VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ RESYNTÉZA KOMBINAČNÍCH OBVODŮ

EVOLUTIONARY RESYNTHESIS OF COMBINATIONAL CIRCUITS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. JITKA KOCNOVÁ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZDENĚK VAŠÍČEK, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/18259/2016/xkocno00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Kocnová Jitka, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Evoluční resyntéza kombinačních obvodů**
Evolutionary Resynthesis of Combinational Circuits
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s problematikou evolučních algoritmů, logickou syntézou a optimalizací číslicových obvodů.
2. Zpracujte studii na výše uvedené téma.
3. Vytvořte algoritmus pro evoluční resyntézu kombinačních obvodů. Algoritmus bude identifikovat vhodné podobvody, optimalizovat je pomocí existujícího optimalizátoru využívajícího SAT solver a vracet je zpět do původního obvodu.
4. Navržený algoritmus implementujte s ohledem na maximální výkonnost.
5. Pomocí sady benchmarkových obvodů experimentálně vyhodnoťte účinnost navrženého řešení.

Literatura:

- Dle pokynů vedoucího.

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: **Vašíček Zdeněk, Ing., Ph.D.**, 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

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

Abstrakt

Diplomová práce se zabývá resyntézou kombinačních obvodů pomocí evolučních principů. První část se zabývá logickou syntézou a jejími problémy, evoluční syntézou a výhodami evolučního přístupu, a taktéž jsou zmíněny některé existující syntézní nástroje. V druhé části jsou pak přiblíženy vybrané grafové algoritmy a jejich možné využití v navrhovaném programovém rozšíření pro vybraný syntézní nástroj. Zde je také popsán návrh rozšíření a jeho samotná implementace. Třetí část se zabývá testováním vzniklého rozšíření. Čtvrtou částí je závěr shrnující získané poznatky a dosažené výsledky.

Abstract

This master thesis is concerned about the resynthesis of combinational circuits with the help of evolutional principles. The first part of this text describes logic synthesis and its weak spots, evolutional synthesis and its advantages, and also some of the existing synthesis programs. The second part shows usage of graph algorithms in logic synthesis and their possible usage in an extension for the chosen synthesis program. Suggested design and practical implementation of the extension is also described in this part. In the third part extension testing is mentioned. The fourth part is the last one and concludes gained knowledge and results.

Klíčová slova

kombinační obvod, syntéza, resyntéza, evoluční syntéza, evoluce, řez, graf, grafový algoritmus

Keywords

combinational circuit, synthesis, resynthesis, evolutional synthesis, evolution, cut, graph, graph algorithm

Citace

KOCNOVÁ, Jitka. *Evoluční resyntéza kombinačních obvodů*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Vašíček Zdeněk.

Evoluční resyntéza kombinačních obvodů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením pana Ing. Zdeňka Vašíčka, Ph.D. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Jitka Kocnová
31. května 2017

Poděkování

Děkuji vedoucímu své práce, Ing. Zdeňku Vašíčkovi, Ph.D., za poskytnuté informace, rady, ochotu při konzultacích, a vstřícný přístup.

Obsah

Kapitola 1

Úvod

Tvorba kombinačních obvodů je velmi komplexním a náročným procesem. Nedílnou součástí vzniku obvodu je jeho syntéza, při které z popisu chování obvodu vzniká jeho vlastní reprezentace v podobě logických prvků (hradel) a jejich propojení.

Tato práce si klade za cíl seznámit čtenáře s evoluční syntézou kombinačních obvodů a v praktické části poté ukázat využití grafových algoritmů a řezů pro výběr podobvodů z rozsáhlých kombinačních obvodů v rámci rozšíření již existujícího syntézního nástroje.

Zmiňované rozšíření je vyvíjeno za účelem lepšího využití evolučních principů v průběhu logické syntézy rozsáhlých kombinačních obvodů. V současné době je totiž možné efektivně aplikovat evoluční algoritmy pouze na obvody o nízkém počtu primárních vstupů a taktéž o nízkém počtu hradel. Obvody, ve kterých se nachází veliké množství logických hradel (například v řádu tisíců či desetitisíců), je náročné evolučně zpracovat – čím větší obvod je, tím více rostou nároky na čas, paměť a energii nutnou k provedení výpočtu.

Podobvody vybrané tímto rozšířením budou sloužit jako vstupy pro speciální program, který nad nimi provede řadu evolučních kroků a pozměněné je vrátí zpět do původního zpracovávaného obvodu, jenž poté bude vybraným rozšiřovaným nástrojem syntetizován jako celek.

Diskutovány budou různé přístupy k syntéze obvodů a rozdíly mezi nimi. Práce se zaměřuje především na srovnání klasické syntézy a syntézy využívající evoluční algoritmy. V textu práce se taktéž nachází stručný přehled již existujících syntézních nástrojů.

Součástí práce je také vyhodnocení evoluční syntézy s využitím řezů, především pak v podobě experimentů. Pro tento účel bude sloužit zejména výše zmiňované programové rozšíření, kterému se tato práce věnuje.

V závěru textu budou shrnuty dosažené experimentální výsledky a diskutována další možná vylepšení vzniklého rozšíření.

Kapitola 2

Syntéza kombinačních obvodů

Digitální obvody mohou být prezentovány na různých úrovních abstrakce. Na začátku vývoje koncového zařízení bývá obvykle obvod popsán pomocí vhodného programovacího jazyka (např. VHDL či Verilogu). Pro úspěšné sestrojení požadovaného obvodu je ale třeba převést jeho popis z vyšší abstrakční úrovně na reprezentaci s nižší abstrakcí. Stále je přitom potřeba zachovat zamýšlenou funkci obvodu. Tato transformace na jinou formu popisu se nazývá syntéza[20].

Tato kapitola seznamuje čtenáře se základními prvky a principy logické syntézy. Popisuje také využití evolučních metod během syntézy a jejich možný vliv na výsledný produkt syntézního procesu. Uveden je též výčet nejznámějších syntézních nástrojů.

2.1 Logická syntéza

Logická syntéza je proces, který z popisu chování obvodu na úrovni RTL (register-transfer-level) pomocí syntézního nástroje vytváří reprezentaci obvodu využívající logická hradla implementující požadovanou činnost. Úkolem syntézy je najít vhodnou implementaci logické funkce při splnění určitých omezujících podmínek.

Matematickým základem logické syntézy je tzv. Booleova algebra[1]. Claude E. Shannon ve své práci[2] dokázal, že je možné formalizovat návrh a analýzu klopných obvodů pomocí Booleovy algebry, a také že tyto obvody mohou být využity pro řešení problémů Booleovy algebry. Každý kombinační obvod reprezentuje určitou Booleovskou funkci[5].

Booleovské funkce lze reprezentovat pomocí mnoha vhodných datových struktur, jakými jsou např. pravdivostní tabulky, and-inverter grafy[3], binární rozhodovací diagramy[4], Booleovské sítě produkt sum[5] či suma produktů[6][5]. Booleovská síť je orientovaným acyklickým grafem[7], ve kterém uzly odpovídají logickým hradlům a hrany propojům mezi nimi.

Logickou syntézu je možné rozdělit na dvě oblasti:

- syntéza na úrovni LUT[8] (pro FPGA),
- syntéza na úrovni hradel (pro ASIC obvody).

[1] George Boole, 1847
[2] A Symbolic Analysis of Relay and Switching Circuits, 1937
[3] and-inverter graphs, AIG
[4] binary decision diagrams, BDD
[5] product of sums, POS, konjunktivní normální forma
[6] angl. sum of products, SOP, disjunktivní normální forma
[7] DAG, direct acyclic graph
[8] Look-up Table

2.1.1 Postup logické syntézy

Logická syntéza byla plně automatizována a je tedy možné za jejím účelem využít služby některého z mnoha komerčních či volně dostupných syntézních nástrojů.

Průběh syntézy se dělí na dvě základní části:

- technologicky nezávislá část (zpracování vstupu, aplikace omezení, optimalizace),
- technologicky závislá část (mapování vysyntetizovaného popisu obvodu na cílovou technologii).

Vstupem pro syntézní nástroj bývá obvykle soubor, který obsahuje popis syntetizovaného obvodu ve vhodném jazyce – např. VHDL či Verilog.

Během syntézy lze nastavovat různá omezení – tzv. constraints. Tyto omezující podmínky ovlivňují výsledek syntézního procesu. Každý syntézní nástroj má svůj vlastní seznam omezení, které je možné použít.

Nejčastějšími omezujícími podmínkami bývají například:

- omezení syntézy (speciální příkazy pro mapování HDL popisu na cílovou reprezentaci)
- omezení na plochu a umístění (specifikace umístění některých prvků obvodu),
- omezení na časování (specifikace šíření hodinového signálu obvodem či jeho částmi),
- omezení na vstupy a výstupy (specifickým vstupním či výstupním pinům/bankům jsou přiřazeny požadované signály).

Posledním krokem logické syntézy je tzv. technologické mapování[9], při kterém dochází k transformaci logické sítě, nazývané také jako subjektový graf[10], nezávislé na cílové technologii, na síť složenou z logických hradel. Subjektový graf je často reprezentován and-inverter grafem.

Úkolem mapování je výběr vhodných hradel z technologických knihoven s předdefinovanými logickými prvky k sestavení výsledného obvodu vzhledem k jeho vlastnostem – např. volba rychlých prvků pro sestavení kritických částí. Mapování by nemělo být závislé či omezené na použití určitých technologických knihoven, protože se tím tak snižuje možnost optimalizace obvodu a také širšího využití mapovacího algoritmu, případně celkového syntézního nástroje[5].

Systematický přístup k mapování je založen na nalezení minimální ceny pokrytí grafu. Pokrytí je množina grafů taková, že každý uzel vytvářeného grafu (obvodu) je obsažen v jednom či více vzorových grafech.

Výsledkem syntézního procesu je tzv. popis obvodu na úrovni logických hradel. Některé syntetizátory produkují tzv. bitstreamy pro programovatelná hradlová pole, jakými jsou například FPGA či PAL. Existují také syntézní programy schopné vytvářet popisy pro aplikačně-specifická zařízení (tzv. ASIC).

Kromě klasické logické syntézy se lze setkat také s pojmem high-level syntéza. Tato syntéza slouží k převodu popisu obvodu na vyšší úrovni abstrakce (např. pomocí jazyků C či C++) na úroveň RTL. Příkladem nástroje, který je vhodný pro tento druh syntézy, je Catapult C Synthesis.

V této práci není high-level syntéza dále využívána, tento odstavec proto slouží jen jako stručné doplnění části o syntéze kombinačních obvodů.

[9] angl. technology mapping
[10] angl. subject graph

2.2 Optimalizace

Optimalizace je jednou z nejdůležitějších vlastností klasické logické syntézy. Jejím cílem je zredukovat Booleovskou funkci, která je vyvíjeným obvodem implementována, a také přizpůsobit daný obvod cílové technologii. Díky ní lze zmenšit nejen plochu potřebnou pro umístění obvodu, ale také snížit jeho cenu či spotřebu a zvýšit jeho rychlost.

Součástí optimalizace je minimalizace. Jedná se o proces, kdy se ze zpracovávaného obvodu vhodně zvolenou matematickou metodou odstraní části, které nemají vliv na funkčnost obvodu jako celku. Dojde tak ke zjednodušení Booleovské funkce, ktera obvod reprezentuje. Z tohoto faktu vyplývá úspora logických členů potřebných k realizaci obvodu a tím také možná úspora plochy a ceny[4].

K nejznámějším konvenčním minimalizačním metodám patří:

- Karnaughova mapa (ve formě disjunktivní i konjunktivní),
- Quinn-McCluskey,
- Espresso.

Nevýhodou Karnaughovy mapy je její obtížné použití pro funkce obsahující více jak čtyři proměnné. Mapa se totiž stává velmi rozsáhlou a je složité vybírat v ní pole určená k minimalizaci. Taktéž se hůře rozhoduje o tom, zda je daný výběr polí nejvýhodnější ze všech možností. Vyskytuje-li se ve funkci proměnných méně, pak je tato metoda velmi efektivní a snadno interpretovatelná[4].

Metoda Quinn-McCluskey překonává nedostatky Karnaughovy mapy v oblasti limitního počtu proměnných. Smyslem této metody je poskytování algoritmického postupu pro získání tzv. přímých implikantů. Přímé implikanty jsou všechny termy (proměnné), které jsou kandidátními pro zařazení do finální zjednodušené funkce. I tato metoda má však svá omezení. Délka výpočtu algoritmu Quinn-McCluskey roste exponenciálně v závislosti na vstupu. Jedná se o NP-úplný problém[4].

Vedle správné funkčnosti navrženého obvodu je taktéž důležitá jeho rychlost. Proto je časová optimalizace významnou součástí optimalizačních technik syntézního procesu. Pro tuto optimalizaci je stěžejní správně provedená analýza časování. Ta se odvíjí od výpočtu zpoždění logických členů (hradel a propojů)[11] v obvodu a také od zpoždění celého obvodu.

Nejčastější je tzv. topologická časová analýza. Obvykle se měří pomocí fixního modelu, kdy každé hradlo i spoj má své dané zpoždění. Typicky se měří následující údaje:

- příchozí čas signálu[12] – čas, kdy se hodnota signálu ustálí,
- čas, po který je třeba, aby hodnota signálu zůstala stabilní[13],
- tzv. slack – rozdíl mezi druhým a prvním výše uvedeným časem. Negativní slack indikuje chybu v časování

Jsou-li známy příchozí časy všech signálů na primárních vstupech, je možné vypočítat příchozí časy na každém hradle v topologii obvodu směrem k primárním výstupům. Obdobně je možné dopočítat čas, po který musí být signál na vstupu hradla stabilní, tentokrát však od primárních výstupů k primárním vstupům.

[11] např. propagační zpoždění hradel, u propojů se pak jedná hlavně o zpoždění vlivem parazitních odporů a kapacit

[12] angl. arrival time

[13] angl. required time

Topologicky nejdelší cesta v obvodu je ta, která má minimální hodnotu slack. Zpoždění této cesty je označováno jako kritické – proto se nejdelší cesta v obvodu často označuje jako kritická cesta.

Slabým místem topologické časové analýzy je fakt, že ne každá nalezená kritická cesta musí nutně ovlivňovat zpoždění daného obvodu – může jít o falešnou kritickou cestu[14]. Kritické zpoždění obvodu je definováno jako zpoždění jeho skutečné nejdelší cesty[15]. Kritické zpoždění kombinačních obvodů je totiž dáno také Booleovskou funkcionalitou každého jejich prvku, což ovšem topologická časová analýza nebere v úvahu[5].

Časovou optimalizaci lze provést během technologicky nezávislé části syntézy nebo během technologického mapování. V případě provedení během technologicky nezávislé části se využívá restrukturalizace a redekompozice, kdy optimalizační algoritmus manipuluje s topologií obvodu za účelem jeho zrychlení, dokud nejsou splněna všechna časová omezení nebo dokud lze snižovat zpoždění obvodu. Rekompozicí kritických cest lze obvod výrazně zrychlit, je ale třeba brát v úvahu možnost vzniku nových kritických cest.

