VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATIZACE VERIFIKACE POMOCÍ NEURONOVÝCH SÍTÍ

AUTOMATION OF VERIFICATION USING ARTIFICIAL NEURAL NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MARTIN FAJČÍK**
AUTHOR

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

BRNO 2017

Zadání diplomové práce/18486/2016/xfajci00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Fajčík Martin, Bc.**
Obor: Inteligentní systémy
Téma: **Automatizace verifikace pomocí neuronových sítí**
Automation of Verification Using Artificial Neural Networks
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s problematikou neuronových sítí.
2. Seznamte se s procesorem Codasip uRISC.
3. Prostudujte techniky a postupy vytváření verifikačních prostředí v jazyce SystemVerilog podle metodiky UVM.
4. Navrhněte neuronovou síť pro řízení generátoru verifikačních stimulů.
5. Implementujte ji ve vhodném jazyce (SystemVerilog, Python, C++).
6. Experimentálně ověřte na modelu Codasip uRISC.

Literatura:

- Dle pokynů vedoucí.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zachariášová Marcela, Ing., Ph.D.**, UPSY FIT VUT
Konzultant: Krčma Martin, Ing., UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Úlohou tejto práce je analýza a riešenie optimalizačných problémov vychádzajúcich z automatizácie funkčnej verifikácie hardvéru pomocou umelých neurónových sietí. Verifikácia ľubovoľného integrovaného obvodu (*Design Under Verification*, DUV) pomocou techniky verifikácie riadenej pokrytím (*Coverage-Driven Verification*) a metodiky UVM (*Universal Verification Methodology*) prebieha tak, že do DUV sú zasielané vstupné stimuly, pri ktorých verifikačné prostredie monitoruje percentuálne pokrytie DUV pomocou predom určenej špecifikácie sledovaných vlastností. Pokrytím v tomto kontexte myslíme merateľnú vlastnosť DUV, ako napríklad počet overených aritmetických operácií, či počet aktivovaných riadkov kódu. Na základe dosiahnutej veľkosti pokrytia a stanovenej špecifikácie je možné prehlásiť DUV za zverifikovaný. Súčasným trendom v automatizácii funkčnej verifikácie hardvéru je pseudonáhodné generovanie vstupných stimulov s obmedzeniami (*constraints*) pomocou techniky *constrained-random stimulus generation*. V tejto práci sa preto zaoberáme ovládaním pseudonáhodného generátoru stimulov (PNG), pričom obmedzenia pre generátor sú ovládané externým prostriedkom a to konkrétne neurónovou sieťou. Využívame tak vlastnosti neurónových sietí pre riešenie optimalizačných problémov vhodné pre prehľadávanie stavového priestoru pokrytia DUV. Riešenými optimalizačnými problémami sú priebežná úprava obmedzení PNG takým spôsobom, aby došlo k čo najrýchlejšiemu zverifikovaniu DUV a hľadanie najmenšej množiny stimulov takej, že táto množina zverifikuje DUV. Kvalitatívne vlastnosti navrhnutých riešení sú overené na 32-bitových aplikačne špecifických procesoroch (ASIPs) s názvom Codasip uRISC a Codix Cobalt.

Abstract

The goal of this thesis is to analyze and to find solutions of optimization problems derived from automation of functional verification of hardware using artificial neural networks. Verification of any integrated circuit (so called *Design Under Verification*, DUV) using technique called *coverage-driven verification* and universal verification methodology (UVM) is carried out by sending stimuli inputs into DUV. The verification environment continuously monitors percentual coverage of DUV functionality given by the specification. In current context, coverage stands for measurable property of DUV, like count of verified arithemtic operations or count of executed lines of code. Based on the final coverage, it is possible to determine whether the coverage of DUV is high enough to declare DUV as verified.
Otherwise, the input stimuli set needs to change in order to achieve higher coverage. Current trend is to generate this set by technique called *constrained-random stimulus generation*. We will practice this technique by using pseudorandom program generator (PNG). In this paper, we propose multiple solutions for following two optimization problems. First problem is ongoing modification of PNG constraints in such a way that the DUV can be verified by generated stimuli as quickly as possible. Second one is the problem of seeking the smallest set of stimuli such that this set verifies DUV. The qualities of the proposed solutions are verified on 32-bit application-specific instruction set processors (ASIPs) called Codasip uRISC and Codix Cobalt.

Kľúčové slová

verifikácia, funkčná verifikácia, neurónová sieť, Hopfieldova sieť, UVM, verifikácia riadená pokrytím, optimalizačný problém, neurón, automatizácia verifikácie

Keywords

verification, functional verification, neural network, Hopfield network, UVM, coverage-driven verification, optimization problem, neuron, automation of verification

Citácia

FAJČÍK, Martin. *Automatizace verifikace pomocí neuronových sítí.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Marcela Zachariášová, Ph.D.

Automatizace verifikace pomocí neuronových sítí

Prehlásenie

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

.....
Martin Fajčík
24. mája 2017

Poďakovanie

Ďakujem všetkým, ktorý ma podporovali nielen v písaní tejto práce, ale aj životnej ceste, ktorej kapitolu táto práca uzatvára. Touto cestou by som chcel poďakovať mojej rodine a priateľom, ktorí mi umožnili stať sa kým som a neustále zo mňa robia lepšieho človeka, Ing. Marcele Zachariášovej Ph.D. za vedenie mojej diplomovej práce a všetky jej rady a mojim kolegom z Codasipu, ktorí so mnou spolupracovali.

Obsah

Kapitola 1

Úvod

V súčasnosti sa ľudia pohybujú vo svete plnom zariadení, ktoré definujú ich životný štýl. Každým rokom dochádza k rozvoju integrácie vstavaných systémov (anglicky *Embedded System*, skratkou *ES*) vyvinutých na mieru do okolitých zariadení. Od týchto elektronických systémov sa očakáva, že budú plniť svoje funkcie tak dokonale, ako je to len možné. Dnešným trendom je realizovať ES ako systém na čipe (anglicky *System on Chip*, skratka *SoC*), ktorý býva implementovaný na technológiách typu hradlové pole , ako FPGA (*Field Programmable Gate Array*), či v podobe pevného integrovaného obvodu (anglicky *Integrated Circuit*, skratkou *IC*) ako ASIC (*Application-Specific Integrated Circuit*). Mnoho príkladov vstavaných systémov môžeme nájsť napríklad v aute. Sú to systémy ako ABS, elektronický stabilizačný systém (ESC) či systém pre reguláciu preklzovania (TCS).

Návrh vstavaného systému prebieha v niekoľkých etapách. Typicky sa jedná o špecifikáciu, formuláciu a dekompozíciu problému, osobitný návrh hardvéru (*HW*) a softwaru (*SW*), systémovú integráciu, verifikáciu a testovanie. Tento postup návrhu sa môže líšiť v závislosti od realizovaného projektu, avšak vzhľadom nato, že dodatočná oprava chýb môže byť v hardvéri veľmi obtiažna a sú kladené nároky na vysokú spoľahlivosť týchto systémov, veľký dôraz sa dáva práve na etapu zaisťujúcu korektnosť navrhnutých obvodov – *verifikáciu*.

1.1 Motivácia

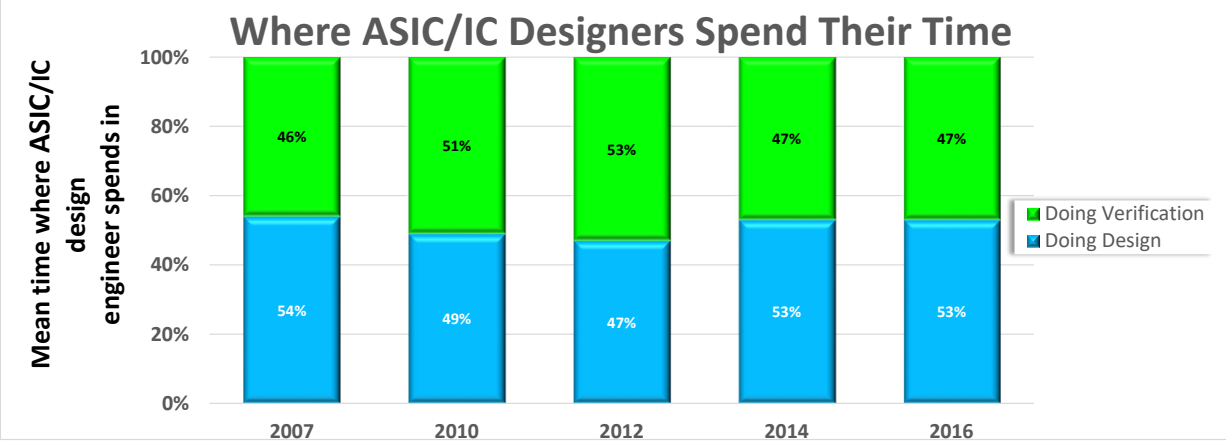
Existuje niekoľko prístupov pre verifikáciu počítačového hardvéru [16]:

- `Ladenie cez simuláciu` (*Bug Hunting*) – Ladenie v simulátore IC sa uplatňuje najmä v skorých fázach jeho návrhu. Tento prístup je však neúplný a pri rozsahu dnešných projektov zvyčajne slúži len na overenie základnej funkcionality.
- `Formálna analýza a verifikácia` – V praxi sú využívané dve varianty [4]:
  - `Kontrola formálnych vlastností` (*Formal Property Checking*) – Verifikovaný obvod (anglicky *Design Under Verification*, skratka *DUV*) je vyjadrený pomocou výrazov temporálnej logiky (CTL, CTL*, LTL, PLTL), ktoré popisujú všetky možné stavy, do ktorých sa dokáže DUV dostať a následne sa overí ich pravdivosť systematickým preskúmaním stavového priestoru. Keďže sa pri tomto prístupe v podstate tvorí formálny model DUV, v angličtine sa alternatívne nazýva aj ako *Model Checking*.

  - `Kontrola rovnosti` (*Equivalence Checking*) – DUV je porovnávaný s referenčným modelom (anglicky nazývaný aj *Golden Model)*, pričom cieľom tohto prístupu je dokázať, že chovanie DUV je ekvivalentné chovaniu referenčného modelu.

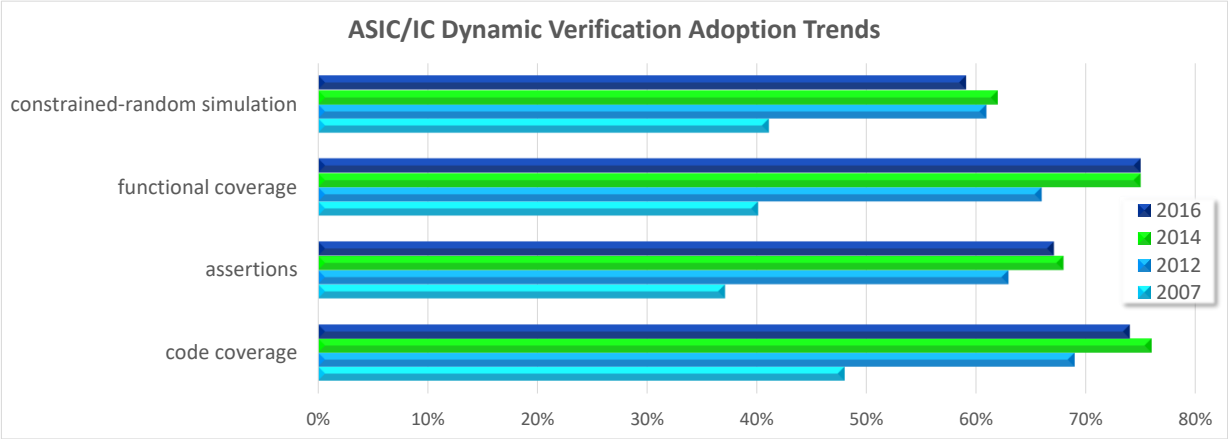
- `Funkčná verifikácia` – Zvyčajne sa jedná o simuláciu DUV a verifikačného prostredia (niekedy označovaného ako *Testbench*), ktoré môžu byť reprezentované na rôznych úrovniach popisu, typicky na úrovni registrov (anglicky *Register-Transfer Level*, skratkou *RTL*) alebo na úrovni brán (*Gate Level*, skratkou *GL*). Týmto typom verifikácie sa budeme v rámci tejto práce ďalej zaoberať.

Podľa nezávislej štúdie výskumnej skupiny Wilson Research Group [8] (podporovanej spoločnosťou Mentor) bol v rokoch 2007 - 2016 čas strávený inžiniermi na funkčnej verifikácii HW a na návrhu systému približne rovnaký (viď Obr 1.1).



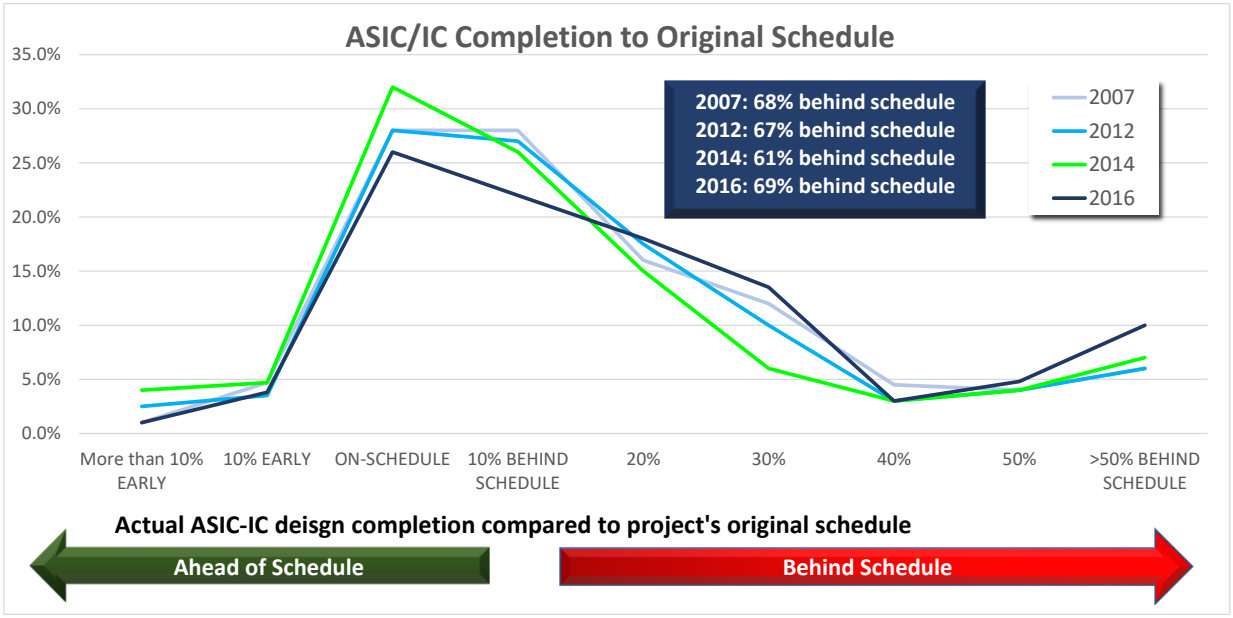
Obr. 1.1: *Pomer času stráveného návrhom a verifikáciou integrovaných obvodov podľa [8].*

Funkčná verifikácia je teda v praxi časovo náročnou etapou vývoja hardvéru. Preto je zvyčajne realizovaná použitím vhodných systematických techník ako verifikácia riadená pokrytím (*Coverage-Driven Verification*), verifikácia založená na tvrdeniach (*Assertion-Based Verification*) alebo pseudonáhodné generovanie stimulov s obmedzeniami (*Constrained-Random Stimulus Generation*), ktoré majú jej priebeh urýchliť a pritom zachovať kvalitu tejto etapy. Popularitu týchto techník je možné vidieť na obrázku 1.2.



Obr. 1.2: *Použitie rôznych verifikačných techník pri návrhu IC podľa [8].*

Súčasný trh s elektronikou je vysoko citlivý na čas, ktorý spoločnostiam zaberie vývoj ich nových produktov (anglicky nazývaný *Time to Market*). Naneštastie odhad tohto vývoja je veľmi náročný pri projektoch tohto typu dochádza často k omeškaniu, čo dokazuje aj obrázok 1.3.



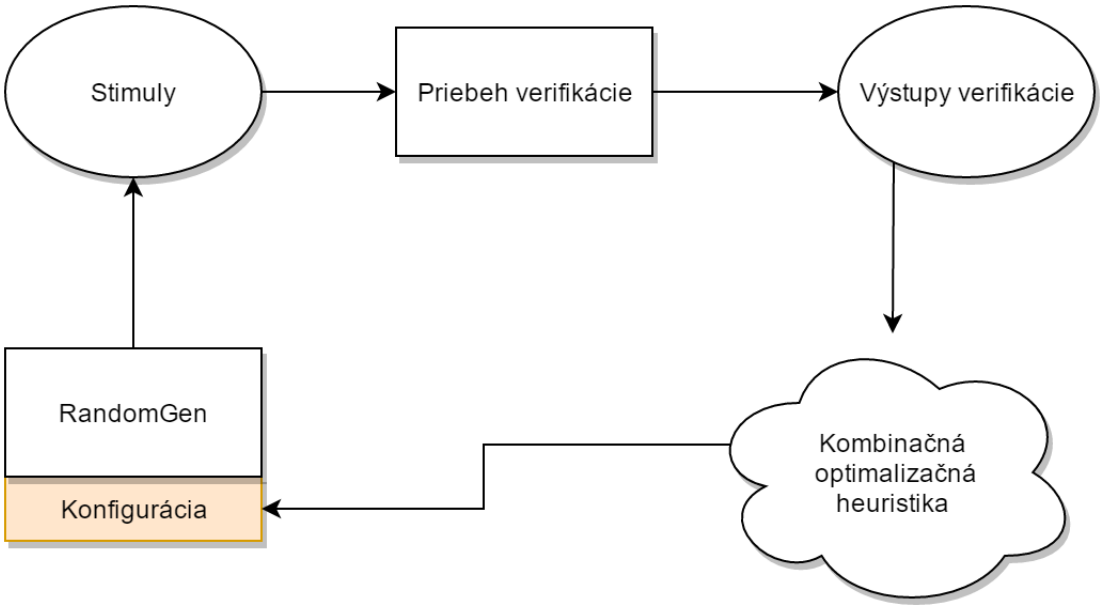
Obr. 1.3: *Vzťah doby vývoja ASIC/IC projektov vzhľadom k ich pôvodnému plánu. Zdroj [8].*

Preto sa v rámci tejto práce pomocou prostriedkov umelej inteligencie (neurónových sietí) a následnou automatizáciou vybraných verifikačných techník zameriavame na zvýšenie efektivity funkčnej verifikácie a skrátenie časového intervalu, ktorý inžinieri strávia verifikáciou IC projektov.

1.2 Popis problému

V spoločnosti Codasip, ktorá sa zaoberá tvorbou procesorov na mieru (alternatívne nazývané ako procesory s aplikačne špecifickou inštrukčnou sadou, anglicky *Application-Specific Instruction Set Processors*, skratkou ASIPs, ďalej tiež nazývané aj ako *jadro*, resp. jadrá v pl.) zostrojil výskumný tým pseudonáhodný generátor (PNG, alternatívne nazývaný *RandomGen*), ktorý umožňuje na základe stanovenej konfigurácie pseudonáhodne vygenerovať vstupné stimuly pre verifikovaný ASIP.

Stimulmi sú v tomto prípade programy v asembleri kompatibilné s inštrukčnou sadou verifikovaného jadra. Po aplikácií týchto stimulov na jadro je v prostredí RTL simulátoru možné pozorovať priebežné chovanie verifikovaného jadra, či rôzne parametre ako celkové pokrytie funkcií ASIP a zdrojového kódu (RTL) jadra pomocou vygenerovaných stimulov.



Obr. 1.4: *Grafické znázornenie problému riešeného touto prácou.*

Úlohou tejto práce je vytvoriť techniky optimalizácie konfigurácie ($conf^*_{PNG}$) nástroja RandomGen (viď 1.4), pomocou kombinačnej optimalizačnej heuristiky. Riešenie tejto úlohy v sebe zahŕňa niekoľko rôznych prístupov, z ktorých formulujeme dva nasledujúce optimalizačné problémy:

- Prvým problémom je hľadanie priebežných zmien $conf^*_{PNG}$ tak, aby v čo najkratšom čase bolo dosiahnuté čo najvyššie pokrytie ASIP. Praktické využitie riešenia tohto problému môžeme nájsť v rýchlej a adaptabilnej verifikácii akéhokoľvek jadra s dostupným verifikačným prostredím.

- Druhým problémom je hľadanie najmenšej množiny programov vygenerovanej pomocou rôzne nastavených $conf^*_{PNG}$ takej, aby dosiahla čo najvyššie pokrytie ASIP. Tento problém umožňuje v praxi udržiavanie stabilných a teoreticky bezchybných jadier pomocou minimálnej verifikačnej sady stimulov, ktorá je pre každé jadro vytvorená samostatne.

Kombinačnou optimalizačnou heuristikou použitou na nájdenie tejto konfigurácie je v tejto práci Hopfieldova neurónová sieť a jej modifikácie. Funkcionalita tejto heuristiky je následne experimentálne overená na jadrách navrhnutých spoločnosťou Codasip.

1.2.1 Optimalizačné problémy

Optimalizačné problémy tvoria širokú triedu spravidla výpočtovo náročných problémov. Hľadaním ich riešenia sa zaoberá výskumná oblasť zvaná kombinačná optimalizácia (*Combinatorial Optimization*). V takýchto problémoch sa snažíme optimalizovať (teda maximalizovať alebo minimalizovať) nejakú kvantitu, pričom sa snažíme o zachovanie platnosti logických obmedzení.

Formálnejšie, optimalizačný problém (OP) môžeme definovať ako dvojicu (f, C), kde $f(x_1, x_2, ..., x_n)$ je tzv. objektívna funkcia a C je množina obmedzení (teda logických formúl) definičného oboru premenných funkcie f. Cieľom je nájsť také ohodnotenie premenných $x_1, x_2, ..., x_n$, ktoré vedie k optimálnej hodnote takejto funkcie (minimum alebo maximum), pričom toto ohodnotenie zachováva platnosť všetkých obmedzení. OP môžeme rozdeľovať z rôznych hľadísk.

- Na základe požadovanej optimálnej hodnoty môžeme rozlišovať medzi maximalizačným a minimalizačným OP.
- V závislosti od definičného oboru premenných $x_1, x_2, ..., x_n$ vo funkcii f rozlišujeme medzi spojitou a diskrétnou optimalizáciou.
- S pohľadu linearity f rozlišujeme medzi lineárnymi a nelineárnymi OP.

Hľadanie optimálnej hodnoty (alebo hodnôt) OP je pre ich zložitosť (veľká časť z nich je NP-ťažká) pomocou konvenčných prístupov v praxi na súčasných výpočtových zariadeniach typicky veľmi náročné. Oblasť kombinačnej optimalizácie sa preto zaoberá návrhom heuristík umožňujúcich aspoň ich približné riešenie [19].

1.3 Obsah práce

Diplomová práca pozostáva z teoretickej a praktickej časti. Cieľom teoretickej časti je oboznámenie čitateľa s formálnymi modelmi neurónových sietí, ich základnými princípmi a predovšetkým ich významom v oblasti riešenia optimalizačných problémov (kapitola 2) a úvod do teórie súvisiacej s funkčnou verifikáciou, jej významom, terminológiou či spôsobmi jej implementácie podľa metodiky UVM v jazyku SystemVerilog (kapitola 3).

Poznatky získané v teoretickej časti následne aplikujeme v praktickej časti. Tá pozostáva z analýzy nástrojového prostredia použitého pri riešení tejto práce (kapitola 4) a jeho mapovania na sieť (podkapitola 4.4), z návrhu modelov neurónových sietí (kapitola 5) a z popisu riadiacich algoritmov a utilít využitých v implementácii riešenia (kapitola 6). V závere tejto časti analyzujeme efektivitu navrhnutého riešenia v kapitole Experimenty (7). Súhrn poznatkov získaných z experimentálnej časti práce a návrh rozšírení diplomovej práce je následne obsahom záverečnej kapitoly 8.

Kapitola 2

Umelé neurónové siete

Neurónové siete (v kognitívnej vede[1] nazývané ako *"konekcionizmus"*) v súčasnosti tvoria dôležitú rolu v oblasti počítačovo orientovanej umelej inteligencie, kde zaujali postavenie *univerzálneho matematicko-informatického prístupu* k štúdiu a modelovaniu procesov učenia založenom na metafore učenia ľudského mozgu. Okrem umelej inteligencie majú neurónové siete nezastupiteľné uplatnenie aj v kognitívnej vede, lingvistike, neurovede (*neuroscience*), prírodných a spoločenských vedách, kde sa s ich pomocou modelujú nielen procesy učenia a adaptácie, ale aj široké spektrum rôznorodých problémov klasifikácie objektov a tiež problémov riadenia komplexných priemyselných systémov.

Univerzálnym matematicko-informatickým prístupom v tomto kontexte sa rozumie skutočnosť, že neurónové siete majú tri nasledujúce vlastnosti:

- Neurónová sieť s dopredným šírením signálu a s aspoň jednou skrytou vrstvou je *univerzálny aproximátor*[15]. To znamená, že ľubovoľná kognitívna aktivita, ktorá sa dá vyjadriť pomocou tabuľky obsahujúcej dvojice vstup/požadovaný výstup, sa dá vypočítať pomocou umelej neurónovej siete (tabuľka v podstate predstavuje tréningovú množinu siete).
- Rekurentná neurónová sieť (anglicky *recurrent neural network*, skratkou *RNN*) je schopná simulovať *deterministický konečný automat* [2]. Táto vlastnosť RNN znamená, že sú schopné klasifikovať konečné reťazce znakov.
- Idealizované neurónové siete sú buď *turingovsky ekvivalentné* alebo majú *super-turingovskú výpočtovú silu* [28].

2.1 História

Dejiny konekcionizmu je možno rozlíšiť do dvoch etáp [18]. Prvou etapou bolo tzv. *obdobie klasického konekcionizmu*. Zavŕšením tejto etapy bola publikácia W. McCullocha (neurovedec) a W. Pittsa (logik) [21], v ktorej boli po prvýkrát popísané **logické neuróny** a bolo v nej preukázané, že neurónové siete sú efektívnym výpočtovým prostriedkom v doméne Boolových funkcií – ľubovoľnú Boolovu funkciu je možné simulovať pomocou neurónovej siete obsahujúcej logické neuróny. Zaujímavosťou je, že táto práca je z pohľadu dnešného

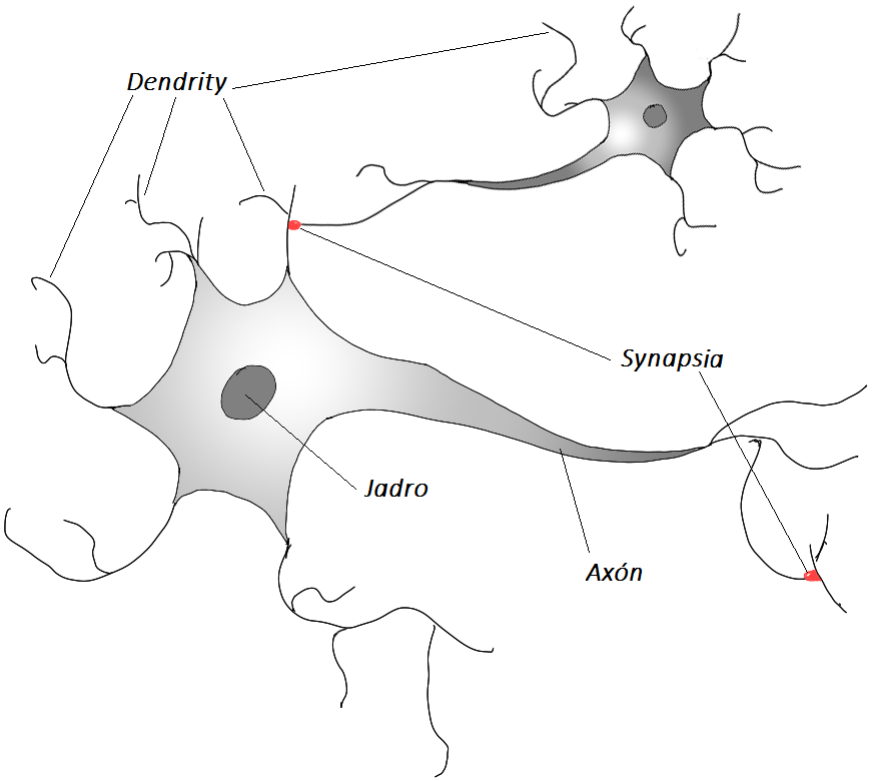
[1] Kognitívna veda je vedou interdisciplinárneho charakteru. Leží na rozhraní neurovedy, psychológie, lingvistiky a umelej inteligencie. Hlavným cieľom kognitívnej vedy je vysvetliť kognitívne procesy ľudskej mysli pomocou teoretických predstáv, ktoré sú plauzibilné so súčasnými predstavami neurovedy, psychológie a umelej inteligencie [18].

