VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

VYUŽITÍ SAT SOLVERŮ V ÚLOZE OPTIMALIZACE KOMBINAČNÍCH OBVODŮ

APPLICATION OF SAT SOLVERS IN CIRCUIT OPTIMIZATION PROBLEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. VOJTĚCH MINAŘÍK**
AUTHOR

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

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Minařík Vojtěch, Bc.**
Program: Informační technologie Obor: Bioinformatika a biocomputing
Název: **Využití SAT solverů v úloze optimalizace kombinačních obvodů**
Application of SAT Solvers in Circuit Optimization Problem
Kategorie: Umělá inteligence
Zadání:

1. Seznamte se s problematikou návrhu a optimalizace kombinačních obvodů pomocí evolučních technik využívajících SAT solver. Seznamte se s principy moderních SAT a #SAT solverů, dostupnými řešeními a možnostmi podpory inkrementálního režimu.
2. Navrhněte programový systém, který bude umožňovat optimalizaci kombinačních obvodů s využitím inkrementálního SAT případně #SAT solveru.
3. Zpracujte studii na výše uvedené téma.
4. Navržený systém implementujte s ohledem na maximální výkonnost, tzn. snažte se maximalizovat počet ohodnocených kandidátních obvodů za jednotku času.
5. Pomocí sady benchmarkových obvodů experimentálně vyhodnoťte parametry navrženého řešení. Zaměřte se na počet ohodnocených kandidátních řešení za jednotku času a velikost obvodů po optimalizaci.
6. Diskutujte dosažené výsledky a možnosti dalšího pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Vašíček Zdeněk, doc. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 26. října 2018

Abstrakt

Tato práce zavádí využití řešení problému SAT a jeho modifikací v úloze evolučního návrhu kombinačních obvodů. Motivací využití těchto problémů je zrychlení ohodnocování chromozomů kandidátních řešení fitness funkcí během evoluce v případech, kdy selhává metoda klasické simulace. Využití problému SAT, respektive #SAT umožňuje oproti simulaci zrychlení zejména pro komplikované obvody s velkým počtem vstupů. Implementované řešení se zalkádá právě na problému #SAT. Celkem byly implemenyovány dvě různé varianty využití tohoto problému. Varianty se liší metodou kontorly rozdílných hodnot na výstupech obvodu. Protože implementované řešení využívá k reprezentaci obvodu logickou formuli a zkoumá její splnitelnost, časová složitost algoritmu závisí především na logické složitosti navrhovaného obvodu.

Abstract

This thesis is focused on the task of application of SAT problem and it's modifications in area of evolution logic circuit development. This task is supposed to increase speed of evaluating candidate circuits by fitness function in cases where simulation usage fails. Usage of SAT and #SAT problems make evolution of complex circuits with high input number significantly faster. Implemented solution is based on #SAT problem. Two applications were implemented. They differ by the approach to checking outputs of circuit for wrong values. Time complexity of implemented algorithm depends on logical complexity of circuit, because it uses logical formulas and it's satisfiability to evaluate logic circuits.

Klíčová slova

Evoluce, evoluční návrh hardwaru, hardware, HW, SAT, splnitelnost, kombinační obvod, návrh kombinačních obvodů, genetické programování, logika, umělá inteligence, AI.

Keywords

Evolution, Evolution HW design, HW, SAT, satisfiability, logic circuit, logic, genetic programming, artificial inteligence, AI.

Citace

MINAŘÍK, Vojtěch. *Využití SAT solverů v úloze optimalizace kombinačních obvodů.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vašíček Zdeněk, Ph.D.

Využití SAT solverů v úloze optimalizace kombinačních obvodů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doc. Ing. Zdeňka Vašíčka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vojtěch Minařík
21. května 2019

Poděkování

Chtěl bych poděkovat vedoucímu mé diplomové práce panu doc. Ing. Zdeňku Vašíčkovi, Ph.D. za jeho odbornou pomoc, cenné rady a poskytnutí testovacích prostředků a dat.

Obsah

Kapitola 1

Úvod

Metodiky řešení problémů založených na splnitelnosti formulí (problém SAT) nabízejí efektivní možnost řešit mnoho problémů, které by jinak byly velmi náročné. Obsah této práce se zaměřuje na konkrétní využití řešení problémů SAT a SAT při evolučním návrhu kombinačních obvodů jako alternativu k použití simulace při ohodnocování jedinců populace. Simulace se při nasazení na velmi komplikované obvody s velkým počtem vstupnů stává neefektivní. Tuto situaci pomáhají zlepšit řešení založená na převodu kombinačních obvodů na logické formule a následné vyšetřování jejich splnitelnosti. Evoluční návrh hardwaru umožňuje navrhovat nová řešení, aniž by návrhář musel předem znát veškeré informace o vnitřní struktuře požadovaného produktu. Kromě toho mohou evoluční algoritmy nalézt nová neobvyklá řešení mnohdy dosahující znatelně lepších výsledků než řešení tradiční. K tomu využívá přírodních principů aplikovaných na svět techniky.

Text této práce čtenáře nejdříve seznámí s teoretickými znalostmi důležitými ke správnému porozumění pozdějších kapitol zaměřených podrobně na konkrétní problematiku a využití teoretických znalostí v praxi.

Nejprve si představíme v kapitole 2 splnitelnost jako matematickou disciplínu a její vývoj od jednoduchých problémů až po řešení moderních komplikovaných problémů ve světě informačních technologií. V tématu matematický znalostí pokračuje kapitola 3 popisem základních pojmů a principů výrokové logiky, na které se celý problém řešení splnitelnosti zakládá.

Praktické nasazení znalostí z předchozích kapitol nalezneme v oblasti řešení problému SAT. Jeho popisem, principy, které používá, algoritmy jeho řešení a dalšími problémy se základem v tomto problému se zabývá kapitola 4. V této kapitole také nalezneme popis využití problému SAT a jeho variant při evolučním návrhu hardwaru.

Evoluční návrh hardwaru nezbytně potřebuje prostředky schopné ohodnotit chromozomy jedinců populace a zároveň tvořit nové generace. Způsobům ohodnocování jedinců a metodám jejich tvorby (zmiňme v této práci použité kartézské genetické programování) je věnována kapitola 6.

V rámci této práce byli implementovány tři varianty evolučního algoritmu s využitím kartézského genetického programování. Jednotlivá řešení se liší především způsobem ohodnocování kandidátních řešení. Zatímco jedno využívá klasickou simulaci a bylo implementováno především pro účely srovnání s ostatními, další dvě řešení využívají k ohodnocení populace algoritmus založený na řešení problému SAT. Jejich popisem a rozborem principů, které používají se zabývá kapitola 7.

Dosažené výsledky, výsledky experimentů, srovnání efektivity jednotlivých řešení a jejich metriky nalezneme v kapitole číslo 8.

Kapitola 2

Splnitelnost

Splnitelnost označuje vlastnosti formule vyjadřující skutečnost ,zda existuje alespoň jedno ohodnocení všech proměnných tvořících zkoumanou formuli takové, že vyhodnocení celé formule je kladné. Existuje několik matematických problémů zabývajících se splnitelností logických formulí jako například: SAT, SAT, maxSAT... V kontextu této práce se budeme zabývat především problémem booleovské splnitelnosti. Ta řeší otázku, zda je formule složená z proměnných nabývajících pravdivostních hodnot 0 a 1, logických operací, jejichž výsledkem jsou opět pravdivostní hodnoty.

Tato disciplína by neměla být zanedbávána, protože mnoho problémů lze nejrychleji řešit pomocí SAT solverů [2]. Navíc se splnitelnost formulí nachází na pomezí mezi mnoha obory, například: logika, teorie grafů, informační technologie ... Proto se zde nabízí široká škála problémů řešitelných touto metodou. Pokud je logická formule splnitelná, ohodnocení proměnných účastnících se na tomto řešení, nazýváme „modelem formule".

Historie zkoumání splnitelnosti pokrývá poslední dvě tisíciletí. Problém splnitelnosti byl známý již v době života Aristotela [2]. Během té doby se splnitelnost zvládla vyvinout z prostředků popisování základních vlastností nějakých objektů na dnešní složité konstrukce popisující funkcionalitu počítačových programů nebo samotných hardwarových komponent.

2.1 Historie splnitelnosti a matematické logiky

Matematická logika se poprvé objevila v Athénách v dobách antického Řecka v díle s názvem Organon, jehož autorem byl Aristoteles (stalo se tak mezi lety 384-322 před naším letopočtem) [2]. Na jeho práci navázali další učenci ve středověku, kteří vyvinuli mnohem pokročilejší systém matematické logiky. „Ve středověku znali prostředky jako kategorické výroky, omezující i neomezující relativní klauzule a skládali výrokové spojitosti do složitých vět. Navíc v té době neznali teorii množin, ale popisovali interpretace predikátů jako skupiny nebo kolekce„ (přejato z [2]). Cílem symbolické logiky bylo dokázat, že nějaká formule je teorií na sadě axiomů nebo že formuli lze pomocí daných axiomů odvodit.

Na znalosti z předchozích období se postupně nabalovaly další a další nové poznatky až došlo k vytvoření pro dnešní svět zajímavých objevů jako je Booleva algebra. Tu vytvořil George Bool v první polovině 19. století.

Za zmínku jistě stojí práce matematika Kurta Göedela, který ve 30. letech dvacátého století publikoval svůj teorém neúplnosti. Dokázal, že systém axiomů „Principia" a každý jiný systém axiomů schopný formulovat zákony aritmetiky, je a musí být nekompletní (neúplný) v tom smyslu, že vždy bude existovat nějaká pravdivá hodnota aritmetiky, která

nebude teorií daného systému axiomů [2][9]. Jinými slovy dokázal nepravdivost ideje, že celý obor matematiky jako takový lze odvodit z matematické logiky. Toto ovšem platí pouze pro matematiku jako celek. Samozřejmě existují její části, které lze korektně a úplně popsat pomocí logických axiomů.