2.3 Evoluční programování a evoluční syntéza

Úkolem evolučního programování je nalezení propojení funkčních bloků obvodu(a tím i funkcí, které implementují) podle předem dané specifikace (např. pravdivostní tabulky). Hlavním záměrem je vytvořit kvalitnější, efektivnější či inovativní obvodové struktury[6]. Je třeba rozlišovat mezi dvěma přístupy[3]:

- evoluční návrh obvodu – jedná se o evoluci jediného obvodu
- evoluční hardware – zde se evoluční algoritmus stará o kontinuální adaptaci systému v měnících se okolních podmínkách.

V dalším textu bude uvažován především první zmíněný přístup.

Základní myšlenkou evolučního návrhu je popis obvodu jako tzv. chromozomu. Chromozom je obvykle reprezentován jako binární řetězec kódující daný obvod. Na tento chromozom je poté možné aplikovat genetické operátory, jakými jsou například mutace, křížení či selekce, kombinací kterých pak z původních chromozomů vznikají noví jedinci (jedná se o tzv. populaci kandidátních řešení). Použitím genetických operátorů se zvyšuje různorodost populace, a tak vzrůstá i pravděpodobnost nalezení vhodného výsledku[12].
Nedílnou součástí je také ohodnocení jedinců vzniklé populace fitness funkcí. Tato funkce bývá založena na porovnání výsledků produkovaných jedincem po předložení ověřovacích dat na jeho vstupy s referenčními výsledky. Jedná se tedy o simulaci funkčnosti nově vzniklého obvodu. Tato simulace může být dvojího druhu[3]:

- intrinsic – chromozom se konvertuje na binární řetězec, kterým je poté naprogramován rekonfigurovatelný hardware; tato metoda umožňuje zahrnout do testování i aspekty, jakými jsou např. teplota, elektromagnetické pole, vlastnosti čipu a další,
- extrinsic – chromozom je transformován na model obvodu, jehož činnost je poté softwarově simulována (např. CGP, viz dále).

[14] angl. false critical path
[15] angl. true critical path

Každému jedinci je tak přiřazena hodnota, která vyjadřuje úroveň, do jaké jedinec splňuje očekávání.

Pro další evoluční iteraci je vybrán jedinec (případně množina jedinců) s nejvyšší hodnotou fitness funkce (nebo s hodnotou vyšší než je předem daná mez). Evoluční proces je zastaven v momentě, kdy bylo nalezeno řešení s maximálním ohodnocením fitness funkce, případně když už nedochází k výraznému zlepšení hodnoty fitness. Mezi jedinci poslední evoluční generace je možné najít i více než jedno vyhovující řešení daného problému[13].

Oproti klasické syntéze obvodů poskytuje evoluční syntéza využité různých biologií inspirovaných algoritmů, pomocí kterých je syntetizován výsledný obvod.

Využití evoluce v syntézním procesu tak může vést k lepším či zajímavějším výsledkům, než jaké by mohla nabídnout klasická syntéza. Takto vytvořené obvody mohou být rychlejší, méně náročné na spotřebu, či mohou zabírat méně plochy a potřebovat menší množství komponent. Výsledek také může být originální a nabývat podoby, jaká původně nebyla vývojářem uvažovaná, a může tak přispět k přezkoumání celého řešeného problému a případně vyústit v neobvyklá a efektivní řešení.

2.3.1 Kartézské genetické programování

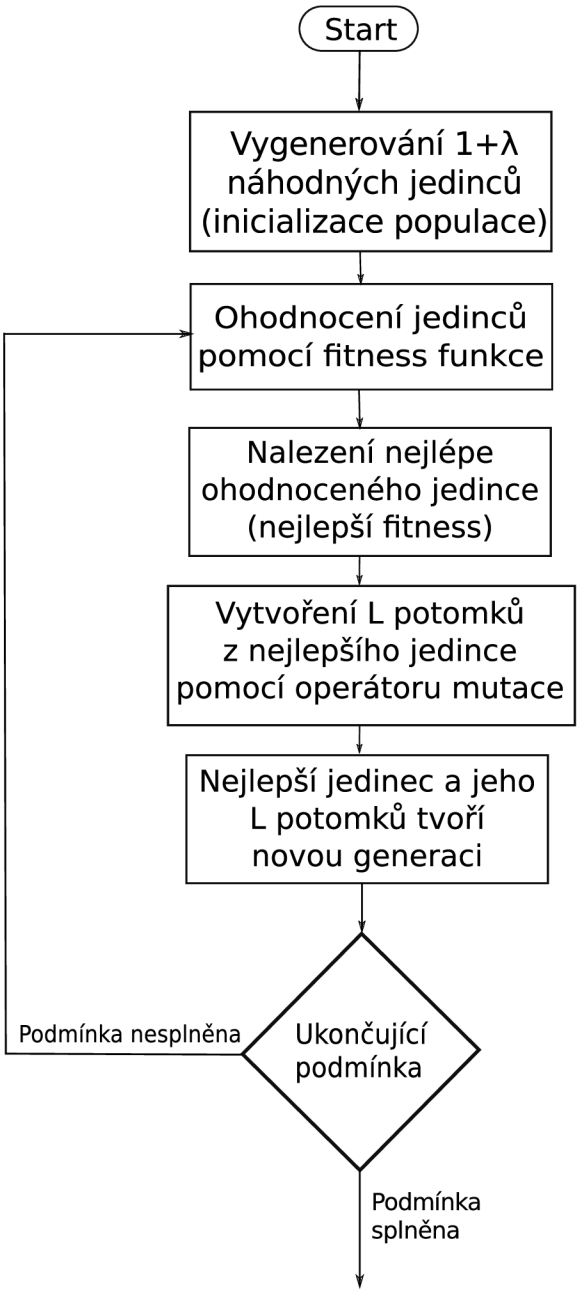
Příkladem využití evolučních principů v programátorské praxi může být kartézské genetické programování (dále jen CGP), které je považováno za jednu z nejefektivnějších metod návrhu kombinačních obvodů a jejich optimalizace po proběhnutí syntézy. Optimalizace pomocí CGP se využívá za účelem co největšího zmenšení plochy nutné k umístění obvodu. Ostatní kritéria, jako zpoždění či spotřeba energie nebývají pomocí CGP řešena.

Jedinci jsou v CGP reprezentováni pomocí orientovaných acyklických grafů. Hlavním genetickým operátorem je mutace, která mění h náhodně vybraných genů v chromozomu. Jako evoluční strategie se uplatňuje princip 1+λ, tedy každá nová generace je tvořena novými jedinci a jejich tzv. rodičem (jedincem, který získal v předchozím vyhodnocení nejvyšší hodnotu fitness a byl vybrán jako zdroj pro tvorbu nových jedinců). Pokud se v předchozí generaci vyskytuje více vhodných rodičovských jedinců, je vybrán ten, jenž nebyl rodičem současné generace – tímto způsobem je zajištěna různorodost populace. Výpočet je zastaven po splnění ukončující podmínky (obvykle po dosažení nejlepší hodnoty fitness či po několika opakováních výpočtu, během kterých již ke zlepšení fitness nedošlo)[18]. Průběh CGP je graficky znázorněn na obrázku 2.1.

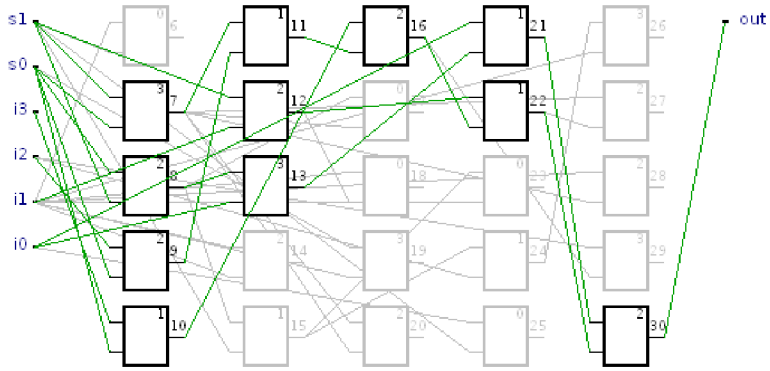
V CGP je obvod situován do podoby pole programovatelných jednotek (hradel) o n_c sloupcích a n_r řadách. Pole má fixní počet vstupů (n_i) a výstupů (n_o). Každé hradlo lze připojit k primárním vstupům, logickým konstantám, a nebo k výstupům hradel umístěných v předchozích l sloupcích[16]. Zpětná vazba v obvodu není podporována[18]. Příklad reprezentace obvodu v CGP znázorňuje obrázek 2.2

Chromozom se skládá z n_c*n_r trojic celočíselných hodnot. První a druhá položka trojice jsou čísla výstupů předchozích hradel, které jsou připojeny na vstup hradla popsaného touto trojicí. Třetím parametrem je pak číselný identifikátor funkce, kterou hradlo realizuje. Na konci chromozomu je připojena l-tice, která obsahuje číselné identifikátory hradel, jejichž výstupy jsou zároveň primárními výstupy celého obvodu.

[16] l-back parametr



Obrázek 2.1: *Grafické znázornění průběhu CGP.*



Obrázek 2.2: *Multiplexor vytvořený pomocí CGP.*

Každé hradlo je schopno implementovat jednu z funkcí definovaných v množině funkcí Γ. Hradlo je kódováno $n_a + 1$ hodnotami, kde čísla 1 až n_a vyjadřují indexy vstupních signálů a poslední číslo označuje typ funkce hradla.

Fitness funkce je v tomto případě založena na simulování funkčnosti obvodu a kontrole jeho výstupů nebo alternativně na tzv. function equivalence checking. První zmíněná varianta výpočtu fitness funkce zahrnuje otestování 2^n vstupních kombinací pro obvod s n primárními vstupy. Hodnota fitness pak odpovídá počtu shod s předem zadanými výstupními hodnotami pro všechny vstupní kombinace. Čas, potřebný k vyčíslení všech možných vstupních vektorů, s každým dalším primárním vstupem exponenciálně vzrůstá, což značně omezuje škálovatelnost. Tato metoda je tudíž výhodná při práci s obvody o cca 10-20 primárních vstupech.

Druhá varianta, equivalence checking, porovnává každý vzniklý kandidátní obvod s referenčním obvodem za účelem zjištění jeho funkcionality. V případě, že je obvod korektně funkční, je jeho hodnota fitness vypočtena na základě počtu jeho hradel. Jedná se tedy o porovnání dvou Booleovských funkcí a určení jejich ekvivalence[17] [18].

Čas potřebný pro equivalence checking je možné zredukovat, je-li brán v úvahu operátor mutace. Hlavní myšlenkou v tomto případě je fakt, že původní obvod a obvod vzniklý mutací (potomek) mají určité své části (podobvody) shodné. Potom tedy stačí ověřit u nového obvodu funkčnost těch částí, které se s jeho rodičovským obvodem neshodují[19]. Tento princip je využitelný při tvorbě sat-solveru sloužícího k ověřování výsledků CGP.

2.3.2 Problémy evolučního programování

Problémem evolučního programování je zejména velikost vytvářeného obvodu. Čím více logických členů obvod obsahuje, tím více existuje možností změn a propojení jednotlivých částí. Další problémovou oblastí je ověřování funkčnosti řešení simulací (stejná metoda je využívána výše zmíněnou fitness funkcí), kdy simulace složitého obvodu může trvat neúnosně dlouhou dobu. Se zvyšující se složitostí obvodu navíc může docházet ke zhoršující se škálovatelnosti. Škálovatelnost je také nepříznivě ovlivněná samotným principem evolučního procesu, kdy jsou evoluční operátory (např. mutace) aplikovány na náhodné části chromozomů[13].

[17] NP-úplný problém

Z tohoto důvodu bylo dosud možné setkat se pouze s malými obvody získanými evoluční cestou. Komplexní obvody je třeba zakódovat pomocí velkých chromozomů, což ovšem zároveň implikuje rozsáhlý prohledávaný prostor řešení[3]. Velikost prostoru, kterou je možné prozkoumat, je závislá na dostupných výpočetních zdrojích. Z toho tudíž plyne limitování komplexnosti obvodů, které je možné evolučně získat, v závislosti na využitelných výpočetních prostředcích.

Možným řešením je využití pouze podmnožiny testovacích vstupů či testování jen určitých strukturních vlastností kandidátních obvodů, což může vést ke kvadratické až lineární časové složitosti vyhodnocení, a to i u komplexních obvodů[9].

Ke zmenšení prostoru kandidátních řešení je možné přispět využitím vyšší úrovně abstrakce. Evoluční algoritmy tak nebudou pracovat například s jednotlivými logickými hradly, ale s jejich množinami tvořícími určité logické celky (např. multiplexor, násobička, ...). Chromozom kódující obvod tak nebude nabývat na velikosti i v poměrně rozsáhlých obvodech. Dochází tak ke značné úspoře času potřebného k evolučnímu zpracování obvodu a k jeho následné simulaci[9].

V [6] je diskutována tzv. dvouúrovňová evoluce. V první fázi je fitness funkce tvořena na základě procentuálního počtu správných výstupů obvodu. V okamžiku, kdy je evolučním algoritmem nalezen obvod se 100% správností výpočtu, přejde se k druhé fázi evoluce. V ní je fitness funkce počítána podle počtu logických hradel, které jsou v obvodu během jeho činnosti skutečně aktivní. Tato metoda tedy dovoluje evolučně vyvíjet obvody, které poskytují správné výstupní hodnoty a jsou zároveň i minimální v případě potřebných hradel. Nutností při použití takového přístupu je ovšem velké množství generací, z čehož vyplývá delší doba potřebná pro nalezení výsledku.

Jako další řešení problematiky evoluce rozsáhlých obvodů se nabízí tzv. inkrementální evoluce.[18] V tomto případě je evoluce nejdříve aplikována na množinu základních prvků. Ty poté slouží jako stavební bloky pro evoluci obvodů.

Při evolučním návrhu obvodů se lze také setkat s pojmem doménová znalost[19]. Jejím hlavním přínosem je definování stavebních bloků obvodu a jejich možného propojení.

Délka chromozomu není vždy přímo spojená s doménovou znalostí. Dlouhý chromozom ovšem obvykle znamená, že o vyvíjeném systému chybí dostatek znalostí a že byla evolučnímu algoritmu ponechána větší volnost při jeho činnosti. Kratší chromozom znamená, že k vznikajícímu systému existuje dobrá informační základna a též představa, jak by měl vypadat[9].

2.4 Přehled existujících syntézních nástrojů

V současnosti je k dispozici značné množství syntézních nástrojů. Následující část textu se věnuje popisu nejznámějších programů. Tyto programy lze rozdělit podle dvou kritérií:

- Komerční a open-source nástroje:
  - komerční nástroje – Xilinx Vivado, Quartus, Precision RTL,
  - open-source nástroje – ABC, SIS, MVSIS, Yosys, Cirkit, QFlow.

[18] v angličtině také pod pojmem *increased complexity evolution*
[19] angl. domain knowledge

- Jakou podobu má výsledný obvod:
  - syntéza do hradel – Yosys, Qflow, Cirkit
  - syntéza do LUT – Xilinx Vivado, Quartus, Precision RTL.

2.4.1 ABC

ABC je open-source systém pro syntézu a verifikaci logických obvodů. Využívá škálovatelnou logickou optimalizaci založenou na AND-inverter (AIG) grafech, DAG-mapping optimalizovaný na zpoždění pro technologické mapování pro LUT i standartní hradla a pokročilé algoritmy pro syntézu a verifikaci.

