení definovaných pojmů.



Obrázek 3.3: Graf toku řízení pro ukázku počítání dominance mezi uzly

Z grafu (na obrázku č. 3.3) nejprve vytvoříme strom dominance, který je zobrazen na obrázku č. 3.4. Z toho stromu pak snadno zjistíme dominanci mezi jednotlivými uzly. Získané informace jsou zobrazeny v tabulce č. 3.1.



Obrázek 3.4: Strom dominance grafu toku řízení z obrázku č. 3.3

uzel	***Dom(n)***	***d sdom n***	***d idom n***	***DF(n)***
1	{1}	-	-	-
2	{1, 2}	{1}	1	{3}
3	{1, 3}	{1}	1	-
4	{1, 3, 4}	{1, 3}	3	-
5	{1, 3, 4, 5}	{1, 3, 4}	4	{7}
6	{1, 3, 4, 6}	{1, 3, 4}	4	{7}
7	{1, 3, 4, 7}	{1, 3, 4}	4	-
8	{1, 3, 4, 7, 8}	{1, 3, 4, 7}	7	-
9	{1, 3, 4, 7, 9}	{1, 3, 4, 7}	7	-

Tabulka 3.1: Získané informace o dominanci mezi jednotlivými uzly z příkladu na obrázku č. 3.3

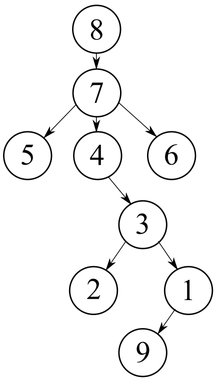
Analogicky k dominanci mezi uzly existuje i **post-dominance**. Uzel d v grafu toku řízení je post-dominantní uzlu n, pokud každá cesta z uzlu n do koncového uzlu vede přes uzel d. Tento vztah značíme *(d pdom n)* a říkáme, že uzel d je **post-dominátor** uzlu n. [7, 26]

- Každý uzel je post-dominantní sám sobě $n \in$ *Pdom(n)*.
- Uzel d je **striktně post-dominantní** pro uzel n, pokud $d \in$ *Pdom(n)* a $d \neq n$ značíme *(d spdom n)* .
- Všechny uzly, kromě výstupního uzlu, mají unikátního **bezprostředního post-dominátora** m, který je posledním post-dominátorem pro n v libovolné cestě od n do výstupního uzlu.

- Post-dominence je ekvivalentní dominanci, pokud je tok v grafu převrácen (procházíme graf od koncového uzlu k počátečnímu).

Hranice post-dominance (postdominance frontier) uzlu *n* zkráceně *PDF(n)* je množina všech uzlů *d*, kde *n* je post-dominátor bezprostředního následníka *d*, ale *n* není striktně post-dominantní vůči *d*. Pro usnadnění můžeme říci, že hranice post-dominance uzlu *n* je množina nejbližších n-cestných předchůdců uzlu *n*, ve kterých se rozhoduje, zda bude uzel *n* dosažen či nikoliv.

Podobně jako strom dominance lze sestrojit i **strom post-dominance**. V grafu toku řízení se pouze změní směr toku (orientace hran). Při vytváření stromu postupujeme od výstupního uzlu ke vstupnímu stejným způsobem jako při vytváření stromu dominance.



Obrázek 3.5: Strom post-dominance grafu toku řízení z obrázku č. 3.3

uzel	***Pdom(n)***	***d spdom n***	***d ipdom n***	***PDF(n)***
1	{1, 3, 4, 7, 8}	{3, 4, 7, 8}	3	-
2	{2, 3, 4, 7, 8}	{3, 4, 7, 8}	3	{1}
3	{3, 4, 7, 8}	{4, 7, 8}	4	-
4	{4, 7, 8}	{7, 8}	7	-
5	{5, 7, 8}	{7, 8}	7	{4}
6	{6, 7, 8}	{7, 8}	7	{4}
7	{7, 8}	{8}	8	-
8	{8}	-	-	{7}
9	{1, 3, 4, 7, 8, 9}	{1, 3, 4, 7, 8}	1	{7}

Tabulka 3.2: Získané informace o post-dominanci mezi jednotlivými uzly z příkladu na obrázku č. 3.3

3.1.2 Smyčky v grafu toku řízení

V grafu toku řízení se běžně vyskytují smyčky (cykly). Ty představují neprázdnou množinu uzlů, které jsou dosaženy samy ze sebe. Přesněji se jedná o silně souvislou komponentu v grafu toku řízení neboli množinu uzlů $S \subset N$, kde $q, r \in S$ a existuje cesta z uzlu q do r i cesta z r do q. [18] Maximálně silně souvislý podgraf je takový podgraf, do kterého nelze přidávat žádné další uzly, aby zůstal silně spojený. **Vnější smyčka** (outermost loop) je maximální silně souvislý podgraf s alespoň jednou vnitřní hranou. Každý cyklus má **vstupní uzel**. Jedná se o speciální uzel $h \in S$, který má přímého předchůdce $p \notin S$. Každá cesta od vstupního bodu v grafu toku řízení do libovolného bodu v S prochází bodem h, jedná se tedy o dominátora pro všechny uzly v cyklu S. [7] Smyčky mohou obsahovat i více **vnořených smyček** (nested loops). Vnořená smyčka ve smyčce S s hlavičkou h je silně souvislá komponenta L, kde $L \subset S$ a uzel $h \notin L$.

Identifikace cyklu v grafu toku řízení je poměrně snadná, stačí nalézt takzvanou **zpětnou hranu** (back edge) vedoucí z uzlu n_j do uzlu n_i, kde n_i je zároveň dominátorem pro n_j. Pro vyhledávání zpětných hran se obvykle používá algoritmus prohledávání do hloubky (DFS).

Smyčky můžeme rozdělit do dvou odlišných kategorií, a to redukovatelné a neredukovatelné. Redukovatelné smyčky obsahují pouze jeden vstupní uzel smyčky, kdežto neredukovatelné mají vstupních uzlů více. Mezi redukovatelné patří i **přirozené smyčky** (natural loops). Jedná se o nejmenší sadu uzlů, která obsahuje zpětnou hranu vedoucí do vstupního uzlu smyčky a zároveň uzly v sadě nemají žádného předchůdce mimo sadu, kromě vstupního uzlu smyčky.

Pokud se cyklus v grafu nevyskytuje, říkáme, že graf je acyklický. Programovací jazyky běžně disponují konstrukcemi, které cykly v grafu toku řízení vytváří, jako například *for*, *loop*, *while* atd. Smyčku lze vytvořit i pomocí přímé či nepřímé rekurze při volání funkcí, procedur nebo metod.

Některé smyčky mohou mít při vykonávání nekonečně mnoho iterací. Tyto **nekonečné smyčky** nemají žádnou ukončovací podmínku nebo ji mají nastavenou tak, že nebude nikdy splněna. Někdy jsou tyto smyčky vytvořeny programátorem nevědomky a můžou způsobit zamrznutí aplikace. Nekonečné smyčky mohou být v programu objeveny pečlivou kontrolou kódu, ale neexistuje žádná obecná metoda, která by dokázala určit, zda daný program někdy skončí, nebo bude fungovat navěky.

3.1.3 DFST

Optimalizace programů může být provedena zredukováním některých jeho příkazů. Pro určení, zda je graf toku řízení možno redukovat, se používá depth-frist spanning tree (DFST). Při prohledání grafu toku řízení do hloubky (depth-first search) se nejdříve začíná vstupním uzlem a postupuje se do následníka každého vrcholu, pokud jej ještě nenavštívil. Pokud jej již navštívil nebo z uzlu již nevede hrana, vrací se zpět pomocí backtrackingu a pokračuje prohledávání doposud nenavštívených uzlů. Trasa tohoto hledání vytváří DFST. [7] Po vytvoření DFST můžeme hrany v grafu toku řízení rozdělit do tří kategorií.

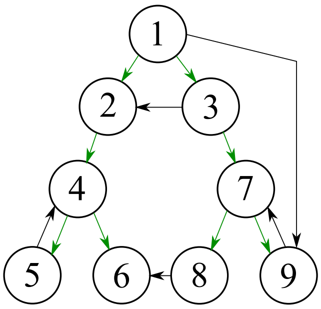
- **Postupující hrana** vede z uzlu m do jakéhokoliv jeho potomka. Do této kategorie patří všechny hrany v DFST. V grafu se mohou vyskytnout i hrany, které jsou postupující a zároveň patří do této kategorie.

- **Příčná hrana** propojuje dva uzly m a n, kde m není předek ani potomek n v DFST.

- **Ustupující hrana** vede od uzlu m k předchůdci m v DFST nebo zpět do uzlu m.

Všechny zpětné hrany ve smyčkách (viz sekce - 3.1.2) jsou zároveň ustupující, ale ustupující hrany nemusí být nutně zpětnými. Graf toku řízení je redukovatelný, pokud všechny ustupující hrany v DFST jsou zároveň zpětnými. Pokud je graf redukovatelný, tak po odstranění zpětných hran nám vznikne acyklický graf.

Na obrázku č. 3.6 je zobrazen graf toku řízení. Zeleně jsou v něm vyznačeny hrany, které náleží DFST. V obrázku je kromě hran DFST další postupují hrana 1 → 9. Příčné hrany jsou v příkladu dvě 3 → 2 a 8 → 6. Hrany 5 → 4 a 9 → 7 patří do kategorie hran ustupujících. Tento graf však není redukovatelný, protože ustupující hrana 9 → 7 není zároveň zpětná.

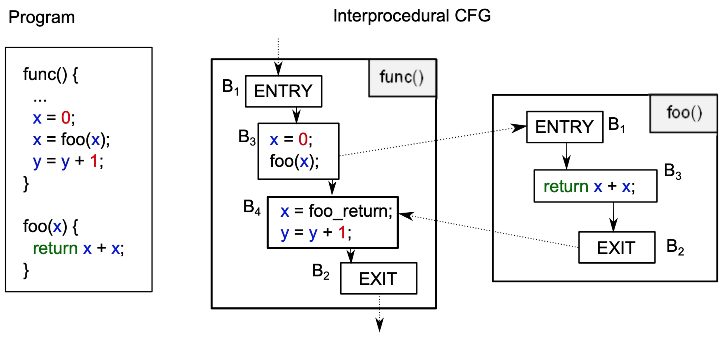


Obrázek 3.6: Ukázka grafu toku řízení, ve kterém je vyznačen DFST

Při použití programových konstrukcí, jako *while-loops*, *for-loops*, *repeat-loops*, *if-then(-else)* apod. je graf toku řízení vždy redukovatelný. Některé programovací jazyky, mezi které patří například Java, jsou stavěny tak, aby vytvářely pouze redukovatelný kód. Příkaz *GOTO*, který se může objevit například v kódu napsaném v programovacím jazyce C, C++ nebo FORTRAN, může vytvořit neredukovatelný kód. [26, 33]

3.2 Interprocedurální graf toku řízení

Obecně bývá graf toku řízení konstruován individuálně z metod, procedur nebo podprogramů programu, které mají jeden vstupní základní blok. Tok řízení mezi metodami je reprezentován volajícími základními bloky. Propojením grafů toku řízení jednotlivých procedur, podle těchto volajících bloků vznikne interprocedurální graf toku řízení - ICFG. [13] Jedná se vlastně o kolekci grafů toků řízení $\{G_i\}$, kde G_i reprezentuje proceduru v programu. [27] ICFG může obsahovat více vstupních a výstupních uzlů, protože nemusí nutně rozvíjet volání v jedné hlavní proceduře.



```
func() {
  ...
  x = 0;
  x = foo(x);
  y = y + 1;
}

foo(x) {
  return x + x;
}
```

Obrázek 3.7: Příklad interprocedurálního grafu toku řízení [13]

3.3 Control flow testování

Testování softwaru na základě toku řízení (control flow) se řadí mezi strukturální testování, které využívá tok řízení programu jako model. Vyžaduje znalost kódu a zaměřuje se především na pokrytí velkého množství cest v programu.

Testovací nástroj nejdříve vytvoří graf toku řízení z testovaného kódu. Poté zkoumá všechny možné cesty vedoucí od vstupu programu k jeho ukončení a snaží se pro tyto cesty vygenerovat příslušná testovací data. Tento úkol není jednoduchý, neboť graf toku řízení programu může být velmi složitý, mohou v něm existovat nedosažitelné cesty nebo lze narazit na problémy popsané v sekci - 2.5. [8, 11, 29]

3.4 Nástroje pro extrakci toku řízení

V dnešní době existuje mnoho nástrojů pro extrakci grafu toku řízení z programového kódu různých programovacích jazyků. Mnoho nástrojů je jednoúčelových a jejich použití je velmi omezené. Vzhledem k tématu diplomové práce se zaměřím pouze na nástroje, které extrahují graf toku řízení z kódu či bytecodu programovacího jazyka Java.

3.4.1 TRGeneration

Jedním z velmi jednoduchých programů na extrakci grafu toku řízení z Java kódu je TRGeneration[1]. Jde spíše o návrh softwaru, protože dokáže analyzovat jen omezenou sadu příkazů a jeho použití v praxi by vyžadovalo velký zásah do zdrojového kódu nástroje. TRGeneration konstruuje graf toku řízení při načítání souboru, který obsahuje **pouze** tělo metody zdrojového kódu Java (viz příloha A). Interní reprezentaci grafu dokáže převést do DOT formátu a za pomoci externí knihovny Graphviz (viz sekce - 5.6) i vizualizovat.

Rok verze	*2014*
Licence	*neuvedena*
Stav projektu	*neaktivní*
Technologie	*Java*
Vstupní soubory	*zdrojové kódy těl metod Java*
Dokumentace	*bez dokumentace*

Tabulka 3.3: Informace o TRGeneration

3.4.2 Control Flow Graph Factory

Jedná se o zásuvný modul do vývojového prostředí Eclipse. Byl vytvořen společností Dr. Garbage jako jeden ze sady nástrojů s licencí s otevřeným zdrojovým kódem (open source). Control Flow Graph Factory[2] dokáže generovat řídící toky grafů z bytecodu, upravovat je a exportovat do různých formátů, jako je například DOT nebo GraphXML. Jeho předchůdcem byl plugin nazývaný po vyvíjející společnosti tedy Dr. Garbage.

---

[1] `https://github.com/evplatt/TRGeneration`

[2] `https://marketplace.eclipse.org/content/control-flow-graph-factory`

Rok první verze	*2008*
Rok poslední verze	*2019*
Licence	*Apache 2.0*
Stav projektu	*aktivní* *(s ohledem na podporu verze Eclipsu 4.9)*
Technologie	*Java*
Vstupní soubory	*java bytecode*
Dokumentace	*bez dokumentace*

Tabulka 3.4: Informace o Control Flow Graph Factory

3.4.3 PROGEX

Program Graph Extractor neboli PROGEX[3] je nástroj pro získávání různých grafických reprezentací (CFG, PDG, AST) ze zdrojového kódu zkoumaného programu. Software je napsaný v programovacím jazyce Java a pro zpracování obsahu vstupních souborů využívá nástroj ANTLR[4]. Vytvořené grafy dokáže uložit do různých grafických formátů, jako jsou například JSON, DOT atd.

