VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

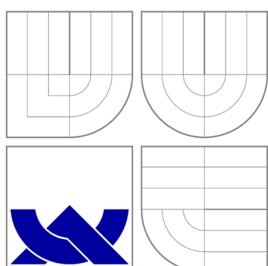
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

APLIKACE EVOLUČNÍHO ALGORITMU PŘI TVORBĚ REGRESNÍCH TESTŮ

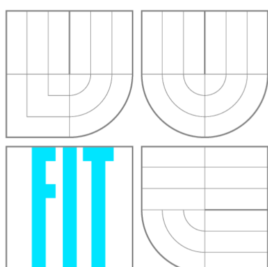
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MICHAELA BELEŠOVÁ
AUTHOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

APLIKACE EVOLUČNÍHO ALGORITMU PŘI TVORBĚ REGRESNÍCH TESTŮ

APPLICATION OF EVOLUTIONARY ALGORITHM IN CREATION OF REGRESSION TESTS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MICHAELA BELEŠOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. MARCELA ŠIMKOVÁ
SUPERVISOR

BRNO 2014

Abstrakt

Cílem této diplomové práce je aplikace evolučního algoritmu při tvorbě a optimalizaci regresních testů. V teoretické části práce je popsána teorie spojená s funkční verifikací, verifikační metodikou, regresními testy a evolučními algoritmy. Dále je vytvořen návrh evolučního algoritmu, který umožní zredukovat počet testovacích vektorů vygenerovaných v procesu funkční verifikace za účelem tvorby optimalizovaných regresních testů. Vytvořený návrh je implementován a je na něm provedena sada experimentů. Dosažené výsledky jsou diskutovány.

Abstract

This master thesis deals with application of an evolutionary algorithm in the creation of regression tests. In the first section, description of functional verification, verification methodology, regression tests and evolutionary algorithms is provided. In the following section, the evolutionary algorithm, the purpose of which is to achieve reduction of the number of test vectors obtained in the process of functional verification, is proposed. Afterwards, the proposed algorithm is implemented and a set of experiments is evaluated. The results are discussed.

Klíčová slova

Funkční verifikace, verifikace řízená pokrytím, UVM, SystemVerilog, regresní testy, evoluční algoritmy, genetické algoritmy.

Keywords

Functional verification, coverage-driven verification, UVM, SystemVerilog, regression tests, evolutionary algorithms, genetic algorithms.

Citace

Michaela Belešová: Aplikace evolučního algoritmu při tvorbě regresních testů, diplomová práce, Brno, FIT VUT v Brně, 2014

Aplikace evolučního algoritmu při tvorbě regresních testů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod odborným vedením Ing. Marcely Šimkové

....
Michaela Belešová
27. května 2014

Poděkování

Chtěla bych poděkovat vedoucí práce Ing. Marcele Šimkové za její odbornou pomoc a rady při psaní této práce.

© Michaela Belešová, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **2**

2 Teoretický úvod **4**
2.1 Funkčná verifikácia . . . 4
2.1.1 SystemVerilog . . . 8
2.1.2 UVM . . . 9
2.2 Regresné testy . . . 14
2.3 Evolučné algoritmy . . . 15
2.3.1 Genetické algoritmy . . . 19

3 Zadanie a analýza úlohy **24**
3.1 Nameraná redundancia vektorov v konkrétnom obvode . . . 25
3.2 Evolučný algoritmus pre optimalizáciu regresných testov . . . 25

4 Základný návrh riešenia **28**
4.1 Evolučný algoritmus . . . 28
4.2 Automatizácia testov . . . 30
4.3 Verifikačné prostredie UVM . . . 30
4.4 Súhrn . . . 32

5 Návrh a implementácia **34**
5.1 Verifikačné prostredie . . . 34
5.2 Automatizácia testov . . . 36
5.3 Základný genetický algoritmus . . . 36
5.4 Fitness funkcia . . . 43
5.5 Jedinci s nedostatočným pokrytím vlastností . . . 45

6 Namerané výsledky **50**

7 Záver **58**

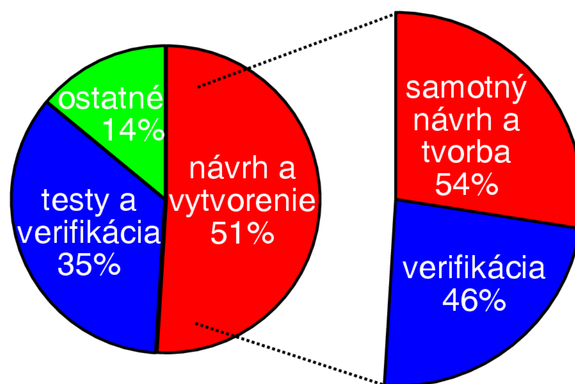
A Zoznam parametrov genetického algoritmu **62**

Kapitola 1

Úvod

Dnešné hardvérové obvody sú veľmi komplexné, obsahujú desiatky dokonca až stovky miliónov hradiel, zabudované mikroprocesory a rôzne vstupno-výstupné rozhrania. Overovanie správnosti takýchto zariadení starým spôsobom, teda opakovanou syntézou a ladením priamo v hardvéri, už nie je možné. Na vylepšenie procesu overovania správnosti tak vzniklo niekoľko rôznych verifikačných a testovacích nástrojov a metodík, ktoré sa stali neoddeliteľnou súčasťou vývojového procesu hardvéru.

V súčasnosti trend zväčšovania komplexnosti obvodov pokračuje a je nevyhnutné, aby verifikácia a testovanie dokázali držať krok. Výskumy a literatúra, napr. [11] a [2] sa zhodujú, že aktuálne proces verifikácie a testovania bežne zaberá viac než polovicu celkového úsilia pri vývoji, čo je zbytočne veľa. Na obrázku 1.1 je znázornený graf demonštrujúci výsledky jedného z takýchto výskumov. Modré výseky, ktoré spolu zaberajú viac ako polovicu grafu, reprezentujú úsilie venované verifikácií a testovaniu. Súčasný trend je stúpajúca podpora zo strany priemyslu o oblasť *funkčnej* verifikácie a testovania [10] a skúmanie nových možností ich optimalizácie.



Obrázek 1.1: Testovanie a verifikácia zaberajú vyše polovicu úsilia na projekte [2].

Zaujímavou oblasťou, ktorú by bolo vhodné optimalizovať je regresné testovanie. Regresné testovanie znamená uchovávanie sady testov a ich opätovné spúšťanie. Pravidelný beh komplexného ale čo najrýchlejšieho regresného testovania je nevyhnutný, keďže modifikovanie kódu i v odľahlých komponentoch je veľmi náchylné na zavádzanie nových chýb do celkovej funkcionality systému. Rýchlosť behu testovania umožňuje frekventovanejšie spúšťanie testov a teda zvyšuje rýchlosť nachádzania novo zavedených chýb. Tá je dôležitá

aj pre skrátenie spätného dohľadania týchto chýb. Otázkou ostáva, odkiaľ získať komplexné a rýchlo prevediteľné regresné testy. Ako možné riešenie, ktorého správnosť má za úlohu overiť táto diplomová práca, je využiť ako zdroj regresných testov sekvencie vektorov získaných z procesu funkčnej verifikácie, ktoré sú následne optimalizované evolučným algoritmom. Myšlienka je tá, že funkčná verifikácia dokáže vygenerovať tak komplexnú sadu vektorov, ktorá splňuje pokrytie (angl. *coverage*) špecifikáciou daných vlastností obvodu na 100 %. Sada vektorov z funkčnej verifikácie je však vysoko redundantná tým, že overuje mnohé vlastností niekoľkokrát. Daný fakt je pre koncept funkčnej verifikácie žiadaný, u regresných testov je však potrebná optimalizácia.

Táto diplomová práca sa venuje návrhu a implementácií konceptu optimalizačného prostredia, ktoré umožní zredukovať počet testovacích vektorov v sade regresných testov získaných z procesu funkčnej verifikácie vybraného obvodu. Druhá kapitola práce, teoretický úvod, popisuje teóriu ohľadne funkčnej verifikácie, jazyka SystemVerilog, metodiky UVM (angl. *Universal Verification Methodology*), regresných testov a evolučných algoritmov s podrobnejším zameraním na genetické algoritmy. V ďalšej kapitole je spresnené a analyzovanie zadanie práce. Štvrtá kapitola sa venuje návrhu optimalizačného prostredia pre redukciu počtu testovacích vzoriek v sade regresných testov. Piata kapitola popisuje implementáciu navrhnutého optimalizačného prostredia. Šiesta kapitola popisuje, graficky znázorňuje a analyzuje namerané výsledky optimalizácie sady regresných testov. Posledná kapitola, záver, zhrňuje získané výsledky a naznačuje ďalšie možné pokračovanie výskumu.

Kapitola 2

Teoretický úvod

Témou tejto diplomovej práce je návrh evolučného algoritmu na redukciu počtu testovacích vektorov regresných testov získaných v procese funkčnej verifikácie. V kapitole teoretický úvod je popísaná teória súvisiaca so zadaným problémom. Je vysvetlený pojem funkčná verifikácia, jej princíp, význam a vlastnosti. Ďalej sa kapitola venuje regresným testom, ktoré budú získavané v procese funkčnej verifikácie. Poslednou popísanou oblasťou sú evolučné algoritmy bežne používané k riešeniu rôznych optimalizačných problémov a v tejto práci budú využité pre redukciu počtu testovacích vektorov regresného testu.

2.1 Funkčná verifikácia

Funkčná verifikácia je prostriedok na zisťovanie, či bude vybraný hardvérový obvod (angl. *Device Under Test*, DUT) fungovať tak, ako je definované v jeho špecifikácií. Bežne pozostáva z nasledujúcich štyroch častí [11]:

1. definovanie požadovaného správania obvodu v podobe špecifikácie,
2. zistenie, ako DUT v skutočnosti funguje formou **simulácie**,
3. porovnanie, či sa požadované a skutočné správanie DUT zhodujú,
4. určenie stupňa istoty verifikačného procesu.

Funkčná verifikácia je pomerne nová oblasť a aj preto je stále predmetom ďalšieho výskumu. Jej zavedenie vzniklo ako dôsledok zväčšovania komplexnosti vytváraných obvodov, na ktoré nestačili vtedajšie verifikačné prostriedky. Dnes už nedostatočným predchodcom funkčnej verifikácie sú prototypy. Prototypy testovali správnu funkčnosť DUT až po tom, čo bol DUT vyrobený. Na takéto testovanie však musel prototyp spĺňať určité podmienky. Muselo byť možné sledovať vnútorné operácie prototypu z dôvodu umožnenia nájdenia chyby. Ďalej bolo nevyhnutné, aby mohli byť prototypy modifikované kvôli odstráneniu chýb a navyše počet chýb nemohol byť príliš vysoký. Dnešné hardvérové obvody však obsahujú milióny hradiel a tak ich verifikácia pomocou prototypov nie je možná ako z finančného pohľadu, tak ani z časového. Moderné obvody teda vyžadujú testovaciu metódu, ktorá je použitá pred tým, ako je vyrobený prototyp. Toto je cieľom funkčnej verifikácie [11].

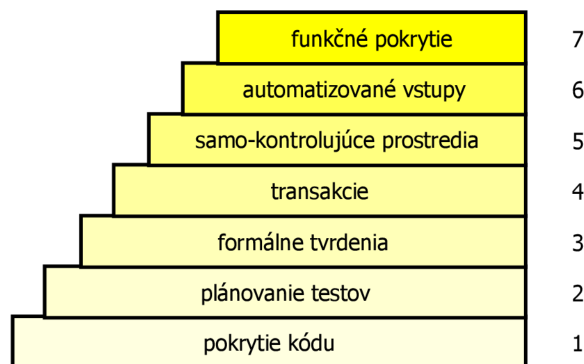
V oblasti funkčnej verifikácie v súčasnosti existuje mnoho heuristík, ktoré sú využívané v odlišných mierach. Využívanie týchto heuristík môže viesť ku kvalitnému verifikačnému

prostrediu (angl. *testbench*). Kvalitné verifikačné prostredie vykonáva nasledujúce tri činnosti [16]:

1. generuje vlastné vstupné testovacie dáta, aby užívateľ nemusel ručne písať veľa testov,
2. samo kontroluje výsledky DUT, aby overilo či pracuje správne,
3. zistí a ohlási chyby alebo stav, keď je DUT kompletne zverifikovaný.

Proces verifikácie je ukončený po tom, ako je DUT kompletne zverifikovaný (odsimulovaný) a nie sú nájdené žiadne chyby. Posudzovanie kompletnosti zverifikovania DUT je však jedna z najväčších výziev funkčnej verifikácie, pretože vyskúšanie všetkých kombinácií možných hodnôt vstupov a ich sekvencií, pozícií v pamäti atď. je náročné. Obvykle sa tento problém rieši nadefinovaním všetkých možných rozdielnych **logických** stavov, ktoré môžu nastať, a meraním pokrytia (angl. *coverage*) týchto stavov počas verifikovania. V kvalitnom verifikačnom prostredí sa pokrytie meria automaticky [16].

Verifikačný inžinier Ray Salemi v [16] popísal sedem krokov, ktoré treba splniť, k vytvoreniu kvalitného verifikačného prostredia. Tieto kroky sú znázornené na obrázku 2.1. Kroky sú zoradené nie podľa poradia, v akom by sa mali vykonávať počas procesu verifikácie DUT, ale v akom poradí je pre užívateľov najprirodzenejšie sa ich naučiť využívať.



Obrázek 2.1: Sedem krokov vedúcich ku kvalitnému verifikačnému prostrediu podľa Salemiho [16].

V ďalšom texte budú uvedené kroky popísané podrobnejšie, pretože sa s nimi čitateľ stretne aj v nasledujúcich kapitolách.

1. Prvým Salemiho krokom vedúcim ku kvalitnému verifikačnému prostrediu je meranie pokrytia kódu. Pokrytie kódu udáva, aká časť zdrojového kódu bola zverifikovaná. Jedná sa o jednoduchú a ľahko merateľnú mieru, ktorá nevyžaduje zmenu prístupu a navyše ju obvykle dokáže zmerať aj simulátor. Existuje niekoľko druhov merania pokrytia kódu, v závislosti na tom, čo sa štatisticky vyhodnocuje. Môže sa napríklad počítať percento vyskúšaných spustiteľných príkazov, alebo percento prejdených vetiev, či percento prejdených vetiev pre všetky možné príčiny a iné.
2. Druhým krokom je plánovanie testov. Tento krok urýchľuje písanie testov a navyše napomáha zverifikovaniu všetkých potrebných vlastností skôr, než je obvod vyrobený. Čo najlepšie zverifikovanie pred výrobou obvodu je dôležité, pretože nezverifikované časti môžu obsahovať chyby a každá z týchto chýb môže po výrobe zabrať hodiny

alebo dokonca až týždne na opravu. Plánovanie testov sa môže zdať na prvý pohľad ako práca navyše, no skutočnosť je presne opačná.

V procese verifikovania určitého DUT by malo byť plánovanie testov prvým krokom a pozostávať z ďalších troch podkrokov:

A. spísanie zoznamu rozhraní a funkcionality DUT,
B. popísanie vstupných dát a im odpovedajúcich výstupov,
C. vytvorenie zoznamu testov, pričom tento zoznam obsahuje len dvojice vstupov a výstupov, nerieši implementáciu či spôsob kontroly, slúži na orientačné zistenie, koľko úsilia bude treba na verifikáciu vynaložiť.

Výsledkom plánovania testov je efektívnosť testovania s minimálnym počtom potrebných testov.

3. Tretím krokom sú formálne tvrdenia (angl. *assertions*). Formálne tvrdenia sú kontrolnými bodmi, ktoré sú simulované paralelne s DUT a po zachytení chyby okamžite informujú užívateľa. Takýmto spôsobom sa chyby nemusia hľadať spätne až od výstupných pinov. Formálne tvrdenia tým znižujú čas hľadania chýb až na polovicu. Existujú jednoduché ale aj viac-cyklové formálne tvrdenia, ktoré kontrolujú napríklad spĺňanie protokolu. Ďalej sa zvykne kontrolovať aj korektné používanie blokov, či komunikácia medzi blokmi. Existujú formálne tvrdenia založené na automatoch. Dokonca boli pre formálne tvrdenia vyvinuté špeciálne jazyky ako napríklad PSL (angl. *Property Specification Language*) a SVA (angl. *SystemVerilog Assertions*).

4. Po použití prvých troch krokov má tím testovací plán, meranie pokrytia kódu a spôsob, akým sa dá proces hľadania chýb skrátiť na polovicu. Mnoho tímov v tomto bode končí, pretože je to pre nich dostatočné. Niektoré iné tímy však vidia, že iba presunuli úzke hrdlo z procesu hľadania chýb do písania testov, pretože musia otestovať príliš veľké množstvo možných prípadov a preto potrebujú nástroj, ktorý im uľahčí písanie testov a kontrolu výsledkov. Odpoveďou na tieto problémy je štvrtý Salemiho krok, transakcie.

   Transakcie zjednodušujú písanie testov a v neskorších krokoch umožňujú vytvorenie verifikačného prostredia, ktoré si samo generuje verifikačné vstupy a kontroluje výsledky. Čo sú to teda transakcie? Sú to obálky, ktoré zaobaľujú zložitú komunikáciu na báze signálov do modulov. Transakcie skrývajú pred užívateľom zložité komunikačné detaily a tak umožňujú písanie komplexných testov bez nutnosti vyvíjať úsilie na nastavovanie hodnôt jednotlivých pinov v správnych časoch. Transakcie taktiež zjednodušujú znovu použitie častí verifikačného prostredia.

   Použitie transakcií vo verifikácií má aj výraznú podporu zo strany priemyslu. Vo verifikačných jazykoch nájdeme pre ne predurčené dátové štruktúry. Napríklad UVM, ktoré bude popísané v neskôr, implementuje transakcie ako objekty, ponúka u nich možnosť využitia vstavanej náhodnosti a nástrojov pre meranie pokrytia.

5. Piatym krokom ku kvalitnému verifikačnému prostrediu je vytvorenie samo-kontrolujúceho verifikačného prostredia. Pre realizáciu je potrebné do verifikačného prostredia dodať dva nové komponenty: prediktor (angl. *predictor*) a komparátor (angl. *comparator*), ktoré spolu tvoria referenčný modul (angl. *scoreboard*). Schéma jednoduchého samo-kontrolujúceho verifikačného prostredia je znázornená na obrázku 2.2. Z obrázku

vidno, že verifikačné prostredie celkovo obsahuje niekoľko základných členov (vrátane nových):

`prediktor`: pre výpočet očakávaných výsledkov zo vstupných dát,

`komparátor`: porovnávajúci výsledky prediktoru a DUT,

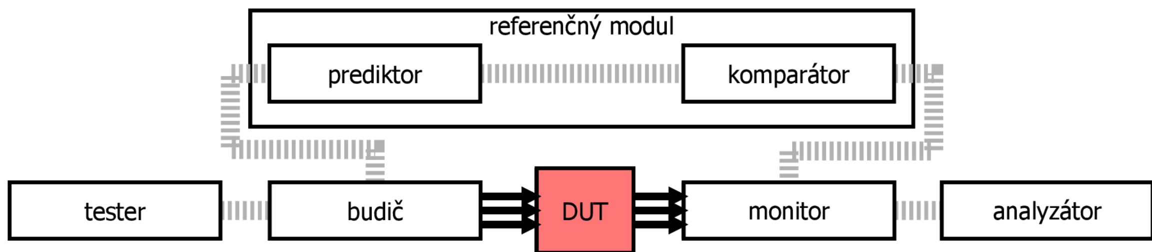
`budič` (angl. *driver*): slúžiaci na predávanie vstupných dát do DUT a do prediktoru,

`monitor` (angl. *monitor*): na získavanie výsledkov DUT,

`tester` (angl. *tester*): pre posielanie vstupných dát do verifikačného prostredia pre zverifikovanie DUT,

`analyzátor` (angl. *subscriber*): vypisujúci dosiahnuté výsledky, či už v počte chýb alebo množstve pokrytia.

Šedé čiarkované spojovníky na obrázku predstavujú tok transakcií ukladaných do front.



Obrázek 2.2: Schéma jednoduchého samo-kontrolujúceho verifikačného prostredia [16].

6. Pri verifikovaní je potrebné vyskúšať všetky významné vlastnosti DUT, no pri dnešnej komplexnosti hardvérových obvodov je ich počet obrovský. Preto ručné písanie testov (angl. *directed testing*) je časovo náročné a môže trvať aj niekoľko dní. Z toho dôvodu bol zavedený šiesty krok v tvorbe verifikačných prostredí a to automatické generovanie vstupov. Vstupy, na ktorých má byť DUT verifikovaná sú teda generované automaticky, aby ich nemusel písať človek. Zvyšok verifikačného prostredia je rovnaký ako na obrázku 2.2, vstupy sú preposielané do DUT a prediktoru a komparátor porovná správnosť výsledkov. Pri automatickom generovaní vstupov sa využívajú metódy, ktoré generujú náhodné hodnoty. Vstupné verifikačné dáta sú teda náhodné (angl. *random stimulus*). Dnešné verifikačné jazyky už na generovanie náhodných hodnôt umožňujú použitie rozličných pokročilých techník.

   Príkladom pokročilej techniky je napríklad možnosť obmedzenia náhodnosti vstupných dát (angl. *constrained random stimulus*) v jazyku SystemVerilog. Účelom obmedzenia náhodnosti je obmedzenie generovania nelegálnych hodnôt, zvýšenie pokrytia a urýchlenie verifikácie napríklad zvýšením pravdepodobnosti vygenerovania krajných hodnôt [16].

7. Siedmym a zároveň posledným krokom ku kvalitnému verifikačnému prostrediu je funkčné pokrytie. Myšlienka za týmto krokom je tá, že aj dosiahnutie 100 % pokrytia kódu nemusí nutne znamenať kompletné zverifikovanie DUT. Inými slovami, to že bol nejaký blok kódu pokrytý neznamená, že bola skontrolovaná aj jeho funkcionalita. Pokrytie kódu v skutočnosti ukáže len to, čo bolo vynechané, nie čo všetko bolo

skutočne zverifikované. Funkčné pokrytie kontroluje výsledky vzhľadom na zoznam vlastností, ktoré treba verifikovať. Pomocou funkčného pokrytia sa kontroluje, či bola zverifikovaná všetka funkcionalita definovaná v špecifikácií. V praxi sa oplatí využívať kombináciu funkčného i kódového pokrytia, pomocou ich kombinácie sa dajú odhaliť mŕtve časti kódu, teda kódu, ktorý sa už nepoužíva, alebo zistiť, že zoznam funkcionality pre funkčné pokrytie nie je kompletný.

Pri použití funkčného pokrytia sa zachytáva všetka funkcionalita do zoznamu a nad týmto zoznamom sa potom spustí simulácia, ktorá užívateľa informuje o tom, či boli zverifikované všetky položky zoznamu. Ostáva definovať, čo presne sú položky simulovaného zoznamu. Položky zoznamu sa označujú ako body pokrytia (angl. *coverpoints*) a môžu byť jednoduché i komplexné. Body pokrytia je možné rozdeliť na dva základné typy:

1. **signálové**: sa zameriavajú na prechody hodnôt signálov,
2. **dátové**: pozerajú sa na hodnoty premenných a počítajú koľkokrát bola nadobudnutá určitá hodnota alebo kombinácia hodnôt.

Plán testov založený na pokrytí je potom len veľmi dlhý zoznam signálových a dátových bodov pokrytia. Čo sa týka implementácie, body pokrytia sa realizujú napríklad pomocou formálnych tvrdení alebo tzv. skupín pokrytia (angl. *cover groups*) v jazyku SystemVerilog.

Všeobecne existuje niekoľko metód funkčnej verifikácie, metóda zakladajúca sa na sledovaní pokrytia sa volá verifikácia riadená pokrytím (angl. *coverage-driven verification*). Verifikácia riadená pokrytím sa iteratívne snaží dosiahnuť úplné pokrytie. Pri použití tejto metódy sa po každom behu simulácie zistí, ktoré položky zoznamu funkcionality k overeniu ešte neboli overené a na tie sa zameria pozornosť v ďalšom kroku.

2.1.1 SystemVerilog

SystemVerilog je jazyk slúžiaci na popis hardvérových obvodov (angl. *Hardware Description Language*, HDL) a zároveň obsahuje špeciálne prostriedky a konštrukcie jazyka pre ich verifikáciu (angl. *Hardware Verification Language*, HVL). História vzniku tohto jazyka siaha do deväťdesiatych rokov minulého storočia, keď veľkosť vytváraných hardvérových obvodov narástla tak, že sa vtedajšie prostriedky využívané na verifikáciu stali výrazne nedostačujúce. Dominantné HVL ako napríklad OpenVera a *e*, boli platené a navyše určené len pre verifikáciu (HVL ale nie HDL). Niektorí naopak pre verifikáciu využívali zdarma dostupné C++. Verifikační inžinieri boli takto nútení ovládať viac komplexných jazykov. Verilog, dovtedy najpoužívanejší HDL, bol pre verifikáciu nedostatočný a začal tak stagnovať v snahe stať sa priemyselným štandardom. Táto stagnácia bola prekonaná až koncom deväťdesiatych rokov, keď začínajúca firma Co-Design odštartovala vytváranie jazyka, ktorý je dnes známy pod názvom SystemVerilog a je priemyselným štandardom. Výhody existencie jedného jazyka pre popis obvodov i verifikáciu sú značné, ale toto spojenie zároveň prináša zvýšenie komplexnosti výsledného jazyka [18]. Zvyšok tejto podkapitoly podrobnejšie popisuje jazyk SystemVerilog.

SystemVerilog je postavený nad jazykom Verilog pridaním abstraktných jazykových konštrukcií zameraných na zjednodušenie verifikácie. Jeden z kľúčových prídavkov je umožnenie definovania tried. Triedy v jazyku SystemVerilog umožňujú aplikáciu techník objektovo orientovaného programovania do verifikácie. Rovnako, ako v iných objektovo orientovaných jazykoch, napríklad v C++ a v Jave, v jazyku SystemVerilog sú definície tried

šablónami pre objekty vytvárané v pamäti. Po vytvorení objekt pretrváva v pamäti pokiaľ existujú referencie odkazujúce sa naň, po odstránení príslušných referencií je objekt odstránený automaticky. Pri automatickom odstránení je využívaný tzv. *garbage collector*. Šablóna triedy definuje jej členov, ktorými môžu byť buď premenné alebo metódy. Metódy sa v jazyku SystemVerilog delia na funkcie a úlohy (angl. *tasks*). Rozdiel medzi funkciou a úlohou je v tom, že funkcia sa vykoná v nulovom simulačnom čase a úloha môže mať definované určité časové trvanie [10].

Triedy sú označované za dynamické objekty, pretože môžu vznikať a zanikať počas behu simulácie. Naproti tomu modul je statickým objektom, ktorý je prítomný hneď na začiatku simulácie. Pretože objekty tried musia byť vytvorené skôr, ako existujú v pamäti, je v jazyku SystemVerilog pri vytváraní verifikačného prostredia nutné konštrukciu hierarchie tried iniciovať z modulu. Z uvedených dôvodov taktiež vyplýva, že triedy nemôžu obsahovať moduly.