Dalším vývojem došlo k objevení a prozkoumání SAT problému, který je relevantní pro tuto práci. Problém SAT byl prvním problémem, pro který se podařilo dokázat nejen NP těžkost, ale rovněž NP úplnost.

Kapitola 3

Výroková logika

Výroková logika je často charakterizována jako logika zkoumající logické vztahy mezi výroky [11]. Výrokem rozumíme větu (ustanovení), které může být označeno buď jako pravdivé nebo nepravdivé. Analogii výrokové logiky v matematice můžeme nalézt v přirozeném jazyce, kde výroky jsou věty a funkci logiky zastávají spojky mezi těmito větami, například: a, nebo ...

Základní logický výrok v přirozeném jazyce můžeme popsat následujícím triviálním příkladem. Mějme 2 výroky:

1. pokud mrzne, pak je zima (roční období),
2. mrzne,

pak můžeme s použitím výrokové logiky dojít k následujícímu závěru:

- je zima.

Pro zobecnění lze jednotlivé výroky nahradit písmeny proměnných reprezentujících pravdivost daného výroku.

Výsledky logických operací nad výroky lze přehledně zapsat do takzvané pravdivostní tabulky. Ukázku pravdivostních tabulek pro základní logické operace najedeme v následují skupině tabulek:

A	B	A$\wedge B$
1	1	1
1	0	0
0	1	0
0	0	0

A	B	A$\vee B$
1	1	1
1	0	1
0	1	1
0	0	0

A	B	A $\Longrightarrow$ B
1	1	1
1	0	0
0	1	1
0	0	1

Tabulka 3.1: Pravdivostní tabulky základních logických operací.

Výroky jsou zde reprezentovány písmeny a mohou nabývat pouze dvou hodnot - logická 1 a 0. Hodnota 1 značí, že je výrok pravdivý a 0 značí případ, kdy je výrok nepravdivý. To platí též pro výsledek logické operace.

3.1 Tautologie a kontradikce

Zde se vyskytují dva speciální případy:

1. logická formule (věta) je vždy pravdivá – pak se jedná o tautologii
2. logická formule (věta) není nikdy pravdivá – pak použijeme název kontradikce.

Následující pravdivostní tabulky ukazují primitivní příklady tautologie a kontradikce:

A	B	$(A \implies B) \vee (B \implies A)$
1	1	1
1	0	1
0	1	1
0	0	1

A	B	$B \vee \neg(A \implies B)$
1	1	0
1	0	0
0	1	0
0	0	0

Tabulka 3.2: Jednoduché příklady tautologie a kontradikce.

3.2 Dvojí negace a de-Morganovy zákony

Ve výrokové logice platí zákon dvojí negace, který říká, že pokud dvakrát znegujeme nějaký výrok, jako výsledek dostaneme opět výrok původní:

$$\overline{\overline{X}} = X \tag{3.1}$$

De-Morganovy zákony:

$$\overline{(A \wedge B)} = \overline{A} \vee \overline{B} \tag{3.2}$$

$$\overline{(A \vee B)} = \overline{A} \wedge \overline{B} \tag{3.3}$$

De-Morganovi zákony se využívají při modifikaci logických formulí.

3.3 Atomický výrok

Nechť A B je složený výrok skládající se z výroků A a B spojených operací konjunkce. Pak pokud výroky A a B již neobsahují žádné dílčí výroky, označíme je jako výroky atomické.

3.4 Formule

Pro složené výroky se běžně používá název formule. Formulí může ovšem být nazván i atomický výrok. Formule lze spojovat pomocí logických spojek do složitějších formulí. Proto musí platit:

– pokud jsou A i B formule

– • je logická operace,

pak:

– (A • B) je rovněž formule.

Kapitola 4

Problém SAT

Problém řešící splnitelnost formulí nazýváme problém SAT. Tento problém lze popsat vyjádřením: vstupní formule je v konjunktní normální formě a zároveň je splnitelná. Jinými slovy existuje alespoň jedno ohodnocení všech proměnných tvořících formuli takové, že vyhodnocení celé formule odpovídá kladné logické hodnotě. Řešení tohoto problému hraje důležitou roli v mnoha oblastech od teoretické informatiky přes umělou inteligenci až k návrhu fyzických logických obvodů.

Problém SAT patří do třídy NP těžkých jazyků, tedy lze sestrojit nedeterministický Turingův stroj, který problém rozhodne v polynomiálním čase. Problém není pouze NP těžký, ale dokonce NP úplný. NP úplnost SAT problému, dokázal v roce 1973 Stephen Cook. Tím se stal SAT problém prvním člověku známým NP úplným problémem. Tento důkaz dostal jméno po jeho tvůrci, tedy Cookův teorém, někdy též zvaný Cook-Levinova teorie. Nicméně řešit velmi rozsáhlé problémy obsahující až několik desítek tisíc proměnných, umožnil až dramatický nárůst výkonu výpočetní techniky a návrh efektivních algoritmů řešících tento problém v posledních dvou desetiletích.

To učinilo tento problém využitelným při automatizovaném vývoji hardwaru, formální verifikaci atd.

Jako vstup problému SAT lze označit jakoukoliv logickou formuli v konjunktní normální formě. Pokud formule není již v tomto tvaru je třeba ji do konjunktní normální formy převést některým z mnoha algoritmů.

4.1 Konjunktní normální forma

Předtím než lze zkoumat splnitelnost obvodů je třeba jejich reprezentaci logickými formulemi převést do konjunktní normální formy. Tato forma využívá konjunkci klauzulí ici, kde každá klauzule ci má podobu disjunkce j jednotlivých prvků xj, které jsou buď proměnné nabývajících logických pravdivostních hodnot (1 nebo 0) anebo negace těchto proměnných xj. Každá formule v konjunktní normální formě má tedy následující tvar:

$$(x_0 \vee ... \vee x_k) \wedge ... \wedge (x_m \vee ... \vee x_n) \tag{4.1}$$

Pro každou logickou formuli F lze vytvořit ekvivalentní formuli F‘ v konjunktní normální formě. Aby byli formule ekvivalentní musí platit, že F‘ je splnitelná právě tehdy, když je splnitelná formule F‘.

Tato kapitola se dále zabývá převodem formulí do konjunktní normální formy. Převod je nutný, protože původní podoba formulí reprezentující logické obvody není vhodná k řešení

jejich splnitelnosti. Převod do konjunktní normální formy umožňuje použití SAT solverů, které jako svůj vstup požadují, jak vyplývá z toho, že řeší problém SAT, právě formule v konjunktím normálním tvaru.

4.2 Převod do CNF

V této podkapitole již víme, jak CNF vypadá a proč je převod do ní nezbytně nutný. Proto se můžeme zaměřit na vlastní převod formulí odpovídajícím logickým kombinačním obvodům sestavených z logických hradel na ekvivalentní formule v konjunktí normální formě.

4.2.1 Převod s využitím Booleovy algebry

Na logicky ekvivalentní formuli v podobě CNF lze převést formuli aplikací pravidel, které poskytuje Booleova algebra. Tato pravidla se soustředí na tři základní pravidla nad logickými hodnotami, konjunkce $\wedge$, disjunkce $\vee$ a negace $\neg$. Všechna pravidla (zákony) znázorňuje následující tabulka:

Zákon	Disjunkce	Konjunkce
Komutativní	$A \vee B = B \vee A$	$A \wedge B = B \wedge A$
Asociativní	$A \vee (B \vee C) = (B \vee A) \vee C$	$A \wedge (B \wedge C) = (B \wedge A) \wedge C$
Distributivní	$(A \vee B) \wedge (A \vee C) = A \vee AB$	$(A \wedge B) \vee (A \wedge C) = A \wedge (A \vee B)$
Komplementarita	$A \vee \neg A = 1$	$A \wedge \neg A = 0$
Agresivita 0 a 1	$A \vee 1 = 1$	$A \wedge 0 = 0$
Neutrálnost 0 a 1	$A \vee 0 = A$	$A \wedge 1 = A$
Absorbce	$A \vee A = A$ $A \vee (A \wedge B) = A$	$A \wedge A = A$ $A \wedge (A \vee B) = A$
Absorbce negace	$A \vee (\neg A \wedge B) = A \vee B$ $\neg A \vee (A \wedge B) = \neg A \vee B$	$A \wedge (\neg A \vee B) = A \wedge B$ $\neg A \wedge (A \vee B) = \neg A \wedge B$
Dvojité negace	$\neg\neg A = A$	

Tabulka 4.1: Zákony platné v rámci Booleovy algebry.

Dále v rámci Booleovy algebry platí de Morganovy zákony. Ty byly popsány v předchozí kapitole.

Nevýhodou této metody může být výrazný nárůst délky, v nejhorším případě exponenciálně vzhledem k velikosti původní formule, výsledné formule v CNF. Na druhou stranu negeneruje žádné nové proměnné.

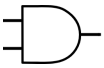
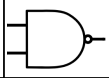
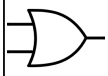
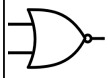
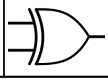
4.2.2 Transformace pomocí Tseitinova kódování

Tato metoda vytvoří výslednou formuli v CNF s využitím specifického kódování jednotlivých logických operací. Tseitinovo kódování obsahuje překlad všech operací, které se nemohou vyskytovat ve formuli v konjunktní normální formě, na části výsledné formule v konjunktní normální formě. Tyto části formule musí být splnitelné tehdy, když a jen tehdy, když je splnitelná také původní část formule. Tento přístup však vyžaduje generování nových proměnných reprezentujících původní části formule. V případě použití pro převod logického obvodu do CNF je třeba vytvořit novou proměnnou pro každé hradlo nacházející se v obvodu.