Rok release verze 2.0	*2017*
Rok beta verze 3.0	*2019*
Licence	*Apache 2.0*
Stav projektu	*aktivní*
Technologie	*Java*
Vstupní soubory	*zdrojový kód Java*
Dokumentace	*bez dokumentace*

Tabulka 3.5: Informace o nástroji PROGEX

3.4.4 ASM

Velmi dobrým analytickým aplikačním frameworkem pro analýzu bytecodu Java je ASM[5]. Může být použit pro úpravu existujících tříd nebo k dynamickému generování tříd přímo v binární podobě. Dokáže také vytvořit graf toku řízení pro konkrétní metodu. ASM je zaměřen na výkon, byl tedy implementován tak, aby byl co nejmenší a nejrychlejší. Komunikace s tímto

---

[3] `https://github.com/ghaffarian/progex`

[4] `https://www.antlr.org/`

[5] `https://asm.ow2.io/`

frameworkem je pomocí jednoduchého modulárního API, které je dobře zdokumentováno[6]. Stejně tak jako Control Flow Graph Factory, tak také celý framework ASM může být použit jako zásuvný modul do vývojového prostředí Eclipse a umožňuje uživateli svobodné užívání softwaru k různým účelům. Při hledání nástrojů pro extrakci grafu toku řízení jsem se setkal z mnoha nástroji, které jsou založeny na tomto frameworku. ASM je také jeho komunitou neustále vyvíjen a udržován, o čemž svědčí i jeho poslední verze ASM 7.1, která byla vydána v březnu 2019. Díky vlastnostem tohoto aplikačního frameworku je vhodný pro použití ve velkém množství aplikací. [9]

Rok verze	*2019*
Licence	*3-Clause BSD License*
Stav projektu	*aktivní*
Technologie	*Java*
Podpora verze Java	*Java SE 11*
Vstupní soubory	*java bytecode*
Dokumentace	*Ano*

Tabulka 3.6: Informace o frameworku ASM

3.4.5 Soot

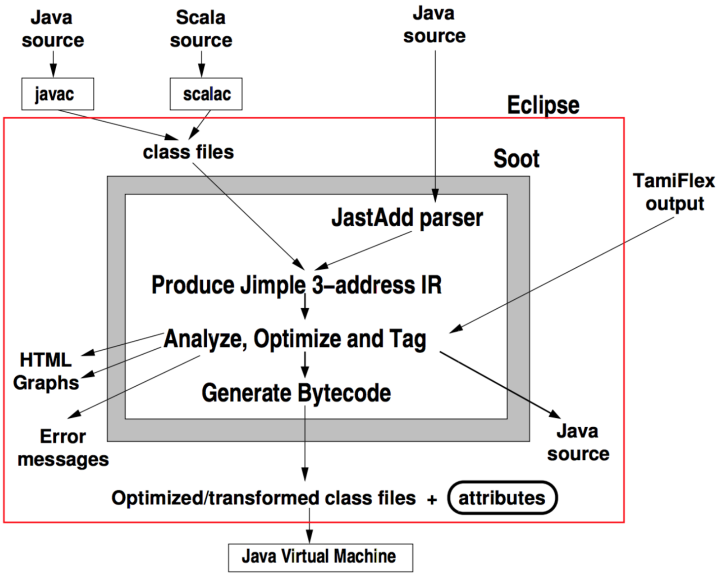
Soot je stejně jako ASM úspěšný optimalizační framework, jehož první verze byla vydána již kolem roku 2000 výzkumnou skupinou Sable na univerzitě McGill v Kanadě. Cílem tohoto frameworku je poskytnout nástroje vedoucí k lepšímu porozumění a rychlejšímu provádění programů Java. Poskytuje čtyři mezilehlé reprezentace pro analýzu a transformaci Java kódu.

- **Baf** - zjednodušená reprezentace bytecodu.
- **Jimple** - mezilehlá reprezentace tříadresního kódu, vhodná pro optimalizaci. Může být vytvořen ze zdrojového kódu či bytecodu Java.
- **Shimple** - SSA forma Jimple.
- **Grimp** - agregovaná verze programu Jimple, která je vhodná pro dekompilaci a kontrolu kódu. V porovnání s Jimple blíže připomíná zdrojový kód Java a je snadnější pro čtení.

---

[6] `https://asm.ow2.io/asm4-guide.pdf`

Framework je rovněž překladačem. Díky tomu lze načíst zdrojový či bytecode Java, upravit jeho mezilehlou reprezentaci a nechat přeložit do spustitelného bytecodu. Pro lepší představu můžeme vidět na obrázku č. 3.8 diagram překladu kódu či bytecodu Java ve vývojovém prostředí Eclipse za použití tohoto frameworku.



Obrázek 3.8: Diagram frameworku Soot v prostředí Eclipse [24]

Soot[7] negeneruje graf toku řízení z Java programu přímo, ale dokáže graf vygenerovat z jeho mezilehlých reprezentací. Tvůrci Soot se při vývoji spíše zaměřili na analýzu Java programů z bytecodu. Aktuálně framework dokáže načítat .class soubory kompilované pro verzi Java SE 9. Při načítání zdrojových souborů má však problém a některé programové konstrukce, které přináší novější verze jazyka, nedokáže zpracovat. Proto se při práci se zdrojovými soubory .java doporučuje pracovat ve verzi Java SE 7. Soot má dobrou přehlednou dokumentaci[8] i zdrojové kódy dostupné na githubu[9]. [15, 24]

---

[7] `http://sable.github.io/soot/`

[8] `https://www.brics.dk/SootGuide/`

[9] `https://github.com/Sable/soot`

Rok vytvoření	*2006*
Rok poslední verze	*2019*
Licence	*GNU Lesser General Public License*
Stav projektu	*aktivní*
Technologie	*Java*
Podpora verze Java	*Java SE 7*
Vstupní soubory	*java bytecode a zdrojový kód*
Dokumentace	*Ano*

Tabulka 3.7: Informace o frameworku Soot

4 Navrhované řešení

Cílem této diplomové práce je navrhnout metodu pro získání parametrů jednotkových testů pro programy vytvořené v jazyce Java. Tyto parametry mají být získávány procházením grafu toku řízení. Programových konstrukcí, které lze v programovacím jazyce Java sestrojit, je mnoho a nelze je všechny pokrýt v rozsahu této diplomové práce. Při návrhu metody pro generování testovacích dat jsem postupoval od úplně nejjednodušších primitivních datových typů až po složitějších konstrukce, jako například pole, objekty, rozhraní atd. V této kapitole rozeberu jednotlivé kroky a problémy, které mě dovedly ke stávajícímu návrhu.

4.1 Získání toku řízení z Java kódu

Abychom mohli navrhnout metodu, která dokáže vygenerovat testovací data z toku řízení programu, je prvním úkolem samotné získání grafu toku řízení. V kapitole 3.4 bylo popsáno několik dostupných nástrojů pro získání grafu toku řízení z Java kódu. Všechny tyto nástroje vytváří graf pomocí statické analýzy, proto bude také základ navrhované metody **založen na statické analýze kódu**.

Při bližším testování nástrojů jsem se rozhodl pro framework Soot, který dokáže načíst a zpracovat i .java soubory, které ve většině případů obsahují více informací než zkompilovaný bytecode. Další výhodou tohoto nástroje je jeho interní reprezentace testovaného kódu (viz sekce - 4.1.1), která poskytuje mnoho užitečných funkcí. Soot také dokáže vygenerovat pro graf řízení toku příslušný strom dominance a post-dominance (viz sekce - 3.1.1). Tato funkcionalita se hodí při analýze závislostí jednotlivých větví v grafu toku řízení. Hlavní nevýhodou tohoto frameworku je slabší podpora nejnovějších verzí Javy použité ve zdrojovém kódu testovaného programu.

Protože framework negeneruje graf toku řízení přímo, ale z jeho mezilehlých reprezentací (viz sekce - 3.4.5), bude se lišit graf toku řízení skutečného programu od grafu toku řízení vygenerovaného pomocí Soot. Odlišnosti jsou však jen v počtu a obsahu základních bloků. Výsledná funkcionalita a struktura grafu vygenerovaného pomocí Soot by měla být totožná se skutečnou funkcionalitou programu. Příklad si můžeme ukázat na jednoduchém následujícím kódu, který zjišťuje nadváhu osoby pomocí body mass indexu.

```
boolean isOverweight(double weight, double height){
        double bmi = weight / (height * height);
        if (bmi > 25) {
            return true;
        }
        return false;
}
```

Zdrojový kód 4.1: Ukázka metody v programovacím jazyce Java pro zjištění nadváhy osoby pomocí body mass indexu

Zpracováním předešlého kódu pomocí Soot získáme Jimple reprezentaci, která je znázorněna níže. Můžeme si všimnout, že díky tříadresnímu kódu jsou složitější výrazy rozloženy. To je velká výhoda při následující analýze, jelikož nemusíme řešit například pořadí operátorů. Abychom lépe pochopili vytváření grafu toku řízení a celkovou práci s frameworkem je nutné v následující sekci - 4.1.1 nástroj detailněji popsat.

```
boolean isOverweight(double, double) {
        object.Person this;
        double weight, height, bmi, $d0;
        byte $b0;

        this := @this: object.Person;
        weight := @parameter0: double;
        height := @parameter1: double;
        $d0 = height * height;
        bmi = weight / $d0;
        $b0 = bmi cmpl 25.0;
        if $b0 <= 0 goto label0;

        return 1;

     label0:
        return 0;
}
```

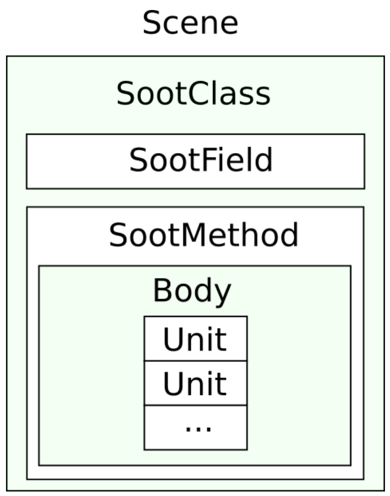
Zdrojový kód 4.2: Jimple reprezentace metody pro počítání nadváhy osoby

Z vygenerovaného kódu si můžeme všimnout, že Soot při překladu Java kódu do vnitřní formy Jimple použil před podmínkou konstrukci *cmpl* (viz sekce - 4.2.4) a prohodil operátor porovnání > za <=. Kód je však reprezentován správně, neboť prohodil i pořadí dosažených uzlů při splnění a nesplnění podmínky.

4.1.1 Soot interní reprezentace

Soot je psaný v jazyce Java a lze ho bez nesnází do projektu nahrát například jako maven projekt. Poskytuje příjemné API, přes které se s nástrojem pracuje. Soot při načítání vytváří několik datových struktur, se kterými je potřeba pro generování toku řízení pracovat, jako je například:

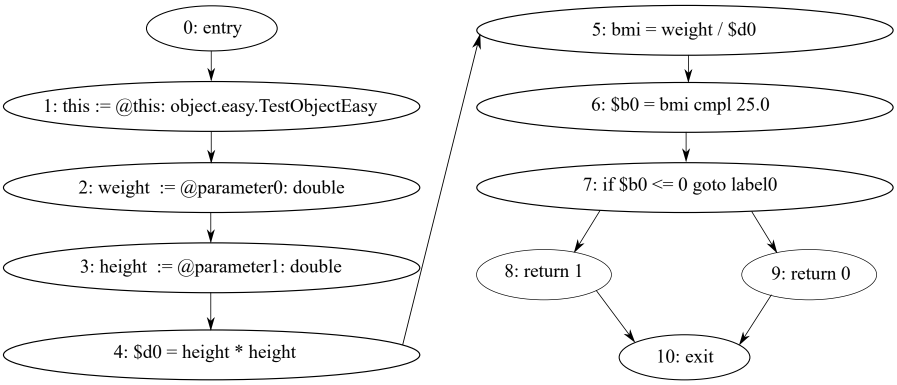
- **`Scene`** - představuje aplikaci, která je v Soot analyzována. Obsahuje všechny třídy (`SootClass`), které se podařilo z aplikace načíst. Způsob načtení tříd lze ovlivnit mnohými nastaveními, která se nastavují pomocí třídy **`Options`**.
- **`SootClass`** - reprezentuje třídu Java objektu. Má v sobě referenci na metody (`SootMethod`) a atributy (`SootField`). V Soot existují čtyři různé typy tříd:
  - **`Application`** - aplikační třídy, které lze plně kontrolovat a měnit.
  - **`Library`** - třídy knihoven. Mohou být libovolně kontrolovány, ale nelze je měnit.
  - **`Context`** - u těchto tříd není známa implementace, ale mohou obsahovat informace o metodách a atributech, které třída obsahuje. Přístup k implementaci vyvolá výjimku.
  - **`Phantom`** - třídy, o kterých je známo, že existují. Je na ně odkazováno v třídách aplikace, ale Soot je nemohl načíst. Mohou také obsahovat phantomová pole a phantomové metody.
- **`SootMethod`** - představuje metodu třídy. Obsahuje všechny potřebné informace o metodě, jako je například počet a typ jednotlivých parametrů. Při detailnějším zkoumání jsem zjistil, že Soot aktuálně **nepodporuje přetížení metod**. To je první omezení, které použití tohoto frameworku přináší.
- **`Body`** - rozhraní, které reprezentuje tělo metody, ze kterého se graf toku řízení generuje. Skládá se ze základních příkazů, které jsou reprezentovány pomocí rozhraní `Unit`.
- **`Unit`** - nejdůležitější rozhraní v Soot, protože slouží k reprezentaci pokynů nebo výkazů. V zobrazení kódu v Jimple formě je `Unit` tříadresní příkaz.



Obrázek 4.1: Soot interní reprezentace

- **`BriefUnitGraph`** - nejjednodušší reprezentace grafu toku řízení, která neobsahuje tok řízení vzhledem k výjimkám.
- **`ExceptionalUnitGraph`** - složitější struktura, která zachycuje i hrany pro výjimky (např. try, catch a throw klauzule). Bere v úvahu výjimky, které mohou být implicitně vytvořeny VM (např. `ArrayIndexOutOfBoundsException`).
- **`TrapUnitGraph`** - struktura podobná `ExceptionalUnitGraph`. Hlavní rozdíly jsou ve způsobu propojení uzlů při možném vyhození výjimky. Například pokud z každého `Unit` v try-catch bloku vede hrana do obsluhy výjimky.

Pro vygenerování konkrétního grafu toku řízení nejdříve potřebuje načíst metodu třídy, kterou chceme testovat. Poté z metody reprezentované rozhraním `SootMethod` získáme `Body`, ze kterého vygenerujeme konkrétní graf toku řízení. Na obrázku č. 4.2 můžeme vidět graf typu `BriefUnitGraph` vygenerovaný z metody pro počítání nadváhy člověka, který je uveden v sekci - 4.1.



Obrázek 4.2: Graf toku řízení vytvořený pomocí Soot

Je-li v metodě volána jiná metoda, Soot volání metody vyhodnotí jako jeden uzel v grafu toku řízení. Pokud chceme rozvinout i volanou metodu, je zapotřebí z uzlu reprezentovaným `Unit` získat volanou metodu (`SootMethod`). Dostaneme-li volanou metodu, můžeme se pokusit z jejího těla vygenerovat její tok řízení. Soot tedy volané metody sám automaticky nerozvíjí a chce-li uživatel procházet všechny možné cesty i v metodách volaných, je na uživateli, aby si graf toku řízení z metod získal.