Program ABC vznikl jako nová verze systému MVSIS. Hlavními vylepšeními byly zefektivnění práce s rozsáhlými obvody, zjednodušení datových struktur, flexibilita binární syntézy, použití AND-inverter grafů pro reprezentaci obvodů, možnost využití SOP a BDD pro řešení specializovaných částí a další. Zavedení AIG dopomohlo k uniformním výpočtům a snadnější komunikaci se SAT solvery využívanými k řešení Booleovských problémů. ABC umožňuje také práci s několika variantami obvodu, a tak podporuje tzv. bezeztrátovou syntézu[20], která nabízí manipulaci s více verzemi syntetizovaného obvodu s různou strukturou a jejich kombinaci, což přispívá k menšímu zpoždění obvodu. Syntézní postup je založen na vyvažování, přepisování a refaktorování AND-inverter grafu[14].

ABC poskytuje příkazy jak pro kombinační tak pro sekvenční syntézu. Kombinační syntéza je založena na přepisování AIG pomocí knihovny obsahující předpřipravené čtyřvstupé AIG nebo pomocí zmenšování a přeuspořádávání logických částí AIG o 10-20 vstupech. Experimentálně bylo zjištěno, že kombinace těchto dvou postupů společně s vyvažováním přispívá ke zmenšování zpracovávaného AIG a k redukování počtu jeho úrovní. Narozdíl od starších systémů SIS a MVSIS, které zachovávají vazby mezi uzly, či je inkrementálně mění, dojde na počátku syntézy v ABC ke zničení vazeb mezi uzly pomocí strukturního hashování, které tím tak přetvoří vstupní logickou síť na AIG[14]. Základem sekvenční syntézy je pak tzv. retiming, který ponechává původní strukturu obvodu a zaměřuje se na práci s tzv. latchi.

Technologické mapování na úrovni LUT je v současnosti limitováno maximální možnou velikostí LUT – je možné vytvářet nejvíce šestivstupé LUT. Každá LUT je charakterizována plochou a zpožděním. Mapování využívá klasické algoritmy založené na DAG technologickém mapování za účelem dosažení co nejmenšího zpoždění obvodu. Jedná se o generické LUT mapování, při kterém nejsou brány v úvahu specifika FPGA architektury, která může obsahovat tzv. programovatelné makrocely obsahující například různá logická hradla, ale i LUT.

Mapování na úrovni logických hradel využívá podobný princip jako mapování na úrovni LUT - DAG mapování využívající k-feasible řezy s ohledem na minimální zpoždění[14].

Tento nástroj je využíván např. programem Yosys (sekce 2.4.3) či Cirkit (sekce 2.4.4).

2.4.2 SIS a MVSIS

Tyto programy slouží pro syntézu synchronních i asynchronních sekvenčních obvodů. SIS umožňuje provádět minimalizaci, optimalizace na plochu a zpoždění pomocí retimingu či optimalizaci založenou na standardních Booleovských a algebraických metodách, a také

[20] lossless synthesis

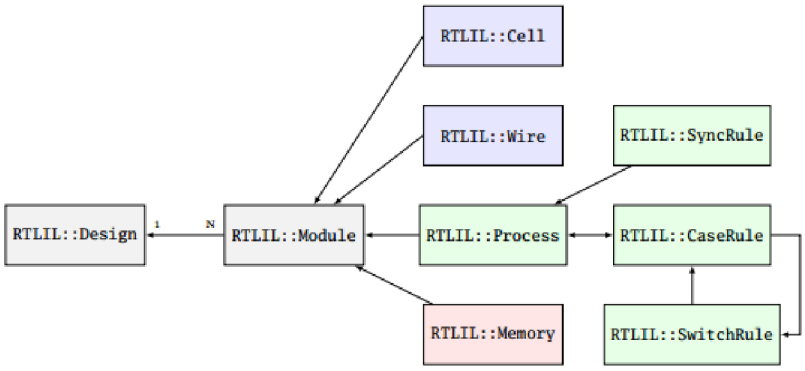
technologické mapování s ohledem na optimální plochu obvodu a zpoždění. Obvod je v SIS reprezentován pomocí DAG[21][16].

MVSIS je novou verzí programu SIS pro multi-valued syntézu a verifikaci. Hlavními položkami, na které se MVSIS zaměřuje, jsou např. generalizace klasické minimalizace binární logiky, využívání resyntézy a minimalizace plochy za účelem zlepšení mapování na úrovni LUT i klasických hradel, kombinační verifikace, a fyzické aspekty, jakým je např. délka propojů. V technologickém mapování zavádí pojem „superhradla“. Jedná se o virtuální hradlo s jediným výstupem složené z více reálných hradel. Tento přístup vede při mapování k lepším výsledkům, protože umožňuje práci s většími celky[15].

2.4.3 Yosys

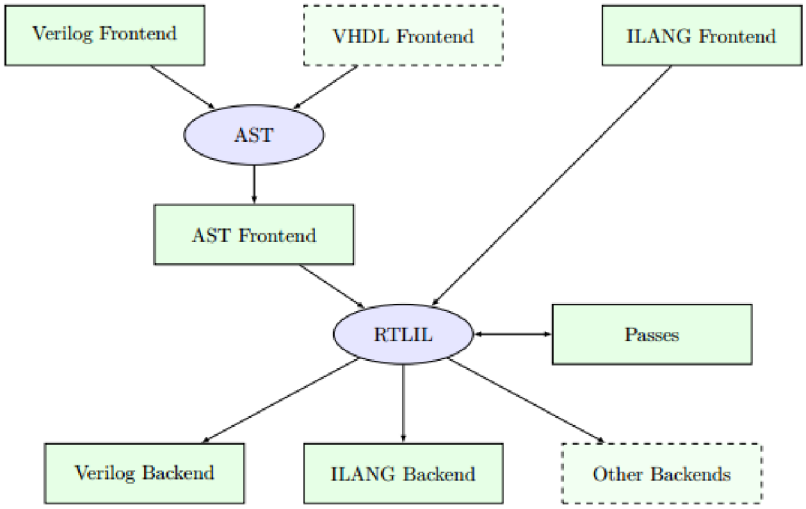
Yosys je Open Source syntézní nástroj vyvinutý Cliffordem Wolfem na Vienna University of Technology. Záměrem tvůrce bylo vyvinout syntézní nástroj, který by byl snadno rozšiřitelný o novou funkcionalitu a tím vhodný například k testování možných nových syntézních metod. Program je napsán v jazyce C++ a je poskytován pod licencí ISC[22].

Tento nástroj používá pro práci s obvody RTLIL[23]. Důvodem je schopnost tohoto popisu reprezentovat vytvářený design během všech částí syntézy. Kořenovým objektem je RTLIL::Design. Jedná se o kontejner jazyka C++, který obsahuje moduly RTLIL::Module. Tyto moduly korespondují s moduly v jazyce Verilog a s entitami jazyka VHDL. Každý RTLIL::Module smí obsahovat objekty RTLIL::Cell, RTLIL::Wire, RTLIL::Process a RTLIL::Memory. Po ukončení syntézy jsou objekty RTLIL:Process a RTLIL::Memory nahrazeny za RTLIL::Cell a RTLIL::Wire. Struktura Yosysu je naznačena na obrázku 2.3. Yosys převádí vstupní data během svého výpočtu tak, jak je znázorněno na obrázku 2.4.



Obrázek 2.3: *Struktura RTLIL v programu Yosys.* [20]

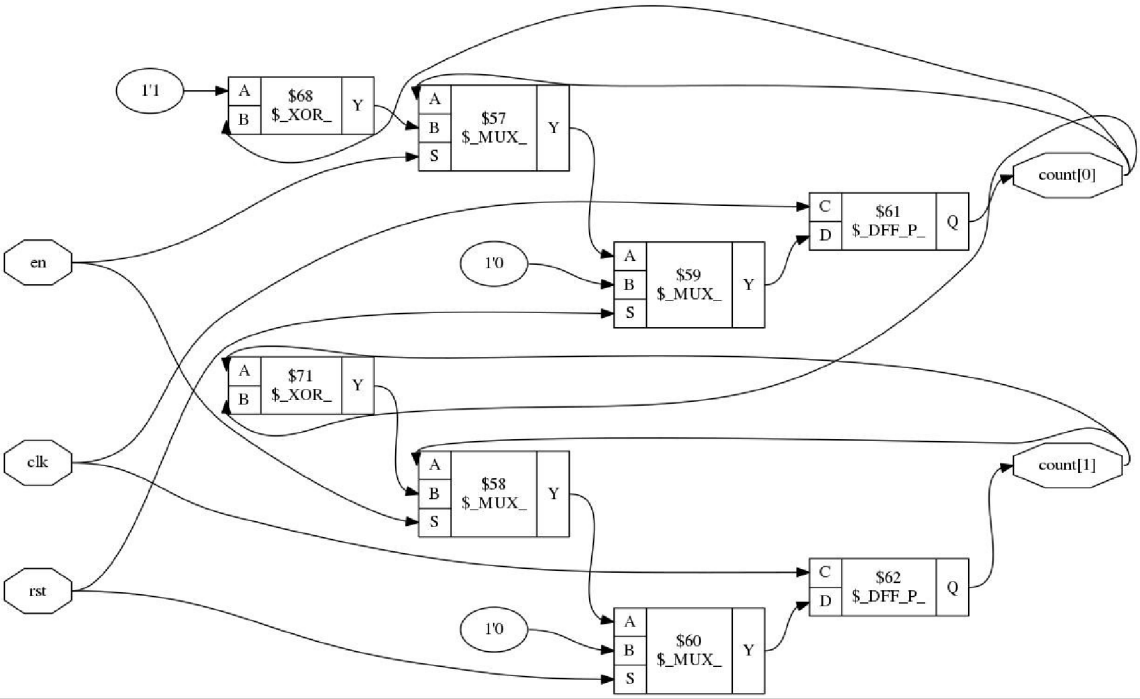
[21] Direct Acyclic Graph
[22] Internet Systems Consortium licence
[23] RTL Intermediate Language



Obrázek 2.4: *Manipulace s daty v programu Yosys.* [20]

Ovládání Yosysu probíhá v prostředí konzole za pomocí příkazů (zde se nazývají „passes“), či syntézních skriptů. Je třeba, aby všechny příkazy pracovaly nad výše zmíněnou strukturou RTLIL.

Je také možné nechat do souboru vytisknout podobu zpracovávaného obvodu po každém kroku práce, jako například na obrázku 2.5. Yosys obsahuje i adresář s několika vzorovými příklady pro seznámení s programem. [20]



Obrázek 2.5: *Reprezentace čítače v programu Yosys (sekvenční obvod).* 2.4.3

2.4.4 Cirkit

Tento open-source syntézní nástroj je vytvořen v jazyce C++ a má pouze konzolové rozhraní. Je možné ovládat ho buď skrze interaktivní rozhraní postupným zadáváním příkazů, nebo přes tzv. bash mode, kdy se programu zadají všechny příkazy jako parametry příkazové řádky, či pomocí tzv. batch mode, kdy program čte příkazy ze vstupního souboru.

Zpracovávaná data jsou ukládána v datových strukturách, jakými jsou např. pravdivostní tabulky, and-inverter grafy či binární rozhodovací diagramy. Každá struktura má svůj vlastní úložný prostor. Cirkit umožňuje využívání nástroje ABC při práci s AND-inverter grafy[10].

2.4.5 Qflow

Qflow je open-source framework, který se zaměřuje na syntézu obvodů popsaných v jazyce Verilog. Při své práci využívá Yosys jako nástroj pro syntaktickou analýzu zdrojových kódů, high-level syntézu, logickou optimalizaci (spolu s programem ABC) a verifikaci. Pro simulaci a testování pak lze využít program iVerilog[11].

2.4.6 Xilinx Vivado synthesis

Vivado je komerčním nástrojem, který poskytuje podporu pro syntetizovatelnou podmnožinu jazyků Verilog, System Verilog a VHDL. Umožňuje práci v tzv. project mode (grafické uživatelské rozhraní) i v tzv. non-project mode za použití TCL[24] příkazů či skriptů.

Syntéza je plně automatizovaná. Nástroj dovoluje opakování syntézních procesů s lišícími se cílovými zařízeními, syntézními či implementačními nastaveními, a také fyzickými a časovými omezeními. Fyzická omezení se týkají např. umístění určitých pinů a bloků jako jsou RAM, klopné obvody či LUT, a též konfigurace zařízení. Časová omezení mají vliv na frekvenční požadavky na vznikající obvod. Nejsou-li specifikovány, program provede časovou optimalizaci na základě délky propojů a umístění funkčních bloků obvodu.

V tomto nástroji je také možné zkoumat schéma syntetizovaného obvodu a jeho hierarchii (funkční bloky, signály přiřazené pinům, atd.)[1].

2.4.7 Quartus

Quartus je komerční platformě nezávislý nástroj pocházející od společnosti Altera. Poskytuje propracované grafické prostředí. Podporuje syntézu obvodů popsaných v jazyce Verilog a VHDL a také ve specifických jazycích firmy Altera, např. AHDL. Pomocí Quartusu lze konfigurovat programovatelná logická zařízení[25] jako např. Arria, Cyclone či Stratix.

Je možné přednastavit určité vlastnosti syntézy, jakými jsou např. určení verze VHDL a Verilogu, specifikace možností uživatelských knihoven, upřesnění optimalizační fáze, a další. Quartus kromě logické syntézy také poskytuje funkční a časovou simulaci, časovou analýzu, programování a kontrolu zařízení, atd[2].

2.4.8 Precision RTL

Precision je komerční nástroj od společnosti Mentor Graphics. Program má grafické uživatelské rozhraní, ale lze ho také ovládat pomocí příkazové řádky. Syntetizované obvody

[24] Tool Command Language

[25] angl. PLD, Programable Logic Devices

mohou být popsány pomocí VHDL či Verilogu. Program posyktuje editor pro tvorbu kódu, pokročilou logickou syntézu, optimalizaci na základě nastavených omezení, časovou analýzu, zobrazení výsledné podoby obvodu, a také jeho kritické cesty.

Kapitola 3

Rozšíření vybraného syntézního nástroje

Tato kapitola se zabývá představením myšlenek a implementace rozšíření vybraného syntézního nástroje. Obsahuje shrnutí důležitých poznatků z oblasti problematiky syntézy a evoluce, možnosti využití grafových algoritmů v logické syntéze, a taktéž návrh samotné implementace rozšíření.

3.1 Hlavní myšlenky

Jak již bylo zmíněno v kapitole 2, syntéza obvodů i evoluční přístupy k ní se potýkají s některými problémy. Hlavním bodem pro tuto práci je otázka optimalizace a škálovatelnosti.

Díky využití evoluce v syntézním postupu dochází k vylepšení optimalizační části, ovšem problémy se škálovatelností přetrvávají. Toto je dáno především principem, na jakém evoluční postup funguje. Jedná se o tzv. generate-and-test princip, kdy jsou evoluční operátory (např. mutace) aplikovány na chromozomy náhodně. Tento přístup je tedy nepříliš výhodný při práci s rozsáhlými obvody, kdy nalezení vhodného kandidátního řešení může být zdlouhavé a výsledek nemusí být optimální.

Možným řešením by bylo ve velkých obvodech vyhledávat menší podobvody a nahrazovat je již existujícími známými a optimalizovanými implementacemi, jak již bylo zkoumáno v řadě prací. Tento způsob ovšem skýtá riziko uváznutí řešení v lokálním extrému, kdy kvůli zmíněným změnám nebude již možné dále obvod evolvovat.