Výsledná formule dosahuje velikostí lineárně závislých na velikosti původní formule. Splnitelnost výsledné formule v konjunktní normální formě se shoduje se splnitelností formule původní. Tedy pro formuli F a formuli F' vzniklou převodem formule F do CNF platí:

$$F \iff F' \tag{4.2}$$

Příklad Tseitinova kódování:

Název	Hradlo	Funkce	Ekvivalentní formule v CNF
not A		C = ¬A	(A ∨ C) ∧ (¬A ∨ ¬C)
A		C = A	(¬A ∨ C) ∧ (A ∨ ¬C)
AND		C = A & B	(¬A ∨ ¬B ∨ C) ∧ (A ∨ ¬C) ∧ (B ∨ ¬C)
NAND		C = ¬(A & B)	(¬A ∨ ¬B ∨ ¬C) ∧ (A ∨ C) ∧ (B ∨ C)
OR		C = A | B	(A ∨ B ∨ ¬C) ∧ (¬A ∨ C) ∧ (¬B ∨ C)
NOR		C = ¬(A | B)	(A ∨ B ∨ C) ∧ (¬A ∨ ¬C) ∧ (¬B ∨ ¬C)
XOR		C = (A ^ B)	(¬A ∨ ¬B ∨ ¬C) ∧ (A ∨ B ∨ ¬C) ∧ (A ∨ ¬B ∨ C) ∧ (¬A ∨ B ∨ C)

Obrázek 4.1: Výběr Tseitinovy transformace základních typů hradel.

4.3 Rozšíření SAT problému

Postupem času začali vznikat další speciální verze SAT problému. Tyto varianty se odlišují například specifickou strukturou formule v konjunktní normální formě nebo jiným přístupem ke splnitelnosti celé formule. Druhá možnost obsahuje algoritmy hledající všechna pravdivá ohodnocení formule, nalezení Hammingovy vzdálenosti pravdivých a nepravdivých ohodnocení proměnných formule nebo splnitelnost pouze části formule.

Za nejvýznamnějšího člena skupiny vyznačující se specifickou strukturou formule můžeme považovat k-SAT problém. Tento problém zkoumá splnitelnost formulí v konjunktní normální formě s uniformní délkou všech klauzulí. Uniformní délce klauzulí konjunktní normální formy odpovídá hodnota k v názvu problému. Nejznámějším zástupcem této skupiny variant SAT problému je problém 3-SAT. Tento problém se zakládá na formulích v konjunktní normální formě, jejich klauzule obsahují vždy právě 3 proměnné. Taková formule může vypadat následovně:

$$(x_1 \vee x_2 \vee x_3) \wedge ... \wedge (x_4 \vee x_2 \vee x_5) \tag{4.3}$$

Dalším významným rozšířením problému SAT je SAT (sharp-SAT). Tento problém neřeší pouze splnitelnost formule, ale jako hlavní cíl má spočíst počet všech kladných ohodnocení formule.

Mezi rozšíření problému SAT hledající splnitelnost alespoň části formule, nikoliv pouze celé formule, patří zejména problém maximální splnitelnosti značený MAX-SAT. Problém hledá maximální možný počet klauzulí, které mohou být splněny při jakémkoliv ohodnocení proměnných formule.

4.4 Algoritmy řešení problému SAT

Protože je problém SAT NP-úplný problém, není známý žádný algoritmus jeho řešení, který by dosahoval lepší než exponenciální časové složitosti výpočtu. Algoritmy používané při řešení SAT problému používají různé přístupy jako jsou DPLL („Davis–Putnam–Logemann––Loveland“), CDCL („Conflict Driven Clause Learning“).

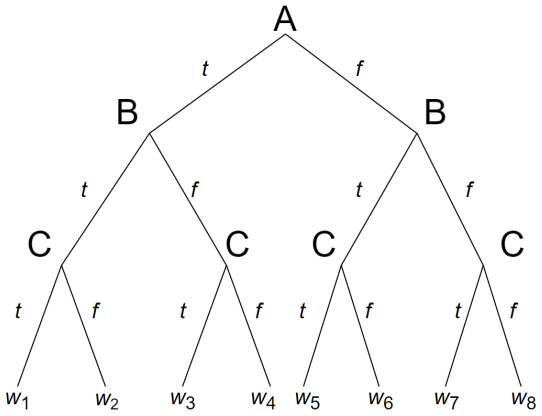
4.4.1 DPLL

Algoritmus prohledává prostor ohodnocení proměnných do hloubky s využitím backtrackingu a propagace hodnot proměnných a jednotkových klauzulí (jednotková klauzule je taková klauzule, kde se nachází pouze jedna dosud neohodnocená proměnná, která však rozhoduje o splnitelnosti celé klauzule). Nejprve vybere jednu proměnnou, které přidělí pravdivé ohodnocení, následuje zjednodušení formule a nakonec se rekurzivně ověřuje splnitelnost zjednodušené formule. Pokud toto ohodnocení selže pro všechna ohodnocení ostatních proměnných, formule není splnitelná. V tomto případě se postup opakuje s původní proměnnou nastavenou na opačnou pravdivostní hodnotu.

Obrázek 4.2 znázorňuje vyhledávací strom ekvivalentní k prohledávanému prostoru ohodnocení proměnných pro formuli obsahující tři proměnné X, Y a Z. Listy tohoto stromu odpovídají v poměru jedna ku jedné pravdivostním ohodnocení všech proměnných formule [2]. To vyplývá ze samotné struktury vyhledávacího stromu, která má podobu binárního stromu, kde každý uzel má dva následovníky rozlišené hodnotou proměnné (pravdivé a nepravdivá) odpovídající dané úrovni stromu a listy jsou celkové vyhodnocení formule. Z toho lze odvodit výšku rozhodovacího stromu h jako vztahu:

$$h = n + 1 \tag{4.4}$$

kde n označuje počet proměnných přítomných ve formuli. Pro tři proměnné bude tento binární strom vypadat následovně.



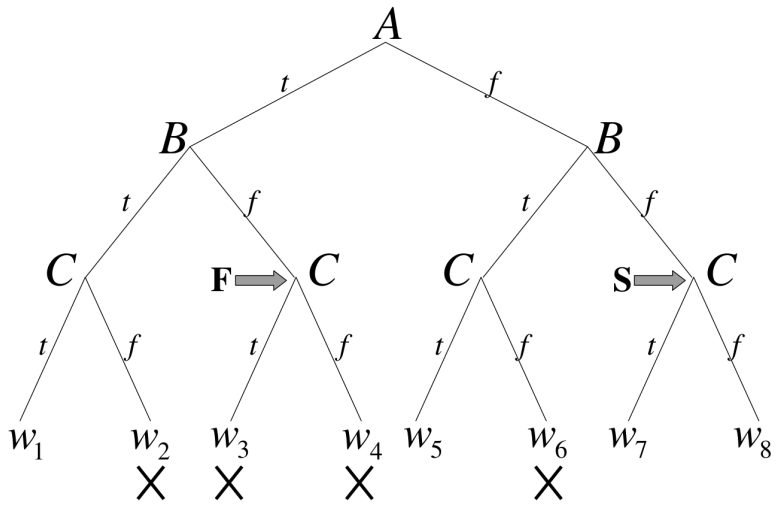
Obrázek 4.2: Vyhledávací strom určený ke hledání všech formuli splňujících přiřazení hodnot proměnným A,B a C [2].

Použití algoritmu si nyní předvedeme na příkladu převzatém z knihy Handbook of Satisfiability [2]. Nechť f je formule v CNF obsahující klauzule $(\overline{A}, B)$ a $(\overline{B}, C)$. podoba CNF bude tedy:

$$f = (\overline{A} \vee B) \wedge (\overline{B} \vee C) \tag{4.5}$$

Dále uzel označený písmenem F na obrázku 4.3 odpovídá takovému ohodnocení formule, že formule není splnitelná. Ohodnocení proměnných bude mít následující podobu:

$$f(A, \overline{B}) = (false \vee false) \wedge (true \vee C) = \{\{\}\} \tag{4.6}$$



Obrázek 4.3: Vyhledávací strom určený ke hledání všech formuli splňujících přiřazení hodnot proměnným A,B a C. Přiřazení označená písmenem X nejsou modelem formule [2].

Jelikož listy *w3* a *w4* nemohou vést ke kladnému vyhodnocení formule *f*, nemohou tyto listy být modelem formule f [2].

Schopnost detekovat sporná ohodnocení v uzlech, jenž nejsou listy stromu, umožňuje eliminovat všechna kladná ohodnocení, která jsou popsána daným uzlem, aniž by muselo dojít k jejich propočítávání. Tento algoritmus také dokáže detekovat úspěšné řešení na uzlu (ne listu) stromu, jehož všechna kladné ohodnocení specifikovaná tímto uzlem jsou modely formule f v konjunktní normální formě. Pokud vezmeme v úvahu uzel S na obrázku 4.3, zjistíme, že tento uzel reprezentuje kladná ohodnocení *w7* a *w8*. Tento uzel odpovídá ohodnocení $A = false$, $B = false$, tedy:

$$f(\overline{A}, \overline{B}) = \{\{true, false\}, \{true, C\}\} \tag{4.7}$$

Ze vztahu je zřejmé, že jakékoliv ohodnocení (proměnná C může nabývat jak kladné tak negativní hodnoty) následujícího uzlu S bude vždy splnitelné a tedy modelem formule f [2].

Algoritmus DPLL lze popsat pseudokódem na obrázku 4.4.

```
if Δ = {} then
   return {}
else if {} ∈ Δ then
   return UNSATISFIABLE
else if L =DPLL-(Δ|P_{d+1}, d + 1) ≠ UNSATISFIABLE then
   return L ∪ {P_{d+1}}
else if L =DPLL-(Δ|¬P_{d+1}, d + 1) ≠ UNSATISFIABLE then
   return L ∪ {¬P_{d+1}}
else
   return UNSATISFIABLE
```

Obrázek 4.4: Pseudokód algoritmu DPLL. Vrací buď množinu literálů nebo UNSATISFIABLE, když je formule nesplnitelná. V kontextu předchozího textu je zde formule f označena písmenem Δ. Proměnné jsou značeny P_1 a P_2 [2].