4.2 Analýza toku řízení

Získáme-li graf toku řízení, je možné začít s jeho analýzou, pomocí které chceme získat parametry pro jednotkové testy. Vygenerované hodnoty budou tedy přímo odpovídat vstupním parametrům testované metody. Z grafu toku řízení můžeme vyčíst jakou roli má parametr na výběr možné cesty programem a vygenerovat takové hodnoty, abychom při testování pokryli co nejvíce možných cest a dosáhli tím maximálního pokrytí testované metody.

Během procházení grafu je nutné **analyzovat všechny uzly**, které do daného uzlu vedou, neboť proměnná obsažená v uzlu může být v předchozích uzlech měněna. Abychom tedy vůbec dokázali vyhodnotit nějakou proměnnou, je zapotřebí uchovávat její stav. K tomuto účelu jsem navrhl **tabulku symbolů**, viz sekce - 4.2.2.

Dalším důležitým rozhodnutím je, jak bude docházet k procházení grafu toku řízení či jeho jednotlivých cest. V úvahu připadají dvě možné varianty, a to buďto od jeho začátku tj. od vstupního uzlu, nebo od uzlů koncových. Protože vstupní bod testované metody je pouze jeden a na začátku těla každé metody v Jimple formě jsou deklarovány použité proměnné, rozhodl jsem se pro procházení grafu od jeho **vstupního uzlu**.

4.2.1 Prořezávání hran grafu

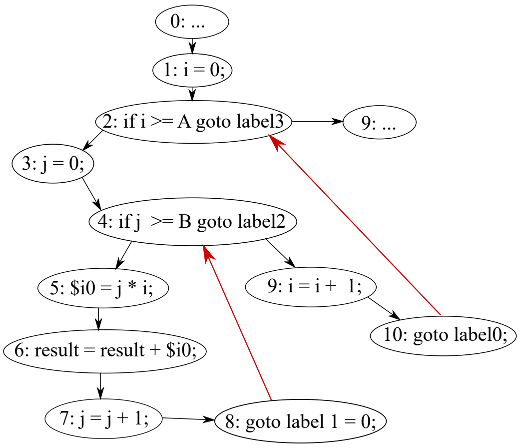
V mnoha grafech řídicích toků se vyskytují smyčky (viz sekce - 3.1.2), které komplikují jejich postupný průchod. Konstrukce, které cykly v grafu toku řízení v programovacím jazyce Java vytváří, jsou obvykle příkazy *for* a *while*. Dalším způsobem, jak může cyklus vzniknout, je například použitím přímé či nepřímé rekurze.

Při statické analýze grafu toku řízení jsou cykly problematické především proto, že ve většině případů nedokážeme ukončovací podmínku cyklu správně vyhodnotit. Nedokážeme tedy ve všech případech jasně určit, kolikrát se tělo cyklu provede a jaký dopad průchod cyklem bude mít na stav použitých proměnných.

Abychom se vyhnuli komplikacím, které cykly při analýze grafu toku řízení přináší, budeme hrany, které vedou k vytvoření cyklu odstraňovat. Díky tomu se stane z grafu toku řízení graf **acyklický**. Tento krok si můžeme dovolit díky interní reprezentaci Soot, kde jsou všechny cykly tvořeny pomocí příkazů *if* a *goto* (protože interní reprezentace Soot vychází z kódu Java, vytváří příkaz *goto* vždy redukovatelný graf toku řízení). Jednoduchý zdrojový kód č. 4.3 je převeden na graf toku řízení zobrazený na obrázku č. 4.3. Při prořezávání hran jsou odstraněny pouze ty hrany, které vedou z příkazu *goto* zpět do rozhodovací podmínky tvořené příkazem *if*, **zpětné hrany** popsané v sekci - 3.1.2. Během následného prohledávání grafu jsou tedy dosaženy všechny uzly v těle cyklu i uzly po skončení smyčky.

```
...
for (int i = 0; i < A; i++) {
    for (int j = 0; j < B; j++) {
        result += j*i;
    }
}
...
```

Zdrojový kód 4.3: Ukázka zdrojového kódu obsahující dva cykly

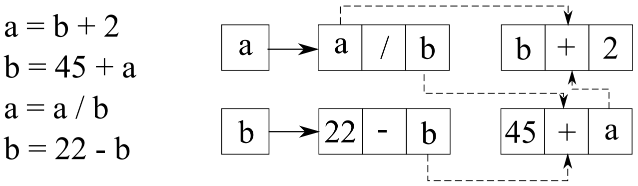


Obrázek 4.3: Graf toku řízení pro zdrojový kód č. 4.3

4.2.2 Tabulka symbolů

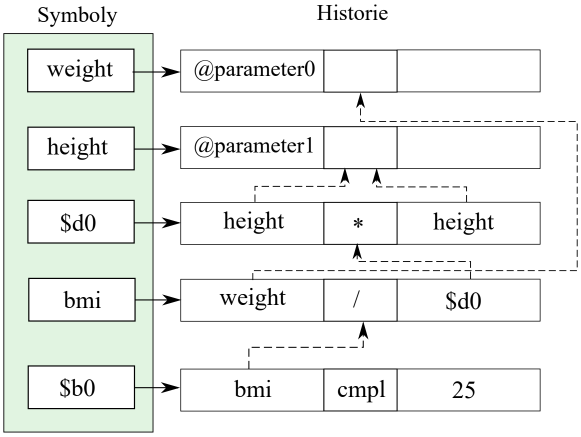
Procházíme-li postupně testovaný program, je zapotřebí ukládat stav použitých symbolů. Nejvíce nás budou zajímat symboly, které jsou použité v rozhodovacích podmínkách, neboť stav těchto symbolů rozhoduje, jakou cestou kódem se vykonávaný program vydá. Pokud použitý symbol není závislý na vstupním parametru metody, ale například na aktuálním stavu systému, je jeho vyhodnocení pomocí statické analýzy v některých případech nemožné. Podmínky, které nejsou závislé na vstupním parametru, nebudeme proto vyhodnocovat.

Tabulka symbolů je mnou navržená struktura, která v sobě uchovává jednotlivé symboly (nejedná se o tabulku symbolů, která se běžně používá pro překlad nebo interpret pro uložení všech identifikátorů nalezených ve zdrojovém kódu programu). Každý symbol v tabulce symbolů odkazuje na svou historii (může být implementována například spojovým seznamem). Historie se skládá z jednotlivých buněk, které obsahují informaci o konkrétním stavu. Díky tříadresnímu kódu, který je v Jimple formě použitý, se každá buňka skládá maximálně ze **dvou operandů** a jednoho **operátoru**. Buňky jsou navzájem propojeny s ostatními buňkami historie symbolů, podle symbolů v nich použitých. Je nutné uchovávat veškeré informace o změnách stavu objektu, neboť aktuální stav může být závislý na jeho předešlém stavu. Příklad jednoduché tabulky symbolů pro dva výrazy **a** a **b** je možno vidět na obrázku č. 4.4. Každý jednotlivý operand v buňce může odkazovat právě na jinou buňku, ze které se dozvíme informaci o jeho předešlém stavu.



Obrázek 4.4: Tabulka symbolů obsahující dva symboly

Sestrojenou tabulku symbolů z příkladu pro počítání nadváhy osoby v sekci - 4.1 můžeme vidět na obrázku č. 4.5. Na tomto příkladě lze vidět propojení symbolu $b, který je použitý v podmínce s oběma vstupními parametry.

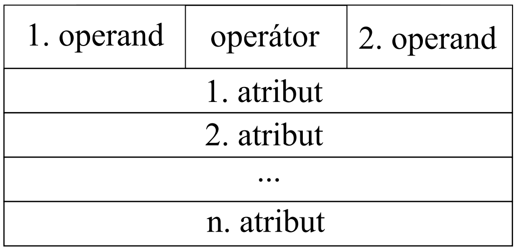


Obrázek 4.5: Tabulka symbolů pro příklad ze sekce - 4.1

Při procházení grafem řízení toku může nastat situace, kdy do jednoho uzlu vede více cest. Je tedy zřejmé, že tabulka symbolů může být pro každou zvolenou cestu odlišná.

Objekty

Tabulka symbolů může obsahovat i složitější datové typy například objekty. Ty na rozdíl od primitivních datových typů mohou uchovávat i informace o jejich **atributech**. Protože objekt může být v průběhu vykonávání měněn, musí být informace o atributu přímo vázána k buňce historie. Struktura buňky historie bude mít tedy rozšířenou strukturu, která je patrná z obrázku č. 4.6.



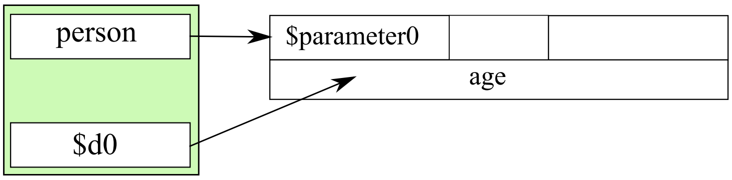
Obrázek 4.6: Struktura buňky historie symbolu

Atribut této buňky nese také informaci o jeho historii a chová se podobně jako buňka historie. Může odkazovat na jiné buňky nesoucí informaci o jeho předešlých stavech podle použitých operátorů. Abychom si lépe představili strukturu tabulky, kde jsou použity objekty, využijeme část následujícího příkladu.

```
boolean limitedAge(package.Person) {
                ...
        person := @parameter0: package.Person;
        $d0 = person.age;
        if $d0 >= 18 goto label0;
                ...
}
```

Zdrojový kód 4.4: Metody pro počítání věkové hranice osoby v Jimple kódu

V příkladu je uveden objekt `Person`, který vstupuje do metody jako `@parameter0`. Je zde uvedena jediná podmínka, která rozhoduje o dalším průběhu programu. V této podmínce se objevuje symbol `$d0`, který ukazuje na věk vstupní třídy. Celá provázanost buněk je k vidění na obrázku č. 4.8. Z jejich propojení je pak možno usoudit, že podmínka je závislá na stavu atributu vstupní třídy, konkrétně parametru `age`. Generovaná testovací data pro tento parametr budou rozhodovat, jakou cestu kódem vykonávaný program zvolí.



Obrázek 4.7: Tabulka symbolů obsahující objekt

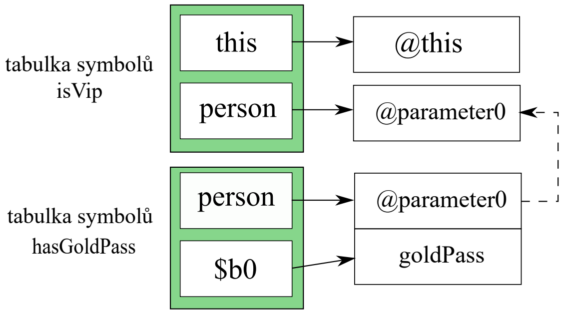
Volání metod

V těle analyzované metody mohou být volány i další metody. Pro oddělení lokálních proměnných každé metody, musíme vytvořit pro každou metodu **vlastní tabulku symbolů**. Při následné analýze je důležité vědět, jaká metoda z jakého objektu je volána a jaké jsou vstupní parametry. Bez těchto údajů bychom nemohli vyhodnotit závislost parametrů na výběru cesty v grafu toku řízení volané metody. Buňky historie v jedné tabulce symbolů mohou mít referenci na buňky historie nadřazené tabulky, ne však opačně.

```
boolean isVip(package.Person) {
        this := @this: someObject;
        person := @parameter0: package.Person;
        return this.hasGoldPass(person);
}

boolean hasGoldPass(package.Person) {
        person := @parameter0: package.Person;
        $b0 = person.goldPass;
        if $b0 == 1     return true;
        return false;
}
```

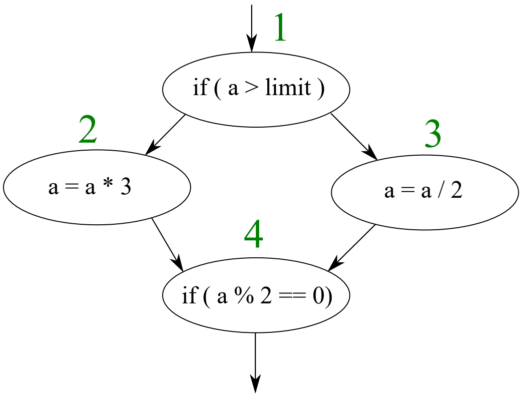
Zdrojový kód 4.5: Dvě metody v Jimple pro ukázku volání metod



Obrázek 4.8: Tabulky symbolů pro metodu `isVip` a `hasGoldPass`

4.2.3 Rozhodovací podmínky

Při analýze grafu toku řízení nás nejvíce zajímají rozhodovací podmínky, které mají vliv na možné větvení programu. V těchto podmínkách budeme zkoumat, zda jsou symboly, které se v těchto podmínkách vyskytují, závislé na vstupním parametru metody. Abychom mohli podmínky vyhodnocovat, potřebujeme vždy k příslušné podmínce znát aktuální tabulku symbolů. Jak již bylo řečeno v sekci - 4.2.2, pro každý uzel v grafu toku řízení může mít tabulka symbolů odlišné hodnoty. Některé uzly mohou mít i **více tabulek symbolů**, například uzel č. 4 na obrázku č. 4.9 může mít dvě tabulky symbolů v závislosti na výběru cesty. Pokud se rozhodneme pro cestu z uzlu č. 1 do uzlu č. 3, bude tabulka symbolů v uzlu č. 4 obsahovat hodnotu pro symbol a dvakrát menší. Když se však rozhodneme jít druhou cestou (z uzlu č. 1 do uzlu č. 2), bude hodnota pro symbol a v uzlu č. 4 třikrát větší.



Obrázek 4.9: Ukázka odlišnosti tabulek symbolů při výběru cesty

Získáme-li z grafu toku řízení pro rozhodovací uzly všechny možné tabulky symbolů, lze zjišťovat jejich vztah se vstupními parametry (postupným procházením vztahů v buňkách historie). Díky Jimple formě, se kterou pracujeme, se v rozhodovacích podmínkách vyskytují jen dva operandy a jeden relační operátor. Skládá-li se podmínka v jazyce Java z více logických operátorů && nebo ||, je při překladu do Jimple formy rozložena na několik konstrukcí *if*.

Ze zkoumané podmínky, která rozhoduje o větvení, se pokusíme nejdříve vyjádřit vztah operátorů se vstupním parametrem. Při tomto vyhodnocení může nastat několik následujících typů vztahů:

1. **Operandy a parametry nejsou na sobě závislé**. Průběh podmínky, ve které operátory nejsou závislé na vstupním parametru metody, neovlivníme žádnou konfigurací vstupních parametrů. V některých případech je ale nutné inicializovat některé objekty, abychom mohli volání metod vůbec provádět. Inicializace těchto objektů pak není řízená grafem řízení toku programu, ale zcela náhodně.

2. **Operandy a parametry jsou na sobě závislé**, ale některý z použitých symbolů **nedokážeme vyjádřit** pomocí statické analýzy kódu.

Vstupní parametr tedy ovlivní, jakou cestu vykonávaný program zvolí. Problém je v určení konkrétních hodnot pro symboly, které nejsou parametry a jejich stav je inicializován (znám) až za běhu programu (například podle stavu některého z objektů v systému).