čitateľa veľmi ťažko pochopiteľná, keďže Walter Pitts bol v oblasti logiky a matematiky samoukom. Práca bola v oblasti vedy vyzdvihnutá až v druhej polovici 50. rokov minulého storočia, keď bola "preložená" zásluhou S.C. Kleeneho a N. Minskeho.

Druhou etapou bolo *obdobie počítačového konekcionizmu*. Tá započala popisom perceptrónu v práci Rosenblatta [24][25], avšak jej dnes kľúčovými časťami sú práce D. Rumelharta, G. Hintna a R. Williamsa zaoberajúce sa učením neurónových sietí so skrytými vrstvami a hodnotením ich chyby [26][27], ktoré výrazne zdokonalili schopnosť počítačového modelovania procesov schopných učiť sa, práca D.S. Broomheada a D. Lowea, ktorí svetu predstavili použitie radiálnych bázových funkcií v neurónových sietiach [3], či práca J.J. Hopfielda [13], ktorý po prvý raz popísal siete schopné efektívne riešiť optimalizačné problémy v reálnom čase, ktorými sa budeme v tejto práci zaoberať.

2.2 Model umelého neurónu

Prvým modelom umelého neurónu bol logický neurón spomenutý v predchádzajúcej kapitole. Jeho koncept bol odvodený od tvaru a funkcie biologickej neurónovej bunky (Obr. 2.1).



Obr. 2.1: *Schéma biologického neurónu. Dendrity sú tkanivo, ktoré je vstupom neurónu. Na rôzne časti dendritov sa pripájajú výstupné tkanivá iných neurónov, tzv. axóny. Miesta, v ktorých dochádza k spojeniu medzi axónom a dendritom sú zvané synapsie alebo synaptické spojenia. V prípade biologického neurónu takýchto spojení existujú desaťtisíce [29].*

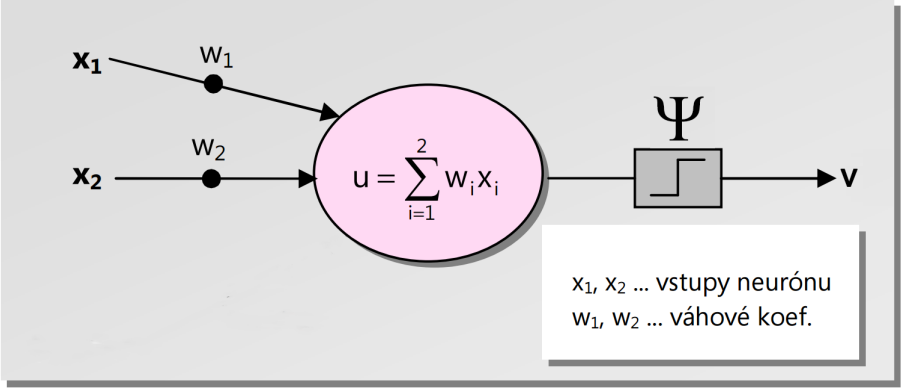
Prenos signálu cez neuróny je zložitý elektrochemický proces. Nervové impulzy vo forme akčných potenciálov sú dopravené do synapsií, odkadiaľ prechádzajú do dedrít, kde vzniká

elektrická reakcia. Tá môže byť buď zosilujúca (excitačná), alebo utlmujúca (inhibičná) v závislosti od povahy membrán u dendrít (tie sa nachádzajú v spojoch so synapsiami), tzv. synaptických váh. Vo vnútri tela prijímajúcej bunky sa tak kumuluje elektrický potenciál. Ak tento potenciál presiahne určitú úroveň, bunka sa prebudí (excituje) a vypudí elektrický signál do axónu [29].

Logický neurón (Obr. 2.2) pracuje k tomuto biologickému modelu analogicky. Zakončenia dendritov predstavujú vstupný vektor neurónu. Každý vstup má svoju vlastnú synaptickú váhu, ktorá určuje či sa jedná o inhibičný (typicky váha s hodnotou -1) alebo excitačný (typicky váha s hodnotou 1) vstup. Vnútorný potenciál neurónu u sa kumuluje pomocou sumy excitačných a inhibičných vstupov, a ak je jeho rozdiel s prahovou hodnotou θ nezáporný, je generovaný binárny výstup 1, inak 0, formálne uvedené vzťahmi 2.1 a 2.2.

$$u = \xi(\vec{x}, \vec{w}) = \sum_{i=0}^{|\vec{x}|} w_i x_i \tag{2.1}$$

$$\Psi(u) = \begin{cases} 0 & u \leq \theta \\ 1 & u > \theta \end{cases} \tag{2.2}$$

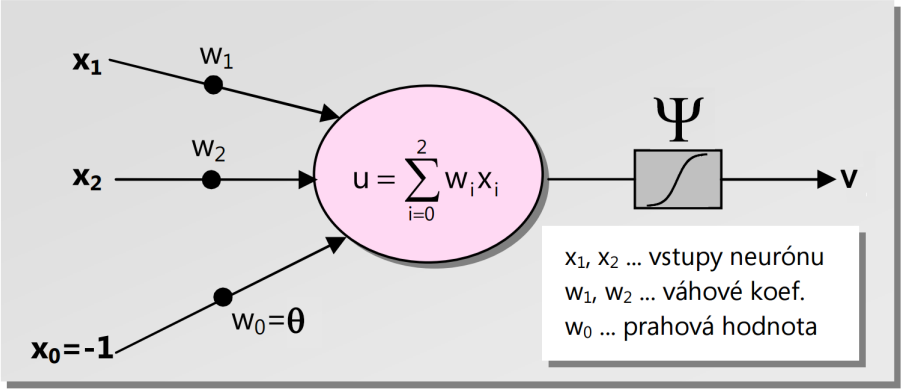


Obr. 2.2: *Schéma logického neurónu McCulloch-Pittsa.*

Funkciu, ktorá sa stará o kumuláciu potenciálov (ξ) nazývame **bázová** funkcia. Funkciu, ktorá rozhoduje, či sa neurón aktivuje (Ψ) nazývame **aktivačná** (niekedy v literatúre označovaná aj ako *prenosová*) funkcia.

Následníkom v oblasti tohto modelu je umelý neurón popísaný F. Rosenblattom v päťdesiatych rokoch zvaný perceptrón [24]. U McCulloch-Pittsovho neurónu bola predstavená teória, že ich model neurónu v princípe dokáže spočítať akúkoľvek aritmetickú či logickú funkciu. V ich práci však nebola vypracovaná žiadna tréningová metóda. Perceptrón prichádza s tréningovou metódou, ktorá po prvýkrát umožňuje rozpoznávať známe aj neznáme vzory. Naviac dokazuje, že pokiaľ existujú váhy, ktoré tento problém rieši, potom k nim táto metóda konverguje. Nadšenie z perceptrónu však začalo upadať, keď sa zistilo, že takýto model vie riešiť iba lineárne separovateľné úlohy (nezvládne tak napr. funkciu XOR). F. Rosenblatt sa snažil tento problém vyriešiť, ale nepodarilo sa mu to. Rozlúštiť sa ho podarilo až o 25 rokov neskôr P. Werbosovi [32] doplnením BIAS vstupu, ktorý umožnil posun prahovej hodnoty u aktivačnej funkcie na 0 a zámenou skokovej aktivačnej funkcie za spojitú (zvyčajne *sigmoid* (2.3) alebo *hyperbolický tangens*), čo umožnilo na sieť aplikovať metódy pre zostup po gradiente.

$$\Psi(u) = \frac{1}{1 + e^{-\lambda u}} \tag{2.3}$$



Obr. 2.3: *Schéma logického neurónu P. Werbosa. λ je koeficient sigmoidy, ktorý ovplyvňuje jej strmosť. V prípade že $\lambda \to \infty$ aktivačná funkcia nadobúda charakter skokovej funkcie.*

Týmto modelom sa inšpiroval aj J.J. Hopfield pri návrhu Hopfieldovej siete, ktorá zohráva významnú úlohu pri riešení asociačných a optimalizačných úloh.

2.3 Hopfieldova sieť

2.3.1 Základný model

Hopfieldova sieť je plne prepojená[2] rekurentná[3] neurónová sieť. Prepojenia medzi neurónmi sú vážené, i.e. pre každé dva neuróny i, j existuje váha ich prepojenia w_{ij} (2.4). Diagonálne váhové koeficienty v modeli sú nulové (2.5) a nediagonálne váhové koeficienty sú symetrické (2.6) (pre príklad viď Obr. 2.4). Formálne, nech N je množina neurónov Hopfieldovej siete, potom v modeli platí:

$$\forall i, j \in N : \exists w_{ij} \tag{2.4}$$

$$\forall i \in N : w_{ii} = 0 \tag{2.5}$$

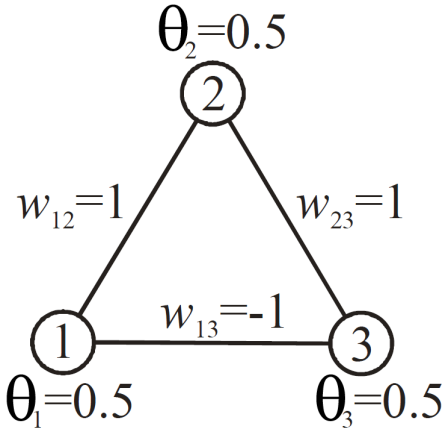
$$\forall i, j \in N : w_{ij} = w_{ji} \tag{2.6}$$

Každý neurón i je charakterizovaný jeho:

- **výstupom** (v literatúre nazývané aj *stavom*) v_i
- **vstupom** (v literatúre nazývané aj *aktiváciou*) u_i
- **prahom** θ_i

[2]Plne prepojenou neurónovou sieťou označujeme sieť, v ktorej je každý neurón prepojený s každým. Inak povedané, graf neurónovej siete je úplný.

[3]Rekurentná neurónová sieť je každá neurónová sieť so spätnou väzbou. Inak povedané, graf neurónovej siete obsahuje cyklus.



Obr. 2.4: *Príklad Hopfieldovej neurónovej siete. Diagonálne spojenia s váhou 0 na nákrese uvedené nie sú.*

Keďže sa jedná o rekurentnú sieť, ktorej stav sa v priebehu výpočtu mení, je nutné aby obsahovala časový model. Preto sú vstupy $u_i^{(t)}$ a výstupy $v_i^{(t)}$ modelu parametrizované premennou $t, t \in \mathbb{N}$, ktorá modeluje výpočtový čas.

Vstupom $u_i^{(t)}$ každej jednotky i v čase t sú výstupné hodnoty ostatných neurónov a hodnoty synaptických váh na spojoch s nimi. Výstupom každej jednotky i je v čase t je hodnota $v_i^{(t)}$. Celkový výstup (*stav*) siete v čase t je potom daný vektorom[4] 2.7.

$$\vec{v}^{(t)} = v(v_1^{(t)}, v_2^{(t)}, ..., v_n^{(t)}) \tag{2.7}$$

Spôsob (poradie výberu, tvar funkcií), akým sa ich výstup aktualizuje je definovaný tzv. *dynamikou siete*. Vo všeobecnom modeli je aktualizácia výstupu neurónu i stanovená ako je uvedené v rovnici[5] 2.8,

$$v_i^{(t+1)} = \Psi\big(u_i^{(t)}\big) = \Psi\big(\xi_i(\vec{v}^{(t)}, \vec{w}, \theta_i)\big) = \Psi\Big(\sum_{j=1}^{n} w_{ij} v_j^{(t)} + \theta_i\Big) \tag{2.8}$$

kde n je počet neurónov siete ($n = |N|$).

Ako už bolo naznačené v kapitole 2.2, aktivačná funkcia Ψ môže naberať niekoľko podôb. V prípade, že chceme modelovať diskrétnu Hopfieldovu sieť, musíme použiť skokovú aktivačnú funkciu. Používa sa buď *binárna funkcia* (*Heaviside Step Function*) 2.9,

$$Bin(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{2.9}$$

[4] Počiatočný stav siete sa zvyčajne generuje náhodne. To umožní odlišnú evolúciu riešenia pri opätovnom štarte siete.

[5] Je dôležité poznamenať, že v tejto práci sa budeme na prahovú hodnotu θ_i pozerať ako na záporné číslo pripočítané ku kumulovanému vstupu (oproti kladnému číslu vynásobenému pomocou váhy -1, ako to bolo uvedené u P. Werbosovho modelu s BIAS v kapitole 2.2)

alebo *znamienková funkcia* 2.10.

$$Sign(x) = \begin{cases} -1 & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{2.10}$$

V prípade, že chceme modelovať analógovú Hopfieldovu sieť, používa sa buď *sigmoida*[6] (v literatúre tiež nazývaná ako *logistická funkcia*) 2.11,

$$g_\lambda(x) = \frac{1}{1 + e^{-2\lambda x}} \tag{2.11}$$

prípadne funkcia pre *hyperbolický tangens* 2.12.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.12}$$

Všimnime si, že platia vzťahy 2.13.

$$\begin{aligned} Sign(x) &= 2Bin(x) - 1 \\ tanh(x) &= 2g_\lambda(x) - 1 \end{aligned} \tag{2.13}$$

Predstavený model s rôznymi variantami jeho dynamiky je deterministický v zmysle jeho stavovej logiky, teda že aktuálny stav siete je explicitne daný funkciou predošlého stavu siete. Existujú však aj stochastické Hopfieldove siete, ktorých stav sa odvíja od pravdepodobnostnej distribúcie, napríklad od Boltzmannovej či Cauchyho (viď kapitola 2.3.5).

U špecifikovaného modelu sa tiež neuvádza systém, ktorým dochádza k aktualizácii stavov. Používajú sa dva typy systémov:

- **Synchrónny systém** – Počas jedného kroku v čase dochádza k aktualizácii všetkých neurónov v sieti.
- **Asynchrónny systém** – V priebehu jedného kroku dochádza k aktualizácii jedného neurónu v sieti. Výber vhodného neurónu sa robí buď sekvenčne, alebo náhodne.

2.3.2 Energetická funkcia

Aby sme pomocou definovanej Hopfieldovej siete mohli riešiť optimalizačné problémy, chýba nám ešte funkcia, ktorá plní úlohu Ljapunonovej funkcie z kvalitatívnej teórie diferenciálnych rovníc – objektívna funkcia. V teórii neurónových sietí sa táto funkcia zvyčajne nazýva ako energetická funkcia. Ďalej uvažujme, že cieľom tejto funkcie bude hľadanie globálneho minima. Hopfieldova sieť s energetickou funkciou sa bude chovať tak, že po vygenerovaní nového stavu siete sa spočíta hodnota energie siete. Nový stav siete je prijatý vtedy a len vtedy, ak je jeho energia nižšia než jeho súčasná energia.

[6] Oproti sigmoide z minulej kapitoly na obrázku 2.3 sme pridali konštantu -2 kvôli tomu, aby nám vznikol v rovnici 2.13 funkčný vzťah vzhľadom k funkcii hyperbolického tangensu analogický k tomu u diskrétnych Hopfieldových sietí.

Už z tohto konceptu vyplýva, že použitie Hopfieldových sietí týmto spôsobom dáva zmysel iba v prípade, že uvažujeme siete s asynchrónnym systémom aktualizácie stavov[7]. Pre jednoduchosť ďalej tiež uvažujme diskrétnu Hopfieldovu sieť s binárnou aktivačnou funkciou.

Nech priestor stavov neurónovej siete je označený ako $S = \{0,1\}^n$, potom *trajektória riešení* s počiatočným stavom $\vec{v}^{(1)}$ je tvorená postupnosťou stavov $\vec{v}^{(1)}, \vec{v}^{(2)}, \vec{v}^{(3)}, ..., \vec{v}^{(t)}, \vec{v}^{(t+1)}, ...$ pričom stav $\vec{v}^{(t)}$ vznikol v čase t z predchádzajúceho stavu $\vec{v}^{(t-1)}$ pomocou asynchrónnej aktualizácie použitím vzťahu 2.8.

Nech energia Hopfieldovej siete pre binárny stav $\vec{v}^{(t)} \in \{0,1\}^n$ je definovaná ako 2.14.

$$E(\vec{v}) = -\frac{1}{2}\sum_{jk,j\neq k}^{n} v_j v_k w_{jk} + \sum_{j=1}^{n} v_j \theta_j \tag{2.14}$$

Dosadením dvoch stavových vektorov, ktoré sa líšia v aktivácii i-tého neurónu $\vec{v}^{(t)} = v(v_1, v_2, , ..., v_i, ..., v_n), \vec{v}^{(t+1)} = v(v_1, v_2, , ..., v_i^{'}, ..., v_n)$ vieme odvodiť vzťah 2.15, ktorý vyjadruje energetický rozdiel pri aktivácii i-tého neurónu ΔE_i.

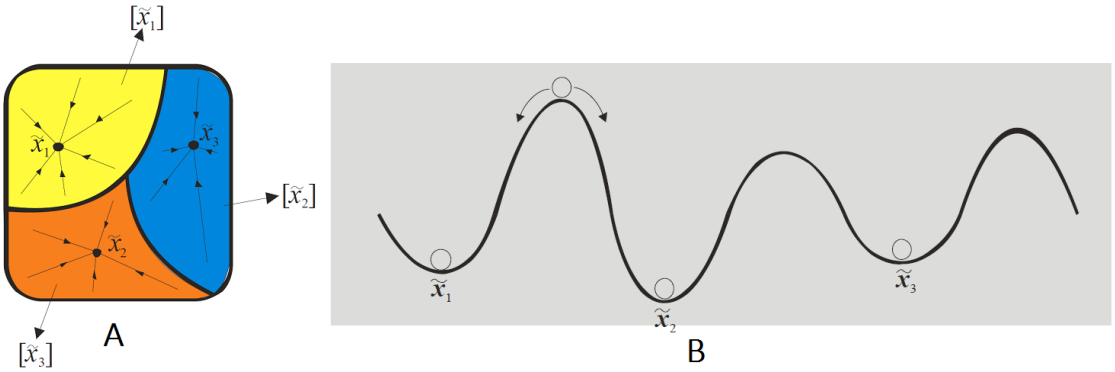
$$\Delta E_i = -\Big(\sum_{j=1}^{n} w_{ij} v_j + \theta_i\Big)\Delta v_i \tag{2.15}$$

Nech $\vec{v}^{(1)}, \vec{v}^{(2)}, \vec{v}^{(3)}, ..., \vec{v}^{(t)}, \vec{v}^{(t+1)}, ...$ je trajektória stavov. Pre každú trajektóriu musí vždy platiť, že je pre ňu funkcia E nerastúca, formálne formula 2.16.

$$\forall t \in \mathbb{N} : E(\vec{v}^{(t)}) \leq E(\vec{v}^{(t-1)}) \tag{2.16}$$

Za dodržania platnosti formúl 2.14, 2.15 a 2.16 by funkcia E mala konvergovať k jednému z riešení a v ideálnom prípade dostať sa do ustáleného stavu (Obr. 2.5) s nulovou energiou siete. Riešenie však tiež môže uviaznuť v lokálnych minimách či oscilovať medzi ustálenými stavmi. Sieť, ktorá dospela do takéhoto stavu, budeme nazývať *zaseknutá sieť*.

[7] V prípade synchrónnych sietí by sa musel výpočet zastaviť pri prvom nesplnení podmienky nerastúcej energie, keďže neexistuje žiadna ďalšia alternatíva pre postup výpočtu. Pravdepodobnosť zaseknutia sa v lokálnom minime by v tomto prípade bola omnoho vyššia než u asynchrónnej siete, kde sa môžeme po neúspechu pokúsiť vygenerovať nový stav výpočtov výstupu pre ďalší/náhodný neurón.



Obr. 2.5: *Premenné $\tilde{x}_1$, $\tilde{x}_2$, $\tilde{x}_3$ predstavujú ustálené stavy (equilibriá), do ktorých dospela trajektória Hopfieldovej siete. Obrázok (A) znázorňuje rozklad priestoru stavov $S = \{0,1\}^n$ na bazény príťažlivosti miním energie $E(\vec{v})$. Ak je trajektória zahájená v určitom bazéne príťažlivosti, potom aj celá trajektória leží v tomto bazéne. Obrázok (B) znázorňuje priestor funkcie energie a jeho lokálne minimá, z ktorých jedno môže byť charakterizované ako globálne minimum. Prevzaté z [18].*

2.3.3 Príklad Hopfieldovej siete

Uvažujme nasledujúcu Hopfieldovu sieť z obrázka 2.4. Sieť obsahuje tri neuróny, v ktorých sú nasledujúce váhové a prahové koeficienty

$$\mathbf{W} = \begin{pmatrix} 0 & 1 & -1 \\ 1 & 0 & 1 \\ -1 & 1 & 0 \end{pmatrix}, \boldsymbol{\theta} = (0.5, 0.5, 0.5)$$

a funkcia energie má nasledujúci tvar:

$$E(\vec{v}) = -0.5(v_1v_2 - v_1v_3 + v_2v_3) + (0.5v_1 + 0.5v_2 + 0.5v_3)$$

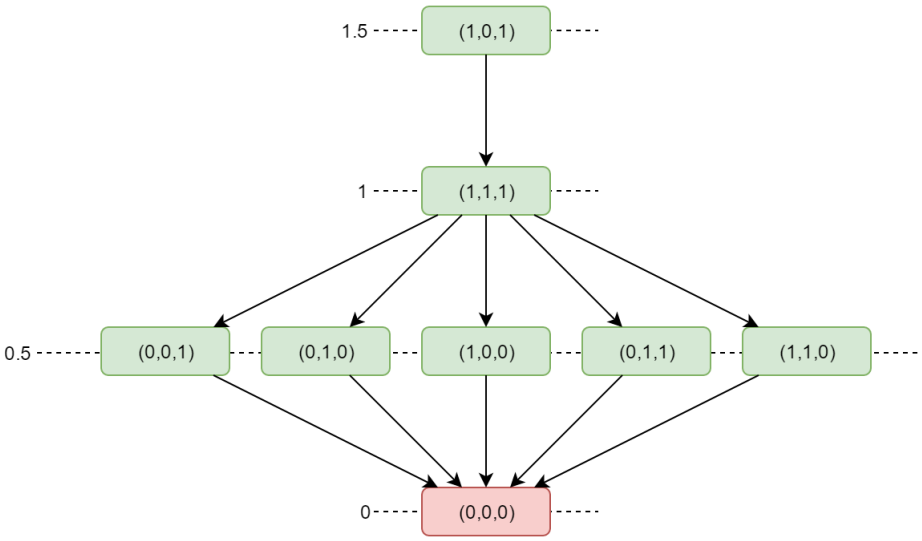
Možné hodnoty stavového vektoru spolu s energiou siete sú uvedené v tabuľke 2.1.

stavový vektor $\vec{v}$		$E(\vec{v})$
$\vec{v}_1$	$(0,0,0)$	0
$\vec{v}_2$	$(0,0,1)$	0.5
$\vec{v}_3$	$(0,1,0)$	0.5
$\vec{v}_4$	$(0,1,1)$	0.5
$\vec{v}_5$	$(1,0,0)$	0.5
$\vec{v}_6$	$(1,0,1)$	1.5
$\vec{v}_7$	$(1,1,0)$	0.5
$\vec{v}_8$	$(1,1,1)$	1

Tabuľka 2.1: *Funkčné hodnoty energie pre stavy z $S = \{0,1\}^3$*

.

Z tabuľky môžeme vytvoriť stavový diagram, viď obrázok 2.6.



Obr. 2.6: *Stavový priestor prehľadávaný funkciou E. Vlavo od prerušovaných čiar je znázornená veľkosť energie pre stavy na úrovni grafu. Červenou farbou je znázornený stav s najnižšou energiou.*

Prehľadávanie priestoru môže začať v ktoromkoľvek stave, keďže počiatočný stav siete je náhodne generovaný. Skončiť by však vždy malo v niektorom z lokálnych miním, ideálne v globálnom minime ako je to v tomto príklade. Takéto chovanie siete ukazuje, že sieť dokáže riešiť optimalizačné problémy, ako napríklad rozpoznávanie písmen (príklad na funkciu asociatívnej pamäti), či riešenie TSP[8], ako preukázal aj J.J. Hopfield a D.W. Tank [14].

2.3.4 Mapovanie problému na sieť

Ideou pre riešenie optimalizačného problému (OP) na Hopfieldovej sieti je jeho zakódovanie do základného modelu siete. OP najskôr zakódujeme do energetickej funkcie a to tak, že zachováme jej vlastnosti, tzn. že pre akúkoľvek trajektóriu $\vec{v}^{(1)}, \vec{v}^{(2)}, \vec{v}^{(3)}, ..., \vec{v}^{(t)}, \vec{v}^{(t+1)}, ...$ evolúcie siete bude energetická funkcia nerastúca. Následne vhodne nastavíme parametre siete (počet neurónov, synaptické váhy, prahy, spôsob aktualizácie neurónov a dynamiku siete) tak, aby reflektovali charakter problému. Potom môže byť sieť náhodne inicializovaná do nejakého stavu, z ktorého necháme výpočet bežať až kým nedosiahne stav equilibria (energetická funkcia prestane klesať, alebo klesá veľmi pomaly), ktoré predstavuje nájdené optimum.

Konštrukcia vlastnej energetickej funkcie môže byť náročná. Základným obmedzením, ktoré je nutné dodržať je, že musí byť kvadratická, tak ako je to v prípade 2.14. Najčastejší prístup je modelovať túto funkciu pomocou penalizačných funkcií 2.17 [20].

$$E(\vec{v}) = mf(\vec{v}) + \sum_{c \in C} a_c P_c(\vec{v}) \tag{2.17}$$

Parametre funkcie $E(\vec{v})$ sú:

[8] Problém obchodného cestujúceho (anglicky *travelling salesman problem*, skratkou *TSP*) je jedným z najpopulárnejších optimalizačných problémov pre demonštráciu optimalizačných algoritmov.

- m je znamienková konštanta určujúca charakter optima. Ak je OP maximalizačný problém $m = -1$, ak je minimalizačný $m = 1$.
- $f(\vec{v})$ je objektívna funkcia OP (viď 1.2.1).
- C je množina logických obmedzení.
- $P_c(\vec{v})$ predstavuje penalizačnú funkciu, ktorá v závislosti od pravdivosti obmedzenia c mení svoju hodnotu. $P_c(\vec{v}) = 0$ vtedy a len vtedy ak obmedzenie c je na $\vec{v}$ pravdivé, inak má hodnotu kladnej konštanty $P_c(\vec{v}) = p, p > 0$, ktorá je zvolená podľa závažnosti porušenia (*degree of violation*) logického obmedzenia c.
- a_c je relatívna váha obmedzenia c, oproti ostatným definovaným obmedzeniam. V niektorých prípadoch je túto konštantu možné parametrizovať ako $a_c^{(t)}$ a meniť v priebehu výpočtu.

Logické obmedzenia z množiny C sú zvyčajne navrhnuté ako rovnice alebo nerovnice. Rovnice majú všeobecný tvar 2.18,

$$c_k : \sum_j v_j = \gamma_k \tag{2.18}$$

kde γ_k je konštanta. Pre takýto typ obmedzenia môžeme skonštruovať penalizačnú funkciu 2.19,

$$P_{c_k}(\vec{v}) = \Big(\sum_j x_j - \gamma_k \Big)^2 \tag{2.19}$$

pre ktorú očividne platí, že v prípade pravdivosti naberá hodnotu 0, inak rastie podľa závažnosti porušenia.