Ako bolo popísané v predošlom texte, jazyk SystemVerilog ponúka niekoľko výhod. Uvedenými výhodami sú objektovo orientovaný prístup a jednotný jazyk pre popis aj verifikáciu komponentov. Ďalšími dôležitými prínosmi jazyka SystemVerilog je možnosť vytvorenia spoľahlivých a ľahko rozšíriteľných verifikačných prostredí, ktoré navyše môžu byť znovupoužité medzi viacerými projektmi [18]. Vďaka týmto výhodám je SystemVerilog úspešný a sú nad ním rozvíjané mnohé verifikačné metodiky. Veľké zásluhy v tejto oblasti majú najmä spoločnosti a užívatelia združení pod organizáciou Accellera. Výsledkom ich činnosti sú napríklad metodiky OVM a UVM, ktoré prinášajú do procesu verifikácie viaceré výrazné vylepšenia.

2.1.2 UVM

UVM (angl. *Universal Verification Methodology*) je prostriedok na vytváranie verifikačných prostredí v jazyku SystemVerilog. Je určený pre komponenty, ktoré sú popísané v jazykoch SystemVerilog a Verilog, ale aj pre komponenty vo VHDL alebo SystemC. Táto podkapitola sa bude ďalej podrobnejšie venovať UVM a je založená na zdrojoch [4, 10]. V nasledujúcom texte sú vymenované základné myšlienky UVM a popísaná štruktúra typického vytváraného verifikačného prostredia. Popis UVM začne stručnou históriou.

UVM vznikla po úspešnej spolupráci pri vývoji OVM (angl. *Open Verification Methodology*) medzi Mentor Graphics a Cadence. OVM je teda predchodca UVM, ktorá dosiahla veľkú podporu zo strany priemyslu. OVM zlúčila ďalších výrobcov a používateľov pod organizáciu Accellera, aby vytvorili UVM. UVM je teda Accellera štandardom.

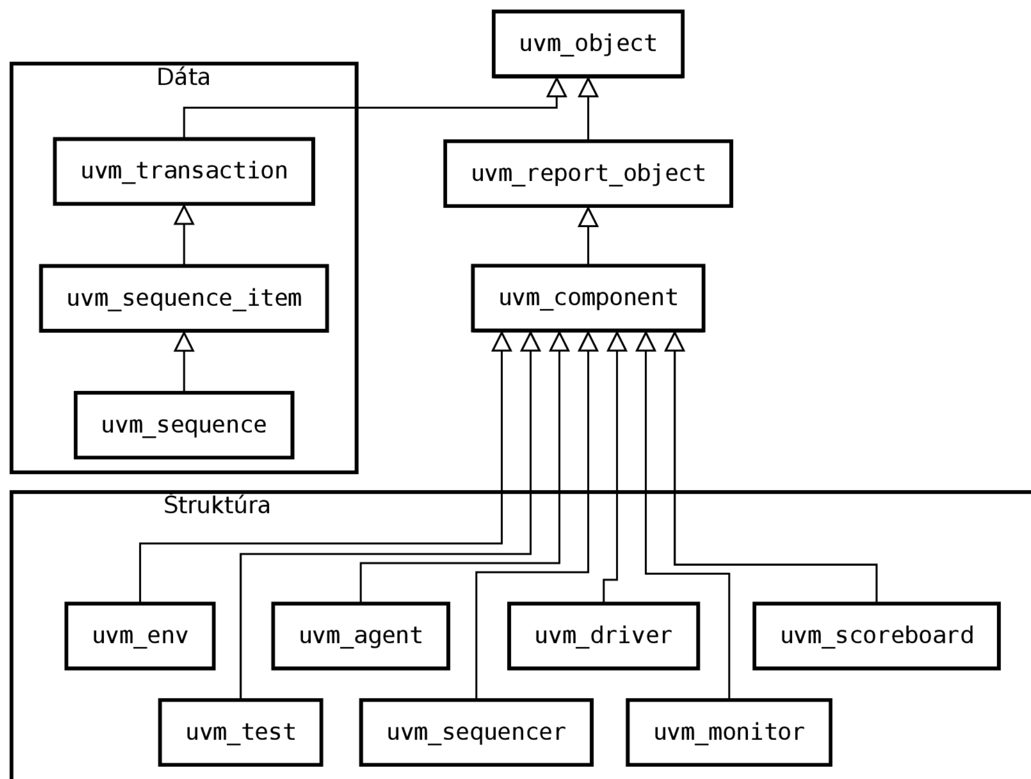
UVM sa šíri ako voľne dostupná (angl. *open source*) knižnica nad jazykom SystemVerilog a je rozširovaná pod Apache licenciou. UVM obsahuje základné triedy, ktoré uľahčujú vytváranie štruktúrovaných verifikačných prostredí. UVM je možné využívať na každom SystemVerilog IEEE 1800 simulátore a je takmer spätne kompatibilná s OVM.

Do budúcnosti sa dá predpokladať rozširovanie UVM medzi ďalších výrobcov a to vďaka jeho početným výhodám. Medzi výhody a hlavné myšlienky UVM patria nasledujúce:

- automatické generovanie vstupných dát,
- verifikácia riadená pokrytím (angl. *coverage-driven verification*),
- obmedzenie náhodnosti vstupných dát (angl. *constrained-random stimulus*) za účelom zvýšenia pokrytia,
- ľahká rozšíriteľnosť a možnosť konfigurácie vytváraných verifikačných prostredí,

- znovupoužiteľnosť zdrojových kódov ako v rámci toho istého projektu (angl. *vertical reuse*), tak aj medzi rozdielnymi projektmi (angl. *horizontal reuse*),
- oddelenie testovacích prípadov (angl. *test-cases*) od pevného verifikačného prostredia,
- komunikácia medzi komponentmi pomocou transakcií (angl. *Transaction-level Communication*, TLM), ktorá zvyšuje abstrakciu a zjednodušuje znovupoužiteľnosť,
- vrstvené sekvencie vstupných dát, ktoré umožňujú koordináciu viacerých rozhraní,
- štandardizovaný systém reportovacích správ,
- možnosť sledovania hodnôt registrov.

Značná časť výhod UVM ťaží zo štruktúry verifikačného prostredia, ktorá je založená na triedach. Hierarchia tried UVM je znázornená na obrázku 2.3. Niektoré dôležité triedy z obrázka sú popísané v ďalšom texte.

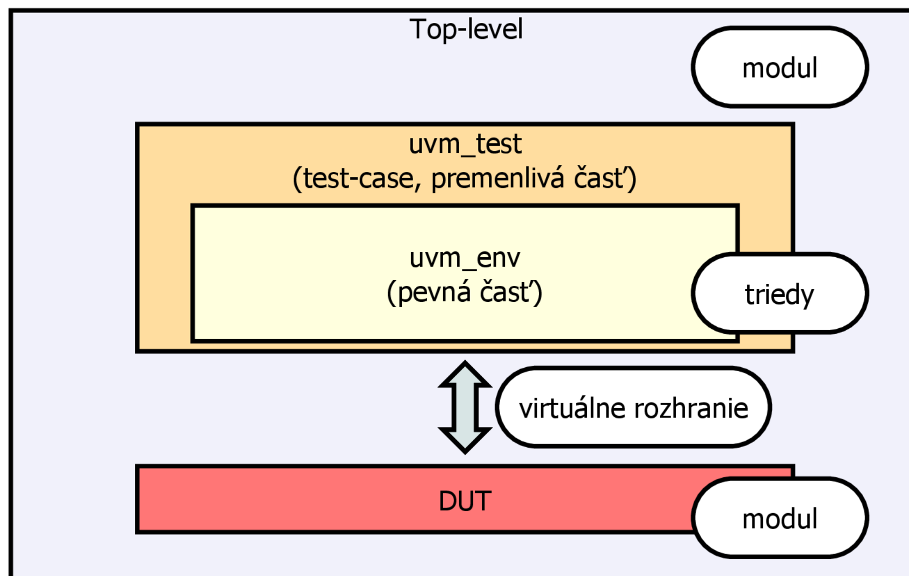


Obrázek 2.3: Hierarchia tried v UVM [5].

Knižnica UVM obsahuje tri hlavné typy tried:

1. všeobecné objekty vytvorené dedičnosťou z nadradenej triedy `uvm_object`,
2. transakcie dediace z `uvm_transaction`,
3. komponenty dediace z `uvm_component`.

Objekty slúžia ako dátové štruktúry pre konfiguráciu verifikačného prostredia. Transakcie sú využívané pri generovaní a analýze vstupných dát. Komponenty sa používajú na vytváranie triedne založeného verifikačného prostredia, ktoré má v UVM hierarchickú štruktúru.



Obrázek 2.4: Štruktúra verifikačného prostredia v UVM [6].

Na vrchole hierarchie verifikačného prostredia UVM je vždy jeden modul (angl. *Top-level*), ktorý obsahuje verifikovanú jednotku (DUT) a pripojenia k nej cez virtuálne rozhranie. Štruktúru najvrchnejších častí hierarchie verifikačného prostredia je možné vidieť na obrázku 2.4. Modul na vrchole hierarchie taktiež obsahuje inicializačný blok volajúci metódu UVM `run_test`. Táto metóda štartuje vykonávanie UVM fáz, ktoré slúžia na realizáciu činností verifikačného prostredia v správnom poradí. UVM fázy budú bližšie rozobraté neskôr v tejto podkapitole. Ďalšie odseky sa podrobne venujú popisu pevnej časti verifikačného prostredia `uvm_env`. Variabilný `uvm_test` je popísaný ako posledný.

Ako už bolo spomenuté, ďalšie prvky hierarchie verifikačného prostredia sú postavené z tried dediacich z `uvm_component`. Hierarchia verifikačného prostredia je definovaná sledom vzťahov typu *obsahuje*. Inými slovami, hierarchia definuje, aké komponenty sú obsadené v iných komponentoch. Existujú dve hlavné triedy komponentov, ktoré sú zároveň tzv. kontajnermi, teda obsahujú ďalšie komponenty. Týmito dvoma typmi kontajnerov sú:

1. agent dediaci z `uvm_agent`,
2. prostredie dediace z triedy `uvm_env`.

Pričom prostredie môže obsahovať niekoľko agentov. Úlohou kontajnerov je to, že výrazne uľahčujú znovupoužitie častí verifikačného prostredia.

UVM agent zastupuje zoskupenie verifikačných komponentov určených k práci s jedným logickým rozhraním DUT. Účelom agenta je poskytnutie verifikačnej komponenty, ktorá umožní užívateľom generovať a monitorovať dáta na pinoch DUT. Presná štruktúra agenta závisí na jeho konfigurácií uloženej v konfiguračnom objekte. Štruktúra agenta sa môže líšiť pri jeho použití v rozdielnych testoch.

Medzi komponenty obsiahnuté v agentovi najčastejšie patria komponenty z nasledujúcich tried:

uvm_sequencer: jednotlivé preposielanie generovaných vstupných testovacích transakcií, ktoré sú pôvodne združené v sekvenciách, do zvyšku verifikačného prostredia,

uvm_driver: prevod prijatých vstupných transakcií od uvm_sequencer na signály pre piny DUT,

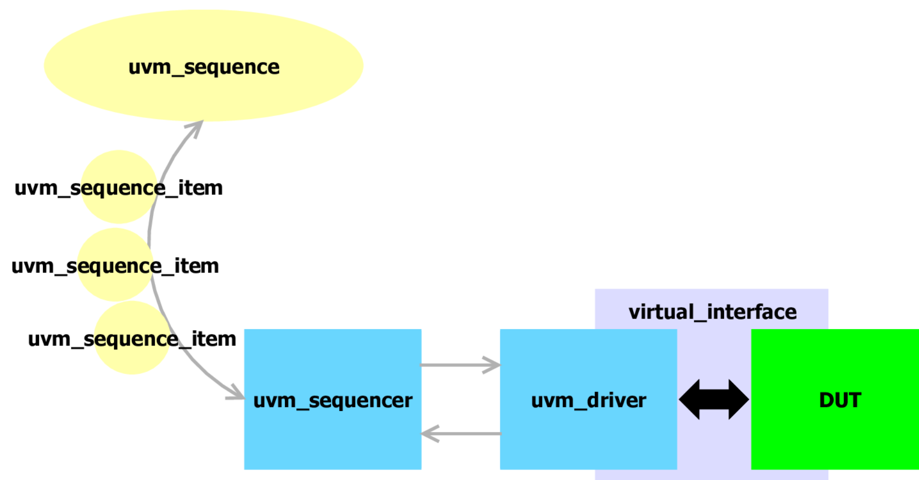
uvm_monitor: pozorovanie a konvertovanie signálov z pinov DUT spätne na transakcie, ktoré sú následne preposielané tzv. analytickým komponentom.

Agent obvykle taktiež obsahuje objekty a transakcie:

uvm_sequence_item: trieda reprezentuje jednotlivé transakcie, ktoré sa zlučujú do sekvencií transakcií uvm_sequence.

konfiguračný objekt: kontajner využívaný na predávanie informácií agentovi, ktoré ovplyvňujú vytváranie, prepájanie a činnosť príslušného agenta a jeho podkomponent; konfiguračný objekt navyše obsahuje referenciu na virtuálne rozhranie k pinom DUT, ktoré je dôležité pre komponenty tried uvm_driver a uvm_monitor.

Aby mohol čítateľ lepšie pochopiť účel a funkcionalitu väčšiny uvedených komponentov, objektov a transakcií kontajneru agent, nadsledujúci text podrobnejšie vysvetluje proces generovania a dodania pseudo-náhodných testovacích vstupov do DUT. Ako už bolo povedané, dodanie generovaných vstupov a zber výstupov z DUT sú totiž dve najhlavnejšie úlohy agenta.



Obrázek 2.5: Generovanie testovacích dát a ich distribúcia ku DUT [10].

Na obrázku 2.5 je znázornený princíp generovania testovacích dát a ich dodanie do DUT. Na procese sa podieľa uvm_sequencer, ktorý získava jednotlivé vstupné transakcie uvm_sequence_item zo sekvencie transakcií uvm_sequence. Komponent uvm_sequencer následne predáva každú získanú transakciu komponentu uvm_driver, ktorého úlohou je prevedenie dát v transakcií na signály pre piny DUT. Komunikácia medzi uvm_driver v agentovi

a DUT, ktoré sa nachádza v hierarchii verifikačného prostredia mimo agenta, prebieha pomocou virtuálneho rozhrania. Po vykreslení základného princípu generovania a dodania dát pre DUT, nájde čítateľ v ďalších odsekoch podrobnejší popis tohto procesu počínajúc vysvetlením virtuálneho rozhrania.

Virtuálne rozhranie je dynamická premenná, ktorá obsahuje odkaz na statickú inštanciu rozhrania DUT. Virtuálne rozhrania sú používané z dôvodu, že triedne založené verifikačné prostredie nemôže priamo odkazovať na porty DUT popísané v jazykoch Verilog, či VHDL. Namiesto toho je k portom DUT pripojená inštancia rozhrania z jazyka SystemVerilog a verifikačné prostredie komunikuje s DUT pomocou tejto inštancie. Verifikačné prostredie teda číta a zapisuje na piny DUT nepriamo, cez virtuálne rozhranie. V prípade viac logických rozhraní je doporučené použitie osobitného virtuálneho rozhrania pre každé.

Virtuálne rozhranie umožňuje napríklad zasielanie vstupných dát do DUT z komponentu triedy `uvm_driver`. Vstupné dáta sú generované sekvenciami vo forme transakcií. Sekvencie dedia z triedy `uvm_sequence` a vnášajú objektovo orientovaný prístup do generácie vstupných dát. Takýto prístup je veľmi flexibilný a prináša nové možnosti. Sekvencia je funktor (angl. *functor*), čo znamená, že je objektom využívaným ako metóda. UVM sekvencia obsahuje úlohu, teda metódu, ktorá môže trvať nejaký čas, nazývanú `body`. Táto metóda je spustená hneď po vytvorení sekvencie a používa sa na generovanie jednotlivých vstupných dát triedy `uvm_sequence_item` zasielaných komponentom triedy `uvm_driver` cez `uvm_sequencer`. Uvedená metóda `body` môže byť ale taktiež využitá na vytvorenie a spustenie ďalších sekvencií. Sekvencia je vytvorená pri jej použití a po prebehnutí je vymazaná. Životnosť sekvencie je teda kratšia než doba simulácie.

Fakt, že prvky tried `uvm_sequence` a `uvm_sequence_item` sú objekty znamená, že je možné ľahko docieliť ich náhodnosť a tak vytvoriť zaujímavé vstupné dáta. To, že sekvencie a prvky sekvencií sú objekty, taktiež umožňuje jednoduchú manipuláciu s nimi. Tieto objekty však nie sú súčasťou hierarchie verifikačného prostredia z dôvodu ľahšieho vytvárania testov, čo však prináša nutnosť využívania komponenty `uvm_sequencer`. Týmto je ukončený ako popis procesu generovania a dodania testovacích dát do DUT, tak popis kontajneru typu agent.

Ďalším kontajnerom, je **UVM prostredie** dediace z triedy `uvm_env`. Prostredie spája podkomponenty, ktoré môžu byť buď kontajnery typu agent alebo niektoré ďalšie komponenty:

konfiguračný objekt: umožňuje nastavenie, ktoré podkomponenty prostredia majú byť vytvorené, tento objekt by mal navyše obsahovať odkazy na konfiguračné objekty každého podkomponentu,

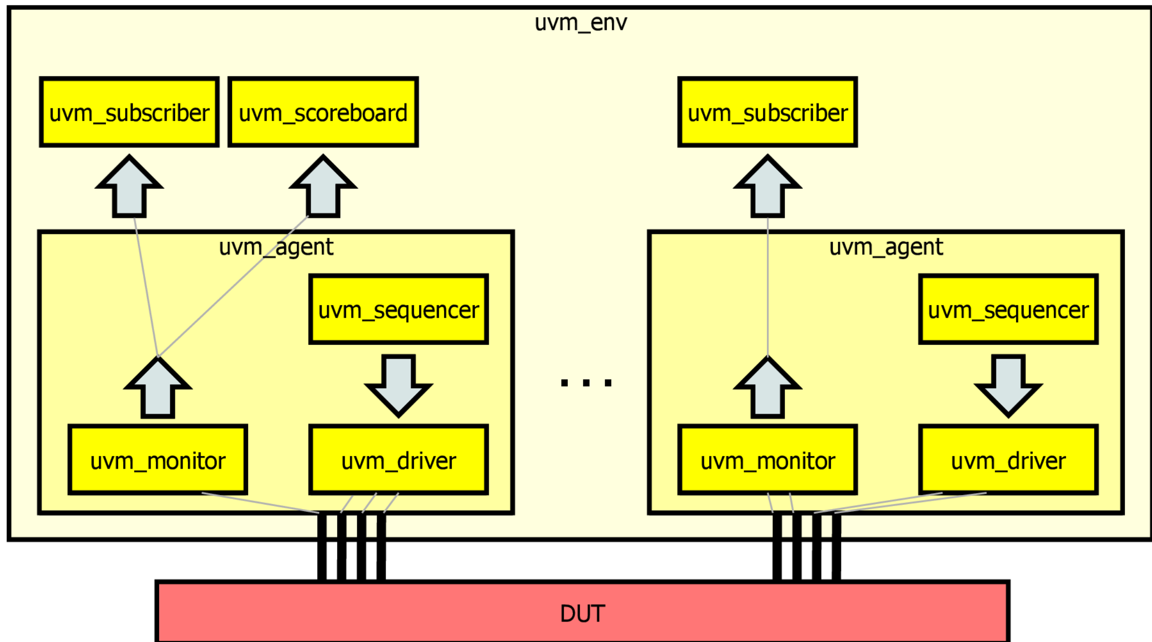
`uvm_subscriber`: analytický komponent pre vyhodnocovanie funkčného pokrytia,

prediktor: komponent slúžiaci na počítanie očakávaného výsledku DUT, obvykle sa spája s inými komponentmi ako napríklad `uvm_scoreboard`,

`uvm_scoreboard`: analytický komponent porovnávajúci očakávané výstupy a výstupy DUT zprostredkované cez `uvm_monitor`; na základe zhodnosti sa určí správnosť verifikovanej jednotky.

Príklad hierarchickej štruktúry prostredia je znázornený na obrázku 2.6. Okrem podkomponentov prostredia je na obrázku vidno aj typické zloženie agenta.

Táto podkapitola sa doposiaľ venovala vytvoreniu nemennej zložky UVM prostredia triedy `uvm_env` zloženej z hierarchicky usporiadaných komponentov. Pre vytváranie konkrétnych testov (*test-case*) sa využíva rozširovanie triedy `uvm_test`. Testy počas svojho



Obrázek 2.6: Príklad štruktúry UVM kontajnerov prostredie a agent [7].

behu vytvárajú a spúšťajú požadované sekvencie, čím definujú na aké druhy vstupných dát má byť DUT zverifikované. Príklad štruktúry verifikačného prostredia obsahujúceho test bol znázornený na obrázku 2.4. Z obrázku vidno, že testy obsahujú prostredia a prostredia potom obsahujú agentov, ako bolo ukázané na obrázku 2.6.

Na odštartovanie behu verifikácie v rámci vytvoreného verifikačného prostredia je potrebné zavolať metódu `run_test()` zo statickej časti, ktorou je obvykle inicializačný blok modulu na vrchole hierarchie (`Top-level`). Zavolanie uvedenej metódy spôsobí spustenie jednotlivých fáz verifikácie. Táto metóda očakáva ako vstupný argument názov testovacieho prípadu (angl. *test-case*), ktorý má byť spustený.

Na záver kapitoly o UVM je ešte dôležité zmieniť ďalšiu zaujímavú súčasť funkcionalitty UVM, ktorou sú fázy. Fázy slúžia na zaistenie konzistentného poradia vykonávania hlavných krokov verifikačného prostredia počas simulácie. UVM fázy sa delia do troch hlavných skupín, ktoré sú vykonávané v nasledujúcom poradí:

konštrukčné fázy: (angl. *build phases*) počas týchto fáz je vytvorené a konfigurované prostredie, vytváranie jednotlivých komponentov hierarchie sa vykonáva v poradí zhora dole, prepájania komponentov naopak prebieha zdola hore,

verifikačné fázy: (angl. *run-time phases*) fázy trvajúce nenulový čas, tieto fázy slúžia na priamu komunikáciu verifikačného prostredia s DUT,

ukončovacie fázy: (angl. *clean up*) slúžiace na zber a vyhodnocovanie výsledkov.

UVM prináša niekoľko nových technológií týkajúcich sa fázovania. K novým technológiám patrí napríklad možnosť skákania dopredu či dozadu k jednotlivým fázam, pridávanie nových, užívateľom definovaných fáz a vzťahov medzi nimi a iné.

2.2 Regresné testy

Uchovávanie testov, ktoré pokrývajú a testujú funkcionalitu softvérového programu alebo hardvérového obvodu a ich opätovné spúšťanie v prípade modifikácii kódu tohto programu alebo obvodu, je známe pod názvom regresné testovanie. Regresné testovanie je používané z dôvodu, že modifikácia existujúcich zdrojových súborov je veľmi náchylná na zavádzanie nových chýb, dokonca náchylnejšia než napísanie úplne nového kódu. Regresné testy sú teda spúšťané po zmenách v kóde, aby bolo overené, že zavedené zmeny neporušili (angl. *regress*) funkcionalitu programu alebo obvodu [13].

Modifikácia kódu je typicky spojená s vylepšovaním, optimalizáciou, opravou chýb a mazaním alebo rozširovaním existujúcej funkcionality. Jedná sa teda o pomerne časté zmeny, ktoré môžu spôsobiť chyby. Regresné testovanie je preto nevyhnutnou súčasťou vývojového procesu, ale zároveň obvykle nákladné z dôvodu veľkého počtu testov [14].

Ako bolo popísané v predošlých kapitolách, pri verifikácií funkcionality DUT pomocou UVM sú generované pseudo-náhodné vstupné dáta až pokiaľ sa nepodarí overiť celú funkcionalitu DUT. Vygenerované vstupné dáta je možné uložiť a použiť pre regresné testovanie namiesto neustáleho generovania náhodných vstupov pri každom spustení verifikácie. Vygenerovaných vstupov je však veľký počet a sú redundantné, čo má za následok zbytočne drahý beh regresných testov. Na druhej strane uloženie a znovupoužitie tých istých vstupov so sebou nesie veľkú výhodu a to konkrétne možnosť ich offline editácie. Ako užitočné sa tu ponúka zníženie redundancie a tým aj zníženie ceny opätovného spúšťania testov.

2.3 Evolučné algoritmy

Celá rada úloh umelej inteligencie sa rieši pomocou techník hľadania riešenia v priestore, ktorý reprezentuje všetky možné riešenia, prípadne všetky povolené riešenia danej úlohy. V prípade, že je tento priestor malý vzhľadom k dostupnej výpočetnej sile, tak sa za riešenie označí globálne optimum nájdené preskúmaním všetkých možností. Ak však je prehľadávaný priestor príliš veľký na to, aby bolo možné preskúmať všetky možnosti v rozumnom čase, je potrebné použiť nejakú heuristickú stochastickú metódu, napríklad evolučný algoritmus. Pod pojmom stochastický algoritmus sa rozumie algoritmus, ktorý je potreba spustiť niekoľkokrát a z jeho nezávislých behov je možné získať štatisticky významné dáta. Nasledujúci text sa opiera o zdroje [17, 12].

Termín evolučný algoritmus vznikol až začiatkom 90. rokov. Evolučné algoritmy zastrešujú stochastické prehľadávacie algoritmy, ktoré vznikali už dávnejšie a nezávisle na sebe. Spoločným rysom jednotlivých evolučných algoritmov je to, že nachádzajú inšpiráciu v Darwinovej evolučnej teórií a rôznych teóriách neodarwinizmu. Darwinova evolučná teória vraví, že u všetkých druhov v prírode platí, že jedinci, ktorí sa lepšie adaptujú na zmeny prostredia, majú vyššiu pravdepodobnosť prežitia a tým aj vyššiu pravdepodobnosť produkovať potomkov. Takýto proces sa opakuje celé generácie a ako výsledok viac prispôsobiví jedinci a gény väčšinou pretrvajú a neprispôsobiví jedinci vymiznú.

Príkladmi evolučných algoritmov sú genetické algoritmy, evolučné stratégie, evolučné programovanie a genetické programovanie. Využitie týchto algoritmov je najčastejšie na problémy optimalizácie, teda nájdenie optimálnych hodnôt parametrov optimalizovaného systému. V poslednej dobe sa však evolučné algoritmy využívajú aj pre úlohy návrhu, kde je okrem parametrov hľadaná aj vlastná štruktúra cieľového systému.

Klasické využitie evolučných algoritmov, problém optimalizácie, je v matematike definované ako úloha hľadania globálneho extrému funkcie: $f : A \to \mathcal{R}$, kde definičným oborom

je najčastejšie karteziánsky súčin uzavretých, často obmedzených, intervalov $[a_i, b_i]$ obvykle v reálnej, celočíselnej alebo binárnej doméne. Pokiaľ je cieľom nájsť globálne minimum (minimalizácia argumentu), potom sa hľadá také $x* \in A$, pre ktoré platí $f(x*) \leq f(x)$ pre všetky $x \in A$. Prvky množiny A sa nazývajú kandidátne riešenia. Samotná množina A potom predstavuje prehľadávaný priestor. Funkcia f sa v oblasti evolučných algoritmov nazýva fitness funkcia a slúži k ohodnoteniu kvality kandidátnych riešení. Fitness funkcia nemusí byť spojitá, mať deriváciu alebo byť úplne definovaná. Navyše často obsahuje lokálne extrémy. Lokálnym minimom resp. maximom sa nazýva každé x^-, resp. x^+ pre ktoré platí, že pre všetky $x \in A$, ktoré sa nachádzajú v jeho okolí definovanom parametrom δ (teda $\|x - x^-\| \leq \delta$ resp. $\|x - x^+\| \leq \delta$), je $f(x^-) \leq f(x)$ resp. $f(x^+) \geq f(x)$. Fitness funkcia je čiernou skrinkou, ktorá vracia funkčnú hodnotu pre zadaný argument. Je však výhodné, ak výpočet funkčnej hodnoty prebehne dostatočne rýchlo. V reálnych systémoch však býva fitness funkcia často zložitá a často sa jedná napríklad o nejakú simuláciu systému.