3. **Operandy a parametry jsou na sobě závislé** a **dokážeme je vyjádřit** (například číselně) všechny použité symboly. Tento typ můžeme ještě rozdělit na dvě kategorie.
    - Operandy se skládají pouze ze vstupních parametrů nebo konstant. V tomto případě o splnění či nesplnění rozhodovací podmínky rozhoduje pouze konfigurace vstupních parametrů.
    - Operandy se skládají ze vstupních parametrů, konstant nebo symbolů, které mohou být inicializovány před voláním testované metody. Na tyto symboly můžeme nahlížet jako na vstupní parametry, které ale nejsou předány testované metodě přes vstupní parametry.

V diplomové práci se zaměřím **pouze na třetí typ**, kde v podmínce po dosazení symbolů zůstanou neznáme jen hodnoty pro vstupní parametry či symboly, které mohou být inicializovány před spuštěním testované metody. Pro první dvě varianty by se dal spíše využít nástroj založený na dynamické analýze kódu.

4.2.4 Rozdělení podmínek podle složitosti

Z předchozích sekcí jsme získali návod, jak vyhodnocovat, zda je podmínka závislá či nezávislá na vstupním parametru metody. Abychom mohli vygenerovat konkrétní hodnoty, musíme nejdříve do podmínek dosadit hodnoty symbolů, které známe a ponechat jen vstupní parametry. Po dosazení získáme rovnici či nerovnici, kterou je potřeba vyřešit pro následné generování vstupních dat metody. Podmínky v rozvinutém tvaru mohou být dost komplikované, proto je rozdělím do dvou kategorií podle složitosti.

Do **první kategorie podmínek** patří pouze podmínky obsahující primitivní datové typy a jednoduché relační či aritmetické operátory. Za jednoduché relační operátory považuji všechny relační operátory v programovacím jazyce Java kromě operátoru *instanceof*, který pracuje s objekty. Do **druhé kategorie podmínek** jsou zařazeny všechny ostatní podmínky, které pracují s objekty nebo obsahují funkce.

První kategorie podmínek

Tato kategorie obsahuje pouze symboly vstupních parametrů představující **primitivní datové typy** a jednoduché relační a aritmetické operátory. Všechny primitivní datové typy v jazyce Java lze reprezentovat pomocí číselné hodnoty. Rozvinuté podmínky, které obsahují číselné neznámé a jednoduché operátory, můžeme vyhodnocovat stejně jako klasické **matematické rovnice** či **nerovnice**. Jako příklad si uvedeme algoritmus č. 4, kde nemusíme v podmínkách dohledávat vztahy symbolů s parametrem, ale v podmínkách je parametr rovnou uveden.

Algoritmus 4

```
function isTeenager(a)
    if $a >= 13$ then
        if $a <= 19$ then
            return true;
        else
            return false;
        end if
    else
        return false;
    end if
    return false;
end function
```

Při analýze grafu toku řízení jako první narazíme na podmínku $a \geq 13$. Tato podmínka nelze dále rozvinout, neboť parametr v ní obsažený se v předchozích uzlech neměnil. Tato podmínka je zařazena do první kategorie podmínek vzhledem k tomu, že obsahuje jen primitivní datové typy a jednoduché operátory. Z výše uvedeného vyplývá, že z řešené podmínky můžeme vytvořit matematickou rovnici, kterou se pokusíme vyhodnotit.

Abychom pokryli hrany v grafu toku řízení vedoucí z této podmínky, musíme vygenerovat testovací data vedoucí ke splnění i nesplnění podmínky. Budeme tedy hledat řešení pro rovnice:

$$a \geq 13, \tag{4.1}$$

$$a < 13. \tag{4.2}$$

Řešením těchto rovnic získáme intervaly $(-\infty, 13)$ a $\langle 13, +\infty \rangle$. Maximální a minimální hranice těchto intervalů jsou určeny podle datového typu parametru. Dosadíme-li jakoukoliv hodnotu z prvního intervalu, bude vždy podmínka vyhodnocena jako nepravdivá. U hodnot z druhého intervalu bude podmínka vždy pravdivá.

Při analýze aktuálního uzlu v grafu toku řízení musíme vzít v potaz i **uzly**, jejichž vyhodnocení má **vliv na dosažitelnost** zkoumaného uzlu. Například při vyřešení uzlu na řádce č. 3 s podmínkou $a \leq 19$ musíme brát v potaz i uzel na řádce č. 2, jehož rozhodovací podmínka musí být vyhodnocena jako pravdivá. Chceme-li vygenerovat testovací data, která pokryjí i uzly vedoucí z uzlu na řádce č. 3, musíme najít řešení pro následující rovnice:

$$a \geq 13 \cap a \leq 19, \tag{4.3}$$

$$a \geq 13 \cap a > 19. \tag{4.4}$$

Řešením těchto rovnic dostaneme opět intervaly, ze kterých můžeme získat testovací data. Na tomto vzoru jsem chtěl poukázat na narůstající složitost nerovnic při rostoucím větvení testovaného kódu.

Složitější analýzu grafu toku řízení můžeme předvést na příkladu pro počítání nadváhy osoby ze sekce - 4.1. V kódu je použita jen jedna rozhodovací podmínka

$$\$b0 <= 0. \tag{4.5}$$

Díky tabulce symbolů zjistíme, že podmínka je závislá pouze na vstupních parametrech. Můžeme ji tedy rozvinout do tvaru

$$(weight\,/\,(height\, *\, height))\ cmpl\ 25.0 <= 0. \tag{4.6}$$

Operátory **cmpl** a **cmpg** jsou v Jimple formě často použity. *Cmpl* operátor porovná oba své operandy a vrátí nulu, jsou-li hodnoty stejné, jedničku, je-li hodnota pravého operátoru menší a mínus jedničku, je-li pravý operátor větší. V případě, že jedna z hodnot je *NaN*, vrací mínus jedničku. *Cmpg* pracuje podobně jako *cmpl* jen v situaci, že jedna z hodnot je *NaN*, vrací jedničku. V Soot formě existuje ještě operátor *cmp*, který se používá při porovnávání primitivního datového typu *long*.

Používáme-li výsledek těchto dvou operátorů jako hodnotu pro porovnání s nulou, při ignoraci, že by některá z hodnot byla *NaN*, mohou být tyto operátory nahrazeny operátorem mínus. Příklady možných nahrazení operátorů můžeme vidět v tabulce č. 4.1.

Podmínka v jazyce Java	**Jimple forma**	**Alternativa**
$a >= b$	*a cmpg b >= 0*	*a-b >= 0*
$a <= b$	*a cmpl b <= 0*	*a-b <= 0*
$a < b$	*a cmpl b <0*	*a-b <0*
$b != a$	*b cmpg a != 0*	*b-a != 0*
$b == a$	*b cmpg a == 0*	*b-a == 0*
$b > a$	*b cmpg a >0*	*b-a >0*

Tabulka 4.1: Tabulka alternativního zápisu operátorů *cmpl* a *cmpg*

Nahradíme-li operátor cmpl v rovnici 4.6 jeho alternativním zápisem dostaneme nerovnici

$$(weight \,/\, (height * height)) - 25.0 <= 0. \tag{4.7}$$

Možnost alternativního zápisu operátorů *cmpl* a *cmpg* je poměrně důležitým krokem k vytvoření nerovnic, které již mohou být řešeny pomocí matematických nástrojů.

Druhá kategorie podmínek

Podmínky pro větvení v programovacím jazyce Java mohou obsahovat kromě primitivních datových typů i komplikovanější operátory a struktury. Jako příklad složitějších operátorů můžeme uvést operátory bitové, unární, ternární atd. Do složitějších struktur patří například objekty, rozhraní, pole či výčtové typy (enum). Podmínky z druhé kategorie musíme řešit individuálně podle použitých operátorů a struktur.

Jeden z relačních operátorů, který porovnává, zda objekt je instancí dané třídy nebo rozhraní, je ***instanceof***. Abychom dokázali vygenerovat testovací data, která zajistí pravdivost i nepravdivost podmínky, kde se tento relační operátor vyskytuje, musíme nejdříve zjistit, jakých typů instancí může analyzovaný objekt nabývat.

Jako příklad můžeme použít funkci z algoritmu 5, která vrátí pravdu, je-li vstupní objekt instancí třídy `Woman` a nepravdu je-li vstupní objekt instancí jiné třídy. Pro splnění podmínky na řádce č. 2 můžeme vygenerovat testovací data, kde vstupní parametr `person` je instancí třídy `Woman`. Pro zjištění jakých dalších typů instancí může třída `Person` nabývat, je zapotřebí **prohledat stávající projekt** a analyzovat třídy, které se v projektu vyskytují. Nalezneme-li třídu, která dědí od objektu `Person`, ale není instancí třídy `Woman`, můžeme tuto třídu použít při generování testovacích dat jako vzor pro nesplnění analyzované podmínky.

Algoritmus 5

```
function isWoman(Person person)
    if person instanceof Woman then
        return true;
    else
        return false;
    end if
end function
```

Mezi složitější objekty patří i **pole**. To může obsahovat nejen objekty, ale i primitivní datové typy. Pole může být použito v rozhodovacích podmínkách, kde se zkoumá například jeho délka či vlastnost konkrétního prvku na určitém indexu v poli. Schopnost vygenerovat konkrétní testovací hodnoty pro tento objekt závisí na tom, zda dokážeme vyčíslit index prvku pomocí statické analýzy, což v mnoha případech nelze.

Další často používanou třídou, která v programovacím jazyce Java reprezentuje textové řetězce je `String`. Tento objekt v sobě obsahuje pro reprezentaci řetězce pole primitivního datového typu `char`. Pro vytvoření testovacích dat pro tento objekt pomocí statické analýzy budeme muset řešit stejné problémy jako při generování testovacích dat pro objekt typu pole. Některé metody, které tato třída poskytuje, můžeme však řešit individuálně. Například metodu `equals`, která porovnává dva řetězce. Pro vyřešení podmínky, kde se vyskytuje volání této metody, můžeme vygenerovat sadu hodnot, kde se řetězce shodují a sadu hodnot, kde se neshodují. Tyto hodnoty můžeme vygenerovat bez vnitřního procházení volané metody.

Vyhodnocení podmínek patřící do druhé kategorie je velmi individuální na rozdíl od vyhodnocení podmínek z první kategorie, kde stačí vyřešit matematické nerovnice či rovnice.

4.2.5 Objekty a jejich závislosti

Programovací jazyk Java patří mezi jazyky objektově orientované. Programy napsané v těchto jazycích se skládají ze vzájemně propojených objektů. Toto propojení může vzniknout ve chvíli, kdy objekt v rámci své činnosti využívá činnost jiného objektu nebo dědí či implementuje vlastnosti objektu nadřazeného. O závislostech může rozhodovat například sám objekt, kdy jsou vytvořeny konkrétní instance závislých objektů v jeho konstruktoru, nebo mu mohou být předány z venčí jiným objektem přes jeho setry.

Graf toku řízení testované metody může obsahovat uzly, ve kterých se vyžaduje činnost závislého objektu. Tyto uzly nemusí nutně obsahovat rozhodovací podmínku, která by vedla na větvení programu, ale při neinicializaci závislého objektu může dojít k **softwarovému přerušení** (výjimce), které zabrání v následném pokračování vykonávané metody.

Při statické analýze kódu je problematické určit, jaké konkrétní instance objektů jsou v metodách testovaného objektu použity. Další stěžejní je samotné vytváření objektů, které nemají veřejný konstruktor nebo jejich konstruktor vyžaduje referenci na další objekty.

Pro usnadnění se v této práci počítá s tím, že objekt si reference vytváří sám ve svém konstruktoru, který je navíc bezparametrický. Díky tomu lze při generování jednotkových testů vytvořit objekt, který má všechny potřebné závislosti k vykonávání jeho činnosti.

4.3 Navrhovaný postup

Souhrnný navrhovaný postup pro generování testovacích dat, které zajišťují požadované pokrytí testovaného programu, se tedy bude skládat z několika dílčích kroků. Tyto kroky jsou obecné a měly by být vykonány v následujícím pořadí:

1. Získání grafu toku řízení z testovaného programu.
2. Následná analýza grafu toku řízení, ze které dostaneme informace o použitých proměnných a vstupních parametrech. Tyto informace je nutné uchovávat v nějaké struktuře například v tabulce symbolů 4.2.2.
3. Identifikace vstupních parametrů v tabulce symbolů.
4. Postupně procházet jednotlivé cesty v grafu toku řízení a hledat v nich rozhodovací podmínky, které vytváří větvení programu.
5. Při nalezení rozhodovací podmínky se pokusit vyhodnotit závislost použitých operandů se vstupními parametry. Pokud je zde závislost, můžeme se pokusit pro danou podmínku vygenerovat testovací data, která vedou ke splnění podmínky a testovací data, která vedou k nesplnění podmínky.
6. Pro vygenerovaná data vytvořit odpovídající jednotkové testy.

Při vyhodnocení rozhodovacích podmínek jsme omezeni problematikou vyhodnocení některých symbolů, které nelze vyřešit pouhou statickou analýzou a problémy, které jsou popsány v sekci - 2.5.

5 Implementace

Jedním z úkolů diplomové práce je navrhovaný postup pro generování testovacích dat z kapitoly 4 implementovat a poté důkladně otestovat (viz kapitola - 6). Samotný vývoj knihovny byl rozdělen do tří fází.

V první fázi vývoje jsem se zabýval získáním a **analýzou** grafu toku řízení z testovaného programu. Po analýze grafu byla druhá fáze vývoje zaměřena na samotné **generování testovacích dat**. V poslední fázi vývoje jsem se soustředil na **generování jednotkových testů**.

Vytvořená knihovna je závislá na malém množství knihoven (viz sekce - 5.6), které ale vyžadují minimální verzi Java 8 a vyšší. Knihovna je volně dostupná na stránkách GitLabu[1].

5.1 Analýza grafu toku řízení

Prvním krokem pro vygenerování testovacích dat je analýza grafu toku řízení testovaného programu. Pro tento účel jsem implementoval rozhraní `Analyzer`. Vstupem pro toto rozhraní jsou informace o třídě nebo metodě, pro kterou chceme vytvořit testovací data. Výstupem je datová třída `ControlFlowAnalysisOutput`, která reprezentuje získané informace. Pokud nespecifikujeme konkrétní metodu v testovací třídě, analyzují se všechny metody dané třídy, které mají vstupní parametry. `Analyzer` také dokáže zpracovat všechny soubory *.class* nebo *.java* na zadané cestě a tedy analyzovat celý projekt.

Chceme-li vytvořit testovací případ pro jednu konkrétní metodu, je v některých případech nutné specifikovat informace o této metodě, protože tato metoda může být ve třídě přetížena. Metodu můžeme specifikovat například datovým typem vstupních parametrů či datovým typem návratové hodnoty.

Pro analýzu je podstatný krok získání samotného grafu toku řízení z testované metody. To zajišťuje již zmíněný framework Soot (viz sekce - 3.4.5). Implementovaný `Analyzer` pracuje postupně v následujících krocích:

1. Načtení testovací třídy a všech její závislostí, přičemž implementovaný analyzér načítá nezkompilovaný zdrojový kód i bytecode.
2. Z načtené třídy se pokusí získat metody či metodu, pro kterou se mají vygenerovat testovací data.

