
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ**
**DEPARTMENT OF COMPUTER SYSTEMS**


# AUTOMATIZACE TVORBY SCÉNÁŘŮ PŘENOSITELNÝCH STIMULŮ POMOCÍ EVOLUČNÍCH ALGORITMŮ

**AUTOMATED CREATION OF PORTABLE STIMULI SCENARIOS USING EVOLUTIONARY ALGORITHMS**

**DIPLOMOVÁ PRÁCE**
**MASTER'S THESIS**


**AUTOR PRÁCE** **Bc. ANDREJ TICHÝ**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. MARCELA ZACHARIÁŠOVÁ, Ph.D.**
**SUPERVISOR**



**BRNO 2020**

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

# Zadání diplomové práce


Student: **Tichý Andrej, Bc.**
Program: Informační technologie Obor: Počítačové a vestavěné systémy
Název: **Automatizace tvorby scénářů přenositelných stimulů pomocí evolučních algoritmů**
**Automated Creation of Portable Stimuli Scenarios Using Evolutionary Algorithms**
Kategorie: Počítačová architektura
Zadání:

1. Seznamte se s problematikou evolučních algoritmů, zejména s evolučními strategiemi a genetickým programováním.
2. Seznamte se s problematikou funkční verifikace číslicových systémů a s novým standardem pro přenositelné stimuly (PSS - Portable Test and Stimulus Standard).
3. Vyberte/navrhněte vhodnou evoluční strategii anebo algoritmus geneticého programování pro řízení generátoru verifikačních scénářů pomocí přenositelných stimulů.
4. Implementujte strategii anebo algortimus ve vhodném jazyce (SystemVerilog, Python, C++, ..).
5. Experimentálně ověřte na vybraném modulu (ALU, FPU, Timer, atd.).
6. Zhodnoťte dosažené výsledky.

Literatura:

- Dle pokynů vedoucí.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Zachariášová Marcela, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 20. května 2020
Datum schválení: 25. října 2019

# Abstrakt

Táto práca sa zaoberá automatizáciou tvorby scenárov pre štandard Portable Stimulus. Hlavným cieľom práce je automatické generovanie testov, ktoré sú definované formou grafu pre nástroj Questa InFact od spoločnosti Mentor. K automatizácii som použil evolučný algoritmus s využitím gramatickej evolúcie. Pri implementácii som využil framework PonyGe2, ktorý zastrešuje implementáciu niektorých variačných operátorov gramatickej evolúcie. Vygenerované scenáre sa pripoja k existujúcemu verifikačnému prostrediu, založenom na metodike UVM, a následne je spustená verifikácia pripojeného komponentu. Na základe dosiahnutého funkčného a štrukturálneho pokrytia je vypočítaná fitness hodnota jedinca, ktorá je propagovaná do evolučného algoritmu. V závere práce sú vykonané experimenty nad komponentom časovač a vyhodnotený prínos navrhnutého evolučného algoritmu. Navrhnutý evolučný algoritmus je konfigurovateľný pomocou gramatiky a užívateľom definovaných základných transakcií, čo umožňuje široké spektrum použitia. Evolučný algoritmus dokázal na verifikovanom komponente časovač dosiahnuť vysoké funkčné a štrukturálne pokrytie.



# Abstract

This thesis focuses on the automation of scenarios creation for Portable Stimulus standard. The main goal of the work is an automatic generation of tests, which are defined as graphs for the Questa inFact tool from the Mentor company. For the automation I used an evolutionary algorithm with using a grammatical evolution. The generated scenarios are connected to the existing verification environment based on UVM methodology, then the verification of the connected component is started. Based on the achieved functional and structural coverage, the individual's fitness value is calculated and propagated into an evolutionary algorithm. At the end of the work, experiments are performed on the timer component and the contribution of the proposed evolutionary algorithm is evaluated. The proposed evolutionary algorithm is configurable by grammar and user-defined basic transactions, which allows a wide range of uses. The evolutionary algorithm managed to achieve high functional and structural coverage on the verified timer component.



# Kľúčové slová

Prenositeľné stimuly, evolučný algoritmus, genetické programovanie, gramatická evolúcia, Questa InFact, funkčná verifikácia, UVM

# Keywords

Portable Test and Stimulus Standard, evolutionary algorithm, genetic programming, grammatical evolution, Questa InFact, functional verification, UVM


# Citácia

TICHÝ, Andrej. *Automatizace tvorby scénářů přenositelných stimulů pomocí evolučních algoritmů*. Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Marcela Zachariášová, Ph.D.

# Automatizace tvorby scénářů přenositelných stimulů pomocí evolučních algoritmů

## Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Marcely Zachariášovej Ph.D. Uviedol som všetky literárne zdroje, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Andrej Tichý
31. júla 2020


## Poďakovanie

Ďakujem mojej vedúcej práce Ing. Marcele Zachariášovej Ph.D. za odborné konzultácie a rady pri tvorbe tejto diplomovej práce.

# Obsah

# Kapitola 1

# Úvod

S narastajúcou zložitosťou hardwarových systémov rastie i potreba dôkladnejšej verifikácie, čo prináša problémy v zložitosti a efektívnosti aktuálnych nástrojov používaných k verifikácií hardwarových obvodov. I keď v posledných rokoch bolo vynaloženého mnoho úsilia na zvýšenie produktivity verifikácie, toto úsilie bolo cielené hlavne na oblasť zaoberajúcej sa IP (angl. *intellectual property*) blokmi. Súčasné metodiky, ako napr. UVM (angl. *Universal Verification Methdodology*), prinášajú výhody objektovej orientácie na pole verifikácie, sú úspešne použiteľné na úrovni IP blokov. Na systémovej úrovni pri zapojení mnoho rôznych komponentov tento prístup zlyháva z dôvodu príliš veľkej komplexnosti systémov a zložitej tvorbe testov. [25]

Štandard PSS (angl. *Portable Test and Stimulus*) od firmy Accelera sa snaží posunúť definovanie testu z transakčnej úrovne na úroveň definovania zámeru testu. Cieľom je znovupoužiteľnosť testov, nazývaná tiež vertikálna a horizontálna znovupoužiteľnosť. Vertikálna znovupoužiteľnosť znamená definovanie zámeru testu a jeho použitie naprieč celým spektrom vývoja hardwarového systému, od IP blokov až po komplexné SoC (angl. *system on chip*) systémy. Horizontálna znovupoužiteľnosť cieli na definíciu testu a použitie na rôznych testovacích či cieľových platformách ako je simulácia, emulácia, FPGA (angl. *field-programmable gate array*), prototypovanie a podobne [3]. PSS definuje zámer testu vo forme grafu a pomocou nastavenia stratégie pokrytia dokáže cielene generovať stimuly tak, aby čo najefektívnejšie dosiahol cielené pokrytie. Táto efektivita je dosahovaná pomocou odstránenia redundancie cez definovanú stratégiu pokrytia. [22] Je potrebné poznamenať, že cieľom PSS štandardu nie je nahradiť súčasné technológie, ale kooperovať s nimi.

Cieľom práce je snaha o automatické generovanie či optimalizovanie tvorby scenárov pre portovateľné stimuly, ktoré môžu byť použité ako nástroj na generovanie takýchto stimulov s využitím evolučných algoritmov. Pri implementácií sa používa nástroj Questa InFact, ktorý je proprietárnou implementáciou prenositeľných stimulov a framework PonyGe2, ktorý zaobstaráva podporné funkcie evolučnému algoritmu.

V úvodnej kapitole 2.1 vysvetlím základné pojmy dôležité na pochopenie funkčnej verifikácie a jej tvorby. V nasledujúcej kapitole 2.3 predstavím štandard PSS, jeho hlavnú myšlienku a prínos. Po uzatvorení celku spojeného s verifikáciou sa práca ďalej zaoberá základným vysvetlením evolučných algoritmov 3, kde sú postupne vysvetlené všetky základné aspekty potrebné pre návrh evolučného algoritmu. Návrh riešenia definovaného problému pomocou gramatickej evolúcie predstavím v kapitole 4 spolu s konkrétnymi ukážkami práce s jedincami. Kapitola 5 obsahuje podrobný popis implementácie a nastavenia verifikačného prostredia, použité technológie a konkrétne metodiky použité v navrhnutom riešení. Experimenty a vyhodnotenie účinnosti algoritmu popisujem v kapitole 6.

# Kapitola 2

# Funkčná verifikácia a portovateľné stimuly

Pri procese tvorby hardwarového komponentu či systému je dôležité skontrolovať či výsledný produkt spĺňa požiadavky definované v špecifikácii. Pomocou funkčnej verifikácie overujeme či daný komponent korektne reaguje na vygenerované stimuly. Základom funkčnej verifikácie je model verifikovaného systému, ktorý odborne nazývame DUT (angl. *design under test*), a okolie systému zodpovedné za generovanie, čítanie a vyhodnocovanie stimulov, ktoré nazývame ako verifikačné prostredie. Model verifikačného prostredia je často implementovaný pomocou niektorého z jazykov vhodných pre popis hardwarových obvodov HDL (angl. *Hardware Description Language*), ako napríklad VHDL či Verilog [1].

Pri komplexných systémoch by bola realizácia verifikačného prostredia ťažká a, pravdepodobne, veľmi neefektívna. Preto bolo vyvinutých niekoľko verifikačných metodík (*OVM, UVM, atd.*), ktoré majú za cieľ proces tvorby verifikačného prostredia čo najviac zoptimalizovať.

Pri tvorbe kapitoly venovanej funkčnej verifikácií som čerpal prevažne z týchto publikácií [40, 41], webovej stránky *Chipverify*[1] a z prednášok predmetu *Funkční verifikace číslicových systémů* na Fakulte Informačných Technológií.

## 2.1 Verifikácia založená na simulácií

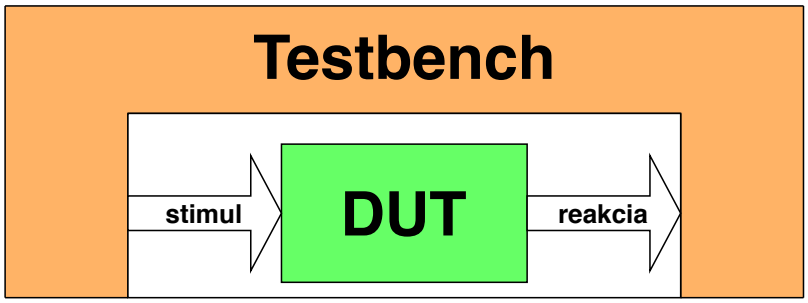
Proces tvorby hardwarového (HW) systému od jeho popisu v jednom z HDL (angl. *hardware description language*) až po výsledný produkt v podobe prototypu pre FPGA (angl. *field programmable gate array*) či v podobe masky pre ASIC (angl. *application specific integrated circuit*) je zdĺhavý a finančne náročný, preto je dôležité mať možnosť verifikovať model čo najskôr, aby sa chyby nezanášali do ďalších úrovní procesu výroby.

Základom verifikácie založenej na simulácii je architektúra zobrazená na obrázku 2.1, ktorá obsahuje simulačný model DUT (angl. *design under test*) a verifikačné prostredie, ktoré simuluje okolie pre daný model. V tejto fáze verifikácie pravdepodobne neexistuje hardwarový prototyp, ale iba simulační model, ktorý reprezentuje komponentu na rôznych úrovniach popisu[2]. Verifikácia pomocou simulácie na RTL (angl. *register-transfer level*) úrovni je v praxi najčastejšie používaná metóda, pretože prináša radu podstatných výhod. Abstrakciou jednotlivých komponentov máme možnosť verifikovať i veľmi zložité systémy.

[1] https://www.chipverify.com/uvm/uvm-tutorial
[2] HDL, RTL, Gate-level.

V rámci simulácie môžeme monitorovať i interné signály modelu, ktoré by v hardwarovej implementácií boli iba ťažko dostupné. Umožňuje odhaliť chyby, ale zároveň ich i znovu reprodukovať.

Jednou z nevýhod tejto verifikačnej metódy je, že i keď sa nám v simulácii podarí danú komponentu dostatočne zverifikovať, nemáme garanciu, že systém už neobsahuje ďalšie chyby. Nezanedbateľný je i fakt, že simulácia RTL modelu je niekoľkonásobne zložitejšia a pomalšia, než beh v reálnom hardwari, preto je dôležité robiť verifikáciu efektívne.




Obr. 2.1: Základná štruktúra verifikačného prostredia, verifikácie založenej na simulácii [40].

Pri simulácii RLT modelu platí, že čím je model zložitejší, tým je verifikácia náročnejšia na výpočtové zdroje. Jedna z možností, ako zmenšiť výpočtovú náročnosť verifikácie, je zjednodušiť viditeľnosť modelu zanedbaním nepodstatných signálov z pohľadu verifikácie.

**Metóda čiernej skrinky** (angl. *black box*) predstavuje techniku, kde úplne zanedbávame detaily vnútornej architektúry DUT a všetkých jeho vnútorných signálov. Z pohľadu verifikačného prostredia sú neviditeľné, zameriavame sa iba na definovanú funkcionalitu a externé rozhrania. Výhoda spočíva v zjednodušení simulácie. Pokiaľ pri zmene implementácie DUT nedošlo k zmene funkcionality či externých rozhraní, táto zmena nevynucuje úpravu vo verifikačnom prostredí. Nevýhoda tejto metódy spočíva v tom, že nemáme kontrolu nad vnútorným správaním modelu, a tak i pri detekcií chyby je často problematické určiť zdroj chyby.

**Metóda bielej skrinky** (angl. *white box*) predstavuje techniku, kde verifikačné prostredie má úplný prístup k vnútornej architektúre a signálom modelu. Hlavná výhoda tohoto prístupu spočíva v jednoduchšej lokalizácii zdroja chýb. Na druhej strane, tento prístup značne zvyšuje zložitosť simulácie a každá zmena v architektúre modelu vynucuje i zmenu vo verifikačnom prostredí.

**Metóda šedej skrinky** (angl. *grey box*) je spojením metód bielej a čiernej skrinky, kedy volíme kompromis medzi viditeľnosťou a zložitosťou simulácie modelu. Snažíme sa o to, aby verifikačnému prostrediu boli prístupné iba kritické a dôležité signály. Výhodami a nevýhodami tejto metódy je kombinácia predošlých metód.

### 2.1.1 Generovanie stimulov

Základnou funkcionalitou verifikačného prostredia, nazývaného tiež angl. *testbench*, je generovanie stimulov pre DUT a monitorovanie reakcií na tieto stimuly. Stimuly sú dáta, ktoré predávame vstupnému rozhraniu DUT, môžu byť v podobe jednoduchých hodnôt signálov až po komplexné transakcie.

V rámci generovania stimulov máme na výber hneď niekoľko možností. Medzi najčastejšie využívané generovanie stimulov patrí pseudonáhodné generovanie, kde generátor podľa dopredu zadaných vzorov generuje náhodné stimuly. Jedna z možností je generovať iba ná-

hodné hodnoty, kedy akceptujeme i invalidné hodnoty a kontrolujeme reakcie i na takéto stimuly.

Priame testy sú stimuly, ktoré cielia na konkrétne scenáre testujúce funkcionalitu definovanú v špecifikácii. Sú často písané ručne a preto je ich údržba značne náročná. K dosiahnutiu cieleného funkčného pokrytia sa v praxi využíva kombinácia priamych testov a pseudonáhodne generovaných stimulov.

### 2.1.2 Funkčná verifikácia riadená pokrytím

Počas funkčnej verifikácie sa využíva kombinácia priamych testov a pseudonáhodne generovaných stimulov. Jednou z úloh verifikačného inžiniera je určiť bod, kedy je možné verifikáciu úspešne ukončiť. Hlavná metrika ukončenia verifikácie je pokrytie (angl. coverage), ktorá predstavuje do akej miery doposiaľ vygenerované stimuly overujú funkčnosť a správanie sa systému. Pokrytie nám poskytuje spätnú väzbu o tom, v akom stave je verifikácia oproti verifikačnému plánu. Táto spätná väzba môže byť ďalej použitá napríklad umelou inteligenciou, ktorá bude cielene generovať stimuly na nepokryté časti [42]. Nevýhoda merania pokrytia je, že pridáva prácu verifikačnému inžinierovi, spomaľuje proces verifikácie a generuje množstvo dát v rámci simulácie.

Štrukturálne pokrytie (angl. *structural coverage*) predstavuje pokrytie implementácie modelu. Toto pokrytie je často automaticky generované a merané v samotnom simulačnom nástroji. Pokiaľ po dokončení verifikácie toto pokrytie nedosahuje 100% hodnoty, znamená to, že je potrebné rozšíriť generovanie stimulov, alebo sa jedná o mŕtvy kód, ktorý sa v modeli nikdy nevykoná. Najčastejšie používané metriky štrukturálneho pokrytia:

- **pokrytie kódu** (angl. *code coverage*) - meria dosiahnutú syntaktickú štruktúru modelu, kde obsahuje i detailnejšie informácie, ako napríklad pokrytie výrazov, podmienok, vetvenia a výrazov. Pomocou tohto pokrytia sa dá odhaliť i mŕtvy kód.
- **FSM pokrytie** (angl. *FSM coverage*) - predstavuje metriku doposiaľ navštívených stavov a možných prechodov medzi stavmi v konečných automatoch.
- **pokrytie prepínania** (angl. *toggle coverage*) - meria aktivitu signálov a registrov, ako často a na akých pozíciách menia svoje bitové hodnoty.

### 2.1.3 Funkčné pokrytie v jazyku SystemVerilog

Funkčné pokrytie je pre verifikačného inžiniera informácia o tom, aká funkcionalita bola pomocou testov doposiaľ testovaná. Funkčné pokrytie definuje verifikačný inžinier a je kriticky dôležité, aby pokrytie zahŕňalo všetku funkcionalitu definovanú v špecifikácií.

SystemVerilog je programovací jazyk určený pre popis hardwaru HDL a tvorbu verifikačného prostredia HVL (angl. *hardware verification language*). Je založený na jazyku Verilog s pridaním rôznych rozšírení. Jednou z najväčších výhod SystemVerilogu je možnosť využívať prvky objektovo-orientovaného programovania (angl. *object-oriented programing*, OOP). SystemVerilog obsahuje možnosť voliť funkcie implementované v iných programovacích jazykoch cez programové rozhranie (angl. *direct programing interface*, DPI).

Definícia funkčného pokrytia je úlohou verifikačného inžiniera a do istej miery odzrkadľuje jeho pochopenie špecifikácie. Základ funkčného pokrytia je definícia skupiny pokrytia (angl. *covergroup*), čo je užívateľom definovaný dátový typ, ktorý slúži ako obálka špecifikácie pokrytia pre DUT. Na riadku 4 v príklade kódu 2.1 je definícia skupiny pokrytia `cg`,

ktorá je vzorkovaná počas každej nábežnej hrany hodinového signálu CLK. V rámci tejto skupiny pokrytia môžeme definovať niekoľko bodov pokrytia (angl. *coverpoint*).

Bod pokrytia predstavuje výraz definujúci podmnožinu validných hodnôt signálu, pri ktorých je daný bod pokrytia aktivovaný. To znamená, že pokiaľ sledovaný signál nadobudne hodnoty, ktorá spadá pod definovanú podmnožinu, aktivuje sa bod pokrytia a zapíše sa do štatistík.

Jednotlivé body pokrytia sa môžu skladať z niekoľkých binov. Bin umožňuje vytvorenie zložitejších popisov podmnožiny hodnôt aktivujúcich daný bod pokrytia. Biny je možné generovať i automaticky.

Krížový bod pokrytia (angl. *cross coverpoint*) umožňuje zachytávať interakciu niekoľkých binov, alebo bodov pokrytia.

```
1   rand bit [2:0]    s_mode;
2   rand bit [1:0]    s_dir;
3
4   covergroup cg @ (posedge clk);              // vzorkovanie pri zostupnej hrane clk
5       data        : coverpoint s_data;        // s_data != 0
6       dir_slave   : coverpoint s_dir[0];      // lsb s_dir == 1
7
8       mode        : coverpoint s_mode {
9           bins mode_A = {0};;                 // s_mode == 0
10          bins mode_B = {[1:2]};              // s_mode == [1,2]
11      }
12
13      direction   : coverpoint s_dir {
14          bins dir_A = {0};                   // s_dir == 0
15          bins dir_B = {1:2}                  // s_dir == [1,2]
16      }
17
18      send_A      : cross {                   // s_dir == 0 && s_mode == 0
19          bins out = binsof(mode.mode_A) && binsof(direction.dir_A);
20      }
21  endgroup
```

Výpis 2.1: Príklad definície funkčného pokrytia v jazyku SystemVerilog.

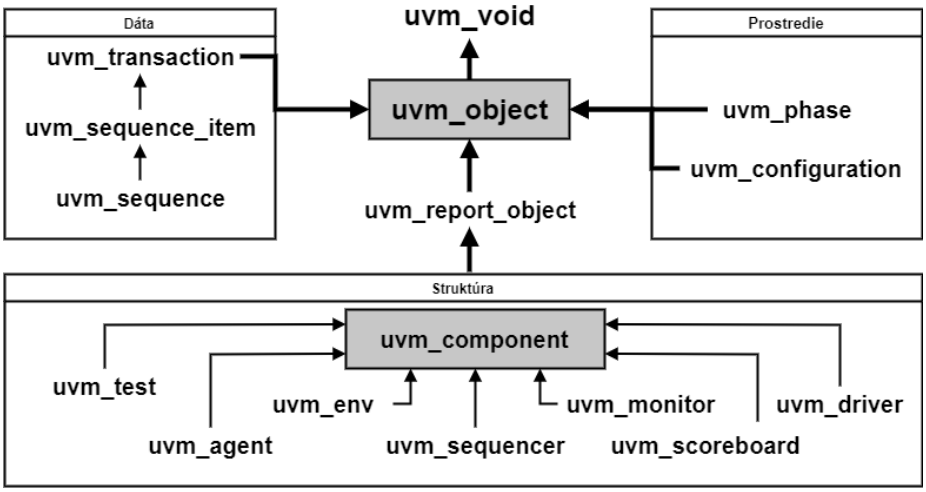
### 2.1.4 Verifikácia založená na formálnych tvrdeniach

Formálne tvrdenia (angl. *assertions*) umožňujú verifikačnému inžinierovi formálne definovať správanie verifikovaného komponentu na tej najnižšej úrovni. Existuje niekol'ko spôsobov použitia takého formálneho tvrdenia. Jedným z nich je formálny popis situácií, ktoré musia v systéme vždy platiť. Respektíve môžme definovať vlastnosti, ktoré v systéme nesmú nastať.

Jedno z d'alších použití formálnych tvrdení je definícia tvrdenia ako bodu pokrytia. Pomocou takého bodu pokrytia dokážeme definovať i viac cyklové vlastnosti, ktoré chceme v rámci verifikácie zachytiť. Zmysel tohto tvrdenia je v definícii situácie, ktorú chceme, aby pri verifikácií komponenty nastala. Verifikácia založená na formálnych tvrdeniach (angl. *assertion-based verification*, ABV) v niektorých prípadoch umožňuje efektívnejšie definovať funkčné pokrytie. [35]

## 2.2 UVM metodika

UVM (angl. *Universal Verification Methdodology*) vznikla zo staršieho štandardu OVM (angl. *Open Verification Methodology*). V praxi má tento štandard širokú podporu dostupných nástrojov a jedná sa o najrozšírenejšiu metódu v oblasti verifikácie [25]. Je to otvorený štandard od firmy Accellera[3], ktorý má za cieľ zefektívniť tvorbu verifikačného prostredia a jeho znovupoužiteľnosť. Tento štandard je skupina tried a knižníc (angl. *Base Class Library, BCL*), napísaných v jazyku SystemVerilog[4], ktoré definujú syntax a sémantiku tvorby verifikačného prostredia. BCL predstavujú generické nástroje, ktoré umožňujú efektívne nastavenie verifikačného prostredia ako je nastavenie databázy, hierarchie komponentov a testbenchu, či podpora automatizácie [17].




Obr. 2.2: Hierarchická štruktúra objekov definujúcich UVM verifikačné prostredie [16].

Popis tried v nasledujúcich podkapitolách vychádzajú z obrázku 2.2 a obrázku 2.3.

### Test

Komponent test je definovaný rozšírením triedy *uvm_test*, kde sa nastavuje a inštancuje verifikačné prostredie. Tento komponent obsahuje konfigurovateľné verifikačné prostredie, ktoré je možné inštancovať v rôznych top-level moduloch. Každý test si môže prispôsobiť verifikačné prostredie, aktivovať alebo deaktivovať agentov, zmeniť generovanie sekvencii a tým meniť konfiguráciu verifikačného prostredia [19].

### Prostredie

Prostredie (angl. *Environment*, Env) je kontajner, ktorý je rozšírením triedy *uvm_env*, a pomocou ďalších UVM komponentov definuje konfigurovateľnú architektúru verifikačného prostredia. Táto hierarchia zjednodušuje a podporuje znovupoužiteľnosť verifikačného prostredia, pretože verifikačný inžinier nemusí neustále vytvárať architektúru prostredia, stačí iba dané prostredia nakonfigurovať. Najčastejšie v sebe obsahuje komponenty ako *scoreboard*, *agent* a niekedy i ďalšie *environment*. Typicky tento komponent nastavuje aké stimuly sa majú generovať a podľa akého pokrytia sa majú verifikovať [15].