Snahou vyvíjeného rozšíření je tedy vybírat z obvodů podobvody o vhodné velikosti, na které potom bude možné aplikovat evoluční vývin v přijatelném čase. Taktéž je zde záměr vyhýbat se náhradám podobvodů za již známé reprezentace a zajistit tak větší obecnost řešení a možnost vybírání i větších podobvodů.

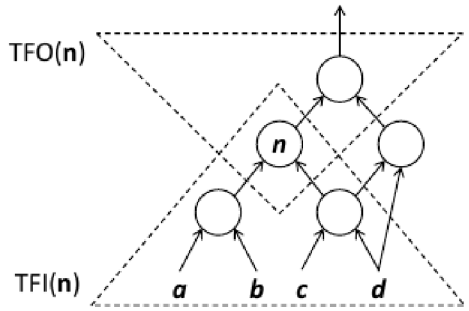
3.2 Grafové algoritmy v oblasti logické syntézy a návrhu rozšíření

V předchozím textu (především v kapitole 2) byl zmíněn častý popis funkcí obvodů pomocí Booleovských výrazů. Tyto výrazy lze transformovat do podoby grafů, jejichž uzly reprezentují dílčí funkce výrazů. Grafová reprezentace je výhodná z mnoha hledisek. Těmi hlavními jsou např. snadná orientace v grafové struktuře, snadnější selekce podgrafů, a také transformace na jinou podobu či typ grafu, čehož se využívá zejména v optimalizační fázi

syntézy. Grafové algoritmy jsou také uplatnitelné při technologickém mapování, kdy se využívá princip pokrytí grafu.

Booleovské výrazy v tomto případě slouží jako předpis pro konstrukci orientovaný acyklických grafů[1].

Graf bude nadále chápán jako množina uzlů V a množina hran E, které uzly propojují: $G = (V, E)$. Každý graf obsahuje primární vstupy PI a primární výstupy PO. Vstupy jednoho uzlu grafu jsou nazývány *fanin* a jeho výstupy *fanout*. *Tranzitivní fanin* jsou všechny uzly na cestách k danému uzlu směrem od PI a *tranzitivní fanout* jsou všechny uzly na cestách od tohoto uzlu směrem k PO. Tyto oblasti popisuje obrázek 3.1



Obrázek 3.1: *Graf: fanin a fanout.* [20]

Pro reprezentaci obvodů lze využít několik typů grafů. Jedním z nich je BDD – binární rozhodovací diagram[2]. Jedná se o orientovaný acyklický graf s koncovými uzly, které obsahují buď 0 nebo 1, a uzly obsahující proměnné reprezentovaného výrazu. Hrany grafu jsou označeny hodnotami 1 nebo 0. BDD je uplatňován při řešení ekvivalence výrazů výrokové logiky a své uplatnění nachází i v oblasti hardwarových optimalizací. Dalším častým typem grafu je BED[3], který obashuje tři druhy uzlů: operátorové (mají přiřazeny dva potomky a libovolnou logickou operaci), proměnné (mají přiřazenu vstupní hodnotu a dva potomky) a koncové (s hodnotou 1 nebo 0). BED je zobecněním BDD grafu.

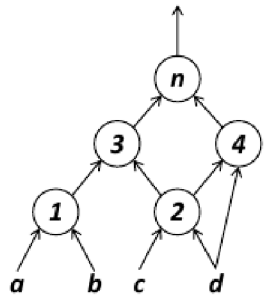
Nejčastějším typem grafu, se kterým se lze v syntézních nástrojích setkat, je tzv. AIG – and-inverter graf. Jedná se o orientovaný acyklický graf.[4] Tento graf je složen výlučně z logických dvouvstupých hradel AND, invertorů a spojů mezi těmito prvky. Výhodou tohoto typu grafu je jeho škálovatelnost.

Příkladem programu, jenž tento typ grafu využívá, je ABC (viz sekce 2.4.1). Během tvorby AIG se v ABC uplatňuje strukturní hashování, které slouží ke kontrole, zda žádné ze dvou AND hradel nemá oba dva své vstupy identické s jiným AND hradlem v grafu. V ABC je používáno přepisování tohoto grafu[5] za účelem optimalizace. Principem je vyhledávání podgrafů a jejich nahrazování již předpřipravenými grafovými konstrukcemi. Pro každý uzel grafu (v topologickém pořadí) se vytvoří všechny jeho 4-feasible řezy (pojem k-feasible řez je vysvětlen níže v této sekci). Následně dojde u každého uzlu s 4-feasible řezy k postupnému nahrazení řezů předpřipravenými podgrafy. Vybráno je poté řešení, které přispívá ke snížení počtu uzlů grafu. Je-li povolen tzv. „Zero-cost replacement“, lze nahradit řez i v případě,

[1] DAG, directed acyclic graph
[2] angl. Binary decision diagram
[3] angl. Boolean Expression Diagram
[4] angl. DAG, directed acyclic graph
[5] angl. rewriting

že žádný nabízený podgraf celkový AIG nezmenší (důležité ale je, aby řešení nezvětšoval). Přepisování AIG je kombinováno s vyvažováním grafu pomocí algebraické redukce výšky grafu. Variantou přepisování je refaktorování AIG, při kterém je vypočítán jediný rozsáhlý řez pro každý uzel grafu. Tento řez je pak nahrazován refaktorovanou podobou jeho funkce v případě, že tím dojde ke zmenšení grafu.

Častou operací, která je nad grafy prováděna, je tvorba řezů. Řezem se rozumí podmnožina uzlů grafu. Každá cesta vedoucí z primárních vstupů grafu do primárního výstupu musí procházet alespoň jedním uzlem z této podmnožiny. Triviální řez zahrnuje pouze uzel, pro který je řez tvořen[6]. Ireduntatní řez je takovou podmnožinou, ze které nelze výběrem uzlů vytvořit jiný řez. K-feasible řez je podgrafem, který obsahuje jeden primární výstup a nejvíce K listových uzlů – funkci takového podgrafu lze pak implementovat pomocí jedné K-vstupé LUT. Pro usnadnění výběru vhodného řezu pro jeho další zpracování se používá proiritizace řezu. K-feasible řezy (či jejich podmnožina) zvoleného uzlu jsou řazeny podle vybraného kritéria (např. počtu uzlů) a n nejlepších řezů je pak vybráno jako řešení. Výhodou prioritizace je zejména zmenšení paměťových nároků při práci s grafem a jeho řezy, protože v každém uzlu se uchovává pouze n řezů namísto všech možných K-feasible řezů. Grafické znázornění řezu je na obrázku 3.2



Obrázek 3.2: *Množina k-feasible řezů pro K=3 a uzel n: C(n) = {{n},{3,4},{1,2,4},{1,c,d},{1,2,d},{2,a,b},{3,c,d},{2,3,d}}.* [20]

3.3 Návrh programového rozšíření

Ve vyvíjeném rozšíření je stěžejní částí vybírání podobvodů a jejich následné zpracování evolučními algoritmy, jak již bylo výše nastíněno. Evoluci zajišťuje již vytvořený evoluční nástroj, a proto je nejpodstatnější částí práce samotná selekce podobvodů.

Za účelem vybírání podgrafů reprezentujících podobvody se jako nejvhodnějším postupem jeví aplikace řezů na grafovou reprezentaci. Uvažovány budou především K-feasible řezy, kde „K“ bude značit počet vstupních hradel vybraného podobvodu a bude tak možné provádět experimenty s různě velkými podobvody.

Evoluční část rozšíření je implementací kartézského genetického programování, které je schopno pracovat s obvodem přímo na úrovni hradel a nevyžaduje jeho konverzi na např. reprezentaci v podobě AIG.

Snytézním programem, pro který bude rozšíření vytvářeno, je Yosys. Vybrán byl pro svou přehlednou interní strukturu, snadnou rozšiřitelnost, a také dobrou podporu a dokumentaci. Více byl Yosys zmíněn v sekci 2.4.3.

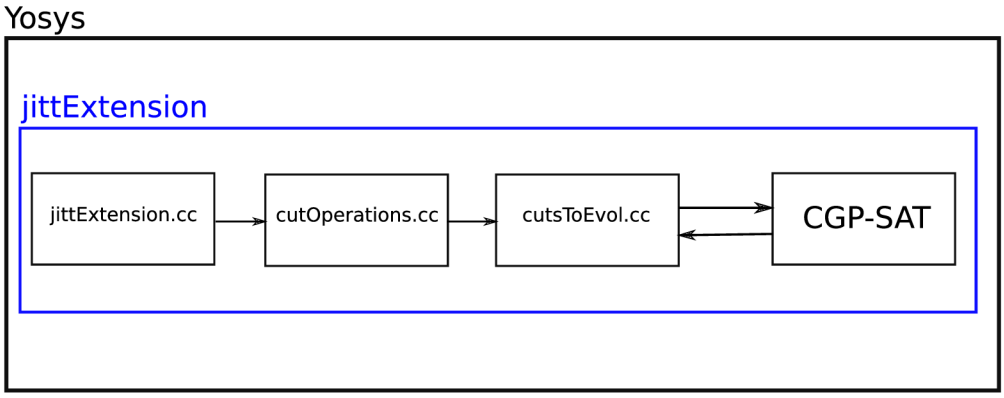
[6] kořen řezu, angl. root

Rozšíření bude obvod zpracovávat v rámci optimalizačních fází syntézy. Nejprve bude z obvodu pomocí vhodných grafových algoritmů vybrán podobvod. Tento bude poté vložen na vstup využívanému evolučnímu nástroji, kterým bude zpracován, a poté odeslán zpět do rozšíření, které tento evolucí změněný podobvod vloží zpět do původního obvodu.

3.4 Programová realizace rozšíření

V této sekci budou popsány jednotlivé moduly, ze kterých se navržené rozšíření skládá. Programová část této práce byla provedena v jazyce C++ (překladač gcc ve verzi 6.2.1) pro program Yosys verze 0.7+154 na operačním systému Arch Linux.

Rozšíření bylo vytvořeno tak, aby jej bylo možné v prostředí Yosysu spouštět jako samostatný příkaz `jittExtension` s příslušnými parametry, které budou popsány v dalším textu. Veškeré soubory s kódem rozšíření jsou k nalezení mezi zdrojovými soubory v adresáři `yosys/passes/cmds/`. Vzniklé moduly znázorňuje obrázek 3.3.



Obrázek 3.3: *Moduly popisovaného rozšíření Yosysu.*

3.4.1 Modul jittExtension.cc

V prvním modulu se nejdříve přistoupí ke zpracování vstupních parametrů a naplnění struktury `JITT::params params`. Parametry jsou následující:

- `iterations` – počet opakování běhu rozšíření,
- `cutSize` – požadovaná velikost k-feaslible řezu,
- `minCellCount` – minimální požadovaný počet hradel v řezu,
- `maxCellCount` – maximální požadovaný počet hradel v řezu,
- `maxCGPGenerations` – maximální počet generací vzniklých v CGP,
- `maxCGPRuntime` – maximální délka evoluce v sekundách,
- `multiout` – hodnota 1 spustí hlavní verzi rozšíření („multiout“), hodnota 0 pak experimentální verzi („singleout“, více popsána v části 3.4.3).

V tomto modulu se nachází struktura s názvem `ExtensionPass` typu public, která obsahuje metody `help` a `execute`. Metoda `help` implementuje nápovědu. V metodě `execute` je nejprve v cyklu vybrán jeden z možných modulů načtených programem Yosys v předešlých krocích (v dalším textu bude dále předpokládána přítomnost jediného existujícího načteného modulu) a ten je pak předán jako parametr pro funkci `jittExtension(std::vector <std::string> args, RTLIL::Module *module)`.

V této funkci se nejprve inicializují struktura s názvem `tree`:

```
typedef struct tree{
RTLIL::Cell *myCell;
RTLIL::SigSpec myPortA;
RTLIL::SigSpec myPortB;
RTLIL::SigSpec myPortY;
tree *cellA;
tree *cellB;
} tree;
```

Součástí této struktury je ukazatel `RTLIL::Cell *myCell` na strukturu obsahující popis jednotlivých hradel. Dále jsou zde ukazatele na vstupy a výstupy daného hradla, které jsou typu `RTLIL::SigSpec` a také dva ukazatele typu `tree`, jež ukazují na synovská hradla `A` a `B`.

Další potřebnou položkou je množina `std::set<JITT::tree*> mySet` která shromažďuje všechny nalezené buňky (hradla), které zatím nebylo možné využít při stavbě stromu `tree`. Tato množina obsahuje odkaz na položku typu `tree`.

Dalším krokem je procházení jednotlivých hradel přítomných v daném modulu a stavba stromové struktury `tree`. Vytvářený strom má počet listových uzlů dvakrát větší, než je požadovaná velikost K-feasible řezu, aby se zvýšila pravděpodobnost nalezení takového řezu, který by obsahoval právě tolik řezových uzlů, kolik je vyžadováno.

Pro každé hradlo je vytvořena položka typu `tree` a tato je pak buď zařazena do vznikajícího stromu, je-li potomkem některého z hradel, které se již ve stromu vyskytují, a nebo je přidána do seznamu `mySet`. Synovská hradla se do `tree` přidávají na základě porovnání názvů vstupních a výstupních rozhraní (jedná se o strukturu `RTLIL::SigSpec` a `RTLIL::Wire`, porovnáváno je vždy výstupní rozhraní možného synovského hradla s jedním a druhým vstupním rozhraním možného otcovského hradla, které již je přítomné ve vznikajícím stromu).

Komunikační rozhraní lze z buňky typu `RTLIL::Cell` zjistit pomocí iterace přes její spojení:

```
for (auto &it : cell->connections())
{
if(strcmp(it.first.c_str(),"\\a") == 0 )
treeCell->myPortA = it.second; //portA je typu RTLIL::SigSpec

else if(strcmp(it.first.c_str(),"\\b") == 0)
treeCell->myPortB = it.second; //portB je typu RTLIL::SigSpec

else //Y
treeCell->myPortY = it.second; //portY je typu RTLIL::SigSpec
}
```

Název drátu (jde o číselný identifikátor), vedoucího mezi dvěma buňkami, lze zjistit následovně (funkce `as_chunk()` pochází přímo z programu Yosys a zjišťuje, zda je signál vedoucí z daného portu spojený s drátem):

```
string wireName = treeCell->myPortA.as_chunk().wire->name.c_str();
```

Při každém přidání nového hradla ke stromu `tree` se projde seznam již navštívených, ale zatím nevyužitých hradel v `mySet` a pomocí funkce `addCellsFromQueue(tree *newCell, int numOfLeaves)` se zkusí přidat k tomuto novému hradlu možné syny z tohoto seznamu. V případě, že k takovému přidání dojde, opakuje se stejný postup i pro toto nově připojené hradlo ze seznamu. Využitá hradla nejsou jsou poté ze seznamu vymazána, protože je možné, že budou sloužit jako zdroje vstupů u ještě nepřidaných hradel.

Poté, co je ukončena tvorba `tree`, se zavolá funkce `createCut(tree *myTree, int cutSize, RTLIL::Module *module)` patřící do následujícího modulu.