Nerovnice majú tvar 2.20,

$$c_k : \sum_j v_j \leq \gamma_k \tag{2.20}$$

kde analogicky k rovniciam je γ_k konštanta. Pre takýto typ obmedzenia môžeme skonštruovať penalizačnú funkciu 2.21,

$$P_{c_k}(\vec{v}) = \Omega\Big(\sum_j x_j - \gamma_k \Big) \tag{2.21}$$

kde $\Omega(y)$ by mala byť funkcia, ktorá penalizuje kladné funkčné hodnoty $y > 0$. Za túto funkciu sa zvyčajne volí buď sigmoida (viď 2.11), ktorá však nadobúda pri raste y maximálnu hodnotu 1 alebo alternatíva 2.22 pozostávajúca z miery porušenia y (ako u rovníc) znásobenej binárnou skokovou funkciou (viď 2.9) [22].

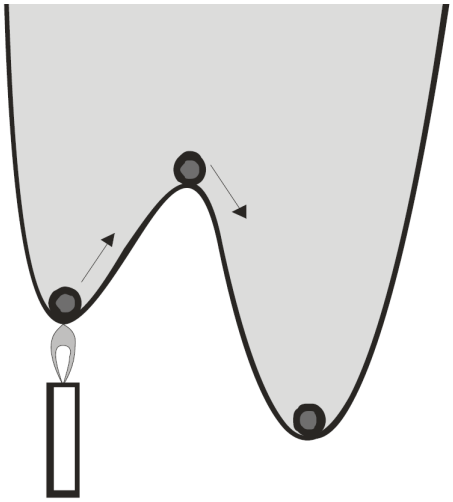
$$\Omega(y) = yBin(y) \tag{2.22}$$

V takomto prípade hodnota penalizačnej funkcie rastie podľa závažnosti porušenia.

V momente, kedy sa nám podarí úspešne skonštruovať energetickú funkciu, ostáva už iba odvodiť parametre našej sieťovej inštancie. Počet neurónov siete n volíme podľa požadovanej veľkosti výstupu siete ($n = |\vec{v}|$). Hodnoty koeficientov váh w a prahov θ "vhodne" (logicky vzhľadom k OP alebo experimentálne) volíme vzhľadom k nastavovanému neurónu i a k zmene sieťovej energie ΔE_i (tú vieme vyjadriť z nami mapovanej energetickej funkcie analogicky, ako to je v prípade 2.15) tak, aby bola zmena pre čo najviac neurónov kladná. Dynamiku siete odvodíme vzhľadom k OP a požadovanému výstupu (napr. ak chceme aby výstupy neurónov boli v intervale $\langle 0, 1 \rangle$, zvolíme spojitú aktivačnú funkciu).

2.3.5 Simulované žíhanie

Metóda simulovaného žíhania (anglicky *simulated annealing*) sa zaraďuje medzi stochastické optimalizačné algoritmy. Podobne ako iné stochastické algoritmy (napr. tie založené na chovaní rojov), jej pôvod pochádza z inej vedy. Počiatkom osemdesiatych rokov prišli Kirckpatrick Gelatt a Vecchi [31] s analógiou problému hľadania globálneho minima objektívnej funkcie optimalizačného problému k žíhaniu tuhého telesa. Zahriatím a postupným ochladzovaním telesa sa odstraňujú jeho štruktúrne defekty. Metafora vzhľadom k optimalizačnému algoritmu je založená na tom, že riešenie, ktoré uviazlo v lokálnom minime sa pri dostatočnom zahriatí z neho môže dostať a ideálne riešenie môže konvergovať ku globálnemu minimu (Obr. 2.7). Algoritmus inšpirovaný fyzikálnou evolúciou systému smerom



Obr. 2.7: *Ilustrácia simulovaného žíhania v optimalizačných algoritmoch. 'Zahriatie' riešenia uviaznutého v lokálnom minime a jeho konvergencia ku globálnemu minimu. Zdroj [18].*

k tepelnej rovnováhe je založený na Metropolisovom algoritme [30]. Ten simuluje evolúciu systému tak, že ak je systém v nejakom aktuálnom stave A, existuje pravdepodobnosť, že dôjde k zmene (poruche) jeho stavu na stav B, pričom táto pravdepodobnosť je symetrická (takže rovnaká pravdepodobnosť platí pre zmenu stavu z B do A). Uvažujme, že týmto systémom je Hopfieldova neurónová sieť. Ak dôjde k poruche, potom zmenu jeho energie môžeme vyjadriť rovnicou[9] 2.23.

[9] $E_{pertrubed}$ značí energiu porušenej siete, zatiaľ čo $E_{current}$ značí energiu siete pred porušením.

$$\Delta E = E_{pertrubed} - E_{current} \tag{2.23}$$

Ak je zmena energie ΔE negatívna, potom je porušený stav prijatý a evolúcia systému pokračuje, v opačnom prípade je pravdepodobnosť prijatia tohto stavu daná exponenciálou založenou na boltzmannovskom rozdelení pravdepodobnosti $e^{\frac{-\Delta E}{kT}}$, kde T značí aktuálnu teplotu systému a k je vhodná konštanta. Pravidlo akceptovania porušeného stavu, zvané *Metropolisovo kritérium* teda definujeme ako 2.24.

$$P(pertrubed \leftarrow current) = min(1, e^{\frac{-\Delta E}{kT}}) \tag{2.24}$$

Všimnime si, že ak je zmena energie ΔE negatívna, exponenciála má hodnotu väčšiu ako 1 a na základe definície 2.24 je určite táto zmena prijatá.

Kapitola 3

Funkčná verifikácia

Úlohou funkčnej verifikácie je vierohodne[1] preukázať, že verifikovaný model obvodu spĺňa požadovanú špecifikáciu. Častou chybou býva zámena pojmov verifikácia a testovanie. Rozdiel medzi verifikáciu a testovaním (v oblasti HW) spočíva v tom, že model produktu verifikujeme pred jeho fyzickou realizáciou, avšak testujeme ho typicky až po jeho fyzickej realizácii[2].

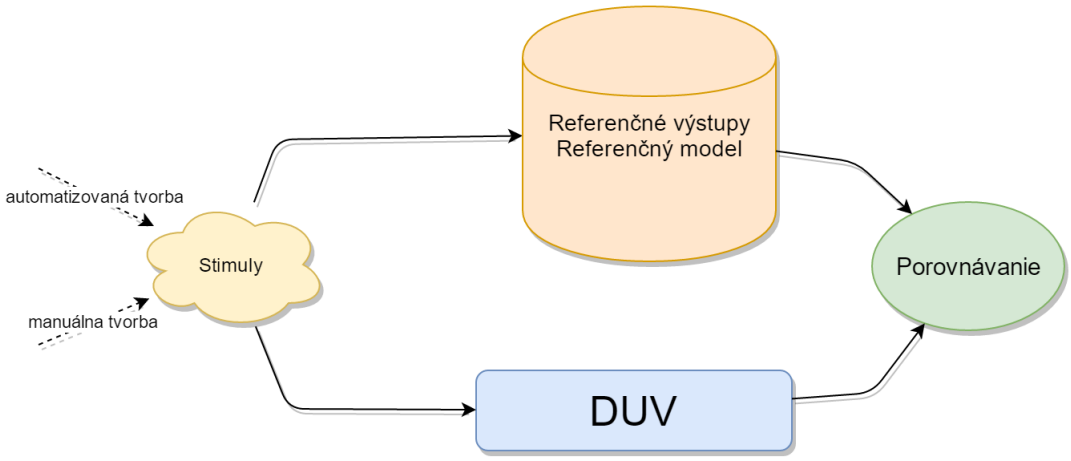
Po designe modelu obvodu HW inžiniermi sa navrhnutý model dostáva k verifikačným inžinierom (stáva sa z neho *DUV*), ktorí následne preň vytvoria stimuly (verifikačnú množinu vstupov). Po aplikovaní stimulov na verifikovaný model je nutné istým spôsobom overiť správnosť jeho chovania (stav modelu a jeho výstupy). Pre overenie korektnosti výstupov verifikovaného modelu sa používajú rôzne prístupy. Tými najpoužívanejšími podľa [33] sú:

- **Referenčný vektor** (*Golden Vector*) je vektor očakávaného výstupu modelu pre určitý stimul. Porovnáva sa so skutočným výstupom DUV.
- **Referenčný model** (*Golden Model*) je model, ktorý sa v zmysle verifikovaného chovania chová presne tak isto ako DUV. V realite býva oveľa abstraktnejší než DUV, pričom typicky je implementovaný vo vyššom programovacom jazyku. Chovanie DUV, ktorý môže byť popísaný na rôznych úrovniach abstrakcie (napr. popis na úrovni registrov alebo brán), je počas verifikácie porovnávané s chovaním referenčného modelu.

V praxi sa častokrát tieto prístupy kombinujú. Všeobecnú schému funkčnej verifikácie teda môžeme znázorniť tak, ako je znázornené na obrázku 3.1.

[1] Funkčnou verifikáciou, zvyčajne vzhľadom na zložitosť obvodu nie je možné v reálnom čase dokázať úplnú bezchybnosť verifikovaného modelu.

[2] Uveďme si príklad na vývoji ASIC čipu. Firma ABCD strávi rok návrhom ich procesoru. Majú špecifikáciu, schémy aj simulácie. Pred tým než však svoj model fyzicky vytvoria, predajú ho verifikačnému oddeleniu, ktoré overí funkčnosť modelu (model sa stáva tzv. *DUV* z anglického *Design Under Verification*) na základe dostupných špecifikácií, schém a simulácií. Po verifikácii je model fyzicky zostavený v továrni a až potom sa produkt dostáva do QA oddelenia, ktoré sa zaoberá testovaním. Tu sa z neho stáva tzv. *DUT* (z anglického *Device Under Test*). Odhalenie chyby v tejto fáze návrhu oproti verifikačnej fáze je však značnou finančnou stratou pre firmu ABCD.



Obr. 3.1: *Všeobecná schéma funkčnej verifikácie hardvéru.*

3.1 Funkčná verifikácia v SystemVerilogu

Programovací jazyk SystemVerilog (*SV*) je jazykom, ktorý vznikol rozšírením jazyka pre popis hardvéru (*Hardware Description Language*, skratkou *HDL*) Verilog o elementy vhodné pre tvorbu verifikačných prostredí pre účely verifikácie HW. Jeho vznik bol ovplyvnený aj jazykom pre verifikáciu HW (tzv. *Hardware Verification Language*, skratkou *HVL*) s názvom Vera. Samotný SystemVerilog je teda kombináciou HDL a HVL, avšak obsahuje aj veľa známych techník z imperatívne orientovaných jazykov ako C++ (OOP), čo výrazne ovplyvňuje jeho obľúbenosť v praxi. Charakteristické sú preň nasledujúce techniky [16]:

- **Náhodné generovanie stimulov s obmedzeniami** (*Constrained-Random Stimulus Generation*). Keďže písanie stimulov pre väčšie obvody je náročné, vhodným spôsobom sa javí automatické generovanie verifikačných stimulov za použitia generátoru, ktorého rozloženie náhodnosti je možné ovplyvniť logickými obmedzeniami (anglicky *constraints*). Obmedzenia nielen že definujú správnu formu generovaných stimulov, ale aj je pomocou nich možné navádzať náhodné generovanie do kritických častí systému, alebo okrajových prípadov.

- **Spolupráca s inými programovacími jazykmi**. Súčasťou SV je rozhranie s názvom *Direct Programming Interface* (*DPI*), ktoré umožňuje volanie funkcií popísaných inými programovacími jazykmi a to tak, ako keby boli popísane v SV. To umožňuje vznik verifikačných prostredí založených na celej rade komponentov a knižníc dostupných v populárnych imperatívnych programovacích jazykoch (ako napr. C++, Python).

- **Verifikácia riadená pokrytím** (anglicky *Coverage-Driven Verification*, skratkou *CDV*). Čo najväčšie pokrytie designu je jedným z kľúčových cieľov funkčnej verifikácie.

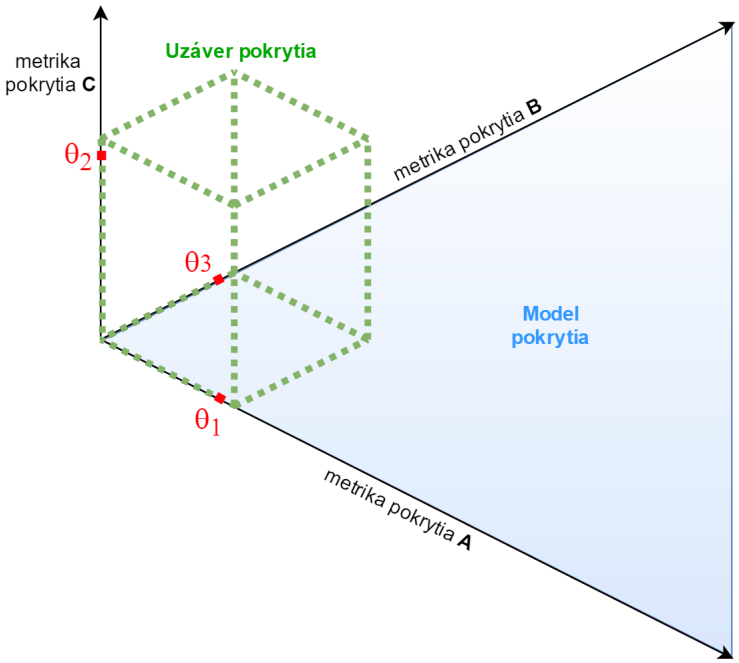
  Podľa vzoru [16] definujeme terminológiu súvisiacu s pokrytím nasledovne.

  Metrika pokrytia je merateľná vlastnosť DUV ako napríklad počet aktivovaných riadkov kódu, či počet skontrolovaných aritmetických operácií.

Priestor pokrytia je n-dimenzionálny priestor, ktorého dimenzie definuje n metrík pokrytia.

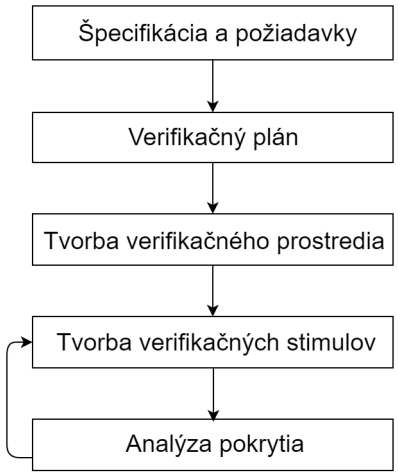
Model pokrytia je i-dimenzionálnym podpriestorom n-dimenzionálneho priestoru pokrytia. Platí teda, že $i \leq n$.

Aby sme dosiahli čo najväčšie pokrytie, je vhodné kombinovať niekoľko metrík pokrytia a vytvoriť tak v priestore pokrytia teleso s čo najväčším objemom. V praxi sa zvyčajne verifikační inžinieri snažia dosiahnuť nejakú prahovú hodnotu pre každú z metrík priestoru pokrytia a dosiahnuť tak tzv. *uzáver pokrytia* (*Coverage Closure*). Ideálne tieto prahové hodnoty predstavujú maximálne pokrytie. Ilustrácia zavedených definícií je obsahom Obr. 3.2.



Obr. 3.2: *3-rozmerný priestor pokrytia tvorený metrikami pokrytia A,B,C. Rovina, ktorú tvoria metriky A a B je modelom pokrytia. Uzáver pokrytia je možné získať dosiahnutím prahových hodnôt pre jednotlivé metriky (θ_1 pre A, θ_2 pre B, θ_3 pre C).*

Na meranie jednotlivých metrík sa používajú RTL simulátory (ako QuestaSim od spoločnosti Mentor, Riviera-PRO od spoločnosti Aldec alebo VCS od spoločnosti Synopsys), ktoré okrem priebehu simulácie ponúkajú široké spektrum štatistík. Pomáhajú tak nájsť medzery v pokrytí a zdokonaliť súbor stimulov natoľko, aby pokrytie bolo v smere každej metriky maximálne. Práve od tohto procesu sa odvíja názov *verifikácia riadená pokrytím*.



Obr. 3.3: *Kroky verifikačného procesu typické pre verifikáciu riadenú pokrytím. Pri špecifikácii sa vytýčia požiadavky na model a referenčné výstupy, verifikačný plán predstavuje súbor plánov, ktoré popisujú spôsoby prístupov k verifikácii, ale aj plán pre financie a ľudské zdroje. Pre podrobnejší popis viď [16].*

V štandarde popisujúcom jazyk SystemVerilog [12] sú vytýčené nasledujúce metriky pokrytia, ktoré rôznym spôsobom pokrývajú funkčnosť verifikovaného obvodu:

- **Funkčné pokrytie** (*Functional Coverage*) udáva veľkosť pokrytia funkčnej špecifikácie DUV súborom vstupných stimulov. Týmto prístupom je vhodné verifikovať napríklad vyvolávanie výnimiek, hodnoty dát v registroch, či chovanie aritmeticko-logickej jednotky, keďže tento prístup sa zameriava na sémantický význam zdrojov. SystemVerilog poskytuje pre účely funkčného pokrytia konštrukciu, ktorú budeme nazývať skupina pokrytia (`covergroup`). V tejto konštrukcii sa nachádzajú tzv. body pokrytia (`coverpoint`), ktoré popisujú vlastnosti vyjadrujúce mieru pokrytia tejto skupiny. Táto konštrukcia je bližšie popísaná v podkapitole 3.1.1.
- **Štrukturálne pokrytie** (*Structural Coverage*) je možné získavať automaticky v závislosti od použitého simulačného nástroja. Bežné RTL simulátory však obsahujú nástroj pre analýzu pokrytia, ktorý v rámci simulácie vkladá do simulátoru vlastný kód, ktorým získava štatistiky, či meria mieru pokrytia podľa popisu DUV. Typickými zástupcami tejto skupiny sú nasledujúce metriky:
  - **Pokrytie zmien** (*Toggle Coverage*) analyzuje zmeny logických hodnôt u signálov a registrov na úrovni bitov. Metrika vníma vzťah stimulu k zmene logických hodnôt istej množiny zdrojov, neberie však do úvahy sémantický význam logických hodnôt týchto zdrojov.
  - **Pokrytie kódu** (*Code Coverage*) analyzuje syntaktickú štruktúru kódu a berie do úvahy aktivované riadky kódu. Rovnako ako pri testovaní SW však tento prístup nepokrýva sémantické konštrukcie jazyka. Tento problém je možné odstrániť upresnením metódy tak, že namiesto všetkých riadkov budeme pokrývať iba riadky s príkazmi (*Statement Coverage*), skokmi (*Branch Coverage*), podmienkami (*Condition Coverage*) či výrazmi (*Expression Coverage*).

* **Pokrytie podľa konečného automatu** (*Finite State Machine Coverage*) verifikuje konečné automaty prítomné vo verifikovanom HDL návrhu, pri ktorom sa snaží pokryť buď všetky ich stavy (*State Coverage*) alebo všetky ich prechody (*Transition Coverage*).

3.1.1 Konštrukcie SystemVerilogu pre definíciu pokrytia

Syntaktická konštrukcia jazyka SV, ktorá zapuzdruje abstrakciu priestoru pokrytia DUV, sa označuje ako skupina pokrytia (`covergroup`). Každá konštrukcia `covergroup` môže obsahovať nasledujúce komponenty:

- **Body pokrytia** (`coverpoint`) predstavujú vlastnosti IC, ktoré budeme chcieť počas RTL simulácie obvodu sledovať. Typicky sa viažu na konkrétne signály obvodu. Môžu to byť napríklad inštrukcie, sekvencie inštrukcií, skupiny ilegálnych hodnôt v registroch apod. Každý `coverpoint` pozostáva z definície tzv. zberných košov (*bins*), do ktorých sú vzorkované konkrétne hodnoty signálov podľa jasne definovaných formálnych pravidiel. Tie napríklad v prípade bodu pokrytia inštrukcií popisujú konkrétne inštrukcie, ku ktorých pokrytiu má dôjsť. Nad bodmi pokrytia je tiež možné realizovať niektoré množinové a logické operácie (viď 3.1.1).
- **Synchronizačné udalosti** určujú okamihy vzorkovania bodov pokrytia v priebehu simulácie, napríklad na nástupnú hranu hodinového signálu (`posedge clk`). Túto informáciu je možné explicitne určiť aj pre samotné skupiny pokrytia.
- **Kombinácie** bodov pokrytia (`cross`) umožňujú modelovať pokrytie vlastností IC vzhľadom na viac signálov obvodu súčasne. Dajú sa preto vyjadriť ako kartézsky súčin bodov pokrytia. Prvky kombinácií bodov pokrytia sa nazývajú cross produkty.

```
bit [7:0] v_a, v_b;// Deklaracia dvoch 8 bitovych signalov v_a, v_b
covergroup cg @(posedge clk);// Definicia covergroup konstrukcie so
                             // vzorkovanim v okamihu nastupnej hrany
  a: coverpoint v_a// Bod pokrytia naviazany na signal v_a
  {
    //Zberne kose pre konkretne hodnoty signalu v_a
    bins a1 = { [0:63] };
    bins a2 = { [64:127] };
    bins a3 = { [128:191] };
    bins a4 = { [192:255] };
  }
  b: coverpoint v_b
  {
    bins b1 = {0};
    bins b2 = { [1:84] };
    bins b3 = { [85:169] };
    bins b4 = { [170:255] };
  }
  // Kombinacia bodov pokrytia naviazana na signaly v_a, v_b
  c : cross v_a, v_b
  {
    bins c1 = ! binsof(a) intersect {[100:200]};// 4 cross produkty
    bins c2 = binsof(a.a2) || binsof(b.b2);// 7 cross produktov
    bins c3 = binsof(a.a1) && binsof(b.b4);// 1 cross produkt
  }
endgroup
```

Kód 3.1: Príklad skupiny pokrytia v SystemVerilogu prevzatý z [12].

- **Nastavenia**. Každá `covergroup` môže mať napríklad priradenú váhu, ktorá je zohľadnená pri výpočte celkového pokrytia DUV.

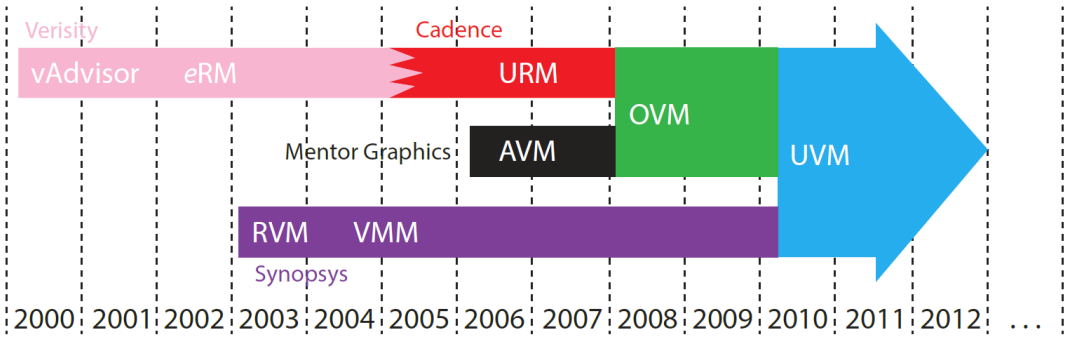
3.2 Verifikačné prostredie

Po špecifikácii a zostavení verifikačného plánu je ďalším dôležitým krokom tvorba verifikačného prostredia, v ktorom bude realizovaná verifikácia riadená pokrytím.

V minulosti bola funkčná verifikácia výrazne jednoduchšou úlohou. Návrhár pripravil stimuly a referenčné výstupy, simuloval niekoľko cyklov DUV, kontroloval chovanie signálov a registrov a porovnával referenčné výstupy. So stúpajúcou zložitosťou čipov však takýto prístup prestal byť v praxi použiteľný a vznikla výskumná oblasť zaoberajúca sa systematickým prístupom k tvorbe prostredí, v ktorých prebieha verifikácia – došlo k vzniku verifikačných metodík.

3.2.1 Verifikačné metodiky

Počiatky verifikačných metodík boli vyvinuté nezávisle rôznymi spoločnosťami v rozmedzí posledných 20 rokov.



Obr. 3.4: *História verifikačných metodík. Zdroj [16].*

Ako sme si už predstavili v úvode tejto sekcie, vznik SV bol inšpirovaný viacerými HDL a HVL jazykmi. Pre tieto jazyky (Vera, *e*) už komerčná sféra vyvinula proprietárne metodiky (RVM, *e*RM), ktoré boli v nich implementované. Po vzniku a rozšírení SV vo verifikačnej komunite došlo v roku 2006 z iniciatívy spoločnosti Mentor k vzniku novej metodiky AVM inšpirovanej konceptmi ich jazyka pre vysoko-úrovňovú syntézu zvaného *SystemC*. V tom istom období aj Synopsis konvertoval ich RVM, ktorý vyvinuli pre jazyk Vera do SV a túto metodiku nazvali VMM (nebola však verejne dostupná). Oneskorene v roku 2007 aj spoločnosť Cadence vytvorila novú metodiku zvanú URM. O rok nato v roku 2008 po spoločnom úsilí spoločností Mentor a Cadence vznikla OVM a až v roku 2011 za pomoci neziskovej organizácie Accelera došlo k zjednoteniu VMM a OVM a vzniku spoločnej verifikačnej metodiky UVM (*Universal Verification Methodology*), ktorá je použitá v praktickej časti tejto práce.

3.2.2 UVM

Hlavnými vlastnosťami UVM oproti predchádzajúcim metodikám sú [17]:

- Je to štandardizovaná unifikovaná metodika, pre ktorú platí:

  - Je nezávislá od spoločnosti, ktorá ju navrhla/používa.
  - Testovacie stimuly sú znovupoužiteľné medzi abstraktnými úrovňami verifikácie (IP, Subsystem, SoC).
  - Je otvorená – v SV je poskytovaná ako open-source knižnica priamo od organizácie Accelera.
  - Do istej miery obsahuje spätnú kompatibilitu, takže je možné jednoduchšie migrovať na túto metodiku.

- Je kompatibilná s nástrojmi využívajúcimi jazyk IP-XACT[3]. To znamená, že je napríklad možné generovať UVM komponenty pomocou nástrojov, ktorých vstupom sú IP-XACT súbory.

Naviac, dnes už je nepísaným štandardom, že UVM je podporovaná simulačnými nástrojmi, ktoré podporujú aj SV. Práve preto sa z nej v súčasnosti stala najrozšírenejšia metodika vo verifikačnej oblasti [7]. Dôležitou súčasťou UVM je práve CDV, predstavená v minulej kapitole.

Architektúra verifikačného prostredia (viď Obr. 3.5) podľa UVM pozostáva z modulárnych samostatných jednotiek nazývaných verifikačné komponenty (v niektorej literatúre nazývané aj ako *univerzálne verifikačné komponenty*). Každý verifikačný komponent je zapuzdrenou konfigurovateľnou jednotkou pripravenou na použitie – môže sa s ním komunikovať pomocou rozhrania s protokolom, môže tvoriť podsystém alebo aj úplný systém v rámci verifikačného prostredia. Architektúra je modulárna aj v tom zmysle, že každý verifikačný komponent môže obsahovať ďalšie verifikačné komponenty. UVM definuje nasledujúce verifikačné komponenty:

- **Transakcia** (*Data Item, Transaction*) je vstupom pre DUV (inštrukcie, zbernicové signály či sieťové pakety).

- **Ovládač** (*Driver*) je komponentom, ktorý zapuzdruje DUV a ovláda jeho vstupy. Jeho typickou funkciou je predávanie transakcií do DUV vo forme nastavovania jeho portov a rozhraní. Napríklad v prípade zápisu sa ovládač stará o vystavenie signálov potrebných pre zápis na dátovú a adresovú zbernicu v správnom čase na potrebný počet cyklov.

- **Sekvencér** (*Sequencer*) generuje transakcie pre pripojené komponenty. Jeho základné chovanie je také, že na žiadosť ovládača vracia (pseudo)náhodné dáta. Toto chovanie je však možné obmedziť pomocou logických obmedzení, rôznorodých nastavení, ako napríklad reakciou na zmeny stavu ostatných komponentov verifikačného prostredia.

- **Monitor** je pasívnou entitou. Zbiera dáta, ktoré môže vyhodnocovať a zasielať ich ďalším komponentom, avšak priamym spôsobom neovplyvňuje stav verifikačného prostredia. Monitor sa stará o:

  - Zbieranie výstupných transakcií – zachytáva informácie zo zberníc, eventuálne konvertuje tieto informácie do transakcií dostupných pre iné komponenty.
  - Zaznamenávanie udalostí – deteguje zmeny stavu prostredia, dostupnosť dát a v prípade dostupnosti ich spracuje, prípadne upozorní iné komponenty na dostupnosť informácie.