Tento algoritmus nabývá exponenciální časové složitosti $O(2^n)$, kde n značí počet proměnných tvořící formuli v CNF. Nicméně, jak již plyne z použití notace O, jedná se o nejhorší možný případ. Množství vykonané práce pro konkrétně daný případ lze vypočíst například pomocí terminačních stromů.

Na tomto algoritmu jsou založeny implementace mnoha SAT solverů, které se pravidelně umisťují na předních příčkách soutěží v rychlosti řešení SAT problému, například Chaff a Minisat.

4.4.2 Metoda zpětného vyhledávání (Backtracking)

edná se o metodu prohledávající stromovou stavovou strukturu řešeného algoritmického problému. Může být využita při hledání jak jednoho tak i všech úspěšných řešení daného problému. Algoritmus lze uplatnit v oblastech jako jsou umělá inteligence (v případě této práce coby nástroj užitý k řešení SAT problému), řešení sudoku, počítačem hrané šachy a mnoho podobných využití. Konkrétním problémem řešitelným touto metodou je problém osmi dam (cílem je umístit na šachovnici osm dam tak, aby se žádné dvě navzájem neohrožovaly). Backtracking našel své využití také v logických jazycích, mezi které patří Prolog, Planner atd.

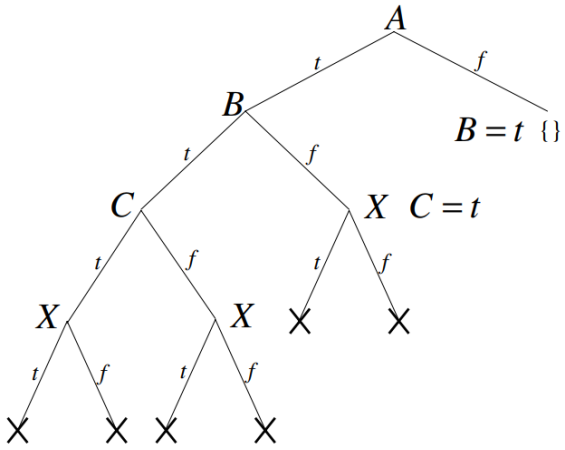
Chronologický backtracking

Výpočet algoritmu začíná od kořenového uzlu prostoru prohledávaného do hloubky. Vždy, když dojde ke konfliktu a nelze tedy touto cestou najít uspokojivé řešení, vrátí se algoritmus k uzlu právě o jednu úroveň výše a pokračuje další cestou. Následující příklad toto chování znázorňuje. Příklad byl převzat z literatury [2].

Nechť f je formule skládající se ze sedmi klauzulí popsaných následovně:

$$f = (A \vee B) \wedge (B \vee C) \wedge (\overline{A} \vee \overline{X} \vee Y) \wedge (\overline{A} \vee X \vee Z) \wedge (\overline{A} \vee \overline{Y} \vee Z) \wedge (\overline{A} \vee X \vee \overline{Z} \wedge (\overline{A} \vee \overline{Y} \vee \overline{Z})) \quad (4.8)$$

Této formuli odpovídá terminační strom na obrázku 4.5.



Obrázek 4.5: Terminační strom odpovídající formuli z příkladu na předchozí straně.

Uvažujme ohodnocení proměnných $A = true$, $B = true$, $C = true$ a $X = true$. Pak bude klauzule způsobující konflikt obsahovat proměnné A, B, C a X. Algoritmus se vrátí na předchozí úroveň, na které je ještě možné vybrat další možnost ohodnocení proměnné a opět pokračuje v prohledávání stavového prostoru do hloubky. V tomto případě se tedy algoritmus vrátí k úrovni stromu, kde dochází k ohodnocení proměnné X a zkusí hodnotu *false*, kde opět dojde ke konfliktu a je třeba vrátit se na úroveň uzlu C, kde se opět použije hodnota *false* a celý postup dále pokračuje. Nakonec se algoritmus musí vrátit až ke kořenovému uzlu A, kde po změně hodnoty na *false* dojde vlivem propagace proměnných k vynucení hodnoty *true* u proměnné B. Toto ohodnocení, tedy $A = false$ a $B = true$ již zajišťuje splnitelnost celé formule bez ohledu na ohodnocení všech ostatních. Proto již tato ohodnocení netřeba vyšetřovat a lze formuli prohlásit za splnitelnou.

Nechronologický backtracking

Jedná se o modifikaci klasického zpětného vyhledávání. Zatímco klasická verze této metody umožňuje pouze návrat na první úroveň, kde lze provést změnu ohodnocení, tato rozšířená verze bere při návratu v úvahu, které proměnné způsobily konflikt bránící splnitelnosti formule. Z této sady proměnných se vybere jedna, jejíž hodnota byla změněna jako poslední. Všechna ohodnocení nacházející se mezi uzlem naposledy změněné proměnné a aktuální pozicí jsou ignorována. Tento princip si nyní předvedeme na příkladu z předchozí kapitoly.

Budeme brát v úvahu ohodnocení proměnných $A = true$, $B = true$, $C = true$, $X = true$. Následně díky propagaci těchto hodnot v klauzuli číslo 3 získáme hodnotu $Y = true$. Dále stejným postupem v klauzuli 5 získáme ohodnocení $Z = true$. Toto ohodnocení ovšem způsobí konflikt ve klauzuli číslo 7, která již nemůže být za těchto podmínek splněna a tedy celá formule v CNF nemůže být splněna. V tomto případě sada proměnných podílejících se na konfliktu obsahuje proměnné A, X, Y, Z, všechny se shodnou kladnou hodnotou *true*. Tato verze zpětného vyhledávání na rozdíl od klasické verze přejde při návratu ke změně ohodnocení proměnné X na *false*. Toto však nestačí a dojde k dalšímu konfliktu, tentokrát způsobený proměnnými A, X, Z, všechny opět nastaveny na hodnotu *true*. Nyní algoritmus již vyčerpal obě možnosti ohodnocení proměnné X, a proto přistoupí ke změně hodnoty

proměnné A na negativní a pokračuje ve výpočtu. Tento příklad byl částečně přejat z knihy "HANDBOOK of satisfiability"[2].

Tento přístup může ušetřit velkou část času potřebného k ověření splnitelnosti formule tím, že vynechává některé části stavového prostoru. Ovšem v nejhorším případě může tato modifikace naopak výkonu uškodit, protože se jedna a ta stejná chyba může objevit vícekrát.

Kapitola 5

Využití SAT solveru při návrhu hardwaru

SAT solvery lze využít také při návrhu a optimalizaci logických obvodů. V této kapitole je takový způsob použití podrobně popsán.

5.1 ATPG

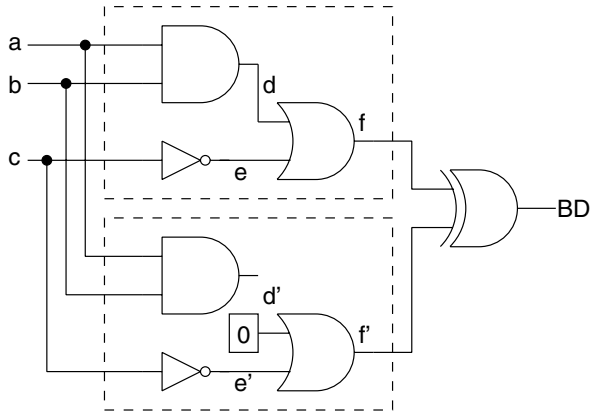
Zkratka ATPG pochází z anglického „Automatic Test Pattern Generation“ a označuje činnost, kdy je třeba po vytvoření nového integrovaného obvodu ověřit jeho správnou funkci (částečně převzato [2]). Tato metoda obsahuje generování testovacích vzorů odpovídajících požadovanému chybovému modelu a následné kontroly činnosti obvodu. Testovací vzor pro daný chybový model je ohodnocení primárních vstupů obvodu, které vyvolává v závislosti na přítomnosti zkoumané chyby rozdílné výstupní hodnoty. Booleovský rozdíl chybného a zcela funkčního obvodu vyprodukuje všechny testovací vzory použitelné k odhalení chyby, která se vyskytuje v konkrétním chybném obvodu [2]. Chyby lze rozdělit do dvou skupin podle toho, zda pro danou chybu existuje nějaký testovací vzor (rozdělení na testovatelné a netestovatelné chyby). Rozhodnutí, do které kategorie chyba spadá je NP-úplný problém [2]. Cílem je kategorizovat všechny možné chyby a pokrýt každou chybu alespoň jedním testovacím vzorem.

Výsledkem ATPG je zjištění správnosti funkce obvodu. První mechanismy ATPG využívaly zpětné prohledávání skrze strukturu navrhovaného obvodu. Nicméně postupem času s narůstající složitostí a počtem logických členů integrovaných obvodů bylo nutné přejít k výkonnějším metodám jejich ověřování. Právě zde našly své uplatnění SAT solvery. V porovnání s původními metodami se Sat solvery jeví jako výrazně výkonnější a robustnější [2]. Tento přístup využívá veškeré výhody kombinace řešení pomocí SAT problému a strukturálního přístupu k výpočtu.

Během ověřování funkčnosti navrženého logického obvodu však není možné ověřit všechny možné nedokonalosti a chyby obvodu. Proto se zavádí abstraktní pojem chybový model, anglicky „fault model“. Dále se budeme věnovat jeho variantě nesoucí anglický název „Stuck-at fault model“, dále jen SAFM.

