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

Diplomová práce

Kombinatorické generování testů

Plzeň 2019 Tomáš Matějka

Místo této strany bude zadání práce.

Prohlášení

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

V Plzni dne 15. května 2019

Tomáš Matějka

Poděkování

Rád bych zde poděkoval vedoucímu diplomové práce panu Ing. Richardu Lipkovi, Ph.D. za jeho podnětné rady a čas, který mi věnoval při řešení dané problematiky.

Abstract

Cílem této práce bylo vytvořit knihovnu pro automatické generování jednotkových testů s využitím nějaké metody kombinatorického testování. V teoretické části je popsáno matematické pozadí kombinatorických metod a podrobněji jsou rozebrány tři kombinatorické metody, které byly v rámci této práce prozkoumány. Zkoumané metody jsou srovnány a ta nejvhodnější je vybrána pro implementaci. Praktická část se zabývá implementací vytvořené knihovny, která se skládá z vybrané kombinatorické metody a generátoru, který na základě testovacích sad generovaných metodou vytváří konkrétní jednotkové testy.

Abstrakt

The aim of this thesis was to create a library for automatic generation of unit tests using a combinatorial test method. The theoretical part describes the mathematical background of combinatorial methods and three combinatorial methods that have been explored in this work are discussed in more detail. The examined methods are compared with each other and the most suitable is chosen for implementation. The practical part deals with the implementation of the created library, which consists of the chosen combinatorial method and a generator that, based on test sets generated by the method, creates specific unit tests.

Obsah

Úvod **8**

1 Automatické generování testů **9**
1.1 Rozdělení tříd ekvivalencí 10
1.2 Analýza hraničních hodnot 10
1.3 Kombinatorické testování 11
1.3.1 Matematické pozadí 11
1.3.2 Generování CA 13
1.4 Metriky pro ohodnocení a porovnání metod 15

2 Zkoumané kombinatorické metody **16**
2.1 VS-PSTG 16
2.1.1 Ohodnocení a výsledky 19
2.2 FSAPSO 22
2.2.1 Ohodnocení a výsledky 25
2.3 ATLBO 30
2.3.1 Ohodnocení a výsledky 34
2.4 Srovnání zkoumaných metod 36

3 Implementace **38**
3.1 ATLBO 38
3.1.1 Konfigurace 39
3.1.2 Populace 39
3.1.3 Interakce 39
3.1.4 Fuzzy logika 41
3.2 CTGen 42
3.2.1 Vstup 42
3.2.2 Transformace 44
3.2.3 Výstup 46

4 Testování **48**
4.1 Jednotkové testování 48
4.2 Porovnání účinnosti ATLBO implementací 48
4.3 Ohodnocení efektivity 52

Závěr **54**

Literatura **55**

Seznam zkratek **61**

Seznam obrázků **63**

Seznam tabulek **65**

A Ilustrace VS-PSTG algoritmu **66**

B Porovnání vygenerovaných sad metodou VS-PSTG **67**

C Ilustrace FSAPSO algoritmu **68**

D Porovnání vygenerovaných sad metodou FSAPSO **69**

E Reakce FSAPSO sady na mutace **72**

F Porovnání vygenerovaných sad metodou ATLBO **73**

G Popis fuzzy systému jazykem FCL **75**

H Vstupní soubor generátoru testovacích tříd **76**

I Příklad vygenerovaných testů **78**

J Obsah přiloženého CD **80**

K Uživatelská příručka **81**

Úvod

Při testování softwarových systémů s velkým počtem konfigurovatelných parametrů se některé chyby projeví jen při specifické interakci několika parametrů s konkrétně nastavenými hodnotami. Kvůli vysokému počtu parametrů by vytvoření manuálních testů bylo příliš časově náročné, takže je potřeba automatizace testů. Interakce mezi parametry znamená, že klasické techniky pro automatické generování testů nebudou tak efektivní, jelikož nezaručují objevení chyb způsobených interakcemi parametrů.

Je tedy potřeba použít techniky, které umí nalézt i chyby způsobené interakcemi libovolného počtu parametrů. Mezi rozšířené a úspěšně používané techniky patří metody kombinatorického testování.

Kombinatorické testování je dlouhodobě zkoumaná oblast a za posledních dvacet let vznikla řada různých kombinatorických metod, které se dělí do několika tříd. V této práci se rozebírají tři konkrétní metaheuritiscké kombinatorické metody, které byly v rámci této práce podrobně prozkoumány. Zkoumané metody jsou porovnány na základě jimi prezentovaných výsledků a nejvhodnější z nich je vybrána pro vlastní implementaci.

Cílem této diplomové práce je tedy vytvořit knihovnu, která bude s pomocí vybrané kombinatorické metody schopna generovat konkrétní jednotkové testy. Tato práce je uspořádána následovně:

- První kapitola (1) se zabývá testovacími technika, především se zaměřuje na kombinatorické testování, jeho matematické pozadí a existující metody pro kombinatorické generování testů a jejich dělení.
- Druhá kapitola (2) se věnuje popisu a srovnání zkoumaných kombinatorických metod.
- Třetí kapitola (3) rozebírá podobu vlastní implementace vybrané kombinatorické metody a kombinatorického generátorů testů, který je založen na této metodě.
- Čtvrtá a poslední kapitola (4) prezentuje jakým způsobem byla otestována funkčnost vytvořené knihovny a zhodnocuje dosažené výsledky.

1 Automatické generování testů

Testování je velmi důležitou součásti vývoje softwaru, která pomáhá zajistit jeho dostatečnou kvalitu. Z finančních důvodů je důležité detekovat chyby v softwaru včas, protože čím déle v životním cyklu je chyba objevena, čím dražší je její odstranění [18, 40]. Vývoj softwaru se stále zrychluje, roste jeho komplexita, ale požadavky na jeho kvalitu jsou minimálně stejné, ne-li vyšší než tomu bylo dříve. Kvůli tomu roste i potřeba a důležitost automatického testování místo drahých a často neefektivních manuálních testů. S automatickými testy přichází několik výhod jako je rychlost, opakovatelnost a možnost otestovat větší množství vstupních dat. Testování je obecně klasifikováno především jako funkcionální a strukturální.

Ve funkcionálním, neboli black-box, testování není k dispozici přístup ke zdrojovému kódu testovaného systému a není přesně známo jak systém pracuje s daty. Zaměřuje se tedy jen na to jaký výsledek se získá pro vstupní data. Naopak při strukturálním, neboli white-box, testování je k dispozici zdrojový kód a je tedy známa vnitřní strukturu testovaného systému. Díky tomu se mohou otestovat neočekávané vstupní hodnoty a další případy získané díky znalosti vnitřní struktury systému.

Z důvodu neznalosti vnitřní struktury programu je získání (generování) testovacích dat důležitým úkolem ve funkcionálním testování. Obecně metody pro generování testovacích dat využívají informace dostupné ve specifikacích požadavků na software, které poskytují znalosti o vstupních požadavcích. Při vytváření testovacích případů se zvažují všechny možná vstupní data. Vytvořit však testy pro všechny možné vstupy je prakticky nemožné. Proto je úloha testovacích technik velmi důležitá.

Testovací techniky jsou použity k systematickému výběru testovacích případů prostřednictvím specifického vzorkovacího mechanismu. Tento postup optimalizuje počet testovacích případů tak, aby bylo dosaženo optimální velikosti sady testů, čímž se eliminuje čas a náklady na fázi testování. Existují různé techniky vytváření testovacích sad jako je rozdělení tříd ekvivalence, analýza hraničních hodnot, analýza příčiny a následku pomocí rozhodovacích tabulek atd. Obecně se při testování používá více než jen jedna testovací technika, protože při použití různých technik mohou být zjištěny různé chyby.

Tyto techniky jsou užitečné pro nalezení a prevenci chyb, nicméně však nemohou detekovat chyby, které jsou způsobeny kombinací vstupních komponent a konfigurací [17]. Provést kompletní testování, ve kterém by se otestovali všechny možné kombinace je ovšem nemožné z časových důvodů. K vyřešení tohoto problému byli vyvinuty různé strategie. Mezi nimi se jako nejúčinnější ukázali strategie **kombinatorického testování**.

1.1 Rozdělení tříd ekvivalencí

Tato technika je založená na rozdělení sady vstupů testovaného systému do takzvaných tříd ekvivalence, od kterých je očekáváno stejné chování. Neboli pokud pro libovolný prvek z jedné třídy systém nevykazuje chybu, tak ji nebude vykazovat ani pro ostatní prvky třídy. Stejně tak pokud pro libovolný prvek z druhé třídy systém vykazuje chybu, bude ji vykazovat i pro ostatní prvky. Pro každou třídu ekvivalence pak stačí vytvořit jeden testovací případ s libovolnou hodnotou třídy.

Mějme jednoduchý program, který akceptuje čísla v rozsahu 0 až 100. Sada vstupních hodnot může být rozdělena do tří tříd. První je třída pro validní hodnoty 0 až 100. Druhou jsou nevalidní vstupy pro hodnoty menší než 0 a třetí jsou nevalidní vstupy pro hodnoty větší než 100. Pro takový příklad by vznikly tři testovací případy.

Rozdělení tříd ekvivalence může být použito na jakékoliv úrovni testování a často je dobré ji použít jako první techniku pro omezení množiny vstupních dat.

1.2 Analýza hraničních hodnot

Analýza hraničních hodnot je rozšířením předchozí techniky. Tato technika testuje chování systému v hraničních hodnotách ekvivalenčních tříd. Vychází z předpokladu, že velká část selhání systému je způsobeno špatným ošetřením hraničních hodnot a pokud tedy systém správně funguje pro extrémní (krajní) hodnoty, bude správně fungovat i pro všechny ostatní. Pro každou hranici třídy tedy vzniknou tři testovací případy a to pro hodnotu přesně na hranici, těsně pod hranicí a těsně nad hranicí.

Pro příklad z předchozí sekce máme dvě hranice, dolní hranici 0 a horní hranici 100. Pro takový případ by se tedy vytvořilo šest testovacích případů, tři pro dolní hranici s hodnotami $-1, 0, 1$ a tři pro horní hranici s hodnotami $99, 100, 101$

Analýza hraničních hodnot je jednoduchá, ale poměrně efektivní technika

pro nalezení chyb. Je aplikovatelná na všechny úrovně testování a obvykle se používá spolu s technikou rozdělení tříd ekvivalencí nebo s jinými black-box technikami.

1.3 Kombinatorické testování

Kombinatorické testování dokáže detekovat chyby vyvolané interakcí mezi několika parametry testovaného systému a je aktivně zkoumáno jíž dvacet let. Kombinatorické metody vychází z pozorování, že se ne na každé chybě podílí všechny parametry a většina chyb je vyvolána jedním parametrem nebo interakcí mezi menším počtem parametrů, obvykle dva až šest [24].

Příkladem chyby vyvolané jedním parametrem může být dělení dvou čísel a/b. Stačí aby byla splněna jedno podmínka - hodnota parametru b je 0 - a nastane chyba. Zatímco příklad 2-cestné chyby je například $a/(b-c)$. Taková chyba je komplexnější, protože dva různé parametry musí mít jedno konkrétní nastavení hodnot, aby se chyba projevila.

Pro parametry datových typů s velkým rozsahem hodnot (*double*, *long* atd.) je vhodné nejprve provést nějakou z výše popsaných technik - rozdělení tříd ekvivalence nebo analýzu hraničních hodnot - a tím získat rozumný počet možných vstupních hodnot parametru.

1.3.1 Matematické pozadí

Pro matematickou reprezentaci kombinací se běžně používá *covering array* (CA) notace zapsaná jako $\mathrm{CA}_\lambda(\mathrm{N};\ t,\ p,\ v)$. kde N je počet testovacích případů, p je počet parametrů testovaného systému a každý parametr má stejný počet hodnot v a t je síla interakce, která je reprezentována celým číslem větším než 1 [16]. Například $t = 2$ představuje párové neboli 2-cestné testování a znamená interakci mezi dvěma parametry systému.

Notace $\mathrm{CA}_\lambda(\mathrm{N};\ t,\ p,\ v)$ reprezentuje matici N × p nad v $(0, ..., v-1)$ hodnotami takovou, že každá množina $B \in \binom{\{0,\ldots,p-1\}}{t}$ je λ-krát pokryta tak, že každá N × t podmatice obsahuje všechny uspořádané podmnožiny z v hodnot o velikosti t alespoň λ-krát [32]. Při testování obecně stačí, aby byl každý případ pokryt jednou, takže $\lambda = 1$ a notace je potom $\mathrm{CA}(\mathrm{N};\ t,\ p,\ v)$ [21]. Pokud CA obsahuje minimální počet řádek (N), nazývá se optimální CA a značí se $\mathrm{CAN}_\lambda(t,\ p,\ v)$.

0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

Obrázek 1.1: Dva příklady pro CA (5, 2, 2^4).

Problém CA notace je, že hodnoty pro každý parametr musí být uniformní, neboli každý parametr musí mít stejný počet hodnot. V reálných případech však většinou mají parametry různý počet hodnot. Proto je zavedena notace *mixed level covering array* (MCA), zapsána jako MCA(N; t, p, (v_1, v_2, ..., v_i)) nebo MCA(N; t, $v_1{}^{p1}$, $v_2{}^{p2}$, ..., $v_i{}^{pi}$), kde v_i udává počet hodnot i-tého parametru [46].

0	0	0	0	0
0	1	1	1	1
1	0	0	1	1
1	1	1	0	0
2	0	1	0	1
2	1	0	1	0

Obrázek 1.2: Příklad pro MCA (6, 2, 3^1 2^4).

Některé parametry spolu mohou interagovat silněji než s ostatními parametry. Pro tento případ se zavedla notace *variable-strength covering array* (VSCA) zapsaná jako VSCA (N; t, p, v, (CA_1 ... CA_j)). VSCA je v podstatě CA nebo MCA, která obsahuje další CA nebo MCA. Síla interakce pro CA_j musí být větší než síla pro VSCA [45].

2	1	1	1	1
0	1	0	0	0
1	0	0	1	1
2	0	1	0	0
1	1	1	1	0
0	0	1	0	1
2	1	0	0	1
0	0	0	1	0
1	1	1	0	1
2	1	0	1	1
2	0	0	0	1
2	0	1	1	1

Obrázek 1.3: Příklad pro VSCA (12, 2, 3^1 2^4, CA(12, 3, 2^4)).

V kombinatorickém testování se CA používá pro reprezentaci testovací sady a řádky CA tedy odpovídají konkrétním testovacím případům.

1.3.2 Generování CA

Generovat CA optimálních velikostí je nedeterministicky polynomiální (NP) výpočetní problém. Výpočetní čas a složitost problému roste exponenciálně s rostoucím počtem vstupních parametrů [31]. Podle [31] se metody pro generování CA dělí na matematické metody, náhodné metody, greedy algoritmy a algoritmy heuristického prohledávání. Greedy a heuristické algoritmy se označují za výpočetní.

Matematické metody

Matematické metody pro konstrukci CA jsou široce zkoumané v matematické komunitě. Existují dva přístupy matematických metod k vytváření testovacích sad. V prvním přístupu se sady vytvářejí přímo na základě matematické funkce, která spočte hodnotu každé buňky podle indexů řádky a sloupce. Tento přístup je většinou rozšířením matematických metod pro konstrukci ortogonálních polí. Druhý přístup je založený na rekurzivním konstruování větších testovacích sad z menších [25]. Nástroje používající matematické metody pro generování CA jsou například TConfig [42], Combinatorial Test Services (CTS) [20] a TestCover. Problémem matematických metod je, že nejsou obecné, to znamená, že selhávají při generaci CA pro větší množství parametrů a hodnot, zejména pokud počet hodnot parametrů není uniformní [6]. Použitelnost těchto metod je kvůli tomu tedy omezena.

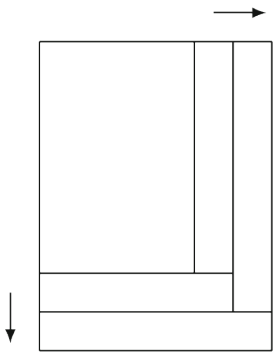
Náhodné metody

Náhodné metody náhodně vybírají testovací případy ze sady všech možných testů na základě nějaké vstupní distribuci. Tyto metody obvykle nedosahují značných výsledků [32] a jsou spíše použity pro porovnání účinnosti s jinými metodami generování a ukázání jejich schopnosti detekování chyb [35].

Greedy algoritmy

Greedy algoritmy konstruují testovací sady tak, aby každý nově vytvořený test pokrýval co nejvíce možných nepokrytích kombinací. Tyto algoritmy se dále dělí do dvou tříd. První je *one-test-at-a-time* nebo také *one-row-at-a-time*. V každé iteraci one-test-at-a-time algoritmu je CA rozšířena o jeden test (řádek), který pokrývá co nejvíce možných nepokrytých kombinací. Strategie one-test-at-a-time byla poprvé použita v algoritmu *Automatic efficient test generator* (AETG) [14]. AETG v každé iteraci vygeneruje sadu kandidátních testů, ze kterých poté chamtivě (greedily) vybere ten, který pokrývá nejvíce kombinací.

Druhou třídou je *one-parameter-at-a-time*. Tyto algoritmy začínají vygenerováním všech možných kombinací pro prvních t parametrů. Tuto sadu pak postupně rozšiřuje horizontálně přidáním jednoho parametru do každé řádky a pokud je potřeba, rozšíří sadu vertikálně o nové řádky. Tato strategie byla poprvé použita v algoritmu *In-parameter-order* (IPO) [9] a dále rozvinuta v modifikacích IPO algoritmu jako jsou IPOG [26], IPOG-D [27], IPOG-F [19] a IPO-s [10].



Obrázek 1.4: Znázornění one-parameter-at-a-time algoritmů.

Algoritmy heuristického prohledávání

Techniky založené na heuristickém prohledávání a umělé inteligenci byly efektivně aplikovány pro generování CA. Tyto techniky obecně začínají s ná-

hodnou sadou možných řešení. Na tuto sadu se pak iterativně aplikuje nějaký transformační mechanismus, který vytvoří novou sadu, která musí být lepší v pokrývání kombinací. Hlavní rozdíl mezi jednotlivými technikami je v transformačním mechanismu. Mezi tyto techniky patří například Simulated annealing (SA) [17], Tabu search (TS) [33], Genetic algortihm (GA) [37], Ant colony algorithm (ACA) [11, 37] a Particle Swarm Optimization (PSO) [7, 23]. Tyto techniky většinou generují menší testovací sady než ostatní metody, ale typicky vyžadují více času [6].

1.4 Metriky pro ohodnocení a porovnání metod

Pro ohodnocení a porovnání kombinatorických metod se v literatuře (např. [2, 6, 30, 46]) používá *efektivita*, *účinnost* a *výkon*.

Efektivita je schopnost metodou generovaných sad detekovat chyby v testovaném softwaru. Metoda A je efektivnější v detekování chyb než metoda B, pokud její testovací sady detekují více chyb v testovaném softwaru. Efektivita detekce chyb se počítá podle rovnice 1.1.

$$\text{Efektivita} = \frac{\text{počet nalezených chyb}}{\text{počet všech chyb}} \times 100\ \% \tag{1.1}$$

Účinnost je měřena velikostí testovacích sad generovaných metodou. Metoda A je účinnější než metoda B, pokud generuje testovací sady menší velikosti. Účinnost se tedy zabývá optimalitou generování sad a efektivita se zabývá použitelností. Výkon měří čas, který metoda potřebuje k vytvoření testovací sady. Samozřejmě, aby porovnání podle času bylo férové, porovnávané metody musí být spuštěny ve stejném prostředí (hardware, operační systém, programovací jazyk atd.).

2 Zkoumané kombinatorické metody

Tato kapitola se věnuje popisu tří vybraných kombinatorických metod, které byly v rámci práce prozkoumány. Všechny popisované metody se řadí mezi metaheuristické. Na konci kapitoly jsou metody srovnány a je řečeno jaká byla vybrána pro implementaci.

2.1 VS-PSTG

VS-PSTG [2], celým názvem *Variable-strength Particle Swarm Test Generator*, je metoda vytvořená Bestoun S. Ahmedem a Kamal Z. Zamlim a publikovaná v roce 2011.

Metoda staví na již existujícím algoritmu Particle Swarm Optimizization (PSO) [23], který se osvědčil v několika oblastech výzkumu ([1, 22, 38]). Hlavní motivací autorů pro použití PSO algoritmu je jeho malá náročnost na paměť a procesor. Třemi hlavními vlastnostmi, které stojí za efektivitou PSO algoritmu jsou *rekombinace*, *mutace* a *selekce*. PSO nemá přímý operátor rekombinace navzdory stochastickému zrychlení částice směrem k její předchozí nejlepší pozici, připomínající rekombinaci z jiných technik. Místo toho se informace vyměňují pouze mezi částicí a nejlepší částicí hejna. Z hlediska mutace, standardní PSO má výhodu v tom, že nepoužívá evoluční operátory jako křížení a mutace a tím se snižuje výpočetní zátěž. Pro selekci PSO nepoužívá koncept přežití nejsilnějšího, takže během procesu optimalizace i částice s nižším ohodnocením přežijí a mohou navštívit jakékoliv místo stavového prostoru.

PSO je optimalizační metoda vytvořená Kennedym a Eberhartem v roce 1995, inspirovaná chováním hejn ryb a ptáků. PSO pracuje nad stavovým prostorem (hejnem), jehož prvky se nazývají částice. Částice jsou v prostoru reprezentovány dvěma složkami - stochastickou rychlostí a deterministickou pozicí. Každá složka je reprezentována D-dimenzionálním vektorem, kde D je rovno počtu parametrů testovaného systému. Pozice je označena jako $X_j = [X_{j,1}, X_{j,2}, ..., X_{j,D}]$ a reprezentuje jedno možné řešení, neboli jeden výstupní testovací případ. Každá dimenze pozice reprezentuje jeden testovací parametr a může nabývat hodnot 0 až v_i, kde v_i je počet hodnot, kterých může nabývat i-tý parametr. Rychlost je označena jako

$V_j = [V_{j,1}, V_{j,2}, ..., V_{j,D}]$ a používá se k pohybu částic prostorem. Každá částice má své ohodnocení, které udává kolik elementů interakce pokrývá. Dále si částice udržuje pozici, ve které dosáhla nejlepšího ohodnocení, značenou *pBest*.