[3] https://www.accellera.org/downloads/standards/uvm
[4] https://standards.ieee.org/standard/1800-2017.html

### Scoreboard

Komponent scoreboard obsahuje kontroléry, ktoré verifikujú, či reakcie modelu DUT na stimuly sú validné. Táto verifikácia spočíva v porovnávaní reakcií na stimul s referenčným modelom (angl. *Golden Model*, GM) nazývaným tiež prediktor. Scoreboard obdrží transakciu od monitoru cez analytický port, ktorú následne distribuuje na rozhranie GM, výsledkom je predikovaný výstup GM.

### Agent

Agent je hierarchický komponent, ktorý zaobaľuje UVM komponenty interagujúce s rozhraním DUT. Definuje sa rozšírením triedy *uvm_agent*. Agent je typicky tvorený komponentom *sequencer*, ktorý generuje a riadi tok stimulov. Ďalej komponentom *uvm_driver*, slúžiacim na aplikáciu sekvencií na rozhranie DUT a *uvm_monitor* pre monitorovanie rozhrania DUT. Agent môže bežať v rôznych režimoch. V aktívnom režime agent generuje stimuly a aplikuje ich na rozhranie, v pasívnom režime môže iba monitorovať rozhranie DUT.

### Driver

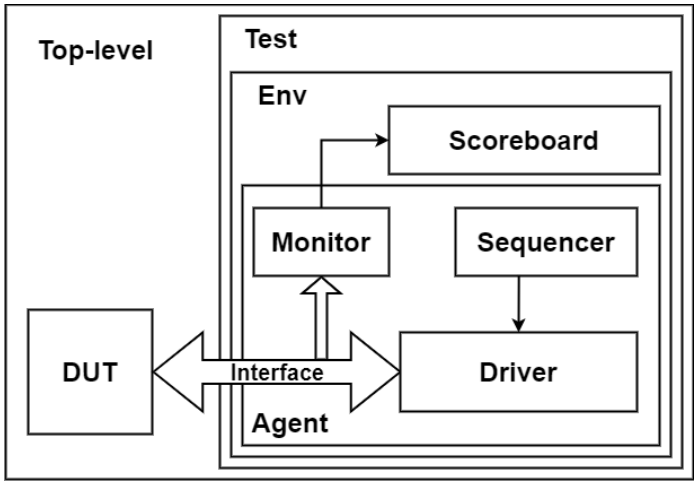
Driver je aktívny komponent, na svojom vstupe očakáva sekvenciu, ktorú následne prevedie na rozhranie DUT. Sekvencia je informácia, ktorú je potrebné previesť do protokolu podporovaného rozhraním DUT nastavením príslušných signálov, čo má za úlohu spomínaný driver.

### Sekvencer a sekvencie

*Sekvencer* je komponent, ktorý sa používa na generovanie stimulov. Nie je súčasťou verifikačného prostredia, ale konkrétneho testu. Sekvencie môžu byť prechodné alebo trvalé, môžu byť súčasťou jednej transakcie alebo celej simulácie. *Sekvencer* v architektúre figuruje ako prostredník medzi sekvenciami a *driverom*, generuje dáta zo vstupnej sekvencie a posiela ich na vstup *driveru*. [18]

### Monitor

Komponent monitor má za úlohu zbierať informácie o vstupoch a výstupoch pripojeného rozhrania. Tieto informácie môže ďalej sprostredkovávať v podobe transakcií iným komponentom v hierarchii. Monitor sa definuje rozšírením triedy *uvm_monitor*. Na pripojenie ku rozhraniu využíva virtuálne rozhranie, definované pre účely monitoru a TLM (angl. *Transacion Level Monitor*), analytický port, ktorým pomocou *broadcast* prenosu transportuje informácie. Komponenty, ktoré chcú dostávať informácie od monitoru, sa musia prihlásiť k odberu. [21]




Obr. 2.3: Príklad zapojenia UVM komponent do testbenchu

## 2.3 Prenositeľné stimuly

Za posledných niekoľko rokov sa výrazne zvyšuje zložitosť hardwarových projektov, čo negatívne ovplyvňuje zložitosť verifikačného prostredia pre dané projekty. I keď sa investuje mnoho úsilia do zvýšenia efektivity a produktivity verifikácie, ukazuje sa, že súčasný model UVM v zložitých projektoch už nestačí. Naviac, väčšia časť týchto zdrojov bola vynaložená na techniky, ktoré sa uplatňujú na úrovni IP blokov. Tieto techniky ale zlyhávajú v náročnejších subsystémoch a SoC (*System On Chip*), kde je potrebné verifikovať mnoho vzájomne komunikujúcich komponentov.

Významnou výzvou pri zefektívnení verifikácie je znovupoužiteľnosť testov počas celého procesu výroby SoC, od popisu v HDL až po výsledný produkt vo forme čipu. Tento problém sa snaží riešiť nový štandard od firmy Accelera PSS (angl. *Portable Test and Stimulus*). Cieľom je zefektívniť popis samotného testu formou zámeru, ktorý potom dokážeme aplikovať na rôzne platformy počas vývoja.

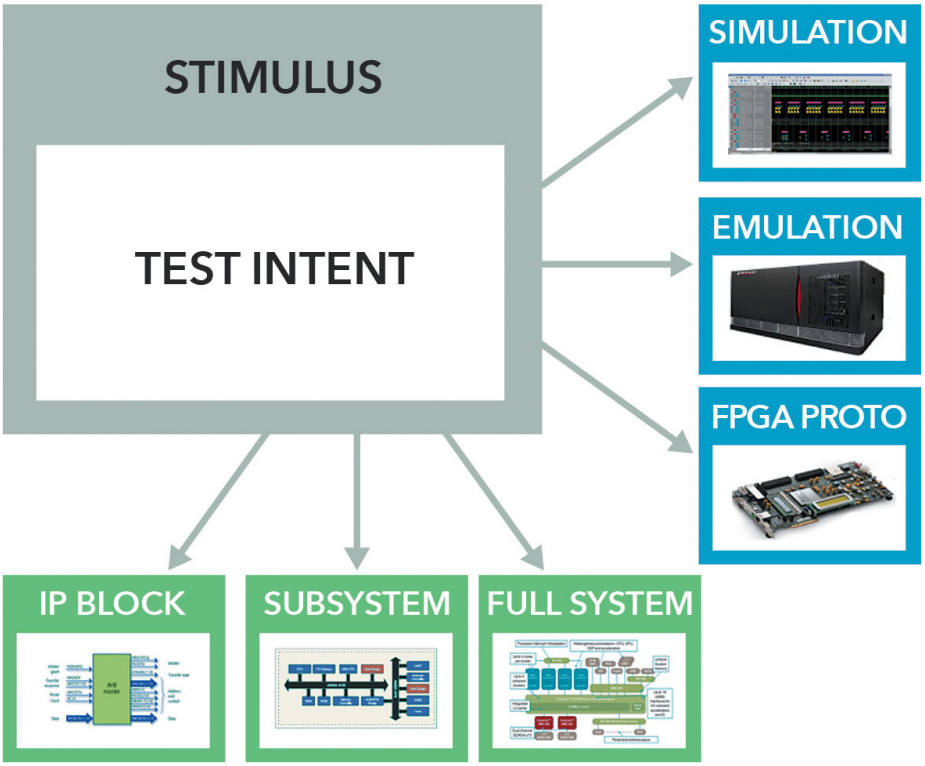
PSS umožňuje definovať jednotnú špecifikáciu prenositeľného testu, ktorá umožní vytvoriť jednu reprezentáciu daného testu, použiteľnú rôznymi inžiniermi na rôznych úrovniach integrácie [20]. Táto jednotná konfigurovateľná reprezentácia nám umožní použiť test pri simulácií, emulácií, prototypovanie na FPGA, ale i na výslednom čipe. Cieľom PSS je snaha o posunutie abstrakcie z úrovne transakcií (angl. *Transaction Level Modeling*, TLM) na úroveň definovania zámeru testu formou scenárov. [3]

Ako je znázornené na obrázku 2.4, PSS cieli na rôzne možnosti znovupoužiteľnosti, ktoré sú znázornené ako osi. Znovupoužitie v rámci osi ma odlišné charakteristiky a mnoho aplikácií počas vývoja prechádzajú niekoľkými fázami.

### 2.3.1 Vertikálne znovupoužitie

V procese tvorby komplexných SoC riešení sa používajú IP bloky alebo subsystémy, ktoré už museli byť verifikované na svojej úrovni. Cieľom PSS je znovupoužitie testovacích zámerov, ktoré boli vyvinuté inžiniermi už počas tvorby IP bloku aj na vyšších úrovniach hierarchie.

Na úrovni IP blokov je dobre zaužívané verifikačné prostredie nad UVM, ktoré je zamerané na transakcie. S využitím štandardu PSS charakterizujeme kľúčové operácie, ktoré môže blok vykonať, a jednotlivé pravidlá o tom, ako sa tieto operácie musia vykonať. Ďal-




Obr. 2.4: Princíp vertikálnej a horizontálnej znovupoužiteľnosti zámeru testu v PSS. [2]

šou úrovňou hierarchie je použitie IP blokov na úrovni subsystémov, kedy PSS umožňuje rozšíriť už definované testy o nové komponenty a obmedzenia bez toho, aby sme museli výrazne meniť pôvodný kód testu.

Na najvyššej úrovni hierarchie, a to pri tvorbe SoC, kedy systém ovláda vložený procesor, je možné testy rozšíriť a generovať konkrétne scenáre, ktoré budú napríklad v jazyku C. Nevýhodou vertikálneho znovupoužitia je, že proces tvorby verifikačných scenárov musí začať už na úrovni IP blokov, ale na úrovni IP blokov je zatiaľ dostačujúca verifikácia pomocou UVM metódy a pseudonáhodného generovania stimulov. Preto je ťažké presvedčiť výrobcov IP blokov o dodávanie verifikačných scenárov nad PSS. [3]

### 2.3.2 Horizontálne znovupoužitie

V rámci vývoja HW[5] systému sa počas celého procesu používajú rôzne testovacie platformy, ktoré majú svoje špecifické charakteristiky na generovanie stimulov či spúšťanie priamych testov. V prvotných fázach vývoja sa HW systém simuluje a v prepojení s UVM verifikačným prostredím je generovanie pseudonáhodných stimulov jednoduché. Simulácia systému je ale mnohonásobne pomalšia, než prototypovanie na FPGA či emulácia.

Na úrovní FPGA prototypov či emulácií sa často používajú testy založené na jazyku C, kde je podpora generovania pseudonáhodných testov obmedzená, preto vzrastá potreba priamych testov. PSS umožňuje definovať iba zámer testu a následne generovať testy, ktoré sú cielené na konkrétny jazyk a prostredie. [3]

[5]HW - hardware

### 2.3.3 Prenositeľné stimuly a UVM

Jedna z hlavých myšlienok PSS sú znovupoužiteľné testy, čo je zároveň jedna z hlavých myšlienok UVM. UVM štandard je pre verifikačných inžinierov dobre pochopiteľný a efektívny, ale pre inžinierov, ktoré nemajú skúsenosti s verifikáciou, je relatívne komplikovaný. Na jednej strane máme verifikačných inžinierov, ktorý majú znalosť UVM, ale nemajú tak dobrú znalosť špecifikácie systému HW komponent. Na druhej strane HW inžinierov a architektov, ktorý majú dokonalú znalosť špecifikácie, ale nepoznajú verifikačné prostredie.

Problém pseudonáhodného generovania testov spočívá v redundancií testov v posledných fázach pokrytia a v slabej konvergencií k 100% pokrytiu. Kdežto test definovaný pomocou grafu PSS dokáže v začiatkoch verifikácie cielene generovať sekvencie, ktoré veľmi rýchlo dosiahnú definované funkčné pokrytie bez zbytočnej redundancie sekvencií.

Zo svojej podstaty je UVM metodika vhodná na verifikáciu IP blokov a subsystémov pomocou pseudonáhodných testov, ale prax ukazuje, že na systémovej úrovni sú pseudonáhodne testy pomerne neefektívne. [26]

## 2.4 Questa inFact

Questa InFact[6] je proprietárna implementácia podmnožiny PSS štandardu od firmy Mentor.

InFact je verifikační nástroj založený na grafovej reprezentácií testu, z ktorého sa generujú stimuly počas funkčnej verifikácie hardwarového designu. Pri použití grafovej reprezentácie testu dokáže InFact cielene aplikovať stimuly na daný bod pokrytia a tak významne redukovať počet redundantných stimulov.

V nasledujúcich podkapitolách postupne vysvetlím základne konštrukcie a princípy, ktoré sa používajú na definíciu grafu a zámerov testu. Nasledujúce podkapitoly čerpajú prevažne z referenčných manuálov a dokumentácie priloženej k nástroju InFact. [8]

### 2.4.1 Štruktúra súboru **.rules*

Základnou štruktúrou verifikačného nástroja Questa InFact je graf, definovaný pomocou textového súboru *.rules*. Tento súbor využíva syntax jazyka určeného pre popis pravidiel a štruktúry grafu. Jednotlivé pravidlá sú spojené do hierarchie, ktorá tvorí graf. Tieto pravidlá sú tvorené z hierarchických výrazov, kde prvky výrazu sú tzv. akcie. Každá z týchto akcii je mapovaná na metódy definované v niektorom z HVL alebo C++, podľa jazyka hlavného testu angl. *testbench*. Pravidlá môžu tvoriť komplexné celky, pomocou ktorých je možné definovať zložitejšie transakcie.

Po kompilácií pravidiel definovaných súborom pravidiel je automaticky zostavený graf, ktorý predstavuje binárnu reprezentáciu súboru pravidiel. Graf je abstraktným zobrazením všetkých možných stimulov definovaných pomocou súboru pravidiel. Počas verifikácie algoritmus prechádza grafom a generuje stimuly na základe definovaných akcií. Priechod týmto grafom generuje sekvencie, ktoré sú následne aplikované na rozhranie DUT a GM. Výhodou grafu je, že predstavuje intuitívnu, ľahko zrozumiteľnú grafickú reprezentáciu stimulov, s ktorou je možné definovať stratégiu pokrytia a tým činom cielene aplikovať stimuly na jednotlivé časti tejto stratégie.

Pri zložitejších systémoch môže graf expandovať a tým zhoršovať prehľadnosť verifikačného prostredia, preto je možné graf rozdeliť na podgrafy, segmenty a deliť ich do jednot-

[6] https://www.mentor.com/products/fv/infact/

livých súborov *.rseg*. Každý graf sa skladá z niekoľkých primárnych konštrukcií, ktoré sú obvykle požadované pri každej verifikácii.

### Akcia

**`action action_name, ... ;`**
Je kľúčové slovo, pomocou ktorého sa deklarujú názvy akcií reprezentujúce funkciu implementovanú v jazyku verifikačného prostredia. Každá takto deklarovaná akcia je súčasťou grafu ako uzol a pokiaľ počas simulácie algoritmus cestou narazí na takúto akciu, iniciuje sa funkcia namapovaná k danej akcii. Akcie je možné použiť v pravidlách alebo sekvenciách, ale musia byť deklarované pred ich použitím. Na príklade kódu 2.2 je znázornené použitie konštrukcie `action`.

```
rule_graph te {
    action init, create_item, finish_item;
    meta_action len [unsigned 15:0];
    eth_tx = init repeat {              // definicia cyklu pomocou repeat
        create_item, len, finish_item   // sekvencne vykonanie akcii
    };
}
```

Výpis 2.2: Príklad použitia kľúčového slova `action`.

### Štruktúra

**`struct struct_name [formal_parameter_list]`**
**`   tends base_struct_name [actual_parameter_list]`**
**`definition_struct ;`**
Kľúčové slovo **struct** sa používa na definovanie hierarchickej štruktúry v rámci pravidiel grafu. Zapuzdruje dáta, biny, obmedzenia, platné na dáta definované v rámci štruktúry, informácie o štruktúre grafu či objektoch, či inštancie iných štruktúr. Tieto objekty môžu byť v rámci grafu niekoľkokrát inštanciované ako uzly grafu.

V rámci grafu je možné definovať pravidlá obsahujúce tieto inštanciované objekty z definovaných štruktúr. Každá štruktúra musí mať definované rozhranie, ktoré sa používa na komunikáciu s objektom počas simulácie. Na ukážke kódu 2.3 je znázornené principiálne použitie štruktúry na zapuzdrenie meta-akcií a k nim priradených obmedzení.

```
rule_graph te {
    struct my_struct {              // definicia struktry s~nazvom my_struct
        meta_action A[0..15];       // definicia meta akcie
        meta_action B[0..15];
        constraint c {A~< B;}       // definicia obmedzenia c pre meta akcie
    }
    my_struct s1, s2;               // instanciace struktury
    ...
}
```

Výpis 2.3: Príklad definície štruktúry `my_struct`, ktorá obsahuje dve meta-akcie `A`, `B`, a k nim priradené obmedzenie `c` na generované dáta.

Jednotlivé štruktúry je možné zanárať a tak tvoriť hierarchiu štruktúr, ktoré sú navzájom prepojené. Štruktúra, ktorá rozširuje nejakú inú štruktúru, dedí všetky meta-akcie a definované obmedzenia. V prípade konfliktných obmedzení sú tieto obmedzenia prepísané novou štruktúrou. Na výpise 2.4 je ukážka definície novej štruktúry `my_struct2`, ktorá rozširuje štruktúru `my_struct`.

```
struct my_struct2 extends my_struct { // rozsirenie struktury my_struct
    meta_action C[0..15];
    constraint c2 { C >= (A+B); }
}
```

Výpis 2.4: Príklad definície štruktúry `my_struct2`. Štruktúra rozširuje štruktúru `my_struct`, z ktorej dedí dve meta-akcie `A`, `B`, obmedzenie `c`, a pridáva novú meta akciu `C` a obmedzenie `c2`.

### Bins a bins scheme

**`bins bins_target bins_name bins_definition;`**
**`bin_scheme bin_scheme_name { bins_defintions };`**
Táto konštrukcia sa používa pokiaľ chceme definovať skupinu hodnôt, ktoré majú v rámci danej meta-akcie rovnakú sémantiku. Biny definujú podmnožiny vstupných dát, z ktorých chceme generovať iba určité množstvo hodnôt, pretože generovanie podobných dát by už nezvyšovalo cielené pokrytie. Pomocou binov dokážeme výrazne redukovať redundantné generovanie stimulov. Pokiaľ pre danú meta-akciu existuje definovaný bin, generujú sa iba hodnoty ktoré spadajú do tohto binu a žiadne iné sa negenerujú. Pri definovaní pravidiel sa používajú implicitné biny, ktoré sa definujú bez názvu, alebo pomenované biny, ktoré dokážu prepísať implicitní bin.

- **bins_target** - parameter mapuje bin k *meta_action* alebo *meta_action_import*.
- **bins_name** - udáva názov binu, ktorý môže byť následne použitý v bin_scheme.
- **bins_definition** - je povinný parameter, obsahuje definíciu validných hodnôt, ktoré môže algoritmus generovať.

Pri špecifikácií je možné použiť niekoľko operátorov. Operátor delenia *'/'*, ktorý delí generovaný rozsah *meta_action* na niekoľko podskupín, kde z každej skupiny generuje jednu hodnotu. Operátor veľkosti binu *':'*, ktorý udáva z daného rozsahu veľkosť skupiny binu. Špeciálny operátor *wildcard '*'* reprezentuje všetky hodnoty, ktoré doposiaľ neboli generované. Detailnejšie vysvetlenie a názorné použitie binov je na ukážke kódu 2.5.

```
rule_graph te {
    meta_action size[0..4095];
    meta_action addr[0..255];
    bins size [1][2][4094][4095][*]; // 5 binov
    bins size [1..4094] : 500        // 8 binov

    bins size sz_bns_1               // meno binu
        [0]                          // 1 bin
        [1..2000] : 500              // 4 biny
        [2001..4000] / 5             // 5 biny vekosti 400
        [4001..4094] [4095];

    bin_scheme sz_addr_sch {
        size sz_bns_1;
        addr [0][1..254][255];
    };
}
```

Výpis 2.5: Príklad definície binov a `bin_scheme`.

### Obmedzenie (angl. *constraint*)

`constraint constraint_name constraint_type { constraint_definition };`
Pomocou kľúčového slova **constraint** definujeme obmedzenie, ktoré môže byť aplikované na celý graf, jednotlivé akcie, premenné alebo prvky vyhodnocujúce pokrytie. Obmedzenia majú rozsiahle a komplexné možnosti definície, často používané konštrukcie sú znázornené na ukážke 2.6.

Existujú tri druhy obmedzení typu **constraint_type**, ktoré môžeme pri definícii použiť:

- **static** - statické obmedzenie, musí platiť vždy pokiaľ sa vyhodnocuje daný komponent grafu, ku ktorému je mapovaný. Implicitne je každé obmedzenie statické.
- **dynamic** - dynamické obmedzenie je závislé na ceste, ktorou algoritmus prešiel počas behu simulácie. Aplikuje sa iba v prípade, že sa vyhodnocoval uzol, ku ktorému je obmedzenie mapované.
- **coverage** - tento typ obmedzenia sa používa pri definícií pokrytia a je priradený k stratégií pokrytia.

**Constraint_definition** poskytuje pomerne komplikované možnosti definície obmedzenia. V príklade 2.6 uvediem najčastejšie používane konštrukcie, pre podrobnejšie pochopenie doporučujem referenčný manuál [8].

```
1  rule_graph te {
2      meta_action src_addr[0..63];
3      meta_action dest_addr[0..63];
4      meta_action len[0..63];
5      constraint len_c {
6          len >= 4;       // len musi byt >= nez 4
7      };
8      constraint on_addr1 {src_addr != dest_addr};
9      constraint on_addr1 static {if (addr == 0) {size inside [small]}};
10     constraint c {
11         dest_addr[9] == dest_addr[8]+2;
12     ...
```

Výpis 2.6: Príklad definície konštrukcie `constraint`.

### Symbol

`symbol symbol_name = rule_declarations;`
Pomocou konštrukcie symbol je možné spojiť skupinu pravidiel, priradiť im názov a takto vytvárať hierarchiu pravidiel. Každý definovaný symbol musí byť použitý v rámci nejakej akcie. Na nasledujúcom obrázku 2.7 je znázornený príklad použitia symbolu.

```
1  rule_graph te {
2      action reset, do_read, do_write;
3      meta_action address[0, 5, 10, 16..32], ctrl[1..5];
4      symbol do_transaction = address ctrl;
5      ...
6  }
```

Výpis 2.7: Príklad definície a využitia konštrukcie `symbol`.

### Tag

`action_node = tag tag_name alternative | tag tag_name alternative;`
Pomocou tagu je možné v štruktúrach, ktoré vykonávajú vetvenie grafu, priradiť jednotlivým uzlom pravdepodobnosť a tak efektívnejšie nastaviť generovanie sekvencií. Na ukážke kódu 2.8 je príklad použitia konštrukcie `tag`. Pomocou tagu môžeme napríklad:

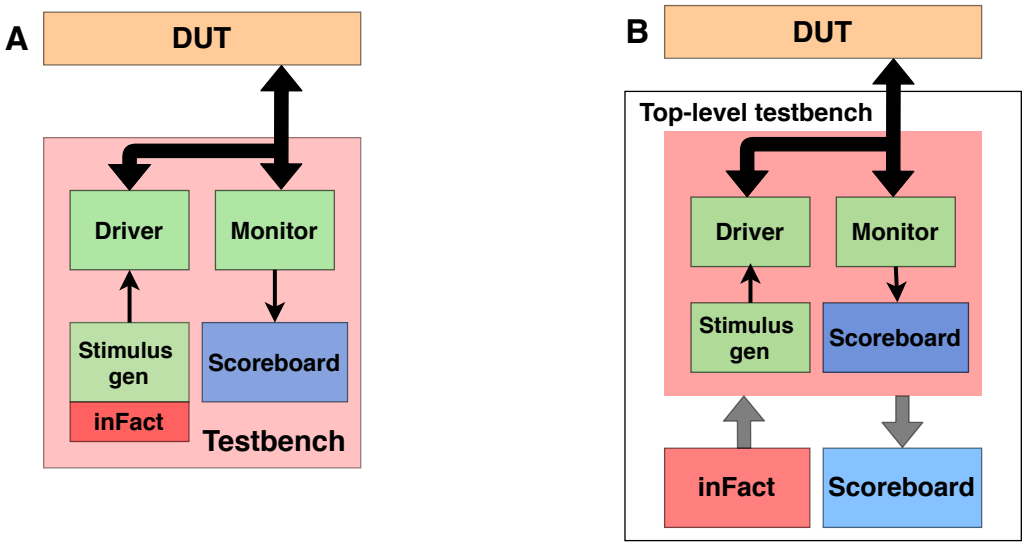
- cieliť simuláciu na iné, zaujímavejšie cesty grafu,
- cieliť na problémovú časť grafu v ktorej môžu byť chyby,
- vypnúť funkcionalitu, ktorá doposiaľ nebola implementovaná,

```
1 rule_graph te {
2     meta_action address[0, 5, 10, 16..32], ctrl[1..5];
3     symbol RW_opts;
4     RW_opts = address ctrl tag read_tag do_read | tag w_prob do_write;
5     te = reset RW_opts;
6 }
```

Výpis 2.8: Príklad definície konštrukcie `tag`.