Model SAFM se zaměřuje na spojem mezi hradly obvodu, které nabývají konstantně stejné hodnoty bez ohledu na hodnoty na vstupu. Podle logické hodnoty, jíž výstupní bod nabývá, lze tento model rozdělit na stuck-at-0 (SA0) pro logickou hodnotu 0 a analogicky

pro logickou hodnotu 1 stuck-at-1 (SA1) [2]. Příklad vzniku chyby SA1 ilustruje obrázek 5.1.



Obrázek 5.1: Rozdíl mezi chybným obvodem a obvodem správným. Přejato z literatury [2].

5.1.1 ATPG s využitím SAT problému

Tato kapitola se podrobněji zaměřuje na verzi ATPG s využitím SAT solverů a dílčích úloh, které musí tento přistup vyřešit. Tři hlavní úlohy , jež musí ATPG řešení zvládnou jsou:

1. propagace
2. implikace
3. odůvodnění

Na rozdíl od metod pracujících nad strukturálním modelem zkoumaného obvodu, ATPG s využitím SAT solveru využívá booleovský model, kde je celá logická funkcionalita obvodu vyjádřena jako jedna formule v konjunktní normální formě. Aby bylo možné formuli dále zpracovávat (řešit SAT problém), je nutné využít pokročilé efektivní algoritmy. Nalézt takový algoritmus však může být komplikované bez znalosti jakékoliv informace o struktuře obvodu. Problém lze řešit vytvořením klauzulí v CNF popisujících strukturu požadované části obvodu a jejich přidání do formule řešené SAT solverem. V poslední době se vyskytují také řešení, která zavádí zcela novou vrstvu zpracování formule do SAT solverů speciálně určených k řešení formulí popisujících logické obvody. Tato vrstva modeluje topologii obvodu. Model je však použitelný pouze při odůvodnění ohodnocení proměnných formule, což výrazně omezuje možnosti zvýšení efektivity výpočtu [15].

inární rozhodovací diagramy (dále jen BDD z anglického „Binary Decision Diagram“) jsou další technikou optimalizace výpočtu. Optimalizace se týká úloh propagace a odůvodnění. BDD má ovšem také nevýhody. Mezi hlavní nevýhody patří paměťová exponenciální náročnost a náročnost odůvodnění hodnoty proměnných, při kterém se vždy musí propočítat všechna možná odůvodnění a to i v případě, že stačí pouze jedno [15]. V nejhorším případě může modelování všech cest propagace proměnných způsobit zahlcení paměti. Řešení pomocí BDD využívá model rozdělení obvodu a metodu Booleovského rozdílu.

Ačkoliv implikace, odůvodnění i propagace dosahují dostatečného výkonu, je třeba rychlý výpočet podpořit také náležitě výkonnými datovými strukturami. Implikační graf

(IG) obsahuje všechny potřebné informace potřebné k efektivnímu a rychlému průběhu kontroly logického obvodu. IG kombinuje jak informace strukturální, tak i funkcionální. Proto lze všemožná vylepšení a zdokonalení technik založených na strukturálním popisu obvodu aplikovat také zde na IG a tím dosáhnou dalšího zlepšení výkonosti a efektivity. Jako další výhoda IG se jeví lineární paměťová náročnost. IG se generuje dynamicky pro danou logiku algoritmus implikace odůvodnění, což umožňuje použít IG bez závislosti na zvolené logice a výběru chybového modelu.

5.1.2 Implikační graf

Tato kapitola se věnuje podrobnému popisu implikačního grafu, jeho definici a ukazuje příklad, jak takový graf může vypadat.

Definice: Implikační graf IG je orientovaný graf IG=(V,E), kde V je množina všech vrcholů grafu IG a dělí se na signálové vrcholy a na ∧-vrcholy. Signální vrcholy odpovídají bitům modelovaného obvodu ∧-vrcholy odpovídají operaci konjunkce nezbytné k vytvoření ternárních klauzulí. Uzly označující signály jsou označeny c_x (c_x*) tak, že x odpovídá signálu x uvnitř logického obvodu, symboly latinské abecedy značí ∧-vrcholy implikačního grafu [15]. Tyto uzly nazýváme ternárními klauzulemi.

Nekonzistentní nebo konfliktní signály lze jednoduše dohledat, protože jsou reprezentovány formulí $c_x \wedge c_x{}^* = 1$. Ohodnocení je označeno jako nekonfliktní, pokud platí:

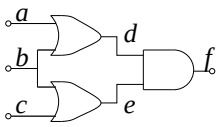
$$c_x \wedge c_x* \iff 0 \tag{5.1}$$

pro všechna proměnné signálu x.

Množina hran E reprezentující všechny implikace se dělí na tři vzájemně disjunktní podmnožiny: dopředné hrany, zpětné hrany a všechny ostatní implikace přítomné v obvodu.

Příklad 5.1 [15]:

– Struktura logického obvodu:



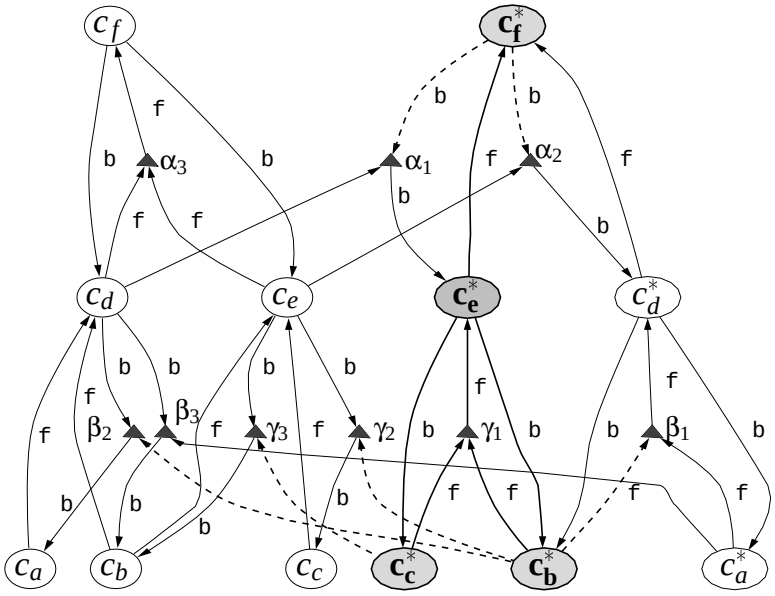
Obrázek 5.2: Struktura v příkladu použitého obvodu [15].

– Formule v konjunktní normální formě:

$$\begin{array}{lll} (\neg c_d^* \vee c_f^*) \wedge (\neg c_e^* \vee c_f^*) \wedge (\neg c_d \vee \neg c_e \vee c_f) & \wedge & f = AND(d,e) \\ (\neg c_a \vee c_d) \wedge (\neg c_b \vee c_d) \wedge (\neg c_a^* \vee \neg c_b^* \vee c_d^*) & \wedge & d = OR(a,b) \\ (\neg c_b \vee c_e) \wedge (\neg c_c \vee c_e) \wedge (\neg c_b^* \vee \neg c_c^* \vee c_e^*) & \wedge & e = OR(b,c) \\ \iff 1 & & \end{array}$$

Obrázek 5.3: Formule v CNF reprezentující obvod v příkladu 5.1 [15].

– Implikační graf G = (V,E) je zobrazen na obrázku 5.4:



Obrázek 5.4: Implikační graf reprezentující formuli v příkladu 5.1 [15].

5.1.3 Implikace

Implikace založená na implikačním grafu IG nevyžaduje kompletní průchod skrze IG, nýbrž stačí pouze částečný, který lze provést na základě počáteční sady ohodnocených uzlů, která je podmnožinou množiny signálních uzlů pro všechny uzly následující. Následujícímu uzlu lze přiřadit hodnotu pokud patří do množiny -vrcholů a všechny předcházející uzly jsou ohodnoceny nebo pokud uzel patří do množiny signálních vrcholů a zároveň je alespoň jeden jeho předchůdce ohodnocen.

5.1.4 Odůvodnění

V kontextu ATPG odůvodnění označuje proces nalezení ohodnocení vstupních signálů obvodu tak, aby byla vynucena požadovaná logická hodnota na požadovaném místě uvnitř obvodu. ATPG využívá tři různé metody odůvodnění založené na strukturálním popisu obvodu, na řešení převodem na matematické formule a řešení jejich splnitelnosti (problém SAT) a metody využívající implikační graf.

Metody založené na strukturálním popisu obvodu nejprve provedou odůvodnění výstupních signálů logických hradel, jejichž výstupní hodnota není nijak ovlivněna hodnotami primárních vstupů celého obvodu.

Metody založené na SAT problému řeší odůvodnění implicitně během hledání ohodnocení formule v CNF reprezentující logický obvod takové, aby byla celá formule splnitelná. Nevýhodou tohoto přístupu může být zanedbání veškerých informací o struktuře logického obvodu. SAT solver hledající řešení tak nemůže rozlišovat mezi interními signály obvodu a jeho primárními vstupy. Nicméně tento fakt již vyplývá z obecného přístupu k řešení SAT problému.

Odůvodnění s použitím implikačního grafu využívá propojení proměnných hranami ke kontrole jejich závislostí a odvození správných logických hodnot proměnných. Zde stále platí, že všechny klauzule musí být splněny. Nalezení nesplněných ternárních klauzulí (-

uzly) nevyžaduje žádné zvýšené úsilí, protože pouze jeden ze dvou předků každé takové nesplnitelné klauzule nabývá kladné logické hodnoty.

5.1.5 Propagace

V kontextu ATPG se jedná o proces změny vnitřního signálu obvodu, který vyvolá změnu alespoň na jednom ze všech primárních výstupů logického obvodu. Konkrétní řešení se liší pro různé přístupy k ATPG.