PSO při prohledávání prostoru upravuje trajektorie jednotlivých částic hejna a snaží se je dostat do pozice s nejlepším ohodnocením. V D-dimenzionálním prostoru se rychlost a pozice d-té dimenze j-té částice aktualizuje podle následujících rovnic:

$$\begin{aligned} V_{j,d}(t) = wV_{j,d}(t-1) + c_1 r_{j,d}(pBest_{j,d}(t-1) - X_{j,d}(t-1)) \\ + c_2 r'_{j,d}(lBest_{j,d}(t-1) - X_{j,d}(t-1)) \end{aligned} \tag{2.1}$$

$$X_{j,d} = X_{j,d}(t-1) + V_{j,d}(t), \tag{2.2}$$

kde t je číslo iterace, d je dimenze j-té částice, w určuje váhu setrvačnosti, r a r' jsou dvě reálná náhodná čísla z intervalu $[0, 1]$ sloužící k vytvoření elementu náhody. Hodnoty c_1 a c_2 jsou koeficienty zrychlení, které přizpůsobují váhu mezi komponentami. Studie [29, 43] doporučují nastavit parametr w na hodnotu mezi $0,1$ a $0,9$. Zjišťování optimálních hodnot se obecně děje během implementace algoritmu laděním jednotlivých parametrů. Pro tento algoritmus autoři zjistili, že hodnota setrvačnosti $0,3$ a hodnoty koeficientů zrychlení $1,375$ poskytují dobré výsledky, pokud je počet iterací algoritmu větší než 30 a počet částic větší než 110.

Jak bylo řečeno, každá dimenze pozice částice je omezena, takže i stavový prostor má své hranice. Během iterování algoritmu se částice může dostat mimo hranice prostoru. Pro řešení této situace existují hraniční pravidla, třemi známými jsou tzv. neviditelné, odrážející a absorbující hranice. Při použití neviditelných hranic je částice, která se nachází mimo stavový prostor, ignorována během výpočtu ohodnocení. Odrážející hranice otočí znaménko rychlosti částice, která se nachází mimo stavový prostor. Absorbující hranice vynulují dimenzi částice, která je mimo stavový prostor. V této metodě však není použita ani jedna z nich, a místo toho autoři používají pravidlo, ve kterém dimenze částice, která překročí jednu hranici pokračuje v pohybu stejnou rychlostí, ale od druhé hranice. Například, mějme parametr s rozsahem hodnot 0-4, pozice větší než 4 je považována za hranicemi stavového prostoru a je resetována na hodnotu 0.

Pohyb částice je omezen maximální rychlostí, které může částice dosáhnout. Pokud by zvolená maximální rychlost byla příliš veliká, částice se ze stavového prostoru dostane příliš rychle. Na druhou stranu, příliš malá

maximální rychlost vede k prohledávání lokálního optima. Autoři zvolili omezení rychlosti podle vzorce $V_{i_{max}} = V_i/2$ a interval omezující rychlost tedy je $[-V_{i_{max}}, V_{i_{max}}]$. Toto omezení tedy znamená, že částice se v jedné aktualizaci dimenze pozice nemůže pohnout o více než polovinu rozsahu aktualizované dimenze.

Dalším problémem během pohybu částic je vznik vektorů obsahující reálná čísla, když se počítají nové hodnoty rychlosti a pozice podle rovnic 2.1 a 2.2. Jelikož stavový prostor je tvořen diskrétnímy hodnotami parametrů, je potřeba se těchto reálných čísel zbavit. Algoritmus tento problém řeší tak, že reálná čísla zaokrouhluje na nejbližší celé číslo.

Obrázek v příloze A znázorňuje celý proces metody VS-PSTG. Vstupem do metody je konfigurace, pro kterou se vygenerují všechny možné elementy interakce a přidají se do seznamu *Ps*. Dalším krokem je inicializace stavového prostoru - vytvoří se daný počet částic, které začínají na náhodne pozici a s náhodnou rychlostí. Poté algoritmus vykonává specifický počet iterací. V každé iteraci se provede ohodnocení všech částic, to znamená, že se spočte kolik elementů interakce ze seznamu *Ps* částice pokrývá. Částice, která dosáhne maximálního možného ohodnocení je ihned přidána do finální testovací sady *Ts* a pokryté elementy interakce jsou odstraněny ze seznamu *Ps*. Částice, které nedosáhnou maximálního ohodnocení, jsou aktualizovány podle rovnic 2.1 a 2.2 a ta nejlepší z nich je označena jako *lBest*. Potom, co jsou všechny částice aktualizovány se vezme aktuální *lBest* a přidá se do finální testovací sady *Ts* a pokryté elementy interakce jsou odstraněny ze seznamu *Ps*. Algoritmus takto pokračuje dokud seznam *Ps* není prázdný.

```
1: Input: main configuration, main strength of coverage t_m;
2: Input:  sub-configuration, sub-strength of coverage t_s;
3: Output: A test case;
4: Manipulate the parameter and values for the sub and main configurations
5: Let Ps be a set of all combinations of parameter values that are not been covered yet;
6: For each configuration {
7:       Generate t-interaction elements;
8:       Add t-interaction elements to Ps;
9: }
10:  Let Ts be a set of candidate tests;
11:  While Ps is not empty do {
12:     Randomly initialize particles Xi(t) and velocities Vi(t);
13:     For a specific number of iterations do {
14:        Evaluate Xi(t) by computing the weight of coverage;
15:        If Xi(t) maximum weight of coverage is reached{
16:           Add Xi(t) to final test suite Ts;
17:           Remove Xi(t) from Ps;
18:           Continue;
19:           }
20:        Else {
21:           Choose the representative particle of best weight to be lBest;
22:           Calculate Vi(t+1) according to lBest;
23:           Move Xi(t) to Xi(t+1) according to Vi(t+1);
24:           }
25:        Evaluate Xi(t+1);
26:        If best weight of lBest(t+1) is achieved;
27:        lBest=lBest(t+1);
28:        }//End for
29:     Let gBest be the best test case found;
30:     gBest = lBest(t+1);
31:     Add gBest to the test set Ts;
32:     Remove those combinations in Ps that covered by Ts;
33: }
```

Obrázek 2.1: Algoritmus VS-PSTG. Zdroj [2].

2.1.1 Ohodnocení a výsledky

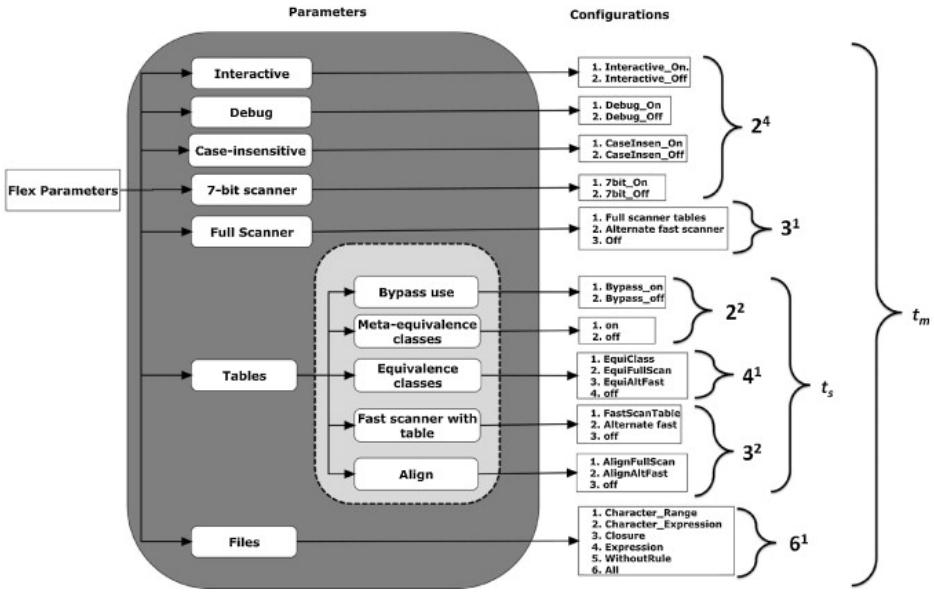
Autoři se při ohodnocení metody VS-PSTG zaměřili na dvě kritéria. První je schopnost generovat testovací sady menší velikosti, než jiné existující metody. Druhou je efektivita generovaných testovacích sad detekovat chyby.

Jak si VS-PSTG daří v ohledu schopnosti generovat co nejmenší testovací sady vůči jiným metodám, které autoři vybrali, můžete vidět v tabulce 2.1 a tabulkách v příloze B. Jelikož VS-PSTG produkuje nedeterministické výsledky, všechny testované konfigurace byly spuštěny desetkrát a nejmenší vyprodukovaná testovací sada byla použita jako výsledek. Z prezentovaných výsledků můžeme říci, že testovací sady vygenerované VS-PSTG metodou jsou ve většině případů menší, obzvlášť pokud je síla interakcí testované konfigurace vysoká.

{C}	PICT	ParaOrder	ACS	TVG	SA	Density	IPOG	ITCH	VS-PSTG
Configuration VSCA (m; 2, 3^{15}, {C})									
∅	35	33	19	22	16	21	21	31	19
CA (3, 3^3)	81	27	27	27	27	28	27	48	27
CA (3, 3^3)^2	729	33	27	30	27	28	30	59	27
CA (3, 3^3)^3	785	33	27	30	27	28	33	69	27
CA (3, 3^4)	105	27	27	35	27	32	39	59	30
CA (3, 3^5)	131	45	38	41	33	40	39	62	38
CA (4, 3^4)	245	NA	NA	81	NA	NA	81	103	81
CA (4, 3^5)	301	NA	NA	103	NA	NA	122	118	97
CA (4, 3^7)	505	NA	NA	168	NA	NA	181	189	158
CA (5, 3^5)	730	NA	NA	243	NA	NA	243	261	243
CA (5, 3^7)	1356	NA	NA	462	NA	NA	581	481	441
CA (6, 3^6)	2187	NA	NA	729	NA	NA	729	745	729
CA (6, 3^7)	3045	NA	NA	1028	NA	NA	1196	1050	966
CA (3, 3^4) CA (3, 3^5) CA (3, 3^6)	1376	44	40	53	34	46	51	114	45
CA (3, 3^6)	146	49	45	48	34	46	53	61	45
CA (3, 3^7)	154	54	48	54	41	53	58	68	49
CA (3, 3^9)	177	62	57	62	50	60	65	94	57
CA (3, 3^{15})	83	82	76	81	67	70	NS	132	74

Tabulka 2.1: Srovnání velikostí vygenerovaných sad pro konfiguraci VSCA(m: 2, 3^{15}, {C}). Zdroj [2].

Pro ověření efektivity detekce chyb autoři použili program "flex v.2.4.7", který je rozšířen o knihovnu pro zavedení chyb do programu. Knihovna umožňuje zavádět 18 různých chyb, které uživatel může ručně vypínat a zapínat. Na obrázku 2.2 můžete vidět parametry flexu a jejich možné hodnoty.



Obrázek 2.2: Parametry flexu a jejich možné hodnoty. Zdroj [2].

Pro testování bylo použito několik konfigurací, viz. tabulka 2.2, které se liší v síle interakce, počtu parametrů a podkonfiguracích. Absence testovací sady T8 není chyba v této práci, ale sada chybí už v originálním článku.

Test no.	Test suite	Size
T1	MCA (N; 2, 2^4 3^1 16^1 6^1)	96
T2	VSCA (m; 2, 2^6 3^3 4^1 6^1{MCA (3, 2^2 4^1 3^2)})	37
T3	MCA (N; 3, 2^4 3^1 16^1 6^1)	290
T4	VSCA (m; 3, 2^6 3^3 4^1 6^1{MCA (4, 2^2 4^1 3^2)})	104
T5	MCA (N; 4, 2^4 3^1 16^1 6^1)	764
T6	VSCA (m; 4, 2^6 3^3 4^1 6^1{MCA (5, 2^2 4^1 3^2)})	315
T7	MCA (N; 5, 2^4 3^1 16^1 6^1)	1708
T9	MCA (N; 6, 2^4 3^1 16^1 6^1)	2579
T10	TSLframe	525
T11	Exhaustive	41472

Tabulka 2.2: Testovací sady a jejich velikosti. Zdroj [2].

Testovací sady označené jako *TSLframe* a *Exhaustive* jsou použity pro porovnání vůči sadám generovaným metodou VS-PSTG. *TSLframe* je sada testů připojená k flexu a *Exhaustive* je sada obsahující všechny možné kombinace vstupů ($2 \times 2 \times 2 \times 2 \times 3 \times 2 \times 2 \times 4 \times 3 \times 3 \times 6$).

Tabulka 2.3 ukazuje jednotlivé sady (sloupce *T1* až *T11*) z předchozí tabulky a kolik z dříve zmíněných 18-ti možných chyb (řádky *f1* - *f18*) nalezly. Za pozornost stojí obzvlášť sada *T4* o velikosti 104 testů, která objevila stejný počet chyb jako sada *Exhaustive* o velikosti 41 472. Podle prezentovaných výsledků se dá říct, že se metoda osvědčila a generuje testovací sady, které jsou podstatně menší a naleznou stejný počet chyb.

f	T										
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
f1	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f2	-	-	-	-	-	-	-	-	-	-	-
f3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f6	-	-	-	✓	✓	✓	✓	✓	✓	✓	✓
f7	-	-	✓	✓	✓	✓	✓	✓	✓	✓	✓
f8	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f11	-	-	-	-	-	-	-	-	-	-	-
f12	-	-	-	-	-	-	-	-	-	-	-
f13	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f14	-	-	-	✓	✓	✓	✓	✓	✓	✓	✓
f15	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f16	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f17	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
f18	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
% f. Cov.	61,11%	66,66%	72,22%	83,33%	83,33%	83,33%	83,33%	83,33%	83,33%	83,33%	83,33%

Tabulka 2.3: Porovnání testovacích sad vzhledem k počtu nalezených chyb. Zdroj [2].

2.2 FSAPSO

Metoda FSAPSO [30], celým názvem *Fuzzy Self Adaptive Particle Swarm Optimization*, byla publikována v roce 2015 a jejími autory jsou Thair Mahmoud a Bestoun S. Ahmed.

Jak název napovídá, stejně jako VS-PSTG, i tato metoda využívá algoritmus PSO. Zkušenost [3, 28] výzkumných pracovníku s PSO algoritmem je taková, že je náchylný k problémům s laděním hodnot parametrů (w, c_1 a c_2), které řídí prohledávání stavového prostoru. Existují důkazy [41], že pro získání optimálních řešení různě komplexních problémů jsou potřeba různé parametry. Tento problém je možné vyřešit pomocí mechanismu, který by monitoroval výkon PSO a adaptivně přizpůsoboval hodnoty parametrů a tím efektivněji řídil směr prohledávání k optimálnějším výsledkům.

Podle hodnot řídících parametrů PSO provádí buď globální nebo lokální prohledávání prostoru. Monitorování výkonu PSO ukazuje, že nízké hodnoty w a vysoké hodnoty c_1 a c_2 vedou ke globálnímu prohledávání, takže se hodí na začátku procesu prohledávání pro nalezení dobrých globálních pozic. Na druhou stranu, vysoké hodnoty w a nízké hodnoty c_1 a c_2 vedou k lokálnímu prohledávání a hodí se na konci procesu k přesnému nalezení optimálních hodnot [13].

V této metodě je jako mechanismus pro monitorování a přizpůsobení parametrů použit *Mamdami-type fuzzy inference system* (FIS). Pro každý

z řídících parametrů je vytvořen jeden FIS, který konkrétní parametr monitoruje. Přizpůsobení hodnot se provádí najednou pro všechny parametry během aktualizace rychlostí částic. Na základě literatury [29, 43] autoři zvolili následující rozsah parametrů:

$$1 \leq C_1 \leq 2; \;\; 1 \leq C_2 \leq 2; \;\; 1 \leq w \leq 2$$

První FIS je použitý pro přizpůsobování parametru w. Nové hodnoty w jsou počítány podle následující rovnice:

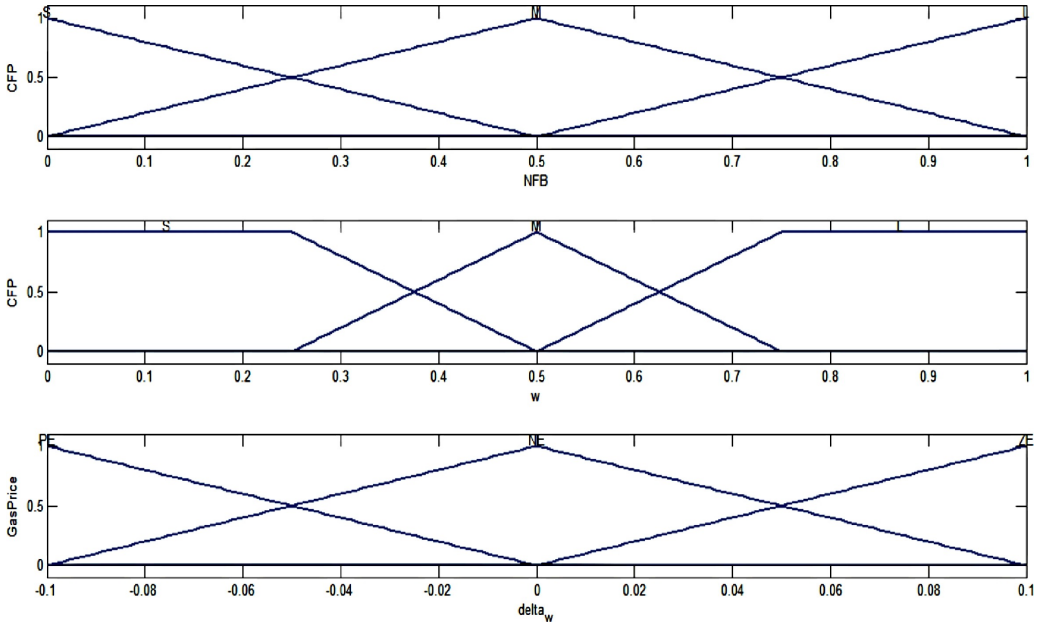
$$w^{k+1} = w^k + \Delta w, \tag{2.3}$$

kde w^k je hodnota parametru w v čase (iteraci) k a Δw je výsledná hodnota z FIS, viz obrázek 2.3 a tabulka 2.4. Δw je vyhodnocena ze dvou vstupů - normalizované nejlepší ohodnocení částice (NBFV viz. 2.4) a hodnoty parametru w v čase výpočtu (k).

$$\text{NBFV} = \frac{\text{Stopping Counter - Number of Achieved Covering}_k}{\text{Stopping Counter}} \tag{2.4}$$

Δw	W		
	S	M	L
S	ZE	NE	NE
M	PE	ZE	NE
L	PE	ZE	NE

Tabulka 2.4: Pravidla pro FIS parametru Δw.



Obrázek 2.3: Členské funkce (dvě vstupní, jedna výstupní) FIS přizpůsobující parametr Δw.

Přizpůsobení parametrů c_1 a c_2 závisí na normalizované hodnotě iterací, ve kterých nebylo nalezeno lepší ohodnocení. Hodnota se spočte pomocí rovnice:

$$\mathrm{NorNUBF} = \frac{\mathrm{NUBF}_{Max} - \mathrm{NUBF}_k}{\mathrm{NUBF}_{Max}}, \tag{2.5}$$

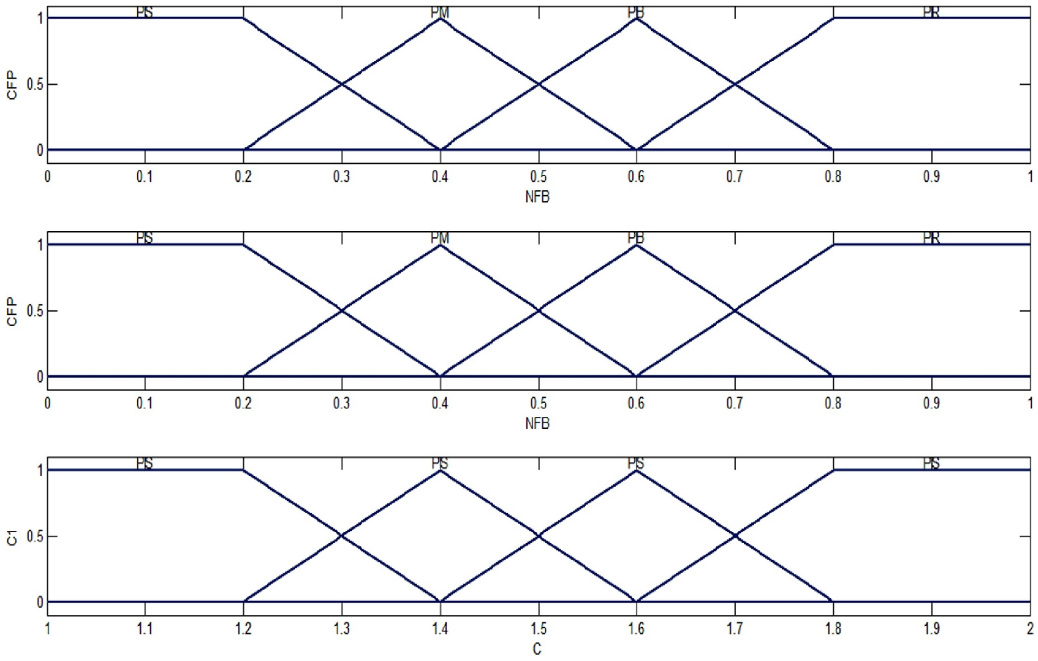
kde $\mathrm{NUBF_{Max}}$ je maximální počet iterací a $\mathrm{NUBF_k}$ je počet iterací, ve kterých se nenalezlo lepší ohodnocení. Tabulky 2.5 a 2.6 ukazují fuzzy pravidla pro parametry c_1 a c_2.

NBF	NU			
	PS	PM	PB	PR
PS	PR	PB	PM	PM
PM	PB	PM	PM	PS
PB	PB	PM	PS	PS
PR	PM	PM	PS	PS

Tabulka 2.5: Pravidla pro FIS parametru c_1.

NBF	NU			
	PS	PM	PB	PR
PS	PR	PB	PM	PM
PM	PB	PM	PM	PS
PB	PB	PM	PS	PS
PR	PM	PS	PS	PS

Tabulka 2.6: Pravidla pro FIS přizpůsobující parametr c_2.

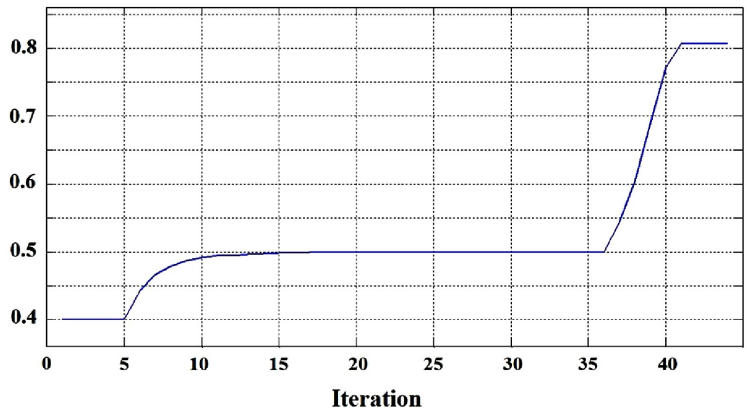


Obrázek 2.4: Členské funkce (dvě vstupní, jedna výstupní) FIS přizpůsobující parametr c_1 a c_2.