### 2.4.2 Prepojenie InFact sequencie s UVM

Na obrázku 2.5 je znázornené prepojenie InFact testovacieho komponentu s existujúcim verifikačným prostredím UVM.




Obr. 2.5: Podobrázok A ukazuje zapojenie inFact testovacej komponenty k účelu riadenia a kontrolovania generovania stimulov. Podobrázok B ukazuje zapojenie s cieľom kontrolovania existujúcej testbench konfigurácie. Všetky tri možnosti zapojenia nepredstavujú zložitú zmenu vo verifikačnom prostredí. V závislosti na aplikácii a verifikačnom inžinierovi môže byť inFact testovacia komponenta prepojená na existujúce verifikačné prostredie niekoľkými spôsobmi. Na obrázku sú znázornené dve možnosti zapojenia, pričom tretie zapojenie spočíva v kombinácií obidvoch prepojení A a B.

# Kapitola 3

# Evolučné algoritmy

Pojem evolučné algoritmy (angl. *evolutionary algorithms*, EA) zahŕňa mnoho heuristických algoritmov, ktoré majú spoločnú inšpiráciu v evolúcií. EA spadajú pod obor algoritmov umelej inteligencie (angl. *Artifical Intelligence*, AI), konkrétnejšie do oboru nazývaného Soft Computing (SC). Oproti ostatným metódam v oboru AI sa vyznačujú väčšou toleranciou nepresnosti a neistoty, riešenia sú založené na aproximácii a značne využívajú prvky náhody. Preto je ťažké matematicky definovať, kedy a za akých podmienok budú algoritmy dobre konvergovať ku globálnemu maximu. [23]

Prvé zmienky o evolučných algoritmoch sa datujú do 40 rokov 19. storočia, kedy boli popísané prvé myšlienky o využití Darwinovej teórie pri automatizácii riešenia problémov. Širšiemu rozvinutiu evolučných algoritmov bránil fakt, že počítače ešte neboli tak dostupné a ich výkon nebol nijak závratný. Za prvý väčší rozvoj evolučných algoritmov sa považujú 60 až 70 roky 19. storočia, kedy vznikli nezávisle na sebe tri rôzne metódy obsahujúce prvky evolúcie za cieľom optimalizácie výpočtu riešenia. Prvú metódu popísali v USA L.Fogel, OWens a Walsh, ktorú nazývali evolučné programovanie. [10]

V roku 1975 Holland nazýval svoju metódu genetický algoritmus [28]. A v 60. rokoch 19 storočia Rechenberg a Schwefel, TU Berlin, skúmali optimalizáciu mechanických konštrukcií pomocou evolučnej stratégie [6]. Koncom 80. rokov bola formulovaná ďalšia metóda EA a to genetické programovanie (GP, angl. *genetic programmming*) [31]. Po dobu asi 15 rokov sa tieto metódy vyvíjali oddelene a nezávisle na sebe, až do začiatkov 1990, kedy na základe publikácie [24] bol definovaný nový obor pod názvom angl. *Evolutionary Computing*.

Všetky evolučné algoritmy majú spoločnú základnú myšlienku a to je darvinovská evolúcia. Jedinec, ktorý ma v rámci daného prostredia najlepšie gény, prežije a odovzdáva svoje gény ďalej.

Jedinec predstavuje potenciálne riešenie problému. Reprezentácia jedinca predstavuje chromozóm a jeho hodnota predstavuje jeden z možných stavov prehľadávaného priestoru. Každý jedinec je v rámci EA ohodnocovaný pomocou fitness funkcie, ktorá určuje kvalitu jedinca v danom prostredí. Jej výstup je fitness hodnota, ktorá predstavuje rozhodujúcu rolu pri mechanizme selekcie jedincov. Každý genetický algoritmus má v priebehu výpočtu niekoľko jedincov, ktorých nazývame populácia. Generácia predstavuje jednu iteráciu evolučného algoritmu, v rámci ktorej dochádza k selekcii rodičov a generovaniu nových jedincov pomocou variačných operátorov. Podmienkou ukončenia evolučného algoritmu býva buď dosiahnutie kandidátneho riešenia v podobe cieľovej fitness funkcie, presiahnutie maximálneho počtu generácií, alebo pokiaľ sa fitness hodnota najlepšieho jedinca už dlhšiu dobu nezlepšuje. [13]

## 3.1 Základné pojmy a techniky EA

I keď sa z pohľadu histórie evolučné algoritmy vyvíjali oddelene, majú spoločný základ, ktorý je v princípe jednoduchý. Vygenerovať populáciu jedincov, ktorú následne vložíme do prostredia s obmedzenými zdrojmi. I kdeže sa to môže zdať kruté, táto evaluácia vyústi v prežití jedincov, ktoré dokážu zaobstarať a využiť zdroje najefektívnejšie. Následne sa ako základ ďalšej generácie vyberie niekoľko jedincov, ktoré prežili evaluáciu. Pomocou variačných operátorov krížením a mutáciou vzniknú nové potomkovia a pokračuje sa ďalšou iteráciou evaluácie. Ako je znázornené na ukážke algoritmu 1, základ evolučného algoritmu je jednoduchý a v intuitívny. To najťažšie na evolučnom algoritme je jeho správne nastavenie tak, aby vykazoval dostatočnú exploráciu a diverzifikáciu v priestore a tak neuviazol iba na lokálnom minime. Zároveň v priebehu simulácie musí smerovať k intenzifikácii riešenia pomocou zužitkovania informácií o okolí stavového priestoru. V nasledujúcich kapitolách vysvetlím jednotlivé kroky algoritmu a detailnejšie popíšem aké techniky sa pri ich aplikáciách využívajú. Nasledujúca kapitola čerpá prevažne z týchto zdrojov [13, 12, 6].

**Algoritmus 1:** Všeobecný algoritmus evolučného algoritmu. Prevzatý z knihy [13]

**Inicializácia** populácie P0
**Evaluácia** jedincov z P0
**while** *podmienka ukončenia* **do**
  **Výber** rodičov z P
  **Kríženie** vybraných rodičov
  **Mutácia** výsledného potomka
  **Evaluácia** nových jedincov
  **Selekcia** najlepších jedincov pre ďalšiu generáciu
**end**

### 3.1.1 Kódovanie jedincov

Jedným z najdôležitejších prvkov nastavenia evolučného algoritmu je zvolené kódovanie jedincov[1]. Kódovanie vytvára prepojenie medzi problémom z reálneho sveta a simulovaným prehľadávaným priestorom evolúcie. Tento prístup vyžaduje určitú abstrakciu a zanedbanie vlastností reálneho sveta, na ktorých nám pri riešení problému nezáleží. Cieľom je zjednodušiť prehľadávaný priestor tak, aby bol dostatočne diverzifikovaný a umožňoval i nekonvenčné riešenia, ale zároveň aby nebol príliš zložitý a tým nezhoršoval konvergenciu evolúcie. [13]

Popis jedinca v reálnom svete sa nazýva fenotyp a jeho reprezentáciu, vhodnú pre použitie v počítačoch, nazývame genotyp. Niektoré algoritmy nerozlišujú medzi genotypom a fenotypom, naopak v niektorých prípadoch sú genotyp a fenotyp úplne odlišný. Najbežnejšie podoby genotypu:

- **Reprezentácia pomocou grafu** - genotyp je zakódovaný pomocou hierarchického grafu, kde uzol predstavuje niektorú z definovaných funkcionalít. V týchto prípadoch je kódovanie často popísané pomocou gramatiky. Variační operátori manipulujú s jednotlivými uzlami grafu alebo s celými podgrafmi. Táto reprezentácia sa používa napríklad pri symbolickej regresii, genetickom programovaní a podobne.

[1] Vhodne zvolené kódovanie problému často rozhoduje o efektívnosti a rýchlosti konvergencie evolúcie, preto je potrebné mu venovať dostatok času.

- **Binárna reprezentácia** - kódovanie genotypu predstavuje vektor bitov, pričom variačné operácie predstavujú základne binárne operácie. Používa sa napríklad v *evolučnom developmente*.
- **Celočíselná reprezentácia** - genotyp je tvorený z postupnosti celých čísel, ktoré reprezentujú nejakú akciu, konštantu, variantu špecifickú pre daný problém. Tento prístup sa používa napríklad pri gramatickej evolúcii, kartézskom genetickom programovaní a podobne.
- **Reprezentácia pomocou reálnych čísiel** - genóm je tvorený postupnosťou reálnych čísel, pričom variační operátori predstavujú jednoduché operácie nad *float-point* aritmetikou. Typickým reprezentantom tohto prístupu je evolučná stratégia.

Ostatné reprezentácie sú špecifické pre riešený problém, kedy sa často využíva kombinácia predchádzajúcich spôsobov.

### 3.1.2 Populácia a jej inicializácia

Populáciu môžeme definovať ako množinu jedincov, ktoré sú v aktuálnom kroku evaluácie základom pre tvorbu novej populácie. V rámci evaluácie je každý jedinec vystavený selekčnému tlaku, čo rozhoduje o jeho prežití. Populácia sa v priebehu evolučného algoritmu mení a vyvíja. Jej veľkosť býva často konštantná, ale nie je to pravidlo. Veľkosť populácie je jeden z faktorov, ktorý vplýva na efektivitu a rýchlosť konvergencie a často je predmetom experimentu v rámci riešenia problému. [12] Inicializácia populácie býva často veľmi jednoduchá a to formou náhodného vygenerovania jedincov. V niektorých aplikáciách sa používajú heuristiky s cieľom vygenerovať jedincov, ktoré by mohli predstavovať základy kandidátneho riešenia, ale v praxi je prínos tejto metodiky veľmi diskutabilný. [13]

### 3.1.3 Výber rodičov

Cieľom heuristiky výberu rodičov (angl. *parent selection*) je na základe ohodnotenia rodičov pomocou fitness funkcie, reprezentujúcej ich kvalitu, umožniť viac kvalitným rodičom posunúť svoje gény do ďalšej generácie. Viac kvalitné jedinci by mali mať väčšiu pravdepodobnosť výberu, ale zároveň i menej kvalitné jedinci musia mať malú ale pozitívnu šancu predať svoje gény, inak by mohol algoritmus uviaznuť v lokálnom maxime. Tento mechanizmus spolu zo selekciou má za cieľ intenzifikáciu kvality jedincov. V nasledujúcich kapitolách popíšem najčastejšie používané metódy výberu rodičov[2]. [13]

#### Výber na základe fitness funkcie

Výber na základe fitness funkcie (angl. *fitness proportional selection*, FPS) je jeden z prvých a najjednoduchších metód výberu, kde výber jedinca je priamo úmerný jeho kvalite. Pravdepodobnosť výberu jedinca je závislá na absolútnej hodnote jeho fitness voči absolútnej hodnote ostatných jedincov v populácii. Súčet pravdepodobností jedincov naprieč celou populáciu musí byť 1. [6]

Na prvý pohľad je to intuitívna metóda, ale bola podrobená niekoľkými štúdiami, kde sa ukázalo, že trpí zásadnými nedostatkami. Jedným z nich je fakt, že jedinci, ktoré nemajú na začiatku moc dobrú fitness hodnotu, sú vyselektované hneď v prvých fázach evolúcie, a to zvyšuje pravdepodobnosť uviaznutia v lokálnom maxime. V situácii, kedy máme jedincov

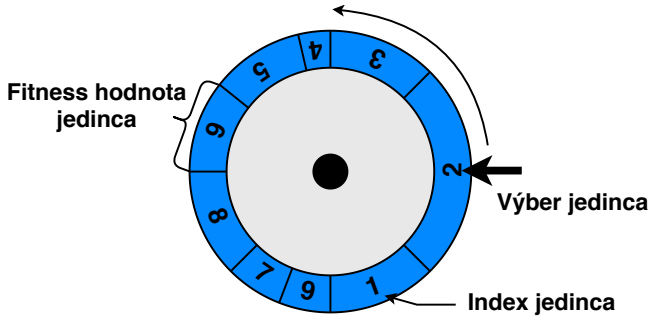
[2]Existuje mnoho ďalších metód výberu rodičov, ale pre jednoduchosť ich nebudem uvádzať.

s veľmi podobnou fitness hodnotou nenastáva skoro žiadny selekčný tlak a tak lepšia fitness hodnota nemá skoro žiadny význam.

Vylepšením tohoto prístupu je výber na základe poradia, kedy hlavný rozdiel spočíva v zoradení jedincov podľa ich fitness hodnoty a pravdepodobnosť ich výberu je závislá na ich poradí. Týmto spôsobom sa eliminujú nedostatky FPS popísané vyššie. [12]

#### Výber na základe rulety

Princíp tejto metódy sa dá prirovnať k rulete alebo otočnému kolesu (viď obrázok 3.1), kedy okraj kolesa bude predstavovať jedincov a veľkosť jednotlivého políčka bude adekvátna k jeho fitness hodnote. Následne zatočíme kolesom a vyberieme jedného jedinca. Nevýhodou selekcie jedincov pomocou rulety je zložitá paralelizácia evolučného algoritmu.




Obr. 3.1: Ilustrácia výberu jedinca na základe rulety.

#### Výber na základe turnaja

Predchádzajúce dva prístupy k výberu rodičov využívali pravdepodobnosť distribuovanú cez celú populáciu, čo môže predstavovať problém v prípade že máme príliš veľkú populáciu[3]. Selekcia na základe turnaja nepotrebuje žiadnu globálnu znalosť o populácii, jej princíp spočíva v náhodnom výbere z jedincov, ktoré navzájom súperia porovnaním ich fitness hodnoty. Jedinec s najvyššou fitness hodnotou v rámci skupiny vyhráva turnaj. Výhoda algoritmu je v rovnomernom zachovaní diverzity a konvergencie.

### 3.1.4 Kríženie

Jeden z variačných operátorov je kríženie (angl. *crossover*), ktorého primárnou úlohou je vytvorenie nového jedinca z vybraných rodičov replikáciou genotypu rodičov. Cieľom je generovať kvalitnejších potomkov než ich rodičia, čo je založené na predpoklade, že pokiaľ vezmeme dvoch kvalitných jedincov a skombinujeme ich genotypy, vznikne podobne kvalitný jedinec. V 90. rokoch bola vznesená pochybnosť prínosu kríženia pri evolúcii, čo vyústilo k rade experimentov, ktoré potvrdili jeho prínos hlavne v neskorších fázach evolúcie [12]. Implementácia kríženia je často závislá na použitom evolučnom algoritme a konkrétnom kódovaní jedincov. [13]

[3]Sekvenčný výber jedincov môže brániť paralelizácii či spotrebovávať až príliš veľa výpočtových zdrojov.

### 3.1.5 Mutácia

Mutácia je jeden z hlavných variačných operátorov evolúcie pri zaisťovaní nových vlastností jedincov, ktoré zatiaľ neboli preskúmané a tak nemôžu vzniknúť pomocou kríženia. Princíp mutácie spočíva v náhodnej zmene časti genotypu jedinca a tým zaistení nových vlastností mutanta. Dôležité je, aby malá mutácia spôsobila malú zmenu vlastností jedinca. V počiatočných fázach EA je dôležitejšia mutácia než kríženie, kedy mutácia zaisťuje lepšie prehľadávanie priestoru do šírky, naopak v neskorších fázach evolúcie je dôležitejšie kríženie medzi jedincami a malá pravdepodobnosť mutácie. Správne nastavenie parametrov variačných operátorov je kľúčové pri rýchlosti konvergencie riešenia a ich ladenie vyžaduje určité skúsenosti. [13]

### 3.1.6 Evaluácia

Evaluácia, taktiež známa pod pojmom fitness funkcia (angl. *Fitness Function, Evaluation Function*), je metóda, ktorej cieľom je kvantifikovať kvalitu jedinca v danom prostredí. Definuje vlastnosti, na ktorých nám záleží a určuje prínos zlepšenia kvality jedinca. Typicky fitness funkcia robí prevod z genotypu na fenotyp, ktorý sa simuluje v abstrahovanom svete, a tým odvodí jeho kvality. Konkrétna implementácia fitness funkcie je veľmi špecifická od riešeného problému a jej podoba je kľúčová v smerovaní EA. Z grafického hľadiska nám fitness funkcia udáva štruktúru stavového priestoru. [12]

### 3.1.7 Selekcia

Selekcia je zodpovedná za manažment populácie rozdelením jedincov na tých, ktoré prežijú do ďalšej generácie, a tých, ktoré neprežijú. Tento krok je v niektorých evolučných metódach spojený spolu s výberom rodičov, ale v obecnom evolučnom algoritme je selekcia rozdelená na výber rodičov a výber jedincov pre nasledujúcu generáciu. Tento proces sa v odbornej literatúre nazýva angl. *replacement*.

Výmena populácie na základe veku jedinca je založená na princípe, že každý jedinec v populácii existuje stanovený počet iterácií evolučného algoritmu, pričom sa zanedbáva jeho fitness hodnota. Výmena populácie potom môže byť generačná, kedy jedna nová generácie vymení celú starú generáciu, alebo postupná výmena, kedy sa jednotlivé jedinci vyberajú na základe veku a náhody. Pri tejto metóde môže dočasne dochádzať k prípadom, že nová generácia ma horšiu priemernú fitness, než predošlá generácia. Pokiaľ k tejto situácii nedochádza príliš často, tak sa nepovažuje za problém. [6]

Výmena populácie na základe fitness hodnoty jedincov je často založená na podobných princípoch ako výber rodičov, popísaných v kapitolách vyššie. Medzi ďalšie zaujímavé techniky patrí výmena najhorších potomkov, kedy vyberieme skupinu jedincov s najhoršou fitness, a tých z populácie odstránime. [13]

Elitizmus je technika, kedy jedinec s najlepšou hodnotou fitness je vždy súčasťou novej generácie.

### 3.1.8 Podmienka ukončenia EA

Ukončovacia podmienka evolučného algoritmu je veľmi individuálna, záleží na riešenom problému a očakávanom výsledku. Záleží na skúsenostiach inžiniera, aby odhadol tu správnu techniku na meranie fáze evolučného algoritmu. Typicky môžeme riešené problémy rozdeliť

na také, pri ktorých vieme hodnotu maximálnej fitness hodnoty, a na také, kedy maximálnu hodnotu fitness nepoznáme. [13] V takom prípade sa často používajú tieto kritéria:

1. Prekročenie alokovaného výpočtového výkonu.
2. Počet evaluácií prekročilo maximálnu zvolenú hodnotu.
3. Fitness hodnota jedincov sa už určitú dobu výrazne nemení.
4. Diverzifikácia populácie sa už výrazne nemení.

## 3.2 Genetické programovanie

Genetické programovanie (angl. Genetic Programming, GP) je relatívne mladá stratégia z oblasti evolučných algoritmov, ktorá sa začala formovať na konci 80. rokov, ale o jej rozvoj sa preslávil hlavne John Koza (1990) s využitím programovacieho jazyka LISP. Hlavným rozdielom oproti ostatným EA je oblasť, v ktorej sa používa kódovanie jedincov, ktoré je typicky formou stromu[4]. Ostatné evolučné algoritmy mali ako hlavný cieľ hľadať optimálne hodnoty parametrov zakódovaných v genóme jedinca, ale GP má za cieľ generovať spustiteľné štruktúry, ktoré produkujú optimálne hodnoty. GP bolo už niekoľkokrát použité na riešenie ťažkých inžinierskych problémov, kde boli dosiahnuté významné úspechy[5]. Ako reprezentácia jedincov je často používaná hierarchická štruktúra strom (anlg. tree), prípadne jazyk symbolických inštrukcií, syntaktický strom ale i reťazec hodnôt. Všetky tieto reprezentácie majú spoločnú vlastnosť a to, že majú rôznu dĺžku genómu a v rámci evolúcie sa táto dĺžka mení. Variačné operátory pracujú nad týmito štruktúrami, kedy kríženie a mutácia musia generovať validné hodnoty, a tak nie sú čisto náhodné, ale musia spĺňať gramatické pravidlá. Výpočet fitness hodnoty spočíva v dekódovaní jedinca na spustiteľnú formu, následne je spustená simulácia so vstupnými parametrami. Po simulácii je z výstupných hodnôt vypočítaná fitness hodnota. [31] Pseudo-algoritmus GP je takmer totožný s všeobecným algoritmom EA (viď výpis 1).

## 3.3 Gramatická evolúcia

Gramatická evolúcia (angl. *Gramatical Evolution, GE*) je relatívne mladý obor na poli evolučných algoritmov, ale v posledných rokoch sa teší veľkej popularite z dôvodu, že GE je možné použiť na riešenie mnoho problémov. Stačí vhodne špecifikovať gramatiku, ktorá bude generovať syntakticky správnych jedincov.

Jedna zo špecifík GE je fakt, že existuje nekonečne mnoho gramatík, ktoré definujú tú istú syntax ale každá gramatika ma trochu inú efektivitu v mapovaní genómu jedincov na zodpovedajúci fenotyp. Pri tvorbe gramatiky je potrebné myslieť na to, že zvolená gramatika je úzko spojená s procesom mapovania a efektivitou prehľadávania priestoru. [32]

Gramatická evolúcia používa lineárne kódovanie jedincov s variabilnou dĺžkou efektívneho genotypu jedinca. Interpretácia kodónu je pozične závislá a závisí na predošlých deriváciách. Z tohoto dôvodu niektoré variačné operátory genetických algoritmov nie sú úplne vhodné pre použitie v gramatickej evolúcii [27]. V nasledujúcej kapitole 4, budú podrobne popísané konkrétne používané techniky v gramatickej evolúcii.

[4] Ale nie nutne iba strom, existujú i iné kódovania používané v genetickom programovaní.
[5] CGP, symbolická regresia a podobne.

# Kapitola 4

# Návrh riešenia

Cieľom navrhovaného riešenia je pomocou gramatickej evolúcie hľadať takú skladbu grafu v proprietárnom jazyku InFact[1], ktorá bude dosahovať čo najlepšie funkčné i štrukturálne pokrytie verifikovanej komponenty.

Pre použitie frameworku musí užívateľ vytvoriť množinu transakcií, ktoré predstavujú základné prvky, z ktorých gramatická evolúcia tvorí graf. Táto množina musí byť podmnožinou terminálnych symbolov gramatiky, z ktorej evolučný algoritmus tvorí kandidátnych jedincov. Podrobné vysvetlenie tohto konceptu i s príkladom tvorby takejto podmnožiny sa rozoberá v kapitole 6.3.

Ďalšou dôležitou časťou evolúcie je gramatika, podľa ktorej algoritmus mapuje genotyp jedincov na fenotyp[2]. V rámci experimentov vzniklo niekoľko gramatík, ktoré sú dostatočné obecné a znovupoužiteľné. Užívateľ si môže upraviť preddefinovanú gramatiku, ale i nepatrná zmena gramatiky môže mať signifikantný dopad na generovaných jedincov.

Vytvorený graf sa pomocou nástroja Questa InFact skompiluje a následne sa za použitia pripraveného verifikačného prostredia v UVM[3] spustí simulácia verifikovanej komponenty. Na základe nameraného funkčného a štrukturálneho pokrytia sa vypočíta fitness hodnota jedinca, ktorá sa ďalej propaguje do evolučného algoritmu. Výsledkom evolučného algoritmu je jedinec či skupina jedincov, ktoré redukujú počet potrebných transakcií[4] a zároveň dosahujú dostatočné štrukturálne a funkčné pokrytie.

Na začiatku tejto kapitoly popíšem návrh riešenia problému (podkapitola 4.1), ktorý bude dostatočne všeobecný pre riešenie zvoleného problému. Zároveň bude načrtnutý pseudokód algoritmu 2 a architektúra frameworku, jednotlivé komponenty a ich vstupy a výstupy na obrázku 4.1.

V ďalších častiach tejto kapitoly podrobnejšie rozoberiem kódovanie jedincov a mapovanie genotypu na fenotyp (viď podkapitola 4.1.1). Navrhovaná metóda pre inicializáciu počiatočnej populácie $P_0$ je vysvetlená v sekcii 4.1.2. Princíp fungovania a navrhované metódy pre variačné operátory evolučného algoritmu sú vysvetlené v podkapitolách Mutácia jedinca 4.1.3 a Kríženie jedincov 4.1.4. V poslednej časti tejto kapitoly 4.2 je popísaný návrh štruktúry a postup evaluácie jedinca, respektíve výpočet jeho fitness hodnoty.