3.4.2 Modul cutOperations.cc

Jak již bylo zmíněno výše, první volanou funkcí z tohoto modulu je `createCut(tree *myTree, int cutSize, RTLIL::Module *module)`. Nejprve je inicializována struktura `cutTree`:

```
typedef struct cutTree{
RTLIL::Cell *cutCell;
cutTree *cutTreeA;
cutTree *cutTreeB;
cutTree *cutTreeParent;
int wireA;
int wireB;
int wireY;
} cutTree;
```

Každá proměnná tohoto typu obsahuje ukazatel na hradlo typu `RTLIL::Cell *myCell`, ukazatele typu `cutTree` na synovské uzly i na rodičovský uzel, a také identifikátory vstupních a výstupních rozhraní typu `int` (získány z původního ukazatele typu `RTLIL::SigSpec` z výše popsané struktury `tree`):

```
myCutTree->wireA = atoi(treeCell->myPortA.as_chunk().wire->name.c_str());
                                                                    .
```

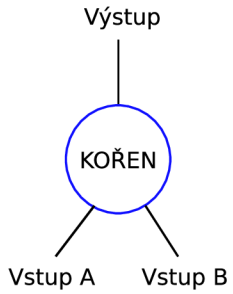
Také se inicializuje první položka jednosměrně vázaného seznamu `cutCellsQueue`, který slouží k procházení stromu `tree` do šířky.[7] V této struktuře je ukazatel na jednu buňku řezového stromu `cutTree`, ukazatel na následující položku v seznamu a proměnná typu `int` značící, zda je buňka v původním stromu levým či pravým synem (pokud je vlevo, proměnná má hodnotu rovnu jedné, pokud je vpravo, hodnota je rovna dvěma). Do seznamu se na jeho konec postupně ukládají ukazatele na právě přidané buňky řezu.

Po inicializaci obou struktur se přistoupí k volání funkce `getCut(cutTree *myCutTree, tree *origTree, int cutSize, int demandedCutSize)`, která má za úkol vyhledat K-feasible řez o zadané velikosti v daném stromu `tree`. Nejdříve dojde k přidání synovských

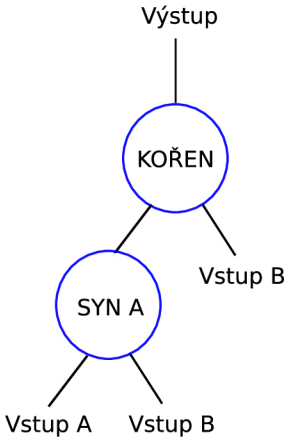
[7] angl. BFS – breadth first search

uzlů kořenového uzlu stromu typu `cutTree`, který je již obsažen ve výše popsaném seznamu typu `cutCellsQueue`. Na tento uzel ukazuje ukazatel `workCutCell`. Synovské uzly jsou také konec seznamu typu `cutCellsQueue` pomocí ukazatele `lastCutCell` a funkce `insertCellToQueue(cutTree *tCell, cutCellsQueue *lastQueueCellPtr, int pos)`. Poté dojde k posunu ukazatele `workCutCell` na další uzel v seznamu a celý proces se opakuje. Dochází tak k průchodu do šířky původním stromem.

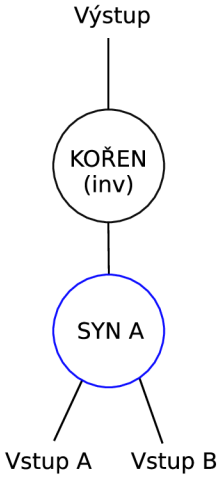
Při každém přidání dvou synovských uzlů se velikost K-feasible řezu zvýší o jedna. Pokud je jeden vstup uzlu primárním vstupem celého obvodu a přidá se pouze jeden synovský uzel, velikost se také o jedna zvýší, protože do řezu je nyní započítáno i toto otcovské hradlo. Tyto různé situace jsou znázorněny na obrázcích 3.4, 3.5, 3.6, 3.7.



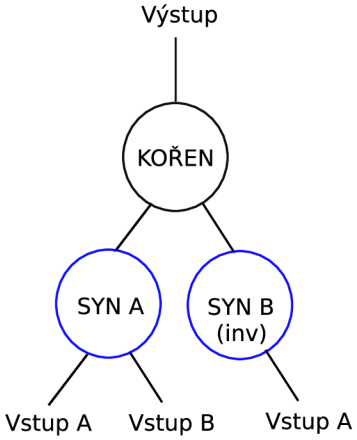
Obrázek 3.4: *Řez o velikosti jedna obsahující pouze kořenový uzel. Modře zvýrazněné hradlo tvoří řez.*



Obrázek 3.5: *Řez o velikosti dva obsahující kořenový uzel a jeden synovský uzel. Modře zvýrazněná hradla tvoří řez.*



Obrázek 3.6: *Řez o velikosti jedna obsahující kořenový uzel a jeden synovský uzel. Modře zvýrazněné hradlo tvoří řez.*



Obrázek 3.7: *Řez o velikosti dva obsahující kořenový uzel a dva synovské uzly. Modře zvýrazněná hradla tvoří řez.*

Pro přidávání uzlů k řezu slouží funkce `addCellToCut(cutTree *myCutTree, tree *newCell, int position)`, která u otcovské buňky `myCutTree` nastaví ukazatel na novou buňku stejného typu. Tato nová buňka získá odkaz na hradlo typu `RTLIL::Cell` a též číselné identifikátory komunikačních rozhraní hradla (položky `int wireA, int wireB, int wireY`), které získá z původní stromové buňky `tree *newCell`.

Otcovská buňka ukazuje na nového syna buď ukazatelem `cutTree *cutTreeA` nebo `cutTree *cutTreeB` podle hodnoty parametru funkce `int position`. Zároveň se u synovské buňky nastaví ukazatel na svého otce `cutTree *cutTreeParent`.

V momentě, kdy velikost řezu dosáhne požadované velikosti nebo kdy již nelze řez více rozšířit, ukončí se jeho stavba a dojde k volání funkce `mainCutEvo(cutTree *myCutTree, RTLIL::Module *module)` z následujícího modulu.

3.4.3 Modul cutsToEvol.cc

Úkolem třetího modulu je transformovat právě vzniklý řez na tzv. chromozozom, se kterým dále pracuje připojený program kombinující kartézské genetické programování a SAT-solver (v dalším textu pod názvem CGP-SAT). Jako první se zde vstupuje do funkce `mainCutEvo(cutTree *myCutTree,RTLIL::Module *module)`, ze které se spouští funkce potřebné pro převedení řezu na chromozom, jeho odeslání do úvodního modulu CGP-SAT, přijmutí pozměněného chromozomu a jeho transformaci zpět do podoby buněk přítomných v právě zpracovávaném modulu v Yosysu.

První invokovanou funkcí je `cutToEvoRepreMulti(cutTree *myCutTree)`. Jejím jediným vstupním parametrem je ukazatel na kořenový uzel řezového stromu `cutTree *myCutTree`. Nejprve se inicializuje první položka globálního obousměrně vázaného seznamu `globalBFSqueue *g_cutBFSqueue`. Každá položka tohoto seznamu obsahuje ukazatel na jednu buňku původního řezového stromu (`cutTree *myCutTreeCell`), ukazatele na sousední položky buňky v seznamu (`globalBFSqueue *left` a `globalBFSqueue *right`):

```
typedef struct globalBFSqueue{
cutTree *myCutTreeCell;
globalBFSqueue *left;
globalBFSqueue *right;
} globalBFSqueue;
```

Návratovou hodnotou funkce `cutToEvoRepreMulti(cutTree *myCutTree)` je počet primárních vstupů řezu, který získá po volání funkce `cutQueueToEvoMulti(int numCells)` popsané níže.

Poté je volána funkce `cutTreeToQueueMulti()`, která naplní již zmíněný globální seznam `globalBFSqueue *g_cutBFSqueue` díky průchodu řezu do šířky.

Jako další v pořadí se zahájí vykonávání funkce `cutQueueToEvoMulti(int numCells)`, ve které se jako první vytvoří globální mapovací pole `string *g_numMapField` pro jednotlivá hradla a primární vstupy celého řezu. Prvními položkami tohoto mapovacího pole jsou identifikátory primárních vstupů řezu (jedná se o položky `wireA` a `wireB` ze struktury `cutTree`). Pokud právě procházená buňka ze seznamu `globalBFSqueue *g_cutBFSqueue` nemá v řezu svého levého (pravého) potomka, zjistí se číselný identifikátor drátu spojující levý (pravý) port buňky s okolím a poznačí se do pole. Takto se v prvním průchodu seznamem naplní mapovací pole právě názvy primárních vstupů. Zároveň se v tomto kroku spočítá počet primárních vstupů řezového obvodu. V druhém průchodu jsou pak do mapovacího pole přidány názvy jednotlivých buněk řezu směrem od listů ke kořenu řezu. Je důležité zajistit, aby žádná položka v chromozomu neměla na pozicích pro identifikátory vstupů hodnotu větší, než je identifikátor této položky. Proto jsou do mapovacího pole postupně přidávána jen ta hradla, jejichž oba vstupy již mají záznam v mapovacím poli. Názvy hradel je možné získat takto:

```
string cellName = (cell->name).c_str();
```

Druhým krokem v této funkci je převedení řezu do podoby chromozomu. Chromozom je deklarován v souboru *cgp.h*:

```
typedef int* chromozom;
```

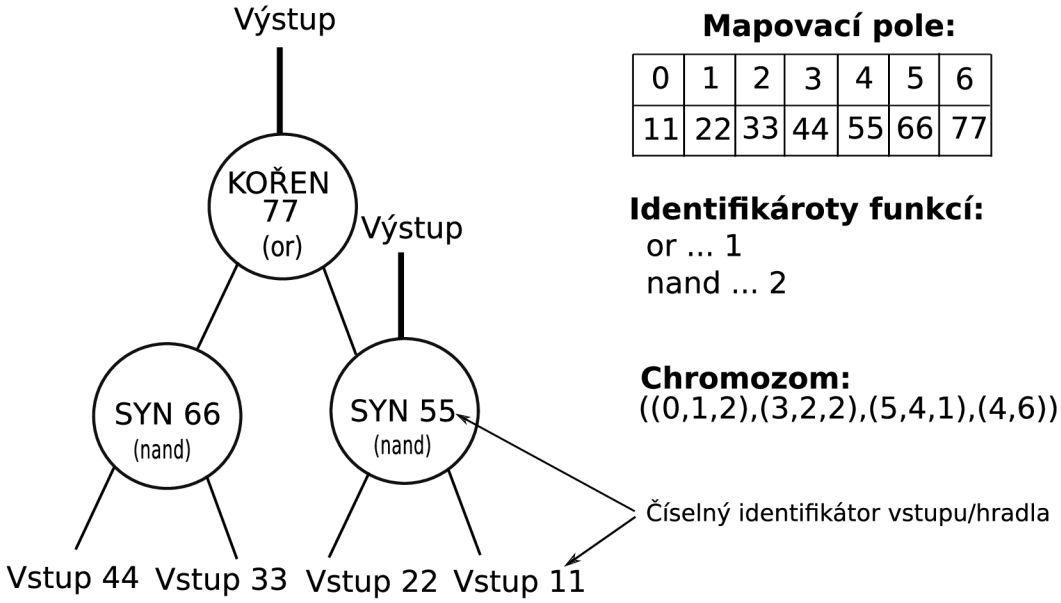
a definován v souboru `chromosome.h`:

```
#define new_chromozome()
new int[outputidx + param_out]
```

Jak je patrné, chromozom je polem hodnot typu `int`. V modulu `cutsToEvol.cc` je reprezentován globálním polem `int *g_chromFieldSEND`. Pro každou buňku v řezu se do chromozomu přidají tři údaje:

- index levého vstupu,
- index pravého vstupu,
- číselné označení typu funkce, kterou buňka implementuje.

Seznam řezových buněk je procházen zprava doleva, vyhledají se identifikátory vstupů a typ funkce buňky. Do chromozomu se přidají indexy, na kterých jsou dané vstupy v mapovacím poli a nakonec identifikátor funkce buňky. Poslední položkou chromozomu je n-tice, která obsahuje identifikátory buněk, které slouží jako zdroj výstupu obvodu. Příklad řezu, jeho mapovacího pole a chromozomu se nachází na obrázku 3.8.



Obrázek 3.8: *Příklad řezu, mapovacího pole a chromozomu.*

Návratovou hodnotou funkce `cutQueueToEvo(int numberOfCells)` je počet primárních vstupů řezu.

V okamžiku, kdy je chromozom připravený, dojde k zavolání funkce `yosys_main(int numOfInputs)`, která náleží do CGP-SAT modulu. Po skončení práce CGP-SAT se tok programu vrací zpět do tohoto modulu *cutsToEvol.cc*. Jeho druhou důležitou částí je totiž převedení chromozomu pozměněného pomocí CGP-SAT zpět do podoby hradel v právě zpracovávaném obvodu v Yosysu.

Jako poslední se vykoná funkce `createNewSubcktMulti(int numOfInputs, RTLIL::Module *module)`, která se věnuje samotnému převedení chromozomu na hradla. Nový chromozom `int *g_chromFieldRECV` je procházen zleva doprava. Nejdříve jsou zjištěny reálné identifikátory vstupů hradla na základě práce s mapovacím polem `int *numMapField`.