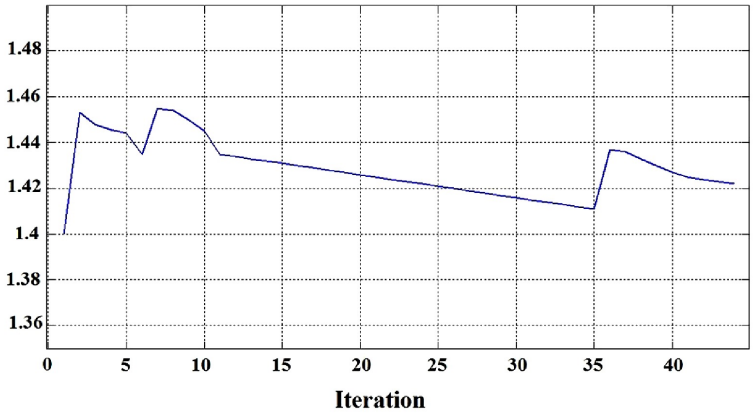
2.2.1 Ohodnocení a výsledky

Stejně jako v předchozí metodě se hodnotí schopnost metody generovat menší testovací sady než existující metody a efektivita generovaných sad detekovat chyby. U této metody se navíc hodnotí míra pokrytí kombinací a čas, jak dlouho trvá vytvořit testovací sadu.

Jelikož na kvalitě výsledků FSAPSO algoritmu hrají roli zavedená fuzzy pravidla a členské funkce, je provedeno testování adaptace parametrů w, c_1 a c_2. Testování je provedeno monitorováním hodnot parametrů v průběhu algoritmu. Obrázek 2.5 ukazuje, že w začíná na nízké hodnotě a postupně roste. Zatímco obrázek 2.6 ukazuje vývoj c_1 a c_2, které začínají na vysokých hodnotách a postupně klesají. Tento vývoj hodnot koresponduje s dřívějším tvrzením, že malé hodnoty w a vysoké hodnoty c vedou ke globálnímu prohledávání, které je vhodné na začátku prohledávání a vysoké hodnoty w a nízké hodnoty c vedou k lokálnímu prohledávání a hodí se na konci prohledávání. Dá se tedy říct, že vytvořený mechanismus dělá to, pro co byl vytvořen - směrování prohledávání od globálního po lokální.



Obrázek 2.5: Hodnota parametru w v průběhu algoritmu. Zdroj [30].



Obrázek 2.6: Hodnota parametru c_1 a c_2 v průběhu algoritmu. Zdroj [30].

Srovnání velikostí sad vygenerovaných algoritmem FSAPSO a jinými známými algoritmy ukazují tabulky 2.7, D.1, D.2, D.3, D.4, D.5, D.6 a D.7. Protože FSAPSO produkuje nedeterministické výsledky, je pro každou konfiguraci spuštěno 40-krát a nejmenší a průměrná velikost sady je vybrána pro prezentaci. Z výsledků se dá říct, že ve většině případů FSAPSO generuje lepší nebo alespoň konkurenceschopné velikosti sad.

v = 3											
	k	Jenny	TConfig	PICT	TVG	IPOG-D	IPOG	PSO		FSAPSO	
		Size	Size	Size	Size	Size	Size	Bst. size	Avg. size	Bst. size	Avg. size
t = 2	3	9	10	10	10	15	11	9	9.55	9	9.21
	4	13	10	13	12	15	12	9	10.15	9	9.95
	5	14	14	13	13	17	14	12	13.81	11	12.23
	6	15	15	14	15	17	15	13	15.11	12	13.78
	7	16	15	16	15	18	17	15	16.94	15	16.62
	8	17	17	16	15	18	17	15	17.57	15	16.92
	9	18	17	17	15	20	17	17	19.38	16	18.31
	10	19	17	18	16	20	20	17	19.78	17	18.12

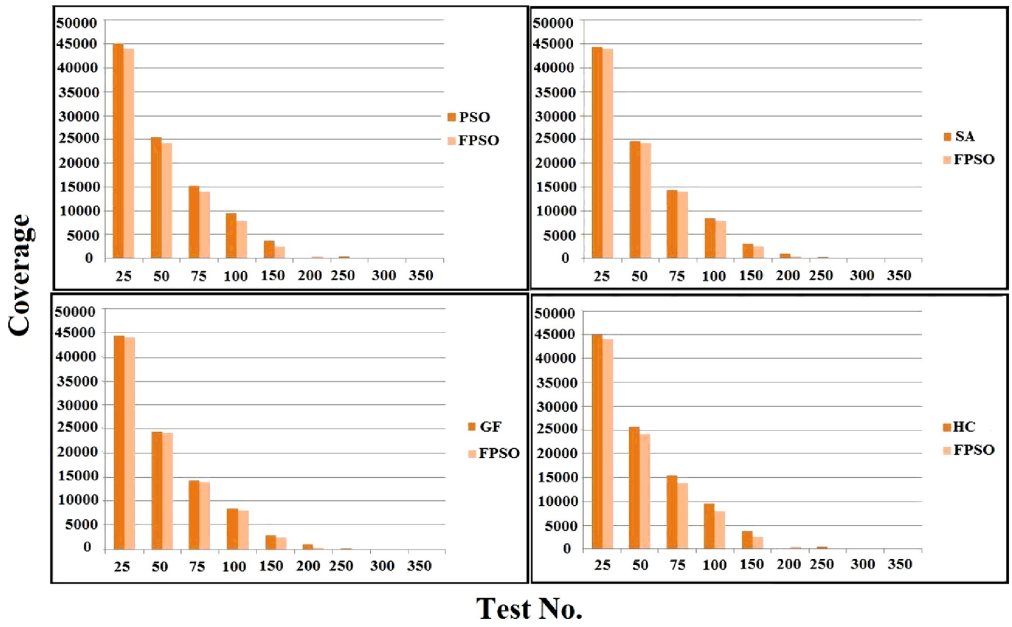
Tabulka 2.7: Srovnání velikostí testovacích sad pro CA (N; 2, k, 3). Zdroj [30].

Tabulky 2.8 a D.8 kromě velikostí navíc srovnávají čas potřebný ke konstrukci testovacích sad. Výsledky ukazují, že metaheuristické metody (PSO, FSAPSO) potřebují více času k vytvoření sad než výpočetní metody, ale vygenerované sady jsou menší.

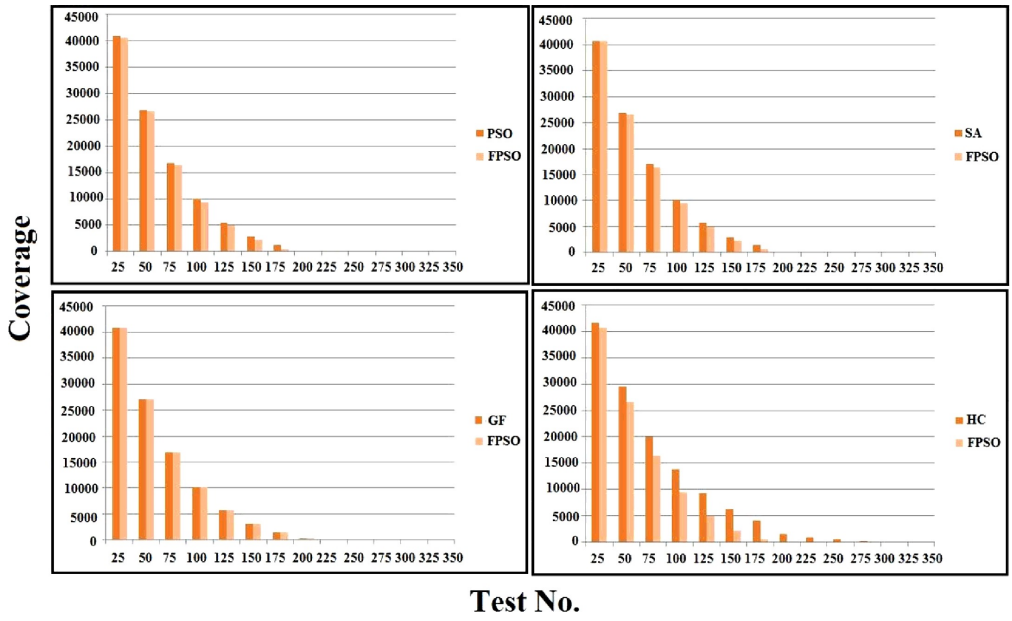
t	Jenny	TConfig	PICT	TVG	IPOG-D	IPOG	PSO		FSAPSO	
	N/Time	N/Time	N/Time	N/Time	N/Time	N/Time	Bst. N/Time	Avg. N/Avg. time	Bst. N/Time	Avg. N/Avg. time
2	16/0.37	15/0.29	16/0.62	15/0.22	18/0.19	17/0.443	15/0.21	15.23/0.32	15/1.42	15.12/2.38
3	51/0.57	55/1.86	51/0.98	55/0.57	63/0.36	57/0.614	50/4.21	55.2/5.56	48/13.5	51.12/16.74
4	169/0.62	166/18.5	168/1.46	167/0.82	NS	185/1.357	155/11.32	157.77/13.95	151/18.96	153.55/22.62

Tabulka 2.8: Porovnání velikostí a časů konstrukce testovacích sad pro CA (N; t, 7, 3). Zdroj [30].

Dále autoři srovnávají míru pokrytí kombinací. Porovnávané metody jsou horolezecký algoritmus (HC), simulované žíhání (SA), velká potopa (GF) a PSO. Obrázky 2.7 a 2.8 ukazují pokrytí jednotlivých metod a jejich srovnání s FSAPSO. Osa X udává počet testovacích případů a osa Y udává počet nepokrytých kombinací. Je vidět, že FSAPSO konzistentně pokrývá více kombinací s menším počtem testů. Jelikož FSAPSO pokrývá více kombinací s menším počtem testovacích případů než standardní PSO, můžeme to brát jako další důkaz funkčnosti mechanismu pro adaptování parametrů.



Obrázek 2.7: Porovnání míry pokrytí pro CA (N; 4, 3^{13}). Zdroj [30].



Obrázek 2.8: Porovnání míry pokrytí pro MCA (N; 4, 2^{10}, 3^3, 4^2, 5^1). Zdroj [30].

Pro ověření efektivity testovacích sad generovaných algoritmem FSAPSO autoři použili netriviální program, viz parametry a jejich hodnoty v tabulce 2.9, do kterého pomocí nástroje *MuClipse* [39] zavedli chyby (mutace).

No.	Factors	Levels
1	New graduate	[Unchecked, checked]
2	Children	[More_than_4, 1, 2, 3, 4]
3	English	[Unchecked, checked]
4	Read	[Unchecked, checked]
5	Degree	[Diploma, primary, secondary, no degree, master, bachelor, doctor]
6	Write	[Unchecked, checked]
7	Marital status	[Widow, single, married]
8	Understand	[Unchecked, checked]
9	Resident	[Foreigner, outsider, local]
10	Experience	[Unchecked, checked]
11	Speak	[Unchecked, checked]
12	Disability	[Unchecked, checked]

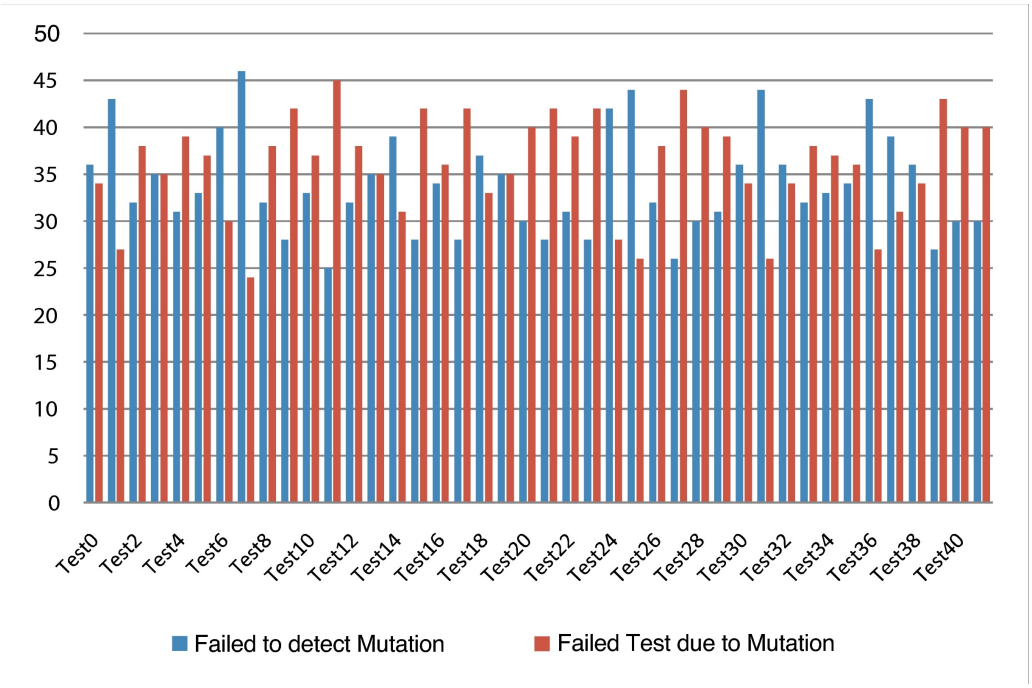
Tabulka 2.9: Parametry programu a jejich příslušné hodnoty. Zdroj [30].

Vstupní konfigurace tedy je MCA (N; t, 7^1, 6^1, 2^8, 3^2). Tabulka 2.10 ukazuje velikosti vygenerovaných sad pro různé hodnoty *t*. Pro ověření všech možných kombinací by bylo potřeba 96 768 ($7 \times 6 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 3 \times 3$) testů.

Interaction strength (t)	CA size
2	42
3	135
4	436
5	1202
6	2890

Tabulka 2.10: Velikosti testovacích sad pro různé *t*. Zdroj [30].

Na obrázku 2.9 je ukázána reakce 42 testů ze sady pro $t = 2$ na 70 mutací. Modrý pruh reprezentuje počet mutací, které nezměnily výstup programu. Červený pruh reprezentuje počet testů, které selhaly vlivem mutace. Těchto 42 testů neobjevilo 12 chyb - v článku se bohužel nepíše jaký je celkový počet chyb. Obrázek E.1 prezentuje reakci testovací sady pro $t = 3$ o velikosti 135 na stejných 70 mutacích. Tato sada objevila všechny chyby, včetně 12 chyb, které předchozí sada neobjevila.



Obrázek 2.9: Reakce testovací sady pro $t = 2$ na mutace. Zdroj [30].

2.3 ATLBO

ATLBO [46], neboli *Adaptive Teaching Learning-Based Optimization*, byla publikovaná v roce 2017 a autory jsou Kamal Z.Zamli, Fakhrud Din, Salmi Baharom a Bestoun S.Ahmed.

Základem pro ATLBO je již existující, jednoduchý a osvědčený algoritmus *Teaching Learning-Based Optimization* (TLBO) [34]. TLBO se inspiruje procesem učení mezi učitelem a jeho studenty. Učitel, který je znalejší než studenti, předává své znalosti studentům a studenti se také mohou učit od sebe navzájem.

Hlavním rozdílem oproti jiným metaheuristickým metodám je, že TLBO nepoužívá žádné kontrolní parametry (např. w, c_1 a c_2 v algoritmu PSO), které by ovládaly prohledávání prostoru. TLBO v každé iteraci algoritmu provede jak operaci globálního, tak lokálního prohledávání. Toto přednastavené dělení operací však může být kontraproduktivní a vést k horším výsledkům, protože výběr mezi globálním či lokálním prohledáváním by měl být dynamický, na základě aktuálně prozkoumávané oblasti stavového prostoru. Tento problém ATLBO řeší zavedením Mamdami-type fuzzy inference system (FIS), díky kterému ATLBO adaptivně volí mezi operacemi prohledávání.

Algorithm 1: The Original TLBO Algorithm

Input: the population $X = X_1, X_2 \ldots X_D$
Output: X_{best} and the updated population $X' = \{X'_1, X'_2, \ldots, X'_D\}$

Initialize random populations of learners X and evaluate all learners X
while *stopping criteria not met* **do**
  for $i = 1$ *to population size* **do**
    `/* Teacher Phase ... Exploration */`
    Select $X_{teacher}$ and calculate X_{mean}
    $T_F = round(1 + r(0, 1))$
    $X_i^{t+1} = X_i^t + r(X_{teacher} - T_F X_{mean})$
    if $f(X_i^{t+1})$ *is better than* $f(X_i^t)$ **then**
      $X_i^t = X_i^{t+1}$
    `/* Learner Phase ... Exploitation */`
    Randomly select one learner X_j^t from the population X such that $i \neq j$
    if $f(X_i^t)$ *is better than* $f(X_j^t)$ **then**
      $X_i^{t+1} = X_i^t + r(X_j^t - X_i^t)$
    else
      $X_i^{t+1} = X_i^t + r(X_i^t - X_j^t)$
    if $f(X_i^{t+1})$ *is better than* $f(X_i^t)$ **then**
      $[X_i^t = X_i^{t+1}$
  Get best result X_{best}

Obrázek 2.10: Původní TLBO algoritmus. Zdroj [46].

ATLBO pracuje nad stavovým prostorem, kterému se říká populace. Člen populace je reprezentován D-dimenzionálním vektorem označený $X_j = [X_{j,1}, X_{j,2}, X_{j,3}..., X_{j,D}]$. Člen populace reprezentuje jedno možné řešení, neboli jeden výstupní testovací případ. Každý člen má své ohodnocení, které udává kolik elementů interakce pokrývá. Každá dimenze reprezentuje jeden parametr a nabývá hodnot 0 až v_i, kde v_i je počet hodnot, kterých může nabývat i-tý parametr. Členovi populace s nejlepším ohodnocením se říká učitel a ostatním členům studenti.

ATLBO rozděluje proces prohledávání na dvě fáze - učitelskou a studentskou. Učitelská fáze provádí operaci globálního prohledávání. Během této fáze se algoritmus pokusí zlepšit ohodnocení studenta X_i tak, že s ním pohne směrem k učiteli podle následující rovnice:

$$X_i^{t+1} = X_i^t + r(X_{teacher} - T_F X_{mean}), \tag{2.6}$$

kde X_i^{t+1} je nová hodnota studenta X_i, $X_{teacher}$ je učitel v době výpočtu, X_{mean} je vektor reprezentující průměr populace, r je náhodné reálné číslo z intervalu $[0, 1]$ a T_F je faktor učení, který může nabývat buď hodnoty 1 nebo 2.

Studentská fáze přestavuje operaci lokální prohledávání. Student X_i^t prohlubuje svoji znalost interakcí s jiným náhodným studentem X_j^t. Student se od druhého studenta učí pouze pokud má druhý student více znalostí (lepší

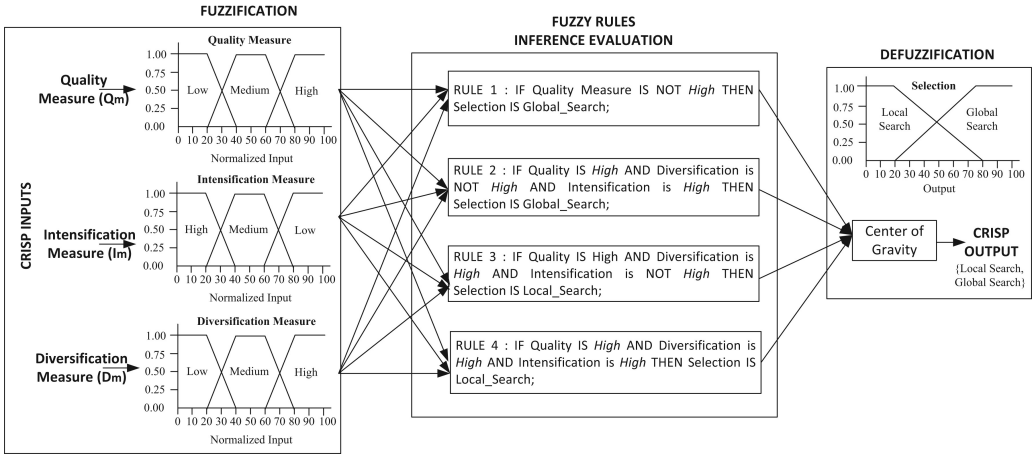
ohodnocení). To znamená, že pokud X_i^t je lepší než X_j^t, X_j^t se pohne směrem k X_i^t (viz 2.7). V opačném případě se X_i^t pohne směrem k X_j^t (viz 2.8).

$$X_i^{t+1} = X_i^t + r(X_j^t - X_i^t) \tag{2.7}$$

$$X_i^{t+1} = X_i^t + r(X_i^t - X_j^t) \tag{2.8}$$

Stavový prostor je omezený rozsahem jednotlivých parametrů a během výpočtu nové pozice se může dimenze prvku dostat mimo hranice prostoru a je potřebo ho vrátit zpět do prostoru. V první metodě byli popsány tři pravidla (neviditelné,odrážející a absorbující hranice) pro řešení této situace. V této metodě autoři použili absorbující hranice. Při překročení dolní hranice dimenze je hodnota dimenze nastavena na hodnotu horní hranice. Analogicky pro překročení horní hranice.

Obrázek 2.11 ilustruje vytvořený FIS pro výběr operace prohledávání. FIS se skládá ze tří vstupů a jednoho výstupu. Vstupy jsou míra kvality Q_m, intenzifikace I_m a diverzifikace D_m a výstupem je *selekce*.



Obrázek 2.11: FIS pro adaptivní výběr operace prohledávání v ATLBO algoritmu. Zdroj [46].

Proces fuzzifikace (převod reálných čísel na lingvistické proměnné) je založen na třech lichoběžníkových členských funkcích s lingvistickými proměnnými *low*, *medium* a *high*. Funkce pro Q_m a I_m jsou identické. Hodnoty v rozsahu 0–20 se označují jako absolutní *low*, hodnoty v rozsahu 20–40 se označují jako částečný *low*, hodnoty v rozsahu 40–60 se označují jako absolutní *medium*, hodnoty v rozsahu 60–80 se označují jako částečné *medium* a hodnoty v rozsahu 80–100 se označují jako absolutní *high*. Pro míru intenzifikace jsou rozsahy *high* a *low* obměněny, rozsah *medium* je beze změny.