---

[1] `https://gitlab.com/jan.albl4/control-flow-testing`

3. Pro tělo dané metody vygeneruje graf toku řízení.
4. Z grafu toku řízení vytvoří acyklický graf (viz sekce - 4.2.1).
5. Postupným prohledáváním grafu toku řízení vytvoří pro rozhodovací uzly příslušné tabulky symbolů (viz sekce - 4.2.2).
6. Po vytvoření tabulek symbolů projde postupně graf toku řízení a narazí-li na rozhodovací podmínku, která způsobuje větvení programu, uloží informace o této podmínce do výstupní datové struktury `ControlFlowAnalysisOutput`. Při analýze rozhodovací podmínky se pokusí vyhodnotit, zda je, či není závislá na vstupním parametru, a zda je pro ni možné vygenerovat testovací data podle navržené metody (viz kapitola - 4). Jsou-li v rozhodovací podmínce neznámé jen symboly, které reprezentují vstupní parametry, tak `Analyzer` rozvine podmínky do základního tvaru.

V kroku č. 3 je z těla metody vytvořen graf toku řízení, který je v Soot reprezentovaný třídou `UnitGraph`. Třída, která tento graf uchovává v projektu, je `ControlFlowGraph`. Protože Soot vytváří `UnitGraph` jen pro načtenou metodu a nijak nerozvíjí ostatní volání dalších metod, musí třída `ControlFlowGraph` uchovávat i `UnitGraph` volaných metod. Důležitým parametrem pro analýzu je také **hloubka prohledávání** volaných metod. Pokud bychom tento parametr nepoužívali a prohledávali všechny volané metody, vznikla by u rekurzivně volaných metod během prohledávání nekonečná smyčka a algoritmus by nikdy neskončil.

Při analýze grafu toku řízení je užitečné, když dokážeme tento graf vizualizovat. K tomuto účelu poskytuje třída `ControlFlowGraph` reprezentaci grafu v DOT formátu. V projektu je použita i knihovna Graphviz[2], díky které je možné uložit graf toku řízení i v obrázkové podobě.

5.1.1 Tabulky symbolů v projektu

Po extrakci grafu toku řízení z testované metody je zapotřebí vytvoření tabulek symbolů, uchovávajících stav symbolů k aktuálnímu uzlu. Ty jsou v projektu reprezentovány třídami `SymbolTable`. Jak již bylo zmíněno v sekci - 4.2.3, každý uzel v grafu má odlišnou tabulku symbolů a může mít i více tabulek podle počtu různých cest vedoucích k danému uzlu.

Vytváření tabulek začíná od vstupního uzlu grafu a končí po dosažení všech uzlů. Během analýzy nás však nejvíce zajímají tabulky symbolů pro rozhodovací uzly, protože z nich je pak možné dohledat závislosti použitých

---

[2] `http://graphviz.org/`

symbolů na vstupních parametrech. Nemá tedy význam uchovávat tabulky symbolů pro ostatní uzly, ze kterých nevede více cest.

Pro komplikovanější grafy toku řízení může být vytváření tabulek symbolů zdlouhavé, protože se pro každou možnou cestu vedoucí od vstupního bloku do bloku výstupního vytváří jedinečná tabulka symbolů. V rámci práce byly tedy implementovány dvě možné varianty vytváření tabulek. První varianta je zastoupena třídou `DetailedAnalyzerSymbolTables`, která vytváří tabulky symbolů běžným způsobem, tedy pro každou možnou cestu je vytvořena unikátní tabulka. Rychlejší, ale méně přesná varianta vytváření tabulek symbolů, je implementována třídou `QuickAnalyzerSymbolTables`. Při této analýze mají dominátoři (viz sekce - 3.1.1) výstupního uzlu grafu toku řízení vždy jednu tabulku symbolů, pokud existuje více rozdílných variant tabulek vedoucích do těchto dominátorů, je vybrána pouze jedna, ze které se bude vycházet při analýze uzlů následujících.

V Javě se pro získání atributu třídy často používají gettery. Ty mohou být obsaženy i v rozhodovacích podmínkách. Abychom nemuseli generovat a následně zdlouhavě analyzovat tuto funkci jako každou jinou, počítá se v implementované knihovně s tím, že pokud se v objektu volá funkce *getNazevAtributu* a objekt obsahuje atribut s tímto názvem, je tento zápis ekvivalentní přímému přístupu do atributu daného objektu. Například volání funkce *person.getAge* je v implementované knihovně ekvivalentní zápisu *person.age*.

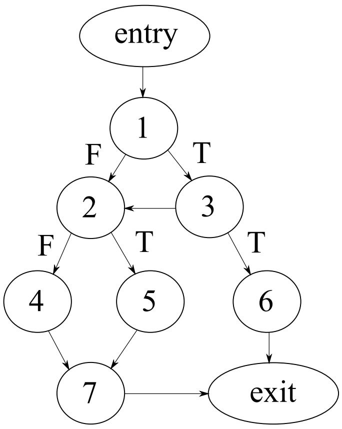
5.1.2 Prohledávání grafu toku řízení

Důležitým úkolem v této práci bylo vymyslet, jakým způsobem budou jednotlivé rozhodovací uzly dosaženy a vyhodnoceny. Protože dosažení uzlu může být v programu podmíněné splněním či nesplněním podmínky v uzlu předcházejícím, je vyhodnocení závislosti mezi jednotlivými uzly velmi důležité.

Na obrázku č. 5.1 můžeme vidět graf toku řízení, který obsahuje tři rozhodovací uzly, přičemž dosažení uzlu číslo tři je závislé na splnění podmínky v uzlu číslo jedna. Abychom dokázali říct, kdy má ještě předcházející rozhodovací uzel vliv na aktuálně procházený, musíme u každé rozhodovací podmínky nalézt jeho **koncový uzel větvení**. Jedná se o bezprostředního post-dominátora (viz sekce - 3.1.1) pro uzel s rozhodovací podmínkou. Pokud je koncový uzel větvení pro rozhodovací podmínku zároveň výstupním uzlem, jsou všechny následující uzly závislé na vyhodnocení rozhodovací podmínky. Tato závislost nemusí být patrná na první pohled ze zdrojového kódu.

Příkladem může být zdrojový kód na obrázku č. 5.1, kterému přísluší vedlejší graf toku řízení. V tomto grafu je možné koncového uzle dosáhnout z uzlu č. 6 nebo z uzlu č. 7. Rozhodovací podmínka v bodě č. 2 je tedy závislá na vyhodnocení podmínky č. 3, pokud bude tato podmínka vyhodnocena jako pravdivá, program ukončí vykonávání této metody, aniž by byla podmínka v bodě č. 2 vyhodnocována. Ve zdrojovém kódu se může na první pohled zdát, že podmínka na řádku č. 7 se vykoná nezávisle na předchozích podmínkách.

```
 1 int calculateMethod(...){
 2       if(...){
 3           if(...) {
 4                return 0;
 5           }
 6       }
 7       if(...){...}
 8
 9       return ...;
10 }
```



Obrázek 5.1: Ukázka grafu toku řízení pro tři rozhodovací uzly

V implementované knihovně se koncové uzly větvení hledají prohledáváním stromu post-dominance. Tento strom je vytvořen použitým frameworkem Soot a informace o něm uchovává ve třídě `MHGPostDominatorsFinder`. Třída poskytuje i metody pracující s tímto stromem a nalezení bezprostředního post-dominátora pro rozhodovací podmínku je velice snadné.

Dokážeme-li nalézt koncové uzly rozhodovacích podmínek větvení, můžeme vytvořit algoritmus, který bude postupně analyzovat graf toku řízení. K vyhodnocení je zapotřebí vytvořit speciální zásobník, který bude uchovávat informace o nezpracovaných rozhodovacích uzlech. V projektu byla za tímto účelem vytvořena třída `BranchStack`. Jednotlivé položky tohoto zásobníku jsou reprezentovány třídou `BranchStackItem`, která uchovává důležité informace o nezpracovaném uzlu větvení, jako je například aktuální tabulka symbolů, koncový uzel, index doposud nezpracované větve a samotnou rozhodovací podmínku. Samotný vyhodnocovací algoritmus pracuje v následujících krocích:

1. Postupně prochází graf toku řízení a pokud narazí na rozhodovací podmínku, vyhodnotí závislost použitých operandů na vstupních parametrech podle příslušné tabulky symbolů. Výsledek vyhodnocení uloží

do výstupní datové struktury `ControlFlowAnalysisOutput` pod aktuálně procházenou větev. Zároveň získá potřebné informace, které uloží do datové struktury `BranchStackItem`, tu pak vloží na vrchol zásobníku.

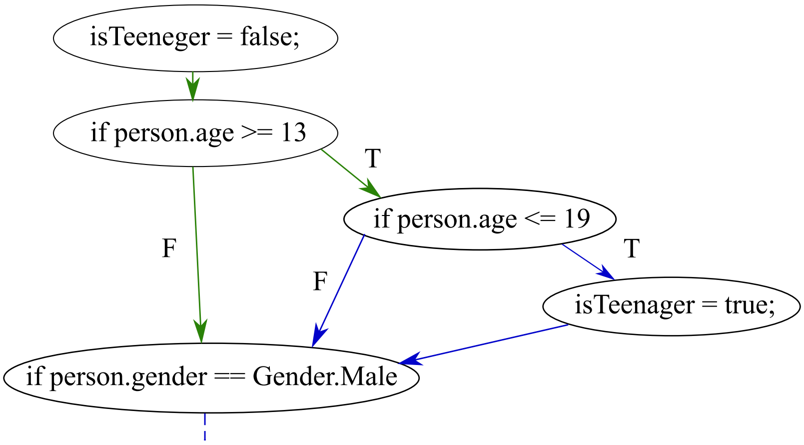
2. Z vrcholu zásobníku vezme podle indexu nezpracovanou větev, kterou pak začne procházet, dokud nenarazí na koncový uzel této podmínky, jinou rozhodovací podmínku či koncový uzel grafu toku řízení.

3. Pokud byla aktuální větev zpracována, sníží index nezpracované větve v položce na vrcholu zásobníku. Zpracoval-li algoritmus všechny dostupné větve rozhodovací podmínky, odebere položku `BranchStackItem` z vrcholu zásobníku a pokračuje ve vyhodnocení další nezpracované větve podle aktuální položky na vrcholu zásobníku.

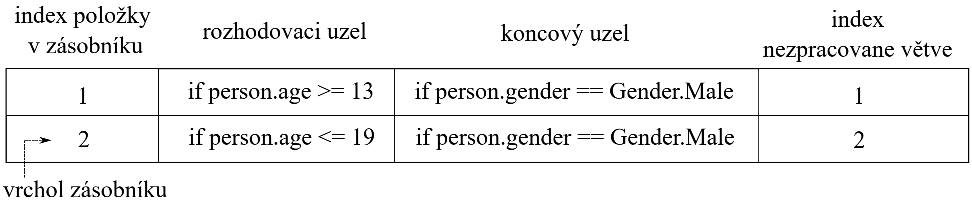
4. Jsou-li zpracovány všechny položky v zásobníku, pokračujeme v prohledávání grafu toku řízení od koncového uzlu poslední odebrané položky ze zásobníku.

5. Po prohledání celého grafu toku řízení se vyhodnotí síla interakce vstupních parametrů (viz sekce - 5.1.3).

Pro příklad procházení a vyhodnocení rozhodovacích podmínek můžeme použít graf toku řízení z obrázku č. 5.2 a příslušný zásobník na obrázku č. 5.3. Zeleně jsou označeny již vyhodnocené hrany a modře hrany, které se musí ještě zpracovat. Zásobník je ve stavu, kdy jsme vyhodnotili již jednu větev první rozhodovací podmínky a právě jsme do něj vložili závislou rozhodovací podmínku. Obě rozhodovací podmínky v zásobníku mají stejný koncový uzel. Po vyprázdnění zásobníku začne algoritmus prohledávat graf toku řízení od tohoto koncového uzlu a hledat další rozhodovací podmínky. Protože je koncový uzel zároveň rozhodovací, bude tedy po zpracování aktuálních podmínek ihned vyhodnocen a vložen do zásobníku.



Obrázek 5.2: Příklad postupného zpracování závislých podmínek



index položky v zásobníku	rozhodovaci uzel	koncový uzel	index nezpracovane větve
1	if person.age >= 13	if person.gender == Gender.Male	1
2	if person.age <= 19	if person.gender == Gender.Male	2

Obrázek 5.3: Ukázka zásobníku rozhodovacích podmínek

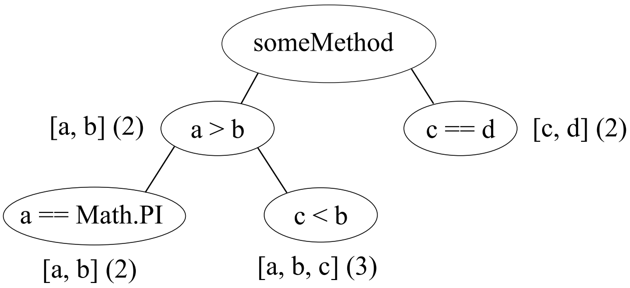
5.1.3 Síla interakce parametrů

Důležitým parametrem, který se po průchodu grafu toku řízení počítá, je síla interakce vstupních parametrů. Ta se využívá například v kombinatorických metodách, které se používají pří automatickém generování testovacích sad. Síla interakce se dá jednoduše spočítat ze závislosti analyzovaných podmínek a parametrů.

Závislost rozhodovacích podmínek můžeme znázornit stromovou strukturou, kde kořen stromu je testovaná metoda. Potomci kořenového uzlu jsou jednotlivé rozhodovací podmínky, které mohou mít jako potomky další rozhodovací uzly, které jsou na rodičích závislé. Sílu interakce můžeme spočítat postupným průchodem tohoto stromu od kořenu k listům. Při průchodu si v každém uzlu udržujeme množinu použitých rodičovských symbolů, do které přidáme i symboly v aktuálním uzlu. Po průchodu vyhledáme v listech stromu množinu s největším počtem symbolů. Počet symbolů v této množině je pak roven síle interakce vstupních parametrů testované metody. V implementované knihovně se o počítání síly interakce starají třídy implementující rozhraní `AnalyzerInteraction`.

```
void someMethod(int a, int b, int c, int d){
        if(a>b){
            if(a == Math.PI) { ... }
            if(c < b){ ...}
        }
        if(c == d){..}
 }
```

Zdrojový kód 5.1: Ukázka metody se sílou interakce vstupních parametrů rovnou třem



Obrázek 5.4: Ukázka počítání síly interakce vstupních parametrů

5.1.4 Výstup analýzy

Důležitým úkolem analýzy grafu toku řízení je vyhodnocení závislostí rozhodovacích podmínek se vstupními parametry a závislosti jednotlivých podmínek mezi sebou navzájem. Výsledek je uložen do datové třídy `ControlFlowAnalysisOutput`, která vlastní další datové třídy obsahující informace o testované třídě, metodě, použitých parametrech a rozhodovacích podmínkách.

Informace o rozhodovacích podmínkách jsou ve výstupu uloženy ve třídě `ConditionElement`. Ta může obsahovat několik různých variant výrazů podmínky, protože v grafu toku řízení může existovat více různých cest vedoucích od výstupního uzlu do uzlu s touto podmínkou (neboli pro rozhodovací podmínku je více aktuálních variant tabulek symbolů, viz sekce - 4.2.3). Jeden výraz podmínky je ve výstupu reprezentován třídou `ConditionalExpression`. V této třídě se nachází i příznak, zda je podmínka závislá na vstupním parametru a lze ji tedy vyhodnotit.