Strukturálně založené algoritmy využívají postupného posouvání D-hranice. Tento algoritmus lze pojmenovat D-drive [4].

Algoritmy zaměřující se pouze na použití logických formulí lze zrychlit zavedením nových klauzulí, pokrývající alespoň nějaké informace o struktuře vstupního obvodu. Tím lze dosáhnout podobného efektu jako při využití algoritmu D-drive. Nicméně tento přístup zvyšuje složitost převodu logického obvodu na ekvivalentní formuli v konjunktní normální formě jako vstup pro použitý SAT solver. Nicméně nárůst náročnosti převodu do konjunktní normální formy společně se značně omezeným množstvím informacím o struktuře obvodu značně omezuje využitelnost toho přístupu.

V případě využití implikačního grafu dosahuje propagace hodnot vnitřních signálů obvodu podobných výsledků jako v případě algoritmů založených na struktuře logických obvodů, protože implikační graf obsahuje dostatečné množství informací o vnitřní podobě obvodu. Využití implikačních grafů se zdá být ovšem jednodušší, neboť obsahuje pouze dva různé typy uzlů. V případě strukturálně založených algoritmů, které vyžadují speciální typ uzlu pro každý druh logického hradla.

5.2 CDCL

CDCL značí konfliktem řízené učení klauzulí, zkratka pochází z anglického (Conflict Driven Clause Learning). Tento přístup přináší možnost vytvořit vysoce výkonné a efektivní SAT solvery. Proto SAT solvery s využitím CDCL pronikly do mnoha různých oborů od návrhu hardwaru a softwaru, hledání závislostí mezi balíčky nějakého softwaru až po kryptografii.

CDCL SAT solvery jsou do značné míry inspirovány technologií DPLL SAT solverů. CDCL používá podobně jako DPLL metodu zpětného vyhledávání. Pokud během některého z kroků větvícího průchod prostorem všech ohodnocení proměnných formule (takovým krokem je přiřazení hodnoty 0 nebo 1 některé z proměnných) dojde ke konfliktu, zapojí se metoda zpětného vyhledávání do výpočtu. Metoda zpětného vyhledávání se postupně vrací skrze stromovou strukturu ohodnocení proměnných dokud nenarazí na větev, u které zatím nebyly vyšetřeny všechny možné hodnoty. Zde lze využít obou variant metod zpětného vyhledávání, chronologické i nechronologické.

Kromě technik poděděných od DPLL přináší CDCL SAT solvery několik nových důležitých metod:

- učení se nových klauzulí z konfliktů během zpětného prohledávání,
- využití struktury konfliktů během učení nových klauzulí,
- vylepšení větvení podle ohodnocení proměnných musí mít nízkou horní hranici náročnosti výpočtu a musí přijímat zpětnou vazbu ze zpětného vyhledávání,
- pravidelné restartování zpětného vyhledávání,

– techniky odstranění naučených klauzulí, propagace jednotkových klauzulí atd.

V této kapitole se budeme setkávat s označením klauzulí *splněná*, *nesplněná*, *jednotková* a *nerozhodnutá*. *Splněná* je taková klauze, kde alespoň jeden její prvek nabývá hodnoty logická 1. Aby byla klauzule *nesplněná* musí všechny její prvky nabývat hodnoty logická 0. Klauzuli lze považovat za *jednotkovou*, pokud jsou všechny prvky klauzule kromě jednoho v logické 0 a jeden prvek prozatím není ohodnocen. Klauzule je *nerozhodnutá* za předpokladu, že nespadá ani do jedné ze tří předchozích kategorií.

SAT solvery využívají zejména jednotkových klauzulí, protože poskytují pravidlo, že zbývající jediný prvek klauzule musí nabývat hodnoty logická 1, aby mohla být klauzule splněna. V případě CDCL, podobně jako ve většině implementací DPLL, jsou logické důsledky odvozeny pomocí propagace jednotkových klauzulí. Propagace se provádí po každém ohodnocení proměnné, která dosud ohodnocena nebyla, tedy přibyla nová větev prostoru ohodnocení. Zároveň slouží k identifikaci proměnných, které musí být nastaveny na specifickou logickou hodnotu. V případě, že je detekována nesplnitelná klauzule, deklaruje se konfliktní podmínka použije se zpětné vyhledávání k návratu ke změně ohodnocení příslušné proměnné.

Každá proměnná v SAT solverech využívajících CDCL je charakterizována popisem více vlastností, mezi které patří především její hodnota, předchůdce a úroveň rozhodování. Proměnná, které je přiřazeno hodnocení na základě propagace jednotkových klauzulí, se nazývá klauzule odvozená. Jako předchůdce proměnné označíme jednotkovou klauzuli, z níž byla hodnota proměnné odvozena. Pokud proměnná zatím nebyla ohodnocena, pak nemá žádného předchůdce a toto pole je tedy prázdné. Úroveň rozhodování odpovídá hloubce, kde došlo k ohodnocení proměnné v rozhodovacím stromu. Rozhodovací úroveň pro dosud neohodnocené proměnné je -1. Hodnota proměnné může nabývat logických hodnot 1 a 0 nebo u pokud ještě nebyla ohodnocena.

5.2.1 Struktura CDCL SAT solverů

Standardní implementace CDCL solverů v základech odpovídá struktuře DPLL s tím, že hlavním rozdílem v CDCL je provedení analýzy konfliktu pokaždé, kdy dojde ke konfliktu a následně k návratu. Dalším rozdílem je možnost využití nechronologického backtrackingu. Algoritmus CDCL lze popsat následujícím pseudokódem na obrázku 5.5.

```
CDCL(φ, ν)
 1  if (UnitPropagation(φ, ν) == CONFLICT)
 2     then return UNSAT
 3  dl ← 0                                          ▷ Decision level
 4  while (not AllVariablesAssigned(φ, ν))
 5        do (x, v) = PickBranchingVariable(φ, ν)                ▷ Decide stage
 6           dl ← dl + 1                  ▷ Increment decision level due to new decision
 7           ν ← ν ∪ {(x, v)}
 8           if (UnitPropagation(φ, ν) == CONFLICT)              ▷ Deduce stage
 9              then β = ConflictAnalysis(φ, ν)                  ▷ Diagnose stage
10                   if (β < 0)
11                      then return UNSAT
12                      else Backtrack(φ, ν, β)
13                           dl ← β          ▷ Decrement decision level due to backtracking
14  return SAT
```

Obrázek 5.5: Pseudokód algoritmu CDCL [2].

V pseudokódu na obrázku 5.5 použity tyto pomocné funkce ke zjednodušení zápisu [2]:

- UnitPropagation – obsahuje iterační provedení propagace jednotkových klauzulí, v případě konfliktu indikuje tento fakt na svém výstupu,
- PickBranchingVariable – tato funkce vybere proměnnou a hodnotu, na kterou má být nastavena
- ConflictAnalysis – funkce provede analýzu posledního konfliktu, ke kterému došlo a naučí se nové konfliktní klauzule,
- Backtrack – provede návrat (backtracking) na úroveň v rozhodovacím stromě určenou funkcí ConflictAnalysis,
- AllVariablesAssigned – kontroluje zda jsou všechny proměnné ohodnoceny, pokud ano, pak je formule vyhodnocena jako splnitelná. Alternativně lze kontrolovat zda jsou splněny všechny klauzule. Tento přístup však moderní SAT solvery nevyužívají, neboť není slučitelný s moderními datovými strukturami umožňujícími rychlejší rozhodnutí SAT problému.

5.2.2 Učení klauzulí pomocí konfliktu

CDCL SAT solver prování propagaci jednotkových klauzulí. Pokaždé, když dojde ke konfliktu, se provede jeho analýza. Během analýzy se SAT solver naučí jednu i více klauzulí na základě kroků propagace jednotkových klauzulí, které tento konflikt způsobily. Struktura propagace jednotkových klauzulí poskytuje dostatečné množství informací pro rozhodnutí, které proměnné, případně jejich negace, mají být přidány do nově naučené klauzule.

Hloubka, v níž se proměnná nachází v rozhodovacím stromě, určuje pořadí proměnných. Počínaje nesplněnou klauzulí způsobující daný konflikt, analýza postupně vyšetřuje proměnné, které byly ohodnoceny jako poslední. Následně vyhledá předky těchto proměnných. Z nich následně vybírá prvky na nižších úrovních rozhodovacího stromu než je aktuální maximální úroveň. Tento postup se opakuje dokud není dosažena proměnná, jejíž hodnota byla změněna jako poslední.

Příklad učení klauzulí z literatury pod číslem [7]:
nechť d je aktuální úroveň v rozhodovacím stromu, x_i proměnná, jejíž rozhodnutí je předmětem zkoumání,

$$v(x_i) = v \tag{5.2}$$

je rozhodující ohodnocení a ωj klauzule identifikována jako nesplněná propagací jednotkových klauzulí. V kontextu implikačního grafu je konfliktní vrchol k identický $\alpha(k) = \omega j$. Dále nechť $\diamond$ reprezentuje rozhodovací operátor. Pro dvě klauzule ωj a ωk, pro které existuje unikátní proměnná x taková, že jedna klauzule obsahuje literál x a druhá $\overline{x}$, $\omega j \diamond \omega k$ obsahuje všechny prvky (literály) klauzulí ωj a ωk kromě x a $\overline{x}$.