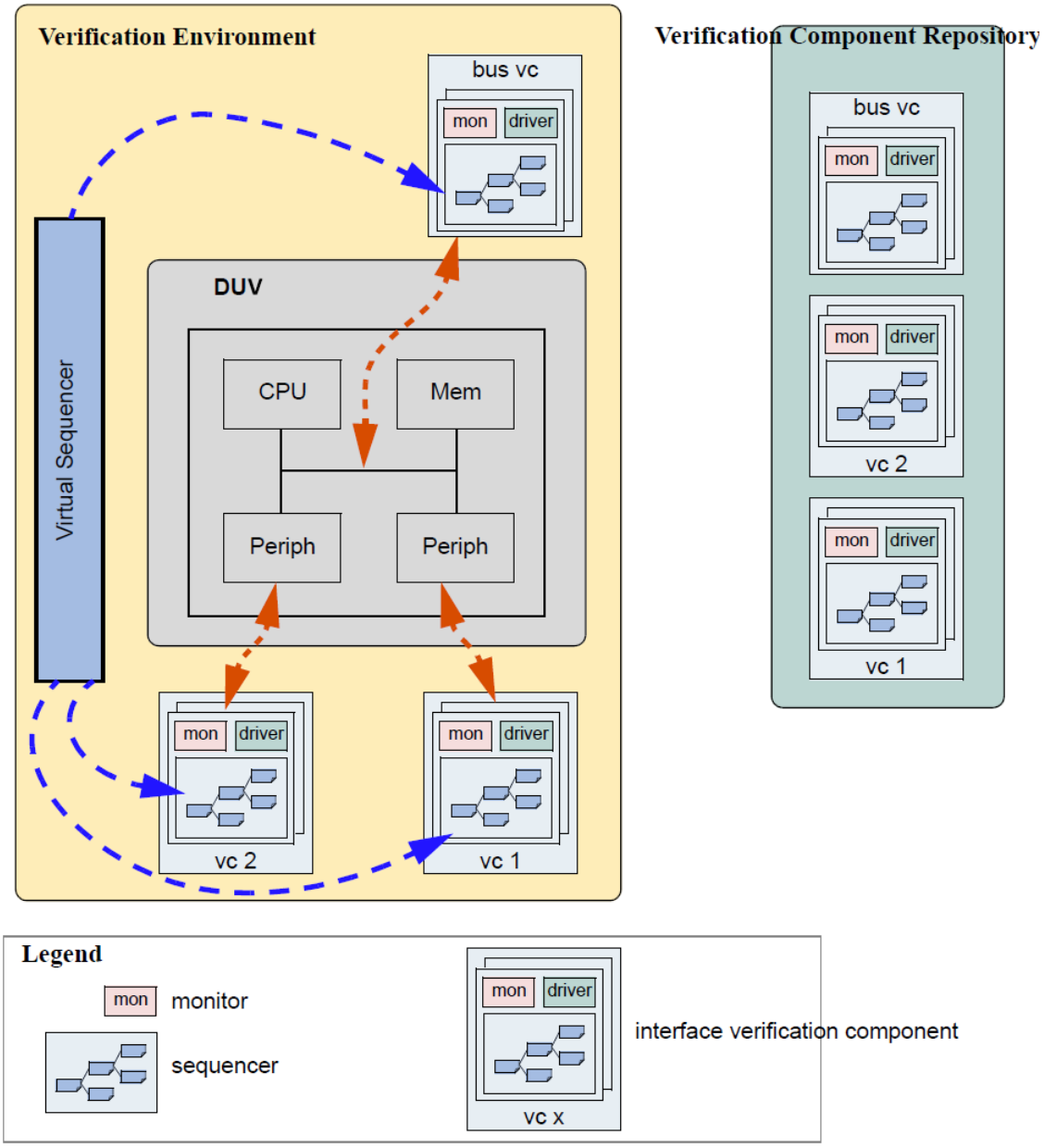
[3]IP-XACT je formát XML, ktorý slúži pre popis elektronických komponentov a ich designov. Je to štandard vytvorený skupinou spoločností nazývanou SPIRIT Consortium [11].

  - Trasovanie verifikácie – monitor môže do simulátoru vypisovať ladiace výstupy.

- **Scoreboard** je kľúčovým elementom funkčnej verifikácie designu. Tento komponent zhromažďuje dáta a tvorí štatistiky (metadáta) analýzou transakcií získaných z monitorov zapuzdrených v rôznych častiach UVM prostredia. Typicky sa tiež stará o ich validáciu vzhľadom k špecifikácii DUV a výpočet metrík pokrytia DUV.

- **Agent** plní úlohu abstraktného kontajneru, ktorý zapuzdruje sekvencéry, ovládače a monitory. Tie sú síce znovupoužiteľné aj bez zapuzdrenia, ale práca s kontajnermi, ktoré obsahujú množinu verifikačných komponentov, je rýchlejšia a pohodlnejšia. Agenti môžu byť aktívni, ak ovplyvňujú prostredie, napr. agenti, ktorí niečo ovládajú (*Master Agent*) alebo zasielajú (*Transmit Agent*). Agenti môžu byť aj pasívni, ak len monitorujú DUV, napr. agenti, ktorí sú niekým ovládaný (*Slave Agent*) alebo len prijímajú dáta (*Recieve Agent*).

- **Prostredie** (*Environment*) je komponentom, ktorý tvorí najvyššiu vrstvu (*top level*). Zaobaľuje celé verifikačné prostredie. Pozostáva zo všetkých ostatných predstavených komponentov, typicky predovšetkým z agentov a monitorov.



Obr. 3.5: *Príklad architektúry UVM s 3 verifikačnými komponentmi pripojených pomocou rozhraní. Zdroj [1].*

Kapitola 4

Analýza prostredia

Riešenie problémov stanovených v podkapitole 1.2 vyžaduje okrem optimalizačnej heuristiky prostredie, prostredníctvom ktorého bude hľadanie riešenia prebiehať. Vo všeobecnosti by navrhované riešenie heuristiky malo byť kompatibilné s akýmkoľvek prostredím, v ktorom je možné analyzovať pokrytie DUV, avšak keďže úlohou tejto práce je verifikáciu automatizovať, budeme sa zaoberať nástrojmi pre automatizovanú tvorbu verifikačných prostredí, stimulov do týchto prostredí a tiež sa oboznámime s RTL simulátormi a ich možnosťami. Model procesoru Codasip uRISC (a aj Codix Cobalt), na ktorých prebieha experimentálna časť tejto práce, boli vytvorené sadou nástrojov zvanou Codasip Studio (*CS*). CS okrem tvorby samotných procesorov typu ASIP umožňuje aj automatizovanú tvorbu prostredia. Prostredím v tomto kontexte bude myslená množina pozostávajúca z nástrojov vygenerovaných CS (prekladač asembleru, linker, ...), skriptov a predovšetkým verifikačného prostredia. V nasledujúcich podkapitolách sa budeme zaoberať jeho tvorbou a neinvazívnou integráciou[1] navrhovanej optimalizačnej heuristiky – neurónovej siete do takto vytvoreného prostredia.

4.1 Codasip Studio

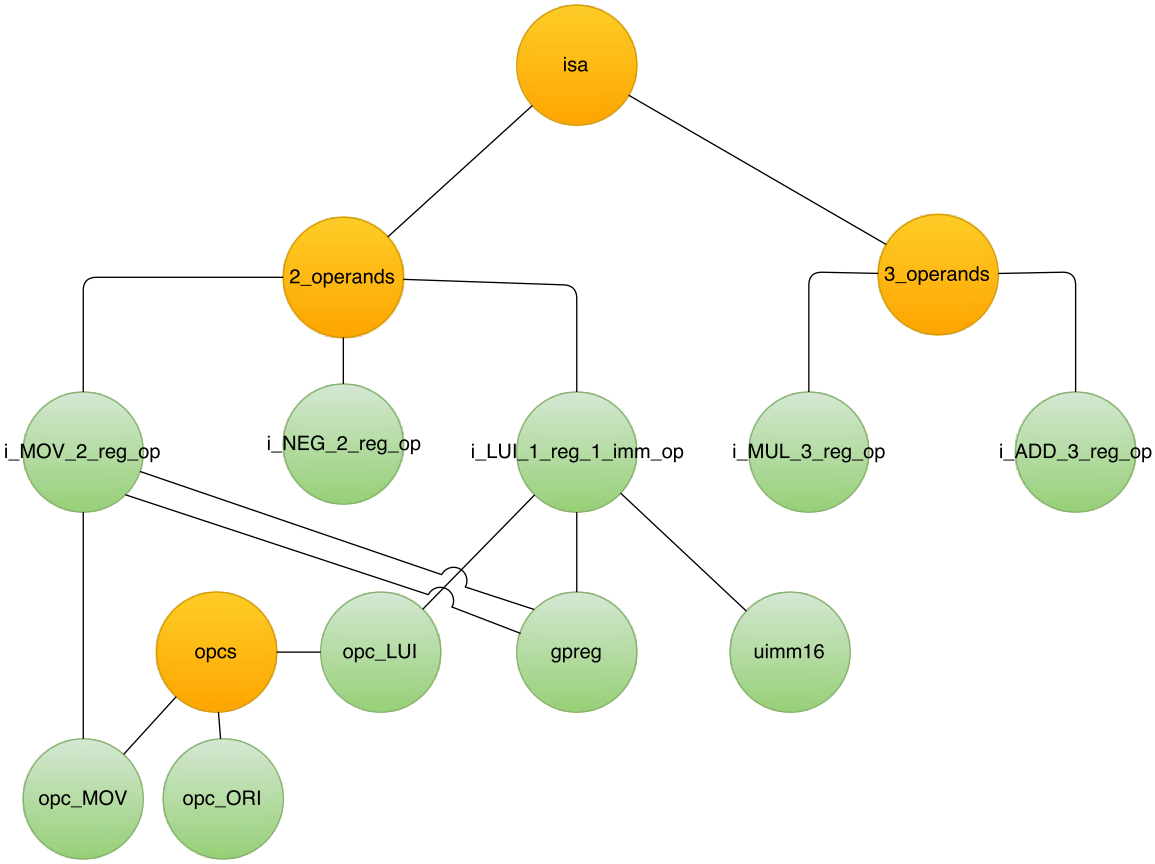
Codasip Studio je sada nástrojov poskytovaná spoločnosťou Codasip, ktorá umožňuje kompletný návrh aplikačne špecifických procesorov (ASIP) a ich prepojenie[2] s pamäťami či perifériami ako FPU, MMU, čítače, či zbernice. Návrh je realizovaný pomocou špeciálne navrhnutého jazyka s názvom CodAL, ktorý bol vyvinutý spoločnosťou Codasip v spolupráci s Fakultou Informačních Technologií na Vysokým Učení Technickým v Brně. Tento jazyk vychádza s jazyka C a zaraďuje sa do skupiny zmiešaných ADL (*Architecture Description Language*) [6]. Jazyk umožňuje popis designu na dvoch rôznych úrovniach.

V počiatočnej fáze návrhu procesoru umožňuje na vysokej úrovni abstrakcie popísať inštrukčnú sadu pomocou tzv. IA modelu (*Instruction Accurate Model*). Popis inštrukcií prebieha hierarchicky, pomocou popisu jednotlivých častí inštrukcií, tzv. subinštrukcií (*SI*). Syntax a sémantika subinštrukcií je popísaná cez tzv. elementy. Návrh inštrukčnej sady prebieha postupným popisom týchto elementov a ich skladaním do čoraz komplexnejších elementov. Štruktúru, ktorá vznikne takýmto postupným skladaním, vieme namodelovať pomocou acyklického grafu. Existuje ešte aj vyššia úroveň abstrakcie, v ktorej sú elementy skladané cez tzv. sety (množiny elementov), nad ktorými je možné hromadne po-

[1] Navrhnuté riešenie nevyžaduje tvorbu zmien vo verifikačnom prostredí.

[2] Proces prepojenia ASIP s ďalšími komponentmi sa nazýva *návrh platformy*.

pisovať syntax a sémantiku, či aplikovať hromadné operácie. Naviac aj samotné sety môžu byť súčasťou ďalších setov. (Podobnú štruktúru vytvára napríklad aj behaviorálny ADL s názvom nML [10]). Architektúra, ktorá modeluje vzťahy medzi všetkými elementami a



Obr. 4.1: *Časť acyklického grafu predstavujúca model inštrukčnej sady. Žlté uzly grafu predstavujú sety, zelené predstavujú elementy – teda inštrukcie a subinštrukcie.*

setmi ASIP modelu (teda modeluje návrh inštrukčnej sady), má taktiež podobu acyklického grafu (viď 4.1). Znázornený graf môžeme interpretovať nasledovne. Napríklad element `i_LUI_1_reg_1_imm_op` popisuje chovanie inštrukcie pre načítanie bezznamienkového integeru (*Load Unsigned Integer – LUI*) a pozostáva z troch SI:

- `opc_LUI` popisuje operačný kód LUI.
- `gpreg` popisuje prvok univerzálneho registrového poľa (*General Purpose Registers*).
- `uimm16` je SI popisujúca bezznamienkovú 16-bitovú konštantu (*Unsigned Immediate 16*).

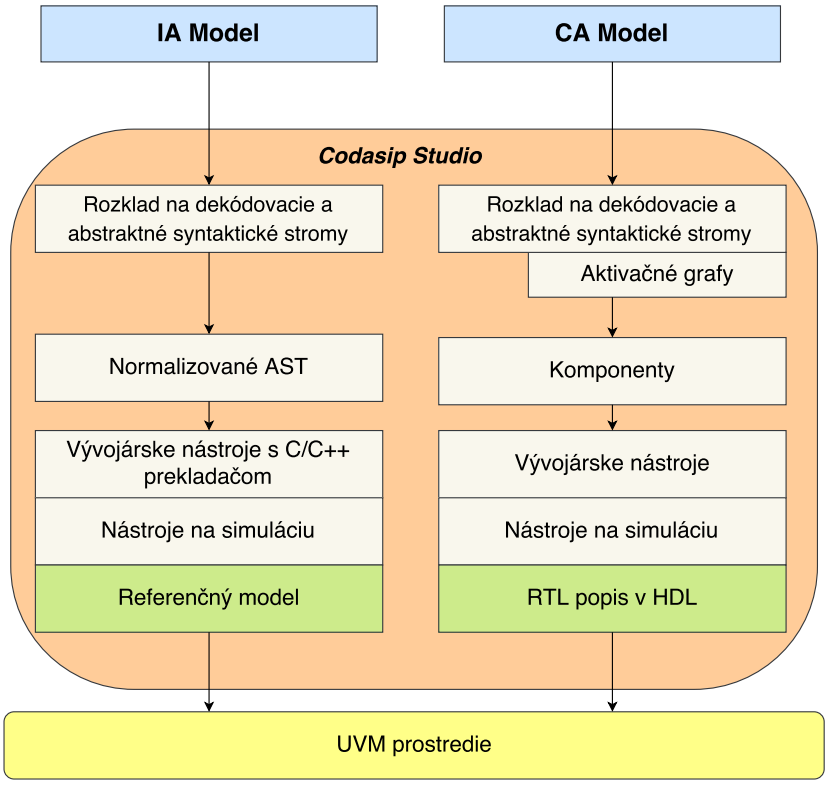
Prostredníctvom tohto modelu je jazyk schopný vygenerovať radu nástrojov, ako napríklad prekladač asembleru, simulátor, C/C++ prekladač založený na open-source platforme LLVM [9] so sadou štandardných knižníc, či množinu náhodne vygenerovaných programov pre definovaný model.

Pre fyzickú realizáciu modelu je nutné popísať ho na úrovni bližšej hardvéru. Jazyk CodAL preto umožňuje súčasne popísať model aj na úrovni hodinových cyklov pomocou tzv.

CA modelu (*Cycle Accurate Model*). Tento model poskytuje špeciálnu jazykovú syntax pre popis dekodéru a predovšetkým pre popis udalostí, ktorými sa zvyčajne modeluje zreťazenie linky (*pipeline*) cez popis jednotlivých úrovní zreťazenia (*pipeline stages*). Podobne ako je tomu pri modelovaní inštrukcií, aj udalosti nadobúdajú hierarchickú štruktúru, keďže jedna udalosť môže spúšťať niekoľko ďalších a tiež môžu byť vkladané do setov. Podrobný tvar CA modelu v tejto práci rozoberať nebudeme, keďže táto znalosť nie je podstatnou pri návrhu automatizovanej verifikácie[3]. Popis CA modelu v CS umožňuje generovanie ďalšej rady nástrojov, ako napríklad CA simulátor, popis modelu na RTL úrovni potrebný pre fyzickú realizáciu a CA simuláciu či UVM verifikačné prostredie.

4.2 Automatizovaná tvorba verifikačného prostredia

Pre účely aplikácie navrhovanej heuristiky bude pre každý model, ktorý je cieľom experimentov tejto práce, nutné vytvoriť netriviálne verifikačné prostredie. CS umožňuje automatickú tvorbu takéhoto prostredia. Každé vygenerované prostredie je založené na UVM a jeho architektúra je vytvorená na základe predgenerovaných šablón v kombinácii s dátami získanými z IA a CA modelu procesoru.



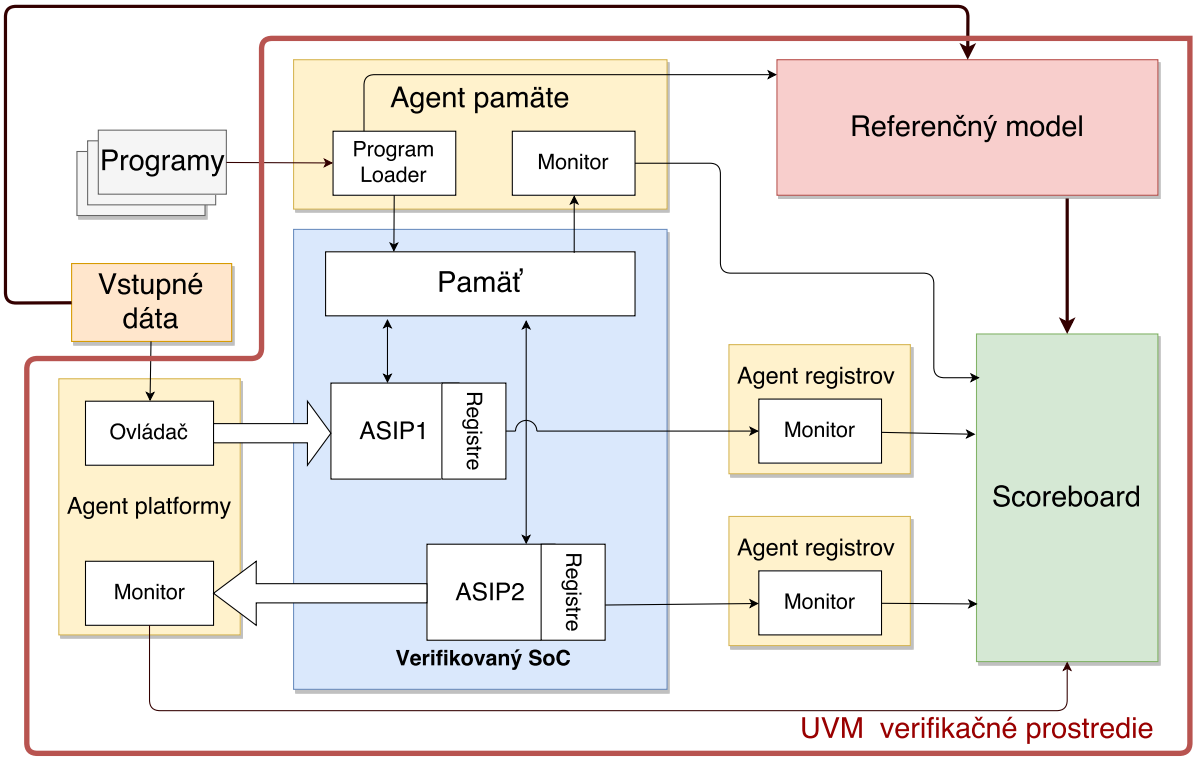
Obr. 4.2: *Priebeh extrakcie dát z IA a CA modelu a následne generovanie referenčného modelu, RTL popisu procesoru a UVM verifikačného prostredia. Skratka AST v tomto obrázku označuje abstraktné syntaktické stromy (anglicky Abstract Syntax Trees).*

Takto vytvorené prostredie realizuje verifikáciu designu simultánnou simuláciou RTL popisu a referenčného modelu a ich postupným porovnávaním. RTL popis je možné získať

[3]Rozboru syntaxe jazyka CodAL, vrátane CA modelu som sa bližšie venoval vo svojej bakalárskej práci [6].

z CA popisu, zatiaľ čo referenčný model je založený na IA modeli (viď 4.2). Samotné prostredie pozostáva predovšetkým z pasívnych agentov operujúcich nad komponentmi systému a transakcií, ktoré zapuzdrujú komunikáciu medzi týmito komponentmi. Agenti monitorujú činnosť jednotlivých komponentov procesoru ako sú registrové polia, zbernice, pamäte, či signály. Každý agent v sebe obsahuje jeden alebo viac monitorov, ktoré v pravidelných intervaloch zasielajú informácie o transakciách a stave komponentov procesoru do UVM komponentu zvaného *scoreboard*, v ktorom dochádza k ich validácii oproti referenčnému modelu. Porovnávanie s referenčným modelom typicky prebieha cyklus po cykle.

Pre ujasnenie štruktúry verifikačného prostredia si uvedieme príklad systému na čipe (SoC) pozostávajúceho z dvoch ASIP, ASIP1 a ASIP2. ASIP1 spracováva vstupné dáta a tvorí pre-procesor pre ASIP2. Ten predspracované dáta vyhodnotí a výsledky zašle na svoje výstupné porty. Obidva systémy zdieľajú tú istú pamäť. Takýto SoC je možné popísať jazykom CodAL a následne uvedeným spôsobom vygenerovať verifikačné prostredie. Pre takto vytvorené prostredia je nutné vytvoriť programy, ktoré budú na systémoch ASIP1 a ASIP2 spustené počas behu verifikačného prostredia (ďalej len VP). V prípravnej fáze UVM VP sú programy nahrané komponentom s názvom *Program Loader* do agenta zapuzdrujúceho pamäť SoC. Počas verifikácie budú pomocou ovládača nastavované vstupy ASIP1. Po ich spracovaní budú z výstupov ASIP2 prečítané dáta, tie budú zaslané do scoreboard a automaticky vyhodnotené oproti výstupom referenčného modelu. Okrem vstupov a výstupov je v každom takte takto navrhnutého systému možné porovnať obsahy registrov a pamätí s referenčným modelom, pretože aj tie sú zapuzdrené samostatnými agentmi. Ilustrácia takéhoto systému je znázornená obrázkom 4.3. Tento príklad bol prevzatý z [16].

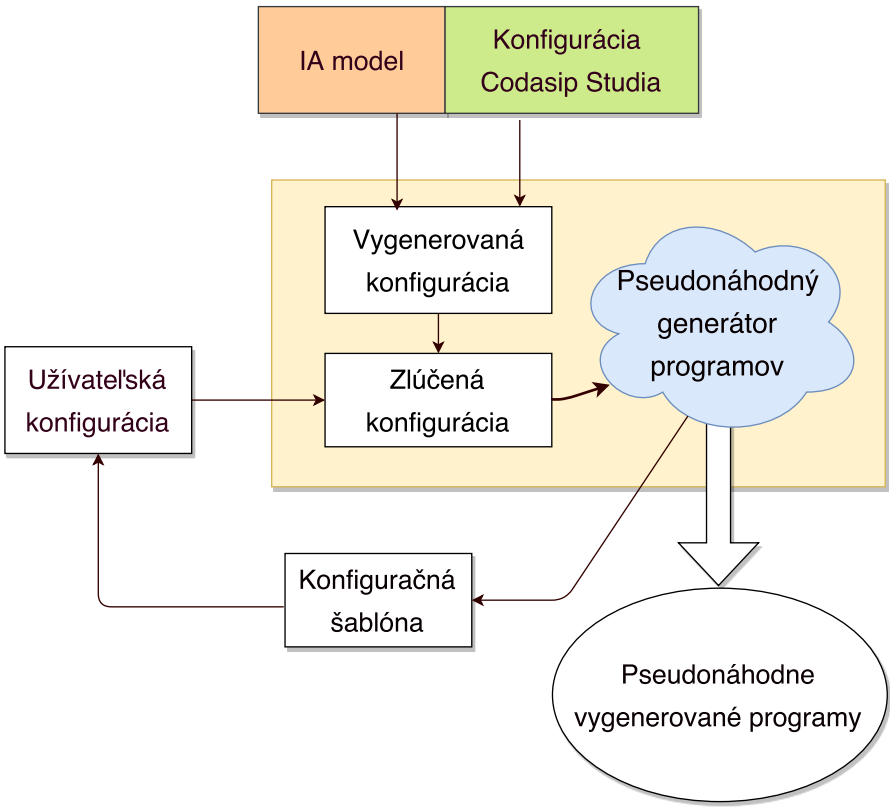


Obr. 4.3: *Príklad SoC pozostávajúceho z dvoch ASIP.*

4.3 Pseudonáhodný generátor programov

Všeobecným cieľom verifikácie hardvéru je pokrytie čo najväčšieho množstva logických stavov, do ktorých sa môže verifikovaný procesor dostať. Problémom tohto prístupu je pokrytie jeho výnimočných stavov[4] (anglicky *corner cases*). Aby sme splnili toto kritérium z hľadiska funkčnej verifikácie, zrejme budeme potrebovať rôznorodú, ale validnú sadu programov, ktorá sa o dosiahnutie všetkých možných prípadov a vlastností procesoru postará. V súčasnosti sa dosiahnutie takýchto výnimočných prípadov v praxi rieši manuálne či poloautomaticky – časť stimulov je automatizovane vygenerovaná a časť, týkajúca sa predovšetkým výnimočných stavov, je písaná verifikačnými inžiniermi ručne. Na automatizovanú časť tohto procesu sa používa už spomenutá technika s názvom *pseudonáhodné generovanie stimulov s obmedzeniami* (viď 3.1). Keďže jedným z cieľov tejto práce je zníženie počtu prípadov, v ktorých automatická verifikácia nestačí a model ASIP sa musí verifikovať manuálne, v tejto práci sa zameriame práve na túto techniku. Bude preto nutné mať k dispozícii nástroj, ktorý dokáže vytvoriť množinu stimulov pre akýkoľvek verifikovaný ASIP model.

Súčasťou Codasip Studia je práve takýto nástroj pre pseudonáhodné generovanie programov. Ten je schopný generovať programy variabilnej dĺžky či rôzneho charakteru na základe vstupnej konfigurácie popísanej pomocou jeho konfiguračného jazyka. Pod pojmom konfigurácia v tejto práci rozumieme obmedzenia pre PNG.



Obr. 4.4: *Architektúra pseudonáhodného generátoru v Codasip Studiu. Konfigurácia PNG pozostáva z konfigurácie vygenerovanej CS a z užívateľskej konfigurácie. PNG naviac umožňuje automatizovanú tvorbu konfiguračných šablón, ktoré urýchlia užívateľom prácu.*

[4]Výnimočnými prípadmi myslíme také stavy IC, ktoré pri jeho behu nastanú len s veľmi nízkou pravdepodobnosťou.

Pre ovládanie pseudonáhodného generátoru dostupného v CS je nutné nastaviť správne jeho konfiguráciu, ktorá popisuje rôzne syntaktické a sémantické vlastnosti verifikovanej architektúry. Manipuláciou s konfiguráciou PNG sa taktiež môžeme zamerať na rôzne časti pokrytia stavového priestoru, do ktorého sa môže verifikovaný model dostať. CS je však schopné vygenerovať prednastavenú konfiguráciu iba na základe IA modelu a nastavení CodAL projektu. Architektúru PNG použitého v CS je možné nájsť na Obr. 4.4.

V následujúcom texte si uvedieme niekoľko základných vlastností, ktoré je možné ovplyvniť konfiguráciou PNG.