`ConditionElement` může obsahovat i další vnořené rozhodovací podmínky, které jsou na této podmínce závislé. Při následném generování testovacích dat je užitečné znát nejen, zda je podmínka závislá, či nezávislá na vstupních parametrech, ale i datové typy použitých symbolů. To se hodí zejména při vyhodnocení atributů vstupního objektu. Informace o datových typech symbolů použitých ve výrazu rozhodovací podmínky jsou také uloženy v jednotlivých třídách `ConditionalExpression`.

Pro následující příklad můžeme vidět výstup po analýze grafu toku řízení převedený do podoby značkovacího jazyka XML v příloze B. Implementovaná knihovna disponuje konvertorem reprezentovaným třídou `OutputConverter`, který dokáže převést výstupní datovou třídu `ControlFlowAnalysisOutput` do podoby XML nebo JSON.

```
public boolean canEnter(Person p, int limitedAge, double limitedHeight){
        boolean canEnter = false;
        if(p.age > limitedAge){
            if(p.height < limitedHeight){
                canEnter = true;
            }
        }
        return canEnter;
}
```

Zdrojový kód 5.2: Ukázka testované metody

5.2 Generování testovacích dat

Po analýze grafu toku řízení testované metody je zapotřebí vygenerovat testovací data pro jednotkové testy, jejichž vykonávání dostatečně pokryje možné cesty testovaného programu. Tento úkol v implementované knihovně zastává třída `DataGenerator`. Vstupem této třídy je výstupní datová třída z analýzy grafu toku řízení `ControlFlowAnalysisOutput`. `DataGenerator` generuje testovací data pouze pro podmínky, které se skládají výhradně ze vstupních parametrů nebo konstant. V sekci - 4.2.3 byly rozhodovací podmínky větvení rozděleny do dvou kategorií podle složitosti.

Testovací data pro **první kategorii podmínek** získáme sestavením a vyřešením příslušných matematických rovnic či nerovnic. Sestavení nerovnic z výstupu po analýze grafu toku řízení je poměrně snadné. Má-li aktuálně vyhodnocovaná podmínka nějakou nadřazenou podmínku, je aktuálně vyhodnocovaná podmínka na nadřazené závislá. Pokud se vyhodnocovaná podmínka nachází v `trueBranch` nadřazené větve, musí být nadřazená podmínka splněna. Je-li ve `falseBranch` nadřazené větve, musí být nadřazená podmínka vyhodnocena jako nepravdivá.

Příkladem základního tvaru sestavené nerovnice pro vyhodnocení druhé rozhodovací podmínky z ukázky výstupu v příloze B je nerovnice

$$!(p.age \leq limitedAge) \cap (p.height\ cmpl\ limitedHeight) \leq 0. \qquad (5.1)$$

Základní tvar této nerovnice můžeme převést do alternativního zápisu zobrazeného v nerovnici č. 5.2. Jak již bylo zmíněno v sekci - 4.2.4, abychom vygenerovali data pro splnění a nesplnění podmínky, musíme vytvořit i nerovnici, jejíž řešení vede k nesplnění aktuální rozhodovací podmínky, viz nerovnice č. 5.3.

$$p.age > limitedAge\ \cap\ p.height - limitedHeight \leq 0 \qquad (5.2)$$

$$p.age > limitedAge\ \cap\ p.height - limitedHeight > 0 \qquad (5.3)$$

Pro vytvoření testovacích dat je zapotřebí číselné vyhodnocení těchto nerovnic (viz sekce - 5.2.1). Výsledné hodnoty jsou uloženy do jednotlivých parametrů testovacích metod ve vstupní třídě `ControlFlowAnalysisOutput` (viz sekce - 5.2.2).

Generování testovacích dat vedoucí ke splnění či nesplnění rozhodovacích podmínek patřící do **druhé kategorie** je složitější, protože ve většině případů se k vyhodnocení podmínky musí přistupovat individuálně podle použitých operátorů a operandů (viz sekce - 4.2.4). V této diplomové práci jsem s ohledem na časovou náročnost zadání implementoval pouze vyhodnocení rozhodovacích podmínek obsahující operátor *instanceof* a podmínek testujících, zda je vstupní objekt inicializován. Instance možných objektů včetně výčtových typů jsou v knihovně vyhledávány pomocí frameworku Soot.

Pokud analyzovaný graf toku řízení neobsahuje větvení nebo nejsou vstupní parametry obsaženy v rozhodovacích podmínkách, jsou pro tyto parametry vygenerována náhodná data.

5.2.1 Symbolické vyhodnocení nerovnic

Důležitým krokem k řešení sestavených nerovnic pro rozhodovací podmínky bylo nalezení nástroje, který dokáže provádět symbolické výpočty. Těchto nástrojů[3] existuje mnoho, ale jen některé dokáží provádět výpočty s nerovnicemi. Jeden z hlavních faktorů při hledání tohoto softwaru byla i jeho dostupnost. Zaměřil jsem se tedy spíše na nástroje s open-source licencí.

---

[3] `https://en.wikipedia.org/wiki/List_of_computer_algebra_systems`

Během hledání jsem našel jen málo nástrojů, které poskytují symbolické výpočty pracující s nerovnicemi a mají open-source licenci. Mezi nejslibnější open-source softwary splňující mé požadavky patří **SageMath**[4] a **Symja**[5].

SageMath je velmi rozšířený matematický software s licencí GPL. Stal se základem mnoha existujících softwarů, jako jsou Sympy, Maxima, NumPy atd. Má komponenty psané v různých programovacích jazycích, jako C/C++, Python, Lisp atd. Symja je matematická knihovna psaná v programovacím jazyce Java. Při řešení nerovnic pomocí těchto nástrojů bychom v ideálním případě chtěli získat intervaly řešení, ze kterých můžeme vygenerovat testovací sady dat. Oba nástroje však poskytují jen omezenou práci s nerovnicemi a místo intervalů řešení poskytují pouze zjednodušenou variantu vstupní nerovnice (či soustavy nerovnic). Některé komerční nástroje, jako například WolframApha[6], dokáží vypočítat ze soustavy nerovnic i intervaly řešení. Závislost na komerčních nástrojích však omezuje použití vytvořené knihovny, a proto jsem hledal jiný způsob řešení.

Zaměřil jsem se na hledání hraničních hodnot intervalů řešení. Ty jsou zajímavé především tím, že jejich blízké hodnoty vedou ve většině případů na změnu pravdivostní hodnoty vyhodnocované rozhodovací podmínky. Tento princip se běžně využívá při testování softwaru (boundary value testing). Hraniční uzly intervalů mohou být vypočteny tím, že z nerovnice vytvoříme rovnici změnou porovnávacího operátoru na rovnost. Rovnice však můžeme tvořit jen z jednotlivých nerovnic, nikoli z celé soustavy (viz níže). Je zřejmé, že rovnice mohou mít více řešení tedy i více hraničních hodnot.

Jako příklad použijeme nerovnici 5.4, kterou převedeme na rovnici 5.5. Z této rovnice dostaneme ihned hodnotu hraničního bodu řešení pro symbol a, tedy č. 4. Z celého intervalu hodnot, které může použitý symbol nabývat, vytvoříme jednotlivé podintervaly rozdělené podle hraničních hodnot. V tomto případě nám vzniknou dva podintervaly $(-\infty, 4)$ a $[4, \infty)$. Z každého intervalu pak můžeme vybrat nějakým způsobem testovací data. Ta by měla obsahovat i krajní hodnoty těchto intervalů, neboť v rozhodovací podmínce může být obsažena i rovnice testující konkrétní hodnotu symbolu.

$$a < 4 \tag{5.4}$$

$$a = 4 \tag{5.5}$$

Nerovnice obsažené v rozhodovacích podmínkách mohou obsahovat i více

---

[4] `http://www.sagemath.org/`

[5] `https://bitbucket.org/axelclk/symja_android_library/wiki/Home`

[6] `https://www.wolframalpha.com/`

parametrů. V takovém případě by z nich sestavená rovnice měla nekonečně mnoho řešení. To je při generování testovacích dat nepříjemné, protože vyžadujeme konkrétní testovací data pro konkrétní parametry, abychom mohli vytvořit příslušné testovací sady. Jedním ze způsobů, jak tento problém částečně vyřešit, je vygenerováním náhodných hodnot pro $n-1$ obsažených symbolů a dopočítáním hodnoty pro n-tý symbol. V mnoha případech se tento postup osvědčí, mohou však nastat případy, kdy po dosazení hodnot pro $n - 1$ symbolů už nebude mít rovnice řešení či výsledná hodnota n-tého symbolu bude větší nebo menší než její maximální rozsah hodnot.

SageMath i Symja dokáží z rovnic dopočítat hodnoty řešení. Při rozhodování, který z těchto dvou nástrojů použít, jsem zohlednil snadnost zabudování nástroje do stávájicí knihovny. S ohledem na to jsem se rozhodl pro nástroj Symja, který je dostupný i jako maven projekt a je plně kompatibilní s programovacím jazykem Java.

V implementované knihovně se rovnice s více parametry počítají dosazením náhodných hodnot pro $n - 1$ symbolů, a dále se dopočítává hodnota n-tého symbolu, jak již bylo uvedeno výše. Pokud hodnoty nejdou dopočítat, je zde ještě možnost přidat hodnoty ručně v uživatelském rozhraní knihovny. Po dopočítání hraničních hodnot se vytvoří intervaly. Do vygenerovaných testovacích dat se vloží hraniční hodnoty všech vytvořených intervalů a jedna náhodná hodnota z každého intervalu, která leží mezi hraničními body tohoto intervalu.

Nevýhoda generování dat při použití převodu nerovnic na rovnice je ztráta informací o závislosti rozhodovacích podmínek. Rovnice je vždy vytvořena jen z jedné nerovnice, nikoli ze soustavy nerovnic vytvořené podle závislostí rozhodovacích podmínek. Kdybychom vytvořili ze soustavy nerovnic příslušnou soustavu rovnic, mohl by snadno nastat případ, kdy řešení soustavy rovnic neexistuje.

Z následujícího příkladu bychom po analýze grafu toku řízení získali pro podmínku na řádce č. 2 soustavu nerovnic 5.6. Kdybychom chtěli tuto soustavu nerovnic převést celou na soustavu rovnic, tato soustava by neměla řešení (viz soustava nerovnic 5.7). Z tohoto důvodu se rovnice vyhodnocují samostatně. Po vyhodnocení těchto dvou rovnic samostatně získáme hraniční hodnoty 0 a 10, které použijeme na vytvoření podintervalů, ze kterých se generují konkrétní testovací data.

```
if(a > 0){
    if(a < 10){
        ...
    }
}
```

Zdrojový kód 5.3: Ukázka testované metody

$$a > 0 \cap a < 10 \tag{5.6}$$

$$a = 0 \cap a = 10 \tag{5.7}$$

5.2.2 Výsledná generovaná data

Vygenerovaná data jsou uložena do datové třídy `ControlFlowAnalysisOutput`, která je zároveň výstupem analýzy grafu toku řízení. Díky tomu máme v jedné datové struktuře uloženy kompletní informace o provedené analýze i vygenerovaných datech. Ve výstupním objektu jsou uloženy informace o konkrétních parametrech nezbytných k vytvoření jednotkových testů. Pokud je vstupní parametr testované metody objekt, může obsahovat atributy, které jsou použity v rozhodovacích podmínkách. V `ControlFlowAnalysisOutput` jsou informace o těchto atributech uloženy spolu s jejich vygenerovanými hodnotami v daném vstupním parametru. Příklad výstupu pro metodu ze sekce - 5.1.4 je uveden v příloze C.

5.3 Generování jednotkových testů

Jsou-li vygenerována testovací data, můžeme pro zkoumanou metodu vytvořit jednotkové testy. Při generování testovacích dat není zohledněna závislost jednotlivých rozhodovacích podmínek (viz sekce - 5.2.1). Abychom tedy získali požadované pokrytí testovacími daty, musíme vytvořit jednotkové testy s různými kombinacemi hodnot vstupních parametrů. S přibývajícím počtem vygenerovaných dat velmi rychle roste i počet všech jejich možných kombinací a tedy i počet jednotkových testů. Existují techniky, které na základě síly interakce vstupních parametrů (viz sekce - 5.1.3) dokáží efektivně snížit velikost testovací sady.

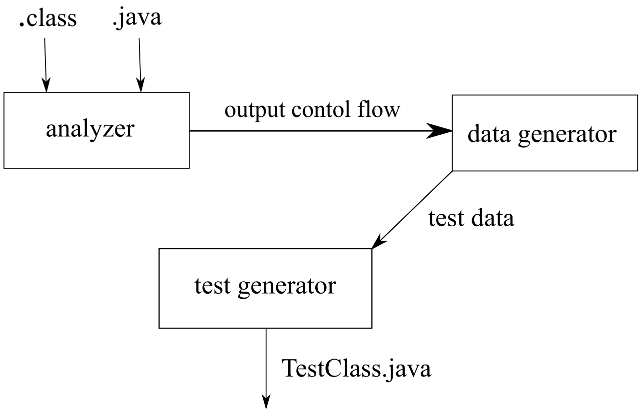
V implementované knihovně slouží pro generování jednotkových testů rozhraní `TestGenerator`. Vstupem do tohoto rozhraní je datová třída `ControlFlowAnalysisOutput`. Výstupem je pak soubor či soubory obsahující vygenerované jednotkové testy. Jak již bylo zmíněno výše, jednotkových testů pro otestování jedné metody může být mnoho, a proto je lze rozdělit do několika testovacích tříd. Vytvořené jednotky testů mohou být určeny pro více metod testované třídy, ale abychom mohli tyto metody volat je důležité, aby testovací metoda měla modifikátor přístupu nastaven na hodnotu *public*. Pokud tomu tak není, nejsou pro metodu vytvořeny jednotkové testy, neboť ji nelze volat z testovací třídy.

Generátor testů neřeší vnitřní závislosti testovaných tříd (viz sekce - 4.2.5) na ostatních objektech, jejichž metody mohou být volány v těle testované metody. Při vytváření testovacího objektu (popřípadě objektů vstupujících do testované metody) je vždy pro jeho vytvoření volán jeho bezparametrický konstruktor. Implementovaný analyzátor zkoumá, zda konstruktor testované třídy splňuje tyto podmínky. Pokud nejsou splněny, generátor testů nevygeneruje pro tuto testovanou třídu jednotkové testy.

5.4 Struktura projektu

Vytvořená knihovna je realizovaná jako Maven multi-module projekt, který je tvořen třemi moduly. Hlavní modul ***cft-generator*** obsahuje analyzér grafu toku řízení, generátor testovacích dat i generátor testů. Druhý modul ***cft-gui*** představuje implementované uživatelské rozhraní (viz sekce - 5.5), které usnadňuje komunikaci s prvním modulem a dovoluje uživateli měnit, odebírat a přidávat vygenerovaná testovací data. Poslední pomocný modul ***cft-examples*** obsahuje pouze ukázky použití knihovny.

Na obrázku č. 5.5 můžeme vidět diagram procesu vytváření jednotkových testů z analyzované třídy.