Vstupy pro fuzzifikaci se spočtou následovně. Míra kvalita Q_m je normalizovaná hodnota ohodnocení zachycující kvalitu aktuálně zpracovávaného studenta, definovaná jako:

$$Q_m = \left[\frac{X_{\text{current fitness}} - \text{min fitness}}{\text{max fitness} - \text{min fitness}} \right] \cdot 100 \tag{2.9}$$

Míra intenzifikace I_m je normalizovaná hodnota Hammingovo vzdálenosti měřící vzdálenost aktuálního studenta X_{current} vůči nejlepšímu prvku X_{best}, definovaná jako:

$$I_m = \left[\frac{\left| X_{\text{best}} - X_{\text{current}} \right|}{D} \right] \cdot 100 \tag{2.10}$$

Míra diverzifikace D_m je normalizovaná hodnota Hammingovo vzdálenosti, která měří odlišnost studenta X_{current} vůči celé populaci X, definovaná jako:

$$D_m = \left[\frac{\sum_{j=1}^{\text{population size}} \left| X_j - X_{\text{current}} \right|}{D} \right] \cdot 100 \tag{2.11}$$

Jsou vytvořeny čtyři fuzzy pravidla, které jsou založeny na následujících situacích:

- Pravidlo 1: míra kvalita je *low* bez ohledu na intenzifikaci a diverzifikaci. Prohledávání je chyceno v lokálním optimu a je zapotřebí globální prohledávání.
- Pravidlo 2: míra kvality je *high* ale postrádá diverzitu. Prohledávání je chyceno v lokálním optimu kvůli nadměrnému lokálnímu prohledávání.
- Pravidlo 3: míra kvality je *high* ale postrádá konvergenci kvůli nadměrnému globálnímu prohledávání.
- Pravidlo 4: Prohledávání je blízko konvergence. Je zapotřebí provést lokální prohledávání.

Výstupní hodnota *selekce* se získá procesem defuzzifikace (převod lingvistických proměnných na reálná čísla). Selekce má definovanou lichoběžníkovou funkci s dvěma lingvistickými proměnnými *local_search* a *global_search*. Hodnoty v rozsahu 0–20 se označují jako absolutní *local_search*, hodnoty v rozsahu 20–80 se označují jako částečný *local_search* a *global_search* a hodnoty v rozsahu 80–100 se označují jako absolutní *global_search*. Hodnota selekce je nastavena na *global_search*, pokud výstup defuzzifikace je větší než 50%. V opačném případě je selekce nastavena na *local_search*.

Algorithm 4: The Adaptive TLBO for Mixed Strength t-way Test Suite Generation

Input: parameters' values p, strength of coverage t, and sub strength of coverage t_{sub}
Output: the final test suite F_s

Initialize random populations of learners X and evaluate the mean of all learners X_{mean}
Initialize the required $t-way\ interaction$ elements in the hashmap H_s based on the values of p, t, and t_{sub}
Define the member functions for the linguistic variables
Define the fuzzy rules
$X_{teacher}$ = Randomly select a learner from X
$X_{best} = X_{teacher}$
while *the hashmap* H_s *is not empty* **do**
    for $i = 1$ *to population size* **do**
        Compute Q_m, I_m and D_m
        Fuzzify based on Q_m, I_m and D_m
        Defuzzify and set Selection = crisp output
        if $Selection > 50$ **then**
            `/* Teacher Phase ... Exploration */`
            Select $X_{teacher}$ and calculate X_{mean}
            $T_F = round(1 + r(0,1))$
            $X_i^{t+1} = X_i^t + r(X_{teacher} - T_F X_{mean})$
            if $f(X_i^{t+1})$ *is better than* $f(X_i^t)$ **then**
                $X_i^t = X_i^{t+1}$
            else
                `/* Learner Phase ... Exploitation */`
                Randomly select one learner X_j^t from the population X such that $i \neq j$
                if $f(X_i^t)$ *is better than* $f(X_j^t)$ **then**
                    $X_i^{t+1} = X_i^t + r(X_j^t - X_i^t)$
                else
                    $X_i^{t+1} = X_i^t + r(X_i^t - X_j^t)$
                if $f(X_i^{t+1})$ *is better than* $f(X_i^t)$ **then**
                    $X_i^t = X_i^{t+1}$
    Get best result (X_{best}) from the current population X and put it in the final test suite list F_s
    Remove the interaction covered by X_{best} from the hashmap H_s

Obrázek 2.12: ATLBO algoritmus. Zdroj [46].

2.3.1 Ohodnocení a výsledky

Ohodnocení ATLBO algoritmu se zaměřuje na porovnání velikosti a času konstrukce sad vůči původnímu TLBO, porovnání velikosti sad vůči dalším jiným metaheuristickým algoritmům a nakonec posouzení distribuce mezi globálním a lokálním prohledáváním. Pro ATLBO byl ve všech experimentech nastaven maximální počet iterací na 100 a velikost populace na 40. Pro TLBO byla použita stejná velikost populace ale maximální počet iterací byl nastaven na 50. Každý experiment byl proveden 30-krát a nejlepší a průměrné velikosti sad jsou prezentovány.

Tabulka 2.11 prezentuje srovnání ATLBO a TLBO algoritmu. ATLBO generuje sady lepší velikosti pro tři konfigurace (CA1, CA2 a VCA1) a pro zbylé tři (CA3, VCA2, VCA3) generuje stejně velké sady jako TLBO. Pro konfigurace s menším počtem parametrů oba algoritmy konstruují sady v podobném čase. S rostoucím počtem parametrů je však ATLBO, kvůli režii fuzzy systému, výrazně pomalejší.

ID	CA and VCA	Original TLBO				ATLBO					
		Size		Time(s)		Size		Time (s)		% mean	% mean
		Best	Mean	Best	Mean	Best	Mean	Best	Mean	exploit	explore
CA1	CA(N; 2, 10^5)	117	118.7	28.80	41.10	116	118.53	23.76	28.13	79.81	20.19
CA2	CA(N; 2, 4^2 5^5)	32	34.00	9.19	10.20	28	28.95	11.55	13.83	62.18	37.82
CA3	CA(N; 2, 2^3 3^5)	13	14.77	5.12	6.15	13	14.16	6.64	8.07	32.20	67.80
VCA1	VCA(N; 2, 5^2 4^2 3^2, CA (3, 4^2 3^2))	104	107.67	40.87	47.36	103	107.90	74.18	66.11	13.20	86.80
VCA2	VCA(N; 2, 5^7, CA (3, 5^3))	125	125.00	66.63	69.10	125	125.00	125.02	131.42	18.69	81.31
VCA3	VCA(N; 2, 3^{13}, CA (3, 3^3))	27	27.26	45.94	49.60	27	27.23	64.37	69.98	23.43	76.57

Tabulka 2.11: Srovnání algoritmů TLBO a ATLBO vzhledem k velikosti a času konstrukce testovacích sad. Zdroj [46].

Tabulka 2.12 a tabulky v příloze F prezentují srovnání ATLBO algoritmu s dalšími metaheuristickými algoritmy včetně původního TLBO. Prezentované výsledky porovnávaných metod pocházejí z následujících zdrojů - PSTG [4], DPSO [44], APSO [30], CS [5], SA [15], HSS [8], ACS [36]. V tabulce 2.12 ATLBO poskytuje nejlepší výsledky pro tři ze sedmi konfigurací a pro zbývající čtyři konfigurace generuje sady velmi dobré velikosti, větší pouze o jeden testovací případ než nejlepší sada generovaná jiným algoritmem. V ostatních tabulkách si ATLBO vede podobně - ve většině případů poskytuje nejlepší, a ve zbylých velmi dobré, výsledky. ATLBO tedy konzistentně produkuje sady velmi dobrých velikostí pro konfigurace s různými t, v a p.

ID		PSTG		DPSO		HSS		Original TLBO		ATLBO			
		Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	% mean exploit	% mean explore
VCA1	$\emptyset$	75	78.96	72*	73.97	75	75.00	73	74.47	73	74.37	24.64	75.36
VCA2	CA (4, 3^4)	91	91.80	86	89.83	87	87.00	90	90.03	85*	89.23	20.36	79.64
VCA3	CA (4, 3^4)^2	91	92.21	88	90.77	90	90.00	86*	89.76	87	90.10	20.24	79.76
VCA4	CA (4, 3^5)	114	117.30	107	111.17	112	112.00	106*	111.90	107	112.13	16.44	83.56
VCA5	CA (4, 3^7)	159	162.23	152*	158.57	159	160.10	155	158.40	153	158.30	12.11	57.89
VCA6	CA (4, 3^9)	195	199.28	193	196.00	199	199.80	190	193.40	189*	193.29	11.15	88.85
VCA7	CA (4, 3^{11})	226	230.64	225*	227.50	242	243.00	226	229.51	225*	227.48	10.01	89.99

Tabulka 2.12: Srovnání ATLBO s jinými metaheuristickými algoritmy pro konfiguraci VCA(N; 3, 3^{15},). Zdroj [46].

Během běhu ATLBO algoritmu byla sledována distribuce mezi globálním a lokálním prohledáváním (viz dva nejpravější sloupce tabulek) a jaký vliv na tuto distribuci mají různé hodnoty t, v a p. Autoři došli k následujícím závěrům:

- ATLBO upřednostňuje lokální před globálním prohledáváním pro malé hodnoty p, t a v ($p \leq 6$, $t \leq 3$, $v \leq 2$)
- s rostoucím p a fixním t a v, ATLBO upřednostňuje globální před lokálním prohledáváním
- s rostoucím t a fixním p a v, ATLBO upřednostňuje globální před lokálním prohledáváním

- s rostoucím v a fixním p a t, ATLBO upřednostňuje globální před lokálním prohledáváním. V tomto případě je však rychlost růstu globálního prohledávání menší než pro rostoucí p a t

t	p	PSTG		DSPO		APSO		CS		Original TLBO		ATLBO			
		Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	% mean exploit	% mean explore
2	4	9	10.15	9	9	9	9.95	9	10.0	9	9.00	9	9.00	96.36	3.64
	5	12	13.81	11	11.53	11	12.23	11	11.80	11	11.43	11	11.33	55.14	44.86
	6	13	15.11	14	14.50	12	13.78	13	14.20	13	14.60	13	14.33	53.49	46.51
	7	15	16,94	15	15.17	15	16.62	14	15.60	15	15.07	15	15.05	52.71	47.29
	8	15	17,57	15	16.00	15	16.92	15	15.80	15	15.70	15	15.90	40.88	59.12
	9	17	19,38	15	16.43	16	18.31	16	17.20	15	16.23	15	15.03	41.46	58.54
	10	17	19,78	16	17.30	17	18.12	17	17.80	16	17.40	16	17.37	37.02	62.98
	11	17	20,16	17	17.70	-	-	18	18.60	16	17.73	16	17.67	36.77	63.23
	12	18	21,34	16	17.93	-	-	18	18.80	17	18.10	17	17.80	37.14	62.86
3	5	39	41.37	41	43.17	41	42.20	38	39.20	38	42.53	38	42.37	61.59	38.41
	6	45	46.76	33	38.30	45	46.51	43	44.20	33	38.87	33	38.43	55.86	44.14
	7	50	52.20	48	50.43	48	51.12	48	50.40	50	50.53	49	50.37	40.27	59.73
	8	54	56.76	52	53.83	50	54.86	53	54.80	52	53.17	52	53.33	38.39	61.61
	9	58	60.30	56	57.77	59	60.21	58	59.80	56	57.77	55	57.50	35.01	64.99
	10	62	63.95	59	60.87	63	64.33	62	63.60	60	60.93	59	60.73	34.09	65.91
	11	64	65.68	63	63.97	-	-	66	68.20	62	63.70	62	63.57	32.17	67.83
	12	67	68.23	65	66.83	-	-	70	71.80	65	66.70	65	66.53	29.93	70.07
4	6	133	135.31	131	134.37	129	133.98	132	134.20	130	133.63	130	134.10	50.50	49.50
	7	155	158.12	150	155.23	154	157.42	154	156.80	146	155.77	152	156.03	40.22	59.78
	8	175	176.94	171	175.60	178	179.70	173	174.80	171	175.83	171	175.50	33.85	66.15
	9	195	198.72	187	192.27	190	194.13	195	197.80	187	190.33	156	189.60	31.76	68.24
	10	210	212.71	206	219.07	214	212.21	211	212.20	205	208.80	207	208.43	27.20	72.80
	11	222	226.59	221	224.27	-	-	229	231.00	221	224.12	221	223.43	24.65	75.35
	12	244	248.97	237	239.83	-	-	253	255.80	236	239.29	235	237.83	22.41	77.59

Tabulka 2.13: Srovnání ATLBO s jinými metaheuristickými algoritmy pro konfiguraci CA (N; t, 3^p). Zdroj [46].

2.4 Srovnání zkoumaných metod

Jelikož každá metoda byla testována na jiné hardwarové konfiguraci, není možné metody porovnávat podle času konstrukce. Stejně tak není možné metody porovnávat podle efektivity, protože všechny metody nebyly otestovány nad stejným programem. Ohodnocení pokrytí kombinací bylo provedeno jen pro metodu FSAPSO, takže také nejde použít pro srovnání. Použitelnou metrikou pro objektivní porovnání zkoumaných metod je tedy pouze účinnost.

Jediné experimenty, ve kterých se všechny tři metody testují vůči stejným konfiguracím, jsou z článku o ATLBO algoritmu [46], viz tabulka 2.12 a tabulky v příloze F. Tyto tabulky obsahují konfigurace s různorodými parametry ($2 \leq t \leq 4$, $4 \leq p \leq 15$, $2 \leq v \leq 6$) a podkonfiguracemi, takže jsou dobrým zdrojem pro porovnávání metod. Výsledky ve sloupci APSO jsou vzaty přímo z FSAPSO článku [30] a sloupec PSTG reprezentuje výsledky pro metodu VS-PSTG. Z 63 případů, ATLBO poskytuje lepší výsledky ve 47 ($74,6\%$), stejné v 13 ($20,6\%$) a horší ve 3 ($4,8\%$) případech

než PSTG a z 30 případů poskytuje lepší v 16 $(53,3\%)$, stejné v 6 $(20,0\%)$ a horší v 8 $(26,6\%)$ případech než FSAPSO. ATLBO tedy bezkonkurenčně produkuje nejlepší velikosti sad.

Kromě dobrých výsledků, ATLBO bylo i nejlépe popsanou a srozumitelnou metodou a proto byla zvolena pro implementaci.

3 Implementace

Implementace knihovny se dělí na dvě části. První je implementace vybrané kombinatorické metody ATLBO. Druhou je vytvoření generátoru, který z testovacích sad vytvořených ATLBO algoritmem vygeneruje testovací třídy. Knihovna byla vyvíjena pro *Javu* verze 8 a je volně dostupná na stránce GitLabu[1].

Knihovna je realizována jako *Multi-module maven project* a je tvořena následujícími čtyřmi moduly:

- **ctgen-atlbo** - tento modul obsahuje kompletní implementaci ATLBO algoritmu.
- **ctgen-core** - součástí tohoto modulu je implementace generátoru testovacích tříd za použití předchozího modulu *ctgen-atlbo*.
- **ctgen-examples** - modul obsahující jakékoliv příklady použití ATLBO algoritmu nebo generátoru testovacích tříd.
- **ctgen-tools** - modul s nástroji pro analýzu a reportování výsledků získaných z běhu ATLBO algoritmu.

3.1 ATLBO

Implementaci ATLBO algoritmu můžeme rozložit na několik důležitých částí, kterými jsou konfigurace, populace, interakce, fuzzy systém a hraniční pravidlo. Prví čtyři části jsou podrobněji popsány dále v textu. Pro hraniční pravidlo byly použity absorbující hranice, které jsou implementovány ve třídě `AbsorbingWalls`.

Samotná implementace ATLBO algoritmu je ve třídě `Atlbo`. Normálně algoritmus iteruje dokud nejsou pokryty všechny elementy interakcí. V práci byl navíc přidán parametr *limit*, který umožňuje nastavit maximální velikost generované testovací sady pro kterou algoritmus skončí, i když ještě nepokryl všechny elementy interakcí.

Během běhu algoritmu se sbírají informace o počtu provedených operací lokálního a globálního prohledávání a ukládají se do třídy `Statistics`. Výstup algoritmu je reprezentován třídou `AtlboOutput`, která obsahuje vygenerovanou testovací sadu a výše zmíněné statistiky.

[1] `https://gitlab.com/matetot/ctgen`

3.1.1 Konfigurace

Konfigurace popisuje testovaný systém a vychází ze zápisu VSCA. Mějme VSCA (N; 2, 2^3 3^1, CA (3; 2^3)), která se skládá ze dvou částí - hlavní konfigurace MCA (N; 2, 2^3 3^1) a podkonfigurace CA (3; 2^3). V programu je VSCA reprezentována abstraktní třídou `Configuration` a od ní dědicími třídami `MainConfiguration` a `SubConfiguration`.

`Configuration` obsahuje základní informace společné pro každou konfiguraci - seznam parametrů a sílu interakce mezi parametry. Každý parametr je instancí třídy `Parameter` a může mít libovolný počet hodnot. `MainConfiguration` reprezentuje hlavní konfiguraci z VSCA a je tedy rozšířena o seznam podkonfigurací. `MainConfiguration` je vstupem pro generování populace a interakcí. `SubConfiguration` reprezentuje podkonfiguraci, která není nijak rozšířena a může se vyskytovat jen jako součást nějaké hlavní konfigurace.

3.1.2 Populace

Pro generování populací slouží rozhraní `PopulationGenerator`. Vstupem do generátoru je požadovaná velikost populace a `MainConfiguration`, která poskytuje parametry potřebné k vytvoření populace.

Populace je reprezentována třídou `Population` a její členové jsou reprezentováni třídou `PopulationMember`, která udržuje jejich aktuální pozici a ohodnocení. `Population` navíc poskytuje metody pro získání nejlepšího, nejhoršího, náhodného a průměrného člena populace. Průměrný člen s j dimenzemi v populaci o velikosti D se spočte jako:

$$X_{\text{mean}} = \left[\frac{\sum_{i=0}^{D} X_{i,0}}{D}, \frac{\sum_{i=0}^{D} X_{i,1}}{D}, ..., \frac{\sum_{i=0}^{D} X_{i,j}}{D} \right]$$

3.1.3 Interakce

K vygenerování interakcí a jejich elementů slouží rozhraní `InteractionsGenerator`. Pro reprezentaci interakcí parametrů se používají binární číslice, přičemž 1 indikuje zahrnutí parametru v interakci a 0 indikuje vyloučení parametru z interakce. Například binární zápis 1100 znamená interakci parametru p_0 a p_1. Elementem interakce se označuje jedno konkrétní nastavení hodnot nějaké interakce a je reprezentován třídou `InteractionElement`. Element je dále tvořen polem tzv. atomů `InteractionElementAtom`. Atom udržuje svoji hodnotu a zda je interagující. Pokud není interagující, hodnota atomu není nikdy použita. Příkladem elementů pro interakci 1100 může být

[6, 5, X, X] a [0, 4, X, X], kde X znamená neinteragující atom a jeho hodnota tedy není důležitá.

Vstupem do generátoru interakcí je pouze `MainConfiguration`, která obsahuje informace - sílu interakce t, počet parametrů m, popřípadě další podkonfigurace - potřebné k vygenerování všech možných elementů interakce. Algoritmus postupně generuje všechny binární čísla o délce m. Pokud vygenerované binární číslo obsahuje počet jedniček jako je hodnota t, je vybráno a pro parametry na jejichž pozicích je 1 se vygenerují všechny možné kombinace pro jejich hodnoty a vytvoří se z nich elementy interakce. Na pozice, ve kterých je 0 se vloží znak 'X', neboli neinteragující atom elementu.

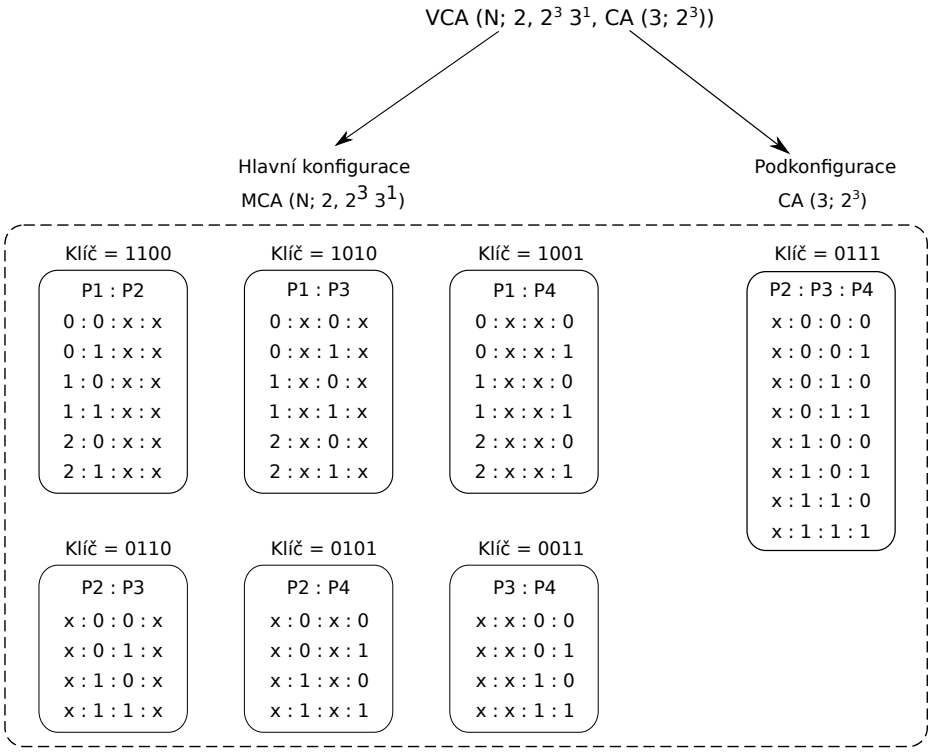
Generování elementů interakce pro podkonfigurace funguje úplně stejně, ale při kontrole, zda vygenerované binární číslo reprezentuje validní interakci nestačí kontrolovat jestli počet jedniček je rovný síle interakce dané podkonfigurace, ale navíc se musí kontrolovat jestli spolu interagují správné parametry.

Algoritmus 1: Algoritmus generování elementů interakcí.

Input: main configuration containing parameters, strength of coverage t and sub configurations S
Output: hash map H_s containing generated interaction elements

```
1  Initialize H_s = ∅
2  Let m = number of defined parameters
3  for index = 0 to 2^m − 1 do
4      Let b = index converted to binary
5      if number of '1's in b is equal to t then
6          generate elements for corresponding parameters and
7          add them into the hashmap H_s using b as hashkey
8          continue;
9      end
10     foreach SubCofiguration sub in S do
11         if (number of '1's in b is equal to sub's t) and
12         (interacting parameters match sub's parameters) then
13             generate elements for corresponding parameters and
14             add them into the hashmap H_s using b as hashkey
15         end
16     end
17 end
18 Return H_s
```

Vygenerované elementy interakcí jsou uloženy v hash mapě, kde klíčem je binární reprezentace interakce (např. 1001, 1010) a hodnotou je seznam elementů interakce (třída `InteractionElement`). Tato mapa je obalená třídou `Interactions`, která navíc poskytuje metody k manipulace s interakcemi, zejména metoda `removeCoveredInteractions`, která pro vstupního člena populace z mapy smaže jím pokryté elementy interakce.



Obrázek 3.1: Hash mapa interakcí a jejich elementů pro konfiguraci VCA (N; 2, 2^3 3^1, CA (3; 2^3)).