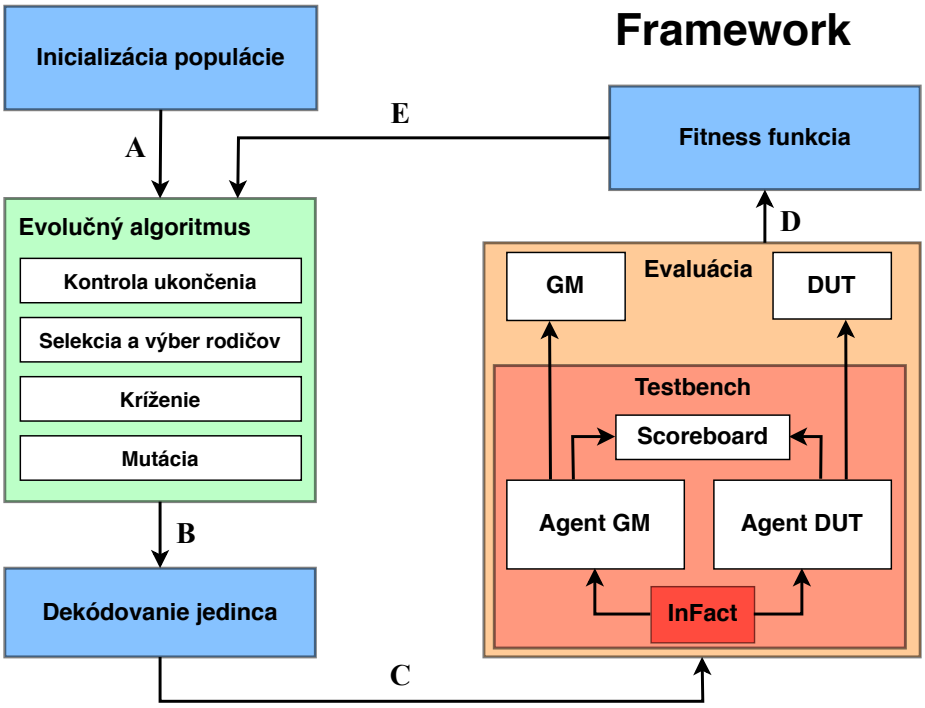
Informácie pre nasledujúce kapitoly boli prevažne čerpané z publikácie [39], ktorá obsahuje množstvo prác vzťahujúcich sa ku gramatickej evolúcii do jednej publikácie.

[1] https://www.mentor.com/products/fv/infact/

[2] Fenotyp jedinca predstavuje graf v proprietárnom jazyku InFact.

[3] UVM - angl. *Universal Verification Methodology*, predstavená v kapitole 2.2.

[4] Redukujú počet transakcií, oproti pseudonáhodne generovaným transakciám v UVM prostredí.




Obr. 4.1: Architektúra frameworku pre generovanie a optimalizáciu grafu.
A - počiatočná populácia $P_0$, B - zakódovaný jedinec, C - graf v proprietárnom jazyku InFact `*.rules` a `*.rseg`, D - metriky pokrytia, E - fitness hodnota evaluovaného jedinca.

## 4.1 Popis funkcionality frameworku

Prvým krokom navrhnutého evolučného algoritmu je inicializácia populácie $P_0$ pomocou zvolenej metódy RHH[5]. Podrobnejší popis funkcionality metódy RHH je v podkapitole 4.1.2. Po inicializácii je populácia $P_0$ dekódovaná, čo znamená, že pre každého jedinca mapujeme genotyp na jeho zodpovedajúci fenotyp. Kódovaním a mapovaním jedincov sa bližšie zaoberá kapitola 4.1.1. Následne musí byť populácia $P_0$ podrobená evaluácii, kde sa každému jedincovi priradí jeho fitness hodnota. Takto inicializovaná a ohodnotená populácia je odovzdaná ako vstup hlavnému cyklu evolučného algoritmu.

Evolučný algoritmus v prvom kroku výpočtu skontroluje podmienku ukončenia, ktorá spočíva v maximálnom počte generácií. Evolučný algoritmus kontroluje, či sa hodnota najlepšieho jedinca za posledných niekoľko generácií zlepšila. V prípade, že sa nezlepšila, algoritmus je ukončený predčasne. Tento prístup bol zvolený práve z dôvodu, že dopredu nepoznáme ani približnú fitness hodnotu, ktorú môžu kandidátni jedinci dosahovať. Počet generácií a ďalšie parametre evolučného algoritmu je možné nastaviť pomocou konzolových parametrov alebo konfiguračného súboru, ktorými sa bližšie zaoberá kapitola 5.2.

Selekcia jedincov do novej generácie a výber rodičov prebieha v jednom kroku, pričom ako metódu som zvolil turnajový algoritmus, ktorý je ľahko paralelizovateľný. Pri selekcii jedincov je zavedený elitizmus, ktorý by mal dopomôcť k stálej konvergencii. Tvorba novej populácie je navrhnutá pomocou uniformného kríženia s variabilným bodom kríženia, kde sa z aktuálnej populácie turnajovým algoritmom vyberú dvaja rodičia, na ktorých je

[5]RHH - angl. *ramped half-and-half*

**Algoritmus 2:** Pseudo-kód evolučného algoritmu

**Inicializácia** populácie $P_0$
**Evaluácia** všetkých jedincov z $P_0$
**while** *Aktuálna iterácia* $i < MAX\_ITERATION$ **do**
  **Selekcia**: na základe turnajového algoritmu vyber $x$ jedincov z $P_i$ do intermediárnej populácie $P'$
  **Elitizmus**: jedinca s najlepšou fitness hodnotou vlož do populácie $P_{i+1}$
  **for** $l = 1$ **to** $u/2$ **do**
    **Vyber rodičov** : pomocou turnaja vyber dvoch jedincov $p_1, p_2 \in P'$
    **Kríženie** : s pravdepodobnosťou $p_{cross}$ vytvor dvojicu potomkov $m_1, m_2$ pokiaľ ku kríženiu nedošlo, tak $m_1 = p_1, m_2 = p_2$
    **Mutácia** : s pravdepodobnosťou $p_{mut}$ mutuj každý gén v $m_1, m_2$
    Vlož nové jedince $m_1, m_2$ do populácie $P_{i+1}$
  **end**
  **for** $k = 0$ **to** $|P_i|$ **do**
    Dekóduj jedinca $P_i$[k] a vytvor reprezentáciu grafu, fenotyp
    Vygeneruj súbor `*.rules` a `*.rseg`
    $fitness^k$ = simulácia jedinca $P_i$[k]
  **end**
**end**

s pravdepodobnosťou $p_{cross}$ aplikovaný operátor kríženia. V prípade, že na vybraných rodičov nieje aplikované kríženie, jedinci postupujú v nezmenenej forme do ďalšej populácie. Po aplikovaní variačného operátora kríženia, ktorý je popísaný v sekcií 4.1.4, a zavedení elitizmu, vzniká nová populácia $P_{i+1}$. Následne je na každého jedinca aplikovaná mutácia 4.1.3 s pravdepodobnosťou $p_{mut}$.

Ako výstup evolučného algoritmu môžeme považovať upravenú populáciu $P_{i+1}$ ktorú je potrebné ohodnotiť pomocou simulácie a výpočtu fitness hodnoty 5.4. Každého jedinca z populácie dekódujeme a vytvoríme tak zdrojový kód, syntaxou popisujúci pravidlá grafu pre nástroj Questa InFact. Po kompilácii grafu je spustená simulácia a generovanie stimulov pomocou grafu, ktorý je napojený na verifikačné prostredie vytvorené nad UVM metodikou. Podrobný popis evaluácie jedinca je navrhnutý v podkapitole 4.2.

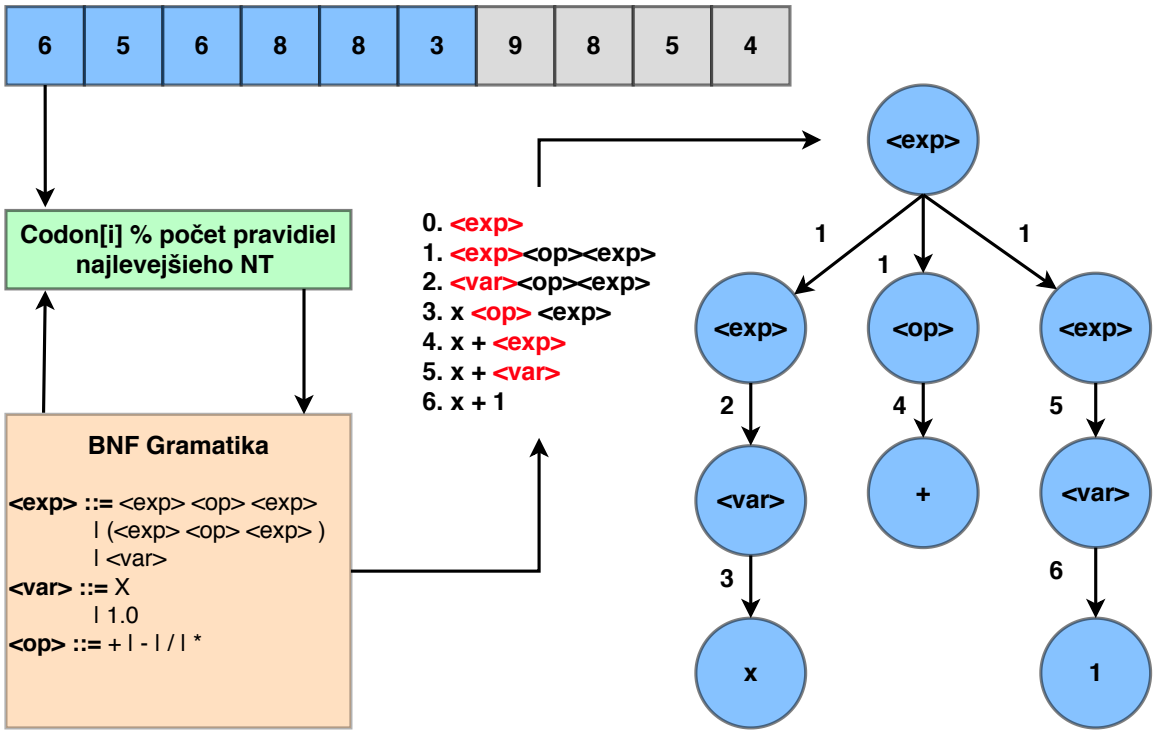
Výstupom simulácie je dosiahnuté štrukturálne pokrytie a funkčné pokrytie, z ktorého je následne vypočítaná fitness hodnota všetkých jedincov v populácii. Takto ohodnotená populácia je predaná ako vstup pre ďalšiu iteráciu evolučného algoritmu - generáciu.

Navrhnutý evolučný algoritmus zo svojho princípu a kódovania jedincov je možné zaradiť do triedy genetických algoritmov, konkrétnejšie do oblasti genetického programovania a optimalizácie softvéru. Na ukážke kódu 2 je podrobnejšie popísaný navrhnutý evolučný algoritmus. Jednotlivé kroky algoritmu budú bližšie vysvetlené v nasledujúcich kapitolách.

### 4.1.1 Kódovanie a mapovanie jedinca

Vhodná voľba kódovania jedinca je jednou z kľúčových vlastností evolučného algoritmu, ktoré ovplyvňujú rýchlosť evaluácie algoritmu či jeho úspešnosť. V gramatickej evolúcii sa často používa lineárne nepriame kódovanie, ktoré predstavuje vektor kladných hodnôt variabilnej dĺžky [29]. Grafická ukážka kódovania jedinca je na obrázku 4.2.

Mapovanie genotypu jedinca na fenotyp spočíva vo využití mapovacej funkcie, znázornenej rovnicou 4.1, ktorá postupne číta kodóny z genotypu, a následne aplikuje jednotlivé pravidlá gramatiky [38]. Jedno čítanie kodónu a aplikácia pravidla z gramatiky predstavuje jeden krok derivácie pri tvorbe derivačného stromu.




Obr. 4.2: Ukážka dekódovania genotypu jedinca na zodpovedajúci fenotyp. Vo vrchnej časti obrázku je znázornený genotyp jedinca. V prvom kroku si mapovacia funkcia na zásobník vloží štartovací NT[6] symbol `exp` a prečíta hodnotu zodpovedajúceho kodónu, ktorá je v našom prípade `6`. Následne sa pomocou mapovacej funkcie vyberie zodpovedajúce pravidlo z gramatiky (v spodnej časti vľavo). U `6 mod 3 = 0` vyberie sa teda `1.` možné pravidlo, na ktoré je možné derivovať a vloží sa na zásobník. Algoritmus vždy derivuje najľavejší NT symbol. Pokiaľ došlo k derivovaniu všetkých NT symbolov, algoritmus končí a reťazec na zásobníku zodpovedá fenotypu jedinca. Pokiaľ počet derivácií presiahol povolenú hranicu, mapovanie jedinca končí neúspešne a jedinec je označený za invalidného.

Vzhľadom k tomu, že som zvolil konštantnú dĺžku genómu jedincov, pri procese mapovania často dochádza k dvom okrajovým situáciám. Jednou z nich je stav, keď počet derivácií gramatiky je menší než je dĺžka genómu jedinca. Zjednodušene povedané algoritmus nevyužil všetky kodóny z genómu jedinca. Je nutné si uvedomiť, že nevyužitá časť genómu nie je síce efektívne využitá pri evaluácii jedinca, ale naďalej bude podliehať variačným operátorom genetického algoritmu. [33]

$$Uzol = Kodon[i] \ \boldsymbol{mod} \ pocet \ pravidiel \ NT \tag{4.1}$$

Druhá situácia nastáva v prípade, že počet derivácií dosiahol dĺžku genómu, ale vo vygenerovanej vete sa stále nachádzajú neterminálne symboly. V tomto prípade dôjde k znovupoužitiu genómu a derivácií ďalších pravidiel.

### 4.1.2 Inicializácia populácie

Podobne ako pri iných evolučných algoritmoch, GE[7] typicky generuje populáciu náhodne, čo často vedie k veľkému počtu invalidných jedincov, prípadne nerovnomernému prehľadávaniu stavového priestoru evolúcie. V rámci evolúcie bude použitá optimalizovaná metóda inicializácie populácie RHH[8]. [30]

Táto metóda spočíva v kombinácií dvoch základných techník inicializácie angl. *full* a angl. *grow*, kedy polovica jedincov je generovaná metódou *full* a druhá polovica je generovaná metódou *grow*. Jednou z kľúčových vlastností tejto metódy je možnosť zvoliť minimálnu a maximálnu hĺbku[9] derivačného stromu jedinca.

Princíp metódy *full* spočíva v tom, že zo štartovacieho neterminálneho symbolu sa postupne generuje syntaktický strom výberom pravidiel, ktoré generujú ďalšie neterminálne symboly. Po dosiahnutí maximálnej hĺbky stromu je derivácia možná iba pomocou pravidiel, generujúcich terminálne symboly. Metóda *full* generuje jedincov, ktorých hĺbka stromu je veľmi podobná k maximálnej hĺbke stromu.

Metóda *grow* je podobná ako metóda *full*, rozdiel spočíva v tom, že obmedzenie výberu pravidiel nastáva až po dosiahnutí maximálnej hĺbky stromu[10]. [36]

Cieľom inicializačnej metódy RHH je zaistiť dostatočnú diverzifikáciu počiatočnej populácie $P_0$, pretože metóda dokáže obmedziť zhora i zdola hĺbku derivačných stromov jednotlivých jedincov.

### 4.1.3 Mutácia jedinca

Bežne používané genetické variačné operátory v gramatickej evolúcií trpia efektom, ktorý sa v zahraničnej literatúre nazýva angl. *ripple effect* [7]. Tento efekt je zapríčinený hlavne z dôvodu pozičných vlastností kódovania a mapovania genotypu jedinca na fenotyp, kedy interpretácia kodónu a výber derivačného pravidla z množiny neterminálov závisí na aktuálne derivovanom neterminále.

Ohľadom mutácie a kríženia v gramatickej evolúcii existuje mnoho výskumov a diskusií, ktoré pojednávajú o pozičnej závislosti kodónu na genotype jedinca [5]. Rôzne techniky mutácie mali nejednoznačné výsledky v rámci celého spektra všeobecne známych problémov riešených v rámci evolučných algoritmov, a preto zvoliť tú správnu metódu nie je úplne jednoduché. Na druhú stranu, žiadna z bežne používaných techník nepriniesla výrazné zhoršenie dosiahnutých výsledkov evolúcie. [39]

V rámci práce bude použitá základná technika mutácie, kedy algoritmus iteratívene prejde celý genotyp jedinca a s pravdepodobnosťou $P_{mut}$, ktorá je rozprestretá na celú dĺžku genómu, aplikuje náhodnú zmenu kodónu. Táto technika je známa pod názvom angl. *int flip per codon*. Na obrázku 4.3 je znázornený proces mutácie jedinca.

### 4.1.4 Kríženie jedincov

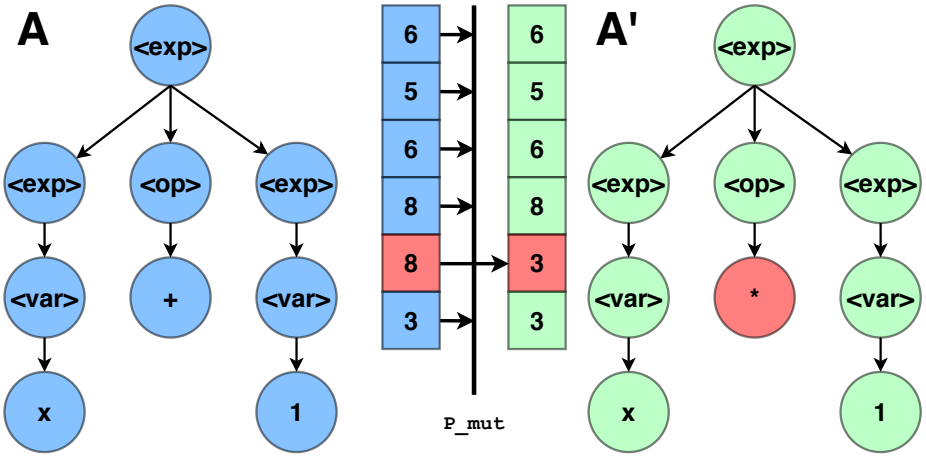
Proces kríženia začína výberom dvoch rodičov $p_1, p_2 \in P'$ pomocou turnajového algoritmu z aktuálnej populácie 3.1.3. Následne sa s pravdepodobnosťou $p_{cross}$ aplikuje na rodičov

[7] GE - gramatická evolúcia, angl. *gramatical evolution* – jedna z metód genetického programovania.

[8] RHH - nazývaná tiež angl. *ramped half-and-half*

[9] Hĺbkou derivačného stromu rozumieme počet hrán od koreňa k najvzdialenejšiemu listu.

[10] To znamená, že proces derivácie stromu nie je obmedzený iba na pravidlá generujúce neterminálne symboly.




Obr. 4.3: Na obrázku je znázornený proces mutácie jedinca A. Iteratívne prejdeme genotyp jedinca a na každý kodón aplikujeme s pravdepodobnosťou $p_{mut}$ mutáciu a vygenerujeme novú validnú hodnotu kodónu. Kodóny, na ktoré nieje aplikovaný operátor mutácie, sú znovupoužité v pôvodnej hodnote. Týmto procesom vznikne nový jedinec A'.

metóda kríženia. V prípade, že sa na rodičov neaplikuje operátor kríženia, postupujú títo rodičia v nezmenenej forme k ďalšiemu kroku evolúcie, ktorým je mutácia.

Ako variačný operátor kríženia je v rámci tejto práce použitá jednobodová variabilná metóda (angl. *variable one-point crossover*), ktorá využíva náhodne vygenerovaný index ako bod, ktorý určí delenie genómu jedinca. Pre oboch rodičov je tento index rovnaký, tak aby bola zachovaná konštantná dĺžka genómu jedincov. Výsledkom kríženia sú dvaja nové potomkovia $m_1$ a $m_2$, kde prvý potomok vznikne spojením prvej časti rodiča $p_1$ a druhej časti rodiča $p_2$. Obdobne druhý potomok $m_2$ vznikne spojením prvej časti rodiča $p_2$ a druhej časti rodiča $p_1$. Tento typ kríženia som zvolil hlavne z toho dôvodu, že na základe experimentov z dokumentu [34] toto kríženie často dosahovalo najlepšie výsledky. Ukážka variabilného jednobodového kríženia je znázornená na obrázku 4.4.

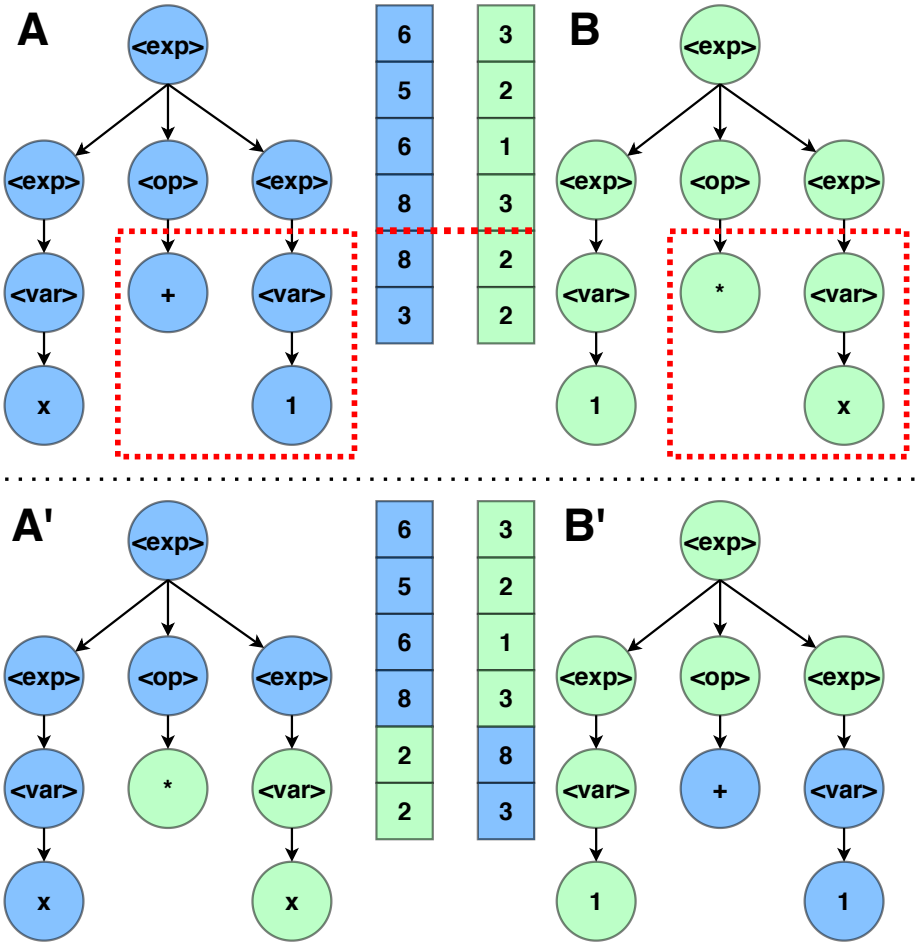
## 4.2 Evaluácia jedinca

Proces evaluácie jedinca začína dekódovaním genómu jedinca na fenotyp, ktorý je detailnejšie popísaný v kapitole 4.1.1.

Fenotyp jedinca predstavuje graf popísaný syntaxou jazyka, ktorý využíva nástroj Questa InFact. Tento graf je tvorený z užívateľom definovaných symbolov, ktoré predstavujú možné transakcie na vstupe verifikovanej komponenty. Príklad fenotypu jedinca je ukázaný na obrázku 4.5. Tieto transakcie sú v gramatike znázornené ako terminálne symboly. Jedna z vlastností nástroju InFact je, že nie je možné použiť ten istý symbol v generovanom grafe viacnásobne. Riešenie tohto problému spočíva v tom, že pokiaľ sa v grafe nachádza ten istý symbol niekoľkokrát, je mu priradený sufix, ktorý definuje i-ty výskyt symbolu v grafe.

Týmto procesom vznikne *sufix fenotyp*[11], z ktorého je vygenerovaný súbor **.rseg*, obsahujúci definíciu meta-akcií a deklaráciu jednotlivých symbolov pre každú použitú transakciu v *sufix fenotype*. Zároveň je *sufix fenotyp* vložený do hlavného cyklu nástroja InFact k symbolu *execute*. Ukážku zdrojových kódov je možné nájsť v podkapitole 5.6.

[11]Sufix fenotyp - je namapovaný fenotyp, kde jednotlivé symboly sú rozšírené o sufix í-tého výskytu symbolu vo fenotype.

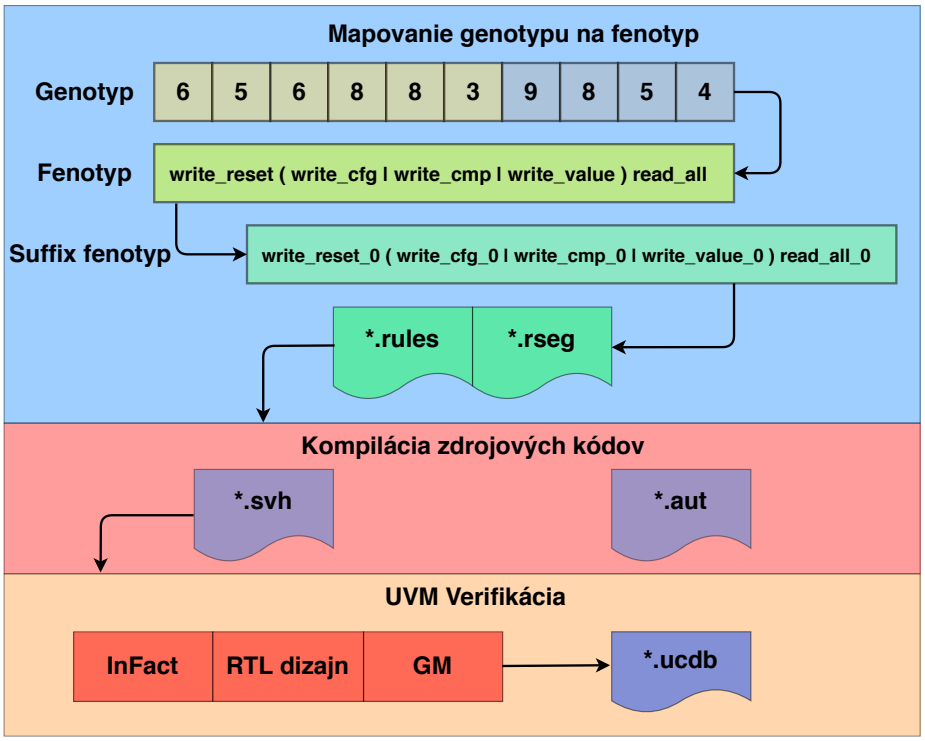



Obr. 4.4: Ukážka kríženia jedincov variabilnou jednobodovou metódou. V hornej časti obrázku je znázornený príklad dvoch jedincov *A, B*, ich genotyp a namapovaný fenotyp. Následne je náhodne zvolený bod kríženia znázornený červenou čiarou. Jedinci si v daných bodoch vymenia fenotyp, z čoho vzniknú nové jedinci *A', B'*, znázornené v spodnej časti obrázku.