Pokud je namapované číslo vstupu menší než počet primárních vstupů původního řezu, vyhledá se reálný identifikátor pomocí pole `int *numMapField`. Je-li tomu naopak, zavolá se funkce `getCellMapID(int mapNum, newCells *myNewCells)`, jejíž první argument je identifikátor vstupu, který ovšem patří hradlu, které vzniklo v předchozích krocích funkce na základě informací z `int *g_chromFieldRECV`. Druhým argumentem je první položka obousměrně vázaného seznamu typu `newCells`, který zaznamenává ukazatele na nově vzniklá hradla společně s jejich reálnými i namapovanými (podle modulu CGP-SAT) identifikátory. Ve funkci se v tomto seznamu vyhledá hradlo s odpovídajícím namapovaným identifikátorem a nazpět se vrátí hodnota reálného identifikátoru. Třetí položka z chromozomu popisuje funkci nového hradla.

```
typedef struct newCells{
RTLIL::Cell *cell;
int mapID;
int origID;
newCells *left;
newCells *right;
} newCells;
```

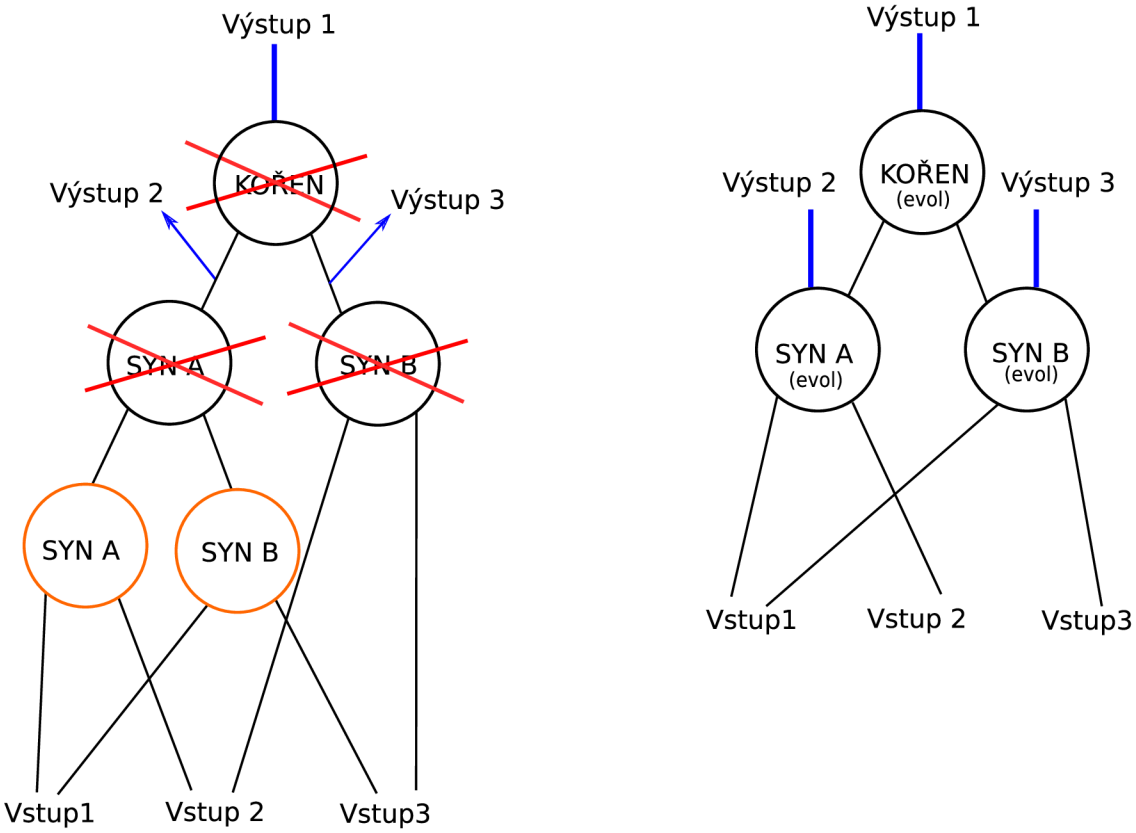
Předtím, než se přistoupí k přidání nového hradla do původního obvodu, se ověří, zda dané hradlo ještě není v obvodu přítomno. To se provede vykonáním funkce `cellExists(int in1, int in2, std::string funkce)`, jejímiž parametry jsou reálné indentifikátory obou vstupů a název funkce, které hradlo implementuje. Protože je pravděpodobné, že shoda nastane v oblasti řezu, porovnají se na shodu pouze buňky z `globalBFSqueue *g_cutBFSqueue` (obsahuje všechny buňky řezu v pořadí podle průchodu do šířky – viz. výše). Nastane-li shoda, funkce vrátí ukazatel typu `RTLIL::Cell` na shodné hradlo, který je pak přiřazen k nové položce v seznamu `newCells`; poté se pokračuje v průchodu chromozomem dále.

Pokud žádné hradlo se stejnými parametry neexistuje, je třeba ho vytvořit. Jako první se do obvodu přidá nová buňka dané funkce pomocí příkazu:

```
RTLIL::Cell newYosysCell = module->addCell(strCellName.c_str(),"\\and2");
```

Pro `newYosysCell` se poté spojí vstupní porty s odpovídajícími komunikačními cestami. Ty jsou nalezeny jako položky typu `RTLIL::Wire` u každé z vstupních hradel (jsou vyhledány výstupní cesty); výstupní port je pak spojen s nově vytvořenou položkou `RTLIL::Wire`, která reprezentuje výstupní drát hradla, a stejně jako hradlo je i ona přidána do obvodu.

Je-li zpracovávané hradlo hradlem, které je jedním ze zdrojů primárního výstupu nově tvořené části, je jeho výstupní port spojen s drátem, na který byl napojena odpovídající buňka původního řezu. Je důležité, aby se nové hradlo připojilo na správný výstupní spoj. Proto je při této operaci využito pole `string *outputField`, které obsahuje názvy výstupních drátů řezu. Index, který se použije pro výběr odpovídající položky z tohoto pole, odpovídá indexu, na kterém se nachází identifikátor zpracovávaného hradla v n-tici výstupů v chromozomu. Tato závěrečná n-tice obsahuje identifikátory výstupních hradel ve stejném pořadí, v jakém byly i výstupy pro vstupní chromozom pro modul `CGP-SAT` a v jakém jsou i položky pole `string *outputField`. Následně již dojde k napojení daného drátu na nové hradlo. Aby se zamezilo vícenásobným zdrojům pro jeden spoj, je původní hradlo z obvodu smazáno. Po zapojení hradla do obvodu se přistoupí k načítání prvního vstupu dalšího hradla v chromozomu. Graficky je postup přepojení znázorněn na obrázku 3.9.

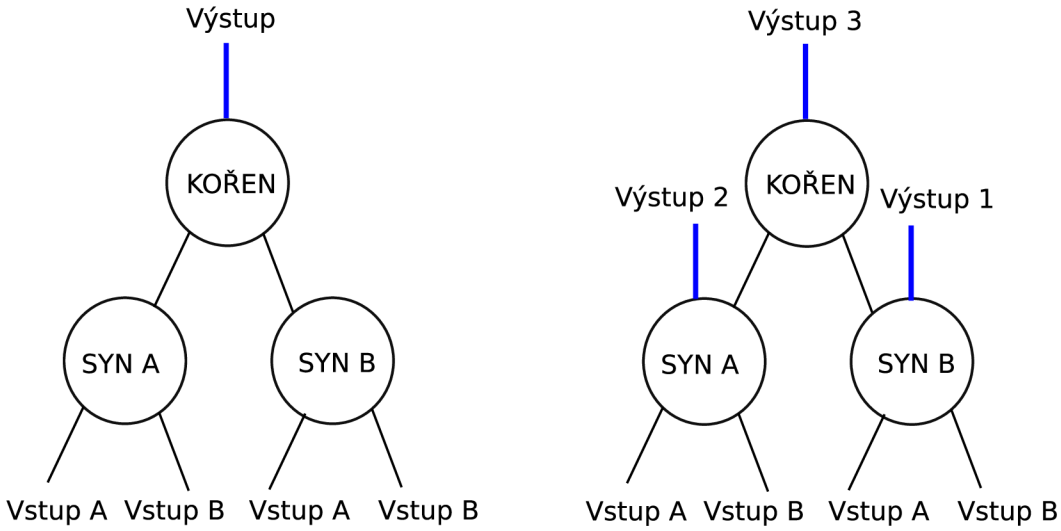


Obrázek 3.9: *Vlevo je původní řez s modře vyznačenými výstupy. Vpravo je evolvovaný řez se stejným počtem výstupů napojených na původní výstupní rozhraní. Vstupy, využívané řezem vlevo, jsou také využity. Červeně přeškrtnutá hradla jsou odstraněna a nahrazena odpovídajícími evolvovanými hradly v obvodu vpravo. Oranžová hradla jsou ponechána v původním obvodu a po skončení rozšíření případně vyoptimalizována Yosysem.*

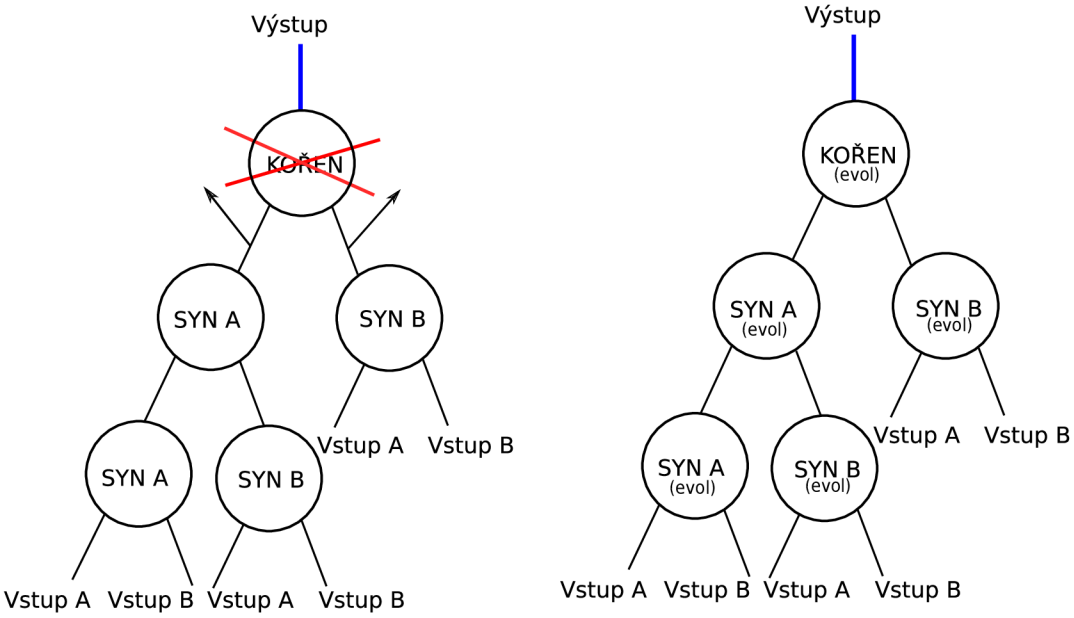
Vykonávání rozšíření tímto krokem skončí a dále již lze spustit jakýkoli další příkaz Yosysu.

Popsaná verze modulu `cutsToEvol.cc` vytváří takové řezy, které jsou po navrácení do původního obvodu napojeny na všechny předchozí primární vstupy i výstupy původního řezu. Experimentálně byla též vyvinuta tzv. „singleout verze“, která se nachází ve stejném modulu a je možné ji spustit pomocí přepínače `-multiout 0`. Narozdíl od hlavní verze vytváří „singleout verze“ řezy s jediným výstupem, který je napojen na kořenové hradlo řezu. Důvodem pro realizaci tohoto řešení byla otázka, zda takový přístup bude mít podobné či horší optimalizační výsledky, než hlavní verze rozšíření. Experimenty odhalily, že obvod se průchodem „singleout verze“ zvětšuje. Příčinou tohoto trendu je tvorba řezu s právě jedním výstupem, kdy lze v původním obvodu bezpečně nahradit pouze kořenové hradlo řezu, a to nově vzniklým kořenovým hradlem. Toto však nepřináší žádné zmenšení obvodu. Ostatní hradla původního řezu musí zůstat na svých místech, protože jejich výstupy mohou potenciálně být zdroji signálu pro hradla, která se nachází v jiné části obvodu. Dojde tak navíc k přidání evolvovaných hradel do obovodu a tím i k jeho zvětšení. Na základě tohoto zjištění nebyla „singleout verze“ dále vyvíjena a důraz byl kladen především na „multiout verzi“. Obrázky 3.10 a 3.11 popisují rozdíly mezi „singleout verze“ a hlavní verzí..

Verzi rozšíření tvořící řezy s n vstupy a m výstupy se nachází mezi zdrojovými kódy v adresáři `yosys-multioutput`.



Obrázek 3.10: *Vlevo je řez s jedním výstupem vyvedeným z kořenového uzlu, který vzniká v „singleout verzi“. Vpravo je řez z „multiout verze“ s totožnými buňkami s výstupy z kořene a jeho obou synů.*



Obrázek 3.11: *Vlevo je původní řez, jehož kořen je po evoluci nahrazen novým kořenem. Tento nový kořen je připojen na stejný výstup, jako kořen původní. Hradla řezu vlevo nejsou z obvodu odstraněna, protože mohou sloužit jako zdroje vstupů pro hradla v jiné části obvodu. Navíc jsou do obvodu přidána všechna vyevolvovaná hradla řezu vpravo.*

3.4.4 Modul CGP SAT

Tento modul je již existující implementací kartézského genetického programování a SAT solveru, které společně evolvují a ověřují vybraný řez původního obvodu.[21] Celý modul je obsažen v adresáři `cgp-sat`. Princip kartézského genetického programování byl již zmíněn v části 2.3.1.

Za účelem propojení modulu CGP-SAT a kódu rozšiřujícího Yosys o možnost vytvoření řezu bylo třeba kód CGP-SAT rozšířit o vlastní funkce starající se o načtení parametrů a chromozomu, a také o uložení pozměněného chromozomu do globálního pole `int *g_chromFieldRECV`.

V souboru `cgp-sat/chromosome.h` tedy vznikly následující tři funkce. První z nich je `my_paramSetup(int numOfInputs, int introns = 0, int init_params = 0)` a slouží k nastavení parametrů CGP matice pro reprezentaci chromozomu. Důležité parametry jsou:

- `param_in`
  – počet primárních vstupů obvodu, nastaven na hodnotu parametru `numOfInputs`,
- `param_out`
  – počet primárních výstupů obvodu, nastaven na hodnotu 1 (u verze s m výstupy je nastaven na hodnotu m),
- `param_m`
  – počet hradel na řádku CGP matice, nastaven na hodnotu počtu hradel v řezu,
- `param_n`
  – počet hradel ve sloupci CGP matice, nastaven na hodnotu 1,
- `block_in`
  – počet vstupů hradla, nastaven na hodnotu 2,
- `block_out`
  – počet výstupů hradla, nastaven na hodnotu 1.

Druhou funkcí je `my_chrLoad(chromozom p_chrom)`, která implementuje průchod chromozomem v poli `int *g_chromFieldSEND` a kopíruje jednotlivé položky do pole `chromozom p_chrom`:

```
*p_chrom =  g_chromFieldSEND[i];
```

Třetí funkce je `my_chrStore(chromozom p_chrom)`. Pomocí ní jsou zkopírovány hodnoty z nového chromozomu `chromozom p_chrom` do pole `int *g_chromFieldRECV`, se kterým nadále pracuje modul *cutsToEvol*.

Hlavní funkce CGP-SAT byla pozměněna na `yosys_main(int numOfInputs)`. Její kód se od původní funkce `main(int argc, char* argv[])` neliší, pouze načítání a uložení chromozomu a parametrů bylo nahrazeny třemi výše popsanými funkcemi. Tato funkce je volána z `mainCutEvo(cutTree *myCutTree, RTLIL::Module *module)` z modulu `cutsToEvol`.

3.4.5 Možná rozšíření programové části práce

Vzniklý modul sloužící k tvorbě řezů a jejich vylepšení pomocí CGP by bylo možné rozšířit o další funkčnost a tím například potenciálně vylepšit jeho výsledky.

Nabízející se rozšíření by se mohlo týkat implementace tvorby KL-feasible řezů či prioritních řezů. KL-feasible řezy rozšiřují K-feasible řezy zahrnutím „L“ výstupních listů grafu do výpočtu. Jsou tedy použitelné pouze na vnitřní uzly grafu. Dovolují ovšem vybrat z obvodu větší úsek než K-feasible řezy při zachování stejného počtu primárních vstupů a takovém počtu hradel, které CGP zvládne zpracovat[7].

Prioritizace řezů pak může pomoci při selekci jednoho řezu, na který bude aplikována evoluce – řezy by bylo možné řadit podle různých cílů (např. podle plochy, hloubky řezu či počtu vstupů)[8]. V souvislosti s prioritizací by bylo taktéž možné vytvářet více řezů před započetím evoluce a z nich vybrat ten, který se pro daný záměr nejlépe hodí.

Kapitola 4

Experimentální vyhodnocení programového rozšíření

Za účelem experimentů s implementovaným rozšířením bylo vybráno devatenáct různých kombinačních obvodů ze sady QUIP[17], které sloužily jako vstupy pro Yosys, a nad kterými se provedla sada různých testů. Tyto vstupní soubory jsou na přiloženém CD v adresáři `/eval/benchmarks/`. Soubory se liší počtem hradel a jejich propojením.