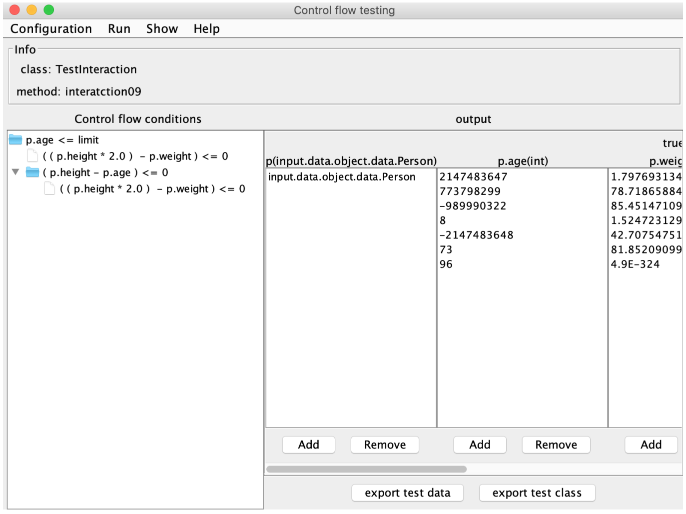
Obrázek 5.5: Proces generování souboru obsahující jednotkové testy

5.5 Uživatelské rozhraní

Během generování testovacích dat nemusí generátor `DataGenerator` nalézt všechny potřebné hodnoty vedoucí k požadovanému pokrytí testovaného programu. Aby uživatel mohl vygenerovaná data upravovat, měnit či přidávat, bylo implementováno uživatelské rozhraní, které usnadňuje práci s generováním testovacích dat a jednotkových testů.

Pro přehlednost může uživatel v tomto rozhraní analyzovat a generovat jednotkové testy pouze pro jednu metodu testovací třídy. Po spuštění analýzy je možné vidět v levé části obrazovky na obrázku č. 5.6 strukturu závislostí rozhodovacích podmínek. V pravé části jsou pak zobrazeny jednotlivé parametry, které jsou použity v rozhodovacích podmínkách nebo slouží jako vstup do metody a je nutné pro ně vygenerovat nějakou výchozí hodnotu, aby bylo možné tuto metodu volat. Těmto parametrům můžeme přidávat či mazat jednotlivé hodnoty, které jsou pak použity při generování jednotkových testů. Rozhraní je primárně stavěno na úpravu hodnot pro primitivní datové typy, ale můžeme v něm specifikovat i konkrétní instance důležitých objektů, které mají vliv na průchod programem.

Rozhraní také dovoluje zobrazit (v podobě DOT) či uložit (ve formátu png) aktuální graf toku řízení testovaného programu. Protože graf toku řízení je generován ze Soot reprezentace, odpovídají jednotlivé uzly řádkám programu v této podobě.



class: TestInteraction

method: interatction09

- p.age <= limit
  - ((p.height * 2.0) - p.weight) <= 0
- (p.height - p.age) <= 0
  - ((p.height * 2.0) - p.weight) <= 0

		true
p(input.data.object.data.Person)	p.age(int)	p.weig
input.data.object.data.Person	2147483647	1.797693134
	773798299	78.71865884
	-989990322	85.45147109
	8	1.524723129
	-2147483648	42.70754751
	73	81.85209099
	96	4.9E-324

Obrázek 5.6: Ukázka implementovaného uživatelského rozhraní

5.6 Použité technologie

Implementovaná knihovna je závislá na několika volně dostupných knihovnách, které usnadnily a urychlily vývoj tohoto projektu.

- **Graphviz**[7] - Grafický vizualizační software s licencí CPL verze 1.0. Dokáže převést graf popsaný v textové podobě (například DOT formátu) do diagramu v několika užitečných formátech, jako jsou obrázky, SVG či Postscript. V implementovaném projektu je tato knihovna zabudována jako Maven závislost, která vyžaduje minimální verzi Java 8. Graphviz je zde používán výhradně pro vizualizaci grafu toku řízení.

- **Symja**[8] - Algebraická knihovna pro programy psané v programovacím jazyce Java. V projektu se používá pro výpočet kořenu nerovnic, jak již bylo zmíněno v sekci - 5.2.1. Licence tého knihovny je GNU General Public Licence verze č. 3.

- **Apache Commons Lang**[9] - Poskytuje třídy, které ulehčují manipulaci se standardními jádrovými třídami Java (zejména práci s řetězci, čísly, reflexí atd.). Implementovaná knihovna používá Apache Commons Lang zejména pro práci s intervaly hraničních hodnot. Aktuálně dostupná verze je vydaná s licencí Apache verze 2.0.

- **Google-java-format**[10] - Jedná se o program, který dokáže přeformátovat zdrojový kód Java tak, aby odpovídal standardu Google Java Style. V projektu je použit při generování testovacích tříd. Licence tohoto softwaru je Apache ve verzi 2.0.

- **Soot** - Tento framework byl již popsán v sekci - 3.4.5. V implementované knihovně se používá starší verze této knihovny 2.5.0, která dokáže analyzovat pouze programový kód napsaný v Java SE 7 a starší. Nejnovější verze 3.3.0 dokáže pracovat s kódem napsaným v novějších verzích Java, ale v této verzi se již nedoporučuje analyzovat nezkompilované zdrojové kódy. Během vytváření knihovny jsem se rozhodl použít starší verzi, která zvládne načíst (s omezením) i nezkompilovaný zdrojový kód, který je pro programátory lépe čitelný.

---

[7] `https://www.graphviz.org/`

[8] `https://bitbucket.org/axelclk/symja_android_library/wiki/Home`

[9] `https://commons.apache.org/`

[10] `https://github.com/google/google-java-format`

5.7 Možnosti rozšíření

Při návrhu a pozdější implementaci generování hodnot pro jednotkové testování z grafu toku řízení jsem narazil na mnoho problémů, které je potřeba pro optimální generování testů vyřešit. Programovací jazyk Java disponuje mnoho konstrukcemi, které se musejí analyzovat i vyhodnocovat individuálně. Z tohoto důvodu roste složitost i časová náročnost implementace řešení, které by dokázalo optimálně generovat jednotkové testy pro analyzované metody. V následujících sekcích jsou uvedeny situace, které nejsou v práci řešeny, ale jsou jistě podstatné pro lepší pokrytí testovaného programu jednotkovými testy.

5.7.1 Pole a kolekce

Pro udržení většího počtu hodnot nebo objektů se často používají pole nebo kolekce. Jejich vyhodnocení v rozhodovacích podmínkách je složité především tím, že index ukazující na objekt v poli nemusí být konstantní. Pro generování testovacích hodnot je tedy nejdříve zapotřebí dopočítat možné pozice objektu v poli. Pokud je proměnná představující index v poli závislá pouze na vstupních parametrech, nemusí být zjištění konkrétních hodnot problematické.

5.7.2 Symbolické vyhodnocení

V sekci - 5.2.1 jsou popsané nástroje pro symbolické vyhodnocení primitivních parametrů. Nástroj Symja, který se v implementované knihovně používá, nedokáže pracovat se soustavami nerovnic. Nalezením a zabudováním nástroje, který je open-source a dokáže vyhodnocovat i soustavy nerovnic, by se zásadně zmenšil počet vygenerovaných hodnot i testovacích případů.

5.7.3 Vyhodnocení cyklů

Je-li v testovaném programu smyčka, která je závislá na vstupním parametru, navrhovaný postup generuje data tak, aby při vykonávání programu se kód ve smyčce vykonal alespoň jednou. Pro kvalitnější otestování programu by bylo ideální, kdyby se smyčka vykonala vícekrát, popřípadě by uživatel měl možnost před analýzou grafu toku řízení určit počet iterací pro smyčky.

5.7.4 Složitější operátory a matematické funkce

Dalším důležitým rozšířením této práce je práce se složitějšími operátory (například bytové operátory, modulo atd.) a běžné matematické funkce standardní třídy `Math`. Metody této třídy lze snadno přepsat tak, aby je dokázal použitý matematický nástroj vyhodnotit.

5.7.5 Závislosti objektů a generika

V navrhovaném postupu ani v implementaci není vyhodnocována závislost mezi objekty či práce s generickými datovými typy. Tato závislost může být důležitá při spouštění testované metody viz sekce - 4.2.5. Z grafu toku řízení lze poznat, kdy se volá metoda závislého objektu, ale už v něm nemusí být uvedena jeho inicializace. Dalším rozšířením této práce může být tedy mapování závislostí mezi objekty a dosazení těchto závislostí při generování jednotkových testů.

6 Testování implementace

Posledním úkolem této práce bylo otestovat funkčnost vytvořené knihovny a navržené metody pro získání parametrů jednotkových testů. Funkčnost knihovny byla otestována jednotkovými testy. V nich jsem se zaměřil především na části, které se zabývaly analýzou grafu toku řízení a generováním testovacích dat pro všechny primitivní datové typy.

Zajímavým testem bylo vyzkoušet projekt na reálných příkladech, pro které by měla být knihovna používána. Protože implementovaná knihovna zatím pracuje převážně s primitivními datovými typy, je zřejmé, že pro mnoho metod nedokáže vygenerovat odpovídající data pro optimální pokrytí kódu. Měla by však dobře pracovat s metodami, do kterých vstupují primitivní datové typy či objekty s primitivními datovými atributy, které rozhodují, jakou cestou se program vydá v rozhodovacích podmínkách.

Vytvořenou knihovnu jsem vyzkoušel na několika projektech. Výsledky testování byly odpovídající mému očekávání. Knihovna dokázala vygenerovat testovací data pro metody pracující s primitivními datovými typy s dostatečným pokrytím, pokud lze pokrytí kódu v těchto metodách ovlivnit hodnotou vstupních parametrů.

Příkladem může být volně dostupný program pro počítání výše hypoték mortgage-calculator[1]. Statistiky o tomto programu jsou zobrazeny v tabulce č. 6.1. Pro 26 veřejných metod byly vygenerovány odpovídající jednotkové testy (některé z nich jsem musel ručně upravit, protože generátor má jistá omezení viz sekce - 5.3). V příloze D je zobrazena vygenerovaná třída obsahující jednotkové testy pro třídu *Calculations*. Pokrytí testovaných metod se při pokusech pohybovalo kolem 80 % až 90 % řádek kódu.

Počet tříd	**18**
Celkový počet metod	**93**
Počet veřejných metod	**65**
Počet metod s parameterm	**41**
Počet metod, pro které knihovna vygenerovala testovací data	**26**
Pokrytí kódu v testovaných metodách s primitivními parametry	**70% - 90%**

Tabulka 6.1: Mortgage-calculator

---

[1] `https://github.com/mposolda/mortgage-calculator/`

Závěr

V této práci byla navržena metoda pro získávání hodnot parametrů jednotkových testů z grafu toku řízení testovaného programu. Na jejím principu byla vytvořena knihovna pro generování testovacích dat z Java kódu. Při analýze grafu toku řízení jsem se setkal s mnoha problémy. Některé z těchto problémů nelze vyřešit pomocí použité statické analýzy kódu, která je součástí navržené metody nebo je jejich řešení příliš komplikované.

Během analýzy grafu je také důležité vyhodnocení závislostí jak mezi vstupními parametry, tak mezi rozhodovacími podmínkami, které určují větvení grafu toku řízení. Programovací jazyk Java má také mnoho různých programových konstrukcí, které může testovaný program obsahovat. Z těchto důvodů bylo velmi časově i konstruktivně náročné vytvořit nástroj, který by dokázal vygenerovat testovací data pro většinu často používaných konstrukcí, jako jsou například numerické proměnné, objekty, pole, kolekce a řetězce.

Implementovaná knihovna generuje testovací data pro primitivní datové typy a instance objektů. Funkcionalitu lze jistě postupně rozšířit o práci s řetězci, poli a kolekcemi. Důležitým krokem pro další vývoj knihovny by také bylo nalezení open-source nástroje pro symbolické vyhodnocení soustav nerovnic. Vyřešením těchto soustav bychom získali jak menší počet vygenerovaných testovacích dat, tak i menší počet vygenerovaných jednotkových testů, které by měly stejné pokrytí testovaného kódu jako implementovaný způsob.

Literatura

[1] *Control Statements*, 9. Dostupné z: `http://download.nos.org/srsec330/330L9.pdf`.

[2] 5 Test Data Generation Techniques You Need to Know, 2017. Dostupné z: `https://www.testbytes.net/blog/5-test-data-generation-techniques-to-know/`.

[3] Best practices in Test Data Generation. Dostupné z: `https://www.techarcis.com/best-practices-in-test-data-generation/`.

[4] Types of Test Data Used During Software Testing, 2018. Dostupné z: `http://qatestlab.com/resources/knowledge-center/5-types-of-test-data-that-should-be-used-during-software-testing/`.

[5] Test Data Generation: What is, How to, Example, Tools. Dostupné z: `https://www.guru99.com/software-testing-test-data.html`.

[6] Unit Testing. Dostupné z: `http://softwaretestingfundamentals.com/unit-testing/`.

[7] Alfred V. Aho, R. S. J. D. U. M. S. L. *Compilers*, s. 531–532, 655–671. Greg Tobin, second edition, 2007. ISBN 0-321-48681-1.

[8] Beizer – Mancoridis. Dependable Software Systems (Control-Flow Testing). Dostupné z: `https://www.cs.drexel.edu/~spiros/teaching/CS576/slides/2.control-testing.pdf`.

[9] Bruneton, E. ASM 4.0 A Java bytecode engineering library. 2011. Dostupné z: `https://asm.ow2.io/asm4-guide.pdf`.

[10] Cifuentes, C. *Reverse Compilation Techniques.* PhD thesis, The Queensland University of Technology, 1994. Dostupné z: `http://www.phatcode.net/res/228/files/decompilation_thesis.pdf`.

[11] Dssouli, R. et al. *Testing the Control-Flow, Data-Flow, and Time Aspects of Communication Systems: A Survey.* 01 2017. doi: 10.1016/bs.adcom.2017.06.002. Dostupné z: `https://www.researchgate.net/publication/319195395_Testing_the_Control-Flow_Data-Flow_and_Time_Aspects_of_Communication_Systems_A_Survey`.

[12] DuPaul, N. Static Testing vs. Dynamic Testing, 2013. Dostupné z: `https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing`.

[13] Dévai, R. et al. Designing and Implementing Control Flow Graph for Magic 4th Generation Language. *Acta Cybernetica*. 01 2014, 21. doi: 10.14232/actacyb.21.3.2014.9.

[14] Edvardsson, J. A Survey on Automatic Test Data Generation. 2002. Dostupné z: `https://www.cc.gatech.edu/~harrold/6340/cs6340_fall2009/Readings/test.data.genration.survey.pdf`.

[15] Einarsson, A. – Nielsen, J. A Survivor's Guide to Java Program Analysis with Soot. 2008. Dostupné z: `http://cs.au.dk/~mis/soot.pdf`.

[16] Ghahrai, A. Static Analysis vs Dynamic Analysis in Software Testing, 2018. Dostupné z: `https://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/`.

[17] Harrold, M. J. – Rothermel, G. – Orso, A. Representation and Analysis of Software. Dostupné z: `https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/rep-analysis-soft.pdf`.

[18] Havlak, P. Nesting of Reducible and Irreducible Loops. *ACM Trans. Program. Lang. Syst.* 07 1997, 19, s. 557–567. doi: 10.1145/262004.262005.