Učení klauzulí používané v SAT solverech lze definovat jako posloupnost selektivních rozhodovacích operací, kde každý krok vytvoří novou dočasnou klauzuli. Nejprve definujme predikát určující, zda klauzule ω má odvozený literál ohodnocený na aktuální rozhodovací úrovni d:

$$\xi(\omega, l, d) = \begin{cases} 1 & \forall\, l \in \omega \wedge \delta(l) = d \wedge \alpha(l) \neq NIL \\ 0 & jinak \end{cases} \tag{5.3}$$

Nechť $\omega_L^{d,i}$, kde $i = 0,1,\ldots$, je dočasná klauzule vytvořená po i rozhodovacích operacích. Užitím předchozího predikátu z rovnice 5.3 lze tuto dočasnou klauzuli definovat následovně:

$$\omega_L^{d,i} = \begin{cases} \alpha(k) & if\ i = 0 \\ \omega_L^{d,i-1} \odot \alpha(l) & if\ i \neq 0 \wedge \xi(\omega_L^{d,i-1}, l, d) = 1 \\ \omega_L^{d,i-1} & if\ \ \neq 0 \wedge \forall_l \xi(\omega_L^{d,i-1}, l, d) = 0 \end{cases} \tag{5.4}$$

Rovnice 5.4 může být použita k formalizaci procedury učení klauzulí. První podmínka, $i = 0$, určuje počáteční krok k v i, kde každý literál nacházející se v nesplněné klauzuli je přidán do první dočasné pomocné klauzule. Následně se v každém kroku i vybere literál ohodnocený na aktuální rozhodovací úrovni a pomocná klauzule se označí jako předchůdce. Tento postup můžeme pozorovat v tabulce 5.1.

$\omega_L^{5,0} = \{x_5, x_6\}$	literály v $\alpha(k)$
$\omega_L^{5,1} = \{\overline{x_4}, x_2 1\}$	řešeno s $\alpha(x_5) = \omega_4$
$\omega_L^{5,2} = \{\overline{x_4}, x_2 1\}$	řešeno s $\alpha(x_6) = \omega_5$
$\omega_L^{5,3} = \{x_2, x_3, x_2 1\}$	řešeno s $\alpha(x_4) = \omega_3$
$\omega_L^{5,4} = \{x_1, x_3, x_2 1, x_3 1\}$	řešeno s $\alpha(x_2) = \omega_1$
$\omega_L^{5,5} = \{x_1, x_2 1, x_3 1\}$	řešeno s $\alpha(x_3) = \omega_2$
$\omega_L^{5,6} = \{x_2, x_2 1, x_3 1\}$	žádná další řešešní

Tabulka 5.1: Kroky řešení během učení se klauzulí [7].

Pokud pro iteraci i platí $\omega_L^{d,i} = \omega_L^{d,i-1}$, pak bylo dosaženo pevného bodu a $\omega_L = \omega_L^{d,i}$,$i$ reprezentuje nově naučenou klauzuli. Moderní SAT solvery implementují dodatečné pročištění rovnice pomocí dalšího zkoumání struktury odvozování hodnot proměnných. Konec přejatého příkladu [7].

5.2.3 Další použití učení se klauzulí

Učení klauzulí lze kromě CDCL SAT solverů použít například ke znovupoužití klauzulí naučených během práce na jedné formuli během zpracování jiné formule podobné formuli první. Další využití učení klauzulí najdeme u inkrementální SAT solverů. V případě CDCL SAT solveru lze z naučených klauzulí identifikovat konkrétní zdroj nesplnitelnosti celé formule a tím lokalizovat konflikt ve zkoumané formuli.

5.3 SAT solvery a CDCL

Tato podkapitola se zaměřuje na konkrétní využití CDCL v rámci implementací SAT solverů dnešní doby. Dále popisuje prostředky využívané SAT solvery s cílem dosažení maximální možné rychlosti a efektivity. Mezi tyto techniky patří zejména restartování vyhledávací procedury, strategie odstraňování naučených klauzulí, konfliktem řízené zefektivnění větvení vyhledávacího stromu a datové struktury podporující tyto činnosti. Pro tyto datové struktury používáme anglický název Lazy Data Structures.

5.3.1 Lazy data structures

V této podkapitole se budeme věnovat podrobnému popisu Lazy data struktur užitečných při řešení SAT problému. Tyto struktury jsou nezbytné pro dosažení požadované rychlosti a

efektivity výpočtu (operace týkající se správy dat by měly dosahovat pouhého zlomku výpočetní náročnosti algoritmu). Zároveň musí datové struktury poskytovat dostatek prostředků k ukládání klauzulí, literálů a proměnných. Dnešní moderní SAT solvery disponují Lazy data strukturami, čímž se minimalizuje výpočetní čas potřebný k průchodu jedním uzlem vyhledávacího stromu. Toto je novinka oproti dřívějším SAT solverům využívajícím klasické datové struktury, které odpovídaly reálné podobě formule a bylo tedy možné v každém okamžiku zjistit aktuální hodnotu každého prvku uvnitř klauzule. Klasický přístup k ukládání klauzulí do paměti může najít u algoritmu GRASP [8]. Využití moderních metod ukládání do paměti pomocí "Lazy data structures"využívají například Chaff [6] a MiniSat [14].

Klasický způsob uchovávání klauzulí v paměti ukládá klauzule jako seznamy jejich jednotlivý prvků. Dále se ukládá pro každou proměnnou seznam klauzulí, které tuto proměnnou obsahují. Pokud je nějaké proměnné přiřazena hodnota, pomocí tohoto seznamu jsou všechny klauzule obsahující tuto proměnnou seznámeny s novou hodnotou. Zde ovšem nastává problém, protože proměnné mohou udržovat odkazy na příliš velký počet klauzulí. V případě použití CDCL tento počet v průběhu výpočtu dále narůstá a důsledkem tohoto narůstá také počet operací, které se musí nutně provést při každém ohodnocení nebo změně ohodnocení každé proměnné [7]. Navíc ne všechny klauzule, kde se proměnná nachází, musí být vyhodnoceny a to i v případě, že daná klauzule není jednotková a zároveň klauzule nemůže přejít do nesplnitelného stavu.

Pokud budeme brát v úvahu pouze klauzule, které nejsou splněny a jednotkové klauzule, pak stačí pouze dva odkazy pro každou klauzuli. Kdykoliv, kdy jeden z odkazů odkazuje na prvek klauzule nabývající hodnoty logická 0, odkaz se posune na další prvek, který je buď v logické 1 nebo mu ještě žádná hodnota nebyla přiřazena. Algoritmus na obrázku 5.6 ukazuje, jak lze pomocí pouze těchto dvou odkazů určit, zda je klauzule jednotková, splněná, nebo nesplněná. Jednotlivé odkazy jsou označeny *RefA* a *RefB*.

```
CDCL(φ, ν)
 1  if (UnitPropagation(φ, ν) == CONFLICT)
 2     then return UNSAT
 3  dl ← 0                                          ▷ Decision level
 4  while (not AllVariablesAssigned(φ, ν))
 5        do (x, v) = PickBranchingVariable(φ, ν)                    ▷ Decide stage
 6           dl ← dl + 1                  ▷ Increment decision level due to new decision
 7           ν ← ν ∪ {(x, v)}
 8           if (UnitPropagation(φ, ν) == CONFLICT)                  ▷ Deduce stage
 9              then β = ConflictAnalysis(φ, ν)                      ▷ Diagnose stage
10                   if (β < 0)
11                      then return UNSAT
12                      else Backtrack(φ, ν, β)
13                           dl ← β          ▷ Decrement decision level due to backtracking
14  return SAT
```

Obrázek 5.6: Algorimus rozhodnutí klauzule pouze s využitím dvou odkazů [7].

Head and Tail datové struktury

Tato forma datových struktur byla představena jako první lazy data structure specializovaná na řešení problému SAT [7]. Tento přístup byl původně použit v implementaci Sato SAT solveru [17]. Jak již název napovídá tato metoda používá dva odkazy Head (dále jen H) a Tail (dále jen T) na prvky klauzulí.

Z počátku H odkaz ukazuje na první prvek klauzule a T odkazuje na poslední prvek. Kdykoliv je přiřazeno některému z prvků, na které tyto dva odkazy ukazují, hledá se nový

dosud neohodnocený prvek. Opakováním předchozího kroku se oba odkazy postupně posouvají od okrajů ke středu klauzule. Pokud narazí na neohodnocený prvek klauzule, stane se tento prvek novým začátkem nebo koncem (místem kam odkazují H a T). H a T tedy opět odkazují na neohodnocené proměnné. Zde však musí být zajištěno, že při případném backtrackingu oba odkazy H a T byly správně posunuty zpět. Pokud je během postupu ke středu klauzule nalezen prvek s kladnou hodnotou, označí se celá klauzule jako splněná. Podobně v případě kdy, není nalezen žádný neohodnocený prvek a oba odkazy H a T se potkají, je klauzule označena jako jednotková klauzule, splněná nebo nesplněná v závislosti na hodnotě referované druhým odkazem.

Při návratu z neúspěšné větve prohledávání, se ruší odkazy H a T a aktivují se jejich předchozí hodnoty. Návrat v rámci odkazů H a T má i v nejhorším možném případě lineární časovou složitost přímo úměrnou délce klauzule.