Pri optimalizácií sa niekedy využíva aj úloha maximalizácie argumentu, teda hľadanie globálneho maxima namiesto uvedeného globálneho minima. Úloha maximalizácie argumentu je však analogická k úlohe minimalizácie.

Nasledujúci text popíše podrobnejšie princíp jednotlivých evolučných algoritmov, no predtým, pre porovnanie, vysvetlí dva jednoduché stochastické algoritmy a to konkrétne náhodné prehľadávanie a horolezecký algoritmus.

Náhodné prehľadávanie, známe tiež ako algoritmus slepého prehľadávania, generuje v každom kroku náhodné kandidátne riešenie $a_i \in A$. Nové kandidátne riešenie si zapamätá v prípade, ak bolo jeho ohodnotenie lepšie, než ohodnotenie doteraz najlepšieho riešenia. Výpočet sa končí buď nájdením uspokojivého riešenia alebo vyčerpaním povoleného počtu krokov. Slepý algoritmus je pre riešenie reálnych problémov veľmi slabý, pretože neobsahuje stratégiu pre riešenie problémov a žiadnym spôsobom neuchováva a nevyužíva informácie získané v priebehu svojej činnosti.

Pseudokód 1 popisuje činnosť algoritmu náhodného prehľadávania, ktorý má za úlohu nájsť maximum funkcie f. Výpočet začína nastavením výstupného ohodnotenia uchovávaného v premennej `f_out` na minimálnu hodnotu. V každom kroku je pomocou funkcie *random* vygenerované nové kandidátne riešenie s ohodnotením $f(a)$. Pokiaľ je $f(a)$ vyššie, než doposiaľ najlepšie známe riešenie `f_out`, dôjde k prepísaniu hodnoty i ohodnotenia najlepšieho riešenia. Algoritmus sa ukončí vykonaním `t_max` krokov.

Algoritmus 1: Náhodné prehľadávanie [17]

```
náhodné_prehľadávanie()
    f_out = MIN; //nastaví ohodnotenie najlepšieho
    t = 0; //inicializácia počítadla krokov
    while t < t_max do
        t = t + 1;
        a = random();
        if f(a) > f_out then
            a_out = a; //uchovanie doterajšieho najlepšieho riešenia
            f_out = f(a); //uchovanie doterajšieho najlepšieho ohodnotenia
        end
    end
    return(a_out, f_out);
```

Vylepšením náhodného prehľadávania je **horolezecký algoritmus**. Horolezecký algoritmus, na rozdiel od náhodného prehľadávania, nové kandidátne riešenie nevyberá slepo. Za

nové kandidátne riešenie je zvolené také, ktoré má najvyššie ohodnotenie v okolí aktuálneho kandidátneho riešenia. Popísaný princíp je demonštrovaný pseudokódom 2.

Algoritmus 2: Horolezecký algoritmus [17]

```
horolezecký_algoritmus()
    a = random(); //náhodné vygenerovanie počiatočného riešenia
    a_out = a;
    f_out = f(a); //uchová ohodnotenie počiatočného riešenia
    t = 0; //inicializácia počítadla krokov
    while t < t_max do
        t = t + 1;
        b = vyber_najlepšieho(U(a)); //najlepšie riešenie v okolí bodu a
        if f(b) > f_out then
            a_out = b; //uchovanie doterajšieho najlepšieho riešenia
            f_out = f(b); //uchovanie doterajšieho najlepšieho ohodnotenia
        end
        a = b; //nastavenie nového kandidátneho riešenia
    end
    return(a_out, f_out);
```

Výrazná nevýhoda horolezeckého algoritmu je tá, že je náchylný k uviaznutiu v lokálnom extréme. Predídenie uviaznutiu je možné zväčšením okolia, v ktorom sa hľadá nové kandidátne riešenie. Zväčšením okolia sa však taktiež zvýši výpočtetná náročnosť algoritmu, pretože algoritmus musí urobiť $C \times t_max$ ohodnotení, kde C je veľkosť skúmaného okolia a t_max je počet krokov algoritmu. V literatúre je možné nájsť aj niekoľko vylepšení horolezeckého algoritmu, tie však nie sú predmetom tejto práce.

Uvedené problémy náhodného prehľadávania i horolezeckého algoritmu riešia evolučné algoritmy, ktoré sú v súčasnosti asi najpopulárnejšou optimalizačnou technikou, pretože boli s úspechom požité v rôznych aplikačných doménach, kde väčšinou prekonali ostatné optimalizačné algoritmy.

Princíp evolučných algoritmov je popísaný pseudokódom 3. Na začiatku výpočtu je vytvorená počiatočná populácia obsahujúca dopredu zvolený počet kandidátnych riešení (jedincov). Počiatočná populácia môže byť vytvorená buď náhodne, alebo podľa nejakej heuristiky. V každom kroku algoritmu, ktorý nazývame generácia, sú všetci jedinci ohodnotení pomocou fitness funkcie. Pomocou hodnotenia je z jedincov vybraná množina rodičov, za týmto účelom vznikla rada selekčných algoritmov. Lepšie ohodnotení jedinci majú obvykle väčšiu šancu stať sa rodičmi. Aplikáciou genetických operátorov (najznámejšie sú kríženie a mutácia) nad množinou rodičov vznikne množina potomkov. Z množín rodičov a potomkov, prípadne i z predchádzajúcich populácií, sa následne vyberú jedinci pre novú populáciu. Pokiaľ je algoritmus dobre navrhnutý, celková priemerná hodnota fitness populácie sa bude zvyšovať. Prostredníctvom fitness funkcie vzniká selekčný tlak, ktorý vedie k prehľadávaniu do výhodnejších oblastí prehľadávaného priestoru. Použitie genetických operátorov, ani selekčných mechanizmov, nie je deterministické, ale je uplatnená užívateľom nastavená pravdepodobnosť. Podobne ako v predošlých uvedených stochastických algoritmoch, evolučných algoritmus je ukončený buď pri nájdení dostatočne kvalitného jedinca, alebo po vyčerpaní povoleného počtu generácií. Kvalitu evolučného algoritmu ovplyvňuje najmä spôsob zakódovania problému, konštrukcia fitness funkcie a použité genetické ope-

rátory.

Algoritmus 3: Evolučný algoritmus [17]

```
evolučný_algoritmus()
    t = 0;
    P(t) = vytvor_počiatočnú_populáciu
    ohodnoť P(t);
    while ukončovacia_podmienka == FALSE do
        Q(t) = vyber_rodičov(P(t));
        Q2(t) = vytvor_nových_jednicov(Q(t));
        ohodnoť (Q2(t));
        P(t + 1) = vyber_jedincov_do_novej_populácie(P(t), Q2(t));
        t = t + 1;
    end
    return(a_out, f_out);
```

Rovnako ako princíp evolučných algoritmov, aj pojmy boli prebraté z genetiky reálneho života. Pri popisovaní evolučných algoritmov je potrebné uviesť niekoľko takýchto pojmov:

chromozóm: zakódovanie jedinca populácie,

gény: zložky chromozómov,

genotyp: povolená kombinácia génov pre určitú vlastnosť tvoriacu chromozóm,

fenotyp: konkrétna charakteristika zakódovaná v chromozóme jedinca,

alela: konkrétna hodnota génu,

locus: pozícia génu v rámci chromozómu,

kríženie: spôsob generovania nových jedincov, ktorý kombinuje dvojice jedincov z existujúcej populácie,

mutácia: väčšina hodnôt génov je dedená po rodičoch tak, ako sú, avšak ojedinele sú hodnoty niektorých génov zmenené vplyvom určitých atypických udalostí a práve takáto zmena hodnôt génov sa nazýva mutácia.

Príkladom genetiky reálneho života je človek, ktorý vzniká po oplodnení vajíčka spermiou, pričom obe bunky nesú 23 chromozómov obsahujúcich tisíce až milióny génov. Gény matky a otca vytvárajú páry génov pre potomka. Každý jeden, či niekoľko génových párov, určuje nejakú charakteristiku (fenotyp) potomka, napríklad farbu očí, či krvnú skupinu. Príkladom chromozómu pre počítačový evolučný algoritmus môže byť binárny reťazec, gén je potom jeden bit a alela nadobúda buď hodnotu logickej nuly alebo jednotky. Najčastejšie sa pre chromozóm používa binárna, celočíselná alebo reálna reprezentácia. Obecne však môže mať chromozóm premennú dĺžku a obsahovať čokoľvek, čo počítač dokáže spracovať.

Každý prehľadávací algoritmus môže byť chápaný ako mechanizmus, ktorý prostredníctvom jedného či viac operátorov produkuje nové kandidátne riešenia z tých, ktoré už boli v priebehu jeho činnosti vyskúšané. Efektívne prehľadávacie mechanizmy by mali dobre balancovať medzi dvomi zdanlivo protichodnými cieľmi:

1. dôkladne preskúmať okolie doteraz najlepšieho nájdeného riešenia,
2. vydať sa do zatiaľ nepreskúmaných oblastí prehľadávacieho priestoru.

Náhodné prehľadávanie i horolezecký algoritmus spĺňali len jeden z týchto bodov. Kým horolezecký algoritmus dobre preskúmal len okolie zatiaľ najlepšieho riešenia, náhodné prehľadávanie skúma celý prehľadávací priestor, ale neprebáda podrobnejšie silné oblasti.

Každý prehľadávací priestor (problém) je iný. Len drobnou zmenou fitness funkcie, reprezentácie, či genetického operátoru, je možné dostať priestor úplne iného charakteru. Preto nie je možné prehlásiť jeden algoritmus za najvýhodnejší pre všetky prípady. Pre uvedený fakt existuje aj matematický dôkaz, tzv. *no free lunch* teorém. Podľa tohto teorému, žiadny prehľadávací algoritmus, ktorý navštívi každý bod priestoru najviac raz, nemôže byť v priemere efektívnejší než systematické prehľadanie všetkých možností. Z uvedeného vyplýva, že pri porovnávaní cez všetky možné optimalizačné úlohy je náhodné prehľadávanie rovnako dobré, ako evolučný algoritmus. Pokiaľ vykazuje algoritmus A vyššiu úspešnosť než B na určitej množine úloh, existuje iná množina úloh, na ktorej je naopak B úspešnejšie než A. Pre prax to znamená, že sa treba zamerať na vytvorenie kvalitného algoritmu pre daný problém a použiť čo najviac znalostí o danom probléme, za účelom vytvorenia čo najlepšieho algoritmu.

Medzi základné druhy evolučných algoritmov patria:

Evolučné stratégie vznikli v 60. rokoch 20. storočia v Nemecku s pôvodným účelom optimalizácie v oblasti aerodynamiky. Evolučné stratégie optimalizujú vektor reálnych parametrov a v základnej variante využívajú len operátor mutácie, ktorý modifikuje každý parameter vektoru. Ďalej sa využíva Gaussovo rozloženie s nulovou strednou hodnotou a rozptylom σ. V základnej variante sa z jedného rodiča vytvára jeden potomok pomocou mutácie pričítaním hodnôt vygenerovaných pomocou rozloženia. Ak má potomok väčšiu hodnotu fitness ako jeho rodič, stáva sa rodičom novej generácie, inak dôjde k novej mutácií rodiča. Špecifickým prvkom pre evolučné stratégie, ktorý sa typicky nevyužíva v žiadnych iných evolučných algoritmoch je samo-adaptácia, pri ktorej sú parametre evolúcie zakódované do chromozómu a evolučne sa menia spolu s parametrami hľadanej funkcie. V súčasnosti sa využívajú rôzne vylepšené varianty.

Evolučné programovanie bolo vyvinuté úplne nezávisle na evolučných stratégiách a to konkrétne o niekoľko rokov skôr v USA, no napriek tomu majú radu spoločných vlastností. Autorom je Lawrence Fogel. Evolučné programovanie typicky používa aplikačne špecifickú reprezentáciu problému, samo-adaptáciu a turnajovú selekciu, obvykle však nepoužíva kríženie.

Genetické programovanie vzniklo v 80. rokoch 20. storočia, dôležitou osobnosťou, ktorá sa zaslúžila o rozvoj, je John Koza. Algoritmus genetického programovania je v princípe rovnaký, ako všeobecný evolučný algoritmus, no genetické programovanie má niekoľko špecifík ako napríklad reprezentácia. Genetické programovanie pracuje so spustiteľnými štruktúrami s premennou dĺžkou. Najčastejšie sa jedná o programy. Ďalej, genetické programovanie má niekoľko svojich pokročilých genetických operátorov pracujúcich nad spustiteľnými štruktúrami. Prevedenie kódu spustiteľných štruktúr slúži k zisteniu hodnoty fitness.

Genetické algoritmy, ktoré sú bližšie popísané v nasledujúcej podkapitole.

2.3.1 Genetické algoritmy

Genetické algoritmy sú počítačové modely založené na genetike a evolúcií v biológií. Genetické algoritmy zahrňujú koncepty ako chromozómy, gény, párenie, kríženie, mutáciu za

účelom náhodnej zmeny, selekciu najlepších jedincov a evolúciu. Vďaka týmto technikám by mali genetické algoritmy nachádzať stále lepšie a lepšie riešenia problémov rovnako, ako sa druhy v prírode adaptujú zmenám prostredia. Genetické algoritmy sa využívajú na široké spektrum optimalizačných problémov, boli v praxi nasadené napríklad na rozvrhnutie obvodov alebo tzv. problém *job-shop* [15]. Ďalšie využitie genetických algoritmov je strojové učenie pre vytváranie funkcií slúžiacich napríklad na klasifikáciu alebo predikciu.

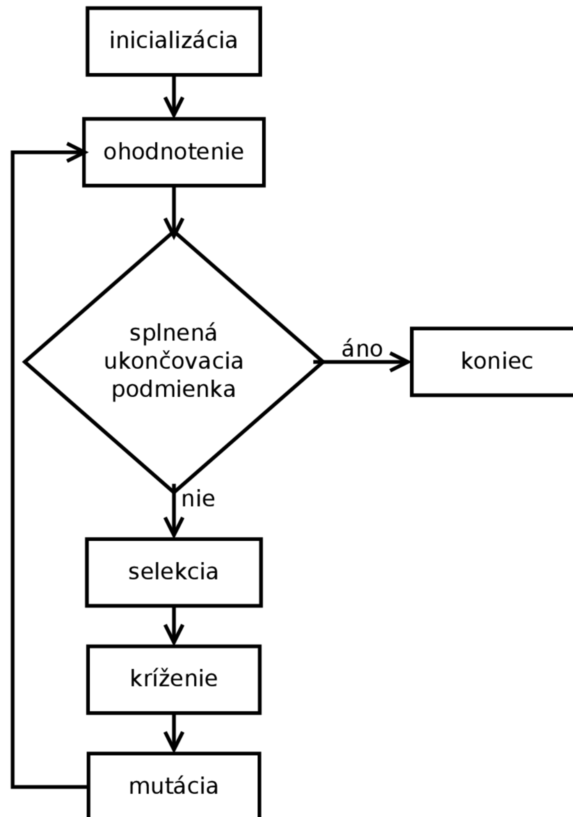
Genetický algoritmus bol navrhnutý v 70. rokoch 20. storočia Johnnon Hollandom, ale stal sa populárny až v 80. rokoch vďaka Hollandovmu žiakovi Davidovi Goldbergovi, ktorý ukázal použiteľnosť genetického algoritmu v rade náročných úloh.

Genetický algoritmus má v základnom princípe rovnakú štruktúru ako evolučný algoritmus. Na začiatku sú náhodne vygenerovaní jedinci(iným názvom kandidátne riešenia). Avšak, pokiaľ je známa nejaká informácia ohľadne rozloženia optimálneho riešenia, môže byť využitá pri konštrukcii počiatočných jedincov. Chromozómy jedincov u genetických algoritmov majú konštantnú dĺžku. Po vytvorení úvodnej populácie sa začnú iteratívne vykonávať kroky. Každý krok pozostáva z niekoľkých podkrokov: výber (selekcia) rodičov, fáza kríženia rodičov a príležitostné zmutovanie niektorých nových jedincov. Vďaka selekcií len dobrých riešení, počítačové genetické algoritmy dosiahnu lepších výsledkov než prirodzená evolúcia. Algoritmus je ukončený pri naplnení nejakej ukončujúcej podmienky, napríklad pri nájdení optimálneho riešenia (jedinca), alebo pri nezlepšovaní riešení určitý počet krokov. Základná štruktúra uvedeného genetického algoritmu je znázornená na obrázku 2.7. Keďže sa jedná o stochastický algoritmus, beh je vhodné zopakovať niekoľkokrát. Dokonca sa často stáva, že v skorších generáciách sú fitness hodnoty niektorých chromozómov lepšie, než fitness najlepšieho chromozómu po niekoľkých generáciách. Preto je niekedy nápomocné uchovávať najlepšie ohodnotený chromozóm.

Výber rodičov závisí od ohodnotenia jedincov na základe fitness funkcie. Pri výbere rodičov sa teda ako prvé vyhodnotia hodnoty fitness všetkých jedincov a na základe týchto hodnôt sú jedincom priradené ich pravdepodobnosti zvolenia za rodičov. Rodičia sú vyberaní náhodne ale v závislosti na priradených pravdepodobnostiach. Riešenia s vyšším ohodnotením majú vyššiu pravdepodobnosť prežiť do ďalších generácií. Výber jedincov musí teda na jednu stranu dostatočne uprednostňovať jedince s vyššou hodnotou fitness, na druhú stranu ale musí zaistiť dostatočnú rôznosť populácie.

Počet vyberaných rodičov záleží od toho, či je využívaný generačný variant evolučného algoritmu, inkrementálny [20] alebo prekrývanie populácií. Generačný variant vytvára nové populácie iba z novo vygenerovaných jedincov.Prekrývanie populácií tvorí nové populácie prekrývaním starých populácií s niekoľkými novo vygenerovanými jedincami. Parametrom genetického algoritmu je v prípade prekrývania populácií percento novo zaradených jedincov, tzv. prekrytie. Pri inkrementálnej variante (angl. *steady state*) je v novej populácií nahradený potomkom jediný jedinec pôvodnej populácie. Elitizmus v terminológií evolučných algoritmov znamená, že najlepší jedinec z predchádzajúcej populácie sa vždy dostane do novej populácie.

Nech počas iterácie t obsahuje genetický algoritmus ps kandidátnych riešení $G(t) = \{x_1^t, \ldots, x_{ps}^t\}$. Ďalej, nech hodnota fitness každého jedinca je rovná $eval(x_i)$, potom celková fitness hodnota populácie F je $F = \sum_{i=1}^{ps} eval(x_i)$, pravdepodobnosť selekcie každého chromozómu p_s^i je $p_s^i = eval(x_i)/F$ a konečne kumulatívna pravdepodobnosť p_{cum}^i pre každý chromozóm sa rovná $p_{cum}^i = \sum_{j=1}^{i} p_s^i$. Najzákladnejšia selekčná metóda, proporcionálna selekcia, využívajúca algoritmus koleso šťastia (ruleta), vyberá rodiča vygenerovaním ná-



Obrázek 2.7: Základná štruktúra genetických algoritmov [1].

hodného čísla r v rozsahu $\langle 0, 1\rangle$. Ak platí, že $r < p^1_{cum}$, vyberie za rodiča prvý chromozóm x_1, v opačnom prípade je vybratý chromozóm x_i, pre ktorý platí $p^{i-1}_{cum} < r < p^i_{cum}$. Z uvedeného postupu vidno, že najlepšie chromozómy budú vybraté niekoľkokrát, priemerné raz a najslabšie chromozómy, hoci majú šancu byť vybrané, pravdepodobne postupne odumrú.

Existuje niekoľko ďalších selekčných metód, niektoré z nich budú uvedené v ďalších odsekoch. Tieto metódy sa odlišujú selekčným tlakom. Čím väčší je selekčný tlak, tým rýchlejšie algoritmus konverguje, pretože populácia po selekcii obsahuje viac jedincov s vysokou hodnotou fitness. Súčasne však vzrastá nebezpečenstvo predčasnej konvergencie.

Najväčšou nevýhodou popísanej proporcionálnej metódy je degenerácia populácie v prípade, že má jeden jedinec výrazne vyššie ohodnotenie než ostatní. Riešením je algoritmus výberu podľa poradia jedincov. Tento algoritmus najskôr zoradí kandidátnych jedincov podľa hodnoty fitness. Pravdepodobnosť výberu je potom úmerná poradiu jedinca a nie jeho ohodnoteniu.

Algoritmus stochastického univerzálneho vzorkovania (angl. *Baker's Stochastic Universal Sampling*) rozdelí kruh na výseky, ktorých veľkosť je úmerná hodnotám fitness jedincov populácie, podobne ako koleso šťastia. Tentokrát sa však náhodné číslo generuje len raz. Selekcia rodičov sa dá predstaviť tak, ako roztočenie kolesa, popri ktorom je rozmiestnených toľko jazýčkov, koľko jedincov má byť vybraných [20].

Ďalšou selekčnou metódou je turnajová selekcia. Táto metóda vyvíja väčší selekčný tlak. Princíp metódy je náhodné vyberanie určitého počtu jedincov populácie (typicky 2 až 8) s rovnakou pravdepodobnosťou a ich porovnanie na základe fitness. Jedinec s najlepším

ohodnotením fitness je víťazom turnaja a postupuje do ďalších krokov algoritmu. Proces náhodného výberu a porovnania prebehne toľkokrát, koľko jedincov má byť vybraných.

Poslednou uvedenou a zároveň najjednoduchšou selekčnou metódou je deterministická selekcia. U deterministickej selekcie sa pre reprodukciu deterministicky vyberie K jedincov s najvyššou hodnotou fitness. Obvykle sa vyberá 10 % až 50 % jedincov predošlej populácie.

Po selekcii nasleduje kríženie. Fáza kríženia využíva dvoch, prípadne viac rodičov naraz. S nemennou pravdepodobnosťou p_c, ktorá je parametrom genetického algoritmu (napríklad $p_c = 0,7$), náhodne rozhodne, či sa uplatní kríženie. Ak nie, vytvoria sa noví potomkovia, ktorí sú presnými kópiami rodičov. V prípade kríženia sú vytvorení ako kópie rodičov so vzájomne vymenenými časťami chromozómov, pričom pozícia kríženia je určená náhodne. Uvedený postup sa opakuje, pokým nie je vytvorený požadovaný počet jedincov.

Existuje niekoľko druhov krížení: jednobodové, viacbodové a uniformné. U jednobodového kríženia je náhodne vygenerovaný bod kríženia, v ktorom dôjde k prehodeniu génov nachádzajúcich sa za týmto bodom. Príklad jednobodového kríženia chromozómov X a Y, kde vzniknú nové dva chromozómy X' a Y' je nasledujúci:

$$X : 11|011010 \quad Y : 00|001111 \quad X' : 11001111 \quad Y' : 00011010$$

Viacbodové kríženie umožňuje definovať väčší počet bodov kríženia a tým pádom aj dôkladnejšie premiešanie genetického materiálu medzi rodičmi. Príklad viacbodového, konkrétne dvojbodového kríženia je:

$$X : 110|1101|0 \quad Y : 000|0111|1 \quad X' : 11001110 \quad Y' : 00011011$$

Uniformné kríženie pre každý gén zvlášť náhodne rozhodne, či dôjde k jeho výmene medzi rodičmi. Príkladom uniformného kríženia je nasledujúce kríženie:

$$X : 1\underline{1}0\underline{1}1\underline{0}10 \quad Y : 0\underline{0}0\underline{0}1\underline{1}11 \quad X' : 10001110 \quad Y' : 01011011$$

Pre gény s reálnymi hodnotami existujú aj heuristické a aritmetické kríženia.

Po krížení môže a nemusí nasledovať mutácia. Mutácia sa vykoná s malou a nemennou pravdepodobnosťou p_m, obvykle $p_m \doteq 0,001$. Počas mutácie sa náhodne vyberie malá podmnožina riešení a umelo sa zmenia, napríklad prevrátením hodnoty niektorého bitu. Ako výsledok mutácie obvykle vznikne jedinec, ktorý by nemohol byť vytvorený z rodičov obvyklou reprodukciou pomocou kríženia. Niekedy je v populácií väčšina jedincov podobných a tak sa budúce generácie jedincov nemôžu príliš zmeniť, hoci sú vzdialení optimálnemu riešeniu. Mutácia pomáha zmeniť populáciu príliš rovnakých a neoptimálnych riešení. Genetický algoritmus vďaka využívaniu kríženia i mutácie kombinuje náhodné prehľadávanie s horolezeckou tendenciou.

Mutácia je vykonávaná bit po bite. Z pravdepodobnosti mutácie p_m, ktorá je ďalším parametrom genetického algoritmu, sa dá jednoducho vypočítať očakávaný počet zmenených bitov ako $p_m \cdot m \cdot ps$. Každý bit chromozómu má rovnakú šancu byť zmutovaný. Fáza mutácie je vykonávaná nasledovne. Pre každý bit vo všetkých chromozómoch sa vygeneruje náhodná hodnota r v rozsahu $\langle 0,1 \rangle$. Ak $r < p_m$, je daný bit zmutovaný. Po fáze mutácie sú už vytvorení noví jedinci do svojej konečnej podoby, ostáva už len ich zaradenie do populácie.

Zaradenie nových potomkov do budúcej populácie (angl. *reinsertion*) môže byť uskutočnené niekoľkými spôsobmi [20]:

- počet potomkov je rovnaký ako počet rodičov, potomkovia plne nahradia rodičov (angl. *pure reinsertion*),

- počet potomkov je menší než počet rodičov, nahradenie je náhodné s rovnomerným rozložením (angl. *uniform reinsertion*),
- počet potomkov je menší než počet rodičov, sú nahradení najhoršie ohodnotení rodičia (angl. *elitist reinsertion*),
- počet potomkov je väčší než počet rodičov, do novej populácie sa zaradia len potomkovia s lepším hodnotením (angl. *fitness-based reinsertion*).

Existuje niekoľko vylepšení uvedeného základného genetického algoritmu, príkladom je ostrovný algoritmus [19]. Princíp fungovania ostrovného algoritmu je popísaný pseudokódom 4. Z pseudokódu vidno, že každý ostrov si spravuje svoju populáciu sám. Veľkosť populácií všetkých ostrovov je rovnaká. Spravovanie populácie spočíva vo výbere rodičov z vlastnej populácie, krížení, mutácií a občasnom procese nazývanom migrácia. Pri migrácií sú vymenení jedinci medzi ostrovmi. Ostrovný algoritmus prináša niekoľko výhod, okrem možnosti paralelného výpočtu vykazuje taktiež lepšiu prehľadávaciu schopnosť v zmysle nájdenia lepšieho riešenia ako aj menšieho počtu vyhodnocovania fitness funkcií. Jednou z príčin vylepšenia kvality prehľadávania je aj fakt, že jednak sú ostrovy nezávislé, teda sú prehľadané rôzne regióny a na druhej strane si ostrovy medzi sebou vymieňajú informácie pomocou migrácie.