[19] Hayes, L. G. *The Automated Testing Handbook*. Software Testing Institute, second edition, 1996; 2004. Dostupné z: `https://assets.ctfassets.net/373no6dgqol9/4x2APEaDtKUckiqg4aSeE0/6ed0e59f892fad8d6826d272eb77ac5f/Automated_Testing_Handbook.pdf`. ISBN 0-9707465-0-4.

[20] Hitesh, T. – Bichitra, K. Automated Software Test Data Generation: Direction of Research. *International Journal of Computer Science and Engineering Survey*. 02 2011, 2. doi: 10.5121/ijcses.2011.2108. Dostupné z: `http://airccse.org/journal/ijcses/papers/0211cses08.pdf`.

[21] Korel, B. Automated Test Data Generation for Programs with Procedures. ACM, 1996. Dostupné z: `ftp://vm1-dca.fee.unicamp.br/pub/docs/jino/acm/p209-korel.pdf`.

[22] Kot, M. The State Explosion Problem, 2003. Dostupné z: `http://www.cs.vsb.cz/kot/down/Texts/StateSpace.pdf`.

[23] Kua, P. *UNIT TESTING*. Oracle Australian Development Centre. Dostupné z: `https://www.thekua.com/publications/AppsUnitTesting.pdf`.

[24] Lam, P. et al. The Soot framework for Java program analysis: a retrospective. 2011. Dostupné z: `https://sable.github.io/soot/resources/lblh11soot.pdf`.

[25] Laplante, P. et al. Software Test Automation. *Adv. Software Engineering.* 01 2010, 2010. doi: 10.1155/2010/163746. Dostupné z: `https://www.researchgate.net/profile/Fevzi_Belli2/publication/220452170_Software_Test_Automation/links/09e4150eee7627d472000000/Software-Test-Automation.pdf?origin=publication_detail`.

[26] Le, W. Control Flow Analysis. 2016. Dostupné z: `http://web.cs.iastate.edu/~weile/cs513x/4.ControlFlowAnalysis.pdf`.

[27] Le, W. Program Representation. 2014. Dostupné z: `http://web.cs.iastate.edu/~weile/cs641/2.ProgramRepresentations.pdf`.

[28] Leitner, A. et al. Reconciling Manual and Automated Testing: the AutoTest Experience. IEEE, 2007. Dostupné z: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.7407&rep=rep1&type=pdf`.

[29] Naik – Thripathy. Software Testing and Quality Assurance. 4: Control Flow Testing. The University of Waterloo. Dostupné z: `https://slideplayer.com/slide/5770731/`.

[30] Pargas, R. P. – Harrold, M. J. – Peck, R. R. Test - Data Generation Using Genetic Algorithms. *Journal of Software testing, Verification and Reliability.* 1999. Dostupné z: `https://www.cc.gatech.edu/~harrold/6340/cs6340_fall2009/Readings/pga.pdf`.

[31] Parkin, R. Software Unit Testing. *IVV Australia.* 1997. Dostupné z: `http://condor.depaul.edu/sjost/hci430/documents/testing/UnitTesting.pdf`.

[32] Roudenský, P. – Havlíčková, A. *Řízení kvality softwaru: Průvodce testováním.* Computer Press, Brno, first edition, 2013. ISBN 0-9707465-0-4.

[33] Ullman, J. D. Flow Graph Theory. Dostupné z: `http://infolab.stanford.edu/~ullman/dragon/w06/lectures/dfa3.pdf`.

[34] Yao, X. – Gong, D. Genetic Algorithm-Based Test Data Generation for Multiple Paths via Individual Sharing. *Computational intelligence and neuroscience.* 10 2014, s. 591294. doi: 10.1155/2014/591294.

Seznam zkratek

API Application Programming Interface. 28

CFG Control-flow graph. 17

CPL Common Public License. 62

GA Genetický algoritmus. 15

JSON JavaScript Object Notation. 55

SSA Static Single Assignment form. 28

SVG Scalable Vector Graphics. 62

XML eXtensible Markup Language. 55

Seznam obrázků

Seznam tabulek

A Ukázka vstupního souboru pro TRGeneration

```java
if (x < y)
{
    y = 0;
    x = x + 1;
}
else
{
  x = y;
}
int month = 8;
String monthString;
int month = 8;
String monthString;
switch (month) {
    case 1:
      monthString = "January";
      x++;
      y++;
             break;
    case 2:  monthString = "February";
             break;
    case 3:  monthString = "March";
             break;
    case 4:  monthString = "April";
             break;
    default: monthString = "Invalid month";
             break;
}
System.out.println(monthString);
System.out.println(monthString);
```

Zdrojový kód A.1: Ukázka souboru obsahující tělo metody napsané v programovacím jazyce Java. Tento soubor slouží jako vstup do programu TRGeneration.

B Ukázka výstupu po analýze grafu toku řízení

```xml
<?xml version="1.0" encoding="UTF-8"?>
<controlFlowAnalysisOutput>
    <classes>
        <class>
            <classPath>../path/</classPath>
            <packageName>input.data.object.medium</packageName>
            <className>TestObjectMedium</className>
            <hasPublicConstructor>true</hasPublicConstructor>
            <methods>
                <method>
                    <name>canEnter</name>
                    <parameterCount>3</parameterCount>
                    <isStatic>false</isStatic>
                    <isPublic>true</isPublic>
                    <isConstructor>false</isConstructor>
                    <returnType>boolean</returnType>
                    <interactionStrength>4</interactionStrength>
                    <methodParameters>
                        <methodParameter>
                            <index>0</index>
                            <name>p</name>
                            <primitive>false</primitive>
                            <isInterface>false</isInterface>
                            <isAbstract>false</isAbstract>
                            <isEnum>false</isEnum>
                            <checkedNull>false</checkedNull>
                            <objects />
                            <primitiveFields />
                            <possibleObjectTypes />
                            <enums />
                        </methodParameter>
                        <methodParameter>
                            <index>1</index>
                            <name>limitedAge</name>
                            <primitive>true</primitive>
                            <values />
                        </methodParameter>
                        <methodParameter>
                            <index>2</index>
                            <name>limitedHeight</name>
                            <primitive>true</primitive>
                            <values />
                        </methodParameter>
                    </methodParameters>
                    <conditions>
                        <condition>
                            <trueBranches />
                            <falseBranches>
                                <falseBranch>
                                    <trueBranches />
```

```
                                    <falseBranches />
                                    <conditionalExpressions>
                                        <expression>(p.height cmpg limitedHeight) >= 0</expression>
                                        <isInputDependent>true</isInputDependent>
                                        <argTypeMap>
                                            <entry>
                                                <key>limitedHeight</key>
                                                <value>double</value>
                                            </entry>
                                            <entry>
                                                <key>p.height</key>
                                                <value>input.data.object.data.Person</value>
                                                <value>double</value>
                                            </entry>
                                        </argTypeMap>
                                        <argLimitValue />
                                        <posibleDataType />
                                    </conditionalExpressions>
                                </falseBranch>
                            </falseBranches>
                            <conditionalExpressions>
                                <expression>p.age <= limitedAge</expression>
                                <isInputDependent>true</isInputDependent>
                                <argTypeMap>
                                    <entry>
                                        <key>p.age</key>
                                        <value>input.data.object.data.Person</value>
                                        <value>int</value>
                                    </entry>
                                    <entry>
                                        <key>limitedAge</key>
                                        <value>int</value>
                                    </entry>
                                </argTypeMap>
                                <argLimitValue />
                                <posibleDataType />
                            </conditionalExpressions>
                        </condition>
                    </conditions>
                </method>
            </methods>
        </class>
    </classes>
</controlFlowAnalysisOutput>
```

Zdrojový kód B.1: Ukázka výstupu analýzy grafu toku řízení

C Ukázka výstupu po generování testovacích dat

```xml
<?xml version="1.0" encoding="UTF-8"?>
<controlFlowAnalysisOutput>
    <classes>
        <class>
            <classPath>/path/</classPath>
            <packageName>input.data.object.medium</packageName>
            <className>TestObjectMedium</className>
            <hasPublicConstructor>true</hasPublicConstructor>
            <methods>
                <method>
                    <name>canEnter</name>
                    <parameterCount>3</parameterCount>
                    <isStatic>false</isStatic>
                    <isPublic>true</isPublic>
                    <isConstructor>false</isConstructor>
                    <returnType>boolean</returnType>
                    <interactionStrength>4</interactionStrength>
                    <methodParameters>
                        <methodParameter>
                            <index>0</index>
                            <name>p</name>
                            <primitive>false</primitive>
                            <isInterface>false</isInterface>
                            <isAbstract>false</isAbstract>
                            <isEnum>false</isEnum>
                            <checkedNull>false</checkedNull>
                            <objects />
                            <primitiveFields>
                                <primitiveField>
                                    <index>0</index>
                                    <name>age</name>
                                    <primitive>true</primitive>
                                    <values>
                                        <value>626284494</value>
                                        <value>2147483647</value>
                                        <value>-2147483648</value>
                                        <value>-2121053268</value>
                                        <value>74</value>
                                    </values>
                                </primitiveField>
                                <primitiveField>
                                    <index>0</index>
                                    <name>height</name>
                                    <primitive>true</primitive>
                                    <values>
                                        <value>1.7976931348623157E308</value>
                                        <value>2.604790513837288</value>
                                        <value>4.2761873970442394</value>
                                        <value>4.968148180772963E307</value>
                                        <value>4.9E-324</value>
```

```
                   </values>
               </primitiveField>
           </primitiveFields>
           <possibleObjectTypes>
<possibleObjectType>input.data.object.data.Person</possibleObjectType>
           </possibleObjectTypes>
           <enums />
       </methodParameter>
       <methodParameter>
           <index>1</index>
           <name>limitedAge</name>
           <primitive>true</primitive>
           <values>
               <value>2147483647</value>
               <value>-943263219</value>
               <value>-2147483648</value>
               <value>74</value>
               <value>719009001</value>
           </values>
       </methodParameter>
       <methodParameter>
           <index>2</index>
           <name>limitedHeight</name>
           <primitive>true</primitive>
           <values>
               <value>1.1929004516803248E307</value>
               <value>1.7976931348623157E308</value>
               <value>4.2761873970442394</value>
               <value>1.5978028187925335</value>
               <value>4.9E-324</value>
           </values>
       </methodParameter>
   </methodParameters>
   <conditions>
       <condition>
           <trueBranches />
           <falseBranches>
               <falseBranch>
                   <trueBranches />
                   <falseBranches />
                   <conditionalExpressions>
                       <expression>( p.height - limitedHeight ) >= 0</expression>
                       <isInputDependent>true</isInputDependent>
                       <argTypeMap>
                           <entry>
                               <key>limitedHeight</key>
                               <value>double</value>
                           </entry>
                           <entry>
                               <key>p.height</key>
                     <value>input.data.object.data.Person</value>
                               <value>double</value>
                           </entry>
                       </argTypeMap>
                       <argLimitValue>
                           <entry>
                               <key>limitedHeight</key>
                               <value>4.2761873970442394</value>
```

```
                                            </entry>
                                            <entry>
                                                <key>p.height</key>
                                                <value>4.2761873970442394</value>
                                            </entry>
                                        </argLimitValue>
                                        <posibleDataType />
                                    </conditionalExpressions>
                                </falseBranch>
                            </falseBranches>
                            <conditionalExpressions>
                                <expression>p.age <= limitedAge</expression>
                                <isInputDependent>true</isInputDependent>
                                <argTypeMap>
                                    <entry>
                                        <key>p.age</key>
                                        <value>input.data.object.data.Person</value>
                                        <value>int</value>
                                    </entry>
                                    <entry>
                                        <key>limitedAge</key>
                                        <value>int</value>
                                    </entry>
                                </argTypeMap>
                                <argLimitValue>
                                    <entry>
                                        <key>p.age</key>
                                        <value>74.76669828549312</value>
                                    </entry>
                                    <entry>
                                        <key>limitedAge</key>
                                        <value>74.76669828549312</value>
                                    </entry>
                                </argLimitValue>
                                <posibleDataType />
                            </conditionalExpressions>
                        </condition>
                    </conditions>
                </method>
            </methods>
        </class>
    </classes>
</controlFlowAnalysisOutput>
```

Zdrojový kód C.1: Ukázka výstupu generování testovacích dat pro konkrétní parametry metody

D Část vygenerované testovací třídy obsahující jednotkové testy

```java
public class Calculations1Test {

  @Test
  public void calculateInstallmentsCountTest1() {
    int capital = 0;
    double annualInterestRate = 38330.19948632043;
    int installment = -6946;

    Calculations.calculateInstallmentsCount(capital, annualInterestRate,
        installment);
  }

  @Test
  public void calculateInstallmentsCountTest2() {
    int capital = 2147483647;
    double annualInterestRate = 38330.19948632043;
    int installment = -6946;

    Calculations.calculateInstallmentsCount(capital, annualInterestRate,
        installment);
  }

  ...

  @Test
  public void calculateCreditInterestTest1() {
    int capital = 0;
    int installment = -9667;
    int installmentsCount = 0;

    Calculations.calculateCreditInterest(capital, installment,
        installmentsCount);
  }


  @Test
  public void formatAmountKplnTest1() {
    int number = 370;

    Calculations.formatAmountKpln(number);
  }
}
```

Zdrojový kód D.1: Ukázka vygenerované třídy obsahující jednotkové testy třídy Calculations

E Uživatelská příručka

Vytvořená knihovna může být spuštěna i z příkazového řádku. Ovládání je relizováno pomocí několika parametrů.

- `-a <typ analýzi>`
  - Definuje jaký typ analýzy se bude po spuštění provádět.
  - Může nabývat tří hodnot `m,c` a `f`. Kde `m` je analýza metody třídy, `c` je analýza všech metod třídy a `f` je analýza celého projektu.
  - Povinný parametr.
- `-pfp <cesta>`
  - Cesta k analyzovanému projektu.
  - Povinný parametr.
- `-op <cesta>`
  - Cesta do adresáře, kam se vygenerují testovací třídy.
  - Povinný parametr.
- `-pp <balíček>`
  - Definuje balíček, který se bude v projektu analyzovat.
- `-c <jméno třídy>`
  - Jméno třídy, která se bude analyzovat.
- `-m <jméno metody>`
  - Jaká metoda se bude v projektu analyzovat.
- `-t <typ>`
  - Jakým způsobem se mají tabulky symbolů vytvářet. Pokud je typ nastaven na hodnotu `d` budou se tabulky vytvářet detailním způsobem, pro všechny možné cesty v grafu toku řízení
- `-d <číslo>`
  - Určuje velikost zanoření do volaných metod.
- `-ml <číslo>`
  - Určuje přibližný maximální počet řádků ve vygenerované třídě.
- `-mrt <datový typ>`
  - Specifikuje datový typ návratové hodnoty metody. Tento parametr může být použit užitečný při specifikaci přetížených metod.

- `-mpt <výčet datových typů>`
    - Specifikuje datové typy parametrů testované metody. Jednotlivé parametry jsou odděleny pomlčkou.
- `-l`
    - Zapne detailní načítání referencí testované třídy, které používá framework Soot.
- `-pm`
    - Budou se generovat jednotkové testy i pro metody s příznakem `private`.
- `-da`
    - Pokud je tento parametr přítomen, budou se generovat jen soubory obsahující vygenerované hodnoty.
- `-debug`
    - Do konzole se budou vypisovat více informací o zpracovávání testované třídy.