3.1.4 Fuzzy logika

Jak bylo řečeno u popisu ATLBO algoritmu, fuzzy systém má tři vstupy (míra kvality, intenzifikace a diverzifikace) a jeden výstup (selekce). Vstupní hodnoty jsou spočteny pomocí tříd `QualityMeasure`, `IntensificationMeasure` a `DiversificationMeasure`. Pro získání výstupní selekce slouží rozhraní `FuzzySystem`. Aktuální implementace `FuzzySystemImpl` využívá pro fuzzifikaci, aplikování fuzzy pravidel a defuzzifikaci open source knihovnu *jFuzzyLogic*[2] [12] s licencí LGPLv3.

Tato knihovna umožňuje jednoduše popsat fuzzy systém pomocí standardního jazyka *Fuzzy control language* (FCL). Jazyk poskytuje konstrukce

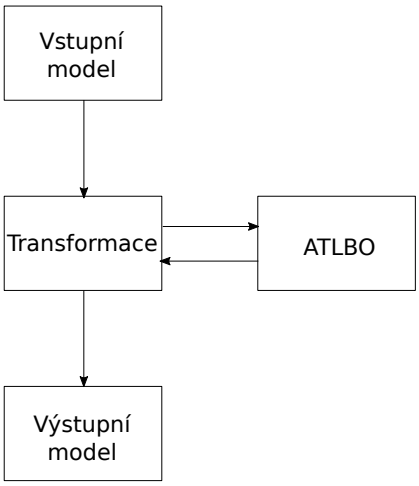
[2] `http://jfuzzylogic.sourceforge.net/html/index.html`

pro definování členských funkcí pro proces fuzzifikace a defuzzifikace, fuzzy pravidel atd. V příloze G můžete vidět FCL definici konkrétního fuzzy systému použitého v implementaci ATLBO algoritmu.

3.2 CTGen

CTGen, neboli *Combinatorial Test Generator*, je nástroj pro generování testovacích tříd za použití ATLBO algoritmu, který byl vytvořen v rámci této diplomové práce. Vstupním bodem do programu je třída `CTGen`.

CTGen pracuje v následujících třech krocích. Prvním je získání popisu testovaného programu. Druhým je transformování tohoto vstupu do výstupní struktury za použití testovacích případů získaných ATLBO algoritmem. Třetím je tuto výstupní strukturu uložit v podobě třídy obsahující jednotkové testy.



Obrázek 3.2: Ilustrace průběhu generátoru testovacích tříd.

3.2.1 Vstup

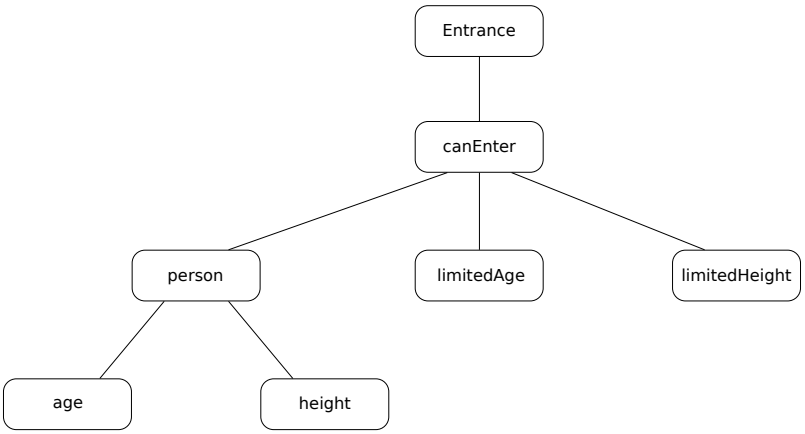
Vstupem do generátoru je soubor obsahující informace o testované třídě, metodách, parametrech a jejich konkrétních hodnotách. Tyto soubory jsou výstupem diplomové práce pana Albla, který pomocí statické analýzy generuje testovací data pro vstupní parametry testovaných metod. Soubory mohou být buď typu XML nebo JSON a pro jejich načtení slouží třídy `XmlParser` a `JsonParser`.

V příloze H můžete vidět příklad vstupního XML souboru. Pro přehlednost byl odstraněn obsah XML elementu *conditions* obsahující popis podmínek metody, protože tyto informace nejsou důležité pro tuto práci.

Relevantní informace jsou ze souboru načteny do vstupní modelu, který je tvořen následujícími třídami:

- `TestClass` - testovaná třída. Obsahuje jméno třídy, balík (package), ze kterého třída je a seznam testovaných metod.
- `TestMethod` - testovaná metoda. Obsahuje jméno, seznam jejích parametrů, sílu interakce parametrů, zda je statická a návratový datový typ.
- `TestParameter` - abstraktní třída definující společné informace pro parametry metod nebo jejich atributy. Těmi jsou jméno, datový typ a index parametru. Tuto třídu dále rozšiřují tyto třídy:
  - `TestPrimitive` - primitivní parametr/atribut, který má navíc seznam konkrétních hodnot.
  - `TestEnum` - reprezentuje parametr výčtového typu `enum`.
  - `TestObject` - reprezentuje objektový parametr, který může mít další objektové, primitivní nebo výčtové atributy.

Načtený vstupní model může být reprezentován stromem, kde kořenem je testovaná třída, v první úrovni stromu jsou její metody, v druhé úrovni jsou vstupní parametry metod a v dalších úrovních jsou atributy objektových parametrů.



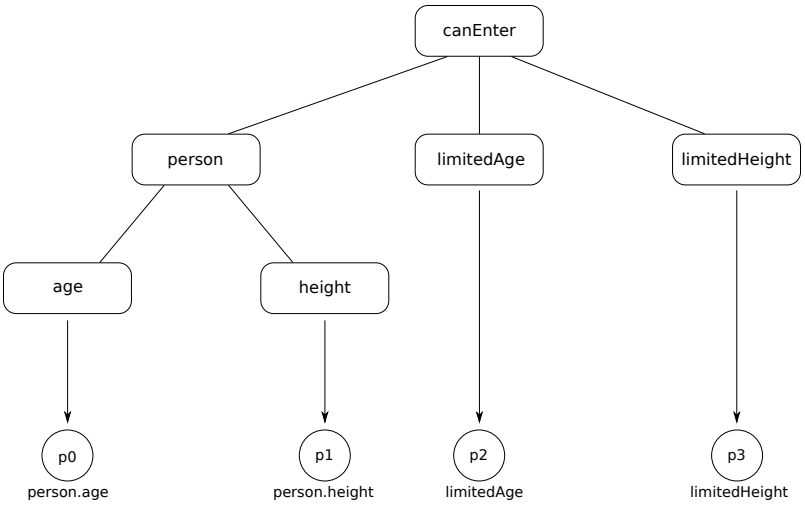
Obrázek 3.3: Strom reprezentující strukturu vstupního souboru z přílohy H.

3.2.2 Transformace

Proces transformace slouží k převodu vstupního modelu na model výstupní (viz sekce 3.2.3). Během transformace se tedy prochází vstupní model a pro každou vstupní třídu (`TestClass`) je vytvořena jedna výstupní třída, která obsahuje testy pro všechny metody vstupní třídy.

Každá metoda vstupního modelu představuje proces spuštění ATLBO algoritmu a vytvoření konkrétních jednotkových testů na základě vygenerované testovací sady. Pro spuštění ATLBO algoritmu je potřeba vytvořit vstupní konfiguraci na základě parametrů zpracovávané metody.

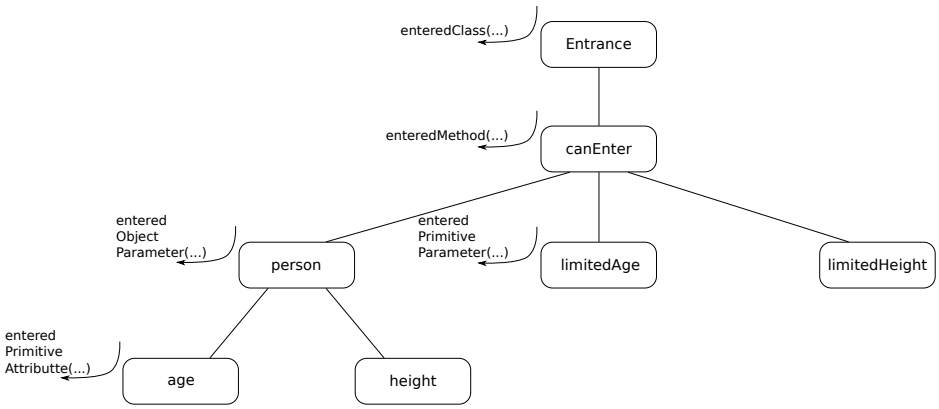
Parametr konfigurace lze vytvořit pouze z tříd typu `TestPrimitive` a `TestEnum`, kterými jsou buď primitivní a výčtové parametry testované metody nebo primitivní a výčtové atributy objektů, neboli listy stromu 3.3 reprezentující vstupní model. Po vygenerování testovací sady je potřeba přiřadit konkrétní hodnoty z testovacích případů zpět na korespondující primitivní a výčtové parametry metody nebo atributy objektů vstupního modelu. Proto je pro každý ATLBO parametr vytvořen jedinečný identifikátor, který si daný parametr uchovává. Identifikátor parametru je tvořen názvy všech uzlů oddělených tečkou po cestě stromem, která začíná uzlem parametru testované metody a končí listem, kterému daný ATLBO parametr odpovídá. Takže identifikátor nejlevějšího listu stromu 3.4 označeného jako *age* bude *person.age*, zatímco identifikátor primitivního parametru *limitedAge* bude jen *limitedAge*. Takhle vytvořený identifikátor garantuje svoji jedinečnost. Pokud by vznikly dva stejné identifikátory, znamená to, že již v kódu, ze kterého byl vytvořen vstupní model, je konflikt v pojmenování parametrů nebo atributů.



Obrázek 3.4: Ilustrace mapování parametrů vstupního modelu na parametry vstupní konfigurace ATLBO algoritmu.

Třída `AbstractTransformer` slouží k průchodu vstupním modelem, aniž by vytvářela konkrétní výstupní model. Tato třída jen definuje abstraktní metody, které se během procházení volají a odpovídají vstupům do uzlů stromu směrem dolů, popřípadě odchodům z uzlů směrem nahoru. Tímto je převedena zodpovědnost o vytvoření konkrétní podoby výstupního modelu na třídy dědící od `AbstractTransformer`. Díky tomu je vytváření výstupního modelu obecné, nezávislé na průchodu vstupním modelem a je lehké implementovat nové konstruování výstupního modelu, stačí jen implementovat několik abstraktních metod.

Do uzlů reprezentující testovanou třídu a testované metody se vstoupí pouze jednou, zatímco uzly parametrů metod a jejich potomci se prochází pro každý testovací případ z vygenerované testovací sady.

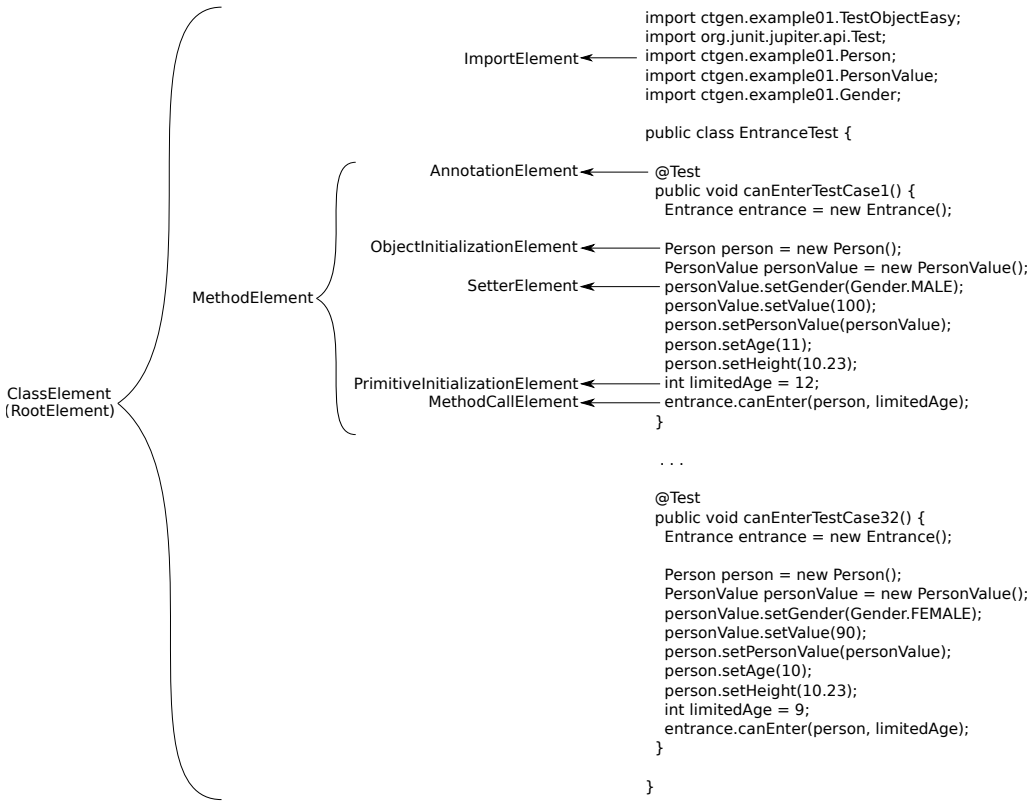


Obrázek 3.5: Volané abstraktní metody během průchodu vstupního modelu.

3.2.3 Výstup

O budování výstupního modelu se stará třída `JavaTransformer` reagováním na volání abstraktních metod, které byly popsány v sekci o transformaci (3.2.2). Výstupem procesu transformace je seznam kořenových elementů, tříd implementující rozhraní `RootElement`, které definuje metodu `getName`. Tato metoda poskytuje jméno souboru, do kterého se uloží kód reprezentovaný daným kořenovým elementem. Implementací tohoto rozhraní je třída `ClassElement`, která reprezentuje jednu výstupní třídu.

Základním stavebním prvkem výstupního modelu je rozhraní `Element`, které definuje jednu metodu - `write`. Tato metoda vrací textovou reprezentaci elementu. Elementem může být cokoliv, nějaký logický celek, který obsahuje další elementy (např. element `MethodElement`) nebo konkrétní konstrukce programovacího jazyka (např. element `ObjectInitializationElement`), viz obrázek 3.6. Kořenový element (`RootElement`) také implementuje rozhraní `Element` a volání metody `write` nad kořenem tedy vrátí textovou reprezentaci celé výstupní třídy.



Obrázek 3.6: Příklad vygenerované výstupní třídy s korespondující elementy výstupního modelu.

Vygenerovaný kód může být libovolně naformátován pomocí rozhraní `CodeFormatter`. V knihovně je momentálně ve třídě `GoogleJavaFormatter` implementované formátování s využitím Google knihovny *google-java-format*[3], která formátuje *Java* kód v souladu se standardem *Google Java Style*[4].

Kromě uložení vygenerovaného kódu do souboru je také možné vygenerovaný kód testů ihned spustit. Třída `CodeRunner` poskytuje in-memory překladač, který zkompiluje libovolný *Java* kód aniž by na disku vznikly *.class* soubory. Z přeložené třídy lze poté spustit libovolnou metodu, stačí jen znát její jméno.

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

[4] `https://google.github.io/styleguide/javaguide.html`

4 Testování

Ověření správné funkčnosti a ohodnocení vytvořené knihovny se dělí na tři částí. První je jednotkové testování implementace ATLBO algoritmu. Druhou je porovnání velikostí generovaných sad vlastní implementací ATLBO algoritmu vůči výsledkům prezentovaných ve zdrojovém článku [46]. Třetí částí je zhodnocení efektivity testů generovaných vytvořeným generátorem.

4.1 Jednotkové testování

Pro jednotkové testování byla využita knihovna *jUnit*[1] verze 5.1.0. Jednotkově byly testovány jednotlivé důležité součásti ATLBO algoritmu, kterými jsou populace, interakce, fuzzy systém, hraniční pravidlo a vektorové operace.

U populace bylo důležité otestovat správnost generování jejích členů a získání nejlepšího a průměrného prvku populace. Pro interakce bylo nejdůležitější otestovat generování interakcí a jejich elementů, výpočet ohodnocení libovolného člena populace na základě toho kolik elementů interakce pokrývá a odstraňování pokrytých elementů z mapy interakcí. Testování fuzzy systému je zaměřeno na to, zda se pro určité vstupní hodnoty aktivuje správné fuzzy pravidlo a tím pádem se správně rozhodne mezi globálním a lokálním prohledáváním a dále se testuje korektnost výpočtu tří vstupů systému - míra kvality, intenzifikace a diverzifikace.

4.2 Porovnání účinnosti ATLBO implementací

Stejně jako ve zdrojovém článku [46] je pro testování nastavena velikost populace na 40, počet iterací na 100 a pro každý experiment je ATLBO algoritmus spuštěn 30-krát a ve výsledcích se prezentují nejlepší a průměrné velikosti vygenerovaných testovacích sad. Dále se u experimentů sleduje průměrný počet provedených operací lokálního a globálního prohledávání a ve výsledcích se prezentuje poměr mezi operacemi v procentech.

Modul knihovny **ctgen-tools** obsahuje nástroje, které usnadňují mnohonásobné spuštění ATLBO algoritmu a uložení získaných výsledků. Dále

[1] `https://junit.org/junit5/`

obsahuje nástroje pro analýzu takto získaných výsledků a nástroje pro reportování analyzovaných výsledků.

ID	CA and VCA	ATLBO				Vlastní ATLBO			
		Size		% mean	% mean	Size		% mean	% mean
		Best	Mean	exploit	explore	Best	Mean	exploit	explore
CA1	CA(N; 2, 10^5)	116	118.53	79.81	20.19	155	163.59	31.51	68.49
CA2	CA(N; 2, 4^2 5^5)	28	28.95	62.18	37.82	40	43.68	19.17	80.83
CA3	CA(N; 2, 2^3 3^5)	13	14.16	32.20	67.80	13	15.19	10.23	89.77
VCA1	VCA(N; 2, 5^2 4^2 3^2, CA (3, 4^2 3^2))	103	107.90	13.20	86.80	51	55.40	12.83	87.17
VCA2	VCA(N; 2, 5^7, CA (3, 5^3))	125	125.00	18.69	81.31	125	125.56	46.45	53.55
VCA3	VCA(N; 2, 3^{13}, CA (3, 3^3))	27	27.23	23.43	76.57	28	29.86	14.30	85.70

Tabulka 4.1: Srovnání pro různé konfigurace.

Tabulky 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7 prezentují srovnání výsledků vlastní implementace s výsledky z článku [46]. Výsledkům z článku se dále bude říkat porovnávané výsledky.

Ze získaných výsledků je patrné, že vlastní implementace algoritmu generuje sady větších velikostí ve skoro všech případech. Konkrétně z 76 testovaných konfigurací, vlastní implementace v 61 případech generuje sady větší, v 10 případech stejné a v 5 případech menší velikosti.

Za pozornost stojí procentuální poměr operacemi globálního a lokálního prohledávání. Na rozdíl od porovnávaných výsledků, ve kterých někdy převažuje globální nad lokálním prohledáváním, někdy převažuje lokální a někdy je poměr prohledávání přibližně vyrovnaný, u vlastní implementace vždy silně převažuje globální nad lokálním prohledáváním. Tato silná převaha globálního prohledávání může vysvětlit rozdíl ve velikosti generovaných sad, protože je prohledávání prostoru vedeno méně vhodným směrem.

Ačkoliv vlastní implementace neprodukuje testovací sady tak dobrých velikostí, jakých lze údajně dosáhnout podle výsledků prezentovaných v [46], alespoň algoritmus doběhne pro libovolnou testovanou konfiguraci.

ID		ATLBO				Vlastní ATLBO			
		Best	Mean	% mean exploit	% mean explore	Best	Mean	% mean exploit	% mean explore
VCA1	∅	73	74.37	24.64	75.36	97	100.10	07,26	92,74
VCA2	CA (4, 3^4)	85	89.23	20.36	79.64	104	110.30	07,50	92,50
VCA3	CA (4, 3^4)^2	87	90.10	20.24	79.76	112	118.81	06,09	93,91
VCA4	CA (4, 3^5)	107	112.13	16.44	83.56	122	129.43	05,52	94,48
VCA5	CA (4, 3^7)	153	158.30	12.11	87.89	149	154.57	04,68	95,32
VCA6	CA (4, 3^9)	189	193.29	11.15	88.85	229	236.24	03,61	96,39
VCA7	CA (4, 3^{11})	225	227.48	10.01	89.99	278	286.19	03,58	96,42

Tabulka 4.2: Srovnání pro konfiguraci VCA(N; 3, 3^{15},).

t	v	ATLBO				Vlastní ATLBO			
		Best	Mean	% mean exploit	% mean explore	Best	Mean	% mean exploit	% mean explore
2	2	7	7.00	50.87	49.13	6	7.57	14.85	85.15
	3	15	15.07	51.60	48.40	15	16.90	15.98	84.02
	4	23	25.17	57.74	42.26	28	30.19	19.53	80.47
	5	34	35.47	63.82	36.18	45	48.15	19.84	80.16
3	2	15	15.12	48.42	51.58	12	15.27	11.37	88.63
	3	49	50.29	39.60	60.40	54	57.54	13.50	86.50
	4	111	115.67	43.16	56.84	136	141.17	17.84	82.16
	5	216	219.40	44.77	55.23	276	283.28	18.48	81.52
4	2	31	33.68	46.04	53.96	27	30.68	11.18	88.82
	3	150	155.24	39.42	60.58	168	175.19	15.70	84.30
	4	478	484.69	39.90	60.10	558	571.06	19.71	80.29
	5	1166	1173.45	40.14	59.86	1425	1441.24	19.13	80.87

Tabulka 4.3: Srovnání pro konfiguraci CA(N; t, v^7).