Z důvodu omezených výpočetních prostředků jsem zvolila následující experimentální nastavení parametrů. Minimální počet hradel byl nastaven postupně na hodnoty 5, 10, 25, 50 a 75. Velikost k-feasible řezu byla zvolena jako $5 * minimalni\_pocet\_hradel$. Hodnoty, na které byly parametry během testování nastaveny, byly zvoleny tak, aby mohla probíhat tvorba velkých i malých řezů. Pro každou zvolenou dvojici těchto měnících se parametrů byl obvod otestován pětkrát. Optimalizace je omezena na dvě hodiny běhu (optimalizace je volána tak dlouho, dokud nevyprší čas). Z důvodu zjednodušení je doba evoluce určena takto: evoluce končí, pokud vyprší 10 sekund nebo 500 tisíc generací. Parametry CGP byly použity ty, které byly experimentálně ověřeny jako nejvhodnější (mutace tří genů, l-back minimální, evoluční strategie 1+1) - viz [17].

4.1 Naměřené výsledky

Tabulka 4.1 uvádí přehled výsledků spuštění rozšíření s různými vstupními obvody s různým nastavením hodnot velikosti řezu a minimálního počtu hradel v řezu (jména sloupců „Vel. řezu“ a „Min. hr.“). Výsledky zahrnují průměrné počty vyoptimalizovaných hradel v daných obvodech a největší dosažený počet odstraněných hradel, a to jak absolutní počet odstraněných hradel, tak procentuální hodnotu vůči celkovému počtu hradel v obvodu.

Vstupní obvod	**Vel. řezu**	**Min. hr.**	**Odstr. hradel**		**% odstraněných**	
			Průměr	**Nejlepší**	**Průměr**	**Nejlepší**
Altera_ac97_ctrl	375	75	0,00	0	0,00	0,00
Altera_ac97_ctrl	250	50	13,00	13	0,09	0,09
Altera_ac97_ctrl	125	25	159,80	181	1,14	1,30
Altera_ac97_ctrl	50	10	657,80	685	4,71	4,90
Altera_ac97_ctrl	25	5	659,00	718	4,72	5,14
Altera_aes_core	375	75	21,50	42	0,11	0,21
Altera_aes_core	250	50	17,40	67	0,09	0,33
Altera_aes_core	125	25	252,00	368	1,26	1,84

Vstupní obvod	Vel. řezu	Min. hr.	Odstr. hradel		% odstraněných	
			Průměr	Nejlepší	Průměr	Nejlepší
Altera_aes_core	50	10	308,20	412	1,54	2,06
Altera_aes_core	25	5	227,00	255	1,13	1,27
Altera_oc_aes_core	375	75	406,20	662	4,24	6,91
Altera_oc_aes_core	250	50	1125,20	1645	11,75	17,18
Altera_oc_aes_core	125	25	1799,00	2790	18,79	29,14
Altera_oc_aes_core	50	10	3108,20	3746	32,47	39,13
Altera_oc_aes_core	25	5	3018,40	3188	31,53	33,30
Altera_oc_aes_core_inv	375	75	882,40	1577	7,80	13,94
Altera_oc_aes_core_inv	250	50	2004,40	2559	17,71	22,61
Altera_oc_aes_core_inv	125	25	2127,80	3893	18,80	34,40
Altera_oc_aes_core_inv	50	10	3996,00	4120	35,31	36,41
Altera_oc_aes_core_inv	25	5	2782,00	3662	24,58	32,36
Altera_cord1	375	75	0,00	0	0,00	0,00
Altera_cord1	250	50	1068,20	1206	10,93	12,34
Altera_cord1	125	25	1300,40	2371	13,30	24,25
Altera_cord1	50	10	2366,80	3001	24,21	30,69
Altera_cord1	25	5	2399,00	3183	24,54	32,56
Altera_cord2	375	75	328,75	336	2,57	2,63
Altera_cord2	250	50	738,60	1126	5,78	8,81
Altera_cord2	125	25	412,40	924	3,23	7,23
Altera_cord2	50	10	199,80	267	1,56	2,09
Altera_cord2	25	5	277,40	549	2,17	4,30
Altera_ethernet	375	75	229,75	377	0,34	0,55
Altera_ethernet	250	50	357,20	558	0,52	0,82
Altera_ethernet	125	25	104,80	169	0,15	0,25
Altera_ethernet	50	10	17,40	47	0,03	0,07
Altera_ethernet	25	5	43,60	149	0,06	0,22
Altera_oc_ssram	375	75	0,00	0	0,00	0,00
Altera_oc_ssram	250	50	0,00	0	0,00	0,00
Altera_oc_ssram	125	25	0,00	0	0,00	0,00
Altera_oc_ssram	50	10	0,00	0	0,00	0,00
Altera_oc_ssram	25	5	114,00	114	29,16	29,16
Altera_sasc	375	75	0,00	0	0,00	0,00
Altera_sasc	250	50	0,00	0	0,00	0,00
Altera_sasc	125	25	0,00	0	0,00	0,00
Altera_sasc	50	10	59,00	62	7,88	8,28
Altera_sasc	25	5	113,20	118	15,11	15,75
Altera_wb_dma	375	75	374,50	457	8,98	10,95
Altera_wb_dma	250	50	437,80	469	10,49	11,24
Altera_wb_dma	125	25	393,20	529	9,42	12,68
Altera_wb_dma	50	10	498,80	567	11,96	13,59
Altera_wb_dma	25	5	389,00	484	9,32	11,60
Altera_spi	375	75	380,00	854	9,65	21,68
Altera_spi	250	50	276,00	928	7,01	23,56

Vstupní obvod	Vel. řezu	Min. hr.	Odstr. hradel		% odstraněných	
			Průměr	Nejlepší	Průměr	Nejlepší
Altera_spi	125	25	524,20	1131	13,31	28,71
Altera_spi	50	10	413,80	494	10,51	12,54
Altera_spi	25	5	219,20	368	5,56	9,34
Altera_oc_mem_ctrl	375	75	1123,00	2205	6,17	12,11
Altera_oc_mem_ctrl	250	50	703,40	1942	3,86	10,66
Altera_oc_mem_ctrl	125	25	1790,60	3167	9,83	17,39
Altera_oc_mem_ctrl	50	10	2096,60	2991	11,51	16,42
Altera_oc_mem_ctrl	25	5	1509,00	2745	8,29	15,07
Altera_fip_risc8	375	75	147,60	212	1,41	2,03
Altera_fip_risc8	250	50	281,60	616	2,69	5,89
Altera_fip_risc8	125	25	776,40	2663	7,42	25,45
Altera_fip_risc8	50	10	1730,20	2058	16,54	19,67
Altera_fip_risc8	25	5	519,40	877	4,96	8,38
Altera_oc_pavr	375	75	411,00	462	1,83	2,06
Altera_oc_pavr	250	50	244,80	546	1,09	2,43
Altera_oc_pavr	125	25	850,80	1561	3,79	6,96
Altera_oc_pavr	50	10	1411,60	2715	6,29	12,10
Altera_oc_pavr	25	5	1147,00	2085	5,11	9,30
Altera_oc_ata_ocidec3	375	75	116,75	168	2,34	3,36
Altera_oc_ata_ocidec3	250	50	601,80	621	12,04	12,43
Altera_oc_ata_ocidec3	125	25	1087,60	1105	21,77	22,11
Altera_oc_ata_ocidec3	50	10	1143,00	1424	22,87	28,50
Altera_oc_ata_ocidec3	25	5	1347,00	1470	26,96	29,42
Altera_fpu	375	75	944,40	1425	4,15	6,26
Altera_fpu	250	50	990,20	1557	4,35	6,84
Altera_fpu	125	25	614,20	1157	2,70	5,08
Altera_fpu	50	10	3373,80	4110	14,82	18,06
Altera_fpu	25	5	2046,80	3216	8,99	14,13
Altera_mux64_16bit	375	75	984,80	1237	20,75	26,07
Altera_mux64_16bit	250	50	1047,80	1650	22,08	34,77
Altera_mux64_16bit	125	25	1206,80	2005	25,43	42,26
Altera_mux64_16bit	50	10	1682,60	2130	35,46	44,89
Altera_mux64_16bit	25	5	1617,60	1876	34,09	39,54
Altera_oc_oc8051	375	75	290,60	456	1,89	2,97
Altera_oc_oc8051	250	50	300,60	522	1,96	3,40
Altera_oc_oc8051	125	25	530,80	1064	3,45	6,92
Altera_oc_oc8051	50	10	1201,20	2245	7,82	14,61
Altera_oc_oc8051	25	5	2232,00	2862	14,52	18,62
Altera_tv80	375	75	51,20	131	0,54	1,39
Altera_tv80	250	50	27,00	51	0,29	0,54
Altera_tv80	125	25	297,00	628	3,14	6,65
Altera_tv80	50	10	460,00	585	4,87	6,19
Altera_tv80	25	5	143,00	188	1,51	1,99

Tabulka 4.1: *Výsledky pro všechny obvody a parametry.*

Jak je z tabulky 4.1 patrné, některé kombinace vstupního obvodu a parametrů dosahovaly lepších hodnot, než jiné. Výpočet totiž závisí jak na parametrech, tak i na velikosti a struktuře obvodu. Je-li obvod malý, bude velmi obtížné, až nemožné, nalézt v něm řez o mnoha položkách; naopak nastavení, kdy se vytvářejí řezy menší, má mnohem větší šanci na úspěšný průběh většiny běhů a tím pádem i na lepší výsledky. I v případě, že obvod obsahuje veliké množství hradel, může dojít k neúspěšným vyhledáním větších řezů. Pokud totiž obvod nemá dostatečnou hloubku, úspěšná v něm opět budou spíše spuštění požadující vznik menších řezů.

Nižší úspěšnost může být také ovlivněna tím, že tvorba řezu v obvodu začne z náhodného místa. Pokud pro zvolené kořenové hradlo nelze nalézt řez, nedojde k optimalizaci. I přesto, že každé spuštění dostane až sto možností nalezení vhodného řezu, stále je třeba mít na paměti náhodný přístup, a tedy možnost, že se v žádném z opakování vhodný řez nenajde.

Faktorem, který je také třeba vzít na vědomí, je evoluce v CGP. Může se totiž stát, že po začlenění evolvovaného řezu do obvodu se vyoptimalizuje několik původních hradel, ale zároveň se přidá takové množství nových hradel, které počtem vyváží hradla vyoptimalizovaná, a velikost obvodu zůstane stejná.

Z výše uvedených důvodů vyplývá, že je úspěšnost výpočtu je úzce svázána s volbou vhodných parametrů – a to jak v případě zvoleného obvodu, tak v případě nastavení velikosti řezu a minimálního počtu hradel v řezu. Použití stejných parametrů při zpracovávání různých obvodů tedy může vést k rozdílné úspěšnosti. Stejně tak vede k různě dobrým výsledkům zpracování jednoho obvodu při měnících se parametrech. Problematika správného nastavení je vhodným tématem na další výzkum a experimenty.

Tabulka 4.2 uvádí pro každý testovací obvod nejlepší dosažený optimalizační výsledek, tedy maximální počet hradel, který se z daných obvodů o daných velikostech podařilo odstranit. Nejlepší záznam vykazuje dokonce vyoptimalizování až 44,89% hradel v původním obvodu – obvod se tedy zmenšil téměř na polovinu. Taktéž je patrné, že tabulka obsahuje mnoho záznamů, kdy se optimalizace pohybovala mezi 15%-40% a u daných obvodů tak docházelo k významnému zmenšení. Je třeba si ale uvědomit, že parametry (v tabulce neuvažované), které k těmto datům vedly, nemusí být těmi nejlepšími, a proto by jejich případná změna mohla vést k lepším výsledkům.

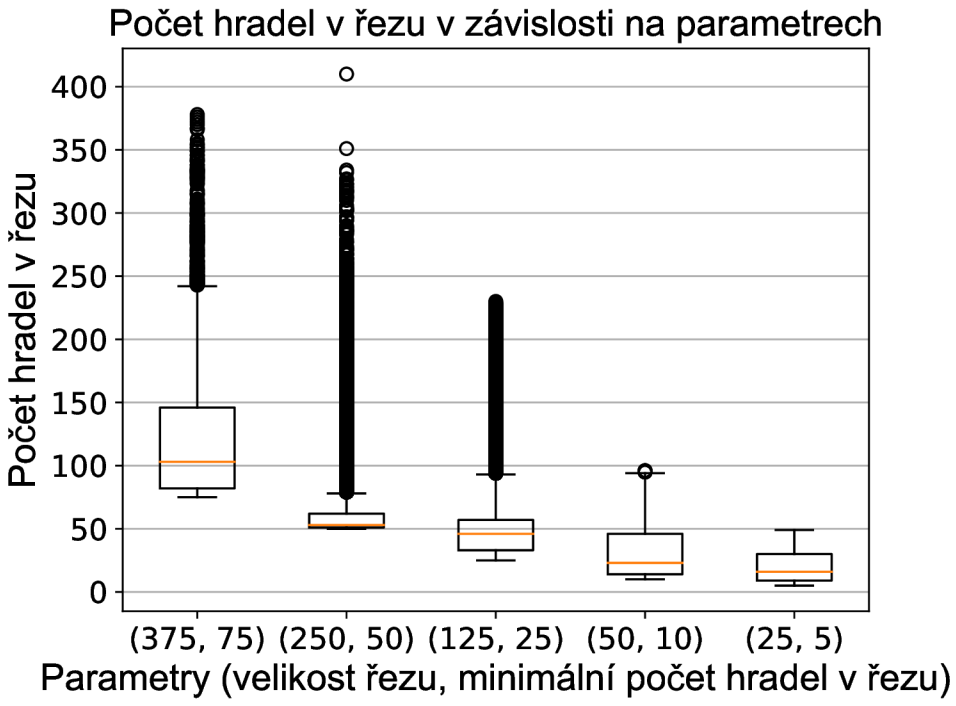
Obrázek 4.1 znázorňuje dosažené počty hradel ve vytvořených řezech v závislosti na zadaných parametrech (velikost k-feasible řezu, minimální počet hradel v řezu). Velikost řezu se může pohybovat od zadané minimální hodnoty do tisíce hradel. Jak je patrné, velikost vzniklých řezů je blíže dolní povolené hranici – více než 400 hradel v řezu bylo dosaženo pouze v jediném případě.

Obrázek 4.2 ilustruje postupnou optimalizaci obvodu během jednotlivých iterací rozšíření. Zdrojem dat pro obrázek byla optimalizace obvodu `Altera_oc_aes_core_inv` s požadovanou velikostí k-feasible řezu 50 a minimálním počtem hradel v řezu nastavenou na hodnotu deset. Na obrázku můžeme pozorovat zprvu prudký pokles počtu hradel v bvodu. S pibývajícími iteracemi již počet hradel klesá pomaleji.