Algoritmus 4: ostrovný algoritmus [9]

ostrovný_algoritmus(dĺžka_trvania_algoritmu, počet_ostrovov, počet_jedincov_na_ostrove)

```
nezávislá inicializácia všetkých subpopulácií
for i=0 to dĺžka_trvania_algoritmu do
    Nezávislé vykonanie niekoľkých krokov jednoduchého genetického algoritmu
    nad každou subpopuláciou
    Migrácia a asimilácia jedincov medzi všetkými susednými subpopuláciami
end
return(najlepší_jedinec_medzi_všetkými_populáciami);
```

Genetický algoritmus je v literatúre označovaný za riadenú náhodnú prehľadávaciu metódu alebo za stochastický hill-climbing prehľadávajúci veľký stavový priestor. Genetické algoritmy majú niekoľko výrazných výhod i nevýhod. K výhodám patria autonómnosť, schopnosť strojového učenia, robustnosť, flexibilita, jednoduchý a priamočiary výpočet, ľahká paralelizácia výpočtu. Nevýhodami sú premenlivý výsledok a dĺžka výpočtu, možnosť nájdenia len lokálneho extrému, prípadne nenájdenia existujúceho výsledku.

Ďalšou výraznou nevýhodou genetického algoritmu je, že napriek existencii veľa možných nastaviteľných parametrov, ako napríklad dĺžka behu, veľkosť populácie, pravdepodobnosť či použitý algoritmus mutácie a mnohé ďalšie, neexistuje teoretická metóda, ktorá by dokázala jednoznačne určiť najvýhodnejšie nastavenie týchto parametrov. Parametre genetického algoritmu sa určujú experimentálne.

V budúcnosti sa predpokladá preskúmanie stále nevyriešených problémov ohľadne genetických algoritmov ako napríklad definovanie problémov, pre ktoré sú genetické algoritmy efektívne a pre ktoré nie sú. Ďalšou nepreskúmanou oblasťou je porovnanie genetických algoritmov s inými metódami, ako napríklad neurónovými sieťami na konkrétne typy úloh. Užitočné by bolo taktiež teoreticky vedieť určiť najlepšie hodnoty parametrov. Ďalej zistiť, ako základné operácie napríklad mutácia či kríženie, ovplyvňujú celkový vývoj genetického algoritmu konkrétne napríklad konvergenciu riešení.

Kapitola 3

Zadanie a analýza úlohy

Pri vývoji dnešných komplexných hardvérových obvodov majú aj minimálne zmeny zdrojového kódu tendenciu zavádzať nové chyby. Častá a výpočtovo rýchla kontrola správnosti aktuálneho kódu môže výrazne skrátiť dobu hľadania novo-zavedenej chyby. Ako je však možné rýchlo overiť správnosť celého obvodu?

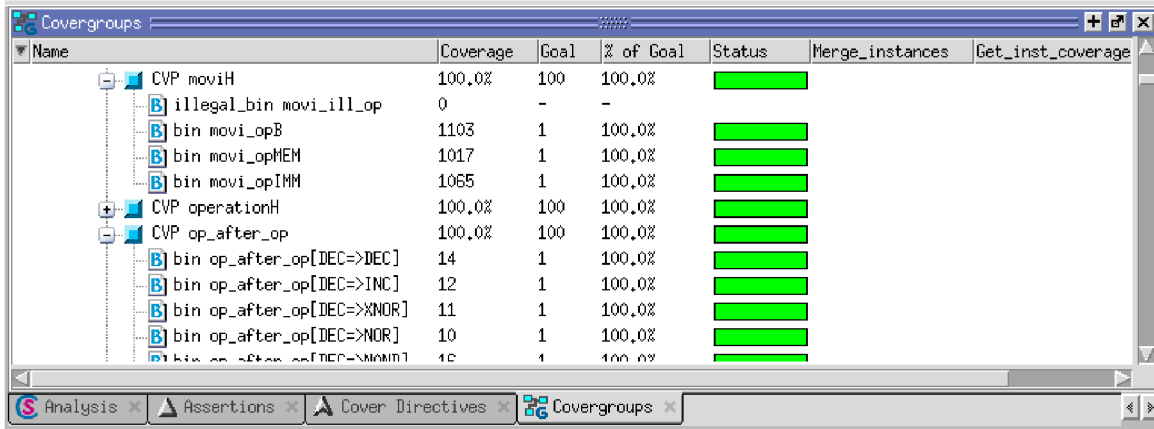
Na overenie správnosti celého obvodu sa ponúka funkčná verifikácia, ktorá dokáže overiť 100% **špecifikáciou zadaných** vlastností. Beh verifikácie však rozhodne nie je rýchly, môže trvať až niekoľko dní. Uvedené je jednak dôsledkom toho, že simulovanie inherentne paralelného hardvéru je veľmi pomalé, ale aj dôsledkom generovania pseudonáhodných vstupných transakcií pre verifikovaný obvod. To znamená, že transakcie overujúce nejakú vlastnosť sa podarí generátoru vygenerovať niekoľkokrát, zatiaľ čo transakciu overujúcu nejakú okrajovú podmienku sa vygenerovať nedarí. Dosiahnuté redundancie sú výrazné, ale pre funkčnú verifikáciu dokonca často žiadané, pretože sa overia i vlastnosti, ktoré v špecifikácií môžu chýbať. Pre opakované overovanie správnosti kódu pomocou regresných testov je však redundancia nežiadaným javom, dôležitá je rýchlosť behu testov a overenie konkrétnych kľúčových vlastností. Možným riešením tohto problému je odstránenie redundantných transakcií zo sady transakcií získaných v procese funkčnej verifikácie.

Cieľom tejto práce je zistiť, či sú transakcie získané v procese funkčnej verifikácie vhodným zdrojom pre vytvorenie sady regresných testov. Zároveň je cieľom zistiť, či sú evolučné algoritmy vhodným prostriedkom pre optimalizáciu sady uvedených vysoko redundantných regresných testov získaných z funkčnej verifikácie. K splneniu uvedeného cieľa vedie niekoľko postupných krokov. Prvým krokom je naštudovanie problematiky regresných testov, funkčnej verifikácie pomocou metodiky UVM, jazyka SystemVerilog a evolučných algoritmov. Ďalším krokom je navrhnutie a implementácia evolučného algoritmu umožňujúceho redukciu počtu testovacích vektorov regresných testov. Jeden testovací vektor regresných testov v tomto prípade odpovedá jednej vstupnej transakcii z funkčnej verifikácie. K práci bolo poskytnuté vopred pripravené UVM verifikačné prostredie. Verifikačné prostredie klasicky umožňuje vygenerovanie neoptimálnej sady transakcií s pokrytím vlastností 100%. Verifikačné prostredie ďalej taktiež umožňuje meranie pokrytia vlastností na sade transakcií. Verifikačné prostredie tak poskytuje zásadnú funkcionalitu pre evolučný algoritmus. Z toho dôvodu je potrebná ich vzájomná interakcia. Po implementácií optimalizačného prostredia je potrebné previesť sadu experimentov. Koniec práce má za úlohu porovnať novo vytvorenú zoptimalizovanú sadu regresných testov s pôvodnou sekvenciou transakcií získanou z procesu funkčnej verifikácie a diskutuje dosiahnuté výsledky.

3.1 Nameraná redundancia vektorov v konkrétnom obvode

Závažnosť problému redundancie vektorov je vhodné demonštrovať na konkrétnom príklade. Aritmeticko-logická jednotka (ALU) využívaná v tejto diplomovej práci má definovaných 404 vlastností. Aby sa dosiahlo maximálne pokrytie týchto vlastností, je potrebné overiť každú z nich aspoň raz. Zároveň, žiadnu vlastnosť nie je nutné overiť viac než jedenkrát. V experimente s pseudonáhodným generovaním postupnosti vstupných vektorov v procese funkčnej verifikácie, bolo na dosiahnutie 100 % pokrytia potrebné vytvoriť 4000 vstupných vektorov. Pri použití týchto vektorov bolo prevedených až 15818 preverení definovaných vlastností, teda $15818 - 404 = 15414$ overení bolo zbytočných. To poukazuje na približne 39-násobnú redundanciu a poskytuje tak veľa priestoru na optimalizácie.

Nameranie a zobrazenie pokrytia vlastností obvodu bolo realizované pomocou nástroja ModelSim. Obrázok 3.1 ukazuje vlastnosti (body pokrytia) tak, ako ich zobrazuje ModelSim. Z obrázku možno vyčítať, koľkokrát bol ktorá vlastnosť overená ako aj minimálny požadovaný počet overení danej vlastnosti, čo je v prípade použitej ALU vždy 1.



Obrázek 3.1: Nameranie a zobrazenie pokrytia vlastností obvodu pomocou nástroja ModelSim.

K uvedeným hodnotám je potrebné zmieniť ešte niekoľko faktov. Počet vygenerovaných vstupných vektorov, ako aj samotné vektory sa môžu výrazne líšiť v závislosti od nastavených parametrov generátoru pseudonáhodných čísel (napr. *seed*). Ďalším dôležitým faktom je výsledok veľkosti redundancie, ktorý je čisto teoretický a preto odpovedajúca redukcia počtu vstupných vektorov nemusí byť reálne dosiahnuteľná.

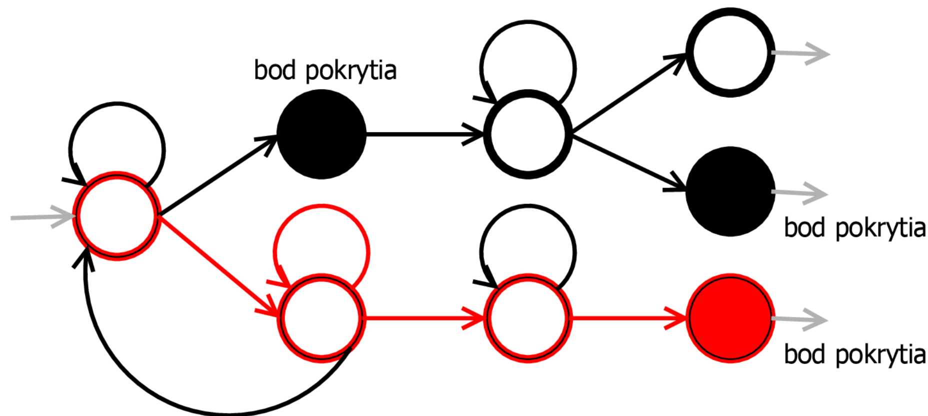
3.2 Evolučný algoritmus pre optimalizáciu regresných testov

Táto podkapitola sa venuje analýze a zdôvodneniu, prečo majú evolučné algoritmy potenciál byť vhodné pre optimalizáciu sady regresných testov získaných z procesu funkčnej verifikácie.

Problém optimalizácie sady regresných testov získaných z procesu funkčnej verifikácie je NP-úplný problém [3], ktorého riešeniu sa venuje už niekoľko publikácií. V niektorých z týchto publikácií, napríklad [3], sa uvedený problém označuje ako problém množiny pokrytia (angl. *Set-Covering Problem*). V literatúre je taktiež možné nájsť niekoľko metód navrhnutých pre riešenie tohto problému. Medzi navrhované riešenia patria napríklad: metódy

greedy, LP relaxácie (angl. *LP relaxations*), spätná eliminácia (angl. *Backward elimination*) [8] a ďalšie. Jednotlivé metódy majú rôzne vlastnosti a teda rôzne výhody, ale i nevýhody, ktoré sú detailnejšie popísané v nasledujúcich odsekoch. Použitie evolučných algoritmov pre zadaný problém optimalizácie som v literatúre nenašla, no vďaka svojím vlastnostiam má šancu sa metódam z publikácií vyrovnať, či dokonca ich v niektorých prípadoch poraziť.

Metódy greedy [8] navrhujú vytváranie sady optimalizovaných regresných testov priamo počas behu funkčnej verifikácie generujúcej neoptimálnu sadu transakcií a to nasledujúcim spôsobom. Vždy, keď použitie vygenerovanej transakcie povedie k zvýšeniu pokrytia zverifikovaných vlastností, je táto transakcia pridaná do vytváranej sady optimálnych regresných testov. Tento spôsob sa však javí ako nedostatočný. Problém s metódou greedy je ten, že v súčasných verifikačných prostrediach je možné a zároveň využívané nadefinovanie aj komplexných bodov pokrytia ako napríklad sekvencie niekoľkých transakcií. Beh verifikácie a postupná aplikácia transakcií za účelom zvyšovania pokrytia sa dá potom predstaviť ako stavový automat. Nech je malá časť takéhoto automatu znázornená na obrázku 3.2. Potom prechody odpovedajú použitým transakciám. Stavy graficky reprezentované plným krúžkom vyjadrujú dosiahnutie bodu pokrytia, tj. pokrytie nejakej našpecifikovanej vlastnosti. Červenou farbou je znázornená možná cesta automatom. Vidno, že automat obsahuje mnoho hrán, ktoré nemenia stav automatu. To znamená, že transakcie spôsobujúce prechod takýmito hranami sú redundantné. Celý automat je veľmi komplexný a funkčná verifikácia ho nepozná. Preto funkčná verifikácia neposkytuje spôsob, ako je možné pri bodoch pokrytia pozostávajúcich s viac než jednej transakcie, určiť presnú postupnosť takých transakcií, ktoré viedli k pokrytiu nejakej vlastnosti. Získavanie sady optimálnych regresných testov na základe popísaného prístupu v [8] sa preto nejaví ako dostačujúce.



Obrázek 3.2: Pokrývanie bodov pokrytia vektormi z funkčnej verifikácie sa dá predstaviť ako prechod konečným automatom.

Metóda spätnej eliminácie [8] sa taktiež pre súčasné komplexné verifikované vlastnosti nejaví ako vyhovujúca. Pri tejto metóde v základnej verzii sa v každej iterácií ohodnotia vzniknuté sady regresných testov po odstránení každého z obsiahnutých vektorov v aktuálnej sade. Pre sadu regresných testov obsahujúcu n vektorov sa tak urobí až n ohodnotení v jedinej iterácií, čo je veľa. Následne je odstránený ten vektor, ktorého odstránenie prinesie najlepšie zlepšenie. Prehľadávaný priestor možných riešení je obrovský, pretože obsahuje všetky možné kombinácie odstránených a neodstránených, prípadne inak upravených vektorov počiatočnej neoptimálnej sady.

Veľkosť prehľadávaného priestoru je pre riešenú úlohu optimalizácie sady regresných testov získaných z funkčnej verifikácie problémom u všetkých algoritmov so **systematickým prehľadávaním** priestoru. Uvedený fakt vylučuje radu algoritmov. Ako jedna z potenciálne vhodných metód, ktoré zvládnu veľkosť prehľadávaného priestoru, sa ponúka evolučný algoritmus. Evolučný algoritmus navyše disponuje ďalšími výhodnými vlastnosťami pre zadaný problém:

- Jednou z popredných výhod evolučného algoritmu je jeho univerzálnosť. Princíp, akým evolučné algoritmy fungujú, nevyžaduje žiadnu znalosť o špecifikovaných a verifikovaných vlastnostiach obvodu. Navyše tieto vlastnosti môžu byť aj komplexné a skladať sa tak napríklad z ľubovoľne dlhých sekvencií transakcií.
- Výhodou evolučného algoritmu je aj súčasné prehľadávanie niekoľkých vzdialených bodov prehľadávaného priestoru možných riešení.
- Ďalšou z výhod evolučného algoritmu je možnosť variabilných krokov medzi iteráciami, čím sa v tomto prípade rozumie vymazanie a zmena niekoľkých transakcií naraz. Možné sú skoky naprieč veľkou časťou prehľadávaného priestoru riešení.

Evolučný algoritmus má tak potenciál nájsť dostatočne optimálne riešenie rýchlo a to za cenu, že toto nájdené riešenie nebude globálnym extrémom v rámci celého prehľadávaného priestoru. To však nevadí, je potrebné nájsť vhodný kompromis medzi optimálnosťou nájdeného riešenia a časovou náročnosťou jeho hľadania.

Analyzované výhody evolučného algoritmu pre optimalizáciu sady regresných testov získaných z procesu funkčnej verifikácie sú odvodené na základe teoretických poznatkov v kapitole 2.3. Zvyšok práce sa venuje návrhu, implementácií a následnému overeniu vhodnosti evolučných algoritmov pre riešený problém.

Kapitola 4

Základný návrh riešenia

Navrhované optimalizačné prostredie sa skladá z nasledujúcich troch častí:

1. evolučný algoritmus,
2. automatizácia testov,
3. verifikačné prostredie podľa UVM.

Základný princíp procesu optimalizácie prebiehajúceho v navrhnutom optimalizačnom prostredí je nasledovný. Z verifikačného prostredia sa získa zoznam vstupných transakcií, ktorý spĺňa požadované pokrytie špecifikovaných vlastností hardvérového obvodu. Získaný zoznam transakcií je však vysoko redundantný. Cieľom implementovaných evolučných algoritmov je skrátiť získaný zoznam vstupných transakcií tak, aby bol čo najkratší a pokrytie vlastností sa nezmenšilo. Meranie zachovania či zmenšenia pokrytia zoznamu transakcií po jeho redukcii evolučným algoritmom je realizované pomocou existujúceho verifikačného prostredia. Funkcionalita merania pokrytia vlastností je vo verifikačnom prostredí už zahrnutá a to primárne pre inú úlohu, ktorou je meranie pokrytia transakcií generovaných verifikačným prostredím v aktuálnom behu verifikácie. Je však potenciálne možné túto funkcionalitu merania pokrytia využiť aj pre ďalšie účely ako je napríklad požadované zmeranie pokrytia zoznamu transakcií vygenerovaných externým programom akým je aj evolučný algoritmus. Zvyšok kapitoly podrobnejšie popisuje všetky tri časti optimalizačného prostredia, každému venuje jednu podkapitolu.

4.1 Evolučný algoritmus

Prvou časťou optimalizačného prostredia je **evolučný algoritmus**. Otázkou, ktorú si treba položiť ako prvú pri návrhu evolučného algoritmu je, aký typ evolučného algoritmu je vhodné použiť. V podkapitole teoretického úvodu týkajúcej sa evolučných algoritmov 2.3, boli popísané rôzne typy:

- genetické algoritmy,
- genetické programovanie,
- evolučné stratégie,
- evolučné programovanie.

Uvedené druhy evolučných algoritmov sa od seba vzájomne líšia napríklad:

- veľkosťou populácie,
- využívaním genetickej operácie kríženia,
- využívaním ďalších špeciálnych genetických operácií,
- samo-adaptáciou, inými slovami premenlivými pravdepodobnosťami genetických operácií,
- reprezentáciou jedincov, napr.: spustiteľné stromové štruktúry, vektor reálnych parametrov,
- využívanie Gaussovho rozloženia pre genetické operácie.

Pre účely tejto diplomovej práce boli z uvedených typov evolučných algoritmov zvolené genetické algoritmy a to na základe analyzovania a porovnania vzájomných odlišností jednotlivých druhov evolučných algoritmov. Porovnaním odlišností môžeme napríklad vylúčiť genetické programovanie pretože reprezentuje jedincov ako spustiteľné stromové štruktúry a evolučné stratégie reprezentujúce jedincov ako vektor reálnych čísel v kombinácií s využívaním Gaussovho rozloženia pre genetické operácie.

Ďalšie odlišnosti, ktoré treba zvážiť sú veľkosť populácie, využívanie operácie kríženia či používanie samo-adaptácie. Genetické algoritmy v svojej základnej verzii využívajú kríženie a mutáciu, pracujú s ľubovoľnou ale konštantnou veľkosťou populácie, ale nepodporujú samo-adaptáciu. Pomocou nastaviteľných parametrov a prípadného dodatočného rozšírenia je však do určitej miery možné genetickým algoritmom simulovať aj niektoré koncepty evolučného programovania a evolučných stratégií, ktoré sa viac opierajú o mutáciu. Preto sa genetické algoritmy pre riešený problém optimalizácie sady regresných testov javia byť najviac vhodné, aj keď v základnom koncepte nie dokonalé.

Vybraný druh evolučných algoritmov - genetické algoritmy existujú v niekoľkých vylepšeniach, pre prvotné riešenie úlohy redukcie počtu testovacích vektorov regresných testov sa však použije najzákladnejší genetický algoritmus, ktorého princíp bol popísaný pomocou diagramu 2.7 v podkapitole 2.3.1. Pri návrhu genetického algoritmu pre účely optimalizácie sady regresných testov však treba čeliť zásadnému problému. Ten spočíva v rôznej dĺžke chromozómov vyplývajúcej zo snahy o ich skracovanie, čo pre genetické algoritmy nie je typické. Na skracovanie dĺžky chromozómov je potrebné prihliadať pri navrhovaní procesu mutácie a kríženia, kde je nutné existujúce prístupy prispôsobiť. Navrhovaný GA potom vyzerá nasledovne:

Kandidátne riešenie, jedinec: skladá sa zo sady vstupných transakcií pre DUT.

Chromozóm: sekvencia transakcií.

Počiatočné kandidátne riešenie: nie je vygenerované náhodne, ale je získavané z funkčnej verifikácie ako sekvencia vstupných transakcií pre DUT dosahujúca pokrytie vybraných vlastností 100 %.

Mutácia: je navrhovaná dvomi spôsobmi:

1. vypustenie niektorej z transakcií jedinca,
2. vzájomná výmena poradia dvoch transakcií v rámci chromozómu jedinca.

Kríženie: je možné aplikovať známe prístupy ako napríklad jednobodové či uniformné kríženie. Avšak možnosť zmeny dĺžky chromozómov umožňuje realizovať aj všeobecnejší prístup a to taký, pri ktorom sú medzi dvomi jedincami zamenené bloky transakcií rôznej dĺžky a rôzneho umiestnenia v rámci chromozómu. Príklad uvedeného všeobecného kríženia chromozómov X a Y, kde vzniknú nové dva chromozómy X' a Y' je nasledujúci:

$$X : v_1^1 v_1^2 | v_1^3 v_1^4 v_1^5 | v_1^6 v_1^7 \quad Y : v_2^1 | v_2^2 | v_2^3 \quad X' : v_1^1 v_1^2 v_2^2 v_1^6 v_1^7 \quad Y' : v_2^1 v_1^3 v_1^4 v_1^5 v_2^3$$

Pričom v_i^j označuje j-tu transakciu i-teho chromozómu.

Fitness funkcia: vhodne kombinuje pokrytie a počet transakcií obsiahnutých v jedincovi. Navrhovaná fitness funkcia vráti počet obsiahnutých transakcií za predpokladu, že jedinec splňuje pokrytie 100 % špecifikovaných vlastností. Čo implikuje, že hodnota fitness je nepriamoúmerná s optimálnosťou jedinca. Pri nesplnení pokrytia vráti fitness funkcia najhoršiu možnú hodnotu.

Podmienka ukončenia výpočtu: dlhodobá stagnácia ohodnotenia jedincov, pretože nie je možné inak určiť mieru optimálnosti výsledného riešenia. Pre možnosti experimentovania je taktiež navrhovaná možnosť ukončenia výpočtu po prevedení nastaviteľného počtu iterácií genetického algoritmu.

Vymenované špecifiká navrhovaného algoritmu sú spätne premietnuté do základného genetického algoritmu. Vzniknutý algoritmus je pre prehľadnosť znázornený graficky a to na obrázku 4.1. Vývojový diagram popisujúci princíp základného genetického algoritmu, ktorý už bolo možné vidieť kapitole venujúcej sa teórií ohľadne genetických algoritmov, je obohatený stručnými poznámkami priraďujúcimi navrhované špecifiká.

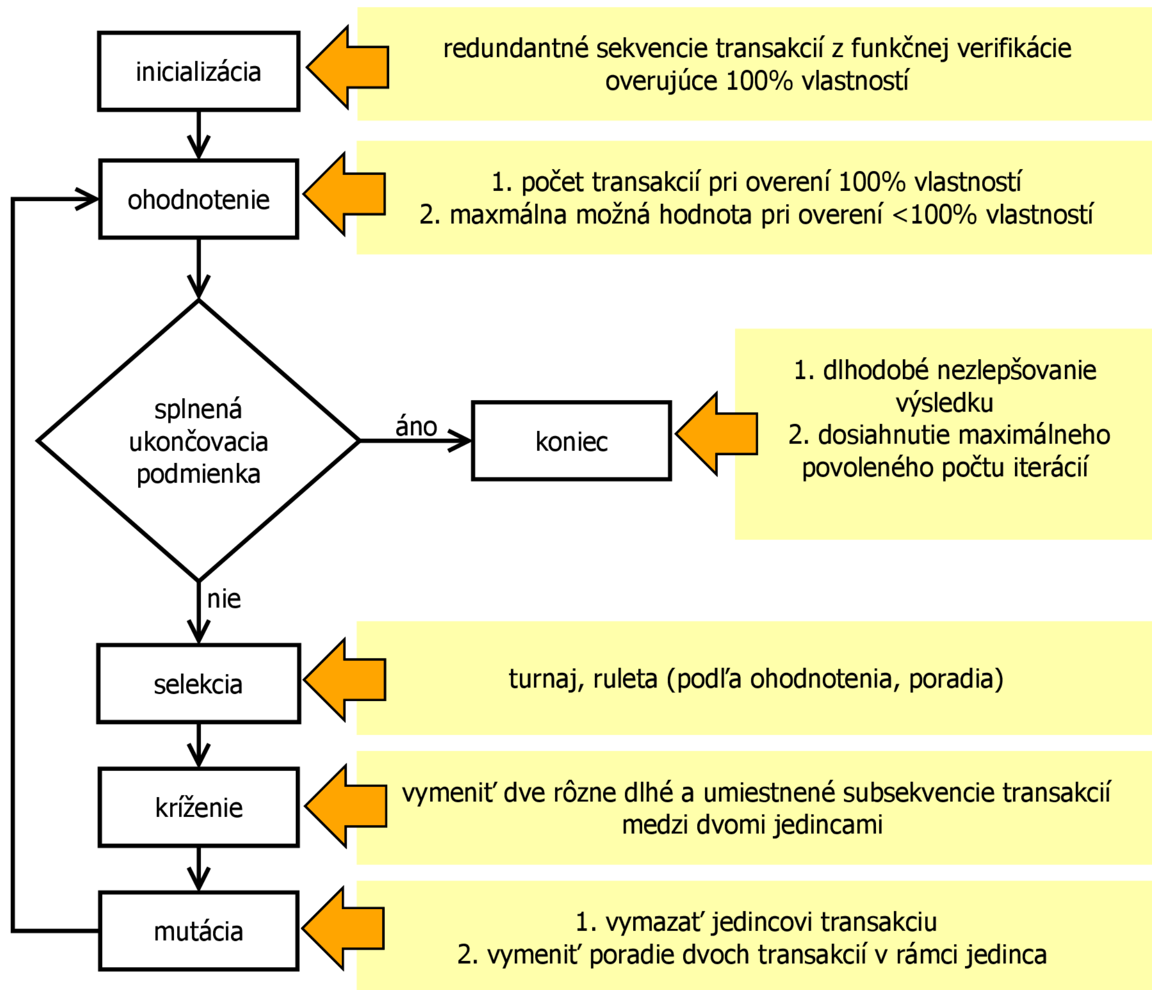
Po nameraní úspešnosti navrhovanej základnej verzie genetického algoritmu je podľa dosiahnutých výsledkov možné pokračovať nejakým vylepšením tohto algoritmu, napríklad úpravou na ostrovný algoritmus. Aktuálne však nie sú známe výhody a nedostatky navrhovaného riešenia a tak je ťažké určiť, kam by mali smerovať ďalšie vylepšenia.