ID		ATLBO				Vlastní ATLBO			
		Best	Mean	% mean exploit	% mean explore	Best	Mean	% mean exploit	% mean explore
VCA1	$\emptyset$	18	19.30	31.30	68.70	22	24.67	10.79	89.21
VCA2	CA(3, 3^3)	27	27.00	22.26	77.74	27	30.54	12.31	87.69
VCA3	CA (3, 3^3)^2	27	27.53	21.46	78.54	29	33.14	08.81	91.19
VCA4	CA (3, 3^3)^3	27	27.43	22.00	78.00	32	35.43	07.81	92.19
VCA5	CA (3, 3^4)	27	27.00	22.26	77.74	31	36.79	07.79	92.21
VCA6	CA (3, 3^5)	38	40.60	16.25	83.75	40	44.25	06.29	93.71
VCA7	CA (3, 3^6)	43	43.67	18.05	81.95	46	51.30	05.67	94.33
VCA8	CA (3, 3^7)	47	49.83	17.96	82.04	56	58.35	05.32	94.68
VCA9	CA (4, 3^4)	81	81.03	7.44	92.56	81	81.59	37.85	62.15
VCA10	CA (4, 3^5)	87	96.90	7.26	92.74	97	104.40	21.47	78.53
VCA11	CA (4, 3^7)	152	156.33	10.74	89.26	167	174.70	07.29	92.71

Tabulka 4.4: Table 7 Srovnání pro konfiguraci VCA(N; 2, 3^{15},).

t	p	ATLBO				Vlastní ATLBO			
		Best	Mean	% mean exploit	% mean explore	Best	Mean	% mean exploit	% mean explore
2	4	9	9.00	96.36	3.64	9	11.52	26.12	73.88
	5	11	11.33	55.14	44.86	11	13.82	21.63	78.37
	6	13	14.33	53.49	46.51	14	15.32	18.37	81.63
	7	15	15.05	52.71	47.29	15	17.01	16.29	83.71
	8	15	15.90	40.88	59.12	17	18.28	14.27	85.73
	9	15	15.03	41.46	58.54	18	19.36	13.89	86.11
	10	16	17.37	37.02	62.98	19	20.23	13.75	86.25
	11	16	17.67	36.77	63.23	20	21.39	12.73	87.27
	12	17	17.80	37.14	62.86	21	22.32	12.59	87.41
3	5	38	42.37	61.59	38.41	39	42.17	23.09	76.91
	6	33	38.43	55.86	44.14	46	49.96	16.26	83.74
	7	49	50.37	40.27	59.73	55	57.50	13.81	86.19
	8	52	53.33	38.39	61.61	61	64.20	11.86	88.14
	9	55	57.50	35.01	64.99	67	70.44	10.78	89.22
	10	59	60.73	34.09	65.91	73	75.91	09.91	90.09
	11	62	63.57	32.17	67.83	79	81.68	09.12	90.88
	12	65	66.53	29.93	70.07	83	86.35	08.51	91.49
4	6	130	134.10	50.50	49.50	137	143.42	20.75	79.25
	7	152	156.03	40.22	59.78	169	175.12	15.64	84.36
	8	171	175.50	33.85	66.15	199	205.36	12.60	87.40
	9	156	189.60	31.76	68.24	226	233.72	10.61	89.39
	10	207	208.43	27.20	72.80	254	260.36	09.39	90.61
	11	221	223.43	24.65	75.35	279	285.63	08.34	91.66
	12	235	237.83	22.41	77.59	302	309.16	07.61	92.39

Tabulka 4.5: Srovnání pro konfiguraci CA(N; t, 3^p).

ID		ATLBO				Vlastní ATLBO			
		Best	Mean	% mean exploit	% mean explore	Best	Mean	% mean exploit	% mean explore
VCA1	$\emptyset$	39	41.63	43.47	56.53	49	52.73	16.66	83.34
VCA2	CA (3, 4^3)	64	64.03	31.11	68.89	66	68.81	18.45	81.55
VCA3	CA (3, 4^3 5^2)	122	124.5	18.31	81.69	135	140.81	07.75	92.25
VCA4	CA (3, 4^3) CA (3, 5^3)	125	125.00	15.88	84.12	125	126.34	30.30	69.70
VCA5	CA (3, 4^3 5^3 6^1)	203	208.68	14.42	85.58	244	250.83	06.28	93.72
VCA6	CA (3, 4^3) CA (4, 5^3 6^1)	750	750.00	12.70	87.30	750	750.03	61.65	38.35
VCA7	CA (4, 4^3 5^2)	451	459.10	6.52	93.48	480	494.79	31.68	68.32

Tabulka 4.6: Srovnání pro konfiguraci VCA(N; 2, 4^3 5^3 6^2,).

t	v	ATLBO				Vlastní ATLBO			
		Best	Mean	% mean exploit	% mean explore	Best	Mean	% mean exploit	% mean explore
2	4	28	28.69	42.42	57.58	35	36.70	15.55	84.45
	5	42	43.53	46.92	53.08	56	59.11	15.60	84.40
	6	58	59.33	50.27	49.73	83	86.50	15.63	84.37
3	4	140	142.80	30.77	69.23	180	186.57	12.91	87.09
	5	272	275.23	31.04	68.96	373	380.41	13.00	87.00
	6	466	469.90	31.53	68.47	658	669.47	12.88	87.12
4	4	661	664.06	25.68	74.32	840	850.27	11.87	88.13
	5	1619	1620.91	22.32	77.68	2158	2175.06	11.42	88.58
	6	3338	3342.10	21.13	78.87	4555	4573.60	11.24	88.76

Tabulka 4.7: Srovnání pro konfiguraci CA(N; t, v^{10}).

4.3 Ohodnocení efektivity

Zatímco předchozí dvě části testování se zaměřují jen na ATLBO algoritmus, tato část využívá vytvořený generátor testovacích tříd pro ověření efektivity vlastní implementace ATLBO algoritmu.

Pro účely otestování efektivity byl vytvořen program, pro který se pomocí knihovny, popsané v 3. kapitole, vygenerují testy a posléze se sleduje, jak dobře vygenerované testy pokrývají kód testovaného programu. Pro měření pokrytí kódu byli použity nástroje poskytované vývojovým prostředím *IntelliJ IDEA*.

Vytvořený testovaný program má 12 vstupních parametrů, viz tabulka 4.8. Jeden vstupní parametr má sedm hodnot, jeden vstupní parametr má šest hodnot, osm vstupních parametrů má dvě hodnoty a dva vstupní parametry mají tři hodnoty.

No.	Parameter	Hodnoty
1	degree	[no_degree, primary, secondary,diploma, bachelor, master, doctor]
2	children	[none, one, two, three, four, more_than_four]
3	read	[true, false]
4	write	[true, false]
5	speak	[true, false]
6	understand	[true, false]
7	new_graduate	[true, false]
8	experience	[true, false]
9	english	[true, false]
10	disabilitty	[true, false]
11	marital status	[single, married, widow]
12	resident	[local, outsider, foreigner]

Tabulka 4.8: Vstupní parametry testovaného programu.

Takovou vstupní konfiguraci lze zapsat jako MCA notaci MCA(N; t, 7^1 6^1 2^8 3^2). Pro kompletní pokrytí všech možných kombinací by bylo potřeba 96 768 ($7\times6\times2\times2\times2\times2\times2\times2\times2\times2\times2\times3\times3$) testovacích případů. Tabulka 4.9 prezentuje velikosti testovacích sad vygenerovaných ATLBO algoritmem pro různá t.

Síla interakce (t)	Velikost sady
2	44
3	167
4	522
5	1412
6	3348

Tabulka 4.9: Velikosti sad notace MCA(N; t, 7^1 6^1 2^8 3^2) pro různá t .

Sada pro $t = 2$ o velikosti 44 testů pokrývá 76% testovaného programu. Sada o velikosti 167 testů pro $t = 3$ pokrývá 92% kódu. Nárůst v pokrytí kódu pro sadu $t = 4$ je velmi malý, 522 testů pokrývá 94% kódu, jen o 2% více než sada pro $t = 3$. Sada o velikosti 1412 pro $t = 5$ dosahuje 100% pokrytí kódu. Samozřejmě sada pro $t = 6$ také pokrývá 100% kódu.

Závěr

V rámci této práce byla úspěšně vytvořena knihovna, která z popisu testované třídy dokáže vygenerovat konkrétní jednotkové testy v jazyce *Java*. Pro vytvoření testovací sady, pro kterou se generují konkrétní jednotkové testy byla implementována kombinatorická metoda ATLBO, která byla jednou ze tří podrobněji zkoumaných metod. Tato metoda byla vybrána pro její dobré prezentované výsledky, novost a srozumitelnost.

Její implementace se dá považovat za napůl úspěšnou. Algoritmus vždy úspěšně doběhne pro libovolnou konfiguraci, ale neprodukuje sady tak dobrých velikostí, jakých lze údajně dosáhnout podle článku, dle kterého byla metoda implementována.

Jelikož ani po dlouhodobém zkoumání vlastní implementace a hledání chyby, které zahrnovalo i komunikaci s autory články, díky které bylo opraveno několik nejasností a chyb z článku, se nepodařilo dojít ke stejným výsledkům, do budoucna bude proto nejlepší najít a implementovat jinou kvalitní kombinatorickou metodu. Díky oddělenosti samotného generátoru testovacích tříd a ATLBO algoritmu je poměrně snadné do knihovny zabudovat novou metodu.

Kromě porovnání velikostí testovacích sad byla knihovna také otestována vzhledem k efektivnosti generovaných jednotkových testů na netriviálním programu. Efektivita byla měřena v procentech pokrytého kódu testovaného programu a toto testování ukázalo, že vytvořené testy jsou schopné dobře pokrývat kód a také, že většina kódu je pokryta již pro sílu interakce 2 a následné zvětšování síly interakce poskytuje relativně malý narůst v pokrytí kódu.

Všechny body zadání této diplomové práce byly splněny a vytvořená knihovna může být použita pro generování jednotkových testů. Aktuální podobu knihovny nepovažuji za výslednou a věřím, že dojde k jejímu dalšímu vývoji.

Literatura

[1] Afzal, W. – Torkar, R. – Feldt, R. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*. June 2009, 51, 6, s. 957 – 976. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2008.12.005. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S0950584908001833`.

[2] Ahmed, B. S. – Zamli, K. Z. A variable strength interaction test suites generation strategy using Particle Swarm Optimization. *Journal of Systems and Software*. December 2011, 84, 12, s. 2171 – 2185. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2011.06.004. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S0164121211001464`.

[3] Ahmed, B. S. – Zamli, K. Z. – Lim, C. P. Application of Particle Swarm Optimization to uniform and variable strength covering array construction. *Applied Soft Computing*. April 2012, 12, 4, s. 1330 – 1347. ISSN 568-4946. doi: https://doi.org/10.1016/j.asoc.2011.11.029. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S1568494611004716`.

[4] Ahmed, B. S. – Zamli, K. Z. – Lim, C. P. Application of Particle Swarm Optimization to uniform and variable strength covering array construction. *Applied Soft Computing*. February 2012, 12, 4, s. 330 – 1347. ISSN 1568-4946. doi: https://doi.org/10.1016/j.asoc.2011.11.029. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S1568494611004716`.

[5] Ahmed, B. S. – Abdulsamad, T. S. – Potrus, M. Y. Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the Cuckoo Search algorithm. *Information and Software Technology*. 2015, 66, s. 13 – 29. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2015.05.005. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S0950584915001020`.

[6] Ahmed, B. S. – Abdulsamad, T. S. – Potrus, M. Y. Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the Cuckoo Search algorithm. *Information and Software Technology*. 2015, 66, s. 13 – 29. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2015.05.005. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S0950584915001020`.

[7] Ahmed, B. – Zamli, K. – Lim, C. Constructing a t-way interaction test suite using the Particle Swarm Optimization approach. *International*

Journal of Innovative Computing, Information and Control. 2012, 8, 1 A, s. 431–451. Dostupné z: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84856956194&partnerID=40&md5=c3e60a2ebf206edf45250ebe62fe1130.

[8] Alsewari, A. R. A. – Zamli, K. Z. Design and implementation of a harmony-search-based variable-strength t-way testing strategy with constraints support. *Information and Software Technology*. 2012, 54, 6, s. 553 – 568. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2012.01.002. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0950584912000134.

[9] K. C. Tai. In-parameter-order: a test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*, s. 254–261, Nov 1998. doi: 10.1109/HASE.1998.731623.

[10] Calvagna, A. – Gargantini, A. IPO-s: Incremental Generation of Combinatorial Interaction Test Data Based on Symmetries of Covering Arrays. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, s. 10–18, April 2009. doi: 10.1109/ICSTW.2009.7.

[11] Chen, X. et al. Variable Strength Interaction Testing with an Ant Colony System Approach. In *2009 16th Asia-Pacific Software Engineering Conference*, s. 160–167, Dec 2009. doi: 10.1109/APSEC.2009.18.

[12] Cingolani, P. – Alcala-Fdez, J. jFuzzyLogic: a robust and flexible Fuzzy-Logic inference system language implementation. In *Fuzzy Systems (FUZZ-IEEE), 2012 IEEE International Conference on*, s. 1–8. IEEE, 2012.

[13] Clerc, M. – Kennedy, J. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*. February 2002, 6, 1, s. 58 – 73. ISSN 1089-778X. doi: 10.1109/4235.985692. Dostupné z: https://doi.org/10.1109/4235.985692.

[14] Cohen, D. M. et al. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*. July 1997, 23, 7, s. 437–444. ISSN 0098-5589. doi: 10.1109/32.605761.

[15] Cohen, M. B. – Colbourn, C. J. – Ling, A. C. H. Augmenting simulated annealing to build interaction test suites. *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* November 2003, s. 394–405. ISSN 1071-9458. doi: 10.1109/ISSRE.2003.1251061. Dostupné z: https://doi.org/10.1109/ISSRE.2003.1251061.

[16] Cohen, M. B. et al. A variable strength interaction testing of components. In *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, s. 413–418, Nov 2003. doi: 10.1109/CMPSAC.2003.1245373.

[17] Cohen, M. B. – Dwyer, M. B. – Shi, J. Interaction Testing of Highly-configurable Systems in the Presence of Constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, s. 129–139, New York, NY, USA, 2007. ACM. doi: 10.1145/1273463.1273482. Dostupné z: http://doi.acm.org/10.1145/1273463.1273482. ISBN 978-1-59593-734-6.

[18] Damm, L.-O. *Early and Cost-Effective Software Fault Detection : Measurement and Implementation in an Industrial Setting*. PhD thesis, Blekinge Institute of Technology, 2007.

[19] Forbes, M. et al. Refining the In-Parameter-Order Strategy for Constructing Covering Arrays. *J Res Natl Inst Stand Technol*. Oct 2008, 113, 5, s. 287–297. ISSN 1044-677X. doi: 10.6028/jres.113.022. Dostupné z: https://www.ncbi.nlm.nih.gov/pubmed/27096128.

[20] Hartman, A. *Software and Hardware Testing Using Combinatorial Covering Suites*, s. 237–266. Springer US, Boston, MA, 2005. doi: 10.1007/0-387-25036-0_10. Dostupné z: https://doi.org/10.1007/0-387-25036-0_10. ISBN 978-0-387-25036-6.

[21] Hartman, A. – Raskin, L. Problems and algorithms for covering arrays. *Discrete Mathematics*. 2004, 284, 1, s. 149 – 156. ISSN 0012-365X. doi: https://doi.org/10.1016/j.disc.2003.11.029. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0012365X0400130X. Special Issue in Honour of Curt Lindner on His 65th Birthday.

[22] Jarboui, B. et al. Combinatorial particle swarm optimization (CPSO) for partitional clustering problem. *Applied Mathematics and Computationy*. September 2007, 192, 2, s. 337 – 345. ISSN 0096-3003. doi: https://doi.org/10.1016/j.amc.2007.03.010. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0096300307003347.

[23] Kennedy, J. – Eberhart, R. Particle swarm optimizationn. *Proceedings of ICNN'95 - International Conference on Neural Networks*. November 1995, 4, s. 1942 – 1948. doi: 10.1109/ICNN.1995.488968. Dostupné z: https://doi.org/10.1109/ICNN.1995.488968.

[24] Kuhn, D. R. – Wallace, D. R. – Gallo, A. M. Software fault interactions and implications for software testing. *IEEE Transactions on*

Software Engineering. June 2004, 30, 6, s. 418 – 421. ISSN 0098-5589. doi: https://doi.org/10.1109/TSE.2004.24. Dostupné z: https://doi.org/10.1109/TSE.2004.24.

[25] Kuhn, D. – Kacker, R. – Lei, Y. *Introduction to Combinatorial Testing*. CRC Press, 2016. Dostupné z: https://books.google.cz/books?id=eUbSBQAAQBAJ. ISBN 9781466552302.

[26] Lei, Y. et al. IPOG: A General Strategy for T-Way Software Testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, s. 549–556, March 2007. doi: 10.1109/ECBS.2007.47.

[27] Lei, Y. et al. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*. 2008, 18, 3, s. 125–148. doi: 10.1002/stvr.381. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.381.

[28] Lessmann, S. – Caserta, M. – Arango, I. M. Tuning metaheuristics: A data mining based approach for particle swarm optimization. *Expert Systems with Applications*. September 2011, 38, 10, s. 12826 – 12838. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2011.04.075. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0957417411005914.

[29] Liang, J. J. et al. Comprehensive learning particle swarm optimizer for global optimization of multimodal functions. *IEEE Transactions on Evolutionary Computation*. April 2006, 10, 3, s. 281 – 295. ISSN 1089-778X. doi: 10.1109/TEVC.2005.857610. Dostupné z: https://doi.org/10.1109/TEVC.2005.857610.

[30] Mahmoud, T. – Ahmed, B. S. An efficient strategy for covering array construction with fuzzy logic-based adaptive swarm optimization for software testing use. *Expert Systems with Applications*. December 2015, 42, 22, s. 8753 – 8765. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2015.07.029. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0957417415004893.

[31] Nie, C. – Leung, H. A Survey of Combinatorial Testing. *ACM Comput. Surv.* February 2011, 43, 2, s. 11:1–11:29. ISSN 0360-0300. doi: 10.1145/1883612.1883618. Dostupné z: http://doi.acm.org/10.1145/1883612.1883618.

[32] Nie, C. et al. Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures. *Information and*

Software Technology. 2015, 62, s. 198 – 213. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2015.02.008. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0950584915000440.

[33] Nurmela, K. J. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics*. 2004, 138, 1, s. 143 – 152. ISSN 0166-218X. doi: https://doi.org/10.1016/S0166-218X(03)00291-9. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0166218X03002919. Optimal Discrete Structures and Algorithms.

[34] Rao, R. – Savsani, V. – Vakharia, D. Teaching–learning-based optimization: A novel method for constrained mechanical design optimization problems. *Computer-Aided Design*. 2011, 43, 3, s. 303 – 315. ISSN 0010-4485. doi: https://doi.org/10.1016/j.cad.2010.12.015. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0010448510002484.

[35] Schroeder, P. J. – Bolaki, P. – Gopu, V. Comparing the fault detection effectiveness of n-way and random test suites. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.*, s. 49–59, Aug 2004. doi: 10.1109/ISESE.2004.1334893.

[36] Shiba, T. – Tsuchiya, T. – Kikuno, T. Using artificial life techniques to generate test cases for combinatorial testing. *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.* September 2004, s. 72–77 vol.1. ISSN 0730-3157. doi: 10.1109/CMPSAC.2004.1342808. Dostupné z: https://doi.org/10.1109/CMPSAC.2004.1342808.

[37] Shiba, T. – Tsuchiya, T. – Kikuno, T. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, s. 72–77 vol.1, Sep. 2004. doi: 10.1109/CMPSAC.2004.1342808.

[38] Sidhartha Panda, N. P. P. Comparison of particle swarm optimization and genetic algorithm for FACTS-based controller design. *Applied Soft Computing*. September 2008, 8, 4, s. 1418 – 1427. ISSN 1568-4946. doi: https://doi.org/10.1016/j.asoc.2007.10.009. Dostupné z: http://www.sciencedirect.com/science/article/pii/S1568494607001330.

[39] Smith, L. B. H. W. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*. June 2009, 14, 3, s. 341 – 369. ISSN 1573-7616. doi: 0.1007/s10664-008-9083-7. Dostupné z: https://doi.org/10.1007/s10664-008-9083-7.

[40] Tassey, G. The economic impacts of inadequate infrastructure for software testing. 2002.

[41] Wang, H. – Geng, Q. – Qiao, Z. Parameter tuning of particle swarm optimization by using Taguchi method and its application to motor design. *2014 4th IEEE International Conference on Information Science and Technology*. April 2014, s. 722 – 726. ISSN 2164-4357. doi: 10.1109/ICIST.2014.69205795. Dostupné z: https://doi.org/10.1109/ICIST.2014.6920579.

[42] Williams, A. W. – Probert, R. L. A practical strategy for testing pair-wise coverage of network interfaces. *Proceedings of ISSRE '96: 7th International Symposium on Software Reliability Engineering*. October 2011, s. 246–254. ISSN 1071-9658. doi: 10.1109/ISSRE.1996.558835. Dostupné z: https://doi.org/10.1109/ISSRE.1996.558835.

[43] Windisch, A. – Wappler, S. – Wegener, J. *Applying Particle Swarm Optimization to Software Testing*. ACM, 2007. ISBN 978-1-59593-697-4.

[44] Wu, H. et al. A Discrete Particle Swarm Optimization for Covering Array Generation. *IEEE Transactions on Evolutionary Computation*. August 2015, 19, 4, s. 575 – 591. ISSN 1089-778X. doi: 10.1109/TEVC.2014.2362532. Dostupné z: https://doi.org/10.1109/TEVC.2014.2362532.

[45] Yilmaz, C. – Cohen, M. B. – Porter, A. Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. *SIGSOFT Softw. Eng. Notes*. July 2004, 29, 4, s. 45–54. ISSN 0163-5948. doi: 10.1145/1013886.1007519. Dostupné z: http://doi.acm.org/10.1145/1013886.1007519.

[46] Zamli, K. Z. et al. Fuzzy adaptive teaching learning-based optimization strategy for the problem of generating mixed strength t-way test suites. *Engineering Applications of Artificial Intelligence*. 2017, 59, s. 35 – 50. ISSN 0952-1976. doi: https://doi.org/10.1016/j.engappai.2016.12.014. Dostupné z: http://www.sciencedirect.com/science/article/pii/S095219761630241X.