Po vygenerovaní a upravení súborov **.rules* a **.rseg* je potrebné tieto zdrojové kódy skompilovať nástrojom InFact. Výstupom kompilácie je UVM komponent **.svh*, ktorý obsahuje riadenie a generovanie transakcií na základe definovaného grafu. Tieto transakcie sa postupne generujú pri verifikácii testovanej komponenty. Výstupom kompilácie je zároveň súbor **.aut*, ktorý predstavuje grafickú reprezentáciu grafu.

Po skompilovaní a vygenerovaní komponentu je potrebné tento komponent integrovať do UVM prostredia a zahájiť simuláciu verifikovaného komponentu. Integrácia vygenerovaných súborov do UVM prostredia je znázornená na obrázku 2.5 a konkrétne kroky, nutné k integrácii, sú rozoberané v podkapitole 5.5.

Výsledkom verifikácie je databáza **.ucdb*, ktorá mimo iné, obsahuje dosiahnuté štrukturálne i funkčné pokrytie. Na základe dosiahnutého funkčného a štrukturálneho pokrytia sa vypočíta fitness hodnota jedinca.




Obr. 4.5: Na obrázku je znázornený proces dekódovania jedinca z genotypu na fenotyp, z ktorého sú vytvorené súbory **.rules*, obsahujúci graf, a **.rseg*, obsahujúci užívateľom definované transakcie. Tieto súbory sú skompilované pomocou nástroja *Questa InFact*, z ktorých je vygenerovaný súbor **.svh* obsahujúci UVM komponent a **.aut* obsahujúci grafickú reprezentáciu grafu. Posledným krokom je spustenie simulácie s vygenerovaným riadiacim UVM komponentom a verifikovaným komponentom, ktorého výstup je databáza **.ucdb* obsahujúca dosiahnuté štrukturálne a funkčné pokrytie.

# Kapitola 5

# Implementácia a použitie frameworku

V tejto kapitole budú podrobne popísané zdroje a frameworky, ktoré boli použité v rámci implementácie navrhovaného riešenia pre automatickú verifikáciu hardwarových obvodov. Zároveň budú v kapitole vysvetlené dôležité prvky implementácie a postup konfigurácie frameworku pre použitie na iných testovacích obvodoch. V úvodnej kapitole 5.1 predstavím framework PonyGe2, ktorý použijem v rámci implementácie na zaistenie variačných operátorov pre gramatickú evolúciu. Následne popíšem základné parametre pre konfiguráciu evolučného algoritmu v podkapitole 5.2. Jednou z najdôležitejších podkapitol implementácie je Evaluácia jedinca 5.3, v ktorej podrobne vysvetlím celý proces výpočtu fitness hodnoty jedincov, s čím súvisí i posledná podkapitola 5.4, v ktorej predstavím spôsob tvorby fitness funkcie.

## 5.1 PonyGe2

I keď je gramatická evolúcia[1] relatívne mladý odbor, na poli evolučných algoritmov existuje niekoľko knižníc a frameworkov, ktoré umožňujú rýchlu a správnu implementáciu evolučného algoritmu založeného na gramatickej evolúcií. Jedným z nich je framework PonyGE2[2], ktorý predstavuje štartovací bod pre implementáciu gramatickej evolúcie v jazyku Python3. PonyGE2 je implementovaný ako open source framework [3], ktorý implementuje niektoré často používane techniky gramatickej evolúcie a tým uľahćuje riešenie nových problémov.

V rámci návrhu evolučného algoritmu som popísal niekoľko variačných operátorov, ktoré framework PonyGe2 úspešne implementuje, a práve preto som sa v rámci implementácie rozhodol zastrešiť variačné operátory evolučného algoritmu pod PonyGE2. V rámci tohto frameworku som využil implementáciu inicializácie populácie pomocou metódy RHH 4.1.2, kríženie jedincov pomocou metódy variabilného jednobodového kríženia 4.1.4 a mutáciu pomocou angl. *int flip per codon* 4.1.3.

[1] GE - gramatická evolúcia, angl. *gramatical evolution*
[2] https://github.com/PonyGE/PonyGE2
[3] Natural Computing Research and Applications group, UCD

## 5.2 Konfigurácia evolučného algoritmu

Nasledujúca kapitola čerpá prevažne z článku [14] a referenčného manuálu priloženého k frameworku[4]. Cieľom kapitoly je priblížiť podstatné konfiguračné parametre a ich dopad na beh evolučného algoritmu. Framework PonyGE2 obsahuje mnoho ďalších parametrov, ktoré je možné nájsť práve v spomenutých materiáloch.

Beh evolučného algoritmu je možné konfigurovať pomocou niekoľkých parametrov, ktoré výrazne ovplyvňujú spôsob manipulácie s jedincami a rýchlosť prehľadávania stavového priestoru. Práca s konfiguračnými parametrami je v frameworku PonyGE implementovaná pomocou slovníka `algorithm.parameters.params`, na ktorý sa odkazuje v celom programe, čo výrazne zefektívňuje a sprehľadňuje prácu s parametrami.

Všetky nižšie spomenuté parametre je možné zadávať ako parametre cez konzolu alebo pomocou konfiguračného súboru. Oba spôsoby je možné kombinovať v jednom behu programu, pričom najvyššiu prioritu pri konfliktných parametroch majú parametre zadávané cez príkazový riadok. Vzhľadom k tomu, že experimenty môžu obsahovať relatívne mnoho parametrov, je vhodnejšie používať konfiguráciu pomocou konfiguračných súborov.

Do konfiguračného súboru je možné jednoducho zadať i vlastné parametre vo formáte `PARAMETER_NAME: PARAMETER_VALUE` a následne ich používať v rámci implementácie.

### Paralelizácia výpočtu

Evaluácia jedincov je implicitne spúšťaná na jednom jadre sériovo, čo je vhodné pre nastavovanie parametrov experimentu alebo ladenie implementácie. Pokiaľ očakávame od evolučného algoritmu rozumné výsledky prehľadávania priestoru, je potrebná masívna paralelizácia evaluácie jedinca, ktorá je častokrát tou výpočtovo najzložitejšou časťou evolučného algoritmu. [4]

Paralelizácia výpočtu je implementovaná pomocou zdieľanej fronty, kde hlavné vlákno postupne vkladá jedincov pripravených na evaluáciu do danej zdieľanej fronty. Ostatné výpočtové vlákna čakajú na vloženie jedinca do zdieľanej fronty. Pokiaľ je do fronty vložený jedinec, jedno z vlákien si rezervuje jedinca z fronty a spustí jeho evaluáciu. Paralelizáciu je možné nastaviť pomocou parametru `--multicore` a pomocou prepínača `--cores [INT]` počet jadier, ktoré sa budú na evaluácii jedincov podieľať.

### Variačné operátory evolučného algoritmu

Pri implementácií evolučného algoritmu som využil niekoľko funkcií frameworku PonyGE2, ktoré implementujú variačné operátory pre gramatickú evolúciu. Prevzatá je implementácia inicializácie populácie metódou RHH, predstavenej v podkapitole 4.1.2, ktorá sa nastavuje parametrom `--initialisation rhh`. Pri využití metódy RHH je možné nastaviť minimálnu a maximálnu hĺbku generovaných jedincov v populácií $P_0$.

Prevzatá je i implementácia metódy kríženia pomocou variabilného jednobodového kríženia (viď podkapitola 4.1.4) a mutácie (vid podkapitola 4.1.3). Kríženie jedincov je možné nastaviť pomocou parametra `--crossover variable_onepoint` a mutáciu pomocou parametra `--mutation int_flip_per_codon`.

Najpodstatnejší prepínač je `--fitness_function [OBJ]`, ktorý nastavuje evaluačnú fitness funkciu jedincov, podľa ktorej sa bude vypočítavať ich fitness hodnota. Názov fitness funkcie musí byť totožný s názvom triedy, ktorá implementuje evaluáciu, a názvom súboru,

[4] https://github.com/PonyGE/PonyGE2/wiki

v ktorom je daná trieda umiestnená. Podrobný popis implementácie evaluácie jedinca je v podkapitole 5.3.

Elitizmus je možné zapnúť pomocou prepínača `--elite_size [INT]`, ktorý pomáha pri prechode na novú generáciu nestrácať doposiaľ najlepšieho jedinca.

### Základne parametre EA

Medzi základné parametre evolučného algoritmu patrí veľkosť populácie, ktorú je možné konfigurovať pomocou prepínača `--population_size [INT]`. Tento parameter udáva počet jedincov vygenerovaných v každej jednej generácii. V genetických algoritmoch sa typicky používa väčšia veľkosť populácie, ktorá môže dopomôcť k lepším výsledkom, ale zároveň zvyšuje výpočtovú náročnosť evolučného algoritmu. [9]

Hlavný cyklus evolučného algoritmu ma nastavenú pevnú ukončovaciu podmienku v podobe počtu generácií, ktorú je možné nastaviť pomocou prepínača `--generations [INT]`. Po dosiahnutí konečného počtu generácií sú na výstup vypísané štatistiky o behu evolučného algoritmu spolu s najlepším jedincom.

Parametrom `MAX_HISTORY: [INT]` môžeme nastaviť maximálnu dĺžku histórie porovnávania zlepšenia najlepšieho jedinca. V prípade, že evolučný algoritmus stagnuje a už dlhšiu dobu nevygeneroval lepšieho jedinca je predčasne ukončený.

### Optimalizácia jedincov

Počas kríženia a mutácie jedincov medzi generáciami sa vyskytuje častý jav, kedy nastáva veľké zvýšenie genotypu jedincov bez toho, aby sa významne zvýšila ich fitness hodnota, tento jav sa nazýva angl. *bloat* [11]. Jav je možné čiastočne obmedziť pomocou niekoľkých techník, bohužiaľ ale nejde ho úplne odstrániť. V rámci implementácie evolučného algoritmu sú použité obmedzenia maximálnej hĺbky stromu jedinca pomocou prepínača `--max_tree_depth [INT]`, kedy jedinci, ktorých derivačný strom presiahne zvolenú hĺbku, sú označené ako invalidní.

Podstatná optimalizácia behu evolučného algoritmu spočíva v ukladaní fenotypu evaluovaných jedincov do slovníkovej štruktúry a v prípade, že počas generácie narazíme na jedinca s rovnakým fenotypom, priradíme mu uloženú fitness hodnotu. Táto technika je známa pod pojmom angl. *caching*. Použitie tejto techniky je vhodné práve z dôvodu vlastností mapovania angl. *many-to-one*[5](vid' podkapitola 4.1.1). Túto optimalizáciu je možné zapnúť pomocou prepínača `--cache [BOOL]`.

### Nastavenie experimentu

Konfigurácia jednotlivých experimentov nezasahuje priamo do nastavení behu evolučného algoritmu, ale nastavuje korektne vstupy, výstupy a prípadne inicializáciu generátoru náhodných čísel. Pre každý experiment je potrebné nastaviť vstupnú gramatiku vo formáte BNF[6] pomocou parametru `--grammar_file [path_to_file]`. Názov experimentu pre lepšiu identifikáciu výstupných súborov, ktoré budú po spustení umiestnené v priečinku s názvom `results/[nazov_expeimentu]` je možné nastaviť prepínačom `--experiment_name`.

Medzi neštandardné parametre frameworku patrí `INFACT_META_ACTION: [DICT]`, ktorý predstavuje zoznam základných transakcií grafu, a parameter `INFACT_ITERATION: [INT]` udávajúci počet vygenerovaných transakcií.

[5] *many-to-one* - je mapovanie, kedy sa viacero genotypov môže mapovať na jeden fenotyp.
[6] BNF - Backusova–Naurova forma je spôsob zápisu bezkontextových gramatík.

## 5.3 Evaluácia jedinca

Framework PonyGE2 umožňuje pomerne jednoducho a efektívne pridať vlastnú funkciu pre ohodnotenie jedinca, respektíve výpočet jeho fitness hodnoty. V tomto prípade bolo potrebné implementovať triedu `fitness.infact_ff`, ktorej implementácia sa nachádza v rovnako pomenovanom súbore `infact_ff.py`. Je dôležité, aby názov triedy fitness funkcie a súboru, kde je funkcia implementovaná, boli totožné[7]. V prípade, že chceme použiť pre evaluáciu jedinca implementovanú fitness funkciu, je potrebné referencovať na túto triedu pomocou parametru `--fitness_function`.

V rámci implementácie triedy `fitness.infact_ff` je potrebné implementovať povinnú metódu `evaluate(self, ind)`, ktorú framework využíva v cykle evolučného algoritmu na evaluáciu jedincov. Výstupom tejto metódy je kvantifikované ohodnotenie jedinca - fitness hodnota. Implementácia metódy `evaluate(...)` je znázornená na ukážke kódu 5.1. V nasledujúcej časti vysvetlím a popíšem jednotlivé metódy, z ktorých je implementovaná spomenutá fitness funkcia.

```
1  def evaluate(self, ind):
2      thread_id = 'thread_' + str(re.findall(r'\d+', str(mp.current_process()))[0])
3      self.createRsegFile(ind, thread_id)          // sbor structs.rseg
4      self.generateFile(ind, thread_id)            // sbor test_component.rules
5      self.compileGraph(thread_id)                 // kompilcia vygenerovanho grafu
6      self.setIterationLimit(params['INFACT_ITERATION'], thread_id)
7      self.runVerification(thread_id)              // spustenie verifikanho prostredia
8      self.fitness(ind, thread_id)                 // extrakcia dosiahnutho pokrytia
9      self.removeWork(thread_id)
10     return ind.fitness                           // fitness hodnota jedinca
```

Výpis 5.1: Ukážka hlavného cyklu evaluácie jedinca. V prvom kroku vygenerujeme `*.rseg` a `*.rule` súbory, ktoré následne skompilujeme pomocou nástroja InFact. Upravíme vygenerovaný UVM komponent `test_component.svh`, ktorý použijeme na generovanie transakcií. Následne spustíme simuláciu a na základe dosiahnutého štrukturálneho a funkčného pokrytia (viď podkapitola Funkčné pokrytie 5.4.2) vypočítame fitness hodnotu jedinca.

Evaluáciu jedincov je možné spustiť paralelne pre niekoľko jedincov (vid podkapitola 5.2) a preto je potrebné zaistiť unikátne pracovné priestory pre jednotlivé vlákna. Paralelizácia je zaistená pomocou spustenia niekoľkých pracovných vlákien, ktoré čakajú v zdieľanej fronte na zadanú prácu vo forme vloženého jedinca. Unikátne cesty pre pracovné prostredie jednotlivých vlákien sú generované na základe identifikácie vlákna, ktorá je uložená v premennej `thread_id`. Ako je možné vidieť na ukážke kódu 5.1, identifikácia vlákna je následne predávaná na vstup metódam, ktoré podľa ID vlákna upravujú svoje prístupové cesty.

### 5.3.1 Generovanie súborov pre nástroj InFact

Vstup metódy `evaluate(...)` je objekt, jedinec, ktorý má podľa zadanej gramatiky a genómu vygenerovaný zodpovedajúci fenotyp vo forme grafu. Objekt jedinca je inštanciovaný z triedy `representation.individual` a práve jeden z jeho atribútov `ind.phenotype` je textová reprezentácia fenotypu jedinca. Tento fenotyp je potrebné ešte upraviť pridaním

[7] Názov triedy a súboru implementácie fitness funkcie je použitý vo vstupnom parametri `--fitness_function [OBJ]`.

*sufixov* k metódam určujúcim i-tý výskyt meta-akcie vo fenotype, tak ako bolo spomenuté v podkapitole návrhu 4.2. Výsledkom tejto úpravy je fenotyp, ktorý neobsahuje syntakticky rovnaké symboly a slovník `params['INFACT_META_ACTION']`, ktorý obsahuje dvojicu symbol a počet výskytov symbolu vo fenotype. Na základe tohto slovníka sa generujú v súbore `structs.rseg` príslušné symboly predstavujúce transakcie a to vo formáte znázornenom v kóde 5.2.

```
'timer_' + meta_action + '_transaction' + meta_action + '_' + i + '_struc;'
'symbol ' + meta_action + '_' + it + ' = do_item(' + meta_action + '_' + i +'_struc);'
```

Výpis 5.2: Ukážka implementácie generovania symbolov z meta-akcií, ktoré budú predstavovať generované transakcie.

Po vygenerovaní hlavného súboru `structs.rseg` je následne vygenerovaný i súbor `test_component.rules`, kedy sa do predpripraveného súboru vloží upravený fenotyp jedinca na miesto symbolu `execute`, presne ako je popísané v podkapitole 4.2. Po tomto kroku sú pripravené všetky potrebné súbory na preklad a kompiláciu grafu pomocou nástroja InFact.

### 5.3.2 Kompilácia vygenerovaného grafu

Kompilácia grafu je implementovaná v metóde `CompileGraph(...)`, ktorá pomocou modulu `genproject` preloží príslušné vygenerované súbory, skompiluje graf a vygeneruje súbory, ktoré je možné použiť ako sekvenciu vo verifikačnej simulácii pomocou nástroju Questa. Pre nás je podstatný vygenerovaný súbor `test_component.svh`, ktorý obsahuje obálku pre UVM sekvenciu a triedu `test_component`, ktorá rozširuje triedu `test_component_base_t`. Táto trieda obsahuje riadenie a generovanie sekvencií grafu.

Generovanie jednotlivých sekvencií prebieha v cykle, pre ktorý je nutné definovať počet iterácií, a to v súbore `test_component.svh`. Táto funkcionalita je implementovaná v metóde `setIterationLimit(...)`, ktorá nastaví premennú `m_iterationLimit` na požadovaný počet iterácií.

### 5.3.3 Simulácia a dosiahnuté pokrytie

Evaluácia jedinca spočíva v spustení simulácie, kde generovanie stimulov je riadené vygenerovaným komponentom `test_component.svh` nástrojom InFakt. Počas simulácie sú zaznamenávané reakcie na vstupné impulzy a porovnávané z reakciami GM[8]. Spustenie a riadenie simulácie implementuje metóda `runVerification(...)`, ktorá je znázornená na ukážke kódu 5.3. Pomocou príkazu v ukážke kódu 5.3 je spustená simulácia bez grafického rozhrania.

```
bashCommand = 'vsim -c -do start_cmd.tcl'
process = subprocess.Popen(bashCommand.split(), stdout=subprocess.PIPE, stderr=subprocess.
    PIPE, cwd=os.path.expanduser(PATH))
output, error = process.communicate()
```

Výpis 5.3: Príklad spustenia simulátora v režime bez GUI módu pomocou skriptu `start_cmd.tcl`.

[8] GM - Golden model

Počas simulácie sa zaznamenáva dosiahnuté funkčné a štrukturálne pokrytie kódu, ktoré sa ukladá do databázy `coverage.ucdb`. Pokrytie kódu je generované automaticky nástrojom simulácie Questa, funkčne pokrytie je definované užívateľom a významne ovplyvňuje fitness hodnotu jedinca. Definícia funkčného pokrytia je znázornená v podkapitole 5.4.2. Fitness hodnota jedinca je vypočítaná ako absolútne dosiahnuté pokrytie jedinca počas simulácie, zahŕňa v sebe i pokrytie kódu i funkčné pokrytie.

Posledným krokom metódy `evaluate(...)` je vymazanie pracovného priestoru simulácie. Ktoré síce vo väčšine prípadov nie je nutné, ale niekedy sa počas behu evolučného algoritmu stávalo, že nebolo možné pustiť novú simuláciu v starom prostredí.

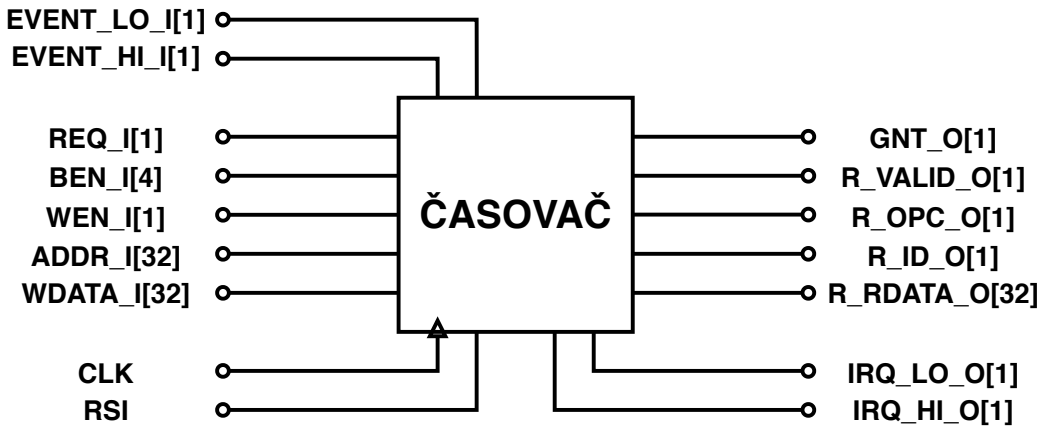
## 5.4 Fitness funkcia

Fitness hodnota jedinca je vypočítaná ako absolútne dosiahnuté pokrytie verifikovaného komponentu jedinca počas simulácie, zahŕňa v sebe pokrytie kódu a funkčné pokrytie. To znamená, že fitness funkcia je vztiahnutá k verifikovanému komponentu. Štrukturálne pokrytie je automaticky generované simulačným nástrojom a patrí sem pokrytie skokov (angl. *branches*), podmienok (angl. *conditions*), direktiv (*directives*), výrazov (angl. *expressions*), stavov automatu (angl. *FSM states*), prechodov automatu (angl. *FSM transitions*) a príkazov (angl. *statements*). Funkčné pokrytie (angl. *covergroups*) je definované verifikačným inžinierom 5.4.2. Celkové pokrytie verifikovaného komponentu je vypočítané ako priemer spomenutých pokrytí.

V nasledujúcich podkapitolách 5.4.1 podrobne popíšem funkcionalitu verifikovaného komponentu časovača a tvorbu funkčného pokrytia pre daný komponent v podkapitole 5.4.2.

Kapitola 5.4.1 čerpá informácie prevažne z dokumentácie priloženej k časovaču [37] a štúdiou zdrojových kódov[9].

### 5.4.1 Verifikovaný komponent - Časovač




Obr. 5.1: Bloková schéma časovača. Podrobná funkcionalita jednotlivých vstupov a výstupov je popísaná v kapitole 5.4.1.

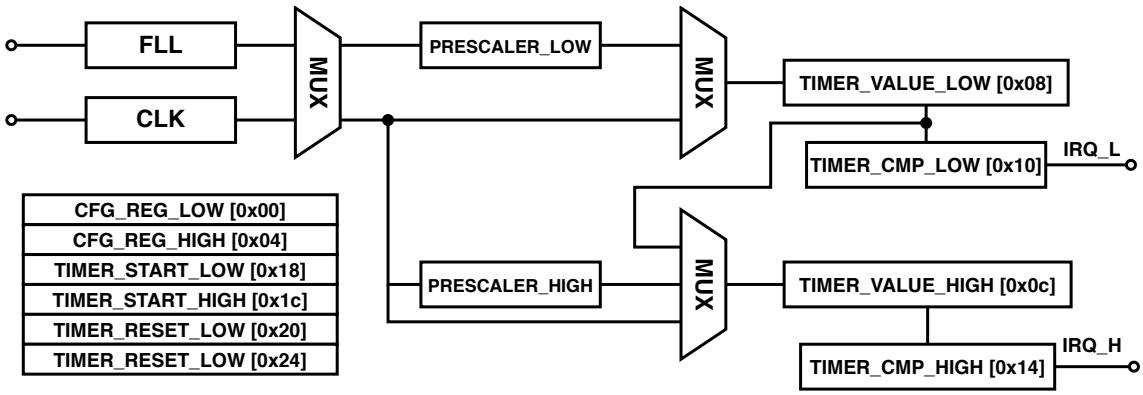
[9] https://github.com/pulp-platform/apb_timer

Verifikovaný komponent je prebraný z projektu PULP[10], kde plní funkcionalitu časovača. Je zložený z dvoch samostatných 32-bitový čítačov, ktoré je možné zapojiť do kaskády (viď obrázok 5.2). Je teda možné používať dva 32-bitové čítače, ktoré bežia nezávisle na sebe alebo jeden 64-bitový čítač, ktorý vznikne vhodnou konfiguráciou vnútorných registrov časovača. 64-bitový mód časovača je možné nastaviť pomocou hlavného konfiguračného registra `CFG_REG_LOW[31]` do logickej `1`, inak je časovač v režime dvoch 32-bitových časovačov.