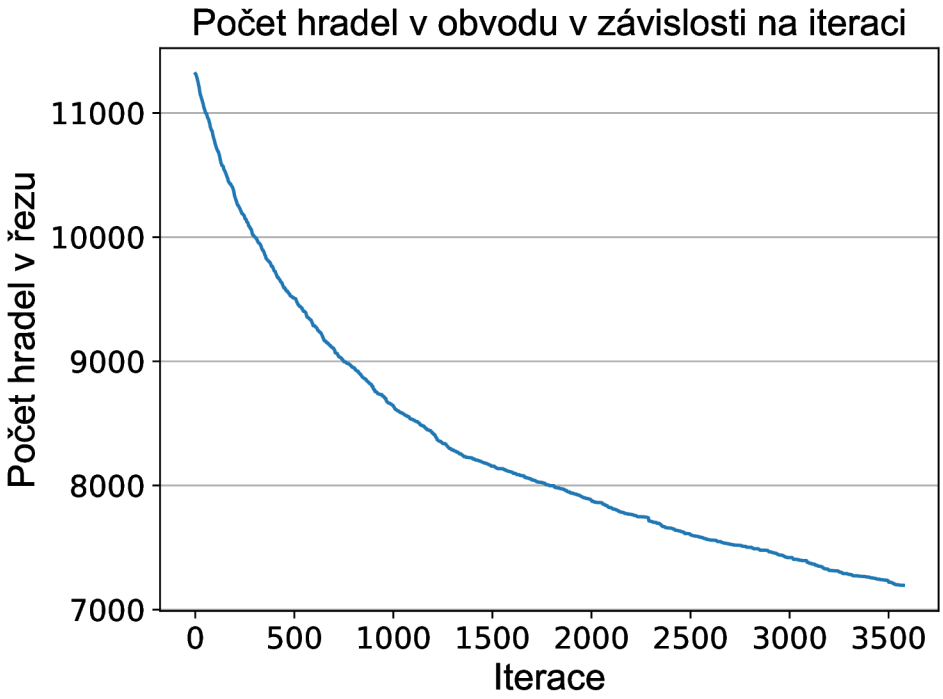
Obrázek 4.3 vyjadřuje počet hradel ve vytvořených řezech během všech iterací optimalizace. V prvních iteracích je vidět lehká tendence k menším počtům hradel v ezech. Statistické rozložení dat v orázku odpovídá trendu prezentovanému na obrázku 4.1 pro parametry (50,10).

Vstupní obvod	Hradel	Odstraněných hradel	% odstraněných
Altera_ac97_ctrl	13975	718	5,14
Altera_aes_core	20024	412	2,06
Altera_oc_aes_core	9574	3746	39,13
Altera_oc_aes_core_inv	11316	4120	36,41
Altera_cord1	9777	3183	32,56
Altera_cord2	12779	1126	8,81
Altera_ethernet	68069	558	0,82
Altera_oc_ssram	391	114	29,16
Altera_sasc	749	118	15,75
Altera_wb_dma	4172	567	13,59
Altera_spi	3939	1131	28,71
Altera_oc_mem_ctrl	18212	3167	17,39
Altera_fip_risc8	10463	2663	25,45
Altera_oc_pavr	22431	2715	12,10
Altera_oc_ata_ocidec3	4997	1470	29,42
Altera_fpu	22758	4110	18,06
Altera_mux64_16bit	4745	2130	44,89
Altera_oc_oc8051	15367	2862	18,62
Altera_tv80	9444	628	6,65

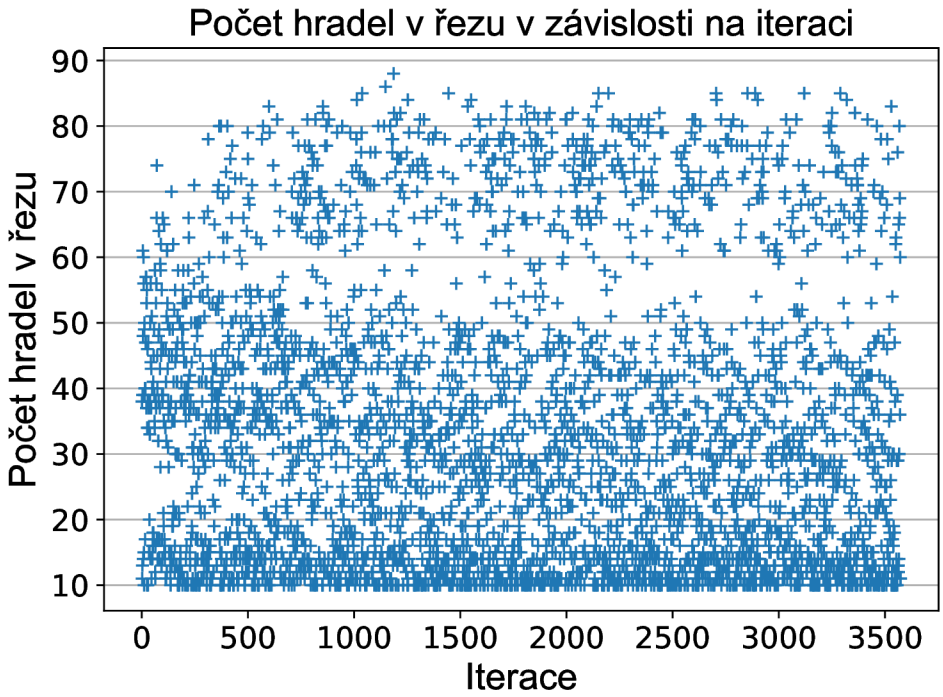
Tabulka 4.2: *Nejlepší dosažené optimalizační výsledky pro jednotlivé obvody.*



Obrázek 4.1: *Znázornění statistického rozložení hodnoty počtu hradel v řezu v závislosti na daných parametrech.*



Obrázek 4.2: *Optimalitzace obvodu během jednotlivých iterací rozšíření.*



Obrázek 4.3: *Maximální dosažený počet hradel během iterací v řezu při daných parametrech.*

Kapitola 5

Závěr

Tato práce měla za úkol vypracovat studii na téma klasické logické syntézy, evolučního programování a navrhnout a implementovat rozšíření, které by klasické syntéze poskytovalo možnost využít prvky evolučního programování.

Byl popsán tradiční přístup k logické syntéze kombinačních obvodů včetně jeho nevýhod v oblasti optimalizací a problémů se škálovatelností v závislosti na velikosti zpracovávaného obvodu. Práce také obsahuje stručný přehled nejznámějších existujících syntézních nástrojů.

Dále bylo zmíněno evoluční programování, jeho principy a možný přínos k logické syntéze. Tímto přínosem je především možnost vzniku menších, rychlejších, či originálnějších řešení, než jakých by byla schopna klasická syntéza dosáhnout. To je umožněno využitím evolučních operátorů, jakými může být například mutace či křížení, a také vybíráním toho nejlepšího vzniklého jedince v každé iteraci evolučního obvodu. Také byly popsány problémy evolučního programování, zejména pak obtížné evolvování rozsáhlého obvodu.

Součástí práce byl návrh a realizace programového rozšíření pro již existující syntézní nástroj Yosys. Rozšíření implementuje výběr části obvodu od náhodně zvoleného hradla pomocí průchodu do šířky. Vznikne tak řez o požadovaném počtu hradel. Následuje převod vytvořeného řezu na chromozom pro využívaný modul CGP. Po evoluci pak rozšíření přetváří evolvovaný chromozom zpět do existujícího obvodu a zajišťuje také napojení všech výstupů evolvovaného řezu na správné spoje v obvodu. Rozšíření dokáže vytvářet změny v obvodu tak, aby obvod jako celek zůstal validní a nezměnily se vstupy ani výstupy jeho funkce.

Bylo experimentováno s dvěmi variantami rozšíření – první tvoří řezy s více výstupy a druhá tvoří řezy s výstupem pouze z kořenového hradla řezu. V průběhu tvorby práce bylo zjištěno, že druhá varianta obvod zvětšovala, a tedy k požadované optimalizaci nedocházelo. Proto byla tato verze ponechána jen jako experimentální a dále se vývoj soustředil na variantu s více výstupy.

S vytvořeným rozšířením byla provedena řada experimentů a to na devatenácti různých kombinačních obvodech ze sady QUIP[17] s různou velikostí a strukturou. Dosažené výsledky do jisté míry závisí i na struktuře optimalizovaných obvodů, a také na zvolených parametrech. Z xperimentů plyne, že při vhodně zvolených parametrech bylo pro většinu optimalizovaných obvodů implementované rozšíření schopno dosáhnout redukce počtu hradel v rozmezí 20 %-35 %. Nejlepším získaným výsledkem byla redukce počtu hradel v obvodu o 44, %.

Možnost dalšího vývoje implementovaného rozšíření spočívá například v tvorbě KL-feasible řezů či vytváření více variant řezu pro dané hradlo a jejich prioritizaci podle zvoleného kritéria.

Literatura

[1] *Vivado Design Suite User Guide*. [Online; navštíveno 3.1.2017].
URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf

[2] *Welcome to the Quartus II Software*. [Online; navštíveno 4.1.2017].
URL http://quartushelp.altera.com/15.0/mergedProjects/quartus/gl_quartus_welcome.htm

[3] Gajda, Z.; Sekanina, L.: *Reducing the Number of Transistors in Digital Circuits Using Gate-Level Evolutionary Design Circuits Using Cartesian Genetic Programming*. [Online; navštíveno 28.12.2016].
URL http://www.fit.vutbr.cz/~sekanina/publ/gecco07/gajda-sekanina.pdf

[4] Jain, T. K.; Kushwaha, D. S.; Misra, A. K.: Optimization of the Quine-McCluskey Method for the Minimization of the Boolean Expressions. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*, March 2008, ISSN 2168-1864, s. 165–168.

[5] Jiang, J.-H. R.; Devadas, S.: *Logic Synthesis in a Nutshell*. [Online; navštíveno 20.12.2016].
URL http://flolac.iis.sinica.edu.tw/flolac09/lib/exe/logic-synthesis.pdf

[6] Kalganova, T.; Miller, J.: Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, 1999, s. 54–63.

[7] Martinello, O.; Marques, F. S.; Ribas, R. P.; aj.: KL-Cuts: A new approach for logic synthesis targeting multiple output blocks. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, ISSN 1530-1591, s. 777–782.

[8] Mishchenko, A.; Cho, S.; Chatterjee, S.; aj.: Combinational and sequential mapping with priority cuts. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, Nov 2007, ISSN 1092-3152, s. 354–361.

[9] Sekanina, L.: Evolutionary Design of Digital Circuits: Where Are Current Limits? In *First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06)*, June 2006, s. 171–178.

[10] Soeken, M.: *CirKit Documentation*. [Online; navštíveno 10.1.2017].
URL http://msoeken.github.io/cirkit_doc.html

[11] Soeken, M.: *Qflow 1.0: An Open-Source Digital Synthesis Flow*. [Online; navštíveno 10.1.2017].
URL http://opencircuitdesign.com/qflow/

[12] Soliman, A. T.; Abbas, H. M.: Combinational circuit design using evolutionary algorithms. In *CCECE 2003 - Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No.03CH37436)*, May 2003, ISSN 0840-7789, s. 251–254 vol.1.

[13] Stoica, A.; Keymeulen, D.; Zebulum, R. S.; aj.: *Scalability Issues in Evolutionary Synthesis of Electronic Circuits: Lessons Learned and Challenges Ahead* . [Online; navštíveno 10.12.2016].
URL https://pdfs.semanticscholar.org/27ae/c607dc2201fcdc7f41f07cc9a6f7321b169b.pdf

[14] The Regents of the University of California: *ABC: A System for Sequential Synthesis and Verification*. [Online; navštíveno 14.1.2017].
URL https://people.eecs.berkeley.edu/~alanmi/abc/

[15] The Regents of the University of California: *MVSIS: Logic Synthesis and Verification*. [Online; navštíveno 15.1.2017].
URL https://embedded.eecs.berkeley.edu/Respep/Research/mvsis/index.html

[16] The Regents of the University of California: *SIS*. [Online; navštíveno 15.1.2017].
URL https://embedded.eecs.berkeley.edu/Respep/Research/sis/abstract.html

[17] Vašíček, Z.: Cartesian GP in Optimization of Combinational Circuits with Hundreds of Inputs and Thousands of Gates. In *Genetic Programming, 18th European Conference, EuroGP 2015*, LCNS 9025, Springer International Publishing, 2015, ISBN 978-3-319-16500-4, s. 139–150.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=10773

[18] Vašíček, Z.; Sekanina, L.: *On Area Minimization of Complex Combinational Circuits Using Cartesian Genetic Programming*. [Online; navštíveno 26.12.2016].
URL http://www.fit.vutbr.cz/research/pubs/index.php?file=%2Fpub%2F9866%2Fcec_ls.pdf&id=9866

[19] Vašíček, Z.; Sekanina, L.: A global postsynthesis optimization method for combinational circuits. In *2011 Design, Automation Test in Europe*, March 2011, ISSN 1530-1591, s. 1–4.

[20] Wolf, C.: *Yosys Manual*. [Online; navštíveno 6.1.2017].
URL http://www.clifford.at/yosys/files/yosys_manual.pdf

[21] Zdeněk Vašíček: *Cartesian Genetic Programming*. [Online; navštíveno 2.3.2017].
URL http://www.fit.vutbr.cz/~vasicek/cgp/

Přílohy

Příloha A

Obsah CD

Obsah přiloženého CD:

- Technická zpráva ve formátu PDF v adresáři `/thesis/`,
- Zdrojové kódy technické zprávy ve formátu LaTeXv adresáři `/thesis-latex/`,
- Zdrojové kódy aplikace v adresáři `/src/`,
  - Zdrojové kódy Yosysu s implementovaným rozšířením `/yosys-code/`,
  - Testovací skript a soubory v adresáři `/eval/`,
  - Soubory s příklady obvodů v adresáři `/eval/benchmarks/`,
  - Soubory s několika příklady výsledků experimentů v adresáři `/eval/results/`,
- Soubor `readme.txt` s návodem na použití aplikace.

Příloha B

Manuál

Nejprve je zapotřebí nainstalovat program Yosys zahrnující vytvořené rozšíření. Postup je stejný pro verzi s více výstupy i s jedním výstupem(ta je poze experimentální); následující popis bude tedy založen na variantě s více výstupy.

Překlad Yosysu:

```
make config-gcc
make
```

Existují dvě možnosti, jak Yosys s rozšířením spustit. První variantou je spuštění rozšíření přímo v Yosysu:

- `./yosys` – příkaz ke spuštění programu Yosys,
- `read_blif passes/cmds/cgp-sat/benchmarks/vybraný_soubor_s_popisem_obvodu.blif` – příkaz k načtení obvodu ze zvoleného souboru,
- `jittExtension -iterations 1 -cutSize 20 -minCellCount 7 -maxCellCount 200 -maxCGPRuntime 1 -maxCGPGenerations 100000 -multiout 1` – příklad příkazu k vykonání kódu rozšíření.

V adresáři `/src/yosys-singleoutput/passes/cmds/cgp-sat/bechmarks/` jsou k nalezení tři obvody popsané v souborech typu `blif`, které je možné využít pro testování.

Druhou možností je spuštění přiloženého skriptu `/src/eval/sge.py`, který již má přednastavené hodnoty parametrů a otestuje rozšíření na všech příkladech obvodů v adresáři `/src/eval/benchmarks/` s využitím verifikace programu ABC. Do tohoto skriptu je pouze zapotřebí nastavit vlastní cestu k adresáři `/src/eval/`.

Činnost rozšíření je u všech možností spuštění stejná. Nejprve je načten vstupní soubor a je aplikováno rozšíření. Výstupem rozšíření je změněný vstupní obvod, který se nachází v souboru `out.blif`.