Seznam zkratek

ACA Ant colony algorithm. 15

ACS Ant colony system. 35

AETG Automatic efficient test generator. 14

APSO Adaptive particle swarm optimization. 35, 36

ATLBO Adaptive Teaching Learning-Based Optimization. 30–32, 34–38, 41, 42, 44, 63, 64

CA Covering array. 11–14

CS Cuckoo search. 35

CTGen Combinatorial Test Generator. 42

CTS Combinatorial Test Services. 13

DPSO Discrete particle swarm optimization. 35

FCL Fuzzy control language. 41, 42

FIS Fuzzy inference system. 22–25, 30, 32, 63, 64

FSAPSO Fuzzy Self Adaptive Particle Swarm Optimization. 22, 25–28, 36, 37

GA Genetic algorithm. 15

GF Great flood. 27

HC Hill climbing. 27

HSS Harmony seach strategy. 35

IPO In-parameter-order. 14

JSON JavaScript Object Notation. 42

MCA Mixed level covering array. 12

PSO Particle Swarm Optimization. 15–17, 22, 27, 30

PSTG Particle swarm test generator. 35–37

SA Simulated annealing. 15, 27, 35

TLBO Teaching Learning-Based Optimization. 30, 34, 35, 64

TS Tabu search. 15

VS-PSTG Variable-strength Particle Swarm Test Generator. 16, 18, 19, 22, 36

VSCA Variable-strength covering array. 12, 39

XML Extensible Markup Language. 42

Seznam obrázků

1.1 Dva příklady pro CA (5, 2, 2^4). 12
1.2 Příklad pro MCA (6, 2, 3^1 2^4). 12
1.3 Příklad pro VSCA (12, 2, 3^1 2^4, CA(12, 3, 2^4)). 13
1.4 Znázornění one-parameter-at-a-time algoritmů. 14

2.1 Algoritmus VS-PSTG. Zdroj [2]. 19
2.2 Parametry flexu a jejich možné hodnoty. Zdroj [2]. 20
2.3 Členské funkce (dvě vstupní, jedna výstupní) FIS přizpůsobující parametr Δw. 24
2.4 Členské funkce (dvě vstupní, jedna výstupní) FIS přizpůsobující parametr c_1 a c_2. 25
2.5 Hodnota parametru w v průběhu algoritmu. Zdroj [30]. . . . 26
2.6 Hodnota parametru c_1 a c_2 v průběhu algoritmu. Zdroj [30]. 26
2.7 Porovnání míry pokrytí pro CA (N; 4, 3^{13}). Zdroj [30]. . . . 28
2.8 Porovnání míry pokrytí pro MCA (N; 4, 2^{10}, 3^3, 4^2, 5^1). Zdroj [30]. 28
2.9 Reakce testovací sady pro $t = 2$ na mutace. Zdroj [30]. . . . 30
2.10 Původní TLBO algoritmus. Zdroj [46]. 31
2.11 FIS pro adaptivní výběr operace prohledávání v ATLBO algoritmu. Zdroj [46]. 32
2.12 ATLBO algoritmus. Zdroj [46]. 34

3.1 Hash mapa interakcí a jejich elementů pro konfiguraci VCA (N; 2, 2^3 3^1, CA (3; 2^3)). 41
3.2 Ilustrace průběhu generátoru testovacích tříd. 42
3.3 Strom reprezentující strukturu vstupního souboru z přílohy H. 43
3.4 Ilustrace mapování parametrů vstupního modelu na parametry vstupní konfigurace ATLBO algoritmu. 45
3.5 Volané abstraktní metody během průchodu vstupního modelu. 45
3.6 Příklad vygenerované výstupní třídy s korespondující elementy výstupního modelu. 46

A.1 Ilustrace průběhu VS-PSTG metody. Zdroj [2]. 66

C.1 Ilustrace FSAPSO algoritmu. Zdroj [30]. 68

E.1 Reakce testovací sady pro $t = 3$ na mutace. Zdroj [30]. . . . 72

Seznam tabulek

2.1 Srovnání velikostí vygenerovaných sad pro konfiguraci VSCA(m: 2, 3^{15}, {C}). Zdroj [2]. 20
2.2 Testovací sady a jejich velikosti. Zdroj [2]. 21
2.3 Porovnání testovacích sad vzhledem k počtu nalezených chyb. Zdroj [2]. 22
2.4 Pravidla pro FIS parametru Δw. 23
2.5 Pravidla pro FIS parametru c_1. 24
2.6 Pravidla pro FIS přizpůsobující parametr c_2. 24
2.7 Srovnání velikostí testovacích sad pro CA (N; 2, k, 3). Zdroj [30]. 27
2.8 Porovnání velikostí a časů konstrukce testovacích sad pro CA (N; t, 7, 3). Zdroj [30]. 27
2.9 Parametry programu a jejich příslušné hodnoty. Zdroj [30]. . 29
2.10 Velikosti testovacích sad pro různé *t*. Zdroj [30]. 29
2.11 Srovnání algoritmů TLBO a ATLBO vzhledem k velikosti a času konstrukce testovacích sad. Zdroj [46]. 35
2.12 Srovnání ATLBO s jinými metaheuristickými algoritmy pro konfiguraci VCA(N; 3, 3^{15},). Zdroj [46]. 35
2.13 Srovnání ATLBO s jinými metaheuristickými algoritmy pro konfiguraci CA (N; t, 3^p). Zdroj [46]. 36

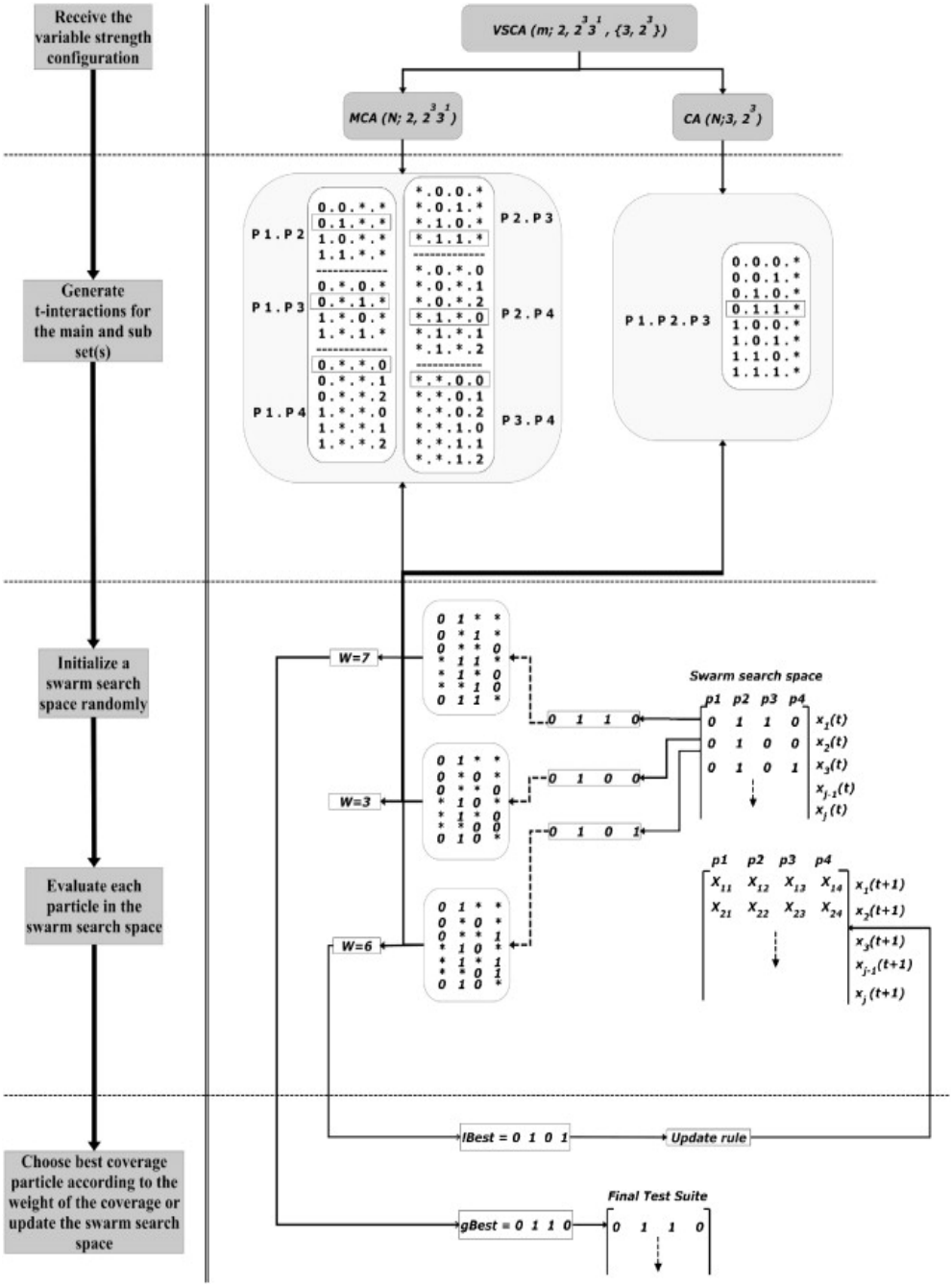
4.1 Srovnání pro různé konfigurace. 49
4.2 Srovnání pro konfiguraci VCA(N; 3, 3^{15},). 49
4.3 Srovnání pro konfiguraci CA(N; t, v^7). 50
4.4 Table 7 Srovnání pro konfiguraci VCA(N; 2, 3^{15},). 50
4.5 Srovnání pro konfiguraci CA(N; t, 3^p). 51
4.6 Srovnání pro konfiguraci VCA(N; 2, 4^3 5^3 6^2,). 51
4.7 Srovnání pro konfiguraci CA(N; t, v^{10}). 52
4.8 Vstupní parametry testovaného programu. 53
4.9 Velikosti sad notace MCA(N; *t*, 7^1 6^1 2^8 3^2) pro různá *t* . . . 53

B.1 Srovnání velikostí vygenerovaných sad pro konfiguraci VSCA(m: 2, 4^3 5^3 6^2, {C}) a VSCA (m; 2, 3^{20} 10^2, {C}). Zdroj [2]. . . 67
B.2 Srovnání velikostí vygenerovaných sad pro konfiguraci VSCA(m: 2, 3^{15}, {C}), VSCA (m; 3, 4^1 3^7 2^2, {C}) a VSCA (m; 2, 10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 {C}). Zdroj [2]. 67

D.1 Srovnání velikostí testovacích sad pro CA (N; 3, k, 3). Zdroj [30]. 69
D.2 Srovnání velikostí testovacích sad pro CA (N; 4, k, 3). Zdroj [30]. 69
D.3 Srovnání velikostí testovacích sad pro CA (N; 2, 7, v). Zdroj [30]. 69
D.4 Srovnání velikostí testovacích sad pro CA (N; 3, 7, v). Zdroj [30]. 70
D.5 Srovnání velikostí testovacích sad pro CA (N; 4, 7, v). Zdroj [30]. 70
D.6 Srovnání velikostí testovacích sad pro konfigurace reálných systémů. Zdroj [30]. 70
D.7 Srovnání metaheuristických algoritmů pro různé konfigurace. Zdroj [30]. 71
D.8 Porovnání velikostí a časů konstrukce testovacích sad pro CA (N; 3, k, 3). Zdroj [30]. 71

F.1 Srovnání ATLBO algoritmu s jinými metaheuristickými algoritmy pro konfiguraci CA (N; t, v^7). Zdroj [46]. 73
F.2 Srovnání ATLBO algoritmu s jinými metaheuristickými algoritmy pro konfiguraci CA (N; t, v^{10}). Zdroj [46]. 73
F.3 Srovnání ATLBO algoritmu s jinými metaheuristickými algoritmy pro konfiguraci CA (N; 2, 3^{15},). Zdroj [46]. 74
F.4 Srovnání ATLBO algoritmu s jinými metaheuristickými algoritmy pro konfiguraci CA (N; 2, 4^3 5^3 6^2,). Zdroj [46]. . . . 74

A Ilustrace VS-PSTG algoritmu



Obrázek A.1: Ilustrace průběhu VS-PSTG metody. Zdroj [2].

B Porovnání vygenerovaných sad metodou VS-PSTG

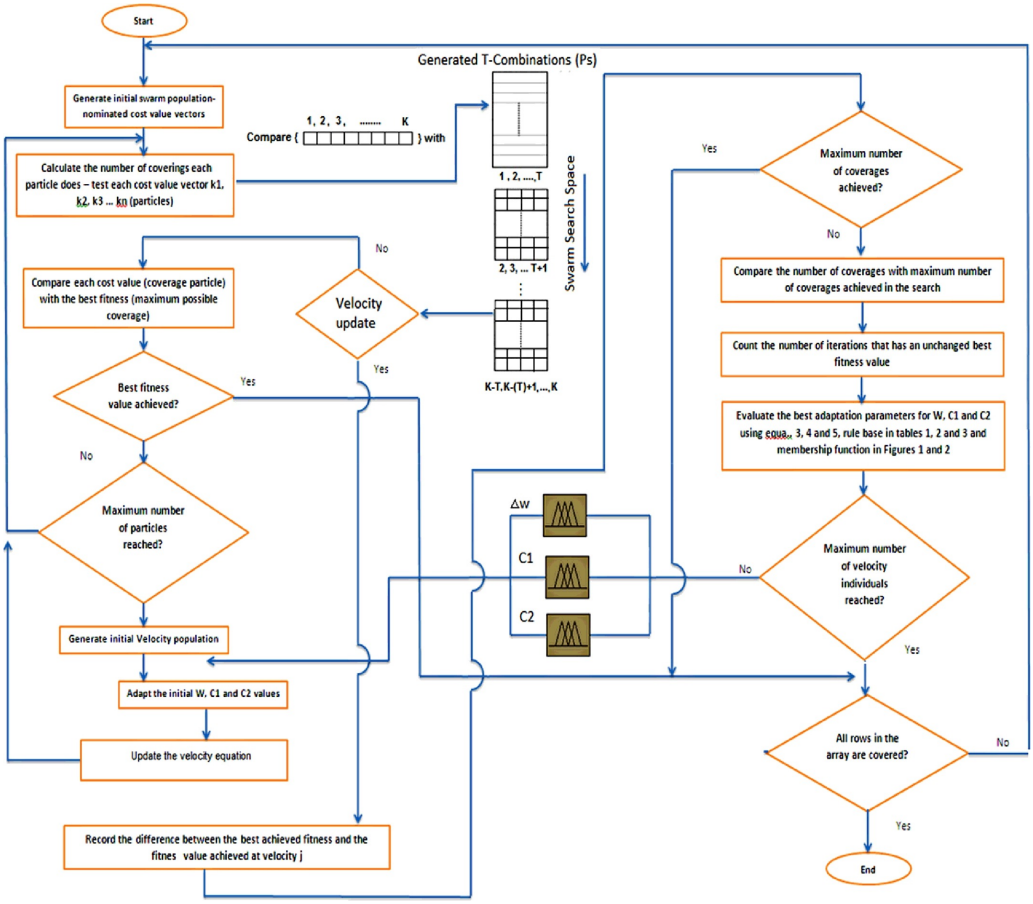
{C}	PICT	ParaOrder	ACS	TVG	SA	Density	IPOG	ITCH	VS-PSTG
Configuration VSCA (m; 2, 4^3 5^3 6^2, {C})									
∅	43	49	41	44	35	41	43	48	42
CA (3, 4^3)	384	64	64	67	64	64	83	97	64
MCA (3, 4^3 5^2)	781	141	104	132	100	131	147	164	124
CA (3, 5^3)	750	126	125	125	125	125	136	145	125
MCA (4, 4^3 5^1)	1920	NA	NA	320	NA	NA	329	354	320
MCA (5, 4^3 5^2)	9600	NA	NA	1600	NA	NA	1602	1639	1600
CA (3, 4^3) CA (3, 5^3)	8000	129	125	125	125	125	136	194	125
MCA (4, 4^3 5^1) MCA (4, 5^2 6^2)	288000	NA	NA	900	NA	NA	900	1220	900
CA (3, 4^3) MCA (4, 5^3 6^1)	48000	NA	NA	750	NA	NA	750	819	750
CA (3, 4^3) MCA (5, 5^3 6^2)	288000	NA	NA	4500	NA	NA	4500	4569	4500
MCA (4, 4^3 5^2)	2874	NA	NA	496	NA	NA	512	510	472
MCA (5, 4^3 5^3)	15048	NA	NA	2592	NA	NA	2763	2520	2430
MCA (3, 4^3 5^3 6^1)	1266	247	201	237	171	207	215	254	206
MCA (3, 5^1 6^2)	900	180	180	180	180	180	180	188	180
MCA (3, 4^3 5^3 6^2)	261	307	255	302	214	256	NS	312	260
Configuration VSCA (m; 2, 3^{20} 10^2, {C}									
∅	100	100	100	101	100	100	101	NA	102
CA (3, 3^{20})	940	103	100	103	100	100	100	NA	105
MCA (3, 3^{20} 10^2)	423	442	396	423	304	401	NS	NA	481
MCA (4, 3^3 10^1)	810	NA	NA	270	NA	NA	273	NA	270
MCA (5, 3^3 10^2)	2430	NA	NA	2700	NA	NA	2700	NA	2700
MCA (6, 3^4 10^2)	7290	NA	NA	8100	NA	NA	8100	NA	8100

Tabulka B.1: Srovnání velikostí vygenerovaných sad pro konfiguraci VSCA(m: 2, 4^3 5^3 6^2, {C}) a VSCA (m; 2, 3^{20} 10^2, {C}). Zdroj [2].

Configuration VSCA (m; 3, 3^{15}, {C})						**Configuration VSCA (m; 3, 4^1 3^7 2^2, {C})**					
{C}	PICT	TVG	IPOG	ITCH	VS-PSTG	{C}	PICT	TVG	IPOG	ITCH	VS-PSTG
∅	83	84	82	75	75	∅	72	70	73	112	65
CA (4, 3^4)	1597	93	87	129	91	MCA (4, 4^1 3^3)	1377	111	108	193	108
CA (4, 3^4)^2	19749	97	91	183	91	MCA (4, 4^1 3^3) CA (4, 3^4)	17496	112	108	253	108
CA (4, 3^4)^3	531441	97	106	237	91	MCA(4, 4^1 3^3) MCA (6, 3^4 2^2)	NS	324	326	497	324
CA (4, 3^4)^3 CA(3, 3^3)	NA	98	106	237	90	MCA (4, 4^1 3^4)	1500	141	149	217	136
CA (5, 35)	5366	244	243	273	243	MCA (4, 4^1 3^5)	1547	183	207	226	171
CA (5, 3^5)^2	177300	245	250	459	245	MCA (4, 4^1 3^4) MCA (5, 3^3 2^2)	NS	141	149	307	136
CA (5, 3^5)^3	NA	245	261	645	245	MCA (5, 4^1 3^4)	3586	325	324	395	324
CA (6, 3^6)	12609	729	729	759	729	MCA (5, 4^1 3^4) MCA (5, 3^3 2^2)	NS	325	324	482	324
CA (6, 3^6)^2	NA	730	744	1431	734	**Configuration VSCA (m; 2 10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1, {C})**					
CA (6, 3^6)^2 CA(3, 3^3)	NA	730	744	1431	734	{C}	PICT	TVG	IPOG	ITCH	VS-PSTG
CA (4, 3^5)	1793	118	119	151	114	∅	102	99	91	119	97
CA (5, 3^6)	5387	323	337	287	314	MCA (3, 10^1 9^1 8^1)	31256	720	720	765	720
CA (6, 3^7)	16792	1018	1215	1044	1002	MCA (3, 7^1 6^1 5^1)	19515	210	221	301	210
CA (4, 3^7)	2781	168	183	219	159	MCA (3, 4^1 3^1 2^1)	2397	99	91	140	97
CA (4, 3^9)	3095	214	227	289	195	MCA (3, 10^1 9^1 8^1 7^1)	22878	784	772	806	742
CA (4, 3^{11})	2824	256	259	354	226	MCA (3, 10^1 9^1 8^1) MCA (3, 7^1 6^1 5^1)	NA	720	720	947	720
CA (4, 3^{15})	NS	327	NS	498	284	MCA (3, 10^1 9^1 8^1) MCA (6, 7^1 6^1 5^1 4^1 3^1 2^1)	NA	5040	5041	5803	5040
CA (5, 3^7)	7475	471	713	481	437	MCA (3, 10^1 9^1 8^1) MCA (3, 7^1 6^1 5^1) MCA (3, 4^1 3^1 2^1)	NA	720	720	968	720
CA (5, 3^8)	8690	556	714	620	516	MCA (4, 5^1 4^1 3^1 2^1)	1200	123	142	237	120
CA (5, 3^{10})	9774	745	862	868	665	MCA (5, 10^1 9^1 4^1 3^1 2^1)	124157	2160	2160	2276	2160
CA (5, 3^{12})	8909	925	1130	NA	805	MCA (6, 7^1 6^1 5^1 4^1 3^1 2^1)	NA	5040	5041	5157	5040
CA (5, 3^{15})	NS	NA	NS	NA	1024						
CA (6, 3^8)	22833	1479	2108	1513	1399						
CA (6, 3^9)	2625	1840	2124	1964	1690						
CA (3, 3^4) CA (4, 3^5) CA (5, 3^6)	NA	331	419	312	312						

Tabulka B.2: Srovnání velikostí vygenerovaných sad pro konfiguraci VSCA(m: 2, 3^{15}, {C}), VSCA (m; 3, 4^1 3^7 2^2, {C}) a VSCA (m; 2, 10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 {C}). Zdroj [2].

C Ilustrace FSAPSO algoritmu



Obrázek C.1: Ilustrace FSAPSO algoritmu. Zdroj [30].

D Porovnání vygenerovaných sad metodou FSAPSO

v = 3											
	k	Jenny	TConfig	PICT	TVG	IPOG-D	IPOG	PSO		FSAPSO	
		Size	Size	Size	Size	Size	Size	Bst. size	Avg. size	Bst. size	Avg. size
t = 3	4	34	32	34	34	27	39	27	29.3	27	29.9
	5	40	40	43	41	49	43	39	41.37	41	42.2
	6	51	48	48	49	49	53	45	46.76	45	46.51
	7	51	55	51	55	53	57	50	52.2	48	51.12
	8	58	58	59	60	63	63	54	56.76	50	54.86
	9	62	64	63	64	71	65	58	60.30	59	60.21
	10	65	68	65	68	71	68	62	63.95	63	64.33

Tabulka D.1: Srovnání velikostí testovacích sad pro CA (N; 3, k, 3). Zdroj [30].

v = 3											
	k	Jenny	TConfig	PICT	TVG	IPOG-D	IPOG	PSO		FSAPSO	
		Size	Size	Size	Size	Size	Size	Bst. size	Avg. size	Bst. size	Avg. size
t = 4	5	109	97	100	105	NS	115	96	97.83	94	96.33
	6	140	141	142	139	234	181	133	135.31	129	133.98
	7	169	166	168	172	NS	185	155	158.12	154	157.42
	8	187	190	189	192	384	203	175	176.94	178	179.7
	9	206	213	211	215	NS	238	195	198.72	190	194.13
	10	221	235	231	233	498	241	210	212.71	214	212.21

Tabulka D.2: Srovnání velikostí testovacích sad pro CA (N; 4, k, 3). Zdroj [30].

p = 7											
	v	Jenny	TConfig	PICT	TVG	IPOG-D	IPOG	PSO		FSAPSO	
		Size	Size	Size	Size	Size	Size	Bst. size	Avg. size	Bst. size	Avg. size
t = 2	2	8	7	7	7	8	8	6	6.82	6	6.73
	3	15	15	16	15	18	17	15	15.23	15	15.56
	4	28	28	27	27	28	28	26	27.22	25	26.36
	5	37	40	40	42	52	42	37	38.14	35	37.92

Tabulka D.3: Srovnání velikostí testovacích sad pro CA (N; 2, 7, v). Zdroj [30].

p = 7											
	k	Jenny	TConfig	PICT	TVG	IPOG-D	IPOG	PSO		FSAPSO	
		Size	Size	Size	Size	Size	Size	Bst. size	Avg. size	Bst. size	Avg. size
t = 3	2	14	16	15	15	14	19	13	13.61	15	13.8
	3	51	55	51	55	63	57	50	52.2	48	51.12
	4	124	112	124	134	112	208	116	118.13	118	120.41
	5	236	239	241	260	292	275	225	227.21	239	243.29

Tabulka D.4: Srovnání velikostí testovacích sad pro CA (N; 3, 7, v). Zdroj [30].

p = 7											
	k	Jenny	TConfig	PICT	TVG	IPOG-D	IPOG	PSO		FSAPSO	
		Size	Size	Size	Size	Size	Size	Bst. size	Avg. size	Bst. size	Avg. size
t = 4	2	31	36	32	31	NS	48	29	31.49	30	31.34
	3	169	166	168	167	NS	185	155	157.77	153	155.2
	4	517	568	529	559	NS	509	487	489.91	472	478.9
	5	1248	1320	1279	1385	NS	1349	1176	1180.63	1162	1169.94

Tabulka D.5: Srovnání velikostí testovacích sad pro CA (N; 4, 7, v). Zdroj [30].