4.2 Automatizácia testov

Druhou súčasťou optimalizačného prostredia je automatizácia testov. Tá je potrebná z nasledujúceho dôvodu. Ako už bolo naznačené, pri genetických algoritmoch je nutné čeliť otázke správneho nastavenia parametrov algoritmu. Obecne majú genetické algoritmy veľký počet nastaviteľných parametrov. Medzi nastaviteľné parametre patria napríklad pravdepodobnosť mutácie a kríženia, ako aj ktorý algoritmus selekcie, kríženia a mutácie sa má použiť. Keďže však neexistuje spôsob, ako je možné jednoznačne určiť najlepšie nastavenie týchto parametrov, je správne nastavenie zisťované empriricky, čo obvykle znamená veľký počet testov. Testovanie môže byť zdĺhavé a pracné, preto je potreba ho čo najviac zjednodušiť.

4.3 Verifikačné prostredie UVM

Poslednou súčasťou optimalizačného prostredia je verifikačné prostredie. Verifikačné prostredie UVM je pre túto diplomovú prácu už pripravené dopredu tak, že poskytuje bežnú funkcionalitu. K poskytovanej funkcionalite patrí okrem iného aj:



Obrázek 4.1: Navrhovaný genetický algoritmus.

- generovanie pseudonáhodných transakcií,
- meranie pokrytia.

Aby bolo možné použiť verifikačné prostredie v optimalizačnom prostredí, treba k nemu navyše doimplementovať dve pridané funkcie:

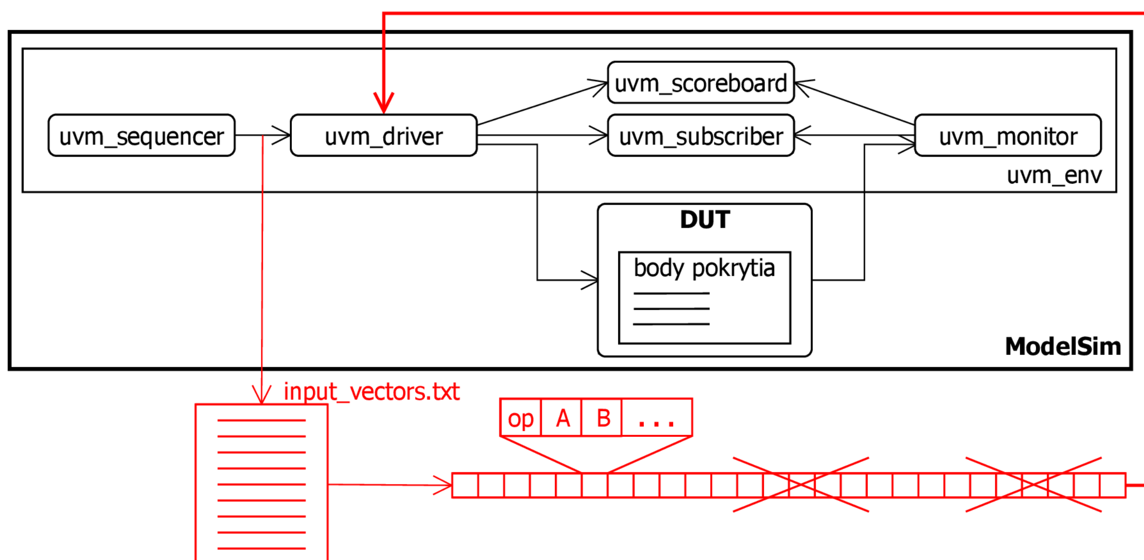
- Pre účely **generovania počiatočnej populácie** je potrebné doprogramovať vygenerovanie sekvencie transakcií pokrývajúcej presne 100 % vlastností a poskytnutie tejto sekvencie transakcií externému programu, konkrétne genetickému algoritmu.
- Ďalej, ako už bolo uvedené v predošlom texte, sa funkčná verifikácia použije pre účely **merania fitness**. Preto je potrebné, aby verifikačné prostredie taktiež dokázalo načítať dopredu pripravenú sekvenciu transakcií, zmeralo pokrytie tejto sekvencie a následne poskytlo výsledok genetickému algoritmu.

Pri popisovaní verifikačného prostredia je ešte dôležité zmieniť, že za hardvérový obvod, na ktorom má prebiehať verifikácia a vývoj aplikácie evolučných algoritmov bola zvolená

ALU. Výber konkrétneho obvodu neovplyvní žiadnu inú časť optimalizačného prostredia, tj. ani genetický algoritmus, ani automatizačný test. Ostatné časti optimalizačného prostredia sú teda plne nezávislé na použitom hardvérovom obvode – a teda univerzálne. Použitie konkrétneho obvodu poslúži k otestovaniu správnosti implementácie a zmeraniu úspešnosti optimalizačného procesu.

4.4 Súhrn

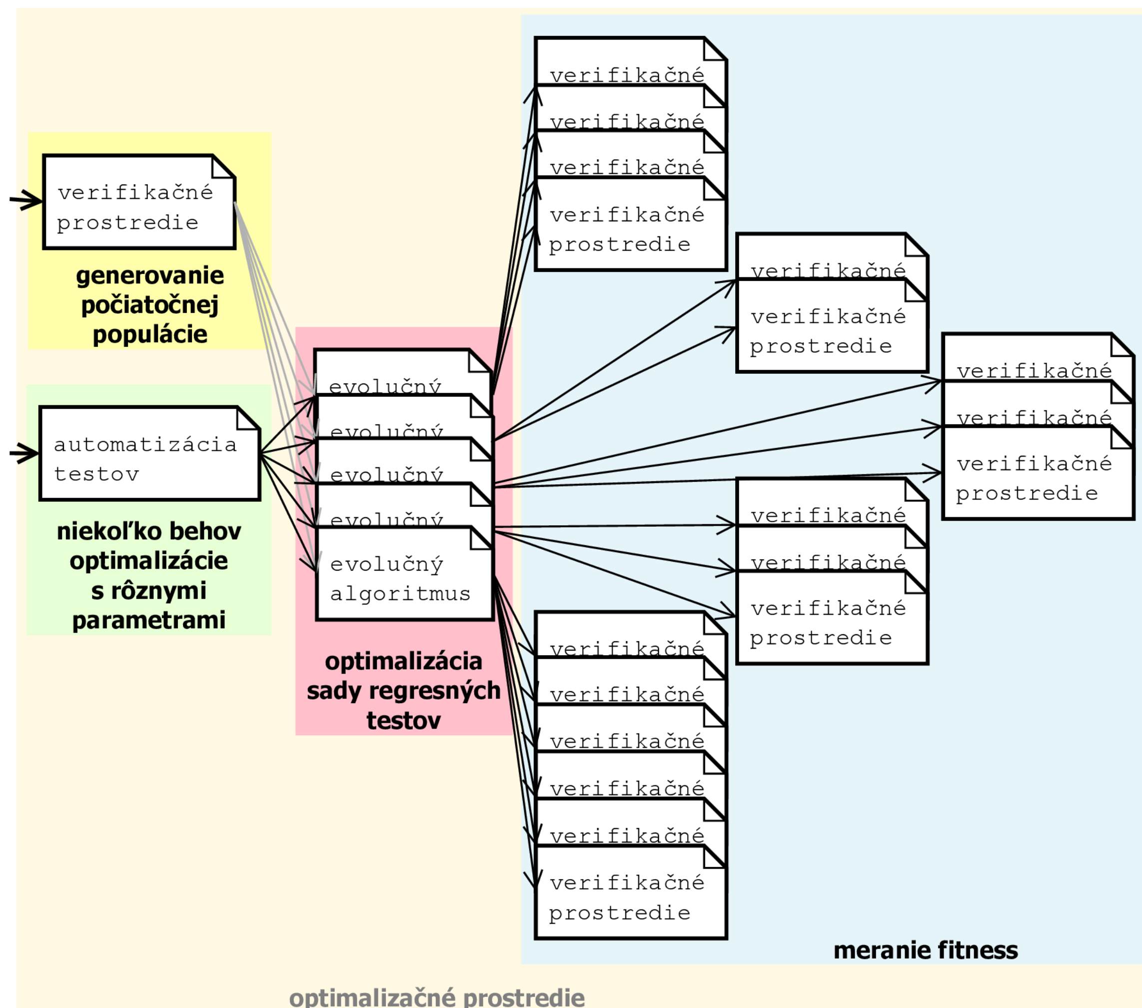
Táto podkapitola popisuje interakciu jednotlivých častí optimalizačného prostredia medzi sebou. Ako už bolo uvedené, optimalizačné prostredie sa skladá z troch častí a to verifikačného prostredia, genetického algoritmu a automatizácie testov. Nasledujúci text tejto podkapity sa najskôr zameriava na spoluprácu verifikačného prostredia a genetického algoritmu. Následne je popísaný vzťah automatizácie testov k ostatným častiam optimalizačného prostredia a celkový pohľad na všetky komunikujúce časti optimalizačného prostredia.



Obrázek 4.2: Interakcia verifikačného prostredia a genetického algoritmu.

Z textu v predošlých podkapitolách vyplýva, že okamžitá interakcia verifikačného prostredia a gentického algoritmu je nevyhnutná. A to z dôvodu častého merania fitness, kde je genetický algoritmus nútený čakať na výsledok od funkčnej verifikácie. Na obrázku 4.2 je zobrazený popisovaný princíp interakcie verifikačného prostredia znázorneného čiernou farbou a genetického algoritmu znázorneného červenou. Obrázok ukazuje počiatok behu optimalizačného procesu, kde dochádza ku generovaniu sekvencie transakcií verifikačným prostredím. Generovaná sekvencia transakcií vždy spĺňa 100 % pokrytie, ale je vysoko redundantná. Na obrázku je táto sekvencia zapísaná verifikačným prostredím do súboru, odkiaľ si ju môže genetický algoritmus prečítať. Uvedená sekvencia transakcií je ďalej použitá genetickým algoritmom ako jeden jedinec počiatočnej populácie. Na jedincov sú následne aplikované genetické operácie kríženia a mutácie, čo ma za následok výmenu alebo vymazanie niektorých transakcií. Vzniká tak nový jedinec, ktorého fitness treba zistiť. Pre tento účel je jedinec predaný verifikačnému prostrediu, ktoré ho chápe ako novú sekvenciu transakcií. Genetický algoritmus postupne iteruje a nachádza tak stále lepších jedincov (riešenia

problému), čo znamená kratšie a teda menej redundantné sekvencie transakcií splňujúce pokrytie. Uvedený princíp a implementačné detaily sú podrobnejšie rozpísané v kapitole 5.



Obrázek 4.3: Ucelený pohľad na fungovanie optimalizačného prostredia.

Ku popisovanému genetickému algoritmu s verifikačným prostredím treba pre vytvorenie uceleného optimalizačného prostredia pridať ešte poslednú časť - automatizáciu testov. V optimalizačnom prostredí je pomocou automatizácie testov jednoducho spustených niekoľko behov genetického algoritmu sekvenčne a to s odlišným nastavením parametrov. Vyskúšanie viacerých nastavení parametrov je nutné z dôvodu empirického zisťovania ich najlepšieho nastavenia. Výsledkom každého genetického algoritmu je do určitej miery optimalizovaná sekvencia transakcií a teda do určitej miery optimalizované regresné testy. Parametre genetického algoritmu, ktorý vytvoril najviac optimálne riešenie, sú považované za najlepšie. Celkový pohľad na interakciu všetkých častí optimalizačného prostredia je znázornený na obrázku 4.3. Celému behu optimalizácie predchádza vygenerovanie počiatočnej populácie verifikačným prostredím. Následná optimalizácia pozostáva z behu automatizovaných testov. Spúšťa sa tak niekoľko genetických algoritmov s odlišným nastavením parametrov. Každý genetický algoritmus počas svojho behu veľakrát spustí celú verifikáciu a čaká na jej ukončenie z dôvodu vyhodnocovania fitness niektorého jedinca.

Kapitola 5

Návrh a implementácia

Prvou úlohou implementácie, ktorú je potrebné vyriešiť pred samotnou implementáciou jednotlivých troch častí verifikačného prostredia (genetický algoritmus, verifikačné prostredie a automatizácia testov) je rozhodnúť, akým spôsobom bude realizovaná interakcia medzi týmito tromi časťami. Pričom treba zvážiť, že verifikačné prostredie a genetický algoritmus sú podľa zadania programy v jazyku SystemVerilog a automatizácia testov je *bash* skript.

Volanie genetického algoritmu automatizačným prostredím, ako aj volanie verifikačného prostredia genetickým algoritmom je realizované systémovým volaním, pričom v jazyku SystemVerilog k tomuto účelu slúži príkaz `$system()`.

Predávanie parametrov a výsledkov je implementované pomocou zápisu a čítania zo súboru. Tento spôsob je síce pomalší, má však aj svoje výhody. Výhodami sú, že je tak umožnené predávať veľa dát, čo je napríklad pre predávanie jedinca s niekoľko tisícom transakcií potrebné. Druhou výhodou je univerzálnosť pre niekoľko operačných systémov.

Realizácia, ktorá môže čitateľa napadnúť a ktorá by priniesla ako zjednodušenie, tak zrýchlenie interakcie je zlúčenie evolučného algoritmu a verifikačného prostredia do jedného programu. Takáto alternatíva však nie je realizovateľná, pretože beh verifikácie v ModelSime neumožňuje nulovanie pokrytia a opätovné vyhodnocovanie pokrytia počas svojho behu. To je však potrebné pre opakované vyhodnocovanie fitness jedincov. Každé počítanie fitness tak vyžaduje vlastný beh.

Po definovaní rozhrania pre komunikáciu s ostatnými časťami optimalizačného prostredia implementácia pokračuje programovaním jednotlivých troch častí tohto prostredia: verifikačného prostredia, automatizácie testov a genetického algoritmu.

5.1 Verifikačné prostredie

Pre účely tejto diplomovej práce je poskytnuté verifikačné prostredie ALU v UVM s bežnou funkcionalitou poskytujúcou generovanie pseudonáhodných transakcií a meranie pokrytia vygenerovaných transakcií. Aby mohlo byť verifikačné prostredie navyše využívané ako generátor počiatočnej populácie, alebo ako prostriedok pre meranie fitness, treba bežnú funkcionalitu rozšíriť. Je vhodné, aby zásahy do verifikačného prostredia boli čo najmenšie za účelom jednoduchosti prípadného budúceho využívania v praxi.

Implementovanou novinkou, ktorá sa u verifikačných prostredí nepoužíva je nadefinovanie niekoľkých módov behu prostredia. Pričom každý mód odpovedá určitému správaniu a úlohe, ktorú má aktuálne prostredie vyriešiť. Bežne verifikačné prostredie plní len jednu úlohu, ktorou je zverifikovanie zadaného hardvérového obvodu. Verifikačné prostredie po

implementovanej úprave poskytuje na výber aspoň dva módy behu verifikácie:

1. `CHROMOSOME_GENERATON` pre vygenerovanie jedného jedinca počiatočnej populácie,
2. `COVERAGE_MEASUREMENT` pre meranie a poskytnutie fitness jedinca načítaného zo súboru.

Pričom prepnutie potrebného módu urobí užívateľ jednoducho a to nastavením parametru `MODE` v konfiguračnom súbore `test_parameters.sv`. Ďalšie odseky sa podrobnejšie venujú implementácií oboch módov.

Hlavná úloha módu `CHROMOSOME_GENERATON` je vygenerovať do súboru sekvenciu transakcií s pokrytím 100 %. Verifikačné prostredie tak treba rozšíriť o:

- zápis každej vygenerovanej transakcie do súboru,
- zisťovanie pokrytia doposiaľ vygenerovaných transakcií aby generovaní jedinci spĺňali pokrytie vlastností 100 %,
- manipuláciu so súbormi.

Navyše je vhodné doimplementovať túto funkcionalitu pri minimálnych zásahoch do pôvodného kódu. Pre implementáciu všetkých rozšírená stačí úprava jedinej triedy a to triedy definujúcej vstupnú sekvenciu transakcií (rozširuje `uvm_sequence`). V prípade poskytnutého verifikačného prostredia sa jedná o súbor `/transaction_sequence.svh`. Ďalej je potrebné poznať fázy UVM a teda poradie vykonávania metód a úloh vo verifikačnom prostredí UVM. Čo sa týka konkrétnych zmien, sú následovne. V `task body`, čo je úloha ktorá sa stará o generovanie transakcií, treba rozšíriť ukončovaciu podmienku generovania transakcií o ukončenie generovania v prípade dosiahnutia pokrytia vlastností 100 %. Zistenie pokrytia sa realizuje jednoducho a to volaním vstavanej funkcie `$get_coverage()`. V pôvodnej verzii bolo určenie počtu transakcií, ktoré splnia pokrytie ponechané na odhad užívateľa, čo je nepresné a nepohodlné. Navyše ukončenie sekvencie transakcií hneď po nadobudnutí požadovaného pokrytia zaručí najminimálnejšiu sekvenciu transakcií, ktorú je verifikačné prostredie schopné pri určitom nastavení generátorov náhodných čísel vygenerovať. To sa hodí pre presnosť neskoršieho výpočtu úspešnosti genetického algoritmu. Ďalej, zápis novo vygenerovanej transakcie do súboru, sa rieši taktiež v `task body`. Kód zápisu do súboru sa umiestni do cyklu generovania transakcií za vygenerovanie transakcie. Otvorenie a zatvorenie súboru pre zápis generovaných transakcií sa vykonajú v úlohach `task pre_body`, resp. `task post_body`, ktoré predchádzajú, resp. následujú po procese samotnej generácie transakcií a verifikácií obvodu. Možností, ako sa dá implementovať mód `CHROMOSOME_GENERATON` je niekoľko, implementované riešenie má však oproti ostatným takú prednosť, že mu stačí editovanie jediného súboru, čo prináša už spomínanú výhodu jednoduchosti používania konceptu. Pre pohodlnejšie generovanie jedincov úvodnej populácie je vytvorený *bash* skript `generate_candidate.sh`

Implementácia módu `COVERAGE_MEASUREMENT` prináša editáciu rovnakého súboru ako mód `CHROMOSOME_GENERATON`. Myšlienka je tá, že sa v hlavnom generačnom cykle transakcií v `task body` na vytvorenie každej transakcie nezavolá generátor pseudonáhodných čísel, ale jedno čítanie zo súboru, kde sú uložené všetky transakcie. Jedna iterácia tak načíta jednu transakciu namiesto vygenerovania jednej transakcie. Ukončovacia podmienka generačného cyklu je pozmenená z vyhovujúceho pokrytia, na prečítanie celého súboru. Ďalej `task pre_body` a `task post_body` sú opäť rozšírené o otváranie a zatváranie potrebných súborov, ktoré sú v tomto prípade súbor s uloženou sekvenciou transakcií, ktorej pokrytie

treba zmerať a súbor pre zapísanie výsledku, ktorým je pokrytie sekvencie transakcií z prvého súboru. Okrem toho sa v `task post_body` zistí a zapíše dosiahnuté pokrytie spracovávanej sekvencie transakcií pomocou `$get_coverage()` do zatváraného súboru. Dôležitým prídavkom je volanie príkazu `$finish()` na konci verifikácie. Tento príkaz spôsobí vypnutie ModelSimu, čo je pre účely veľakrát opakovaného výpočtu fitness v rámci jedného genetického algoritmu dôležité, aby neostalo zbytočne bežať niekoľko po výpočte pokrytia už nepotrebných verifikačných prostredí. Jedno z možných vhodných umiestnení volania tohto príkazu je do `report_phase` vo verifikačnej komponente triedy `uvm_scoreboard` za výpis štatistík verifikácie alebo aj do `task post_body` v `uvm_sequence`.

Celé verifikačné prostredie okrem komponenty so vstupnou sekvenciou transakcií, tj. komponenty typu `uvm_sequence` je odtienené od znalosti módov verifikačného prostredia. To prináša jednoduchosť rozšírenia bežného UVM verifikačného prostredia na prostredie potrebné pre genetický algoritmus.

5.2 Automatizácia testov

Myšlienka automatizácie testov je tá, že beh genetického algoritmu obvykle trvá dlho a doba jeho behu sa dopredu ťažko odhaduje. Zároveň o genetických algoritmoch obecne platí, že ich treba nechať prebehnúť niekoľkokrát a s rôznymi nastaveniami parametrov pre zistenie optimálneho nastavenia týchto parametrov. Užívateľovi by sa tak oplatilo pustiť niekoľko testov naraz a až pokým všetky nedobehnú sa o ne nestarať. Automatizácia testov je *bash* skript `tests.sh`, ktorý postupne sekvenčne spustí niekoľko genetických algoritmov, každý s osobitnými parametrami nadefinovanými užívateľom dopredu.

Realizácia a používanie automatizácie testov je také, že užívateľ si do nastaveného adresára nahrá patričný počet ľubovoľne pomenovaných konfiguračných súborov pre genetický algoritmus. Konfiguračný súbor obsahuje všetky nastaviteľné parametre genetického algoritmu ako napríklad veľkosť populácie, pravdepodobnosť mutácie, cesta k adresáru s počiatočnými jedincami, atd. Presnejšie je tento súbor popísaný v prílohe A. Automatizačný skript potom sekvenčne pre každý súbor v danom adresári spustí genetický algoritmus a uloží vhodne pomenované a prehľadne zapísané výsledky na špecifikované miesto.

5.3 Základný genetický algoritmus

Táto podkapitola sa venuje popisu implementácie základného genetického algoritmu. Popis začína opisom jedinca, ktorý reprezentuje možné riešenia problému. Od popisu jedinca sa v tejto podkapitole následne odrážajú opisy ostatných súvisiacich pojmov a to postupne populácia ako skupina jedincov, genetické operácie manipulujúce s jedincom za účelom vytvorenia nového lepšieho jedinca a tak isto metódy selekcie uprednostňujúce kvalitnejších jedincov.

Jedinec genetického algoritmu, taktiež označovaný ako kandidátne riešenie, ako už názov napovedá, reprezentuje nejaké jedno možné riešenie problému. V genetickom algoritme vytváranom v rámci tejto diplomovej práce je jedinec reprezentovaný ako sekvencia vstupných transakcií pre funkčné verifikovanie hardvérového obvodu ALU. Grafické znázornenie štruktúry jedinca ako sekvencie transakcií je zobrazené na obrázku 5.1. Transakcie majú presne daný heterogénny formát. V prípade použitej ALU sú položky vstupnej transakcie sú nasledovné:

- výber operácie,

- operand A,
- selekčný signál operandu B rozhodujúci medzi použitím hodnoty z registru, z pamäti alebo bezprostrednej,
- operand B v registri,
- operand B v pamäti,
- bezprostredne adresovaný operand B,
- aktivačný signál,
- oneskorenie medzi transakciami.

Položkami výstupnej transakcie sú:

- výsledok operácie, ktorý je reakciou na vstupné transakcie.

Samotné sekvencie sú potom homogénne, pretože sa skladajú len z jedného druhu vstupných transakcií, ktorých je však variabilný počet. Celá reprezentácia jedinca tak vytvára kolekciu štruktúr, ktorá si so sebou môže niesť ešte navyše nejaké metainformácie, ako napríklad fitness hodnota.

Obrázek 5.1: Jedinec ako sekvencia transakcií.

Čo sa týka samotnej implementácie popisovanej reprezentácie jedinca, jedinec je implementovaný ako objekt triedy `jedinec` v súbore `jedinec.sv`. Ako atribúty si jedinec so sebou nesie:

- sekvenciu transakcií,
- hodnotu fitness.

V rámci metód sú implementované:

- inicializácia,
- hlboká kópia,
- výpis,
- výpočet fitness.

Tie zaujímavejšie z uvedených atribútov a metód sú popísané v niekoľkých najbližších odsekoch. Ako prvé, sekvencia transakcií je implementovaná využitím dátového typu `queue`. `queue` [18] sa z ostatných dátových typov podobá najviac na spájaný zoznam, pretože obsahuje zoznam elementov, pričom dĺžka tohto zoznamu môže narastať, či sa zmenšovať. Elementy môžu byť vkladané a mazané nie len na počiatku a konci zoznamu, ale na ľubovoľnom mieste, čo sa pre implementovaný genetický algoritmus veľmi hodí. Element pritom

odpovedá jednej transakcii. Okrem uvedeného, `queue` taktiež disponuje ďalšou pre túto prácu užitočnou funkcionalitu a to vyhľadávanie a radenie (angl. sort) elementov. `queue` tak zachováva univerzálnosť spájaného zoznamu, no je oveľa efektívnejšia a rýchlajšia, pri veľkom počte operácií zachováva rýchlosť poľa fixnej dĺžky. Pre svoje široké spektrum výhodnej funkcionality, univerzálnosť a rýchlosť bol dátový typ `queue` zvolený na reprezentáciu sekvencie transakcií jedinca v rámci tejto diplomovej práce.

Ďalším a taktiež zaujímavým atribútom triedy `jedinec` je hodnota fitness. Hodnota fitness je v tejto realizácií genetického algoritmu počítaná rovno pri vytváraní jedinca a uchovávaná po celú dobu jeho existencie ako atribút, čo má niekoľko výhod. Hodnota fitness jedného jedinca je požadovaná za jeho život niekoľkokrát. Výpočet fitness je však časovo vysoko náročný, pretože volá beh a vyhodnotenie celej funkčnej verifikácie. Preto sa viac oplatí si túto hodnotu spočítať len raz a uchovať. Počítanie hodnoty fitness hneď pri vytváraní jedinca je taktiež dôležité, z dôvodov zlepšenia vlastností algoritmu. Podrobnejší popis čitateľ nájde v neskorších podkapitolách.

Jednou z dvoch bližšie popísaných metód je inicializácia jedinca, ktorá sa postará o vytvorenie jedného z jedincov počiatočnej populácie. V rámci tejto metódy je načítaná sekvencia transakcií a následne je v rámci inicializácie spočítaná už spomínaná fitness. Transakcie sú načítané zo súboru, v ktorom sú zapísané v presne danom formáte. Názov a cesta k súboru sú predávané metóde parametrom. Načítaná sekvencia transakcií bola pre genetický algoritmus vytvorená a do súboru zapísaná verifikačným prostredím a od tejto sekvencie je očakávané, i keď pre fungovanie algoritmu nie nevyhnutné, že má pokrytie špecifikovaných vlastností 100 %.

Druhou dôležitou metódou je výpočet fitness. Funkcia fitness je zdĺhavým vyhodnocovacím procesom, počas ktorého sa volá funkčná verifikácia pre účel zistenia pokrytia špecifikáciou daných vlastností. Aj keď fitness funkcia spadá do témy tejto podkapitoly, pre jej zložitosť je jej však venovaná neskôr osobitná podkapitola, kde je rozobratá detailnejšie.