Ako zdroj hodinového signálu pre časovač je možné zvoliť hodinové impulzy generované FLL[11] obvodom alebo impulzy generované referenčným zdrojom s frekvenciou `32KHz`. Zdroj budiaceho signálu je možné nastaviť pomocou bitu `CFG_REG_LOW[7]`, pričom logická `1` predstavuje použitie 32KHz referenčného zdroja hodín a logická `0` nastavuje ako zdroj FLL obvod. Súčasťou každého 32bitového časovača je taktiež nastaviteľná preddelička, ktorá umožňuje variabilne nastaviť požadovanú budiacu frekvenciu časovača. Preddelička referenčného signálu je zapojená až za multiprexorom vyberajúcim zdroj budiaceho signálu, čo umožňuje ovplyvňovať oba zdroje budiaceho signálu. Hodnotu preddeličky je možné nastaviť pomocou konfiguračných registrov `CFG_REG_LOW[15..8]` a `CFG_REG_HIGH[15..8]`, kde hodnota preddeličky je vypočítaná pomocou rovnice 5.1. Preddeličku je možné aktivovať pomocou `7.` bitu konfiguračných registrov.

$$FREQ = 1/(1 + prescaleVal) \tag{5.1}$$

Spustenie činnosti časovača je možné vykonať tromi spôsobmi. Jednou z možností je zápis logickej `1` do bitu `CFG_REG_LOW[0]` alebo do registra `TIMER_START_LOW`. Posledná možnosť ako spustiť časovač je privedením logickej `1` na vstupný signál `PEVENT_LO_I` a nastavením `CFG_REG_LOW[3]` do logickej `1`, respektíve `EVENT_HI_I` a `CFG_REG_HIGH[3]` pre druhý časovač.




Obr. 5.2: Vnútorná schéma časovača, ktorý sa skladá z dvoch 32-bitových časovačov, registrového konfiguračného poľa, dvoch preddeličiek zdroja hodinového signálu FLL alebo CLK.

Hlavná funkcionalita časovača je meranie zvoleného časového intervalu, kde požadovaná meraná hodnota je nastavená do komparačného registra `TIMER_CMP_LOW`, prípadne `TIMER_CMP_HIGH`. Keď časovač dosiahne cieľovú hodnotu a je povolené generovanie prerušenia pre príslušný časovač pomocou konfiguračného registra `CFG_REG_LOW[2]`, respek-

[10] https://pulp-platform.org/

[11] FLL - angl. *frequency-locked loop* - obvod, ktorý dynamicky upravuje svoju frekvenciu podľa referenčnej.

tíve CFG_REG_HIGH[2], časovač vygeneruje logickú 1 na výstupný signál IRQ_LO, prípadne IRQ_HI. Následné správanie časovača je možné ovplyvniť dvomi režimami.

**One shot** režim znamená, že po dosiahnutí cieľovej hodnoty počítanie končí a časovač prejde do stavu idle. V prípade, že je one shot režim vypnutý, počítanie časovaču pokračuje ďalej. Režim one shot je možné nastaviť zapísaním logickej 1 do konfiguračného registra CFG_REG_LOW[5], respektíve CFG_REG_HIGH[5].

**Cmp&Clr** je možné nastaviť zapísaním logickej 1 do registra CFG_REG_LOW[4], prípadne CFG_REG_HIGH[4]. Pri nastavení tohto režimu, časovač pri dosiahnutí cieľovej hodnoty vynuluje registre TIMER_VALUE_LOW a TIMER_VALUE_HIGH uchovávajúce aktuálnu hodnotu počítania. Pri vypnutí režimu one shot a nastavení režimu CMP&CLR je možné periodicky merať daný časový úsek a generovať prerušenie.

Vynulovať časovač je možné zapísaním logickej 1 do registra CFG_REG_LOW[1], prípadne CFG_REG_HIGH[1], kedy po zapísaní požadovanej hodnoty sú vnútri časovača vynulované registre uchovávajúce hodnotu čítača. Časovač je možné vynulovať taktiež zapísaním logickej 1 do registra TIMER_RESET_LOW, prípadne TIMER_RESET_HIGH. Posledná z možností, kedy dôjde k vynulovaniu časovača je prípad, kedy je v konfiguračnom registri nastavený mód CMP&CLR a v čítači dôjde k dosiahnutiu cieľovej hodnoty.

Na obrázku 5.1 je znázornená bloková schéma čítača, ktorý má niekoľko vstupných a výstupných portov. Vstupný signál REQ_I slúži na overenie platnosti vstupných signálov WDATA_I, ADDR_I, WEN_I a BEN_I. Signál WEN_I slúži na identifikáciu zápisu pri logickej 1, alebo čítania pri logickej 0. Signál ADDR_I udáva adresu cieľového registra. Pri požiadavke na zápis do čítača sú dáta vystavené na signál WEN_I. Signály EVENT_LO_I a EVENT_HI_I slúžia na detekciu externé udalosti, ktorá môže spustiť počítanie časovača.

Výstupné signály R_VALID_O, R_ID_O a R_OPC_O slúžia na identifikáciu a validáciu požiadavky vystavené na vstupné porty časovača. Port R_DATA_O predstavuje požadované dáta z niektorého registra časovača. Signály IRQ_LO_O a IRQ_HI_O sú aktivované pokiaľ je povolené generovanie prerušenia príslušným konfiguračným registrom a hodnota čítača dosiahla požadovanú hodnotu, uloženú v registroch TIMER_CMP_LOW alebo TIMER_CMP_HIGH.

### 5.4.2 Funkčné pokrytie

Funkčné pokrytie verifikovaného komponentu časovača 5.4.1 je implementované v súbore *coverage.svh* a je rozdelené do dvoch skupín (angl. *covergroup*).

Prvá skupina FunctionalCoverage definuje funkčné pokrytie pre vstupné porty časovača pomocou bodov pokrytia (angl. *coverpoints*). Na výpise 5.4 je ukážka definície bodu pokrytia pre vstupný port addr_i[32] (viď obrázok 5.1 alebo podkapitolu 5.4.1), ktorý kontroluje dosiahnutie všetkých validných hodnôt adries vnútorných registrov.

```
1  c_addr: coverpoint m_transaction_h.addr_i{
2      bins cfg_low    = {CFG_REG_LOW};
3      bins cfg_high   = {CFG_REG_HIGH};
4      bins value_low  = {TIMER_VALUE_LOW};
5      bins value_high = {TIMER_VALUE_HIGH};
6      bins cmp_low    = {TIMER_CMP_LOW};
7      bins cmp_high   = {TIMER_CMP_HIGH};
8      bins start_low  = {TIMER_START_LOW};
9      ...
10 }
```

Výpis 5.4: Definícia bodu pokrytia pre adresu registrov. Cieľom je kontrola, či transakcie počas verifikácie pristupovali ku všetkým registrom.

Pri kontrole dosiahnutia rôznych konfigurácií som využil operátor * (angl. `wildcard`). Vo výpise kódu 5.5 je ukážka definície bodu pokrytia pre 32-bitový alebo 64-bitový režim časovaču. Operátor wildcard umožňuje zameranie sa iba na určité časti vstupného portu ktorý chceme monitorovať. Ostatné časti môžeme definovať pomocou operátoru `'?'` ako nezaujímavé.

```
c_cfg_low_mode: coverpoint ivif.s_cfg_lo_reg{
    wildcard bins mode_64b      = {32'b0??????????????????????????????};
    wildcard bins mode_32b      = {32'b1??????????????????????????????};
}
// modes and start
c_cfg_low_conf_mode_start: cross c_cfg_low_mode, c_cfg_low_start;
```

Výpis 5.5: Definícia bodu pokrytia pomocou operátoru `wildcard`. V príklade nastavíme biny na konfiguráciu 31 bitu ktorý určuje mód časovaču.5.4.1

```
c_start_timer: cross c_wdata_reset, c_event_lo, c_wdata_iem{
  bins start_event_low =  binsof(c_event_lo.event_lo_high) &&
                          binsof(c_wdata_iem.en) &&
                          binsof(c_wdata_reset.dis);

    bins start_cfg =        binsof(c_wen_i.write) &&
                            binsof(c_addr.cfg_low) &&
                            binsof(c_cfg_wdata_enable.en) &&
                            binsof(c_cfg_wdata_reset.dis);
    ... // definicia dalsich cross coverpoint
}
```

Výpis 5.6: Definícia krížového bodu pokrytia ktorý kontroluje, či došlo k spusteniu časovaču pomocou signálu `event_lo_i`.

```
ccc_c_flow: coverpoint ivif.s_timer_val_lo{
    bins zero2one = (0=>1);
    bins count6 = (0=>1=>2=>3);
    bins maxminone2max = (32'hFFFF_FFFE=>32'hFFFF_FFFF);
    bins wraparound  = (32'hFFFF_FFFF=>32'h0000_0000);
    option.at_least = 2;
}
```

Výpis 5.7: Definícia bodu pokrytia pre adresu registrov. Cieľom je kontrola, či sa počas verifikácie pristupovalo ku všetkým registrom.

### 5.4.3 Definícia kontrolných bodov

Kontrolné body angl. *assertion* (viď podkapitola 2.1.4) sú použité na kontrolu zložitejších situácií, ktoré by sa tažko definovali vo funkčnom pokrytí.

```
  property pr_valid;
    @(posedge CLK) $rose(req_i) |=> $rose(r_valid_o);
  endproperty

  a_valid: assert property (pr_valid);
  c_valid: cover property (pr_valid);
```

Výpis 5.8: Definícia kontrolného bodu, ktorý monitoruje situáciu, keď počas simulácie dôjde k nastaveniu signálu `req_i` očakávame nasledujúci takt nastavený signál `r_valid_o`

## 5.5 Nastavenie a integrácia verifikačného prostredia

V rámci práce mi bolo od vedúcej práce poskytnuté základné verifikačné prostredie nad UVM, ktoré bolo potrebné upraviť tak, aby bolo možné využívať vygenerovaný verifikačný komponent z nástroja Questa InFact. Tento komponent je napojený na verifikačné prostredie pomocou sekvencií, ktoré sú následne distribuované do verifikovaného dizajnu a zároveň do kontrolného GM. Napojenie komponentu je znázornené na obrázku 2.5.

Pri integrácií komponentu do verifikačného prostredia som musel upraviť niekoľko súborov.

**start_common.tcl** - v tomto súbore bolo potrebné pridať správnu cestu (vid' výpis 5.9) k inicializačnému súboru komponentu nástroja InFact, ktorý sa nachádza v jeho pracovnom priečinku. Zároveň bolo potrebné definovať názov pre ukladanie databázy **.ucdb* a povoliť použitie InFact transakcií.

```
quietly set INFACT_PROJECT_INI "./infact_timer_component.ini"
quietly set INFACT_ENABLED 1
quietly set COVERAGE_FILE "coverage"
```

Výpis 5.9: Úprava súboru `start_common.tcl`

**start_cmd.tcl** - v skripte, ktorý spúšťa samotnú verifikáciu komponentu, bolo potrebné pridať načítanie inicializačného súboru `INFACT_PROJECT_INI` a zapnúť generovanie pokrytia po skončení simulácie do súboru `coverage.ucdb` (viď výpis 5.10).

```
quietly append VSIM_RUN_CMD " +infact=${INFACT_PROJECT_INI}"
coverage save -onexit ${COVERAGE_FILE}.ucdb
```

Výpis 5.10: Úprava súboru `start_cmd.tcl`

**sv_agent_pkg.sv** - do tohto súboru bolo potrebné vložiť cestu ku vygenerovanému komponentu `test_component.svh` z nástroja InFakt, ktorá riadi a generuje sekvencie pre verifikovaný komponent (viď výpis 5.11).

```
`include "../infact_work/infact_timer_component/test_component/test_component.svh"
```

Výpis 5.11: Úprava súboru `sv_agent_pkg.sv`

**test.svh** - súbor (viď výpis 5.12) obsahuje triedu `timer_infact_test`, ktorá rozširuje triedu `timer_t_test_base`. V implementácií tejto triedy sa nachádza jednotné generovanie sekvencií pre DUT[12] a GM[13], ktoré vytvára sekvencie prevzaté z komponentu vygenerovaného nástrojom InFact `test_component.svh`.

```
// sekvencie pre DUT
seq = timer_t_sequence_reset::type_id::create( "dut_reset" );
seq.start( m_env_h.m_timer_t_agent_h.m_sequencer_h );
seq = test_component::type_id::create( "dut_infact_seq" );
seq.start( m_env_h.m_timer_t_agent_h.m_sequencer_h );

// sekvencie pre GM
seq = timer_t_sequence_reset::type_id::create("gold_reset");
seq.start( m_env_h.m_gold_h.m_sequencer_h );
seq = test_component::type_id::create("dut_infact_seq");
seq.start( m_env_h.m_gold_h.m_sequencer_h );
```

Výpis 5.12: Úprava súboru `test.svh` pridaním sekvencií pre DUT a GM.

[12]DUT - angl. *design under test*, predstavuje verifikovaný komponent.
[13]GM - angl. *golden model*, predstavuje referenčný model k DUT modelu.

## 5.6 Štruktúra prostredia pre nástroj Questa InFact

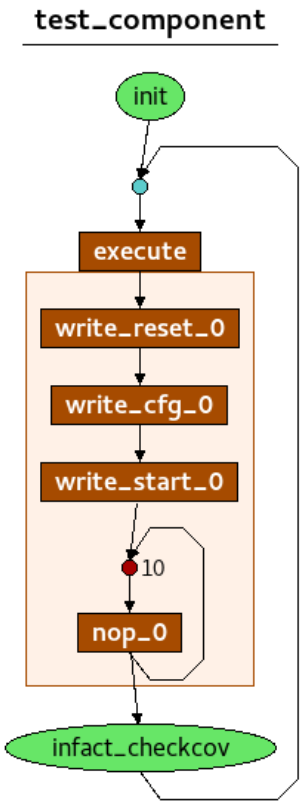
### 5.6.1 Štruktúra grafu

Hlavný súbor prostredia Questa InFact a zároveň testovacieho komponentu, z ktorého sa následne kompiluje graf, je `test_component.rules`. Tento súbor obsahuje inicializačnú akciu `init`, ktorá nakonfiguruje potrebné parametre riadiaceho cyklu testovacieho komponentu. Zároveň sa tu nachádza import súboru `struct.rseg`, ktorý je znázornený na výpise 5.14, 5.15 a 5.16.

Vo výpise 5.13 sa nachádza hlavný testovací cyklus, ktorý v sebe obsahuje symbol `execute`. Telo symbolu `execute` predstavuje mapovaný fenotyp jedinca, ktorý v sebe obsahuje popis grafu. Evolučný algoritmus pomocou metódy `generateFile(...)` na toto miesto vygeneruje spomenutý fenotyp jedinca.

Užívateľ si môže tento graf ľubovoľne upraviť podľa potrieb a parametrov, ktoré mu vyhovujú. V ukážke grafu 5.3 a výpise 5.13 je situácia, kedy symbol `execute` obsahuje celý graf. V tomto prípade evolúcia hľadá kompletný graf, ktorý je zložený z užívateľom definovaných transakcií. Ďalšia z možností využitia tejto štruktúry je situácia, kedy si užívateľ definuje graf do ktorého vloží symbol `execute`. V tomto prípade evolučný algoritmus hľadá optimálny podgraf, zložený z užívateľom definovaných transakcií.

Rozšírenie frameworku bolo navrhnuté s cieľom širokého použitia. V rámci experimentov som použil evolučný algoritmus na hľadanie optimálneho grafu.




Obr. 5.3: Grafická reprezentácia grafu z výpisu 5.13 `*.aut`.

```
1  rule_graph test_component {
2    action init;                // registrcia akci
3    import "structs.rseg";      // vloenie sboru s~defincami transakcii
4
5    // vygenerovan fenotyp jedinca
6    symbol execute = write_reset_0 write_cfg_0 write_start_0 repeat 10 { nop_0 };
7
8    test_component = init repeat {
9        execute                 // vykonvanie tela execute
10   };
11 }
```

Výpis 5.13: Hlavný súbor prostredia InFact `test_component.rules`, ktorý obsahuje vygenerovaný fenotyp jedinca v podobe popisu grafu. Tento popis grafu je vložený do tela symbolu `execute`.

### 5.6.2 Štruktúra transakcií

Vygenerovaný graf je zložený zo základných transakcií, ktoré definuje užívateľ. V kapitole 6.3.1 je ukážka tvorby takýchto transakcií. Definícia transakcií sa nachádza v súbore `structs.rseg` a je tvorená hierarchickým stromom.

Koreňová transakcia `timer_t_transaction` je najobecnejšia a generuje všetky validné hodnoty, ktoré môžu byť vložené na rozhranie testovacieho komponentu. Smerom k listom sú transakcie presnejšie a množina vygenerovaných dát je postupne zmenšovaná pomocou obmedzení (viď výpis 5.15). Na výpise 5.14 je ukážka definície koreňovej transakcie, ktorá pozostáva z meta-akcií[14].

```
struct timer_t_transaction {
    meta_action rst_ni [unsigned 0,1];      // definovanie mnoiny validnch hodnot
    meta_action req_i [unsigned 0,1];
    meta_action addr_i [unsigned 31:0];
    timer_t_transaction = rst_ni req_i addr_i ...;
}
```

Výpis 5.14: Ukážka koreňovej štruktúry, z ktorej dedia ostatné štruktúry.

Na výpise 5.15 je znázornená transakcia `timer_write_all_transaction`, definovaná pomocou štruktúry, ktorá rozširuje štruktúru koreňovej transakcie `timer_t_transaction`. Týmto zápisom štruktúra `timer_write_all_transaction` dedí všetky meta-akcie, obmedzenia a vnútorné symboly, ktoré boli definované v koreňovej štruktúre. Pri implementácii štruktúry `timer_write_all_transaction` je využité obmedzenie `constraint write_c`, ktoré upresňuje možnú množinu generovaných dát pre meta-akcie.

```
struct timer_write_all_transaction extends timer_t_transaction {
  constraint write_c {
      req_i == 1;
      wen_i == 0;
      event_lo_i == 0;
      event_hi_i == 0;
} }
// priradenie typu timer_t_transaction strukture timer_write_all_transakcion
attributes timer_write_all_transaction {
  type = "timer_t_transaction";
}
```

Výpis 5.15: Ukážka definície štruktúry `timer_write_all_transaction`, ktorá dedí atribúty z koreňovej štruktúry `timer_t_transaction`.

Doposiaľ boli uvedené ukážky súborov[15], ktoré musia byť ručne vytvorené a dodané do evolučného algoritmu. Ako je spomenuté v kapitole 5.3.1, do súboru `test_component.rseg` evolučný algoritmus generuje fenotyp jedinca. Na základe fenotypu jedinca sú vygenerované potrebné symboly predstavujúce transakcie, ktoré sú vložené do súboru `struct.rseg`. Formát generovaných symbolov je znázornený na výpise 5.16.

```
timer_write_reset_transaction write_reset_0_struc;
symbol write_reset_0 = do_item(write_reset_0_struc);
```

Výpis 5.16: Ukážka vygenerovaného symbolu `write_reset_0`, ktorý je použitý v grafe 5.13.

[14] Meta-akcie predstavujú generovanie hodnôt pre konkrétny vstup verifikovaného komponentu.
[15] Reprezentácia grafu v súbore `test_component.rseg` a transakcií v súbore `struct.rseg`.

# Kapitola 6

# Experimenty a vyhodnotenie

Cieľom evolučného algoritmu je zjednodušiť návrh verifikačných testov v nástroji InFact 2.4, ktorý môže byť pre neskúseného verifikačného inžiniera časovo náročný. Pri návrhu a implementácii som smeroval k tomu, aby nástroj bol ľahko znovupoužiteľný, i pre verifkáciu iných komponentov, a preto som pri implementácii využil framework PonyGe2 5.1. Navrhnuté riešenie je vďaka širokej možnosti konfigurácií dostatočne obecné a je teda znovupoužiteľné i pre iné komponenty.

V experimentoch skúsim odhadnúť zložitosť použitia navrhnutého riešenia vzhľadom k dosiahnutému pokrytiu komponentu. Jednotlivé experimenty sú rozdelené podľa náročnosti, ktorá sa hodnotí na základe času a úsilia, ktoré musí verifikačný inžinier investovať do tvorby gramatiky a základných transakcií.

V tabuľke 6.1 sú uvedené výpočtové zdroje, na ktorých prebiehali experimenty. Použité experimenty, zložitosť transakcií som musel prispôsobiť zvoleným výpočtovým zdrojom, ktoré boli výrazne obmedzené. Masívna paralelizácia algoritmu nie je jednoduchá z dôvodu obmedzeného počtu licencií k nástroju Questa InFact a Questasim.

Prvá podkapitola 6.1 popisuje zamýšľaný pracovný postup verifikačného inžiniera pri používaní frameworku. Navrhovaný postup je následné aplikovaný na experimenty 1 v podkapitole 6.3 a experiment 2 v podkapitole 6.4. Súčasťou experimentov je i kontrolný experiment (viď podkapitola 6.2), ktorým odhadujem zložitosť verifikovaného obvodu a kvalitu definovaného funkčného pokrytia.

Každá podkapitola experimentu sa skladá z podkapitoly Tvorba transakcií (viď 6.3.1 a 6.4.1), kde popisujem ako som tvoril základné transakcie, z ktorých evolúcia skladá graf. Nasleduje podkapitola Tvorba gramatiky (viď 6.3.2 a 6.4.2) a parametre evolučného algoritmu v podkapitolách 6.3.3 a 6.4.3. V kapitolách Vyhodnotenie experimentu (viď 6.4.4 a 6.4.4) zhodnotím dosiahnuté výsledky evolučného algoritmu. V podkapitole Analýza dosiahnutého pokrytia (viď 6.4.5) skúmam namerané pokrytie a dôvody, prečo jedinci nedosahovali vyššie pokrytie.

V poslednej podkapitole 6.5 navrhnem možné rozšírenia práce, ktoré majú potenciál zjednodušiť použitie frameworku alebo vylepšiť dosahované výsledky. Najväčšie obmedzenie použitia frameworku je dostupnosť licencií, ktoré bránia v masívnej paralelizácii na superpočítači.

| Parameter | Hodnota |
|---|---|
| Procesor | Intel(R) i7-4770 CPU @ 3.40GHz |
| Počet vlákien | 8 |
| SSD | Samsung SSD 840 250GB |
| RAM | 32GB |
| Operačný systém | VirtualBox - CentOS7 |
| Približná doba evaluácie | 0.1~0.2 CPU hodín |

Tabuľka 6.1: Prehľad výpočtových zdrojov na testovacom počítači. Približná doba evaluácie jedinca je 0.1~0.2 CPU hodín, v závislosti na experimentu a dostupnosti licencií.

## 6.1 Použitie frameworku

Zámer, s ktorým bol vyvíjaný framework, je uľahčenie práce verifikačného inžiniera. V tejto podkapitole rozoberiem ideálny pracovný postup, pre ktorý bol framework navrhovaný. Predpokladaný vstup pre použitie evolučného algoritmu je verifikovaný komponent, nastavené verifikačné prostredie a definované funkčné pokrytie pre verifikovaný komponent.

Prvé, čo musí verifikačný inžinier spraviť pri verifikácii, je prečítanie dokumentácie k verifikovanému komponentu, kde je špecifikovaná funkcionalita daného komponentu. Prvý experiment 6.3 predpokladá použitie frameworku bez nutnosti detailného skúmania špecifikácie, kedy verifikačný inžinier identifikuje iba základné transakcie pre vstup verifikovaného komponentu.

Druhý experiment 6.4 predpokladá detailnejšie štúdium špecifikácie, kedy verifikačný inžinier vytvorí podrobnejšie transakcie či časti grafu, z ktorých môže evolúcia tvoriť výsledný graf. Predpokladaný výstup zo spomenutej časti práce verifikačného inžiniera je súbor *.rule* (viď podkapitola 5.6), ktorý obsahuje základnú štruktúru grafu, a súbor *.rseg* (viď podkapitola 5.6.2), ktorý obsahuje definíciu jednotlivých transakcií.

Po vytvorení súboru *.rule* a *.rseg* musí verifikačný inžinier vytvoriť gramatiku (príklad 6.3.2 a 6.4.2), ktorá bude ako terminálne symboly používať navrhnuté transakcie 6.3.1. Tvorba gramatiky je relatívne zložitý proces, ktorý vyžaduje určitú mieru skúseností, pretože i malá zmena gramatiky môže mať veľký vplyv na vygenerované jedince a prehľadávanie priestoru. Pri experimentoch 6.3 a 6.4 boli vytvorené gramatiky, ktoré majú široké použitie, a verifikačný inžinier ich môže použiť ako základ pre tvorbu vlastnej gramatiky.

Po vygenerovaní gramatiky je nutné zvoliť vhodné parametre evolučného algoritmu. Verifikačný inžinier sa môže inšpirovať' použitými parametrami v experimentoch, ktoré boli použité pri verifikovaní komponentu časovač 5.4.1. Pri použití evolučného algoritmu je vhodné nastaviť parametre obmedzujúce veľkosť vygenerovaných grafov. Pri experimentoch som zistil, že pokiaľ evolučný algoritmus generuje príliš veľkých jedincov, nástroj InFact má problém vygenerovať graf. Bezpečné overené parametre sú `max_tree_depth: 25`, `max_tree_nodes: 200` a `max_genome_length: 300`.