System	Config.	Jenny	TConfig	PICT	TVG	IPOG-D	IPOG	PSO		FSAPSO	
		Size	Size	Size	Size	Size	Size	Bst. size	Avg. size	Bst. size	Avg. size
BBS	CA (N; 2, 3^4)	13	10	13	12	15	12	9	9.19	9	9.11
TCAS	MCA (N; 2, 2^7 3^2 4^1 10^2)	106	109	100	100	130	100	100	102.1	100	101.95
Mobile phone	MCA (N; 2, 2^2 3^3)	12	12	10	10	15	11	9	9.23	9	9.47
Spin simulator	MCA (N; 2, 2^{13} 4^5)	26	29	23	27	28	20	24	25.8	22	24.33
BBS	CA (N; 3, 3^4)	34	32	34	32	27	39	27	28.35	27	29.1
TCAS	MCA (N; 3, 2^7 3^2 4^1 10^2)	413	472	400	434	480	400	400	401.75	400	400.29
Mobile phone	MCA (N; 3, 2^2 3^3)	29	30	29	30	34	27	27	29.34	27	29.8
Spin simulator	MCA (N; 3, 2^{13} 4^5)	111	113	96	111	112	78	101	103.72	87	93.89
TCAS	MCA (N; 4, 2^7 3^2 4^1 10^2)	1536	1548	1369	1599	NS	1377	1520	1528.25	1511	1516.32
Mobile phone	MCA (N; 4, 2^2 3^3)	59	56	59	55	NS	54	54	54.63	54	54.82
Spin simulator	MCA (N; 4, 2^{13} 4^5)	412	427	353	288	NS	341	380	384.82	372	376.91

Tabulka D.6: Srovnání velikostí testovacích sad pro konfigurace reálných systémů. Zdroj [30].

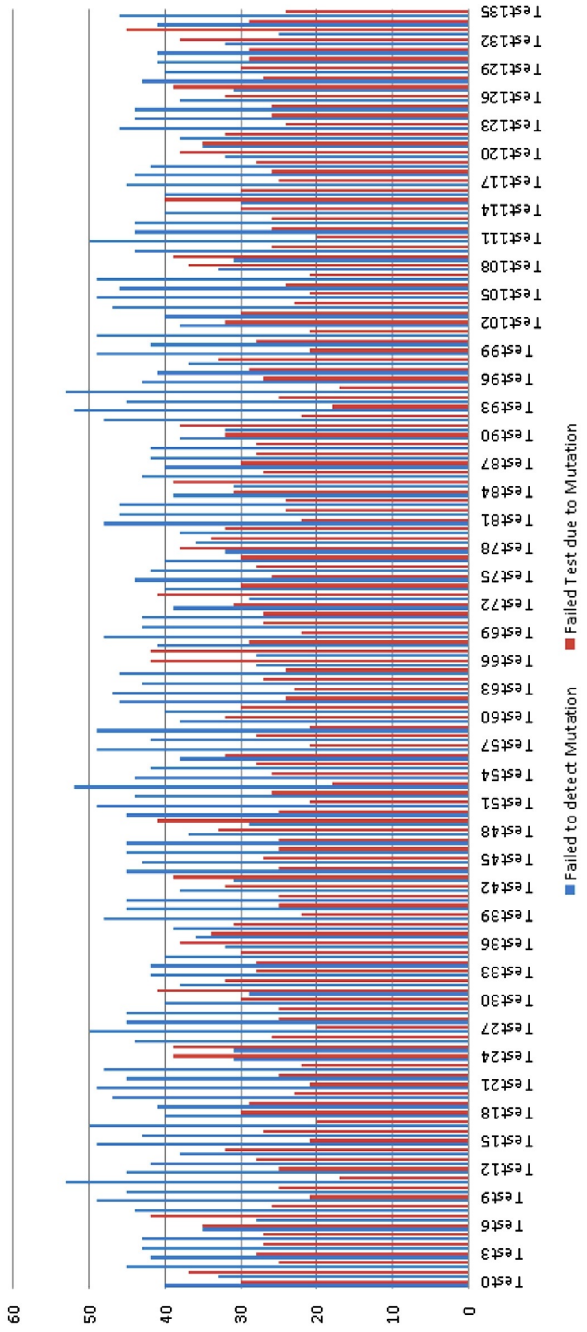
CA Config.	GA	SA	ACA	PSO	FSAPSO
	N	N	N	N	N
CA (N; 2, 3^4)	9	9	9	9	9
CA (N; 2, 3^{13})	17	16	17	17	16
MCA (N; 2, 5^1 3^8 2^2)	15	15	16	21	18
MCA (N; 2, 6^1 5^1 4^6 3^8 2^3)	33	30	32	39	35
MCA (N; 7^1 6^1 5^1 4^6 3^8 2^3)	42	42	42	48	42
CA (N; 3, 3^6)	33	33	33	42	36
CA (N; 3, 4^6)	64	64	64	102	75
CA (N; 3, 5^7)	218	201	218	229	219
CA (N; 3, 6^6)	331	300	330	338	332
MCA (N; 10^1 6^2 4^3 3^1)	360	360	361	385	383

Tabulka D.7: Srovnání metaheuristických algoritmů pro různé konfigurace. Zdroj [30].

k	Jenny	TConfig	PICT	TVG	IPOG-D	IPOG	PSO		FSAPSO	
	N/Time	N/Time	N/Time	N/Time	N/Time	N/Time	Bst. N/Time	Avg. N/Avg. time	Bst. N/Time	Avg. N/Avg. time
4	34/0.08	32/0.17	34/0.14	34/0.17	27/0.04	39/0.27	27/0.17	29.3/0.32	27/1.56	29.65/2.97
5	40/0.12	40/0.25	43/0.45	41/0.21	49/0.12	43/0.34	39/1.739	41.37/2.56	37/8.62	40.22/11.45
6	51/0.47	48/0.67	48/0.83	49/0.48	49/0.12	53/0.58	45/2.25	46.76/3.1	45/10.87	46.86/12.95
7	51/0.57	55/1.86	51/0.98	55/0.57	63/0.36	57/0.614	50/4.21	52.2/5.56	48/13.5	51.12/16.74
8	58/0.73	58/2.48	59/1.3	60/1.251	63/0.49	63/0.98	54/7.15	56.76/9.2	56/30.6	57.43/40.62
9	62/0.82	64/3.32	63/2.76	64/1.812	71/0.111	65/1.36	58/9.03	60.30/12.8	58/42.32	60.33/53.07
10	65/1.16	68/6.71	65/2.94	68/2.414	71/0.112	68/1.92	62/1.92	63.95/16.73	64/64.87	65.19/89.15

Tabulka D.8: Porovnání velikostí a časů konstrukce testovacích sad pro CA (N; 3, k, 3). Zdroj [30].

E Reakce FSAPSO sady na mutace



Obrázek E.1: Reakce testovací sady pro $t = 3$ na mutace. Zdroj [30].

F Porovnání vygenerovaných sad metodou ATLBO

t	v	PSTG		DPSO		APSO		CS		Original TLBO		ATLBO			
		Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	% mean exploit	% mean explore
2	2	6*	6.82	7	7.00	6*	6.73	6*	6.80	7	7.00	7	7.00	50.87	49.13
	3	15	15.23	14*	15.00	15	15.56	15	16.20	15	15.10	15	15.07	51.60	48.40
	4	26	27.22	24	25.33	25	26.36	25	26.40	24	25.27	23*	25.17	57.74	42.26
	5	37	38.14	34*	35.47	35	37.92	37	38.60	34*	35.43	34*	35.47	63.82	36.18
3	2	13	13.61	15	15.06	15	15.80	12*	13.80	15	15.12	15	15.12	48.42	51.58
	3	50	51.75	49	50.60	48*	51.12	49	51.60	49	50.38	49	50.29	39.60	60.40
	4	116	118.13	112	115.27	118	120.41	117	118.40	112	115.37	111*	115.67	43.16	59.84
	5	225	227.21	216*	219.20	239	243.29	223	225.40	217*	219.90	216*	219.40	44.77	55.23
4	2	29	31.49	34	34.00	30	31.34	27*	29.60	31	33.70	31	33.68	46.04	53.96
	3	155	157.77	150*	154.73	153	155.20	155	156.80	151	155.25	150*	155.24	39.42	60.58
	4	487	489.91	472	481.53	472	478.90	487	490.20	480	485.53	478	484.69	39.90	60.10
	5	1176	1180.63	1148*	1155.63	1162	1169.94	1171	1175.20	1166	1173.17	1166	1173.45	40.14	59.86

Tabulka F.1: Srovnání ATLBO algoritmu s jinými metaheuristickými algoritmy pro konfiguraci CA (N; t, v^7). Zdroj [46].

t	v	PSTG		DPSO		CS		Original TLBO		ATLBO			
		Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	% mean exploit	% mean explore
2	4	-	-	28*	29.20	-	-	28*	28.73	28*	28.69	42.42	57.58
	5	45	48.31	42	43.67	45	47.8	41*	43.30	42	43.53	46.92	53.08
	6	-	-	58*	59.23	-	-	58*	59.47	58*	59.33	50.27	49.73
3	4	-	-	141	143.70	-	-	140*	142.57	140*	142.80	30.77	69.23
	5	287	298.00	273	276.20	297	299.20	273	275.70	272*	275.23	31.04	68.96
	6	-	-	467	470.50	-	-	467	470.47	466*	469.90	31.53	68.47
4	4	-	-	664	667.00	-	-	663	668.12	661*	664.06	25.68	74.32
	5	1716	1726.72	1618*	1620.80	1731	1740.20	1621	1621.80	1619	1620.91	22.32	77.68
	6	-	-	3339	3342.50	-	-	3338*	3343.81	3338*	3342.10	21.13	78.87

Tabulka F.2: Srovnání ATLBO algoritmu s jinými metaheuristickými algoritmy pro konfiguraci CA (N; t, v^{10}). Zdroj [46].

ID		PSTG		DPSO		ACS		SA		Original TLBO		ATLBO			
		Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	% mean exploit	% mean explore
VCA1	$\emptyset$	19	20.92	18	18.63	19	-	16*	-	19	19.67	18	19.30	31.30	68.70
VCA2	CA(3, 3^3)	27*	27.50	27*	27.27	27*	-	27*	-	27*	27.33	27*	27.00	22.26	77.74
VCA3	CA (3, 3^3)^2	27*	27.94	27*	27.83	27*	-	27*	-	27*	27.47	27*	27.53	21.46	78.54
VCA4	CA (3, 3^3)^3	27*	28.13	27*	28.00	27*	-	27*	-	27*	27.93	27*	27.43	22.00	78.00
VCA5	CA (3, 3^4)	30	31.47	27*	31.43	27*	-	27*	-	27*	32.73	27*	27.00	22.26	77.74
VCA6	CA (3, 3^5)	38	39.83	38	40.93	38	-	33*	-	38	40.97	38	40.60	16.25	83.75
VCA7	CA (3, 3^6)	45	46.42	43	45.70	45	-	34*	-	43	43.73	43	43.67	18.05	81.95
VCA8	CA (3, 3^7)	49	51.68	47*	49.87	48	-	41	-	49	50.03	47*	49.83	17.96	82.04
VCA9	CA (4, 3^4)	81*	82.21	81*	81.03	-	-	-	-	81*	81.03	81*	81.03	7.44	92.56
VCA10	CA (4, 3^5)	97	99.31	85*	94.50	-	-	-	-	89	97.53	87	96.90	7.26	92.74
VCA11	CA (4, 3^7)	158	160.31	152*	156.83	-	-	-	-	153	156.51	152*	156.33	10.74	89.26

Tabulka F.3: Srovnání ATLBO algoritmu s jinými metaheuristickými algoritmy pro konfiguraci CA (N; 2, 3^{15},). Zdroj [46].

ID		PSTG		DPSO		HSS		ACS		SA		Original TLBO		ATLBO			
		Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	% mean exploit	% mean explore
VCA1	$\emptyset$	42	43.60	40	42.30	42	43.50	41	-	36*	-	40	42.03	39	41.63	43.47	56.53
VCA2	CA (3, 4^3)	64*	65.50	64*	64.00	64*	64.00	64	-	64	-	64*	64.03	64*	64.03	31.11	68.89
VCA3	CA (3, 4^3 5^2)	124	126.60	119	124.70	116	120.90	104	-	100*	-	121	125.67	122	124.5	18.31	81.69
VCA4	CA (3, 4^3) CA (3, 5^3)	125*	127.90	125*	125.00	125*	125.00	125*	-	125*	-	125*	125.00	125*	125.00	15.88	84.12
VCA5	CA (3, 4^3 5^3 6^1)	206	210.20	203	207.50	212	214	201	-	171*	-	203	208.77	203	208.68	14.42	85.58
VCA6	CA (3, 4^3) CA (4, 5^3 6^1)	750	755.70	750*	750.80	750*	750.00	-	-	-	-	750*	750.00	750*	750.00	12.70	87.30
VCA7	CA (4, 4^3 5^2)	472	478.10	440*	450.60	453	454.3	-	-	-	-	459	466.70	451	459.10	6.52	93.48

Tabulka F.4: Srovnání ATLBO algoritmu s jinými metaheuristickými algoritmy pro konfiguraci CA (N; 2, 4^3 5^3 6^2,). Zdroj [46].

G Popis fuzzy systému jazykem FCL

```
FUNCTION_BLOCK fuzzy_atlbo

VAR_INPUT
    quality : REAL;
    intensification : REAL;
    diversification : REAL;
END_VAR

VAR_OUTPUT
    selection : REAL;
END_VAR

FUZZIFY quality
    TERM low := trape 0 0 20 40;
    TERM medium := trape 20 40 60 80;
    TERM high := trape 60 80 100 100;
END_FUZZIFY

FUZZIFY intensification
    TERM low := trape 60 80 100 100;
    TERM medium := trape 20 40 60 80;
    TERM high := trape 0 0 20 40;
END_FUZZIFY

FUZZIFY diversification
    TERM low := trape 0 0 20 40;
    TERM medium := trape 20 40 60 80;
    TERM high := trape 60 80 100 100;
END_FUZZIFY

DEFUZZIFY selection
    TERM local := trape 0 0 20 80;
    TERM global := trape 20 80 100 100;
    METHOD : COG;
    DEFAULT := 0;
END_DEFUZZIFY

RULEBLOCK No1
    AND : MIN;
    ACT : MIN;
    ACCU : MAX;
    RULE 1 : IF quality IS NOT high THEN selection IS global;
    RULE 2 : IF quality IS high AND diversification IS NOT high AND intensification IS high
         ↪ THEN selection IS global;
    RULE 3 : IF quality IS high AND diversification IS high AND intensification IS NOT high
         ↪ THEN selection IS local;
    RULE 4 : IF quality IS high AND diversification IS high AND intensification IS high
         ↪ THEN selection IS local;
END_RULEBLOCK

END_FUNCTION_BLOCK
```

H Vstupní soubor generátoru testovacích tříd

```
<controlFlowOutput>
    <classes>
        <class>
            <classPath>/src/main/java/</classPath>
            <packageName>ctgen.example01</packageName>
            <className>Entrance</className>
            <methods>
                <method>
                    <name>canEnter</name>
                    <parameterCount>3</parameterCount>
                    <isStatic>false</isStatic>
                    <returnType>boolean</returnType>
                    <interactionStrength>2</interactionStrength>
                    <methodParameters>
                        <methodParameter>
                            <dataType>ctgen.example01.Person</dataType
                                ↪ >
                            <index>0</index>
                            <name>person</name>
                            <primitive>false</primitive>
                            <isInterface>false</isInterface>
                            <isAbstract>false</isAbstract>
                            <isEnum>false</isEnum>
                            <objects/>
                            <primitiveFields>
                                <primitiveField>
                                    <dataType>int</dataType>
                                    <index>0</index>
                                    <name>age</name>
                                    <primitive>true</primitive>
                                    <values>
                                        <value>10</value>
                                        <value>11</value>
                                        <value>39</value>
                                    </values>
                                </primitiveField>
```

```
                        <primitiveField>
                            <dataType>int</dataType>
                            <index>0</index>
                            <name>height</name>
                            <primitive>true</primitive>
                            <values>
                                <value>10.23</value>
                                <value>12.23</value>
                            </values>
                        </primitiveField>
                    </primitiveFields>
                    <possibleObjectTypes/>
                </methodParameter>
                <methodParameter>
                    <dataType>int</dataType>
                    <index>1</index>
                    <name>limitedAge</name>
                    <primitive>true</primitive>
                    <values>
                        <value>9</value>
                        <value>12</value>
                    </values>
                </methodParameter>
                <methodParameter>
                    <dataType>double</dataType>
                    <index>2</index>
                    <name>limitedHeight</name>
                    <primitive>true</primitive>
                    <values>
                        <value>9.45</value>
                        <value>12.93</value>
                    </values>
                </methodParameter>
            </methodParameters>
            <conditions>
                ...
            </conditions>
        </method>
    </methods>
</class>
</classes>
</controlFlowOutput>
```

I Příklad vygenerovaných testů

```
import ctgen.example01.Entrance;
import org.junit.jupiter.api.Test;
import ctgen.example01.Person;

public class EntranceTest {
  @Test
  public void canEnterTestCase1() {
    Entrance entrance = new Entrance();

    Person person = new Person();
    person.setAge(11);
    person.setHeight(12.23);
    int limitedAge = 9;
    double limitedHeight = 9.45;
    entrance.canEnter(person, limitedAge, limitedHeight);
  }

  @Test
  public void canEnterTestCase2() {
    Entrance entrance = new Entrance();

    Person person = new Person();
    person.setAge(11);
    person.setHeight(10.23);
    int limitedAge = 12;
    double limitedHeight = 12.93;
    entrance.canEnter(person, limitedAge, limitedHeight);
  }

  @Test
  public void canEnterTestCase3() {
    Entrance entrance = new Entrance();

    Person person = new Person();
    person.setAge(10);
    person.setHeight(10.23);
    int limitedAge = 9;
    double limitedHeight = 9.45;
    entrance.canEnter(person, limitedAge, limitedHeight);
  }
```

```
  @Test
  public void canEnterTestCase4() {
    Entrance entrance = new Entrance();

    Person person = new Person();
    person.setAge(39);
    person.setHeight(12.23);
    int limitedAge = 9;
    double limitedHeight = 12.93;
    entrance.canEnter(person, limitedAge, limitedHeight);
  }

  @Test
  public void canEnterTestCase5() {
    Entrance entrance = new Entrance();

    Person person = new Person();
    person.setAge(39);
    person.setHeight(12.23);
    int limitedAge = 12;
    double limitedHeight = 9.45;
    entrance.canEnter(person, limitedAge, limitedHeight);
  }

  @Test
  public void canEnterTestCase6() {
    Entrance entrance = new Entrance();

    Person person = new Person();
    person.setAge(10);
    person.setHeight(12.23);
    int limitedAge = 12;
    double limitedHeight = 12.93;
    entrance.canEnter(person, limitedAge, limitedHeight);
  }

  @Test
  public void canEnterTestCase7() {
    Entrance entrance = new Entrance();

    Person person = new Person();
    person.setAge(39);
    person.setHeight(10.23);
    int limitedAge = 12;
    double limitedHeight = 9.45;
    entrance.canEnter(person, limitedAge, limitedHeight);
  }
}
```

J Obsah přiloženého CD

Součástí práce jsou následující přílohy, které jsou na přiloženém CD:

- spustitelný *jar* soubor a příklady vstupních souborů spolu se skripty, které pro vstupní soubory vygenerují výstupní testy, ve složce **bin**.
- zdrojové soubory aplikace *CTGen* ve složce **src**.
- elektronická verze textu této diplomové práce se zdrojovými soubory ve složce **doc**.
- poster ve složce **Poster**.

K Uživatelská příručka

Výsledná knihovna je zabalena ve spustitelném **CTGen.jar** souboru, ve složce *bin*. Program má několik parametrů:

1. {**-i**, **-inputFile**} - cesta k souboru s popisem testovaného programu. Soubor musí končit buď příponou *.xml* nebo *.json*. Tento parametr je **povinný!**
2. {**-e**, **-extension**} - přípona vygenerovaných souborů. Zadává se bez tečky (např. 'java', ne '**.**java'). Výchozí hodnota parametru je '*java*'.
3. {**-f**, **-outputFolder**} - cílová složka pro generované soubory. Neexistující složky na této cestě se automaticky vytvoří. Výchozí hodnota parametru je '*./*'.
4. {**-p**, **-populationSize**} - velikost populace pro ATLBO algoritmus. Výchozí hodnota je '40'.
5. {**-l**, **-atlboSuiteLimit**} - maximální velikost testovacích sad generovaných ATLBO algoritmem. Výchozí hodnota je Integer.MAX_VALUE (algoritmus poběží dokud nepokryje všechny možné kombinace).
6. {**-c**, **-compileAndRunImmediately**} - zda se vygenerované testy ihned přeloží a spustí. Výchozí hodnota je '*false*'.
7. {**-h**, **-help**} - vypíše nápovědu

Následující příklad načte popis testovaného programu ze souboru *entrance.xml*, vygenerované testy uloží do složky na cestě *tests/generated* a testy se pokusí ihned zkompilovat a spustit.

java -jar CTGen.jar -i entrance.xml -f tests/generated -c true

Dalším způsobem jak testy spustit je s pomocí vývojového studia. Vygenerované soubory stačí vložit do složky '*ctgen-examples/src/test/java/*' a poté je z vývojového studia spustit jako normální *JUnit* testy. Pokud se soubory vloží do nějakého balíku (package), je potřeba na začátek souboru přidat definici balíku - *package <identifikátor balíku>;*

```
java.lang.NullPointerException
      at memorycompile.CodeRunner.run(CodeRunner.java:30)
      at ctgen.CTGen.generate(CTGen.java:115)
      at ctgen.CTGenBuilder.go(CTGenBuilder.java:153)
      at Main.main(Main.java:23)
```

Zdrojový kód K.1: Some Java code

Pokud při spuštění CTGenu s parametrem *-c* nastavený na *true* nastane výjimka viz kód K.1, musí se při spuštění JAR souboru specifikovat použití JDK, viz níže.

"C:\Program Files\Java\jdk1.8.0_181\bin\java" -jar CTGen.jar -i entrance.xml -c true