V tomto bode je ešte vhodné uviesť príklad nejakých možných špecifikáciou daných a tým pádom verifikovaných vlastností, keďže čitateľa sprevádzajú celou prácou. Navyše je dobré tieto vlastnosti demonštrovať na konkrétnych rozhraniach DUT, ktoré boli uvedené v tejto podkapitole. Príkladom verifikovaných vlastností môžu byť:

- overenie správneho výsledku operácie sčítania pri ľubovoľnom nastavení ostatných parametrov,
- overenie správneho výsledku operácie sčítania pre nejakú konkrétnu okrajovú hodnotu oboch operandov, napríklad nulu,
- overenie správneho výsledku dvoch po sebe nasledujúcich operáciách sčítania pri ľubovoľnom nastavení ostatných parametrov,
- overenie správneho výsledku dvoch po sebe nasledujúcich operáciách sčítania pri ľubovoľnom nastavení ostatných parametrov,
- overenie či bola niekedy ako operand B využitá hodnota v pamäti a zároveň výsledok musí byť správny,
- overenie či bola niekedy ako výsledok generovaná maximálna možná hodnota výsledku a zároveň je tento výsledok správny.

Ako vidno, niektoré vlastnosti sú záležitosťou jednej transakcie, iné potrebujú napríklad presne zoradených niekoľko transakcií po sebe. Na základe toho, koľko zo špecifikovaných

vlastností bolo verifikovaných vráti verifikačné prostredie pokrytie v percentách. Pri pokrytí vlastností 100 % a všetkých výsledkoch správnych je obvod považovaný za správny.

Ďalším dôležitým pojmom genetických algoritmov je **populácia**. Populácia je množina jedincov. Táto množina má obvykle konštantnú dĺžku. Jedinci populácie sa menia každou iteráciou a mali by sa stále viac približovať globálnemu optimu riešenia zadaného problému. V tejto diplomovej práci je populácia implementovaná ako pole konštantnej dĺžky, ktorého prvky sú referenciami na objekty triedy `jedinec`. Podľa popisu genetického algoritmu v literatúre, s každou iteráciou, resp. generáciou, genetického algoritmu starú populáciu jedincov čiastočne, či úplne nahradí nová populácia. Čo sa týka implementovanej realizácie, pole referencií na jedincov sa zachová, samotné referencie sa však medzi iteráciami menia, a odkazujú len na aktuálnych jedincov. Zo starších jedincov sa však uchováva referencia aspoň na jedného a to v bočnej premennej slúžiacej pre zachovanie najlepšieho doposiaľ nájdeného riešenia.

Popisovaná a implementovaná populácia poskytuje užívateľovi niekoľko možností nastavení parametrov genetického algoritmu, ktoré už boli jemne načrtnuté v minulom odseku. Jedná sa o nastavenie veľkosti populácie, počet prenášaných jedincov z minulej populácie do novej (napr. elitizmus) ako aj ktorí jedinci bývalej populácie majú byť prenesení. Pre prenášanie minulých jedincov sú ponúkané dve možnosti a to prenos najlepších, alebo prenos náhodných jedincov minulej populácie.

Tieto parametre užívateľ nájde v zdrojovom súbore `nastavenia_a_konstanty.sv` a jedná sa konkrétne o parametre:

- `POCET_JEDINCOV`,
- `POCET_MINULYCH_ZACHOVANYCH_JEDINCOV`,
- `PRENOS_MINULYCH_JEDINCOV`.

Podrobnejší popis uvedených parametrov vrátane prijímaných hodnôt je možné nájsť v prílohe A, ktorá obsahuje zoznam všetkých nastaviteľných parametrov implementovaného genetického algoritmu.

Ďalšou implementovanou zložkou genetického algoritmu sú **genetické operácie mutácia a kríženie**, ktorých zdrojové kódy sú umiestnené v súboroch `mutacia.sv` a `krizenie.sv`. čo sa týka mutácie, sú navrhnuté a implementované až dva typy, čo pre genetické algoritmy nie je typické. Implementované typy mutácie sú:

1. mutácia typu *swap*, označovaná taktiež skrátene len ako mutácia,
2. mutácia typu *delete*.

Mutácia typu *swap* vymení vzájomné poradie dvoch náhodných transakcií vo vnútri jedného jedinca. Takáto mutácia má zmysel vzhľadom na to, že poradie transakcií je pri pokrývaní vlastností verifikáciou dôležité. Dôležitosť poradia vyplýva z faktu, že overenie niektorých vlastností môže vyžadovať napríklad presnú sekvenciu po sebe nasledujúcich transakcií. Mutácia typu *swap* je navrhovaná pre len zriedkavé využitie, teda pravdepodobnosť aplikovania tohto druhu mutácie by mala byť nízka. Aplikácia tohto druhu mutácie môže znížiť pokrytie, alebo naopak pokryť nejakú vlastnosť špecifikácie, ktorá sa skladá z viacerých transakcií, čo však nie je primárnou funkcionalitou algoritmu. Primárnou funkcionalitou je znižovanie počtu transakcií, čo mutácia typu *swap* nedokáže.

Mutácia typu *delete* naopak vymazáva určitý počet transakcií z daného jedinca. Je to tak jeden z dvoch základných nástrojov pre optimalizáciu, ktorou sa v tomto prípade

rozumie redukcia počtu transakcií pri zachovaní pokrytia. Z uvedeného dôvodu by mala byť pravdepodobnosť tohto druhu mutácie vysoká. Vysoká pravdepodobnosť mutácie pre klasické genetické algoritmy nie je typická, zato s takýmto konceptom sa môžeme stretnúť napríklad pri evolučných stratégiách. Pre účely tejto diplomovej práce sa však vysoká pravdepodobnosť mutácie popisovaného typu hodí, a preto sa tento koncept z evolučných stratégií preberie. Podrobnejšie, implementovaný princíp tohto typu mutácie je nasledovný. Nech je pravdepodobnosť mutácie typu *delete* $p_d = x$. Pri samotnej mutácií je potom každá transakcia vymazaná s pravdepodobnosťou x. Potom je po aplikovaní tohto druhu mutácie na jedinca chromozóm jedinca skrátený o približne x % transakcií.

Jazyk System Vderilog poskytuje pre implementáciu oboch navrhnutých druhov mutácie výhodné prostriedky, ktoré pri ich využití implementáciu výrazne zjednodušia. Dátový typ `queue`, v tejto diplomovej práci využívaný pre uloženie jedincovej sekvencie transakcií, ponúka prostriedky pre užívateľsky prívetivé mazanie a vkladanie elementov (transakcií) na ľubovoľné miesto v sekvencii. Ďalším výhodným prostriedkom je generátor pseudonáhodných čísel, ktorý je v jazyku System Verilog na dobrej úrovni vzhľadom na to, že generovanie pseudonáhodných čísel je obecne pre funkčnú verifikáciu jedna z kľúčových podmienok. V implementovanom genetickom algoritme sa generátor pseudonáhodných čísel využije aj pri určovaní transakcií k mutácií, či aj samotnom rozhodnutí o použití mutácie na jedinca.

Genetická operácia kríženia vzájomne vymení dve ľubovoľne dlhé a ľubovoľne umiestnené subsekvencie transakcií medzi dvomi jedincami. Tieto subsekvencie môžu a pravdepodobne sú vzájomne rôzne dlhé a umiestnené na vzájomne rôznych pozíciách. Inými slovami index prvých, ako aj index posledných, krížených transakcií je pravdepodobne rôzny. Pre implementáciu kríženia sa využívajú rovnaké výhody jazyka System Verilog ako u mutácie. Jedná sa konkrétne o generátor pseudonáhodných čísel a dátový typ `queue` umožňujúci vkladanie a vymazávanie transakcií na ľubovoľnú pozíciu sekvencie.

Čo sa týka podrobnejšieho princípu implementovaného kríženia a zdôvodnenie voľby použitého prístupu, ich vysvetleniu sa venuje tento odsek. V bežne používanom prístupe sú vzájomne vymieňané rovnako veľké úseky chromozómov. Potom po aplikovaní kríženia majú tak chromozómy jedincov rovnakú dĺžku ako pred touto genetickou operáciou. V optimalizačnej úlohe, ktorá sa snaží chromozómy (sekvencie transakcií) skrátiť však takýto druh kríženia príliš veľký zmysel nemá. V použitom genetickom algoritme sa tak použije kríženie vymieňajúce dva ľubovoľne dlhé úseky chromozómov, ktoré môžu byť, ale s vysokou pravdepodobnosťou nie sú rovnako dlhé. Krížením sa tak obecne dĺžka chromozómu jedného jedinca skráti a dĺžka chromozómu druhého predĺži. Za predpokladu zachovania pokrytia špecifikovaných vlastností 100 % sa tak obvykle hodnota fitness jedného jedinca zlepší a druhého zhorší. Celkovo však môžu nastať tri prípady:

1. skrátenie chromozómu kratšieho jedinca a predĺženie chromozómu dlhšieho jedinca,
2. skrátenie chromozómu dlhšieho jedinca a predĺženie chromozómu kratšieho jedinca,
3. zachovanie dĺžky chromozómov.

Prípady rozoberieme pre prípad zachovania pokrytia vlastností 100 %, pretože inak sú všetky prípady rovnako zlé. Prvý z prípadov sa javí pri zachovaní pokrytia byť lepší, pretože za cenu straty aj tak nie príliš dobrého jedinca je získané nové optimalizované riešenie. Druhý prípad len čiastočne vyrovná fitness oboch krížených jedincov, čo z pohľadu optimalizácie nie je krok dopredu. Výskyt druhého prípadu je však pre ľubovoľné vymieňané úseky transakcií o niečo pravdepodobnejšie než výskyt prvého a to z nasledujúceho dôvodu. Nech

je dĺžka chromozómu prvého a zároveň viac optimálneho jedinca n. Dĺžka chromozómu druhého jedinca je $m = n+k$. Nech sú `index1` a `index2` náhodne volené poradové čísla transakcií, ktoré vymedzujú vymieňaný úsek sekvencie transakcií. Pravdepodobnosť voľby nejakej transakcie ako `index1`, prípadne `index2` je v prvom jedincovi $1/n$ a v druhom $1/m$, pričom evidentne $1/n > 1/m$. Potom je u kratšieho jedinca pravdepodobnejšie, že oba vymedzujúce transakcie sa budú vyskytovať len v kratšom rozsahu transakcií. Príklad môže byť úsek dĺžky m. Pravdepodobnosť, že sa obe vymedzujúce transakcie nachádzajú v tomto úseku je v prvom jedincovi 1. U druhého jedinca je pravdepodobnosť, že pre daný `index1` sa bude `index2` vyskytovať vo vzdialenosti maximálne m v intervale $\langle(1/m)^{m-2m}, (1/m)^{m-m}\rangle < 1$, presná hodnota záleží na vzdialenosti `index1` od okrajov sekvencie. Odstraňovaný úsek transakcií z kratšieho jedinca obsahuje tak pravdepodobne menej transakcií, než odstraňovaný úsek transakcií z dlhšieho jedinca a preto má kríženie tendenciu viac zrovnávať jedincov, než hľadať optimum. Pravdepodobnosť nastania tretieho prípadu je nízka a ako už bolo uvedené, výmena sekvencií rovnakej dĺžky ani nemá veľký zmysel.

Sekundárnym prínosom kríženia je možnosť zavedenia nových zaujímavých transakcií do chromozómu jedinca. Hlavným prínosom kríženia však ostáva možnosť optimalizácie jedného z dvojice jedincov, tá je však za cenu zhoršenia druhého z dvojice. Kríženie sa tak zdá byť menej efektívne než druhý z prostriedkov pre optimalizáciu, ktorým je mutácia typu *delete*. Na rozdiel od tohto typu mutácie však kríženie neodstraňuje transakcie po jednej ale po blokoch, čo je rozdielny prístup a môže sa ho občas hodiť použiť. Posudzovanie úspešnosti kríženia je však v tomto bode skôr výpočtom očakávaní, zistenie skutočnej úspešnosti kríženia sa zaistí až meraním výsledkov počas behu optimalizácie.

V tomto bode je ešte dôležité zvýrazniť fakt, že všetky tri implementované genetické operácie, tj. kríženie a oba typy mutácie, nedokážu narábať s menšou jednotkou chromozómu, než je transakcia. Táto myšlienka je dôležitá pre univerzálnosť a lepšiu použiteľnosť algoritmu, kde genetický algoritmus nepotrebuje mať presný prehľad o formáte transakcie, štruktúre hardvérového obvodu, či detajloch verifikačného prostredia.

Všetky tri genetické operácie sú implementované ako funkcie. Popisovaná implementácia genetických operácií kríženia a mutácie priniesla niekoľko nastaviteľných parametrov. Nastaviteľnými parametrami týkajúcimi sa genetických operácií sú:

- `PRAVDEPODOBNOST_MUTACIE`,
- `PRAVDEPODOBNOST_MUTACIE_DELETE`,
- `PRAVDEPODOBNOST_KRIZENIA`.

Tieto parametre užívateľ nájde v zdrojovom súbore `nastavenia_a_konstanty.sv`. Bližší popis uvedených parametrov, ako aj povolené hodnoty sú uvedené v zozname všetkých nastaviteľných parametrov implementovaného genetického algoritmu A.

Po popise jedinca, populácie a genetických operáciách táto podkapitola popisuje ešte jednu implementovanú zložku genetického algoritmu, ktorou je **selekcia**. V prípade selekcie riešená optimalizačná úloha taktiež vyžaduje niekoľko špeciálnych úprav existujúcich prístupov. Selekčných algoritmov je implementovaných niekoľko, keďže literatúra nedokáže presne určiť ktorý je obecne najlepší, dajú sa nájsť len domnienky. Zo selekčných algoritmov sú v tejto diplomovej práci implementované vybrané tri:

1. ruleta, ktorá patrí k najznámejším a najzákladnejším selekčným algoritmom a ktorá využíva pomeru hodnôt fitness jedinca oproti celkovej fitness populácie,

2. ruleta využívajúca poradie jedinca v populácií miesto klasického pomeru hodnôt fitness, pretože toto rozšírenie údajne disponuje oproti klasickej rulete niekoľkými výhodami,

3. turnajová selekcia, na ktorú sa dá nájsť mnoho pozitívnych ohlasov a sama o sebe je navyše nastaviteľná, takže sa dá prispôsobiť konkrétnej situácií.

Najvýraznejšou špeciálnou úpravou pre selekciu je v implementovanom algoritme úprava výpočtu pravdepodobnosti zvolenia jedinca ako rodiča v klasickej rulete. Tu je bežne pravdepodobnosť zvolenia jedinca rovná fitness jedinca vydelená sumou fitness všetkých jedincov (celková fitness populácie). V implementovanom prípade je však priebeh fitness funkcie opačný oproti bežnej situácií a teda správny jedinec má nízku fitness. To by znamenalo, že zvolenie optimálneho jedinca ako rodiča je menej pravdepodobné, než zvolenie zlého jedinca. Princíp zmeny algoritmu tak pozostáva z mapovania fitness hodnôt jedincov na iné hodnoty tak, aby platilo, že čím je jedinec lepší, tým je mapovaný na vyššiu hodnotu. Nazvyme túto hodnotu *fitness 2*. Zvyšok základného princípu výpočtu pravdepodobnosti selekcie jedinca ostáva obdobný a to ako pomer fitness jedinca po mapovaní oproti sume všetkých mapovaných fitness populácie. Výpočet pravdepodobnosti p_x zvolenia jedinca x z množiny všetkých jedincov X je tak navrhnutý a modifikovaný na 5.1 a 5.2:

$$eval_2(x) = \begin{cases} min_fitness + min_fitness - eval(x) & \text{pri pokrytí} = 100\,\% \\ 0\text{, prípadne iná konštanta} & \text{pri pokrytí} < 100\,\% \end{cases} \tag{5.1}$$

$$p_x = eval_2(x)/F_2 \tag{5.2}$$

kde: $max_fitness$ je fitness najhoršieho jedinca, ktorý ale pokrýva 100 % špecifikáciou daných vlastností, $min_fitness$ je fitness najlepšieho jedinca, $eval(x)$ je fitness daného jedinca, $eval_2(x)$ je *fitness 2* daného jedinca a F_2 je celková nová fitness populácie taká, že platí vzorec 5.3.

$$F_2 = POCET_SPARVNYCH_JEDINCOV \cdot (min_fitness + max_fitness) - F \tag{5.3}$$

V 5.3 $POCET_SPRAVNYCH_JEDINCOV$ vyjadruje počet jedincov populácie pokrývajúcich 100 % vlastností a F, popísaná vzorcom 5.4, je celková pôvodná fitness populácie bez jedincov nepokrývajúcich 100 % vlastností, ktorých hodnota fitness je tak vysoká, že by pri započítaní pokrivila celkový výpočet.

$$F = \sum_{\forall y \in Y} eval(y), Y = \{x | x \in X \wedge x \text{ pokrýva } 100\,\% \text{ vlastností}\} \tag{5.4}$$

Odvodenie vzorca F_2 bolo založené na tom, aby platilo 5.5.

$$\sum_{\forall x \in X} p_x = 1 \tag{5.5}$$

Zostáva už len zdôvodniť, prečo bol zvolený uvedený prepočet a čo daný prepočet robí. Navrhovaná *fitness 2*, prevráti pôvodnú hodnotu fitness okolo aritmetického priemeru *stred* 5.6 najnižšej a najvyššej hodnoty fitness populácie, pričom do úvahy sa berú len fitness jedincov, ktorí pokrývajú 100 % vlastností.

$$stred = (min_fitness + max_fitness)/2 \tag{5.6}$$

Nepokrývajúci jedinci sú z výpočtu vylúčení, pretože ich hodnota fitness sa extrémne odlišuje od ostatných, čo by mohlo výrazne ovplyvniť výsledky. *fitness 2* nepokrývajúcich jedincov je potom primárne 0, prípadne iná nízka konštanta. *fitness 2* pokrývajúceho jedinca s pôvodne najhoršou fitness sa tak stáva najnižšia hodnota fitness populácie. *fitness 2* najlepšieho jedinca naopak nadobudne hodnotu rovnú najvyššej fitness populácie. Čitateľa môže napadnúť, že prečo sa hodnoty neotáčajú okolo napríklad mediánu alebo modusu fitness jedincov populácie, odpoveď je tá, že takéto riešenie by mohlo dodať záporné hodnoty *fitness 2*, čo je problém.

Druhým typom selekčného algoritmu je ruleta vyberajúca jedincov podľa ich poradia v populácií. Výhodou tohto druhu rulety je to, že oproti klasickej rulete sa lepšie vyrovná s odľahlými hodnotami fitness a taktiež lepšie zachová diverzitu populácie i v prípade, ak je jeden jedinec populácie výrazne lepší, než ostatní. Výpočet pravdepodobnosti selekcie jedinca je navrhnutý a implementovaný ako 5.7 a 5.8.

$$menovatel = \sum_{i=1}^{POCET_JEDINCOV} i \tag{5.7}$$

$$p_i = \frac{poradie}{menovatel} \tag{5.8}$$

kde p_i označuje pravdepodobnosť selekcie daného jedinca, $poradie$ je jeho umiestnenie v rámci populácie a $POCET_JEDINCOV$ je veľkosť populácie.

Čo sa týka samotnej implementácie, v rámci oboch typov rulety sa využíva pole kumulatívnych pravdepodobností tak, ako bolo vysvetlené v teoretickom úvode 2.3.1. Ruleta podľa poradia navyše využíva poskytované radenie nad dátovým typom `queue`, podľa ľubovoľnej položky, ktorou je v tomto prípade hodnota fitness. Radenie je rýchle, keďže sa jedná len o zoradenie referencií odkazujúcich na konkrétnych jedincov s uchovávanými a dopredu spočítanými fitness hodnotami. Poradie v `queue` potom odpovedá poradiu v populácií.

Poslednou metódou selekcie je turnajová selekcia s možnosťou nastavenia počtu vyberaných jedincov na jedného potomka. Tento druh selekcie je implementovaný podľa popisu v teoretickom úvode 2.3.1 a neprináša tak žiadne vlastné vylepšenia.

Selekcia zaviedla dva nové parametre genetického algoritmu. Jedná sa o výber selekčného algoritmu a bližšie nastavenie turnajovej selekcie, ktoré je samozrejme využité iba v prípade voľby tohto algoritmu selekcie. Podrobnejší popis parametrov je v A, na tomto mieste sú uvedené len názvy týchto parametrov:

- `METODA_SELEKCIE`,
- `TURNAJ_POOL_SIZE`.

5.4 Fitness funkcia

Pri voľbe fitness funkcie pre účely optimalizácie regresných testov treba brať do úvahy dve základné požiadavky:

1. čím kratší chromozóm, tj. čím menej vstupných transakcií pre verifikáciu, tým lepšie,
2. jedinci populácie, ktorí nesplňujú 100 % pokrytie vlastností sú nežiaduci.

Navrhnutá a implementovaná fitness funkcia $f_{fitness}$, ktorá splňuje uvedené požiadavky je je uvedená v 5.9.

$$f_{fitness} = \begin{cases} \text{počet vstupných vektorov pre verifikáciu} & \text{pri pokrytí} = 100\,\% \\ \text{maximum použitého dátového typu} & \text{pri pokrytí} < 100\,\% \end{cases} \quad (5.9)$$

Je zrejmé, že čím lepší jedinec, tým nižšiu hodnotu fitness funkcie nadobudne.

Použitá fitness funkcia sa tak javí správna, pretože uprednostňuje kratších a teda viac optimálnych jedincov a výrazne znevýhodňuje nežiadaných jedincov s pokrytím vlastností menším než 100 %. O výber jedincov s dobrou fitness do ďalších generácií a pomerne rýchle vymiznutie nežiadaných jedincov s pokrytím vlastností menším než 100 % sa vďaka ich výrazne horšiemu ohodnoteniu následne postarajú selekčné mechanizmy. Dôvod pre odstraňovanie jedincov nepokrývajúcich 100 % vlastností je ten, že vytvárané regresné testy netestujúce všetky špecifikované vlastnosti nespĺňajú zadané požiadavky a sú teda bezcenné. Navyše implementovaný optimalizačný algoritmus sa zameriava na optimalizáciu jedincov s pokrytím 100 % a nie na opravu nekompletných jedincov.

Úloha zvyšovania pokrytia je ďalší komplexný problém, ktorý by nachádzanie optimálneho riešenia pravdepodobne aj tak výrazne spomalil. Navyše nie len, že je úloha zvyšovania pokrytia zložitá, samotné zvýšenie pokrytia môže prinášať pridávanie transakcií do jedinca a teda predĺženie jeho chromozómu. To spôsobuje stratu konkurenčnej výhody a zníženie pravdepodobnosti prežitia tohto jedinca do ďalších generácií. V implementovanej verzii genetického algoritmu sa však môže podariť zvýšiť pokrytie jedinca, je to však veľmi nízko pravdepodobné. Môže sa napríklad stať, že:

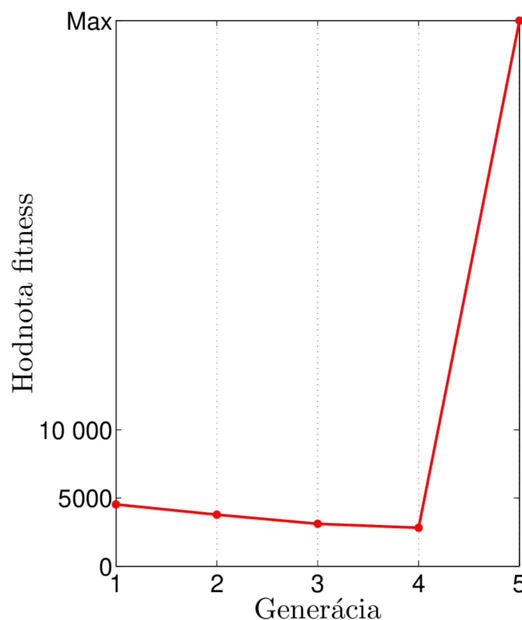
- Novo vzniknutý problém zníženia pokrytia v genetickej operácií kríženia bude zvrátený ešte v tej istej iterácií následnou operáciou mutácie.
- Ďalšou možnosťou je použitie jedinca s nízkym pokrytím pre vytváranie ďalšej generácie, kde by potenciálne mohli prispieť k zvýšeniu pokrytia ako kríženie, tak aj mutácia.

Takéto udalosti nie sú nemožné, ale sú však nízko pravdepodobné, pretože kríženie a mutácia boli primárne navrhnuté pre iné účely a to skracovanie chromozómov.

Vráťme sa však späť k navrhovanej fitness funkcii. Uvádzaná fitness funkcia teda populáciu z vysvetlených dôvodov rýchlo zbaví zlých jedincov nepokrývajúcich 100 % vlastností. Výhodu rýchleho odstránenia nežiadaných jedincov pomocou extrémnej hodnoty fitness však sprevádza aj nevýhoda, ktorou je skokový priebeh fitness funkcie. Príklad skokového priebehu fitness funkcie je znázornený na obrázku 5.2. Problém je ten, že i po prevedení jednej mutácie, či kríženia môžu aj potomkovia rodičov s relatívne dobrou hodnotou fitness prestať spĺňať 100 % pokrytie vlastností a to aj v prípade zrušenia jedinej transakcie. Dokonca, čím majú rodičia lepšiu fitness, tým sú z nich vznikajúci noví jedinci náchylnejší na zníženie pokrytia. Vzniku nechcených jedincov nepokrývajúcich 100 % vlastností sa však nedá úplne zabrániť.

Uvedený problém skokového priebehu fitness funkcie nie je nevyhnutné riešiť, pretože o elimináciu jedincov so zlým ohodnotením fitness by sa mali postarať v niekoľkých iteráciách už spomínané selekčné mechanizmy. O jedincoch nesplňujúcich 100 % pokrytie vlastností, však vieme okamžite naisto povedať, že ich nechceme použiť pre vytváranie ďalších generácií. Jedinci nesplňujúci 100 % pokrytie vlastností nie sú teda pre nachádzanie riešenia využívaní a tak zbytočne zaberajú miesto v populácií jedincom s dobrou fitness.

Napriek tomu, že určitý počet nevyhovujúcich jedincov v populácií pravdepodobne neznamená vážny problém, prináša určitú nevýhodu. Jedná sa o zmenšenie rôznorodosti



Obrázek 5.2: Príklad priebehu priemernej hodnoty fitness funkcie v genetickom algoritme s veľkosťou populácie 1.

populácie. Zmenšenie rôznorodosti populácie môže ďalej viesť k pomalšiemu nachádzaniu riešenia, či nájdeniu horšieho riešenia. I keď vznik jedincov s najhorším možným ohodnotením nejde vylúčiť, je možné ho do určitej miery obmedziť. Prípadne je taktiež možné sa už s vytvorenými jedincami s najhorším možným ohodnotením inak vyrovnať. Uvedenému problému sa venuje celá podkapitola 5.5.

Ďalším problémom, ktorý si treba uvedomiť a sprevádza vyhodnocovanie fitness jedincov je dĺžka výpočtu fitness. Dôvod je ten, že vyhodnotenie fitness vyžaduje spustenie a prebehnutie celej funkčnej verifikácie, čo je časovo náročné. Tento problém je závažný tak, že všetky ostatné náročné operácie a funkcie sú vzhľadom na dĺžku behu v porovnaní s výpočtom fitness zanedbateľné. Uvedený problém sa však taktiež ponúka ako dobrý prostriedok na meranie zložitosti optimalizačného algoritmu. Porovnanie dĺžky behu dvoch rôznych optimalizačných procesov v implementovanom optimalizačnom prostredí potom môže byť uskutočnené aj porovnaním počtu vyhodnotení fitness. Napriek tomu, že dĺžka vyhodnocovania fitness ostane časovo najnáročnejšou operáciou, dá sa vhodným nastavením jej dĺžka behu skrátiť. Urýchlenie behu funkčnej verifikácie v ModelSime je možné uskutočniť pomocou spustenia verifikácie bez grafického rozhrania, či obmedzenia textových výstupov. S nachádzaním a vyhodnocovaním viac optimálnych a teda kratších jedincov sa dĺžka vyhodnocovania fitness taktiež skracuje.