Syntax inštrukčnej sady

Je zrejmé, že pre účely pseudonáhodného generovania programov pre nejaký ASIP model musí PNG poznať detailný popis syntaxe každej inštrukcie z inštrukčnej sady architektúry, pre ktorý sa má množina programov generovať.

Sémantické obmedzenia inštrukčnej sady

Neelementárne inštrukcie IC môžu byť použité len v takom kontexte, ktorý zaručí validitu vygenerovaného programu. Napríklad niektoré inštrukcie môžu byť použité iba s takými operandmi, ktoré na procesore nevyvolajú nečakané prerušenie či dokonca nepozastavia vykonávanie programu. Pre jednotlivé inštrukcie je tak v rámci modelu možné nakonfigurovať tzv. obmedzenia (*constraints*) ako napríklad:

1. *Vynútenie generovania istých sekvencií inštrukcií.* Takéto obmedzenie zaručuje, že po vygenerovaní inštrukcie s týmto obmedzením bude vygenerovaná konštantná séria ďalších inštrukcií. Toto je často používaná rutina v prípade skokových inštrukcií pri zreťazenej linke. Aby procesor nemusel pozastaviť alebo vyčistiť linku v prípade načítania podmienenej skokovej inštrukcie (alebo aby nemusel obsahovať iný špekulatívny HW), vkladá PNG za inštrukcie podmieneného skoku inštrukcie, ktoré nič nerobia (typicky NOP), než dôjde k tomu, že IC vyhodnotí podmienku a rozhodne, či sa má alebo nemá skok uskutočniť.

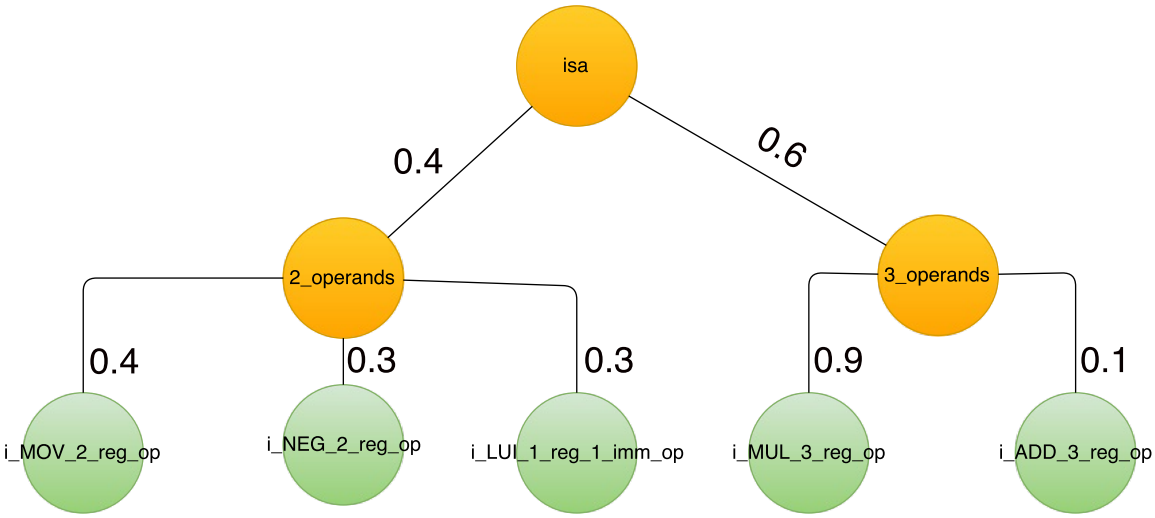
2. *Dočasný zákaz použitia premenných* – je obmedzenie, ktoré zabraňuje použitiu istej skupiny premenných (typicky registrov alebo miest v pamäti) v niekoľkých ďalších inštrukciách, aby nedošlo k štruktúrnemu hazardu. V prípade niektorých inštrukcií môže ich vykonávanie trvať až niekoľko hodinových cyklov, než dôjde k zápisu ich výsledku. Zavedená latencia (oneskorenie) sa postará o vylúčenie možných konfliktov spojených s predčasným čítaním/zápisom do premennej.

3. *Vylúčenie premenných* – zaisťuje, že v rámci jednej inštrukcie sú použité rôzne premenné. Príkladom môže byť inštrukcia pre zápis / čítanie z pamäte, ktorá pracuje s operandmi, v ktorých je nutné popísať základnú adresu (*base address*) a posun (*offset*) pomocou dvoch rôznych registrov, pretože výpočet adresy nie je realizovateľný súčasne z jedného registru.

Pravdepodobnostné rozloženie inštrukcií v programe

Pre účely generovania stimulov, v ktorých bude rozdielne rozloženie inštrukcií programu a ich operandov, je súčasťou konfigurácie PNG mechanizmus schopný popísať tieto časti programu pomocou váhových koeficientov. V podkapitole 4.1 bolo vysvetlené, že výsledkom

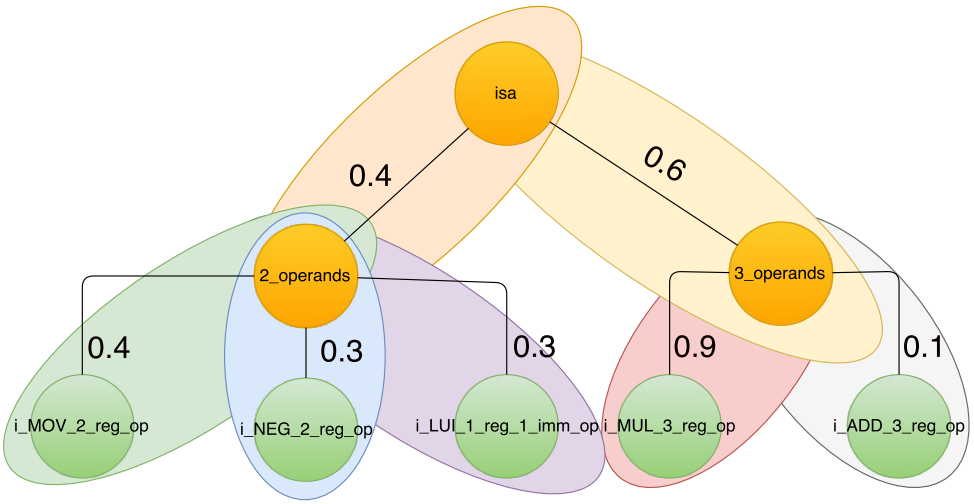
návrhu IA modelu je popis setov a elementov tvoriaci štruktúru acyklického grafu. Optimalizovaná forma tohto grafu je popísaná vo vygenerovanej konfigurácii znázornenej na Obr. 4.4. Táto optimalizovaná forma grafu stále obsahuje štruktúru elementov a setov, pričom pri popise setov je v tomto prípade možné priradiť váhu prvkom, ktoré set obsahuje. Takto váhovaný model v praxi umožňuje generovanie programov, ktorých niektoré inštrukcie sa vygenerujú s vyššou pravdepodobnosťou než iné a niektoré subinštrukcie (napr. operandy inštrukcie) sa vygenerujú s nižšou pravdepodobnosťou než iné operandy validné v generovanom inštrukčnom kontexte (viď 4.5). Pravdepodobnosť vygenerovania elementu PNG je tak daná súčinom váh jeho cesty vynásobeným pravdepodobnosťou vygenerovania najvyššieho setu.



Obr. 4.5: *Časť acyklického grafu z príkladu 4.1 s priradenými váhami.*

4.4 Komunikácia siete s verifikačným prostredím

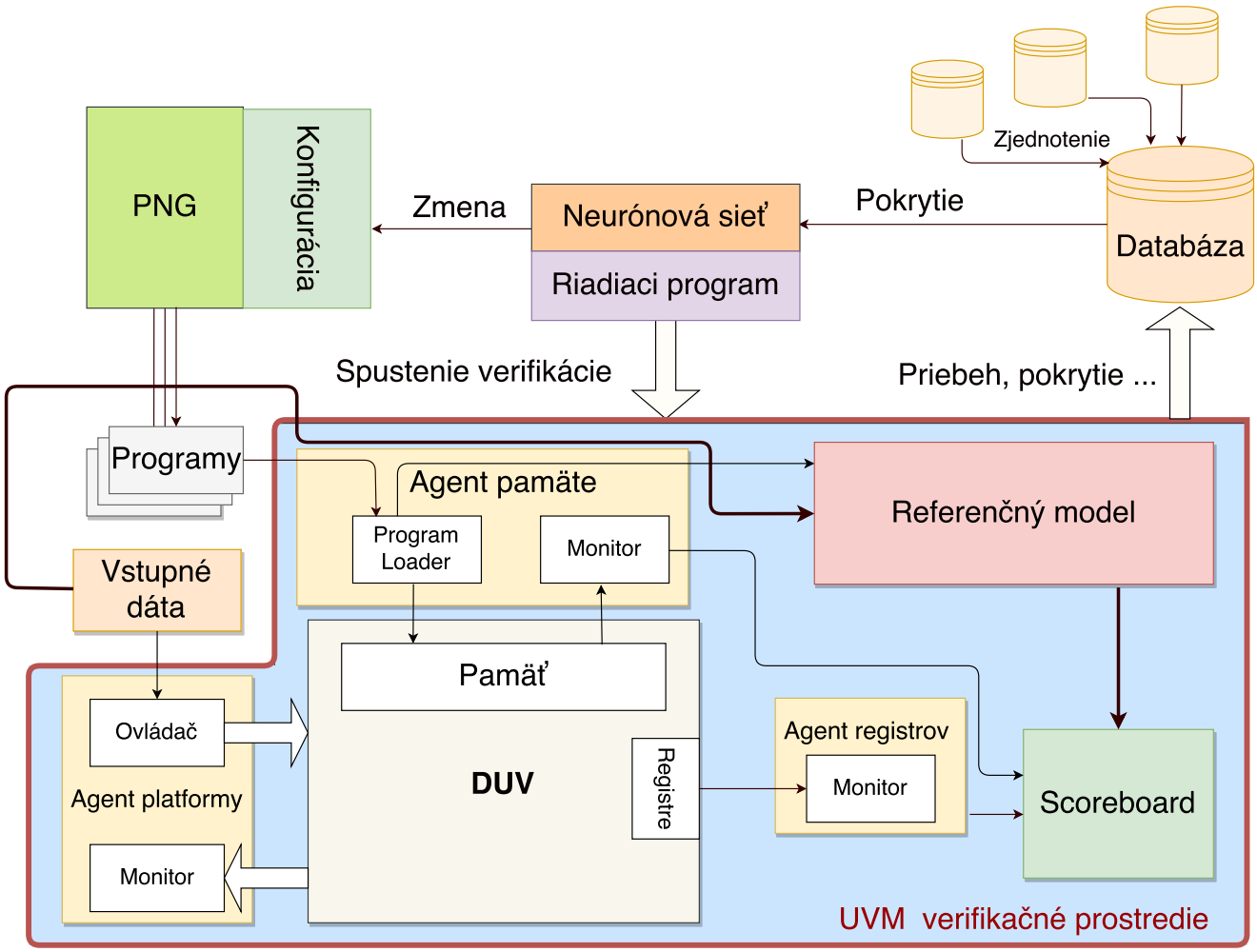
Neurónová sieť (NN) založená na Hopfieldovom modeli bude v konfigurácii PNG prehľadávať stavový priestor pomocou nastavovania váh váhovaného modelu acyklického grafu. Jeden neurón siete teda zodpovedá unikátnej dvojici tvorenej setom a jeho prvkom v acyklickom grafe. Stav každého takto vytvoreného neurónu potom zodpovedá pravdepodobnosti vygenerovania prvku v sete, ktorým tento neurón odpovedá. Táto hierarchia je znázornená na obrázku 4.6.



Obr. 4.6: *Ilustrácia neurónov nad acyklickým grafom z príkladu 4.5. Sieť vytvorená na základe tohto príkladu bude obsahovať 7 neurónov.*

V každej svojej epoche sieť nastaví váhy PNG (viď napr. implementovaný algoritmus 3) a následne spustí verifikačné prostredie v simulátore. RTL simulátory (ako napr. QuestaSim či VCS) umožňujú verifikáciu krokovať a kontrolovať tak jej priebeh, ako aj analyzovať aktuálny stav skupín pokrytia a výsledok verifikácie. V nasledujúcom texte budeme priebeh verifikácie až do jej konca nazývať *verifikačným behom*, alebo skrátene len *behom*. Informácie získané z behu simulácie je možné serializovať vo forme databáz, z ktorých vie NN získať dáta pre výpočet jej energetickej funkcie až keď je to potrebné. Nad týmto databázami je možné vykonávať štandardné databázové (resp. množinové) operácie ako prienik či zjednotenie. Tieto operácie, predovšetkým zjednotenie databáz, nám umožnia náhľad do histórie behu verifikácie riadenej neurónovou sieťou. Navrhovaná NN totiž nebude brať do úvahy len informácie získané z behu aktuálneho verifikačného behu, ale aj tie, ktoré boli akceptované minulými krokmi neurónovej siete v predchádzajúcich behoch.

Výhodou takéhoto riešenia je hlavne jeho nezávislosť na verifikačnom prostredí, keďže samotná optimalizačná heuristika beží nezávisle nad verifikačným prostredím a pseudonáhodným generátorom programov. Diagram takéhoto prístupu je znázornený obrázkom 4.7.



Obr. 4.7: *Ilustrácia behu neurónovej siete nad verifikačným prostredím a PNG.*

Kapitola 5

Návrh neurónových sietí

Heuristika implementovaného riešenia musí byť schopná prehľadávať stavy priestoru pokrytia určené definíciou `covergroup` konštrukcií vo verifikačnom prostredí. Táto heuristika musí preto spĺňať určité požiadavky stanovené v tejto kapitole. Model neurónovej siete, ktorý v tejto práci využijeme bude inšpirovaný Hopfieldovým modelom uvedeným v podkapitole 2.3. Definujeme nasledujúce požiadavky pre vlastnosti heuristiky a návrh rôznych foriem dynamiky siete.

1. Heuristika sa musí svojím riešením čo najviac priblížiť k uzáveru pokrytia.
2. Riešenie pomocou heuristiky by malo byť efektívnejšie, než náhodné riešenie s prednastavenou konfiguráciou PNG.
3. Heuristika musí byť schopná adaptovať sa akémukoľvek verifikovanému procesoru.
4. Spôsob zapojenia heuristiky do procesu optimalizácie by mal byť prenosný na ktorékoľvek verifikačné prostredie.
5. Spôsob zapojenia heuristiky do procesu optimalizácie by mal byť prenosný na akýkoľvek PNG s podporou pravdepodobnostného rozloženia podobnému váhového modelu.
6. Sieť by nemala uviaznuť v lokálnych minimách energetickej funkcie.
7. Sieť by mala byť schopná operovať na celom obore pravdepodobnosti váh pre každú nastaviteľnú váhu. Formálne ak N je množina neurónov, V je množina všetkých možných stavov siete a $t \in \mathbb{N}$ tak potom musí platiť 5.1.

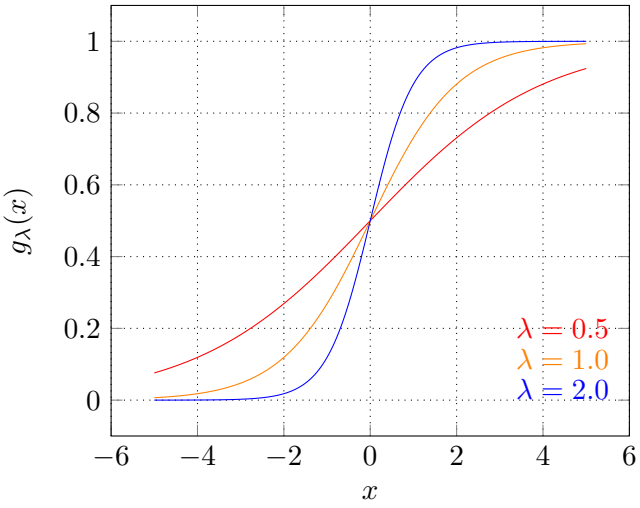
$$\forall n \in N, \forall i \in < 0, 1 >, \exists v_n \in V : v_n^{(t)} = i \tag{5.1}$$

5.1 Pevná konfigurácia siete

5.1.1 Dynamika siete

Aby bola sieť aplikovateľná na váhový model pravdepodobností, je nutné zvoliť vhodnú dynamiku siete. V prvom rade bude treba zvoliť bázovú a aktivačnú funkciu neurónov. Bázovú funkciu ξ zvolíme tak, aby kumulovala potenciál z pripojených neurónov pomocou sumy (2.1). Aktivačná funkcia Ψ musí byť schopná pokryť interval hodnôt v rozsahu $< 0, 1 >$, tak aby bolo možné dosiahnuť požiadavku bodu 7 uvedenú v úvode tejto kapitoly.

Preto volíme Ψ ako logistickú funkciu g_λ (2.11), ktorej rýchlosť konvergencie k hodnotám 0 a 1 je ovplyvniteľná parametrom λ (Obr. 5.1). Optimálna veľkosť tohto parametra je experimentálne odhadnutá v kapitole 7.1.



Obr. 5.1: *Tvar funkcie g_λ s rôznym nastavením hodnôt parametru λ.*

Po zvolení tvaru zobrazení v sieti je nutné špecifikovať poradie aktivácie neurónov. V prípade, že by sme zvolili synchrónny systém, bolo by nutné aktualizovať v každom kroku všetky neuróny siete z ich minulých stavov. Tým by sme ale vytvorili deterministickú sieť, ktorá z počiatočného stavu vždy konverguje k tomu istému cieľovému stavu. Naviac, podobný prístup by bol z praktického hľadiska veľmi neefektívny, pretože by sme ním zmenšili dostupný stavový priestor problému – ak by sme raz narazili na stav, ktorý nezlepšil energiu siete, sieť by bola zaseknutá v lokálnom minime, bez možnosti dostať sa z neho. Preto aby sme sa priblížili k požiadavke bodu 6, bola zvolená aktualizácia stavov pomocou asynchrónneho systému (viď 2.3.1). Nad asynchrónnym systémom sme naviac pre účely optimalizácie vytvorili **tabu zoznam**.

Tabu zoznam predstavuje množinu obsahujúcu také neuróny siete, ktoré už nemôžu byť pri aktivácii náhodného neurónu vybraté – ak je vybraný neurón obsiahnutý v tejto množine, volí sa ďalší náhodný neurón. Do tabu zoznamu sa môže neurón dostať takým spôsobom, že dôjde k jeho aktivácii a tá neprispeje k zlepšeniu energie siete. V prípade, že v sieti dôjde k prijatiu neurónu (teda nájde sa neurón, ktorého aktivácia zlepšia energiu siete), je tento zoznam vyprázdnený. Táto technika zabraní viacnásobnému výberu odmietnutého neurónu pre sieť v tom istom stave a urýchľuje tak konvergenciu siete k riešeniu optimalizačného problému.

5.1.2 Energetická funkcia

Navrhnutá energetická funkcia E bude pozostávať z rovníc vytvorených z metrík pokrytia verifikovaného modelu. Každému obmedzeniu c bola priradená relatívna váha a_c.

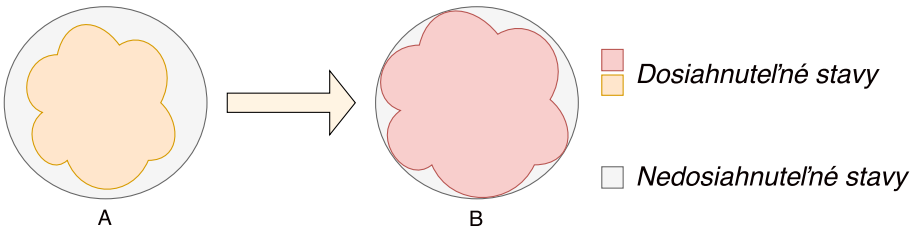
Obmedzenie metrikou pokrytia c (*Coverage*)	**Relatívna váha** a_c
Funkčné pokrytie (*Bin*)	1.0
Pokrytie príkazov (*Statement*)	0.0001
Pokrytie skokov (*Branch*)	0.0001
Pokrytie výrazov (*Expression*)	0.0001
Pokrytie konečných automatov (*FSM*)	0.0001
Totálne pokrytie (*Total*)	0.0001

Tabuľka 5.1: *Metriky pokrytia brané do úvahy pri výpočte energetickej funkcie a ich relatívne váhy.*

V tabuľke 5.1 sú uvedené metriky pokrytia, ktoré sú súčasťou energetickej funkcie riešenia a ich relatívne váhy a_c. Každá váha vyjadruje dôležitosť metriky pokrytia, ktorú reprezentuje v energetickej funkcii. Povšimnime si, že tabuľka obsahuje aj metriku s názvom *totálne pokrytie*. Táto metrika samozrejme koreluje s ostatnými použitými metrikami, keďže tie sú jej súčasťou.

Dôvody zakomponovania metriky totálne pokrytie vo funkcii E sú dva. Prvým je existencia ďalších metrík pokrytia v databáze RTL simulátoru, ktoré pri výpočte explicitne zohľadnené nie sú, ale sú implicitne zahrnuté práve v tejto metrike. Druhá príčina je implementačná – použitý RTL simulátor bol schopný spočítať metriky s absolútnou presnosťou len na 1 desatinné miesto. Pri jemnom náraste metrík, ktoré mali pre nás väčšiu dôležitosť sa tak vzhľadom k orezaniu výsledku nemusela táto zmena prejaviť.

Použitie širokej škály metrík rozširuje schopnosť heuristiky dosiahnuť okrajové prípady. Medzi jednotlivými metrikami existuje istá forma korelácie. Hypoteticky sa preto sa dá očakávať, že v prípade zlepšenia pokrytia menej významných metrík môže v najbližších krokoch dôjsť aj k zvýšeniu významnejšej metriky (funkčného pokrytia). Môžeme teda povedať, že takáto vlastnosť znižuje strmosť energetickej funkcie E, čiastočne tak zabraňuje uviaznutiu v lokálnych minimách (požiadavka 6) a tým zväčšuje dostupný stavový priestor (Obr. 5.2).



Obr. 5.2: *Ilustrácia rozšírenia stavového priestoru pokrytia neurónovej siete pridaním ďalších metrík.*

Keďže sa v tejto práci zameriavame najmä na funkčnú verifikáciu, pri návrhu relatívnych váh bol kladený dôraz na funkčné pokrytie a výrazný nárast hodnoty ostatných metrík nemal mať na hodnotu energie výrazný vplyv. Funkčné pokrytie je teda našou objektívnou funkciou f a keďže energiu minimalizujeme, hodnota znamienkovej konštanty m je rovná 1. Na základe rovnice 2.17 definujeme energetickú funkciu neurónovej siete 5.2,

$$E(\vec{v}) = \sum_{c \in C} a_c P(Coverage_c(\vec{v})) \tag{5.2}$$

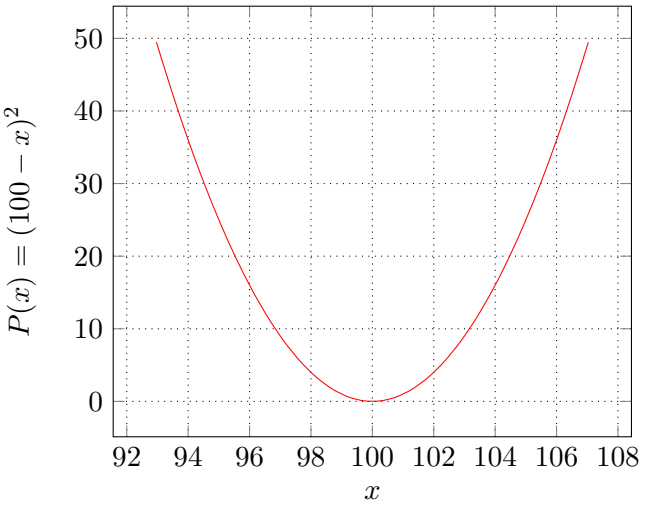
kde množina C definovaná ako 5.3

$$C = \{Bin, Statement, Branch, Expression, FSM, Total\} \tag{5.3}$$

reprezentuje jednotlivé typy pokrytia, koeficienty a_c vyplývajú z tabuľky 5.1, zobrazenie $Coverage_x(\vec{v})$, predstavuje aktuálne pokrytie modelu podľa metriky x pri neurónovej sieti v aktuálnom stave $\vec{v}$ a penalizačné zobrazenie P je definované kvadratickou funkciou 5.4.

$$P(x) = (100 - x)^2 \tag{5.4}$$

Konštanta 100 predstavuje maximálnu hodnotu akejkoľvek metriky pokrytia (100%) a keďže táto funkcia je kvadratická, je aj funkcia E kvadratická a nadobúda tvar paraboly (Obr. 5.3).



Obr. 5.3: *Tvar penalizačnej funkcie $P(x)$.*

5.2 Variabilná konfigurácia siete

Ďalej sa budeme zaoberať nastavovaním parametrov váh w a prahov θ. Súčasťou riešenia tejto práce je návrh takých parametrov, ktoré umožnia vhodne ovplyvniť pohyb po stavovom priestore pokrytia a dosiahnu aj také okrajové prípady, pri ktorých štandardné pseudonáhodné generovanie programov nestačí. Pri ich nastavovaní sa zameriame na experimentálne overenú skutočnosť, ktorá preukázala, že váhy pre váhový model PNG by nemali jednoznačne konvergovať k okrajom ich definičného oboru.

Ako príklad uvažujme model NN s 1000 neurónmi. V prípade, že by sieť akceptovala výstup nejakého neurónu s hodnotou blízkou 0 (v najhoršom prípade v ranných epochách jej evolúcie), zrejme v priebehu evolúcie existuje len nízka pravdepodobnosť, že k aktivácii tohto neurónu dôjde znovu. Ak je týmto neurónom nastavovaný napríklad výskyt nejakej inštrukcie v nejakom verifikačnom scenári, je pravdepodobné že k pokrytiu tohto prvku pokrytia už nedôjde. Naviac takáto aktivácia nemusí byť znovu prijatá.

Budeme sa teda snažiť vytvoriť modely, ktoré sú *váhovo vyvážené*. Podmienka platiaca pre váhový model NN je definovaná tak, aby v prípade, že majú všetky neuróny výstupy rovné 0.5 (polovici definičného oboru ich výstupného intervalu), aktivácia akéhokoľvek neurónu by nemala ich výstup zmeniť. Formálne nech N je množina indexov všetkých neurónov siete, potom pre váhovo vyvážené modely s množinou neurónov N v čase t platí 5.5.

$$\forall i \in N(v_i^{(t)} = 0.5) \implies \forall i \in N\left(v_i^{(t+1)} = \Psi\Big(\sum_{j=1}^{n} w_{ij} v_j^{(t)} + \theta_i\Big) = 0.5\right) \tag{5.5}$$

Preto navrhujeme nasledujúce modely prahov, váh a pravidlá pre prijatie aktivácie neurónu.

Jednotkový model

Prvým a najtriviálnejším modelom je Hopfieldova sieť so všetkými nediagonálnymi váhami rovnými 1. Prah každého neurónu i je daný hodnotou polovice počtu neurónov siete, s ktorými je tento neurón spojený nenulovou váhou (pričom vieme, že nulovou váhou je spojený iba sám so sebou). Tento model patrí medzi váhovo vyvážené modely. Jednotkový model pre sieť s množinou neurónov N, $n = |N|$ definujeme vzťahmi 5.6 a 5.7.

$$\forall i \in N, \forall j \in N\big((i = j \wedge w_{ij} = 0) \vee (i \neq j \wedge w_{ij} = 1)\big) \tag{5.6}$$

$$\forall i \in N\left(\theta_i = -\frac{n-1}{2}\right) \tag{5.7}$$

Bipolárny model

Ďalší navrhovaný koncept pozostáva z náhodného rozdelenia nediagonálnych váh každého neurónu do dvoch podobne veľkých skupín. Ak je počet nediagonálnych pripojení neurónu párny, potom tieto skupiny vytvoríme rovnomerne, jednej skupine priradíme váhy rovné 1 a druhej skupine priradíme váhy rovné -1. V prípade nepárneho počtu pripojení je nutné jednu skupinu váh vhodne upraviť tak, aby bol tento model za každých okolností váhovo vyvážený (viď 5.8). Tento model nie je Hopfieldovou sieťou, pretože jeho váhy pri akomkoľvek rozdelení do skupín nemôžu byť symetrické (bez dôkazu). Pre značnú zložitosť jeho formálneho zápisu definujeme priradenie váh algoritmom 1.