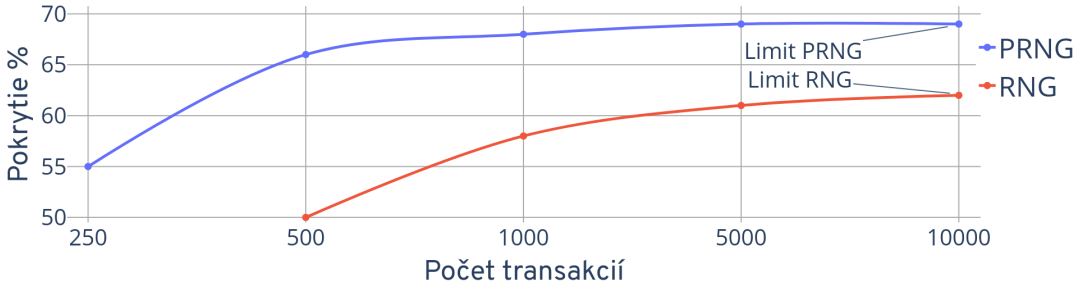
Výstupom evolučného algoritmu je fenotyp jedinca predstavujúci graf pre nástroj InFact, z ktorého je možné vygenerovať testovací komponent *test_component.svh* obsahujúci riadenie a generovanie transakcií pre verifikáciu.

## 6.2 Kontrolný experiment

Ako kontrolný experiment som vyhodnotil dosiahnuté pokrytie verifikovaného komponentu pri použití náhodných transakcií, ktoré sú znázornené v grafe 6.1. Línia RNG[1], predstavuje čisto náhodné generované transakcie a línia PRNG[2] predstavuje pseudonáhodné transakcie, kde pri ich generovaní sú aplikované obmedzenia. Z grafu je zrejmé, že pri použití RNG transakcií, dosiahnuté pokrytie stagnuje na hodnote `~62%` a pri PRNG transakciách na hodnote `~68%`.

V tabuľke 6.2 je podrobná ukážka dosiahnutého štrukturálneho a funkčného pokrytia. Ďalšie zvyšovanie počtu transakcií už neviedlo k výraznému zvyšovaniu dosiahnutého funkčného a štrukturálneho pokrytia.

Význam kontrolného experimentu spočíva v zistení limitov, ktoré sme schopný dosiahnuť pri použití náhodných a pseudonáhodných transakcií v UVM prostredí.




Obr. 6.1: Graf zobrazuje limit dosiahnutého pokrytia pri použití pseudonáhodných sekvencií a náhodných sekvencií. Pseudonáhodné sekvencie dosahovali limitu `~68%` a náhodné sekvencie `~62%` pri počte transakcií `10000`. Línia PRNG predstavuje priemernú hodnotu pokrytia pri použití pseudonáhodných transakcií a línia RNG predstavuje náhodné sekvencie bez obmedzení.

| Parameter | RNG-500 | RNG-10000 | PRNG-500 | PRNG-5000 |
|---|---|---|---|---|
| Skoky | 82.73 | 94.24 | 89.20 | 92.08 |
| Podmienky | 45.88 | 71.76 | 57.64 | 65.88 |
| Funkčné p. | 36.86 | 73.69 | 66.12 | 71.89 |
| Direktivy | 12.50 | 25 | 75 | 75 |
| Výrazy | 33.33 | 38.09 | 28.57 | 28.57 |
| Príkazy | 81.71 | 95.42 | 90.28 | 94.28 |

Tabuľka 6.2: Rozbor druhov dosiahnutého pokrytia pri RNG a PRNG transakciách. Hodnoty v tabuľke sú uvedené v `%` dosiahnutého pokrytia. Formálne tvrdenia angl. *Assertions*, Skoky angl. *Branches*, Podmienky angl. *Conditions*, Funkčné pokrytie angl. *Covergroups*, Direktivy angl. *Directives*, Výrazy angl. *Expressions*, Príkazy *Statements*.

[1]RNG - rozumieme generovanie čiste náhodných transakcií v UVM prostredí, ktoré nás z pohľadu investovaného času stoja zanedbateľné množstvo.

[2]PRNG - predstavuje generovanie pseudonáhodných transakcií v UVM prostredí, na ktoré sú aplikované obmedzenia verifikačným inžinierom, a tak je generovaná iba podmnožina stavového priestoru vstupov verifikovanej komponenty.

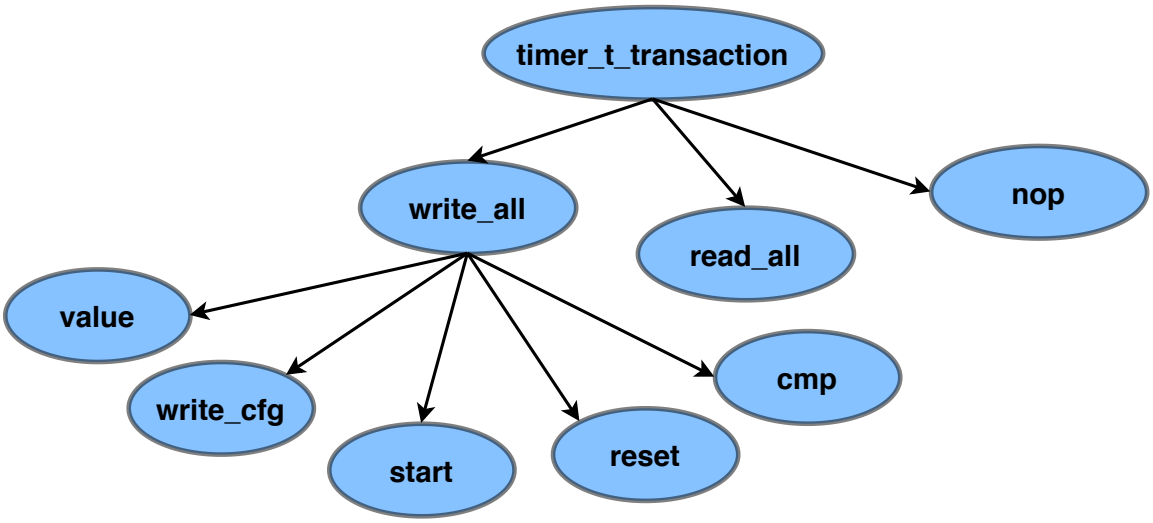
## 6.3 Experiment 1

Nasledujúci experiment predstavuje základné použitie evolučného algoritmu, ktorý predpokladá, že verifikačný inžinier nie je príliš skúsený alebo nechce investovať príliš veľa zdrojov do tvorby zložitej gramatiky a základných transakcií.

### 6.3.1 Tvorba transakcií

Tvorba transakcií pre verifikovaný komponent časovač (viď podkapitola 5.4.1) spočíva v identifikovaní základných akcií, ktoré užívateľ môže interpretovať na vstupné porty časovača. Takéto transakcie môžu byť hierarchicky rozdelené do stromu, kedy koreňová transakcia je najobecnejšia a postupne transakcie smerujúce k listom sú špecifickejšie. Špecifikácia transakcií je implementovaná pomocou obmedzení, ktoré sú popísané v kapitole 2.4.1. Stromová hierarchia je implementovaná pomocou definícií transakcií ako štruktúr, ktoré dedia informácie od nadradenej štruktúry. Implementácia štruktúr je podrobnejšie popísaná v podkapitole 5.6.1.

Na obrázku 6.2 je znázornený hierarchický strom, kde jednotlivé uzly predstavujú základné transakcie. Myslím si, že i neskúsený verifikačný inžinier po prečítaní špecifikácie, dokáže identifikovať a vytvoriť takéto základné transakcie.




Obr. 6.2: Na obrázku je znázornená hierarchia transakcií definovaných verifikačným inžinierom. Koreňová transakcia `timer_t_transaction` je najobecnejšia transakcia, ktorá môže generovať všetky validné dáta na rozhranie časovaču. Smerom k listom sú transakcie špecifickejšie, čo je spôsobené aplikovaním obmedzení na generované dáta. Transakcia `write_all` aplikuje obmedzenia na generované dáta tak, aby transakcia predstavovala zápis do niektorého z registrov čítaču. Transakcie `value`, `write_cfg`, `start`, `reset` a `cmp` dedia všetky meta-akcie, biny a obmedzenia z rodičovskej štruktúry `write_all` a pridávajú ďalšie špecifické obmedzenia, najčastejšie sa jedná o zmenu cieľovej adresy zápisu, prípadne vymedzení validných dát. Transakcia `read_all` aplikuje obmedzenia na generované dáta tak, aby predstavovala čítanie z niektorého z registrov časovača. Posledná transakcia `nop` simuluje stav bez generovania validných transakcií pre časovač.

### 6.3.2 Tvorba gramatiky

V rámci prvého experimentu bola použitá jednoduchá gramatika 6.1, ktorá umožňuje generovať tie najzákladnejšie grafy. Táto gramatika je znovupoužiteľná pre veľké spektrum komponentov a nie je potrebné ju výrazne meniť. Ako bolo spomenuté, cieľom daného experimentu je testovanie prístupu, ktorý vyžaduje čo najmenej zdrojov a skúseností verifikačného inžiniera, preto i samotná gramatika by mala byť čo najjednoduchšia. Nevýhody danej gramatiky spočívajú v jej nevyváženosti a obmedzenom generovaní syntaxe fenotypu pre popis grafu v nástroji InFact. Pre jej jednoduchosť je vhodná na testovanie zvolených transakcií a evolučného algoritmu.

```
<exp> ::=   <seq> <exp> |
            <seq> "|" <exp> |
            ( <seq> <exp> ) "|" <seq> <exp> |
            <seq>
<seq> ::=   write_all | read_all | nop |
            write_cfg | write_value |
            write_cmp | write_start | write_reset
```

Výpis 6.1: Gramatika generuje základné konštrukcie grafu pomocou zanorenia `(`, `)` a symbolu `|`, ktorý predstavuje logický OR. Neterminál `<seq>` obsahuje základné transakcie.

### 6.3.3 Parametre evolučného algoritmu

Hlavné parametre evolučného algoritmu v tomto experimente sú znázornené v tabuľke 6.3. Počet jedincov a generácií som experimentálne odhadol s ohľadom na obmedzené výpočtové zdroje. Dĺžka behu evolučného algoritmu sa dá odhadnúť zo vzťahu 6.1, kedy čas evaluácie jedinca *EvaluationTime(x)* sa na uvedených zdrojoch pohybuje niekde medzi 0.1~0.2 min, v závislosti na zložitosti vygenerovaného grafu, dostupných licencií a podobne. Z uvedeného vzťahu môžeme teda odvodiť, že predpokladaná celková náročnosť jedného behu experimentu je približne ~40 CPU hodín. Tento čas je možné zredukovať paralelizáciou a prípadnou dostupnosťou licencií. Kompletný výpis parametrov sa nachádza v súbore *exp1_param.txt*, prípadne pri výsledkoch experimentu.

$$CPUTime = EvaluationTime(x) * gen * pop \tag{6.1}$$

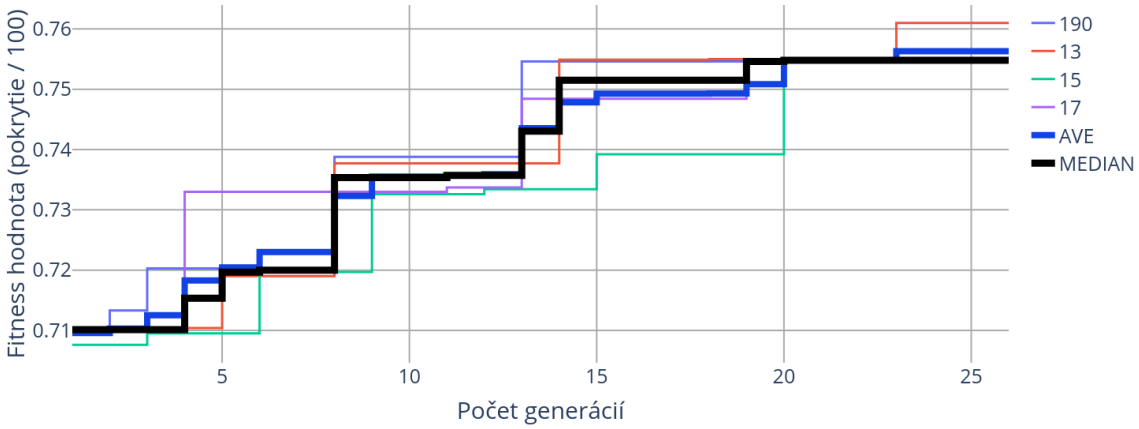
| Parameter | Hodnota | Popis |
|---|---|---|
| POPULATION_SIZE | 16 | veľkosť populácie |
| GENERATIONS | 25 | počet generácií |
| MAX_HISTORY | 5 | parameter ukončenia |
| CROSSOVER_PROB | 0.75 | pravdepodobnosť kríženia |
| MUTATION_PROB | 0.3 | pravdepodobnosť mutácie |
| INFACT_ITERATION | 500 | iterácie grafu |
| MAX_TREE_DEPTH | 25 | maximálna hĺbka stromu |

Tabuľka 6.3: Tabuľka znázorňuje základné parametre evolučného algoritmu, použité v rámci experimentu 1. Podrobnejší popis parametrov sa nachádza v kapitole Konfigurácia evolučného algoritmu 5.2.

### 6.3.4 Vyhodnotenie experimentu

Výsledky experimentu sú znázornené v grafe 6.3 a najlepší jedinec je znázornený na obrázku 6.4. Modrá línia grafu predstavuje priemernú hodnotu najlepšieho jedinca z niekoľkých behov s rôznym počiatočným nastavením RNG. Ostatné línie predstavujú fitness hodnotu najlepšieho jedinca vzhľadom ku generácií evolučného algoritmu. Z uvedeného grafu je vidieť, že evolúcia v počiatočných generáciách dosahuje pokrytie ~70% a postupne dokáže nájsť riešenie, ktoré dosahuje pokrytia ~76%.

Porovnanie s kontrolným experimentom je v tabuľke 6.4. Pri porovnaní dosiahnutého pokrytia pri 500 transakciách je vidieť rozdiel pri PRNG 10% pre beh evolučného algoritmu a pri RNG rozdiel 26%. Keď porovnáme limity pokrytia, ktoré je možné dosiahnuť pri PRNG, dostávame rozdiel 8% a pri RNG rozdiel 12%.




Obr. 6.3: Graf predstavuje výsledky experimentu 1. V grafe je zobrazený priebeh niekoľkých behov s rôznym počiatočným nastavením generátoru náhodných čísiel, priemerná hodnota dosiahnutá pre všetky behy experimentu označená ako AVE a medián zo všetkých behov evolučného algoritmu označený čiernou farbou. Parametre experimentu sú znázornené v tabuľke 6.3.

| Druh pokrytia | Najlepší jedinec-500 | PRNG-500 | PRNG-5000 |
|---|---|---|---|
| Skoky | 95.68 | 89.20 | 92.08 |
| Podmienky | 85.88 | 57.64 | 65.88 |
| Funkčné pokrytie | 70.53 | 66.12 | 71.89 |
| Direktivy | 62.50 | 75 | 75 |
| Výrazy | 61.90 | 28.57 | 28.57 |
| Príkazy | 97.71 | 90.28 | 94.28 |

Tabuľka 6.4: Rozbor druhov dosiahnutého pokrytia najlepšieho jedinca, RNG a PRNG transakcií. Skoky angl. *Branches*, Podmienky angl. *Conditions*, Funkčné pokrytie angl. *Covergroups*, Direktivy angl. *Directives*, Výrazy angl. *Expressions*, Príkazy angl. *Statements*.

### 6.3.5 Analýza dosiahnutého pokrytia

Cieľom tejto analýzy pokrytia kódu je preskúmanie dôvodov, prečo jedinci nedosahujú `100%` funkčné i štrukturálne pokrytie. Na obrázku 6.4 je znázornený jedinec, na základe ktorého generujeme databázu dosiahnutého pokrytia *coverage.ucdb*. Z tejto databázy môžeme pomocou príkazu `vcover report -html -source -details coverage.ucdb` vygenerovať podrobný verifikačný report.

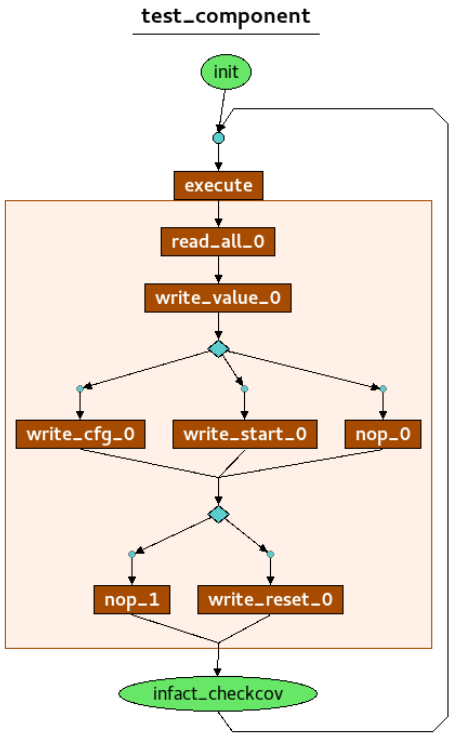
V tabuľke 6.3 je verifikačný report znázorňujúci dosiahnuté definované funkčné pokrytie, ktoré je popísané v kapitole 5.4.2, tento verifikačný report je dostupný v zložke *exp1 best*. Dosiahnuté funkčné pokrytie sa skladá z dvoch skupín. Jedna skupina, `FunctionalCoverage`, kontroluje základne biny, krížové biny a bazálnu funkcionalitu časovača.

Skupina `CornerCasesCoverage` skúma dosiahnutie okrajových situácií, ktoré by mohli pri funkcií časovača nastať, ako napríklad pretečenie časovača. Z verifikačného reportu súdim, že síce základná funkcionalita časovača bola pokrytá, vygenerované sekvencie neboli príliš sofistikované a netestovali okrajové situácie časovača.

Štrukturálne pokrytie výrazov dosiahlo hodnoty `61%` a po preskúmaní vygenerovaného reportu som zistil, že každý nepokrytý výraz súvisel s podmienkou, ktorá detekuje pretečenie časovaču.

Ďalšie štrukturálne pokrytie podmienok dosiahlo hodnoty `83%`. Po preskúmaní verifikačného reportu som zistil, že implementácia preddeličky časovača obsahuje kód, ktorý sa nemôže nikdy vykonať, nazývaný tiež mŕtvy kód zobrazený na výpise 6.3.

Pokrytie direktiv obsahuje sofistikovanejšie situácie, ktoré môžu nastať počas behu časovača a dosiahlo hodnoty `62.5` a posledné pokrytie príkazov dosiahlo dostatočnú hodnotu pokrytia a to `97.71`.




Obr. 6.4: Ukážka jedinca vygenerovaného evolučným algoritmom.

```
1 if (write_counter_i == 1) // OVERWRITE COUNTER
2     s_count = counter_value_i;
3 else
4 ...
```

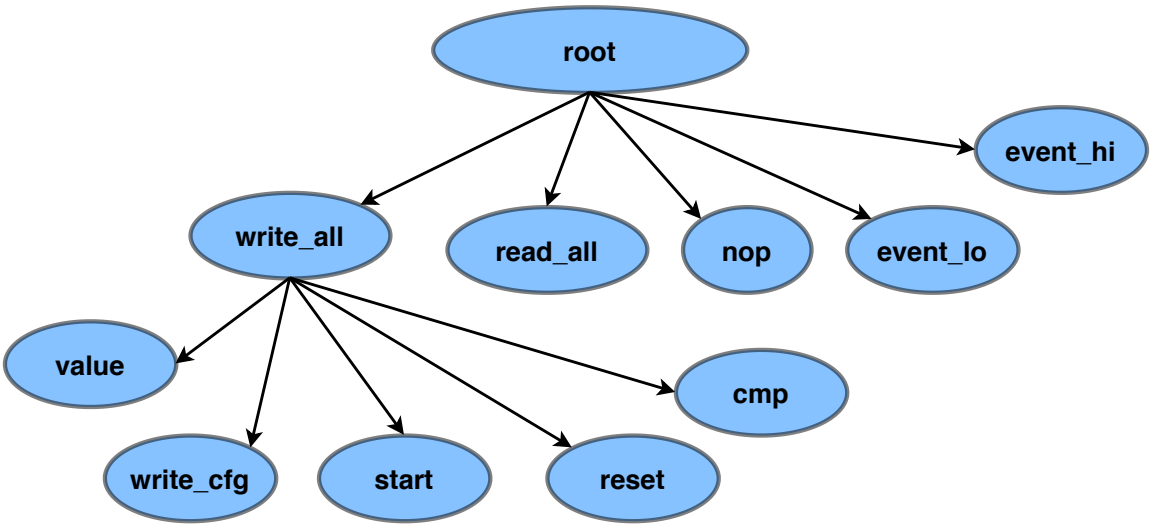
Výpis 6.2: Ukážka mŕtveho kódu modulu preddeličky časovača. Podmienka kontroluje, či vstupný signál `write_counter_i` je nastavený v logickej `1`. Pri inštanciaci modulu preddeličky je tento vstupný signál konštantne pripojený na logickú `0` a tak tato podmienka nemôže nikdy dosiahnuť pravdivú hodnotu.

## 6.4 Experiment 2

Na základe prvého experimentu 6.3 a analýzy dosiahnutého funkčného a štrukturálneho pokrytia 6.3.5 som pripravil pokročilejší experiment, ktorý obsahuje komplexnejšiu gramatiku 6.6 a podrobnejšie transakcie 6.5. Tento experiment predpokladá, že verifikačný inžinier investuje do návrhu transakcií a gramatiky vyššiu mieru úsilia než pri prvom experimente.

### 6.4.1 Tvorba transakcií

Základ pre transakcie druhého experimentu som použil transakcie z prvého experimentu 6.2. Ako prospešné sa ukázalo zaviesť do grafu aj určitú mieru náhody a preto som pridal transakciu `root`, ktorá dokáže generovať všetky hodnoty vstupných portov časovača[3]. Ďalej som pridal transakcie špeciálne na generovanie externých udalostí pre signály `event_lo_i` a `event_hi_i`.




Obr. 6.5: Na obrázku je znázornený hierarchický strom, ktorého základ je prebratý z experimentu 1 6.2. Do stromu boli pridané transakcie `event_lo_i` a `event_hi_i` (viď výpis 6.3) a koreňová transakcia `root`.

```
1 struct timer_eventhi_transaction extends timer_t_transaction {
2     constraint write_c {
3         rst_ni == 1;
4         req_i == 0;
5         event_lo_i == 0;
6         event_hi_i == 1;
7     }
8 }
```

Výpis 6.3: Ukážka definície novej transakcie `event_hi`, ktorá generuje logickú `1` na vstupný signál `event_hi_i`. Obdobným spôsobom je vytvorená i transakcia `event_lo`.

Upravil som transakciu `write_cfg` tak, aby generovala iba validné hodnoty podľa špecifikácie časovača (viď výpis 6.4).

[3]Okrem invalidných adries vnútorných registrov.

```
struct timer_write_cfg_transaction extends timer_write_all_transaction {
    constraint prescaler_write{ wdata_i[15:8] inside [0, 1]};
    constraint reset_off{ wdata_i[1] == 0};
    constraint active{ wdata_i[0] == 0};
    constraint write_reserve{ wdata_i[30:16] == 0};
    constraint reserve {
      if(addr_i == 0x04) {wdata_i[31] == 0;};
    };
}
```

Výpis 6.4: Ukážka úpravy transakcie `write_cfg` tak, aby sa vygenerovali iba validné hodnoty vstupu. Do preddeličky sa vkladajú iba malé hodnoty tak, aby počítanie trvalo čo najmenej taktov.

Ďalej som upravil transakcie `write_cmp` a `write_value` pridaním binov a obmedzení tak, aby sa negenerovali čiste náhodné čísla ale iba čísla, ktoré majú rozumné rozpätie hodnôt. Tento krok je dôležitý pre to, aby častejšie dochádzalo k situáciám, že časovač v registri `value` dosiahne očakávanú hodnotu registra `cmp`. Zároveň som pridal maximálnu hodnotu registra `value 0xffffffff` a `0xfffffffa` tak, aby mala evolúcia väčšiu šancu vygenerovať graf, ktorý testuje funkcionalitu časovača pri pretečení registra `value`. V ukážke kódu 6.5 sú principiálne načrtnuté vykonané zmeny.

```
struct timer_write_cmp_transaction extends timer_write_all_transaction {
  bins wdata_i wdata  [5][110][510][10005][0xffffffff];
  bins addr_i addr  [0x10][0x14];
  bin_scheme combination {
    wdata_i wdata;
    addr_i addr;
  };
}
```

Výpis 6.5: Ukážka úpravy transakcie `write_cmp` pridaním cielených hodnôt. Podobne je upravená i transakcia `write_value`.