5.5 Jedinci s nedostatočným pokrytím vlastností

Ako už bolo popisované v podkapitole venujúcej sa implementácií fitness funkcie 5.4, o jedincoch vieme od ich vzniku usúdiť či sú potenciálne vhodnými kandidátmi pre hľadanie optimálneho riešenia, alebo naopak, že takýto jedinci ako zdroj riešenia veľmi pravdepodobne neposlúžia. Nevhodnými jedincami pre hľadanie sady optimálnych regresných testov

v tejto diplomovej práci sú jedinci nepokrývajúci 100 % vlastností. Fakt, že nevhodných jedincov je možné identifikovať okamžite, ponúka veľkú výhodu. Využitím tejto výhody vznikli dve rozšírenia týkajúce sa nežiadaných jedincov s pokrytím nižším než 100 %:

1. Prvé rozšírenie vychádza z faktu, že znalosť fitness hodnoty a teda nevhodnosti jedinca hneď po jeho vytvorení umožňuje opätovné pregenerovanie tohto jedinca. Inými slovami, toto rozšírenie dáva šancu sa nežiadanému jedincovi opraviť. Počet pokusov na znovu vygenerovanie jedinca v danej generácií je však limitovaný. Dôvodom je možnosť zacyklenia alebo aspoň prevedenie neprijateľne veľkého počtu pregenerovaní pri nesprávnom nastavení parametrov. Nesprávne nastavenie parametrov vedie ku druhému rozšíreniu týkajúceho sa nežiadaných jedincov s pokrytím nižším než 100 %.

2. Druhé rozšírenie vychádza z úsudku, že s postupným zlepšovaním a teda skracovaním chromozómov jedincov je stále zložitejšie z nich odstraňovať, či v nich vymieňať transakcie tak, aby pokrytie vlastností ostalo zachované a teda fitness hodnota nestúpla na najhoršiu možnú hodnotu. A preto aj s nastaveniami genetického algoritmu, ktoré sa pri prvých niekoľkých iteráciách ukázali byť vhodné, môže mať genetický algoritmus po čase problém. Algoritmus sa prípadne môže až predčasne zastaviť na podmienku ukončenia výpočtu, ktorou je dlhodobé nenachádzanie lepšieho riešenia. Výsledkom je potom stále vysoko redundantná a tým pádom neoptimálna sada regresných testov.

   Daný úsudok sa opiera o nasledujúcu úvahu. Nech je pravdepodobnosť mutácie *delete* p. Potom je po aplikovaní tohto typu mutácie z jedinca vymazaných približne $100 \cdot p\%$ transakcií. Nech hľadaný optimálny jedinec sa skladá zo sekvencie m transakcií. Ďalej existujú dvaja jedinci N a O s počtom transakcií n a o tak, že $m < n < o$. To znamená, že počet redundantných transakcií sa v prípade jedinca N rovná $n - m$ a v prípade jedinca O rovná $o - m$. Pritom jasne platí, že $n - m < o - m$, tj. jedinec O je viac redundantný než N. Ďalej platí že, pomer redundantných ku celkovému počtu transakcií je pre N $(n - m)/n$ a pre O $(o - m)/o$ a zároveň $(n - m)/n < (o - m)/o$. U oboch jedincov je vymazávané približne rovnaké percento všetkých transakcií, u jedinca N sa vymazáva okolo $n \cdot p$ transakcií a u O $o \cdot p$, pričom $n \cdot p < o \cdot p$. A teda u redundantnejšieho jedinca je očakávané, že je vymazávaných viac transakcií. p_n a p_o, ktoré udávajú pravdepodobnosti, že všetky mazané transakcie sú redundantné sú popísané vzorcami 5.10 a 5.11.

$$p_n = \frac{\frac{(n-m)!}{(n-m-n \cdot p)!}}{\frac{n!}{(n-n \cdot p)!}} \tag{5.10}$$

$$p_o = \frac{\frac{(o-m)!}{(o-m-o \cdot p)!}}{\frac{o!}{(o-o \cdot p)!}} \tag{5.11}$$

   Keďže $n < o$, tak aj $p_n < p_o$ a teda prevádzanie mutácie *delete* bez adaptívnej zmeny parametrov je so zlepšujúcimi sa jedincami stále zložitejšie.

   Rovnako sa dá ukázať zvýšená problematickosť zmeny či vymazania istého počtu transakcií pomocou všetkých genetických operácií. Kde si postupne za p z predošlej ilustrácie predstavíme $100 \cdot p\,\%$ menených či vymazávaných transakcií pomocou inej genetickej operácie.

   V predošlých odsekoch sa teda dokázalo, že prvé iterácie algoritmu fungujú efektívnejšie s vyššími pravdepodobnosťami genetických operácií, neskoršie iterácie sú zas efektívnejšie pre nižšie. Pre riešenie uvedeného problému sa ponúkajú dve riešenia:

- Nízke pravdepodobnosti genetických operácií počas všetkých iterácií. Takéto riešenie môže riešiť problém predčasného ukončenia výpočtu. Riešenie je však neefektívne vzhľadom na počet vyhodnotení fitness, čo je časovo kritická operácia.
- Druhou navrhovanou a zároveň implementovanou možnosťou je špeciálny druh **samo-adaptácie**. Myšlienkou riešenia je efektivita a teda čo najnižší počet vyhodnotení fitness pri čo najväčšom zoptimalizovaní jedincov. Toto riešenie využíva vysokú redundanciu úvodnej a niekoľko ďalších populácií v svoj prospech. To konkrétne odstraňovaním či výmenou vysokého počtu transakcií v rámci na jedno vyhodnotenie fitness počas úvodných iterácií algoritmu. Zároveň v neskorších iteráciách realizuje jemnejšie zásahy do jedincov. Optimalizačný krok sa tak v priebehu iterácií zmenšuje a to automaticky vzhľadom na celkový stav populácie. Správny okamih na zníženie pravdepodobností genetických operácií sa zisťuje určitým pomerom obsadenia nevyhovujúcich jedincov v celkovej populácií, ktorí sa tam postupne nazbierali aj napriek niekoľkým pokusom pregenerovania, čo je signál pre zníženie nárokov na množstvo odstraňovaných či vymieňaných transakcií. Po znížení veľkosti optimalizačného kroku sa selekčné mechanizmy spolu s fitness funkciou postarajú o opätovné nahradenie nevyhovujúcich jedincov vyhovujúcimi.

  Faktom však ostáva, že ani toto riešenie nerieši problém straty rôznorodosti populácie úplne. Dôvodom je nutnosť zabratia miesta v aspoň časti populácie nevyhovujúcimi jedincami pre detekciu zlých pravdepodobností genetických operácií a teda strata niekoľkých vyhovujúcich jedincov. Napriek neskoršiemu opätovnému naplneniu populácie vyhovujúcimi jedincami, sú noví jedinci primárne generovaní len z ostávajúceho obmedzeného počtu vyhovujúcich rodičov. Potom skupinky jedincov prehľadávajú blízke miesta prehľadávaného priestoru, miesto roztrúsenia sa po celom priestore. Jav určitej straty rôznorodosti populácie nie je pre genetické algoritmy netypický a do určitej nízkej miery môže byť aj žiadaný, no vo vyššej miere môže spôsobiť problém. Náchylnosť k strate rôznorodosti tak treba brať do úvahy pri nastavovaní parametrov.

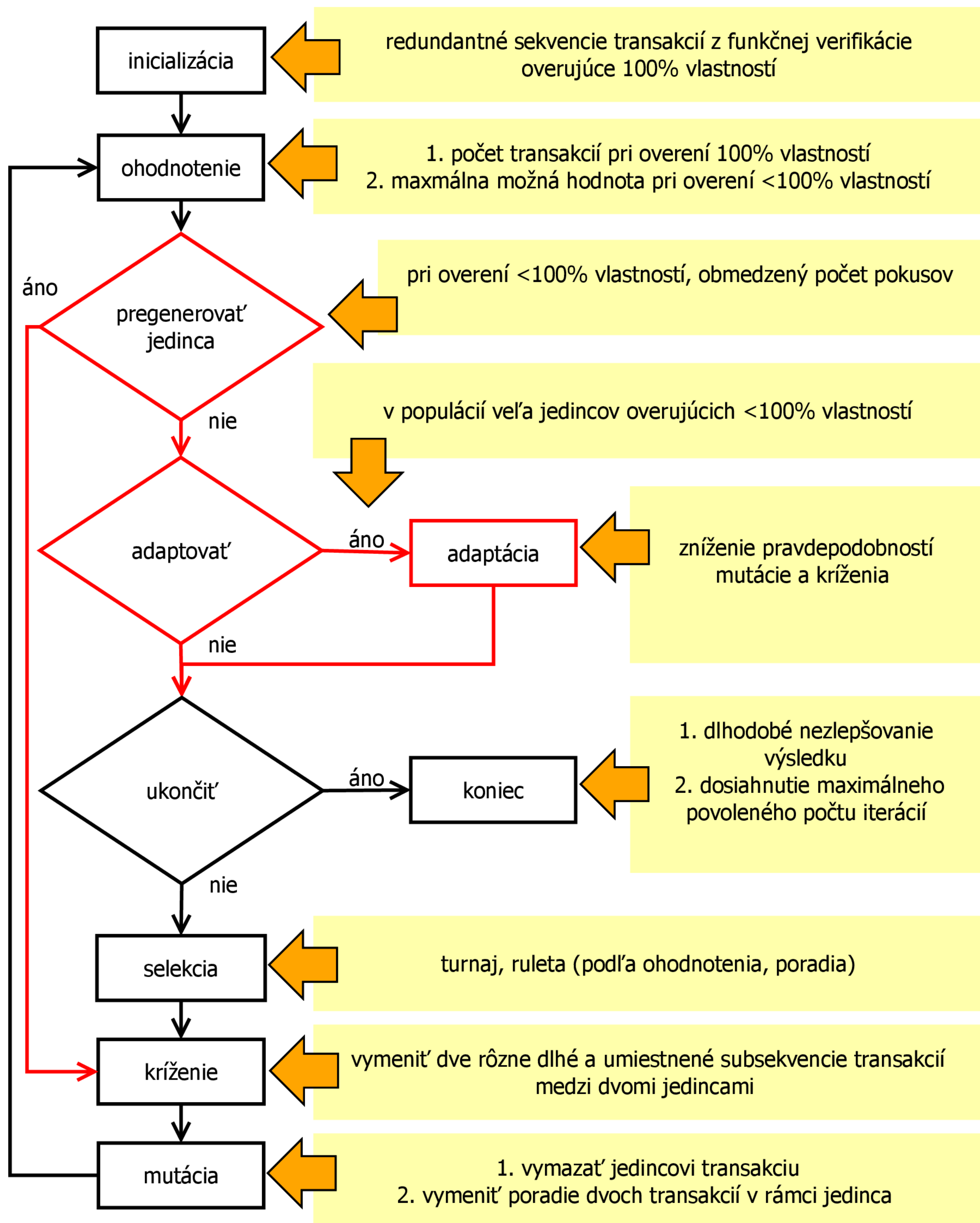
  Napriek náchylnosti k určitej strate rôznorodosti toto riešenie má potenciál priniesť mnoho veľkých výhod, ako je efektivita, možnosť zotavenia sa zo zlých nastavení pravdepodobností genetických operácií a dynamické prispôsobovanie si uvedených pravdepodobností.

  Ďalším zaujímavým faktom o navrhovanom riešení je aj to, že toto riešenie spája koncept genetického algoritmu s evolučnými stratégiami. Kde je zachovaný koncept jedinej pravdepodobnosti vykonania mutácie a kríženia podľa genetického algoritmu, samo-adaptácia je prebratá z evolučných stratégií.

  Ako doplnok k implementovanej samo-adaptácií je ešte doprogramovaná možnosť nadefinovania spodných hraníc pravdepodobností mutácie a kríženia. Algoritmus tak nemôže znížiť pravdepodobnosti pod túto hodnotu. Dôvody, prečo by si užívateľ mohol priať obmedziť spodnú hodnotu pravdepodobností genetických operácií tu napríklad takéto:

  – Pokiaľ algoritmus nedokáže odstraňovať aspoň nejaké percento transakcií, je pravdepodobné, že nájdené riešenie je už dostatočne dobré.
  – Vysoká časová náročnosť algoritmu. Nízke pravdepodobnosti genetických operácií vedú k nižšej efektivite a teda veľkému počtu vyhodnocovaní fitness

funkcií pri malom počte odstraňovaných transakcií. Užívateľ sa tak môže rozhodnúť, že sa mu viac oplatí menej optimálne riešenie.



Obrázek 5.3: Implementovaný genetický algoritmus.

Pre riešenie problému jedincov nesplňujúcich pokrytie 100 % sú teda implementované dve rozšírenia a to: možnosť pregenerovania jedinca a špeciálny druh samo-adaptácie s možnosťou spodných obmedzení. Na obrázku 5.3 je uvedený vývojový diagram kompletizovaného genetického algoritmu aj so stručnými popismi jednotlivých krokov pre pripomenutie. Obrázok mimo iného ukazuje, kam do programovej slučky boli zaradené nové rozšírenia, ktoré sú na obrázku znázornené červenou farbou.

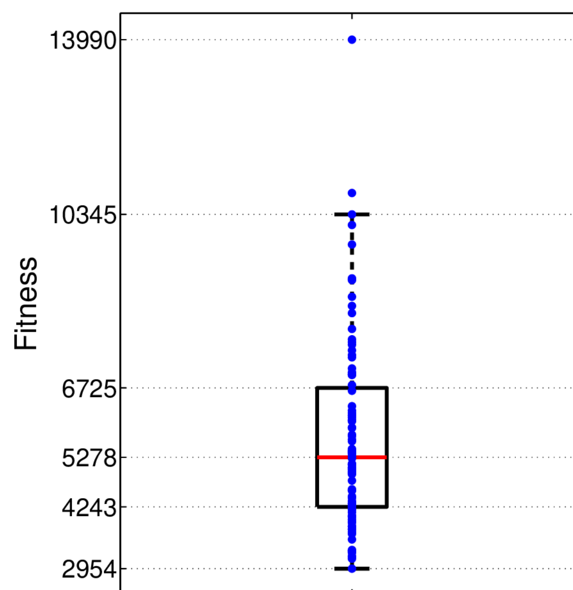
Popisované nové rozšírenia priniesli do implementácie genetického algoritmu niekoľko nových nastaviteľných parametrov. Pridané nastaviteľné parametre definujú minimálne pravdepodobnosti genetických operácií pre samo-adaptáciu, ďalej definujú po akom kroku sú pravdepodobnosti znižované a podmienku, kedy ku zníženiu dochádza. Pre účely možnosti opätovného generovania jedinca pri nepokrytí 100 % vlastností je definovaný maximálny počet pokusov generovania jedinca. Tieto parametre užívateľ nájde v zdrojovom súbore `nastavenia_a_konstanty.sv` a jedná sa konkrétne o parametre:

- `PRAH_ZMENSENIA_PRAVDEPODOBNOSTI`,
- `ZMENSENIE_PRAVDEPODOBNOSTI`,
- `MIN_PRAVDEPODOBNOST_MUTACIE`,
- `MIN_PRAVDEPODOBNOST_MUTACIE_DELETE`,
- `MIN_PRAVDEPODOBNOST_KRIZENIA`,
- `MAX_POCET_POKUSOV_GENEROVANIA_POTOMKA`.

Podrobnejší popis uvedených parametrov vrátane prijímaných hodnôt je možné nájsť v prílohe A, ktorá obsahuje zoznam všetkých nastaviteľných parametrov implementovaného genetického algoritmu.

Kapitola 6

Namerané výsledky



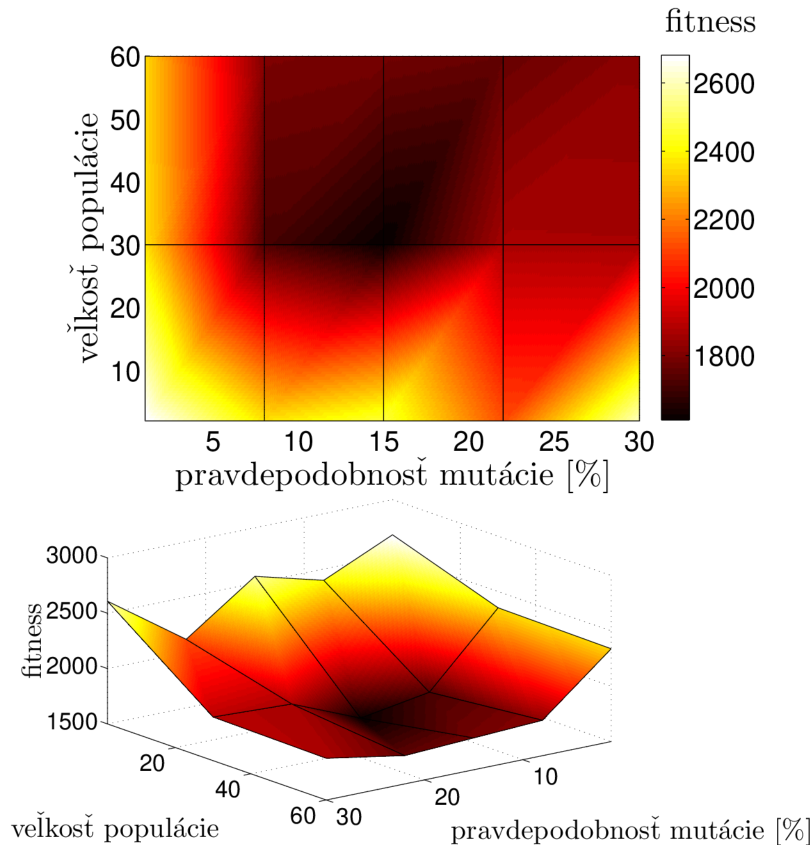
Obrázek 6.1: Fitness hodnoty jedincov počiatočnej populácie.

Pre účely experimentov a meraní na implementovanom optimalizačnom prostredí bola vygenerovaná počiatočná populácia pomocou verifikačného prostredia. Aby boli generovaní jedinci rôzni, bolo potrebné pre každého jedinca nastaviť rôzny *seed* generátora náhodných čísel v generujúcom verifikačnom prostredí. Celkovo popisovaným spôsobom vzniklo 100 jedincov. Rozloženie fitness hodnôt týchto jedincov je možné vidieť na obrázku 6.1, na ktorom je zobrazený krabicový graf. Z grafu je mimo iného vidno hodnotu fitness každého z jedincov, ktorá je ilustrovaná ako modrá bodka. Ďalej z grafu vidno aj najhoršiu a najlepšiu fitness vygenerovaných jedincov, ako aj medián. Maximálna fitness je 13990, minimálna 2954, medián 5278. Čo z krabicového grafu nevidno, je priemerná hodnota fitness, ktorá je rovná 5653,52.

Ďalej, medzi už zrealizovaným získaním počiatočnej populácie a samotným nameraním výsledkov optimalizácie je potrebné uskutočniť ešte jeden náročný krok. Tým krokom je empirické zistenie vhodných parametrov genetického algoritmu. S nesprávnym nastavením parametrov totiž algoritmus pravdepodobne nenájde hľadané riešenie. Navyše neexistuje iný

spôsob, akým sa dá efektívnejšie správne nastavenie parametrov nájsť. Zisťovanie správnych parametrov genetického algoritmu empiricky je však taktiež náročná úloha. Problémom je, že priestor parametrov genetického algoritmu a ich všetkých možných hodnôt je príliš veľký. Preto je potrebné vymyslieť nejaký systematický prístup a rovnako aj vedieť do určitej miery správanie algoritmu odhadnúť.

Ako prvé, bola preskúmaná vhodná veľkosť populácie spolu pravdepodobnosťou genetických operácií, ktoré majú vplyv na skracovanie dĺžky chromozómu. Uvedenými genetickými operáciami sú mutácia typu *delete* a kríženie.



Obrázek 6.2: Dosiahnutá hodnota fitness po 20 generáciách u mutácie *delete*.

Z uvedených operácií je prvá popísaná mutácia typu *delete*. V prevedenom experimente bol skúmaný vzťah pravdepodobnosti mutácie typu *delete* a veľkosti populácie, kde sa hľadalo optimálne vzájomné nastavenie oboch parametrov. Boli vyskúšané všetky kombinácie nastavenia oboch týchto parametrov pre hodnoty:
`PRAVDEPODOBNOST_MUTACIE_DELETE` $\in \{0,01; 0,08; 0,15; 0,22; 0,3\}$,
`POCET_JEDINCOV` $\in \{2, 30, 60\}$.
Presný popis oboch nastaviteľných parametrov čitateľ nájde v prílohe A. Experiment prebehol na pevnom počte iterácií (20) a ako porovnateľný výsledok sa zisťovala dosiahnutá fitness najlepšieho jedinca ako aj počet vyhodnotení fitness hodnoty, čo je časovo kritická operácia.

Výsledky dosiahnutej fitness sú znázornené na obrázku 6.2. Najtmavším miestam grafu odpovedajú najlepší nájdení jedinci. Vidno, že s väčšou populáciou sú nachádzaní lepší

jedinci. Najvhodnejšia pravdepodobnosť mutácie *delete* sa javí byť niekde medzi 0,15 až 0,2.

Na presnejšie určenie vhodných parametrov, je potrebné sa pozrieť na presné namerané hodnoty, koré sú uvedené v tabuľkách. Dosiahnutú fitness nájde čitateľ v 6.1 a počet vyhodnocovaní fitness v 6.2. Tu treba ešte zdôrazniť, že počet vyhodnotení fitness ovplyvňuje niekoľko zložiek, ako počet iterácií algoritmu a veľkosť populácie, tak aj počet opakovaných pregenerovaní jedincov. Vzhľadom na to, že počet opakovaných pregenerovaní jedincov závisí na celkovej správnosti nastavenia všetkých parametrov genetického algoritmu, tak sa správnosť týchto parametrov prejaví aj na meranom počte vyhodnotení fitness v tabuľke 6.2.

Z uvedných nameraných hodnôt v tabuľkách a znalosti fitness najviac optimálneho jedinca počiatočnej populácie (3196), ktorý sa aj vďaka použitému elitizmu propaguje, sa dá odhadnúť efektivita dvojice parametrov ako:

$$(min_pociatocna_fitness - eval(x))/pocet(x) \tag{6.1}$$

kde $min_pociatocna_fitness$ je fitness najviac optimálneho jedinca počiatočnej populácie, $eval(x)$ je fitness najviac optimálneho nájdeného jedinca pri danej dvojici hodnôt parametrov a $pocet(x)$ je počet vyhodnotení fitness pre túto dvojicu. Podľa uvedeného vzorca je populácia s dvomi jedincami už na pohľad výrazne lepšia, než väčšie populácie.

Počet jedincov	Pravdepodobnosť mutácie *delete*				
	0,01	**0,08**	**0,15**	**0,22**	**0,30**
2	2682	2398	2561	2119	2609
30	2354	1718	1712	1866	1894
60	2339	1820	1788	1756	1877

Tabulka 6.1: Dosiahnutá hodnota fitness po 20 generáciách

Počet jedincov	Pravdepodobnosť mutácie *delete*				
	0,01	**0,08**	**0,15**	**0,22**	**0,30**
2	47	64	68	85	80
30	740	1106	1081	1130	1257
60	1457	2253	2200	2317	2312

Tabulka 6.2: Počet vyhodnocovaní fitness v 20 generáciách

Nameraná efektivita podľa uvedeného vzorca všetkých skúmaných dvojíc pravdepodobností mutácie *delete* a veľkosti populácie je uvedená v tabuľke 6.3. Zelenou je zvýraznený najlepší výsledok. Podľa experimentu teda vyšlo, že pre mutáciu typu *delete* sú najvhodnejším nastavením nízka veľkosť populácie a pravdepodobnosť mutácie *delete* rovná 0,22.

Dosiahnuté výsledky však nemusia byť úplne presné a to z niekoľkých dôvodov:

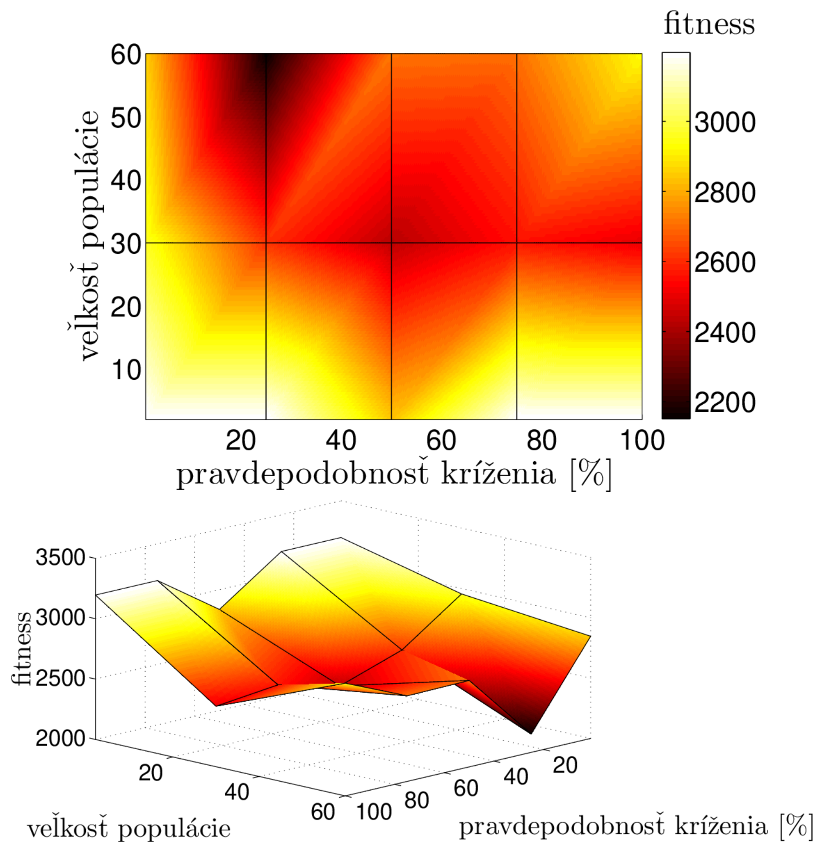
- Keďže vhodná hodnota ostatných parametrov genetického algoritmu ešte nie je zistená, bolo potrebné ju odhadnúť. Odhad nemusel byť urobený správne, čo mohlo mať jemný vplyv na dosiahnuté výsledky.

Počet jedincov	Pravdepodobnosť mutácie *delete*				
	0,01	**0,08**	**0,15**	**0,22**	**0,30**
2	10,94	12,47	9,34	12,67	7,34
30	1,14	1,34	1,47	1,18	1,04
60	0,59	0,62	0,64	0,62	0,57

Tabulka 6.3: Priemerné zlepšenie ohodnotenia fitness medzi dvomi vyhodnoteniami fitness

- Rovnako boli odhadom vybrané aj merané hodnoty parametrov `PRAVDEPODOBNOST_MUTACIE_DELETE` a `POCET_JEDINCOV`, navyše zvolené hodnoty majú pomerne nízku granularitu a optimálne nastavenie tak môže ležať vychýlene od meraných.
- Genetický algoritmus je nedeterministický a tak pri opätovnom spustení testov (s inou hodnotou *seed* pre generátor náhodných čísel), by sa výsledky pravdepodobne do určitej miery odlišovali.
- Pre neskoršie iterácie by sa výsledky taktiež mohli trochu odlišovať.