Algoritmus 1: Priradenie váh bipolárnemu modelu

Data: Počet neurónov n
Result: Matica váh $\mathbf{W}$ veľkosti $n \times n$
$znamienko \leftarrow 1$;
if *n je párne číslo* **then**
  $nepModif \leftarrow \frac{n-2}{n}$;
else
  $nepModif \leftarrow 1$;
end
for $i \leftarrow 0$ **to** n **do**
  $suma \leftarrow 0$;
  for $j \leftarrow 0$ **to** n **do**
    if $i = j$ **then**
      $w_{i,j} \leftarrow 0$;
    $suma \leftarrow suma + znamienko$;
    $w_{i,j} \leftarrow znamienko$;
    $znamienko \leftarrow -znamienko$;
  end
  if $nepModif \neq 1$ **then**
    for $j \leftarrow 0$ **to** n **do**
      `/* Suma určuje, ktorá skupina má viac prvkov */`
      if $(suma > 0$ ***and*** $w_{i,j} > 0)$ ***or*** $(suma < 0$ ***and*** $w_{i,j} < 0)$ **then**
        $w_{i,j} = nepModif * w_{i,j}$
      end
    end
  end
end
return $\boldsymbol{W}$

$$nepModif = \frac{n-2}{n} \tag{5.8}$$

Vzťah 5.8 v algoritme 1 vyjadruje veľkosť modifikácie tej skupiny váh, ktorá ma viac prvkov pri rozdelení, v prípade nepárneho počtu pripojení neurónu (počet nenulových pripojení neurónu je $n-1$). Prahy θ_i sú pre všetky neuróny i nulové.

Acyklický grafový model

Acyklický grafový model je rozšírením bipolárneho modelu. Ideou tohto modelu je vytvoriť také prepojenie medzi neurónmi, ktoré bude odpovedať tvaru acyklického inštrukčného stromu, s ktorým pracuje PNG. Každá dvojica setu a jeho prvku tvorí jednu unikátnu inštanciu $(set, member)$, ktorú optimalizuje neurónová sieť (viď 4.4), respektíve ktorá je reprezentovaná neurónom siete. V tomto modeli prepojíme iba také neuróny, ktorých odpovedajúce $(set, member)$ inštancie majú buď rovnaký set, alebo $member$ patriaci jednému neurónu je rovnaký ako set druhého neurónu. Nech každému neurónu s indexom i odpovedá unikátna inštancia c_i, obsahujúca informácie o dvojici $(set, member)$. Potom tento model definujeme pomocou algoritmu 2.

Algoritmus 2: Priradenie váh acyklickému grafovému modelu

Data: Počet neurónov n, Vektor unikátnych inštancií c
Result: Matica váh $\mathbf{W}$ veľkosti $n \times n$
$znamienko \leftarrow 1$;
for $i \leftarrow 0$ **to** n **do**
  $n_{prepojeni} \leftarrow 0$;
  $suma \leftarrow 0$;
  for $j \leftarrow 0$ **to** n **do**
    if $i = j$ **then**
      $w_{i,j} \leftarrow 0$;
      $n_{prepojeni} \leftarrow n_{prepojeni} + 1$;
    end
    else if $c_i.member = c_j.set$ ***or*** $c_i.set = c_j.member$ ***or*** $c_i.set = c_j.set$ **then**
      $w_{i,j} \leftarrow znamienko$;
      $suma \leftarrow suma + znamienko$;
      $znamienko \leftarrow -znamienko$;
      $n_{prepojeni} \leftarrow n_{prepojeni} + 1$;
    end
  end
  if $n_{prepojeni}$ *je párne číslo* **then**
    $nepModif \leftarrow \frac{n_{prepojeni}-2}{n_{prepojeni}}$;
  else
    $nepModif \leftarrow 1$;
  end
  if $nepModif \neq 1$ **then**
    for $j \leftarrow 0$ **to** n **do**
      if $(suma > 0$ ***and*** $w_{i,j} > 0)$ ***or*** $(suma < 0$ ***and*** $w_{i,j} < 0)$ **then**
        $w_{i,j} = nepModif * w_{i,j}$
      end
    end
  end
end
return $\boldsymbol{W}$

Takýmto prístupom zrejme nemusí vzniknúť (a zvyčajne ani nevznikne) plne prepojená sieť, čo je ďalšie porušenie definície Hopfieldovej siete. Alternatívne môžeme tvrdiť, že táto

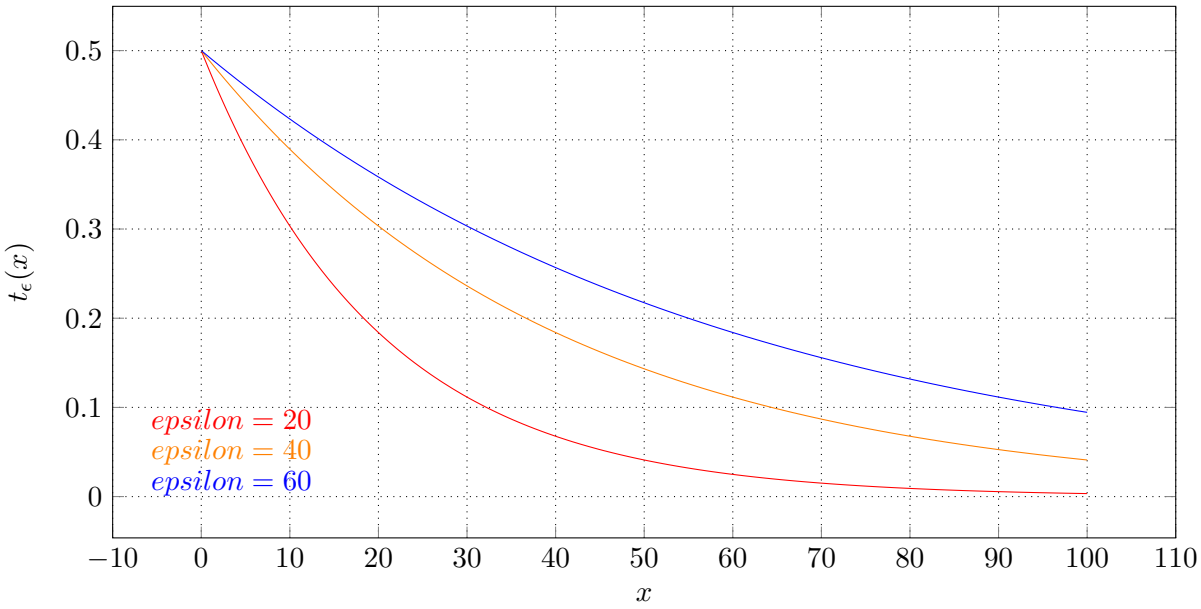
sieť stále je plne prepojenou, pričom chýbajúcim prepojeniam sme nastavili nulové váhy[1]. Podobne ako v prípade bipolárneho modelu sú prahy neurónov nulové.

Model s náhodným prijímaním aktivácie

Tento model je inšpirovaný modelom simulovaného žíhania aplikovanom na Hopfieldovej sieti (viď 2.3.5). Samotné simulované žíhanie na navrhované modely aplikovateľné nie je, kvôli navrhnutej energetickej funkcii E. Keďže pri zmene stavu siete z $v^{(t)}$ na $v^{(t+1)}$ dochádza k zjednoteniu databázy obsahujúcej súčasné pokrytie pri stave $v^{(t)}$, s tou ktorá vznikla v novom verifikačnom behu so stavom $v^{(t+1)}$, nemôže byť zmena ΔE získaná zo zjednotenia všetkých databáz predošlých behov nikdy záporná.

Bola preto navrhnutá alternatívna funkcia, ktorá plní podobný účel, avšak nie je funkciou energie E, ale len počtu iterácií. Aby sme dosiahli efekt podobne klesajúcej funkcie, použijeme exponenciálnu funkciu t_ϵ, ktorej rýchlosť klesania určíme vhodnou konštantou ϵ (experimentálne zvolíme na základe počtu iterácii optimalizácie prebiehajúcej na jednom z predchádzajúcich modelov). Tvar použitej funkcie bude daný predpisom 5.9.

$$t_\epsilon(x) = \frac{1}{2}\exp\left(\frac{-x}{\epsilon}\right) \tag{5.9}$$



Obr. 5.4: *Tvar funkcie t_ϵ s rôznym nastavením hodnôt parametru ϵ.*

V prípade, že $\Delta E = 0$, akceptujeme porušený stav $v^{(t+1)}$ s pravdepodobnosťou 5.10

$$P(pertrubed \leftarrow current) = t(i) \tag{5.10}$$

kde i je index súčasnej iterácie.

[1] Takýto prístup sme zvolili pri implementácii.

Generovanie počiatočných hodnôt

V navrhnutých modeloch budeme experimentovať s niekoľkými možnosťami počiatočnej konfigurácie siete. Nech N je množina neurónov, pre každý neurón $i \in N$ definujeme nasledovné počiatočné výstupy.

1. $v_i^{(1)}$ je náhodná hodnota z intervalu $< 0, 1 >$. Hustota pravdepodobnosti výberu z náhodného intervalu je daná *rovnomerným rozdelením pravdepodobnosti*.

2. $v_i^{(1)}$ je náhodná hodnota z intervalu $< 0, 1 >$. Hustota pravdepodobnosti výberu z náhodného intervalu je daná *trojuholníkovým rozdelením pravdepodobnosti* s extrémom v hodnote 0.5.

3. $v_i^{(1)}$ je náhodná hodnota z intervalu $< 0.4, 0.6 >$. Hustota pravdepodobnosti výberu z náhodného intervalu je daná *rovnomerným rozdelením pravdepodobnosti*.

Tieto scenáre generovania počiatočných hodnôt boli navrhnuté na základe predbežných experimentov. Tie naznačovali vyššiu úspešnosť riešenia pri počiatočných hodnotách v okolí hodnoty 0.5. Túto hypotézu potvrdzujú aj výsledky experimentov 7.5.

Kapitola 6

Implementácia

Táto kapitola ponúka čitateľovi oboznámenie sa s detailmi algoritmov navrhnutých nad rôznymi modelmi neurónovej siete, s detailmi prostredia, nad ktorým pracujú, náhľad na technickú realizáciu riešenia a popis jadier, na ktorých bola implementácia testovaná a na ktorých boli následne uskutočnené experimenty. Implementácia bola realizovaná v jazyku Python s revíziou interpretéra 2.7.9.

6.1 Architektúra riešenia

Pri implementácii uvedenej siete a prostredia bolo v jazyku Python vytvorených niekoľko konštrukcií, ktoré si v tejto podkapitole predstavíme. Ich pochopenie je kľúčom k porozumeniu algoritmom nasledujúcej kapitoly 6.2 a implementácii dostupnej k tejto práci.

Konfigurácia psuedonáhodného generátoru programov

Trieda `NodeConfig` obsahuje prostriedky pre internú reprezentáciu uzlov acyklického grafu, získaného z IA popisu verifikovaného modelu. Nástroje CS analýzou tohto popisu vytvoria súbor, ktorý je v implementácii načítaný (metóda `NodeConfig.parsePdm`) a je na ňom prevedená syntaktická analýza, ktorej výstupom je kompletný acyklický graf vygenerovaný Codasip Studiom (metóda `NodeConfig.createAcyclicGraph`). Uzly tohto acyklického grafu sú reprezentované inštanciami triedy `PdmNode`.

Tento acyklický graf je následne sekvenčne prejdený a sú z neho vybrané všetky dvojice (set, member) (metóda `NodeConfig.getWeightedNodeList`), ktoré vytvoria množinu vážených uzlov. Táto množina je internou reprezentáciou, ktorá je použitá v nasledujúcich prípadoch:

- Pri tvorbe neurónovej siete je podľa nej vytvorený správny počet neurónov.
- Pri akceptovaní zmeny v neurónovej sieti sú tieto zmeny reflektované do tejto množiny (metóda `Hopfield.applyNetToConfig`).
- Pred generovaním pseudonáhodných programov je typicky obsah tejto množiny aplikovaný na PNG (metóda `RngManager.applyConfigToRnG`).
- V acyklickom grafovom modeli (podkapitola 5.2) sú podľa tejto množiny vytvorené prepojenia medzi neurónmi siete.

Generátor pseudonáhodných programov

Abstrakciou nad pseudonáhodným generátorom programov je trieda `RngManager`. Tá zapuzdruje prácu s dynamickou knižnicou jazyka C dostupnou u použitého PNG z Codasip Studia, ako aj s nástrojmi vygenerovanými pomocou Codasip Studia pre nami verifikované modely (*prekladač, linker*). Trieda umožňuje inicializovať PNG pomocou konfiguračného súboru vygenerovaného Codasip Studiom a konfiguračného súboru dopísaného užívateľmi (viď 4.4). Naviac, keďže súčasťou PNG sú klasické pseudonáhodné kongruentné generátory náhodným veličín, stará sa aj o počiatočné číslo pre tieto generátory (*initial seed*).

Trieda teda umožňuje generovanie programov v asembleri popísaného modelu, stará sa o ich preklad a ich korektné uloženie do súborového systému (metóda `RngManager.runGen`). Pomocou internej konfigurácie PNG je tiež jej prostredníctvom možné aplikovať zmeny váh jednotlivých (set,member) inštancií na PNG (metóda `RngManager.applyConfigToRnG`).

Nastavenia

V Python module `settings.py` sú dostupné nasledujúce nastavenia (6.1).

```
#parameter lambda u sigmoidy
LAMBDA=0.9
#parameter epsilon pre model s nahodnym prijimanim aktivacie
EPSILON=50.0
#maximalny pocet iteracii pri priebeznej optimalizacii
MAXITERATIONS=400
#maximalna dlzka epochy pri hladani optimalnej mnoziny programov
MAXEPOCHLENGTH=30
#pocet programov z PNG pre tu istu konfiguraciu pri priebeznej optimalizacii
PROGRAM_COUNT=1
#pocet programov z PNG pre tu istu konfiguraciu hladani
#optimalnej mnoziny programov
BESTPROGRUNS=3
#maximalna vaha, ktoru moze prvok setu dostat
MAXWEIGHT=100
#dlzka vygenerovanych programov
INSTRUCTION_COUNT=100

#Udaje o prostredi modelu
#URISC CONFIGURATION
CODASIP_URISC_PATH="/home/martin/codasip1402/codasip_urisc/"
CODASIP_URISC_IA_PREFIX="codasip_urisc-ia-"
RTLSIM_START_CMD='questasim_start_cmd.sh'
RTLSIM_MERGE_CMD='questasim_coverage_merge.sh'
PATH_TO_STUDIO ="/home/martin/CStudio6.4.1"
MODEL_PATH=CODASIP_URISC_PATH
MODEL_PREFIX = CODASIP_URISC_IA_PREFIX
```

Kód 6.1: Niektoré nastavenia parametrov dostupné v implementovanom riešení.

Ďalšie utility

Modul `utility.py` obsahuje funkcie pre synchrónne aj asynchrónne spúšťanie verifikačného behu, či čistenie prostredia od dočasných súborov, modul `mergedreport.py` poskytuje funkcie pre prácu s databázou produkovanou použitým RTL simulátorom a získavanie informácie o verifikáciou dosiahnutom pokrytí. Pre účely experimentov a získavania dát z výstupných súborov riešenia boli vytvorené moduly `experiments.py` a `runlogparser.py`.

6.2 Riadiace algoritmy

Keďže v úvode sme si predstavili dva optimalizačné problémy (1.2), museli byť vytvorené dva rôzne prístupy k ich riešeniu.

Prvým stanoveným problémom je priebežná optimalizácia konfigurácie PNG pre akékoľvek verifikované jadro. Na rozdiel od čiste náhodného prístupu podmieneného len počiatočnou konfiguráciou PNG, aplikácia priebežnej optimalizácie by mala umožniť priebežne meniť charakter vygenerovaných programov a dosiahnuť tak rýchlejšie a častejšie pokrytie okrajových prípadov. Pre účely riešenia tohto problému bol vytvorený algoritmus 3 pracujúci nad neurónovou sieťou. Tento algoritmus je implementovaný v module `continuousoptimization.py` a jeho okomentovaný kód je možné nájsť aj v prílohách A.1 a A.2.

Algoritmus 3: Navrhnutý algoritmus priebežnej optimalizácie konfigurácie PNG

Result: Ak v žiadnom verifikačnom behu nedošlo k chybe, bol dosiahnutý uzáver pokrytia
Inicializácia PNG;
Načítanie počiatočnej konfigurácie PNG;
Inicializácia neurónovej siete;
Vygenerovanie množiny programových stimulov s počiatočnou konfiguráciou PNG;
Beh verifikačného prostredia nad touto množinou programov;
Získanie pokrytia verifikačných behov;
Inicializácia energie siete zo získaného pokrytia;
Inicializácia prázdneho tabu zoznamu;
for *v evolučnom čase* t *od* 0 *do* `settings.MAXITERATIONS` **do**
    if *tabu zoznam je plný* **then**
        Sieť uviazla v minime – optimalizácia sa ukončí;
    end
    Aktivácia náhodného neurónu;
    Aplikácia nového stavu siete na PNG;
    Generovanie programových stimulov;
    Beh verifikačného prostredia nad novou množinou stimulov;
    Získanie aktuálne dosiahnutého pokrytia;
    Výpočet novej energie zo získaného pokrytia;
    if *došlo k poklesu energie siete* ***or*** $\big($*používame model s náhodným prijímaním aktivácie* ***and*** $Random(0,1) < t_\epsilon(t)\big)$ **then**
        Aktivácia je prijatá a sieť sa dostáva do nového stavu;
        Uloží sa dosiahnutá nová energia;
        Vymaže sa obsah tabu zoznamu;
    else
        Aktivácia je odmietnutá a sieť sa vráti do pôvodného stavu;
        Posledný aktivovaný neurón sa pridá do tabu zoznamu;
    end
end

Druhým problémom je hľadanie najmenšej množiny programov takej, že nájdená množina dosiahne uzáver pokrytia verifikovaného IC. Pre účely implementácie tohto algoritmu bol vytvorený riadiaci cyklus 4, implementovaný v module `bestprograms.py`.

Algoritmus 4: Navrhnutý algoritmus hľadania optimálnej množiny programov

Result: Ak v žiadnom verifikačnom behu nedošlo k chybe, bol dosiahnutý uzáver pokrytia a priečinok najlepších programov obsahuje minimálnu nájdenú množinu programov pre dosiahnutie pokrytia.

Inicializácia PNG;
Načítanie počiatočnej konfigurácie PNG;
Inicializácia neurónovej siete;
Tvorba priečinkov pre dočasné súbory a najlepšie programy;
Inicializácia premenných (počiatočná energia, najlepší nájdený program...);
for *`settings.BESTPROGRUNS` krát* **do**
    Vygenerovanie programu s počiatočnou konfiguráciou PNG;
    Spustenie behu verifikačného prostredia;
    Získanie aktuálne dosiahnutého pokrytia;
    Získanie aktuálnej energie pokrytia;
    if *aktuálna energia je najlepšia doteraz* **then**
        if *existuje už nejaký najlepší program* **then**
            Vymaže sa posledný najlepší program a jeho databáza z priečinku najlepších programov;
        end
        Do priečinku najlepších programov sa uloží aktuálny program a jeho databáza;
        Podľa aktuálneho programu sa aktualizujú premenné popisujúce najlepší nájdený program;
    end
end
Inicializácia tabu zoznamu na prázdny zoznam;
if *`settings.MAXEPOCHLENGTH`> počet neurónov siete* **then**
    Dĺžka epochy = počet neurónov siete;
else
    Dĺžka epochy = `settings.MAXEPOCHLENGTH`;
end
while *Funkčné pokrytie < 100%* **do**
    for *Dĺžka epochy krát* **do**
        Hľadanie takej aktivácie neurónu, ktorá najviac prispeje k pokrytiu modelu;
    end
    Aktivácia najlepšieho neurónu po preskúmaní stavového priestoru v priebehu epochy;
    Presun zjednotenej databázy získanej pri aktivácii najlepšieho neurónu do pracovného priečinka siete;
    Vyprázdnenie tabu zoznamu;
    Reinicializácia premenných;
end

Pre úplnosť uvedieme aj algoritmus 5, ktorý popisuje ako prebieha hľadanie aktivácie najlepšieho neurónu v rámci jednej epochy. Kľúčové časti implementácie týchto algoritmov v jazyku Python je možné v okomentovanej forme nájsť v prílohách A.3 a A.4.

Algoritmus 5: Hľadanie takej aktivácie neurónu, ktorá najviac prispeje k pokrytiu modelu

```
Aktivuj náhodný neurón a vlož ho do tabu zoznamu;
Aplikácia nového stavu siete na PNG;
for settings.BESTPROGRUNS krát do
    Záloha databázy obsahujúcej pokrytie z predošlej epochy;
    Vygenerovanie programu s aktuálnou konfiguráciou PNG;
    Spustenie behu verifikačného prostredia;
    Zjednotenie s databázou predošlej epochy (ak existuje) a získanie pokrytia;
    Získanie aktuálnej energie siete;
    if aktuálna energia je najlepšia doteraz then
        if existuje už nejaký najlepší program then
            Vymaže sa posledný najlepší program a jeho databáza z priečinku
            najlepších programov;
        end
        Do priečinku najlepších programov sa uloží aktuálny program a jeho
        databáza;
        Podľa aktuálneho programu sa aktualizujú premenné popisujúce najlepší
        nájdený program;
    end
    Náhrada aktuálnej databázy za databázu predošlej epochy, ak existovala;
end
Obnova siete do predošlého stavu;
```

6.3 Verifikované modely

Implementácia tejto práce bola testovaná na dvoch modeloch ASIP vyvinutých spoločnosťou Codasip. Prvým bol ASIP s názvom *Codasip uRISC*[1](ďalej uRISC). Ako už názov napovedá, jedná sa o 32-bitovú mikroarchitektúru typu RISC so 4-stupňovou zreťazenou linkou, slúžiacu predovšetkým na výukové a demonštratívne účely. Inštrukčná sada tohto modelu je veľmi jednoduchá a neurónová sieť popisujúca (element, set) acyklický graf v prípade tohto modelu obsahuje len 41 neurónov.

Ďalší testovací model má názov *Codix Cobalt*[2] (ďalej len Cobalt). Tiež sa jedná o 32-bitovú mikroarchitektúru typu RISC, avšak s 5-stupňovou zreťazenou linkou. Je to produkčné jadro stavané na vysoký výkon s optimalizáciami pre syntézu. Ďalšie technické informácie je možné nájsť v [5]. Neurónová sieť vytvorená nad týmto modelom obsahuje 1020 neurónov. Kódovanie inštrukčnej sady medzi týmto modelmi nie je kompatibilné, jedná sa teda o dva odlišné modely. Samotná implementácia prebehla práve pri priebežnom testovaní nad modelom uRISC a až následne bolo experimentálne overené, že navrhnutá heuristika je funkčná aj pre model Cobalt.

[1] ASIP Codasip uRISC vydaný ku Codasip Studiu s verziou 6.4.1.
[2] ASIP Codix Cobalt dostupný ku Codasip Studiu s verziou 6.5.1.

Kapitola 7

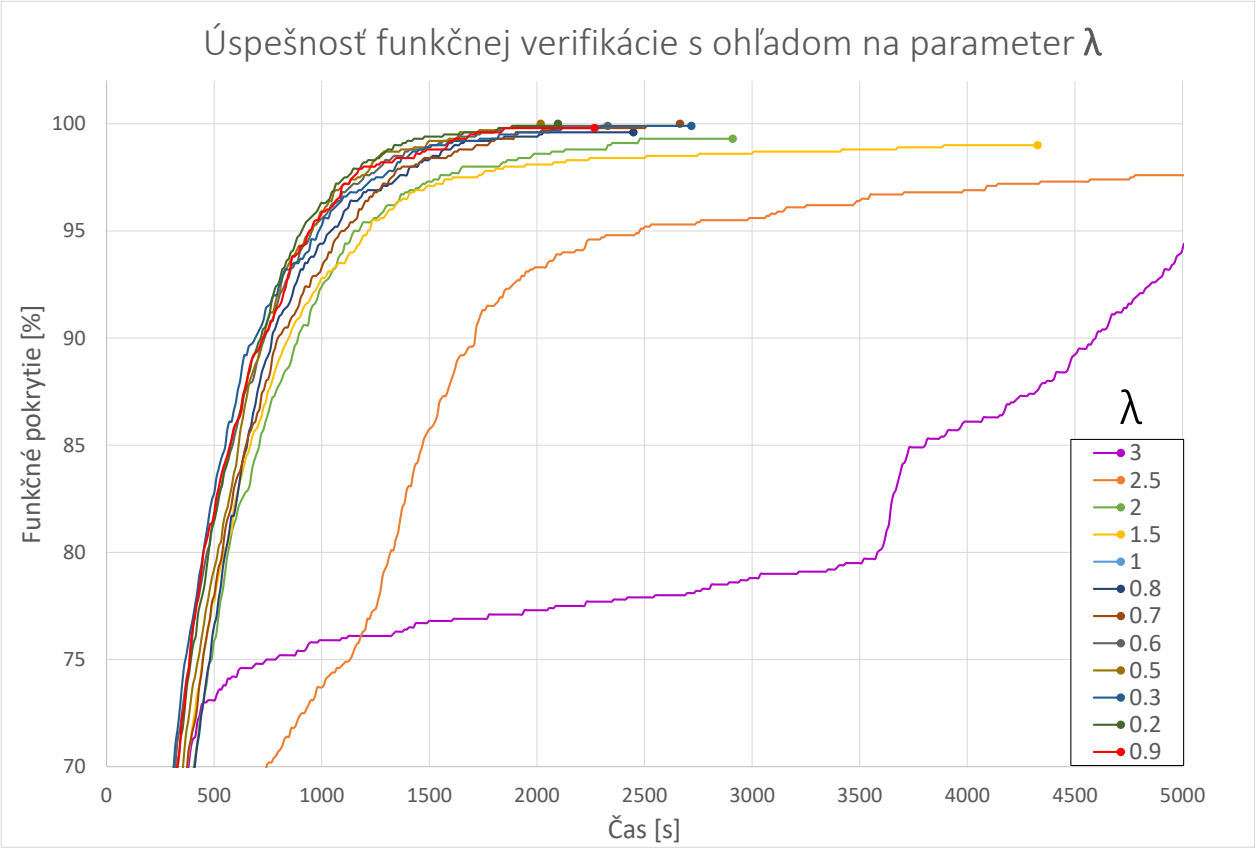
Experimenty

Cieľom tejto kapitoly je experimentálny odhad parametrov neurónovej siete a porovnanie efektivity dosahovania uzáveru pokrytia pri použití modelov navrhnutých v kapitole 5. Všetky uvedené experimenty boli realizované na notebooku s procesorom Intel Core i7 3610QM, 8GB RAM a 64-bitovým linuxovým operačným systémom Debian 8.7. Vo všetkých experimentoch boli pomocou PNG generované programy pre verifikované ASIP o dĺžke 100 inštrukcií. Tieto inštrukcie ale nezahŕňajú prológ programu, epilóg programu, inicializáciu registrov IC a niektoré špeciálne inštrukcie, pred ktorými musí byť zavolaná špecifická sada inštrukcií. V skutočnosti tak programy boli o niečo dlhšie. Experimentálny beh s názvom *Random* predstavuje medián z experimentov realizovaných sústavným generovaním programov cez PNG s počiatočnou konfiguráciou poskytnutou Codasip Studiom.