### 6.4.2 Tvorba gramatiky

V rámci experimentu 2. som rozšíril gramatiku 6.6 o generovanie cyklov pomocou kľúčového slova `repeat` 2.4.1. Táto konštrukcia umožňuje generovať v grafe situáciu, kedy časovač počíta a nechceme generovať žiadne požiadavky, ktoré by mohli prerušiť beh časovača. Gramatika bola upravená tak, aby bola viac vyvážená a tak generovala menej invalidných potomkov. Zároveň bola rozšírená množina terminálov, ktoré predstavujú transakcie časovača (viď podkapitola 6.4.1).

```
<exp> ::=   <seq> <exp> | <seq> <exp> | <seq> <exp> |
            <seq> "|" <exp> | <seq> "|" <exp> |
            ( <seq> <exp> ) "|" <seq> <exp> |
            ( <seq> <exp> ) "|" <seq> <exp> |
            <seq> | <seq> | <seq> |
            repeat <N> { nop }
<seq> ::=   write_all | read_all | nop | write_cfg | write_value | write_cmp |
            write_start | write_reset | root | eventlo | eventhi
<N> ::=     5 | 10 | 15 | 20
```

Výpis 6.6: Gramatika generuje základné konštrukcie grafu pomocou zanorenia `(`, `)` a symbolu `|`, ktorý predstavuje logický OR. Zároveň bolo pridané kľúčové slovo `repeat` a pridaním duplicitných pravidiel došlo k vyváženiu gramatiky.

### 6.4.3 Parametre evolučného algoritmu

Hlavné parametre evolučného algoritmu v tomto experimente sú znázornené v tabuľke 6.5. Vzhľadom k tomu, že zmenou gramatiky a pridaním nových transakcií sa zvýšil prehľadávaný stavový priestor, som musel zvýšiť počet jedincov a maximálny počet generácií evolučného algoritmu. Dĺžka behu evolučného algoritmu sa dá znovu odhadnúť zo vzt'ahu 6.1, pričom *EvaluationTime(x)* sa pohybuje niekde medzi 0.1~0.2 min. Z uvedeného vzt'ahu môžeme teda odvodiť, že predpokladaná celková náročnosť jedného behu experimentu je približne ~96 CPU hodín. Kompletný výpis parametrov sa nachádza v súbore *exp2_param.txt*, prípadne pri výsledkoch experimentu.

| Parameter | Hodnota | Popis |
|---|---|---|
| POPULATION_SIZE | 24 | veľkosť populácie |
| GENERATIONS | 40 | maximálny počet generácií |
| MAX_HISTORY | 10 | parameter ukončenia |
| CROSSOVER_PROB | 0.75 | pravdepodobnosť kríženia |
| MUTATION_PROB | 0.3 | pravdepodobnosť mutácie |
| INFACT_ITERATION | 600 | iterácie grafu |

Tabuľka 6.5: Tabuľka znázorňuje základné parametre evolučného algoritmu použité v rámci experimentu 2. Podrobnejší popis parametrov sa nachádza v podkapitole 5.2.

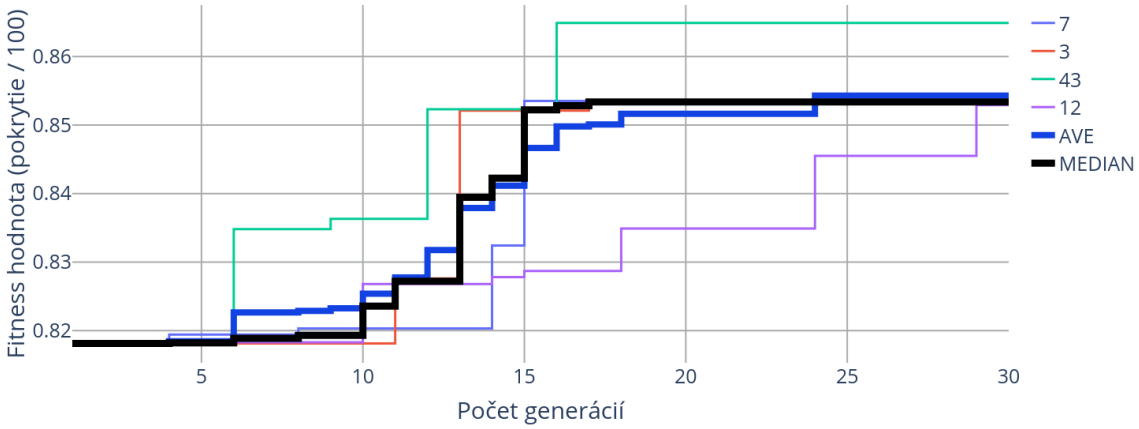
### 6.4.4 Vyhodnotenie experimentu

Línia v grafe 6.6, zobrazená modrou farbou, predstavuje priemernú hodnotu najlepšieho jedinca z niekoľkých behov s rôznym počiatočným nastavením RNG. Čierna línia predstavuje medián všetkých behov experimentu. Ostatné línie predstavujú fitness hodnotu najlepšieho jedinca vzhľadom ku generácií evolučného algoritmu.

Celkové porovnanie PRNG sekvencií, výsledkov experimentu 1 a výsledkov experimentu 2 sú znázornené v tabuľke 6.6. Z tejto tabuľky môžeme sledovať posun v dosiahnutom funkčnom a štrukturálnom pokrytí od pseudonáhodných sekvencií až po experiment 2, ktorý dosiahol najlepšie výsledky. Priemerná hodnota pokrytia jedincov v rámci experimentu bola `~86%`, najlepšia dosiahnutá hodnota jedinca bola `~93%` (vid' 6.7). Pričom priemerná hodnota jedincov v rámci experimentu 1 bola `~76%`. Z uvedených výsledkov môžeme sledovať posun oproti PRNG transakciám niekde medzi `20~30%` a oproti experimentu 1 6.3 približne `~10%`.

| Druh pokrytia | exp2-500 | exp1-500 | PRNG-500 | PRNG-5000 |
|---|---|---|---|---|
| Skoky | 98.51 | 95.68 | 89.20 | 92.08 |
| Podmienky | 92.77 | 85.88 | 57.64 | 65.88 |
| Funkčné p. | 99.65 | 70.53 | 66.12 | 71.89 |
| Direktivy | 87.50 | 62.50 | 75 | 75 |
| Výrazy | 76.19 | 61.90 | 28.57 | 28.57 |
| Príkazy | 100 | 97.71 | 90.28 | 94.28 |

Tabuľka 6.6: Porovnanie dosiahnutého pokrytia a experimentu 1, experimentu 2 a PRNG transakcií.



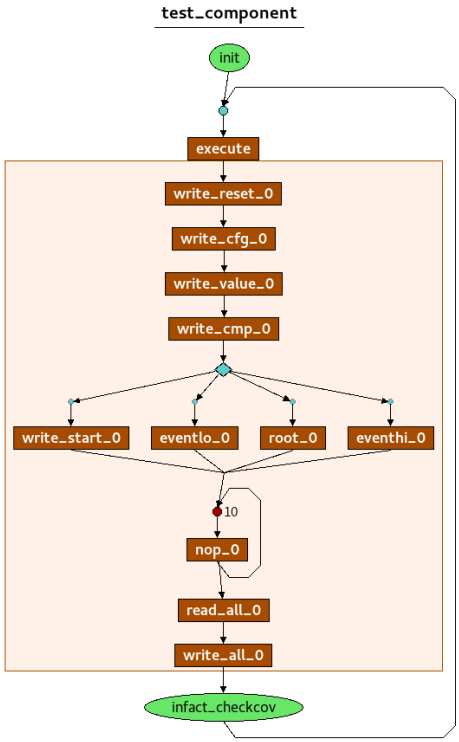

Obr. 6.6: Graf predstavuje výsledky experimentu 2. V grafe je zobrazený priebeh niekoľkých behov s rôznym počiatočným nastavením RNG, priemerná hodnota a medián dosiahnutý pre všetky behy experimentu. Parametre experimentu sú znázornené v tabuľke 6.5

### 6.4.5 Analýza dosiahnutého pokrytia

Podobne ako v experimente 1 bol z databáze *coverage.ucdb* vygenerovaný podrobný verifikačný report dosiahnutý najlepším jedincom 6.7.

Pokrytie skokov dosiahlo hodnoty `98.51` a z analýzy verifikačného reportu som zistil, že jediný nepokrytý skok je v podmienke čítania z registrov časovača. Na ukážke kódu 6.7 je znázornená implementácia čítania z registrov časovača, pričom v podmienkach sú iba registre, ktoré sú podľa špecifikácie povolené k čítaniu. Problém spočíva v tom, že výber signálu `s_addr[5:0]` zahrňuje adresovanie pre všetky registre i tie, ktoré sú určené iba k zápisu. (viď podkapitola Verifikovaný komponent - Časovač 5.4.1) V tomto prípade, i keď to nie je v zdrojových kódoch implicitne zapísané, je vygenerovaný stav pre všetky ostatné nepokryté adresy registrov. Problém spočíva v tom, že transakcia určená k čítaniu generuje iba adresy určené k čítaniu.

Pokrytie podmienok dosiahlo hodnoty `92.77`. Pokrytie výrazov dosiahlo hodnoty `76.19`, chýbajúce hodnoty súvisia s krajovou situáciou, kedy počítanie časovača pretečie, preddelička dosiahne požadovanú hodnotu a referenčné hodiny sú reštartované. Táto situácie je veľmi špecifická a vygenerovať ju náhodne je obtiažne. Podobné okrajové situácie je možné ručne zaviesť do grafu alebo doplniť priamymi testami.




Obr. 6.7: Ukážka jedinca vygenerovaného evolučným algoritmom.

```
case (s_addr[5:0])
    `CFG_REG_LO:
        r_rdata_o = s_cfg_lo_reg;
    `CFG_REG_HI:
        r_rdata_o = s_cfg_hi_reg;
        ...
    `TIMER_CMP_HI:
        r_rdata_o = s_timer_cmp_hi_reg;
endcase
```

Výpis 6.7: Ukážka implementácie čítania z registrov časovaču.

## 6.5 Možné rozšírenie a pokračovanie práce

Pri návrhu a implementácií nástroja som vychádzal z predpokladu, že na vyhodnotenie a experimenty mám veľmi obmedzené zdroje. Generovať zložité konštrukcie v syntaxe jazyku pre nástroj InFact by bolo nereálne. Nástroj som navrhol obecne tak, aby mal širokú možnosť konfigurácie. Počas návrhu a implementácie som prišiel na niekoľko možných rozšírení, ktoré by relatívne jednoducho šli doplniť do aktuálneho nástroja.

Jednou z možností je doplniť grafy o tagy, ktoré umožnia cielenejšie smerovať generované transakcie. Pridanie tagov by spočívalo iba v úprave gramatiky a definícii tagov do súboru `.rseg`, podobne ako užívateľské transakcie.

Dynamické obmedzenia, ktoré by sa aplikovali podobne, ako užívateľom definované transakcie. Pridanie dynamických obmedzení by vyžadovalo doplnenie gramatiky a definíciu obmedzení do súboru `.rseg`, podobne ako užívateľské transakcie.

Ďalšia z možností je pohľad z opačného smeru a to optimalizácia atribútov transakcií, ktoré sú definované v statickom grafe. Jedná sa o optimalizáciu užívateľom definovaného grafu. Tento prístup by vyžadoval väčšie zmeny implementovaného riešenia, ale táto práca by mohla slúžiť ako návod pre riešenie mnoho problémov, ktoré by prístupy mali spoločné. Napríklad skripty evaluácie jedinca, nastavenie verifikačného prostredia, práca s frameworkom PonyGe2 a podobne.

Ďalšia z možností je využitie generovania alebo optimalizácie stratégie pokrytia, a tak ešte výraznejšie znížiť redundanciu generovaných transakcií. Tento prístup by bol mierne zložitejší a pravdepodobne by vyžadoval väčšie výpočtové zdroje a viacej dostupných licencií k nástrojom.

Posledná myšlienka na možné pokračovanie práce spočíva v špecifickejšom definovaní funkčného pokrytia a priradení jednotlivým pokrytím rôzne váhy za cieľom dosiahnutia špecifických testov, ktoré síce nebudú dosahovať veľké celkové pokrytie, ale budú smerovať generovanie stimulov na špecifickejšie situácie. Tento prístup môžeme prirovnať ku generovaniu priamych testov.

# Kapitola 7

# Záver

Narastajúca zložitosť hardwardových systémov zvyšuje potrebu komplexnejšej verifikácie, čo prináša veľa problémov s efektivitou aktuálne používaných nástrojov k verifikácií hardwardových obvodov. V súčastnosti je jednou z najviac používaných metodík verifikácia v prostredí UVM. Tento prístup prináša prvky objektovej orientácie s cieľom znovupoužiteľnosti verifikovaného prostredia. Tento prístup je úspešne používaný na úrovni IP blokov, ale na systémovej úrovni tento prístup zlyháva.

Štandard prenositeľných stimulov PSS má za cieľ zvýšiť mieru znovupoužiteľných testov a odstraňovať mieru redundancie v generovaných transakciách. Znovupoužiteľnosť je aplikovaná v definovaní iba zámeru testu a následným využitím testu na rôznych úrovniach hierarchie (simulátory, FPGA prototypovanie, SoC a podobne). Redundancia testov je odstraňovaná pomocou riadenia generátoru verifikačných stimulov napojeného na UVM prostredie. Štandard PSS sa nesnaží nahradiť súčasné verifikačné metodiky.

Táto práca sa zameriava možnosťami automatického generovania testov pre štandard PSS. Cieľom práce je pomocou evolučných algoritmov generovať také testy, ktoré budu dosahovať čo najvyššie štrukturálne a funkčné pokrytie. V rámci práce som navrhol nástroj, ktorý na základe užívateľom definovaných transakcií a gramatiky dokáže generovať verifikačné scenáre, ktoré cielene dosahujú vysokú mieru funkčného a štrukturálneho pokrytia. Z evolučných algoritmov bola využitá metóda gramatickej evolúcie a jej implementácia v podobe frameworku PonyGe2.

Evaluácia jedincov spočíva vo vygenerovaní súborov potrebných pre nástroj Questa InFact, ktorý predstavuje proprietárnu implementáciu štandardu PSS. Po skompilovaní vygenerovaných súborov nástroj InFact generuje komponent, ktorý riadi generovanie transakcií v rámci verifikačného prostredia UVM. Pri návrhu nástroja bol kladený dôraz na možnosť generovať testy pre široké spektrum hardwardových obvodov a zároveň jednoduchosť použitia. Konfigurácia pomocou užívateľom zvolených transakcií a gramatiky toto umožňuje.

Použitie nástroja som overil na komponentu časovač pomocou dvoch experimentov. V prvom experimente som cielil na jednoduché použitie a čo najnižšiu mieru vynaložených zdrojov verifikačného inžiniera. Pri tomto prístupe evolučný algoritmus dosahoval priemerne o ~10% lepšie celkové pokrytie než pseudonáhodné transakcie 6.4.4. Druhý experiment používal sofistikovanejšie transakcie a o niečo zložitejšiu gramatiku oproti prvému experimentu. Jedinci v tomto experimente dosahovali v priemere o ~20% vyššie celkové pokrytie oproti pseudonáhodným transakciám 6.4.4.

# Literatúra


[1] *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001): IEEE Standard for Verilog Hardware Description Language*. 3 Park Avenue, New York, NY 10016-5997, USA: IEEE, 2006. OCLC: 956668587. Dostupné z: http://ieeexplore.ieee.org/servlet/opac?punumber=10779. ISBN 9780738148502.

[2] Ballance, M. *Automating Tests with Portable Stimulus from IP to SoC Level | Verification Horizons - March 2017 | Verification Academy*. 2017. Dostupné z: https://verificationacademy.com/verification-horizons/march-2017-volume-13-issue-1/automating-tests-with-portable-stimulus-from-ip-to-soc-level.

[3] Ballance, M. *Portable Stimulus: How to master the three axes of reuse*. Apríl 2019. Publisher:. Dostupné z: https://www.techdesignforums.com/practice/technique/how-to-use-three-axes-of-reuse-to-focus-portable-stimulus/.

[4] Brauer, M. J., Holder, M. T., Dries, L. A., Zwickl, D. J., Lewis, P. O. et al. Genetic Algorithms and Parallel Processing in Maximum-Likelihood Phylogeny Inference. *Molecular Biology and Evolution*. október 2002, roč. 19, č. 10, s. 1717–1726. Dostupné z: http://academic.oup.com/mbe/article/19/10/1717/1258966. ISSN 1537-1719, 0737-4038.

[5] Byrne, J., O'Neill, M. a Brabazon, A. Structural and nodal mutation in grammatical evolution. In: Byrne, J., ed. *Proceedings of the 11th Annual conference on Genetic and evolutionary computation - GECCO '09*. Montreal, Québec, Canada: ACM Press, 2009, s. 1881. Dostupné z: http://portal.acm.org/citation.cfm?doid=1569901.1570215. ISBN 9781605583259.

[6] Bäck, T., Foussette, C. a Krause, P. *Contemporary evolution strategies*. 1. vyd. 2013. OCLC: 1132012947. Dostupné z: http://dx.doi.org/10.1007/978-3-642-40137-4. ISBN 9783642401374.

[7] Castle, T. a Johnson, C. G. Positional Effect of Crossover and Mutation in Grammatical Evolution. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F. et al., ed. *Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 26–37. Dostupné z: http://link.springer.com/10.1007/978-3-642-12148-7_3. ISBN 9783642121470 9783642121487.

[8] Corporation, M. G. *Questa inFact User's Manual*. Mentor Graphics Corporation.

[9] Crestani, F., Pasi, G. a Vrajitoru, D. *Soft Computing in Information Retrieval : Techniques and Applications*. 1. vyd. Heidelberg: Physica-Verlag HD Imprint Physica, 2000. Dostupné z: https://link.springer.com/chapter/10.1007/978-3-7908-1849-9_9. ISBN 978-3-7908-1849-9.

[10] De Jong, K., Fogel, D. a Schwefel, H.-P. A history of evolutionary computation. In:. Január 1997, s. A2.3:1–12.

[11] Doerr, B., Kötzing, T., Lagodzinski, J. A. G. a Lengler, J. Bounding bloat in genetic programming. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Berlin Germany: ACM, Júl 2017, s. 921–928. Dostupné z: https://dl.acm.org/doi/10.1145/3071178.3071271. ISBN 9781450349208.

[12] Eiben, A. E. a Smith, J. E. *Introduction to Evolutionary Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. OCLC: 851370480. Dostupné z: https://public.ebookcentral.proquest.com/choice/publicfullrecord.aspx?p=3099871. ISBN 9783662050941.

[13] Eiben, A. E. a Smith, J. E. *Introduction to evolutionary computing*. 2. ed. Heidelberg: Springer, 2015. Natural computing series. OCLC: 934627991. ISBN 9783662448748 9783662448731.

[14] Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., Hemberg, E. et al. PonyGE2: grammatical evolution in Python. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Berlin Germany: ACM, Júl 2017, s. 1194–1201. Dostupné z: https://dl.acm.org/doi/10.1145/3067695.3082469. ISBN 9781450349390.

[15] Fitzpatrick, T. *Connecting Components | Basic UVM | Universal Verification Methodology | Verification Academy*. 2013. Dostupné z: https://verificationacademy.com/sessions/uvm-connecting-components.

[16] Fitzpatrick, T. *Introducing Transactions | Basic UVM | Universal Verification Methodology | Verification Academy*. 2013. Dostupné z: https://verificationacademy.com/sessions/uvm-introducing-transactions.

[17] Fitzpatrick, T. *Introduction to UVM | Basic UVM | Universal Verification Methodology | Verification Academy*. 2013. Dostupné z: https://verificationacademy.com/sessions/introduction-uvm.

[18] Fitzpatrick, T. *Sequences and Tests | Basic UVM | Universal Verification Methodology | Verification Academy*. 2013. Dostupné z: https://verificationacademy.com/sessions/uvm-sequences-and-tests.

[19] Fitzpatrick, T. *UVM "Hello World"| Basic UVM | Universal Verification Methodology | Verification Academy*. 2013. Dostupné z: https://verificationacademy.com/sessions/uvm-hello-world.

[20] Fitzpatrick, T. *Portable Stimulus Basics - Why Portable Stimulus*. 2017. Dostupné z: https://verificationacademy.com/sessions/why-portable-stimulus.

[21] Fitzpatrik, T. *Monitors and Subscribers | Basic UVM | Universal Verification Methodology | Verification Academy*. 2013. Dostupné z: https://verificationacademy.com/sessions/uvm-monitors-and-subscribers.

[22] Fitzpatrik, T. *Portable Stimulus Basics Course | Verification Academy*. 2017. Dostupné z: https://verificationacademy.com/courses/portable-stimulus-basics.

[23] Fogel, D. B., ed. *Evolutionary computation: the fossil record*. New York: IEEE Press, 1998. ISBN 9780780334816.

[24] Fogel, D. B. *Evolutionary computation: toward a new philosophy of machine intelligence*. 3rd ed. Hoboken, N.J: John Wiley & Sons, 2006. ISBN 9780471669517.

[25] Foster, H. D. Trends in functional verification: A 2014 industry study. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2015, s. 1–6. Dostupné z: https://ieeexplore.ieee.org/abstract/document/7167232. ISSN 0738-100X.

[26] Hamid, A. *11 Myths About Portable Stimulus*. Jún 2018. Dostupné z: https://www.electronicdesign.com/technologies/test-measurement/article/21806605/11-myths-about-portable-stimulus.

[27] Harper, R. a Blair, A. A Structure Preserving Crossover In Grammatical Evolution. In: *2005 IEEE Congress on Evolutionary Computation*. Edinburgh, Scotland, UK: IEEE, 2005, s. 2537–2544. Dostupné z: http://ieeexplore.ieee.org/document/1555012/. ISBN 9780780393639.

[28] Holland, J. H. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. 1992. OCLC: 1167518483. Dostupné z: http://cognet.mit.edu/book/adaptation-natural-and-artificial-systems. ISBN 9780262275552 9780262082136 9780262581110.

[29] Hugosson, J., Hemberg, E., Brabazon, A. a O'Neill, M. Genotype representations in grammatical evolution. *Applied Soft Computing*. január 2010, roč. 10, č. 1, s. 36–43. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S1568494609000611. ISSN 15684946.

[30] Koza, J. A genetic approach to the truck backer upper problem and the inter-twined spiral problem. In: *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*. Baltimore, MD, USA: IEEE, 1992, s. 310–318. Dostupné z: http://ieeexplore.ieee.org/document/227324/. ISBN 9780780305595.

[31] Langdon, W. B. a Poli, R. *Foundations of genetic programming*. Berlin; New York: Springer, 2010. OCLC: 968506530. ISBN 9783662047262.

[32] Nicolau, M. a Agapitos, A. Understanding Grammatical Evolution: Grammar Design. In: Ryan, C., O'Neill, M. a Collins, J., ed. *Handbook of Grammatical Evolution*. Cham: Springer International Publishing, 2018, s. 23–53. Dostupné z: http://link.springer.com/10.1007/978-3-319-78717-6_2. ISBN 9783319787169 9783319787176.

[33] O'Neill, M. a Ryan, C. *Grammatical evolution: evolutionary automatic programming in an arbitrary language*. 1. vyd. SPRINGER SCffiNCE+BUSINESS MEDIA, LLC, 2003. OCLC: 1166800104. ISBN 978-1-4615-0447-4.

[34] O'Neill, M., Ryan, C., Keijzer, M. a Cattolico, M. Crossover in Grammatical Evolution. *Genetic Programming and Evolvable Machines*. 2003, roč. 4, č. 1, s. 67–93. Dostupné z: http://link.springer.com/10.1023/A:1021877127167. ISSN 13892576.

[35] Piziali, A. Assertion Coverage. In: *Functional Verification Coverage Measurement and Analysis*. Boston, MA: Springer US, 2004, s. 97–107. Dostupné z: http://link.springer.com/10.1007/978-1-4020-8026-5_6. ISBN 9780387739922.

[36] Poli, R., Langdon, W. B., McPhee, N. F. a Koza, J. R. *A field guide to genetic programming*. [Morrisville, NC: Lulu Press], 2008. OCLC: 837998350. ISBN 9781409200734.

[37] Rossi, D. *PULP Timer User Manual*. 1.0. Zurich and University of Bologna, apríl 2016.

[38] Ryan, C. a Azad, R. M. A. Sensible Initialisation in Grammatical Evolution. In: Barry, A. M., ed. *GECCO 2003: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*. Chigaco: AAAI, 11 July 2003, s. 142–145.

[39] Ryan, C., O'Neill, M. a Collins, J. *Handbook of Grammatical Evolution*. 1. vyd. Springer International Publishing AG, 2018. OCLC: 1165043791. Dostupné z: https://doi.org/10.1007/978-3-319-78717-6. ISBN 9783319787176.

[40] Spear, C. a Tumbush, G. J. *SystemVerilog for verification: a guide to learning the testbench language features*. 3rd ed. New York: Springer, 2012. OCLC: ocn753872650. ISBN 9781461407140.

[41] Wile, B., Goss, J. C. a Roesner, W. *Comprehensive functional verification the complete industry cycle*. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2005. 1. OCLC: ocm60756339. Dostupné z: https://dl.acm.org/doi/book/10.5555/1207144. ISBN 9780127518039.

[42] Zachariášová, M. a Kotásek, Z. Automation and Optimization of Coverage-driven Verification. In: *Proceedings of the 18th Euromicro Conference on Digital Systems Design*. IEEE Computer Society, 2015, s. 87–94. Dostupné z: https://www.fit.vut.cz/research/publication/10951. ISBN 978-1-4673-8035-5.

# Príloha A

# Obsah priloženého CD