Zjemnením pre niektoré z uvedených dôvodov nepresnosti by bolo urobenie väčšieho počtu meraní. Je však potrebné zvážiť časovú náročnosť experimentu a nájsť kompromis s požadovanou presnosťou.



Obrázek 6.3: Dosiahnutá hodnota fitness po 20 generáciách u kríženia.

V ďalšom experimente je skúmaná druhá z dvoch genetických operácií skracujúca dĺžku chromozómov, kríženie, a jej vzťah k veľkosti populácie. Meranie prebiehalo rovnakým princípom, ako v prípade predošlej mutácie typu *delete*. Opäť sa vykonal pevný počet 20 iterácií genetického algoritmu. Skúmané parametre boli nastavené na nasledujúce hodnoty:
`PRAVDEPODOBNOST_KRIZENIA` $\in \{0,01; 0,25; 0,50; 0,75; 1\}$,
`POCET_JEDINCOV` $\in \{2, 30, 60\}$.
Hodnotu fitness najlepšieho jedinca pre všetky dvojice karteziánskeho súčinu oboch uvedených množín nájde čitateľ prehľadovo na obrázku 6.3. Opäť platí, že čím lepší nájdený jedinec, tým tmavšia plôška. V porovnaní s predošlou mutáciou typu *delete* sú však nájdení jedinci menej kvalitní (majú horšiu fitness). Navyše najlepšie riešenia sa javia byť chaoticky roztrúsené, čo učiňuje odhad najvhodnejšieho nastavenia parametrov ťažší.

Presné hodnoty nameranej fitness sú za účelom ďalšieho výpočtu uvedené v tabuľke 6.4. Červenou sú zvýraznené prípady, kedy sa nepodarilo vytvoriť jedinca lepšieho než najlepší jedinec v počiatočnej populácií. Tabuľka 6.5 udáva, koľkokrát bola pri príslušnom behu optimalizácie vyhodnotená fitness.

Počet jedincov	Pravdepodobnosť kríženia				
	0,01	**0,25**	**0,5**	**0,75**	**1**
2	3196	3196	2836	3196	3196
30	2955	2606	2429	2558	2501
60	2847	2152	2716	2709	2934

Tabulka 6.4: Dosiahnutá hodnota fitness po 20 generáciách

Počet jedincov	Pravdepodobnosť kríženia				
	0,01	**0,25**	**0,5**	**0,75**	**1**
2	42	46	48	53	56
30	631	720	806	850	827
60	1266	1434	1620	1582	1593

Tabulka 6.5: Počet vyhodnocovaní fitness v 20 generáciách

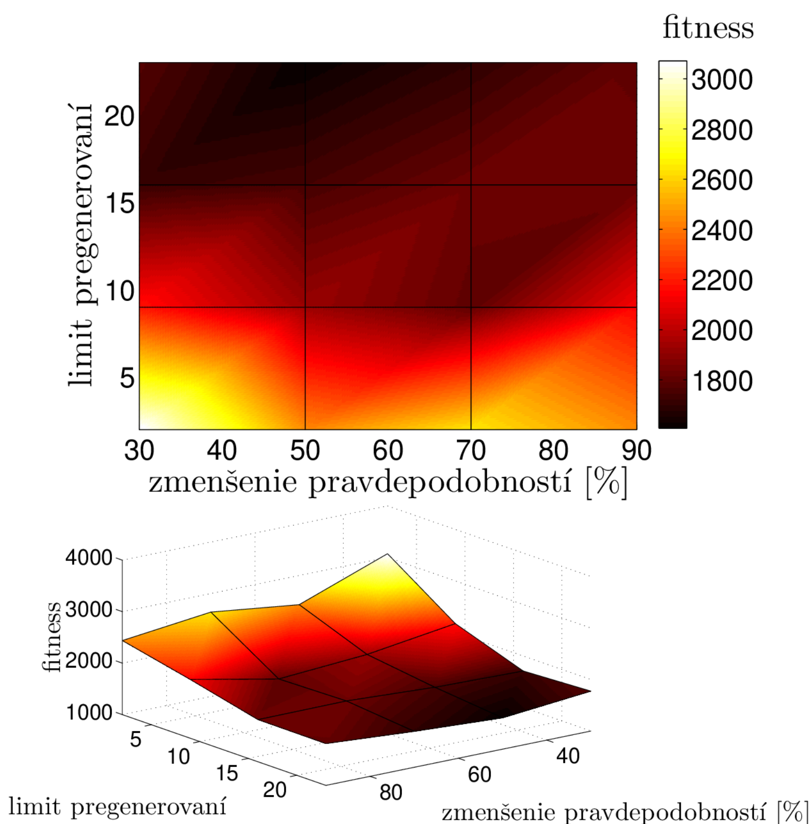
Počet jedincov	Pravdepodobnosť kríženia				
	0,01	**0,25**	**0,5**	**0,75**	**1**
2	0	0	7,5	0	0
30	0,38	0,82	0,95	0,75	0,84
60	0,28	0,73	0,3	0,31	0,16

Tabulka 6.6: Priemerné zlepšenie ohodnotenia fitness medzi dvomi vyhodnoteniami fitness

V tabuľke 6.6 sú potom vypočítané efektivity pre každú dvojicu hodnôt parametrov. Efektivity sú počítané podľa už uvádzaného vzorca 6.1. Z tabuľky vidno, že kríženie oproti mutácií typu *delete* dosahuje výrazne slabšie výsledky. V prípade zelenou zvýraznenou hodnotou sa podarilo dosiahnuť pomerne dobrý výsledok, čo je však pravdepodobne len náhoda. Ďalej treba ešte zdôrazniť že rovnako ako v prípade mutácie typu *delete* sú namerané výsledky len orientačné a to z rovnakých dvôvodov.

Podľa uvedených meraní a úvah, sú pre ďalšie merania považované za správne nastavenia parametrov nasledovné:

- veľkosť populácie 2,
- pravdepodobnosť mutácie typu *delete* = 0.22,
- pravdepodobnosti ostatných genetických operácií = nízka hodnota,
- elitizmus pre uchovávanie verzie jedinca s pokrytím vlastností 100 %,
- prah počtu jedincov v populácií, kedy dochádza ku zmenšeniu pravdepodobností genetických operácií = 1. Nastavenie vyplýva z veľkosti populácie a elitizmu, kedy vieme o jednom z dvoch jedincov vždy povedať že je vyhovujúci. Pri inom nastavení by tak dochádzalo k pregenerovaniu buď v každej iterácií, alebo v žiadnej.



Obrázek 6.4: Dosiahnutá hodnota fitness po 20 generáciách v populácií s 2 jedincami.

Posledný experiment pre zisťovanie správnosti nastavení parametrov sa venuje parametrom súvisiacim so samo-adaptáciou a opakovaným pregenerovaním nevyhovujúcich jedincov. Konkrétne sa jedná o maximálny povolený počet pokusov pregenerovaní nevyhovujúceho jedinca a pomer, akým majú byť znižované pravdepodobnosti vykonania genetických operácií. Skúmané parametre boli nastavené na nasledujúce hodnoty:
`ZMENSENIE_PRAVDEPODOBNOSTI` $\in \{0,3; 0,5; 0,7; 0,9\}$,
`MAX_POCET_POKUSOV_GENEROVANIA_POTOMKA` $\in \{2, 9, 16, 23\}$.

Namerané hodnoty fitness sú uvedené v grafoch 6.4 a v tabuľke 6.7. Na rozdiel od grafov vyobrazených v predošlých experimentoch má graf v tomto experimente už pomerne vyhladený tvar bez výrazných lokálnych extrémov. V tabuľke 6.8 je uvedený počet vyhodnocovaní fitness pre dané parametre a v tabuľke potom efektívnosť jednotlivých nastavení. Z výsledkov sa ukazuje, že vhodný maximálny strop počtu pregenerovaní je 2 a pravdepodobnosti vykonania genetických operácií by mali byť znižované o polovicu.

Maximálny počet pokusov pregenerovania jedinca	Zmenšenie pravdepodobnosti			
	0,3	**0,5**	**0,7**	**0,9**
2	3071	2432	2640	2438
9	2172	1924	1800	2140
16	1709	1744	1829	1818
23	1755	1608	1696	1811

Tabulka 6.7: Dosiahnutá hodnota fitness po 20 generáciách v populácií s 2 jedincami

Maximálny počet pokusov pregenerovania jedinca	Zmenšenie pravdepodobnosti			
	0,3	**0,5**	**0,7**	**0,9**
2	26	31	33	41
9	71	73	110	155
16	125	154	182	254
23	150	223	241	313

Tabulka 6.8: Počet vyhodnocovaní fitness v 20 generáciách v populácií s 2 jedincami

Maximálny počet pokusov pregenerovania jedinca	Zmenšenie pravdepodobnosti			
	0,3	**0,5**	**0,7**	**0,9**
2	48,35	61,16	51,15	46,1
9	30,37	32,93	22,98	14,12
16	20,95	16,78	13,73	9,88
23	17,15	12,2	10,92	8,04

Tabulka 6.9: Priemerné zlepšenie ohodnotenia fitness medzi dvomi vyhodnoteniami fitness pre populáciu s 2 jedincami a elitizmom

Po zdĺhavom zisťovaní vhodných parametrov implementovaného genetického algoritmu sa môže pristúpiť k samotnému optimalizovaniu sady regresných testov. Zvolenými nastaveniami sú tie, čo sa v sade experimentov sa ukázali ako najviac efektívne a to: veľkosť populácie 2, elitizmus, použitie genetickej operácie mutácie typu *delete* s pravdepodobnosťou 0,22, samo-adaptácia, znižovanie pravdepodobností na 0,5 a maximálny možný počet pregenerovaní jedinca rovný 2. Po prebehnutí optimalizácie obsahovala získaná sada regresných testov 440 transakcií, čo je oproti priemernému počtu transakcií v jedincoch vygenerovaných verifikačným prostredím (5653,52) redukcia na 7,78 %. Samotný proces optimalizácie trval 11 hodín a bol spustený na serveri `lissom2.fit.vutbr.cz`.

Pre porovnanie bola optimalizácia vyskúšaná aj s populáciou dvoch jedincov a parametrami, ktoré pri experimentoch popísaných tabuľkou 6.7 dosiahli najlepšiu fitness, nie však

najlepšiu efektivitu podľa 6.9. Optimalizačný proces dokázal zredukovať pôvodnú sadu testov na podobnú veľkosť blížiacu sa 440 transakciám, doba jeho behu však bola 33 hodín.

V tomto bode je ešte dôležité zdôrazniť, že čím bližšie sa jedinec blížil optimu, tým výrazne dlhšie trvalo z neho odstrániť transakciu. Pri uspokojení sa s menej optimálnym riešením by tak šlo proces optimalizácie výrazne skrátiť.

Po nameraní výsledkov optimalizácie je dobré sa na záver zamyslieť nad budúcimi možnými vylepšeniami optimalizačného procesu. V predošlých kapitolách bol ako možnosť do budúcnosti navrhnutý ostrovný algoritmus 2.3.1. Ostrovný algoritmus obecne dosahuje lepších výsledkov, čím sa rozumie nachádzanie o niečo lepších riešení v o trochu kratšom čase. Aktuálne sa však dá posúdiť, že ostrovný algoritmus ako vylepšenie nie je vhodný a to z dôvodu, že pracuje s niekoľkými populáciami zároveň a teda veľkým počtom jedincov. To prináša i po zvážení určitého percenta vylepšenia stále príliš veľký počet vyhodnotení fitness, čo je neakceptovateľné z nasledujúceho dôvodu. V meraniach bolo zistené, že optimalizácia na populácií s dvomi jedincami a obvode ALU trvá rádovo niekoľko hodín, pre väčšiu populáciu sú to až dni. Použitie zložitejšej komponenty uvedený čas ešte pravdepodobne výrazne predĺži. Z popísaného sa dá usúdiť, že v tomto prípade sa pravdepodobne viac oplatí akceptovať o niečo menej optimálne riešenie za cenu vysokej úspory času. Budúce vylepšenie riešenia sa tak javí byť iným smerom a to vylepšením genetických operácií a procesu samo-adaptácie. Koncept nízkeho počtu jedincov populácie, ako aj elitizmus by mali ostať zachované z dôvodu udržania dostatočne nízkej dĺžky behu optimalizácie.

Kapitola 7

Záver

Táto diplomová práca sa venovala zisťovaniu, či je funkčná verifikácia dobrým zdrojom pre vytvorenie sady regresných testov hardvérového obvodu. Testovacie vektory získané z procesu funkčnej verifikácie totiž splňujú pokrytie špecifikovaných vlastností obvodu na 100 %, čo je veľká výhoda. Zároveň je však takáto sada vektorov príliš veľká a redundantná, v zmysle niekoľkonásobného overovania už overených vlastností. Preto je potrebné získanú sadu optimalizovať, na čo sa v tejto práci využíva evolučný algoritmus. K zoptimalizovanej sade regresných testov vedie niekoľko krokov.

V prvej (teoretickej) časti práce bola naštudovaná a popísaná teória ohľadne funkčnej verifikácie, metodiky UVM, jazyka SystemVerilog, regresných testov a evolučných algoritmov s bližším zameraním sa na genetické algoritmy. Druhá kapitola podrobne analyzuje zadanie, ukazuje dôležitosť práce, demonštruje závažnosť redundancie vo vektoroch vygenerovaných v procese funkčnej verifikácie a popisuje prečo sú pre optimalizačnú úlohu vhodné práve evolučné algoritmy.

Ďalšie dve kapitoly sa venujú návrhu a implementácií optimalizačného prostredia tvoreného evolučným algoritmom, verifikačným prostredím UVM a automatizáciou testov. Ako evolučný algoritmus bol navrhnutý genetický algoritmus odlišujúci sa od bežných genetických algoritmov. Použitý genetický algoritmus pracuje s variabilnou dĺžkou chromozómov a z toho dôvodu aj špeciálne navrhnutými genetickými operáciami, ponúka samo-adaptáciu, možnosť okamžitej opravy pri vygenerovaní nevhodného jedinca a použitá fitness funkcia má opačný priebeh oproti väčšine bežne používaných fitness funkcií, čo viedlo k modifikácií použitých selekčných algoritmov. Čo sa týka verifikačného prostredia, k už poskytnutému verifikačnému prostrediu UVM boli boli doprogramované špecifické časti, ktoré umožnili využitie verifikačného prostredia ako generátoru počiatočnej populácie a ďalej ako prostriedok pre meranie fitness. Poslednou časťou navrhnutého a implementovaného optimalizačného prostredia je automatizácia testov, ktorá prináša zjednodušenie testovania a konfigurácie spúšťaných testov.

V kapitole namerané výsledky, ktoré sú demonštrované na konkrétnom obvode ALU, je popísaná počiatočná populácia získaná z funkčnej verifikácie. Ďalej sú tu zisťované správne nastavenia parametrov genetického algoritmu empiricky. Bolo zistené, že z nastavení sú výhodné nízka veľkosť populácie, elitizmus a použitie genetickej operácie mutácie typu *delete* s pravdepodobnosťou 0,22 a samo-adaptáciou. Uvedené nastavenia už prechádzajú do konceptu aj ďalších typov evolučných algoritmov a tak spájajú myšlienku genetického algoritmu s evolučnými stratégiami a evolučným programovaním. Po zistení správnych nastavení parametrov nasledoval samotný beh optimalizácie a bolo namerané, že implementovaný genetický algoritmus dokáže zoptimalizovať sadu regresných testov na 7,78 %, čím sa potvrdilo,

že funkčná verifikácia je vhodným zdrojom sady regresných testov v prípade ich následnej optimalizácie evolučným algoritmom.

Na základe nameraných výsledkov by bolo do budúcnosti možné nájsť ďalšie vylepšenia existujúceho riešenia, napríklad ešte zlepšiť genetické operácie alebo proces samo-adaptácie. Koncept nízkeho počtu jedincov populácie, ako aj elitizmus by mali zostať zachované z dôvodu udržania dostatočne krátkej doby behu optimalizácie, čo je kritická záležitosť.

Literatura

[1] Aliev, R. A.; Aliev, A. R.: *Soft Computing and Its Applications*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2001, ISBN 9810247001.

[2] Collett International Research: IC/ASIC Functional Verification Study. 2004.

[3] Erez, B.; Shmuel, U.: Compacting Regression-Suites On-The-Fly. *Asia-Pacific Software Engineering Conference*, 12 1997.
URL www.research.ibm.com/haifa/dept/svt/papers/simulation/isyse97.ps

[4] Fitzpatrick, T.: UVM Basics. 2013.
URL https://verificationacademy.com/sessions/introduction-uvm

[5] Fitzpatrick, T.: UVM Basics. 2013.
URL
https://verificationacademy.com/sessions/uvm-introducing-transactions

[6] Fitzpatrick, T.: UVM Basics. 2013.
URL https://verificationacademy.com/sessions/uvm-hello-world

[7] Fitzpatrick, T.: UVM Basics. 2013.
URL https://verificationacademy.com/sessions/uvm-reporting

[8] Hekmatpour, A.; Coulter, J.; Salehi, A.: FoCuS: A Dynamic Regression Suite Generation Platform for Processor Functional Verification. In *Computers and Their Applications*, ISCA, 2005, ISBN 1-880843-54-4, s. 373–378.
URL http://dblp.uni-trier.de/db/conf/cata/cata2005.html#HekmatpourCS05

[9] Martin, W.; Lienig, J.; J., C.: Island (migration) models: evolutionary algorithms based on punctuated equilibria. Technická zpráva, 1997.
URL
http://www.cs.virginia.edu/people/faculty/pdfs/Island_Migration.pdf

[10] Mentor Graphics Verification Methodology Team: UVM Cookbook. Verification academy, 2013.
URL https://verificationacademy.com/cookbook/uvm

[11] Meyer, A.: *Principles of Functional Verification*. USA: Elsevier Science, 2003, ISBN 0-7506-7617-5.

[12] Munakata, T.: *Fundamentals of the New Artificial Intelligence: Neural, Evolutionary, Fuzzy and More (Texts in Computer Science)*. London: Springer Publishing Company, Incorporated, druhé vydání, 2008, ISBN 184628838X.

[13] Myers, G. J.; Sandler, C.: *The Art of Software Testing*. New Jersey: John Wiley & Sons, second edition vydání, 2004, ISBN 0471469122.

[14] Parissis, T. C. D. . N. T. B. . I.: Automatic Generation of Test Cases in Regression Testing for Lustre/SCADE Programs. *Journal of Software Engineering and Applications*, 2013.
URL http://dx.doi.org/10.4236/jsea.2013.610A004

[15] Russell, S.; Norvig, P.: *Artificial Intelligence: A Modern Approach*. New Jersey: Pearson Education, třetí vydání, 2003, ISBN 978-0-13-604259-4.

[16] Salemi, R.: *FPGA Simulation: A Complete Step-by-step Guide*. GreatManager series, USA: Boston Light Press, 2009, ISBN 9780974164908.
URL http://books.google.cz/books?id=qSmEPgAACAAJ

[17] Sekanina, L.: *Evoluční hardware: od automatického generování patentovatelných invencí k sebemodifikujícím strojům*. Praha: Academia, 2009, ISBN 978-80-200-1729-1.

[18] Spear, C.: *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. New York: Springer, 2006, ISBN 9780387270364.

[19] Whitley, D.; Rana, S.; Heckendorn, R.: The Island Model Genetic Algorithm: On Separability, Population Size and Convergence. Colorado state university, 1998.
URL http://neo.lcc.uma.es/Articles/WRH98.pdf

[20] Zbořil, F.: Genetické algoritmy. 2012.

Příloha A

Zoznam parametrov genetického algoritmu

V kapitole venujúcej sa teórií ohľadne genetických algoritmov 2.3.1 bolo popísané, že genetické algoritmy majú niekoľko nastaviteľných parametrov. Zároveň úspešnosť genetického algoritmu nachádzať optimálne riešenia pre zadaný problém silno závisí od správneho nastavenia týchto parametrov. Navyše neexistuje žiadna efektívna metóda, ako sa dá určiť správne nastavenie parametrov a preto sa toto nastavenie obvykle zisťuje experimentálne. Správne nastavenie parametrov genetického algoritmu sa v tejto diplomovej práci taktiež zistí experimentálne.

V kapitole 5 boli postupne popísané jednotlivé implementované nastaviteľné parametre. V tejto prílohe sú zhrnuté všetky implementované nastaviteľné parametre a ich povolené hodnoty do prehľadnej a stručnej formy. Všetky nastaviteľné parametre implementovaného genetického algoritmu, tak ako ich užívateľ nájde a môže meniť v zdrojovom súbore `basicGA/nastavenia_a_konstanty.sv` sú nasledujúce:

`UKONCENIE_POCET_GENERACII`: udáva, či je ako ukončovacia podmienka genetického algoritmu použité dosiahnutie určitého počtu iterácií

Povolené hodnoty: $\{0, 1\}$ určujúce či je (1) alebo nie je (0) použitá táto podmienka

`MAX_POCET_GENERACII`: parameter je uvažovaný iba ak `UKONCENIE_POCET_GENERACII`=1. Parameter vyjadruje po koľkých iteráciách má byť genetický algoritmus ukončený.

Povolené hodnoty: $\mathbb{N}$

`UKONCENIE_POCET_NEZLEPSENI_TOP_FITNESS`: udáva, či je ako ukončovacia podmienka genetického algoritmu použitá dlhodobá stagnácia najlepšieho nájdeného jedinca.

Povolené hodnoty: $\{0, 1\}$ určujúce či je (1) alebo nie je (0) použitá táto podmienka

`MAX_POCET_NEZLEPSENI_TOP_FITNESS`: tento parameter je uvažovaný iba v prípade ak `UKONCENIE_POCET_NEZLEPSENI_TOP_FITNESS`=1. Vyjadruje, po koľkých iteráciách, počas ktorých nie je nachádzaný nový najlepší jedinec, má byť algoritmus ukončený.

Povolené hodnoty: $\mathbb{N}$

`POCET_JEDINCOV`: veľkosť populácie.

Povolené hodnoty: $\mathbb{N}$

METODA_SELEKCIE: parameter udáva, ktorý algoritmus selekcie sa má používať. Sú implementované tri selekčné algoritmy a to turnajová selekcia, ruleta podľa ohodnotenia fitness a ruleta podľa poradia.

Povolené hodnoty: {TURNAJ, RULETA_FITNESS, RULETA_PORADIE}

TURNAJ_POOL_SIZE: parameter je uvažovaný iba v prípade ak METODA_SELEKCIE=TURNAJ. Vyjadruje, z koľkých náhodne zvolených jedincov populácie sa vyberá jeden rodič.

Povolené hodnoty: $\mathbb{N}$

POCET_MINULYCH_ZACHOVANYCH_JEDINCOV: udáva, nakoľko sa prekrývajú po sebe nasledujúce generácie.

Povolené hodnoty: $\{x|x \in \mathbb{N}_0 \wedge x \leq POCET_JEDINCOV\}$

PRENOS_MINULYCH_JEDINCOV: súvisí s POCET_MINULYCH_ZACHOVANYCH_JEDINCOV a udáva, či sa z minulej generácie zachovávajú najlepší, alebo náhodní jedinci.

Povolené hodnoty: {NAJLEPSI, NAHODNI}.

Pre POCET_MINULYCH_ZACHOVANYCH_JEDINCOV = 1
a PRENOS_MINULYCH_JEDINCOV = NAJLEPSI sa jedná o elitizmus.

PRAVDEPODOBNOST_MUTACIE: definuje pravdepodobnosť vzájomnej výmeny poradia ľubovoľných dvoch transakcií v rámci jedného chromozómu. Hodnota sa udáva v percentách.

Povolené hodnoty: $\langle 0, 100 \rangle$

PRAVDEPODOBNOST_MUTACIE_DELETE: definuje pravdepodobnosť vymazania pre každú jednu transakciu v chromozóme. Hodnota sa udáva v percentách.

Pre napríklad PRAVDEPODOBNOST_MUTACIE_DELETE = 4% sa po aplikovaní jedného kroku tohto druhu mutácie chromozóm zredukuje v počte transakcií približne o 4%.

Povolené hodnoty: $\langle 0, 100 \rangle$

PRAVDEPODOBNOST_KRIZENIA: udáva pravdepodobnosť, že sa bude dvojica jedincov krížiť. Zadáva sa v percentách.

Povolené hodnoty: $\langle 0, 100 \rangle$

PRAH_ZMENSENIA_PRAVDEPODOBNOSTI: udáva, koľko minimálne jedincov v aktuálnej populácií musí mať najhoršie možné ohodnotenie fitness, aby došlo k zníženiu pravdepodobností mutácie a kríženia.

Povolené hodnoty: $\{x|x \in \mathbb{N}_0 \wedge x \leq POCET_JEDINCOV\}$,

pr.: PRAH_ZMENSENIA_PRAVDEPODOBNOSTI = 0.4 * POCET_JEDINCOV

ZMENSENIE_PRAVDEPODOBNOSTI: parameter udáva, koľkonásobne sa znížia pravdepodobnosti mutácie a kríženia v prípade, že počet jedincov s najhoršou možnou hodnotou fitness prekročí PRAH_ZMENSENIA_PRAVDEPODOBNOSTI

Povolené hodnoty: $\langle 0, 1 \rangle$

MIN_PRAVDEPODOBNOST_MUTACIE: definuje najnižšiu možnú hodnotu, na akú môže byť PRAVDEPODOBNOST_MUTACIE znížená.

Povolené hodnoty: $\langle 0, 100 \rangle$

`MIN_PRAVDEPODOBNOST_MUTACIE_DELETE`: definuje najnižšiu možnú hodnotu, na akú môže byť `PRAVDEPODOBNOST_MUTACIE_DELETE` znížená.

Povolené hodnoty: $\langle 0, 100 \rangle$

`MIN_PRAVDEPODOBNOST_KRIZENIA`: definuje najnižšiu možnú hodnotu, na akú môže byť `PRAVDEPODOBNOST_KRIZENIA` znížená.

Povolené hodnoty: $\langle 0, 100 \rangle$

`MAX_POCET_POKUSOV_GENEROVANIA_POTOMKA`: udáva maximálny počet pokusov, koľkokrát sa program pokúsi pregenerovať novo vzniknutého potomka v prípade, že má najhoršiu možnú fitness.

Povolené hodnoty: $\mathbb{N}$

`VERIFICATION_PATH`: cesta k verifikačnému prostrediu.

`FILE_GA_TO_VER`: cesta a názov súboru, ktorý slúži pre komunikáciu genetického algoritmu s verifikačným prostredím. Konkrétne sa jedná o súbor, kam genetický algoritmus zapíše jedinca, ktorého pokrytie má byť zmerané verifikačným prostredím.

`FILE_VER_TO_GA`: cesta a názov súboru, ktorý slúži pre komunikáciu genetického algoritmu s verifikačným prostredím. Konkrétne je to súbor, do ktorého verifikačné prostredie zapisuje svoje výstupy.

`FILE_TOP_FITNESS`: cesta a názov súboru, kde sa po ukončení behu genetického algoritmu uloží hodnota najlepšej dosiahnutej fitness hodnoty.

`FILE_TOP_FITNESS_JEDINEC`: cesta a názov súboru, kde sa po ukončení genetického algoritmu uloží najlepší nájdený jedinec.

`DIR_UVODNA_POPULACIA`: cesta a adresár, kde sú uložené počiatočný jedinci.