7.1 Codasip uRISC

7.1.1 Hľadanie optimálneho parametru λ

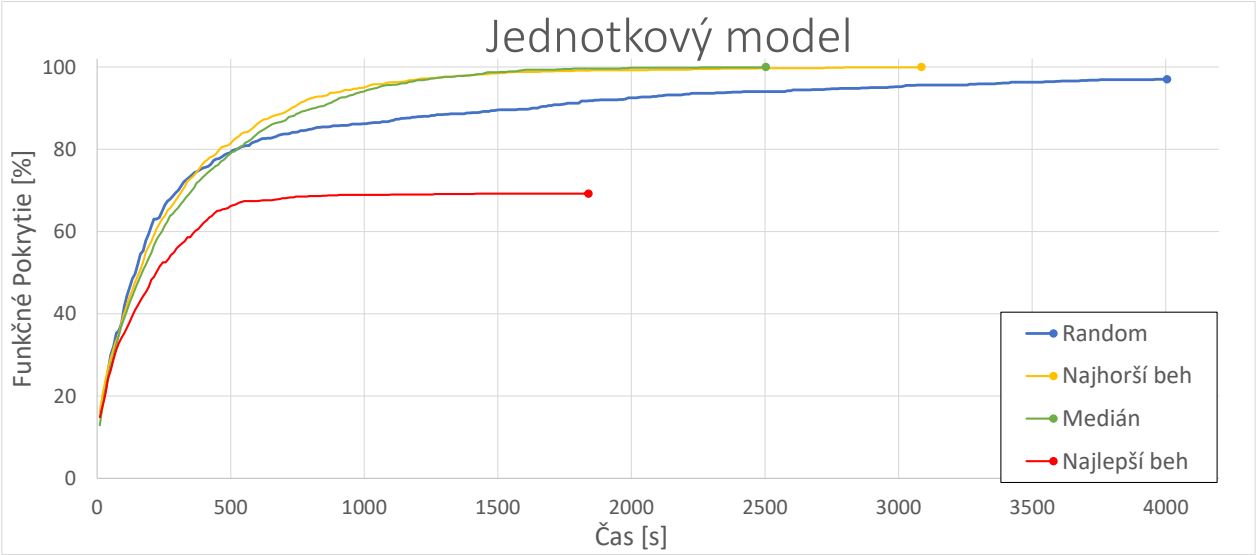
Parameter λ je parametrom strmosti sigmoidy (rovnica 2.11) použitej v každom navrhnutom neuróne. Cieľom experimentu, ktorého výstupy sú zobrazené na Obr. 7.1, bolo hľadanie vhodného tvaru tejto funkcie. Zvolili sme niekoľko hodnôt, ktorých chovanie v predbežných experimentoch naznačovalo rýchlejšiu konvergenciu k riešeniu a na každej z nich sme previedli 5 experimentov. Úspešnosť týchto experimentov sme následne usporiadali podľa času, ktorý zabral ich výpočet a na základe tohto poradia bol vybraný **medián**, ktorý bol zohľadnený vo výstupe experimentu. Tento spôsob výberu nemusí znamenať, že experimentálny beh bol naozaj najúspešnejší, mohlo napríklad dôjsť k zaseknutiu siete pri nízkom funkčnom pokrytí. Vo väčšine prípadov je však toto usporiadanie najvhodnejšie. Takýto prístup bol zvolený preto, aby nedošlo pri okrajových hodnotách experimentálnych behov k skresleniu. Napríklad v prípade priemeru by nám v žiadnom z experimentov nevyšlo funkčné pokrytie 100%, pretože aspoň v jednom experimente bola sieť zaseknutá.



Obr. 7.1: *Hľadanie optimálneho parametru* λ.

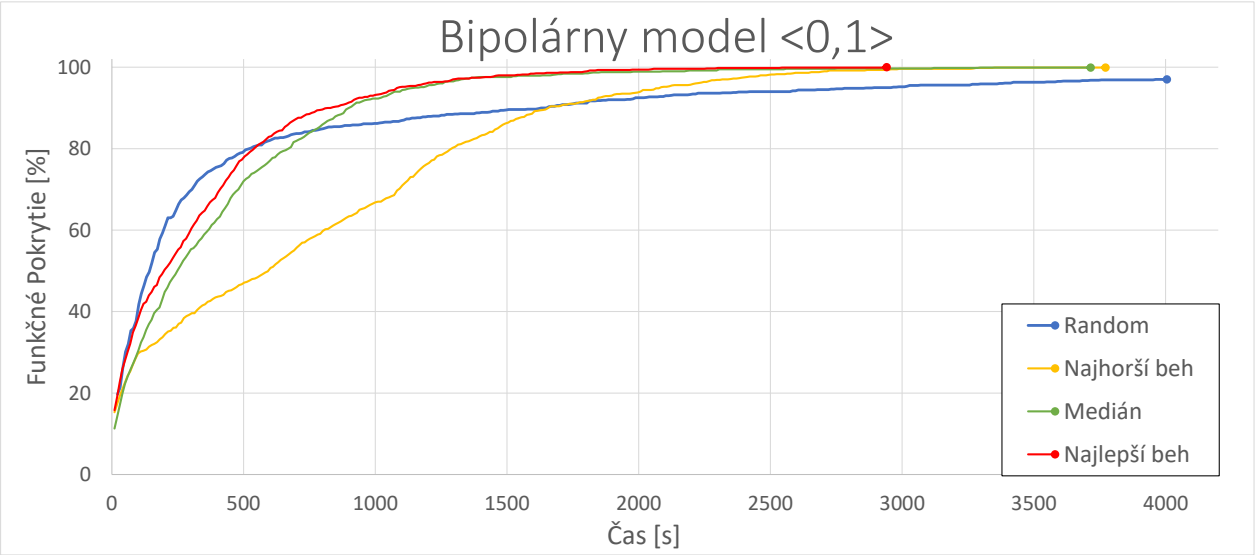
7.1.2 Navrhnuté modely priebežnej optimalizácie

Nasledujúce experimenty vyhodnocujú úspešnosť navrhnutých modelov určených na priebežnú optimalizáciu PNG. Parameter λ bol na základe experimentov podkapitoly 7.1.1 nastavený na hodnotu $\lambda = 0.9$. V experimentoch uvádzame vždy najlepší beh modelu, mediánový beh modelu a najhorší beh modelu v porovnaní s náhodným riešením *Random*. Priebeh experimentov bol zaznamenávaný rovnakým spôsobom ako v predchádzajúcej podkapitole.



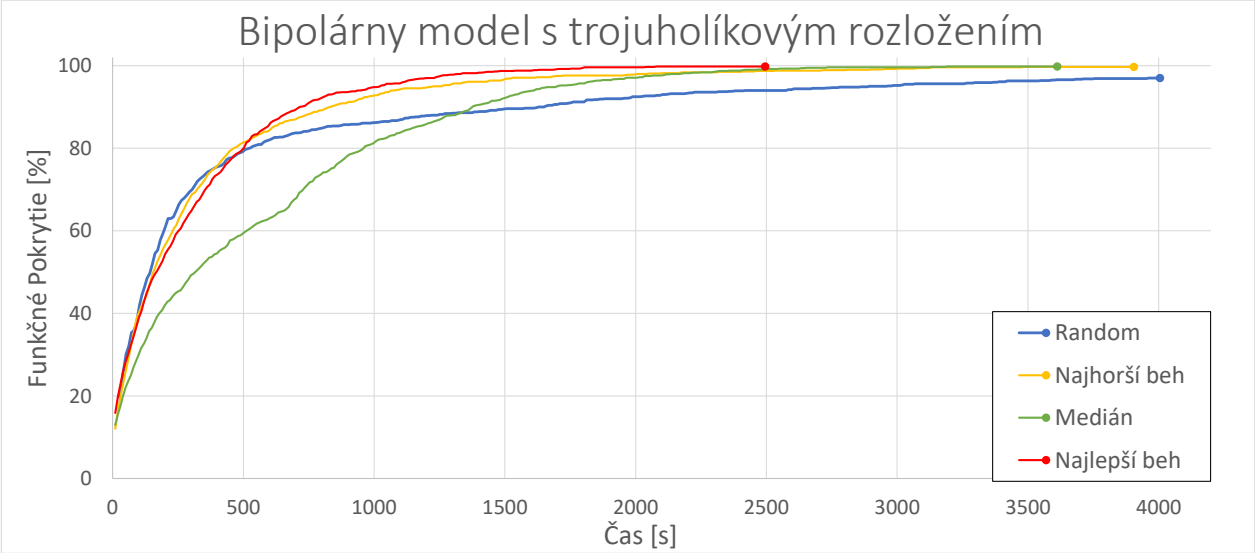
Obr. 7.2: *Priebeh experimentov na jednotkovom modeli.*

Všimnime si, že v tomto prípade (Obr. 7.2) sa najlepší beh v skutočnosti zasekol už na nízkych hodnotách funkčného pokrytia (konkrétne na pokrytí 69.2%). Aj v prípade niektorých ostatných nameraných hodnôt dochádzalo k zaseknutiu na nízkych hodnotách pokrytia (75.3%).

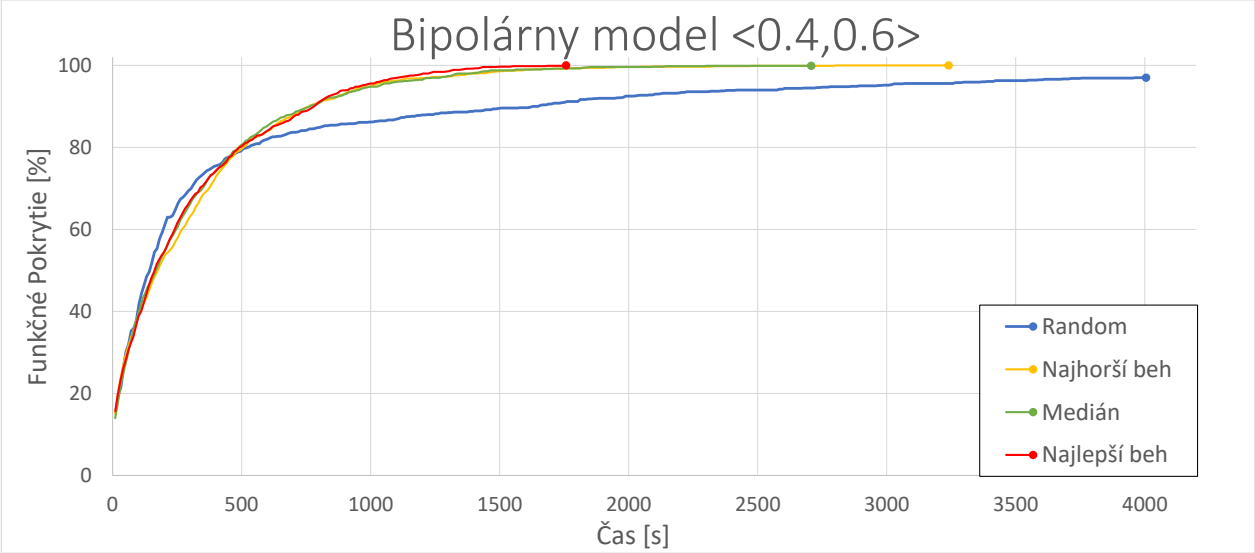


Obr. 7.3: *Experimentálne výsledky bipolárneho modelu s počiatočným stavom neurónov daným rovnomerným rozložením pravdepodobnosti v intervale* $< 0, 1 >$.

Výsledky uvedené na Obr. 7.3 sme sa pokúsili vylepšiť zmenou spôsobu výberu počiatočného stavu siete v experimentoch 7.4 a 7.5. Predbežné experimenty totiž naznačovali, že sieť dospela rýchlejšie k riešeniu, keď bol počiatočný stav jej neurónov v okolí hodnoty 0.5.

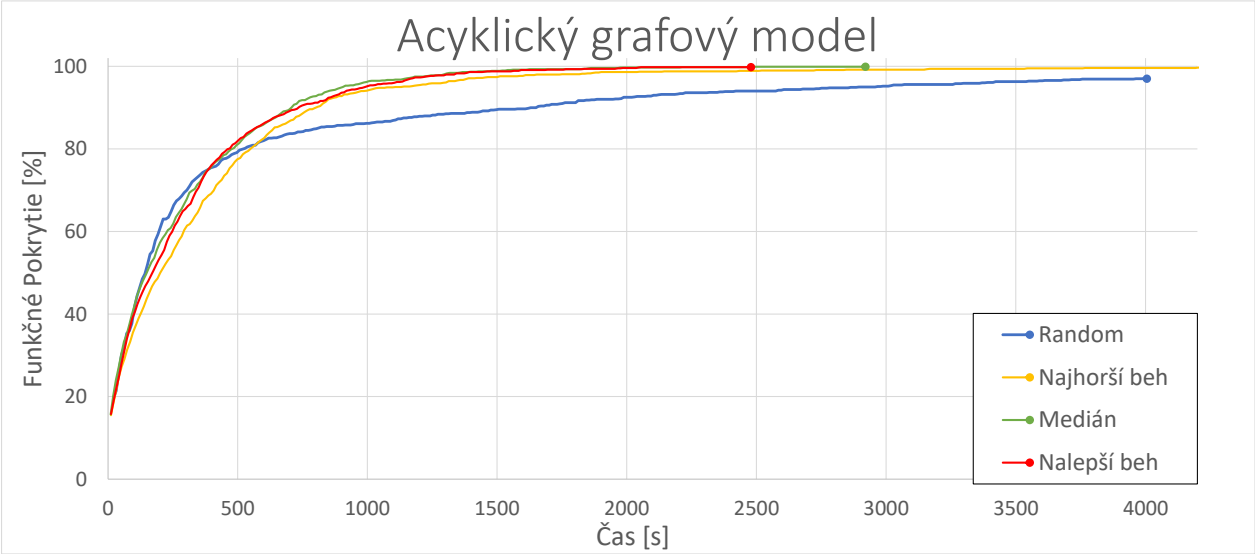


Obr. 7.4: *Experimentálne výsledky bipolárneho modelu s počiatočným stavom neurónov daným trojuholníkovým rozložením pravdepodobnosti v intervale* $< 0, 1 >$.

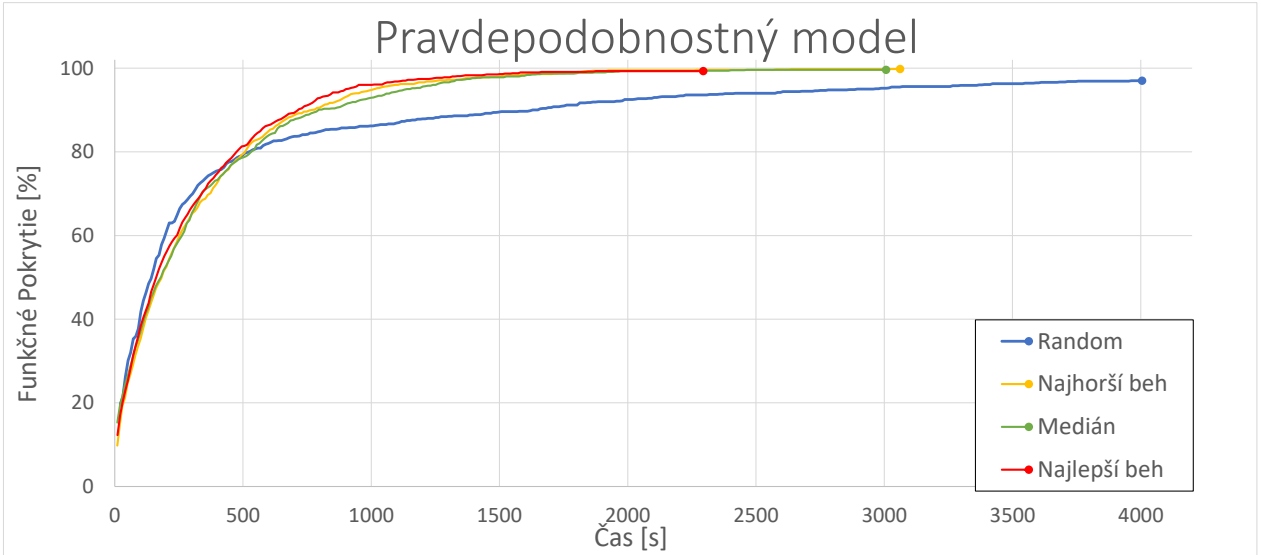


Obr. 7.5: *Experimentálne výsledky bipolárneho modelu s počiatočným stavom neurónov daným rovnomerným rozložením pravdepodobnosti v intervale* $< 0.4, 0.6 >$.

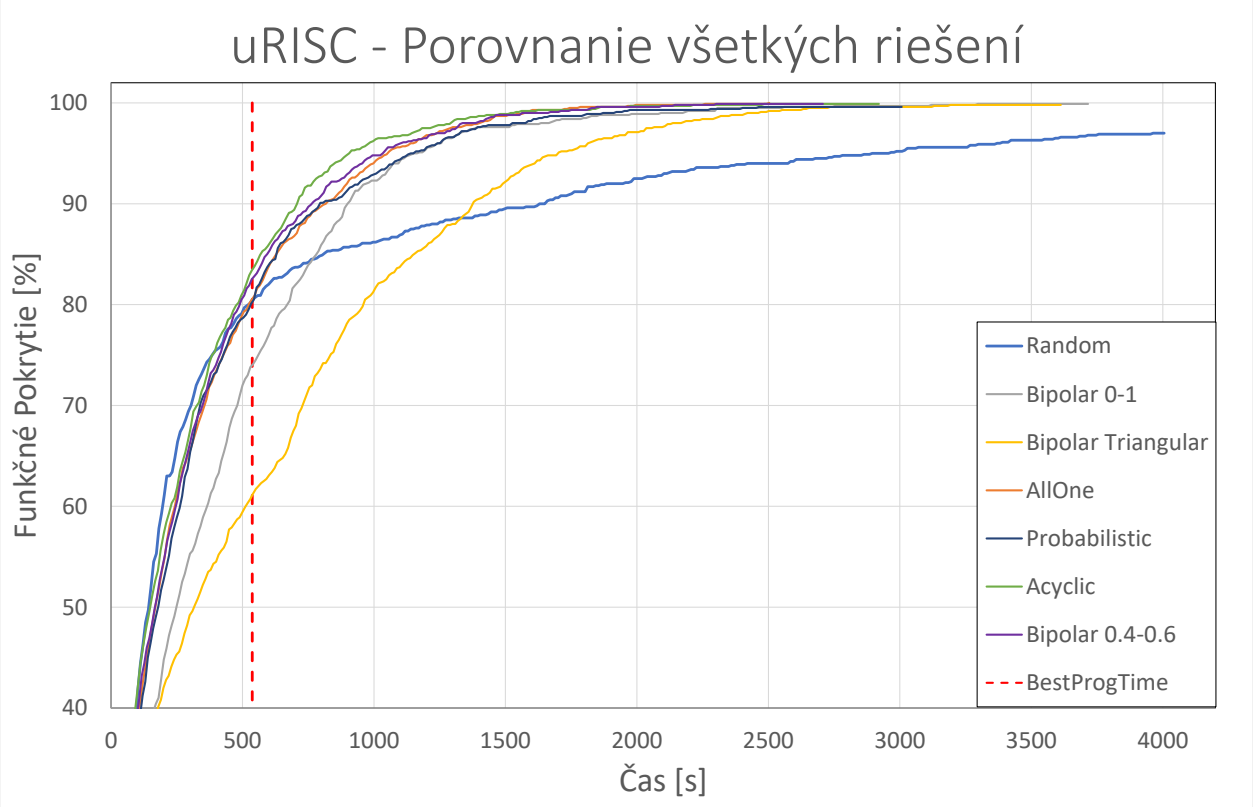
Nasledujúce modely 7.6, 7.7 sú rozšírením zatiaľ najúspešnejšieho bipolárneho modelu s počiatočným stavom neurónov daným rovnomerným rozložením pravdepodobnosti v intervale $< 0.4, 0.6 >$.



Obr. 7.6: *Priebeh experimentov na acyklickom modeli.*



Obr. 7.7: *Výsledky modelu s náhodným prijímaním aktivácie. Parameter ϵ bol zvolený vzhľadom k výsledkom v podkapitole 7.5 na $\epsilon = 50$.*

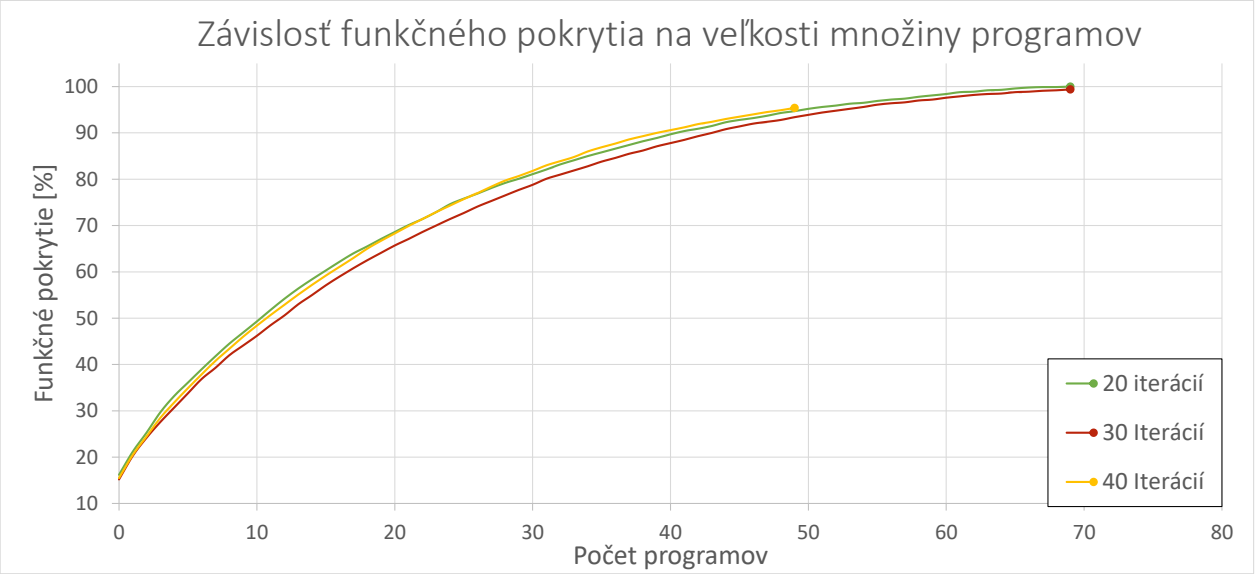


Obr. 7.8: *Výsledné porovnanie.*

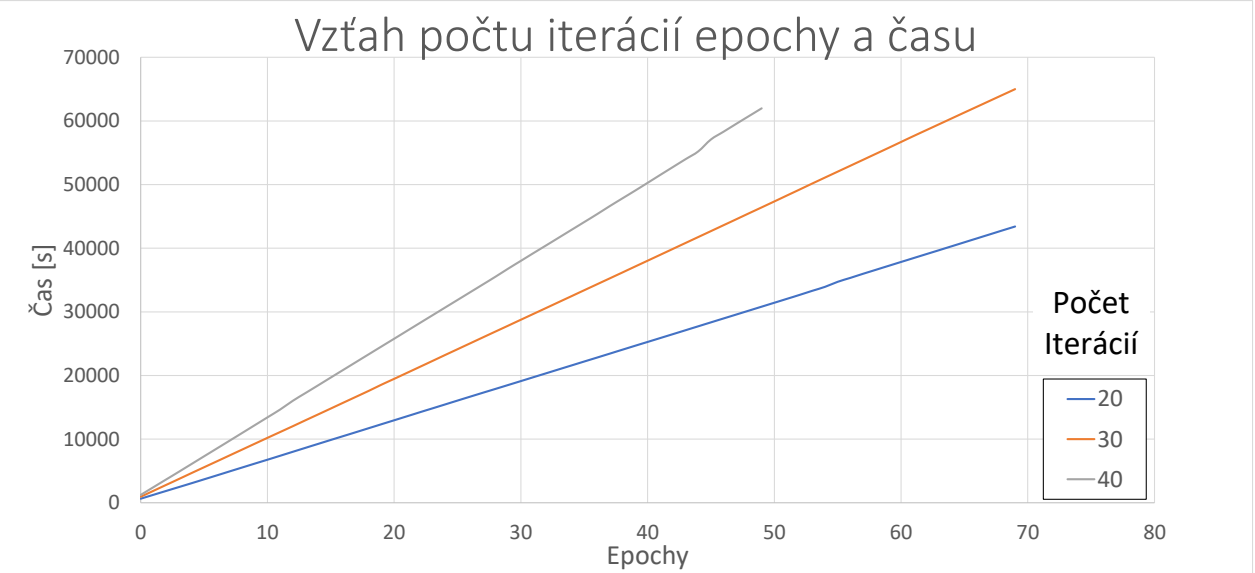
Obrázok 7.8 porovnáva predstavené modely. Bipolar 0-1 značí bipolárny model s počiatočným rovnomerným rozložením stavov neurónov v intervale $< 0, 1 >$ (analogicky platí pre Bipolar 0.4-0.6 a Bipolar Triangular). AllOne značí jednotkový model, Probabilistic značí model s náhodným prijímaním aktivácie a Acyclic značí acyklický grafový model. BestProgTime značí bod v čase, za ktorý bol na modele dosiahnutý uzáver pokrytia optimálnou sadou programov s epochou trvajúcou 20 iterácií (viď ďalšia podkapitola).

7.1.3 Hľadanie optimálnej sady programov

V tejto podkapitole sa budeme zaoberať hľadaním optimálnej sady programov pre verifikáciu modelu podľa algoritmu 4. Na optimalizáciu je použitý bipolárny model s počiatočným stavom neurónov daným rovnomerným rozložením pravdepodobnosti v intervale $< 0.4, 0.6 >$ (Obr. 7.9). Závislosť dĺžky epochy na čase optimalizácie je následne znázornená na Obr. 7.10.



Obr. 7.9: *Pokrytie dosiahnuté rôznym počtom programov nájdených neurónovou sieťou pri rôznych dĺžkach epochy (20, 30, 40).*

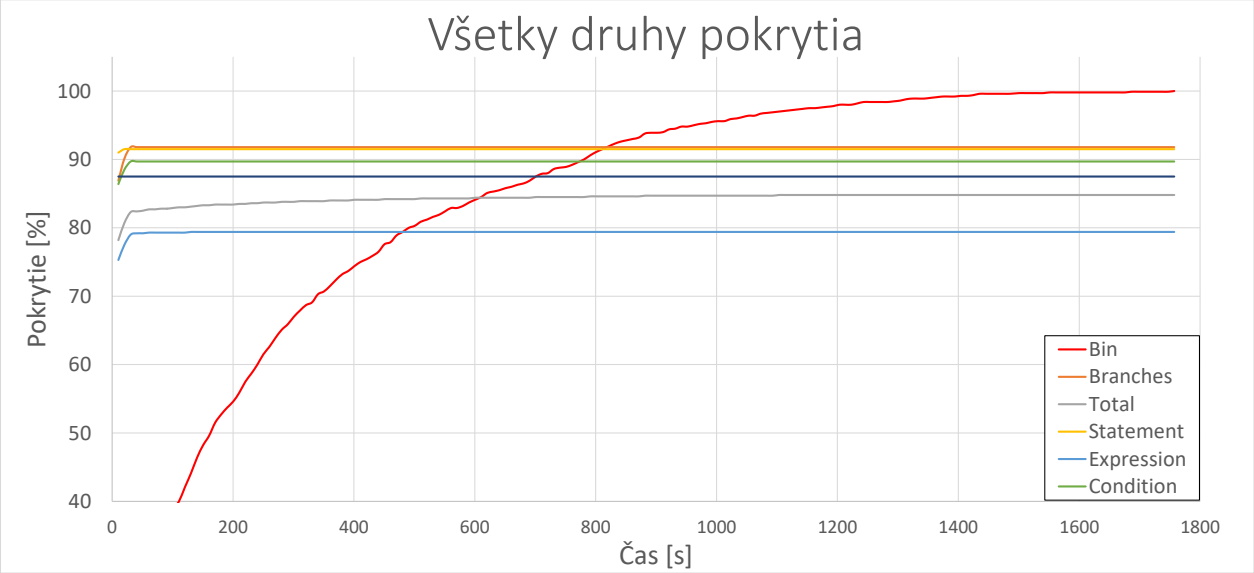


Obr. 7.10: *Závislosť rôznych dĺžok epochy na optimalizačnom čase je zrejme lineárna.*

7.1.4 Ďalšie experimenty

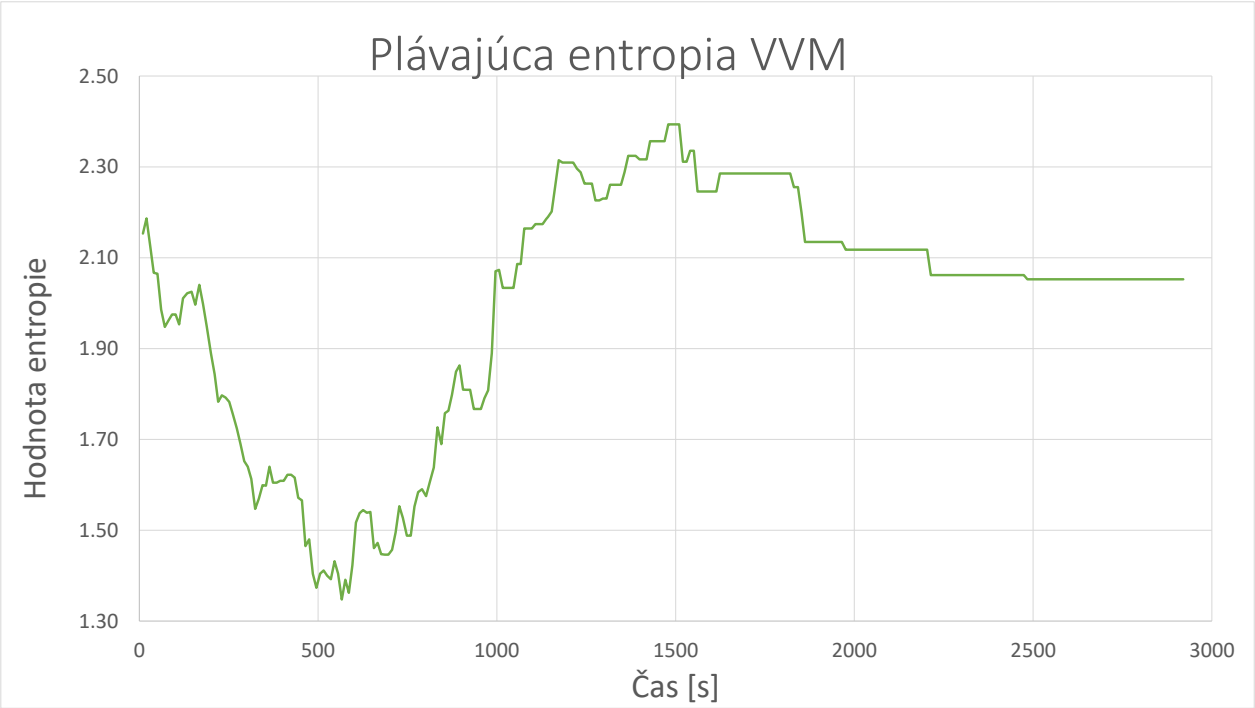
Tieto experimenty ukazujú ďalšie typy pokrytia a ich dosiahnuté hodnoty na ASIP uRISC (Obr. 7.11), vlastnosti váhovo vyváženého modelu (VVM) (Obr. 7.12) a charakteristiku siete v čase podla metriky nazvanej **plávajúca entropia** (Obr. 7.13). Táto metrika znázorňuje, ako veľmi sa hodnoty jednotlivých neurónov líšia od ich priemernej hodnoty. Nech N je množina neurónov siete, potom plávajúcu entropiu v čase t definujeme vzťahom 7.1.

$$FloatingEntropy(t) = \sum_{i=1}^{N} \left| v_i^{(t)} - \frac{\sum_{i=1}^{N} v_i^{(t)}}{|N|} \right| \tag{7.1}$$



Obr. 7.11: *Rôzne typy pokrytia zohľadnené neurónovou sieťou (legenda podľa 5.3). Dáta pochádzajú z experimentálneho behu 7.3.*

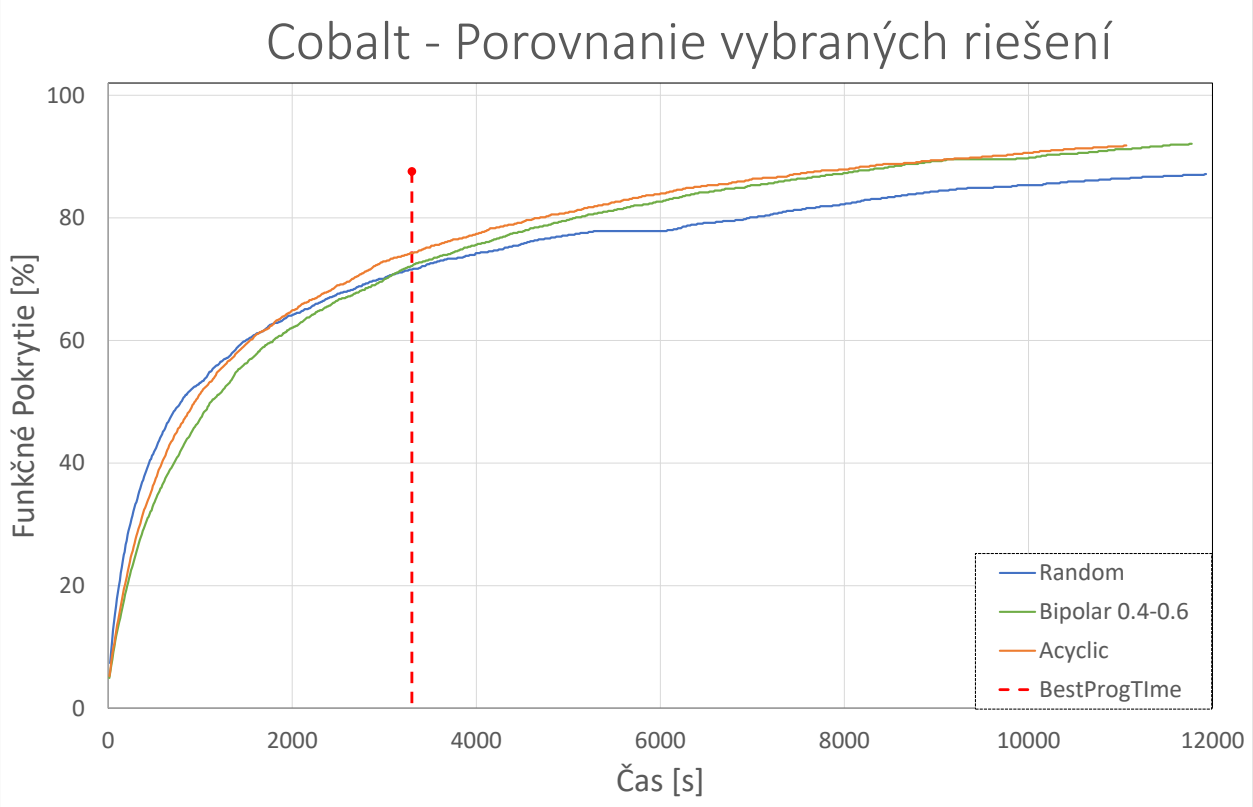
Obr. 7.12: *Napriek rôznym hodnotám neurónov u VVM ich priemerný stav osciluje okolo hodnoty 0.5. Dáta pochádzajú z experimentálneho behu 7.6.*



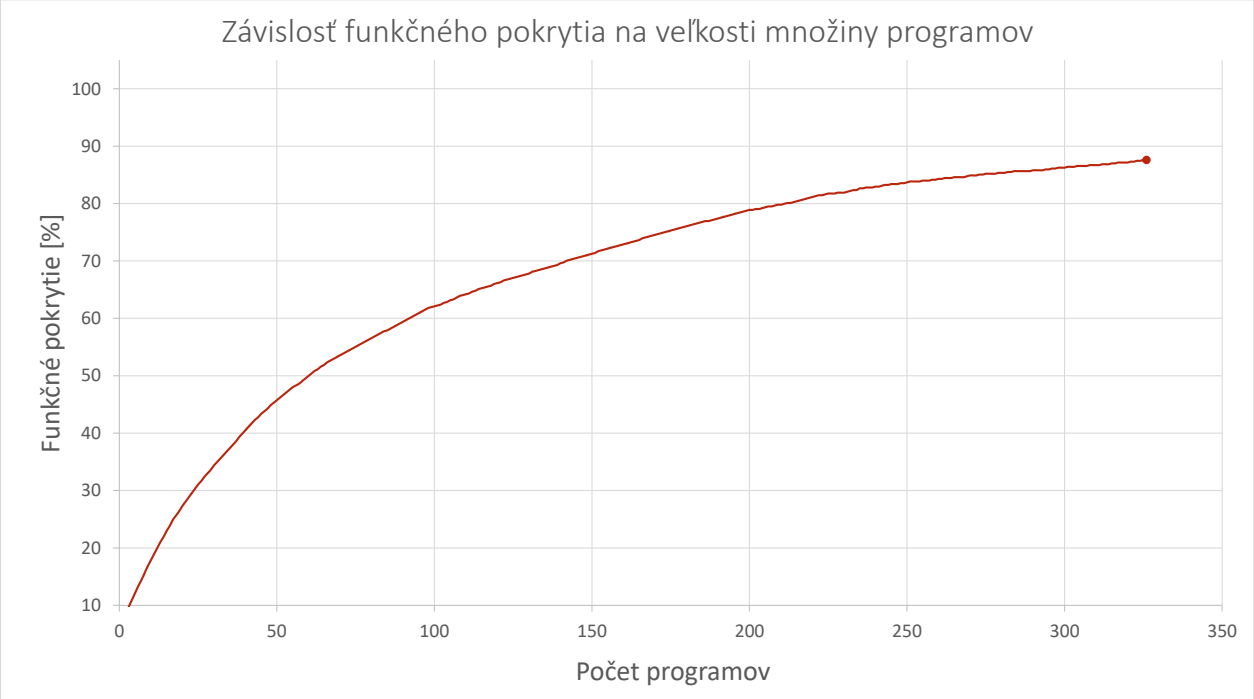
Obr. 7.13: *Hodnoty plávajúcej entropie z experimentálneho behu 7.6.*

7.2 Codix Cobalt

Niektoré úspešnejšie z experimentov uvedených v kapitole 7.1 boli zopakované aj na komplexnejšom ASIP s názvom Codix Cobalt (viď 6.3). Experimenty boli podobne ako pri modeli *uRISC* realizované výberom **mediánu** z piatich realizovaných optimalizačných behov. Parameter sigmoidy λ sme nastavili rovnako na hodnotu $\lambda = 0.9$. Výsledky týchto experimentov sú predmetom obrázkov 7.14 a 7.15.



Obr. 7.14: *Porovnanie vybraných modelov priebežnej optimalizácie oproti náhodnému riešeniu Random. Podobné výsledky ako priebežná optimalizácia dosiahla nájdená optimálna množina programov už za 55 minút (3300s, označené ako BestProgTime) pomocou 327 programov. Význam ostatných prvkov legendy je analogický k 7.8.*



Obr. 7.15: *Priebeh hľadania optimálnej sady programov. Jedna optimalizačná epocha má 20 iterácií prehľadávania.*

Kapitola 8

Záver

Táto diplomová práca sa zaoberá problematikou automatizácie verifikácie integrovaných obvodov pomocou neurónových sietí.

Za zverifikovaný IC sme v tejto práci považovali taký obvod, ktorého funkčné pokrytie dosiahlo 100%. Využili sme teda techniku verifikácie riadenej pokrytím (*Coverage-Driven Verification*). S jej využitým boli vytvorené techniky používajúce kombinačnú optimalizačnú heuristiku schopné riešiť dva optimalizačné problémy popísané v podkapitole 1.2. Funkčnosť týchto techník bola experimentálne overená na aplikačne špecifických procesoroch (ASIP) Codasip uRISC a Codix Cobalt.

Prvá technika (Alg. 3) zaoberajúca sa priebežnou optimalizáciou konfigurácie PNG umožnila efektívnejšie prehľadávanie priestoru pokrytia a skrátila tak čas potrebný pre dosiahnutie uzáveru pokrytia funkčnou verifikáciou komplexných integrovaných obvodov. V prípade prebiehajúceho designu IC tak nie je nutné neustále udržiavať sadu verifikačných stimulov, ktorá sa stará o jeho bezchybnosť.

Druhá navrhnutá technika (Alg. 4) umožňuje hľadanie takej minimálnej množiny programov, ktorá dosiahne uzáver funkčného pokrytia na IC. Pomocou tejto techniky je v praxi možné vytvoriť takú sadu stimulov, ktorej spustením v RTL simulátore dosiahneme uzáver pokrytia za minimálny čas. Tento prístup je vhodný pre pravidelnú verifikáciu stabilných IC, ktorých komunikačné rozhranie (v našom prípade inštrukčná sada) sa príliš nemení.

Navrhnuté neurónové siete vychádzajú z modelu Hopfieldovej siete. V priebehu riešenia tejto práce došlo k jej rôznym modifikáciám, z ktorých sú uvedené len tie experimentálne najúspešnejšie (podkapitola 5.2). Riešenie je možné realizovať nad každým verifikačným prostredím, ktoré využíva techniku pseudonáhodného generovania stimulov (*Constrained-Random Stimulus Generation*), ak je na pseudonáhodný generátor využitý pri tejto technike možné aplikovať váhový model (viď podkapitola 4.4).

Existujú však aj ďalšie modifikácie neurónových sietí, ktoré je možné využiť pre riešenie optimalizačných problémov. V budúcej práci by tak mohli byť využité koncepty ako *Kohonenova mapa* (anglicky nazývaná aj *Self-organizing Map*) s fitness funkciou či *Elastické siete* [23]. Pokračovaním tejto práce by tiež mohla byť optimalizácia jej implementácie, predovšetkým uplatnenie paralelizmu pri spúšťaní verifikačných behov a akcelerácia implementácie pomocou HW (viď kapitola *FPGA-based Acceleration of Functional Verification* v [16]).

Literatúra

[1] Accellera and Cadence Design Systems and Mentor Graphics and Synopsys: *Universal Verification Methodology (UVM) 1.1 User's Guide*. 2011.
URL http://accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.1.pdf

[2] Arbib, M. A.: *The Handbook of Brain Theory and Neural Networks*. MIT Press, Cambridge, MA, 1995.

[3] Broomhead, D. S.; D.Lowe: *Multivariable Functional Interpolation and Adaptive Networks. Complex Systems*, ročník 2, č. 2, 1988: s. 321–355.
URL
http://sci2s.ugr.es/keel/pdf/algorithm/articulo/1988-Broomhead-CS.pdf

[4] Burkhardt, N.: *Introduction to Functional Verification*. Universität Heidelberg, [Online; navštíveno 30.12.2016].
URL https://www.rz.uni-frankfurt.de/39888300/21_7_2011_Burkhardt_Verification.pdf

[5] Codasip: *Codasip Codix Processors*. 2017, [Online; navštíveno 14.5.2017].
URL https://www.codasip.com/codix-cores/

[6] Fajčík, M.: *Statická analýza zdrojového kódu jazyka CodAL*. Bakalářská práce, Ústav informačních systémů FIT VUT, 2015.

[7] Foster, H.: *Trends in functional verification: A 2014 industry study*. 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), 2015.

[8] Foster, H.: *The 2016 Wilson Research Group Functional Verification Study*. online, 2016.
URL https://blogs.mentor.com/verificationhorizons/blog/2016/10/04/part-8-the-2016-wilson-research-group-functional-verification-study/

[9] Foundation, T. L.: *The LLVM Compiler Infrastructure*. 2017, [Online; navštíveno 14.5.2017].
URL http://llvm.org/docs/

[10] Freericks, M.: *The nML Machine Description Formalism*. 1993.
URL http://www6.in.tum.de/Main/Publications/Freericks1991a.pdf

[11] Group, I.-X. W.: IP-XACT. 2016, [Online; navštíveno 6.1.2017].
URL http://accellera.org/downloads/standards/ip-xact

[12] Group, S. L. W.: *IEEE Standard for System Verilog- Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2005*, 2005: s. 1–648, doi:10.1109/IEEESTD.2005.97972.

[13] Hopfield, J.: *Neural networks and physical systems with emergent collective computational abilities. Biophysics*, ročník 79, č. 8, April 1982: s. 2554–2558,.
URL http://redwood.berkeley.edu/vs265/hopfield82.pdf

[14] Hopfield, J.; Tank, D.: *"Neural" computation of decisions in optimization problems. Biological Cybernetics*, ročník 52, č. 3, 1985: str. 141–152, [Online; navštíveno 29.12.2016].
URL http://link.springer.com/article/10.1007/BF00339943

[15] Hornik, K.; Stinchcombe, M.; White, H.: *Multilayer Feedforward Networks are Universal Approximators. Neural Networks*, ročník 2, č. 5, 1989: str. 359–366, [Online; navštíveno 29.12.2016].
URL http://www.sciencedirect.com/science/article/pii/0893608089900208

[16] Šimková, M.: *New Methods for Increasing Efficiency and Speed of Functional Verification.* Dizertační práce, Department of Computer Systems FIT BUT, 2015.

[17] Jain, A.; Bonanno, G.; Gupta, H.; aj.: *Generic System Verilog Universal Verification Methodology based Reusable Verification Environment for Efficient Verification of Image Signal Processing IPs/SoCs. International Journal of VLSI design & Communication Systems*, ročník 3, č. 6, 2012, [Online; navštíveno 6.1.2017].
URL http://www.aircconline.com/vlsics/V3N6/3612vlsics02.pdf

[18] Kvasnička, V.; Čerňanský, M.: *Neurónové siete.* [Online; navštíveno 30.12.2016].
URL http://www2.fiit.stuba.sk/~kvasnicka/NeuralNetworks/

[19] Lagoudakis, M. G.: *Neural Networks and Optimization Problems- A Case Study: The Minimum Cost Spare Allocation Problem.* 1997.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.3419

[20] Lillo, W.; Loh, M.; S.Hui; aj.: *On Solving Constrained Optimization Problems with Neural Networks: A Penalty Method Approach. IEEE Transactions on Neural Networks*, ročník 4, č. 6, 1993: str. 931–940.

[21] McCulloch, W. S.; Pitts, W.: *A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics*, ročník 5, 1943: s. 115–133.

[22] Ohlsson, M.; Peterson, C.; Soderberg, B.: *Neural Networks for optimization problems with inequality constraints: the knapsack problem. Neural Computation*, ročník 5, č. 2, 1993: str. 331–339.

[23] Potvin, J.-Y.; Smith, K. A.: *Handbook of Metaheuristics*, kapitola Artificial Neural Networks for Combinatorial Optimization. Boston, MA: Springer US, 2003, ISBN 978-0-306-48056-0, s. 429–455, doi:10.1007/0-306-48056-5_15.
URL http://dx.doi.org/10.1007/0-306-48056-5_15

[24] Rosenblatt, F.: *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. Psychological Review*, ročník 65, 1958: s. 386–408.

[25] Rosenblatt, F.: *Principles of Neurodynamics: Perceptrons and the Theory of Brain Machines*. Washington: Spartan Books, 1962.

[26] Rumelhart, D. E.; Hinton, G. E.; Williams, R. J.: *Learning Internal Representations by Error Propagation*. In *Neurocomputing: Foundations of Research*, editace J. A. Anderson; E. Rosenfeld, Cambridge, MA, USA: MIT Press, 1988, ISBN 0-262-01097-6, s. 673–695.
URL http://dl.acm.org/citation.cfm?id=65669.104449

[27] Rumelhart, D. E.; Hinton, G. E.; Williams, R. J.: *Learning Representations by Back-propagating Errors*. In *Neurocomputing: Foundations of Research*, editace J. A. Anderson; E. Rosenfeld, Cambridge, MA, USA: MIT Press, 1988, ISBN 0-262-01097-6, s. 696–699.
URL http://dl.acm.org/citation.cfm?id=65669.104451

[28] Siegelmann, H. T.; Sontag, E. D.: *Analog computation via neural networks*. *Theoretical Computer Science*, ročník 131, č. 2, 1994: s. 331–360, [Online; navštíveno 29.12.2016].
URL http://www.sciencedirect.com/science/article/pii/0304397594901783

[29] Stufflebeam, R.: *Neurons, Synapses, Action Potentials, and Neurotransmission*. online, 2008.
URL http://www.mind.ilstu.edu/curriculum/neurons_intro/neurons_intro.php

[30] Teller, N. M. . A. R. . M. R. T.: *Equations of state calculations by fast computing machines*. *J. Chem. Phys.*, ročník 21, 1953: s. 1087–1092.

[31] Vecchi, S. K. C. D. G. M. P.: *Optimization by Simulated Annealing*. *Science*, ročník 220, 1983: s. 671–680.

[32] Werbos, P. J.: *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. New York, NY, USA: Wiley-Interscience, 1994, ISBN 0-471-59897-6.

[33] WILE, B.; GOSS, J.; ROESNER, W.: *Comprehensive Functional Verification: The Complete Industry Cycle*. Elsevier Science, 2005, ISBN ISBN 978-0-0-8047664-3.
URL https://books.google.sk/books?id=btl_OX3kJ7MC

Prílohy

Príloha A

Kód riadiacich algoritmov

```
# inicializacia PNG
with RngManager(RNG_PATH, SRCFOLDER, OBJFOLDER, EXEFOLDER, CONFIG_GEN,
    CONFIG_USER) as rman:
  # nacitanie konfiguracie
  config = NodeConfig(PDM_PATH, rman)
  # Tvorba neuronovej siete
  hopfieldnetwork = Hopfield(config)
  # Generovanie programov
  rman.runGen(settings.INSTRUCTION_COUNT, settings.PROGRAM_COUNT)
  # Beh verifikacneho prostredia s konfiguraciou vygenerovanou CS
  runsimulationsync()
  # Ziskavanie pokrytia
  coverage = getMergedCoverage();
  # Vypocet pociatocnej energie
  oldenergy = hopfieldnetwork.getEnergy(coverage)
  # Inicializacia tabu zoznamu
  tabulist = []
  try:
      # VNUTORNY CYKLUS
  finally:
      # cistenie, zapis finalnych informacii...
```

Kód A.1: Priebežná optimalizácia konfigurácie PNG.

```
#V evolucnom case od 0 do settings.MAXITERATIONS
for iteration in range(settings.MAXITERATIONS):
    # Ak je tabu zoznam plny, siet uviazla v minime
  if (len(tabulist) == len(config.nodes)):
    break
  # Aktivacia nahodneho neuronu
  activated_neuron = hopfieldnetwork.activateRandom(tabulist)
  # Aplikacia aktualnej mnoziny vazenych uzlov na PNG
  rman.applyConfigToRnG(config)
  # Generovanie programov
  rman.runGen(settings.INSTRUCTION_COUNT, settings.PROGRAM_COUNT)
  # Beh verifikacneho prostredia s konfiguraciou danou aktualnou mnozinou
  # vazenych uzlov
  runsimulationsync()
  # Zjednotenie DB a ziskavanie pokrytia
  coverage = getMergedCoverage();
  # Vypocet energie z pokrytia
  newenergy = hopfieldnetwork.getEnergy(coverage)
  # V pripade ze doslo k poklesu energie, aktivacia je prijata
  # ulozi sa energia a zmaze tabu zoznam
  # V pravdepodobnostnom modeli nasledujucu podmienku modifikujeme
     nasledovne:
  # if (newenergy < oldenergy or \
  #   random.uniform(0,1)<hopfieldnetwork.getAcceptProb(iteration)):
  if (newenergy < oldenergy):
    hopfieldnetwork.accept(newenergy, coverage)
    oldenergy = newenergy
    tabulist = []
  # Inak sa siet vrati do povodneho stavu
  # a posledny aktivovany neuron sa prida
  # do tabu zoznamu
  else:
    tabulist.append(activated_neuron)
    hopfieldnetwork.revert()
    # Aplikacia predoslej mnoziny vazenych uzlov na PNG
    rman.applyConfigToRnG(config)
```

Kód A.2: Vnútorný cyklus priebežnej optimalizácie konfigurácie PNG.

```
# Inicializacia PNG, nacitanie konfiguracie, tvorba NN, tvorba priecinkov
# Inicializacia premennych
bestsofar = sys.float_info.max
bestcovsofar = None
bestsofarname = ""
bestneuronsofar = None
ismergedcov = False

# Vygenerovanie programov s pociatocnou konfiguraciou
for _ in range(BESTPROGRUNS):
    genprogs = rman.runGen(INSTRUCTION_COUNT, 1)
    runsimulationsync()
    coverage = getMergedCoverage();
    e = hopfieldnetwork.getEnergy(coverage)
    # Hlada sa najlepsi program, jeho databaza a zdrojovy kod sa ulozia
    if e < bestsofar:
        if bestcovsofar is not None:
            remove(os.path.join(BESTPROGFOLDER,bestsofarname))
            remove(os.path.join(BESTPROGFOLDER,bestsofarname+"_"+
                MERGEDCOV_UCDB))
        copyfile(os.path.join(SRCFOLDER,genprogs[0]), os.path.join(
            BESTPROGFOLDER,genprogs[0]))
        copyfile(os.path.join(UVM_PATH,MERGEDCOV_UCDB), os.path.join(
            BESTPROGFOLDER,genprogs[0]+"_"+MERGEDCOV_UCDB))
        bestsofar = e
        bestsofarname = genprogs[0]
        bestcovsofar = coverage

tabulist = []
epoch = 0
#Epocha nemoze byt dlhsia nez pocet neuronov
EPOCHLENGTH = len(config.nodes) if len(config.nodes)<MAXEPOCHLENGTH \
                                else MAXEPOCHLENGTH
try:
    # VNUTORNY CYKLUS
finally:
    # cistenie, zapis finalnych informacii...
```

Kód A.3: Hľadanie optimálnej množiny programov

```
while bestcovsofar[BINCOVERAGE]<100.0:
    # Dojde k EPOCHLENGTH aktivaciam roznych neuronov
    for iteration in range(EPOCHLENGTH):
        activated_neuron = hopfieldnetwork.activateRandom(tabulist)
        # Aktivovany neuron je ihned vlozeny do tabu zoznamu
        tabulist.append(activated_neuron)
        rman.applyConfigToRnG(config)
        # Pre kazdy aktivovany neuron vygenerujeme BESTPROGRUNS programov
        for _ in range(BESTPROGRUNS):
            # Zaloha databazy obsahujucej pokrytie z predoslej epochy
            if (os.path.isfile(os.path.join(UVM_PATH,MERGEDCOV_UCDB))):
                copyfile(os.path.join(UVM_PATH,MERGEDCOV_UCDB), os.path.join(
                    TMPFOLDER,MERGEDCOV_UCDB))
                ismergedcov=True
            genprogs = rman.runGen(INSTRUCTION_COUNT, 1)
            runsimulationsync()
            # Zjednotenie s databazou a ziskanie pokrytia
            coverage = getMergedCoverage()
            e = hopfieldnetwork.getEnergy(coverage)
            # Hladanie najlepsieho programu v epoche
            if e < bestsofar:
                if bestsofarname!="":
                    remove(os.path.join(BESTPROGFOLDER,bestsofarname))
                    remove(os.path.join(BESTPROGFOLDER,bestsofarname+"_"+
                        MERGEDCOV_UCDB))
                copyfile(os.path.join(SRCFOLDER,genprogs[0]), os.path.join(
                    BESTPROGFOLDER,genprogs[0]))
                copyfile(os.path.join(UVM_PATH,MERGEDCOV_UCDB), os.path.join(
                    BESTPROGFOLDER,genprogs[0]+"_"+MERGEDCOV_UCDB))
                bestsofar = e
                bestsofarname = genprogs[0]
                bestcovsofar = coverage
                bestneuronsofar = activated_neuron
            # Obnova databazy z predoslej epochy, ak existovala
            if not ismergedcov:
                remove(os.path.join(UVM_PATH,MERGEDCOV_UCDB))
            else:
                shutil.move(os.path.join(TMPFOLDER,MERGEDCOV_UCDB),os.path.
                    join(UVM_PATH,MERGEDCOV_UCDB))
        # Obnova siete do predosleho stavu
        hopfieldnetwork.revert()
        rman.applyConfigToRnG(config)
    # Aktivacia najlepsieho neuronu po preskumani stavoveho priestoru epochy
    if not bestneuronsofar is None:
        hopfieldnetwork.activateNeuron(bestneuronsofar)
        hopfieldnetwork.accept(bestsofar, bestcovsofar)
    epoch+=1
    # Nastavenie databazy pre dalsiu epochu
    if (os.path.isfile(os.path.join(UVM_PATH,MERGEDCOV_UCDB))):
        remove(os.path.join(UVM_PATH,MERGEDCOV_UCDB))
    copyfile(os.path.join(BESTPROGFOLDER,bestsofarname+"_"+MERGEDCOV_UCDB),os
        .path.join(UVM_PATH,MERGEDCOV_UCDB))

    # Reinicializacia premennych
    tabulist = []
    bestsofar = sys.float_info.max
    bestsofarname = ""
    bestneuronsofar = None
```

Kód A.4: Vnútorný cyklus pre hľadanie optimálnej množiny programov