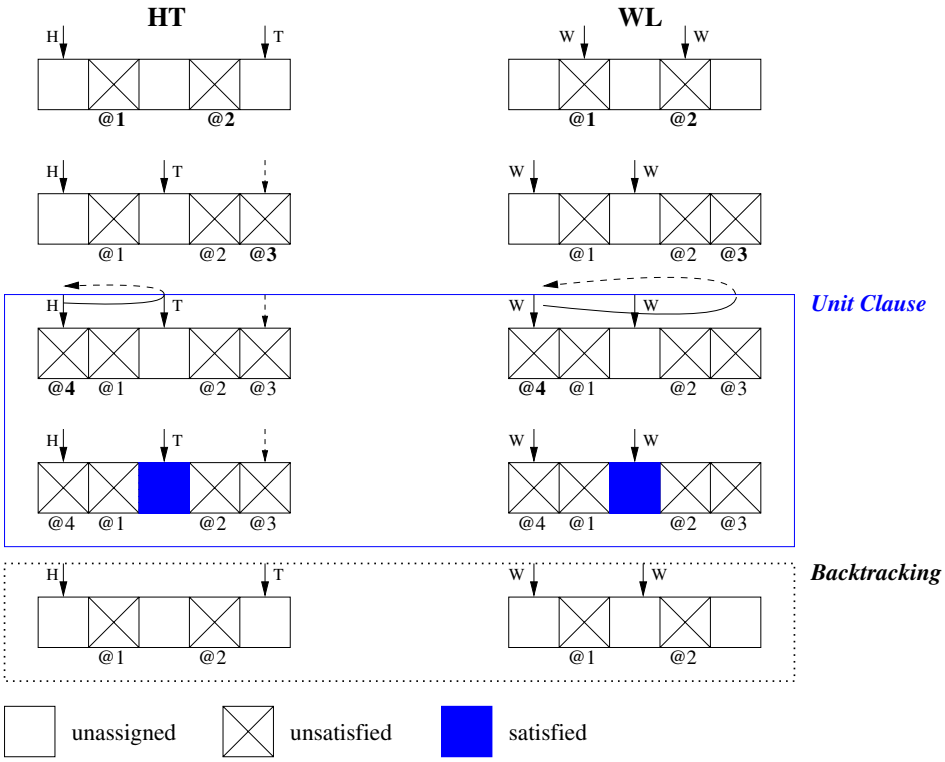
Obrázek 5.7 znázorňuje využití datové struktury H/T. Pro jednoduchost je zde znázorněna pouze jedna klauzule. Klauzule má podobu seznamu jejich prvků. Odkazy H a T odkazují pouze na neohodnocené prvky.

Watched literal data structure

Datová struktura pod anglickým názvem „Watched Literal Data Structure“, dále zkráceně WL, byla vůbec poprvé představena v poslední verzi SAT solveru Chaff [6] a zaměřuje se na odstranění problémů vyskytujících se u seznamů, které využívá H/T [7]. Podobně jako u H/T se udržují dva odkazy na jednu klauzuli. Přístup k pořadí odkazů je ovšem v kontrastu s metodou, kterou používá H/T. Zde již neexistuje žádný vztah mezi pozicemi obou odkazů uvnitř klauzule. Toto umožňuje oběma odkazům pohybovat se nezávisle libovolným směrem. Hlavní výhodu WL datové struktury pozorujeme v momentu, kdy dojde ke konfliktu a musí se vykonat návratová procedura. Tato metoda díky absenci závislosti odkazů na pořadí nemusí zpracovávat žádné odkazy na prvky klauzule. Na druhou stranu se zde objevují i zjevné nevýhody. Klauzuli lze prohlásit za nesplnitelnou nebo jednotkovou až poté, co odkazy navštívily všechny prvky klauzule. Identifikace ostatních stavů klauzule probíhá téměř identicky s předchozí metodou H/T.

Na rozdíl od H/T není třeba v případě WL uchovávat žádné další odkazy v seznamech uchovávajících historii pohybu odkazů mezi prvky klauzule díky tomu, že při vykonávání návratu není potřeba odkazy nijak upravovat. Protože už nejsou uchovávány historické odkazy, zůstává počet odkazů na každou proměnnou konstantní.

Znázornění WL můžeme vidět spolu s H/T na obrázku 5.7. Jeden z odkazů odkazuje na již ohodnocený prvek klauzule. Proto se musí posunout. Z obrázku je patrný rozdíl spočívající v možnosti obousměrného pohybu obou odkazů.



Obrázek 5.7: Použití dvou "lazy data"struktur [7].

Srovnání datových struktur

Popis činnosti datových struktur jak klasických tak i líných datových struktur obsahují předchozí kapitoly. Nyní se zaměříme na porovnání specifických vlastností jednotlivých typů datových struktur. Následující tabulka číslo 5.2 obsahuje srovnání klíčových vlastností klasických datových struktur pod zkratkou AL, „Head and Tail“ datových struktur pod zkratkou HT a „Watched Literal“ datových struktur pod zkratkou WL. Tabulka znázorňuje, které datové struktury jsou "líné", minimální a maximální počet odkazů na proměnné sdružené se všemi klauzulemi a složitost výpočtu při pohybu výpočtu kupředu i při zpětném návratu po výskytu konfliktu. Tabulka a v ní obsažené hodnoty byly přejaty z literatury pod číslem 3 v seznamu literatury [7].

datové strutury			AL	HT	WL
"líné"?			N	A	A
odkazů na literály	min		L	2C	2C
	max		L	L	2C
navštívené literály	během identifikace	min	1	1	W-1
	jednot./nespl. klauzulí	max	1	W-1	W-1
	během návratu		L_b	L_b	0

Tabulka 5.2: Porovnání počtu odkazů na literály sdružené se všemi klauzulemi. L značí počet literálů, C počet klauzulí, W počet literálů v klauzuli a $L_b poet nepia zench liter lb hem provd nnvratu$[7]

5.3.2 Restartování vyhledávací metody

Během výpočtu se může algoritmus dostat do situace, kdy vznikne výpočetně náročný problém, jehož řešení by znamenalo exponenciální nárůst času potřebného k řešení úlohy. Proto některé SAT solvery zavádí takzvané rychlé restarty s náhodným ohodnocením proměnných. Po takovém restartu jsou hodnoty proměnných voleny náhodně, čímž lze dosáhnou zvýšení pravděpodobnosti průchodu různých podstromů po vykonání návratu. Nicméně takový přístup vyžaduje zavedení dalších technik zajišťujících úplnost vyhledávání ve stromové struktuře, protože v případě, že volíme ohodnocení proměnných náhodně, může dojít k situaci, kdy bude existovat nějaká větev ve vyhledávacím stromě, která nebyla prohledána a výsledek tak bude nekompletní.

Druhý přístup v případě využití CDCL provádí analýzu konfliktu, aby získal informace, na jejichž základech může přesněji určit, jak by proměnné měly být ohodnoceny. Tento přístup dělá rychlé restarty vyhledávání výrazně užitečnějšími a výkonnějšími.

5.3.3 Heuristiky spojené s CDCL

Nejčastěji používanou heuristikou vylepšující výběr proměnných k ohodnocení a tedy ovlivňující také větvení vyhledávacího stromu je VSIDS. Jedná se o zkratku anglického názvu „Variable State Independent Decaying Sum“. Tato heuristika byla představena v SAT solveru Chaff [6]. K jejímu vytvoření vedla snaha o efektivnější využití líných datových struktur. VSIDS odstraňuje problém s určením dynamické velikosti zkoumané klauzule.

VSIDS

Cílem VSIDS je sbírání statistik z naučených klauzulí k usměrnění vyhledávání správným směrem. Při sbírání těchto informací se zvýhodňují méně dávno naučené klauzule. Hlavními charakteristikami VSIDS jsou multiplikativní úpadek a aditivní navýšení. Dvě hlavní varianty této heuristiky jsou cVSIDS a mVSIDS. cVSIDS byla představena v Chaff SAT solveru [6] a mVSIDS v SAT solveru MiniSat.

VSIDS přiřazuje každé proměnné ve formuli aktivitu nabývající hodnot z oboru reálných čísel. Na počátku proběhne inicializace aktivity všech proměnných na 0. Seřazení proměnných sestupně podle jejich aktivity nazýváme VSIDS pořadí. VSIDS vybírá k dalšímu zpracování proměnné s nejvyšší aktivitou.

V okamžiku, kdy se SAT solver naučí novou klauzuli vybere množinu proměnných a jejich aktivitu navýší. Toto navýšení označuje pojem aditivní nárůst. V pravidelných intervalech jsou aktivity všech proměnných násobeny konstantou ležící v otevřeném intervalu (0, 1). Tomuto snížení říkáme multiplikativní úpadek.

Varianty VSIDS:

- cVSIDS – aktivity všech proměnných vyskytujících se v nově naučení klauzuli jsou navýšeny o 1. Aktivity všech proměnných se vynásobí konstantou α, pro kterou platí: $0<\alpha<1$. Multiplikativní snížení se provede jednou za i konfliktů. Například MiniSat používá hodnotu i = 1.

- mVSIDS – aktivity všech proměnných, které vedly během analýzy ke konfliktu, jsou zvýšeny o 1. Snižování aktivity proměnných probíhá identicky jako u cVSIDS

Další metoda využitá VSIDS je graf incidence proměnných VIG (z anglického názvu Variable Incidence Graph). VIG pro danou formuli v konjunktní normální formě lze popsat následovně:

– vrcholy grafu zastupují proměnné formule. Pro každou proměnnou existuje právě jeden vrchol grafu VIG,

– klauzule jsou reprezentovány hranami grafu.

Klauzule jsou reprezentovány hranami grafu následujícím způsobem: pro každou dvojici proměnných vyskytujících se uvnitř klauzule existuje hrana spojující vrcholy označující tyto dvě proměnné. Pro každou klauzuli tedy existuje $n!/[n! \times (n-2)!]$ hran, kde n značí celkový počet proměnných uvnitř klauzule. Klauzule jako taková tedy nabývá uvnitř VIG grafu podoby kliky grafu. To znamená, že podgraf tvořený jednou klauzulí je grafem úplným [5]. Každá hrana grafu má navíc svoji váhu popsatelnou rovnicí 5.5.

$$w = \frac{1}{\mid c \mid -1} \tag{5.5}$$

kde w značí váhu hrany a c počet proměnných obsažených uvnitř klauzule, přičemž se nebere v potaz, zda je výskyt proměnné kladný nebo se jedná o negaci. V případě, kdy existuje stejná hrana ve dvou a více klauzulích, výsledná váha se určí jako součet všech vah této hrany pro všechny klauzule, ve kterých se vyskytuje. Formálně tedy můžeme graf VIG definovat následovně: Graf VIG je je uspořádaná dvojice VIG = (V,E), taková, že platí:

$$\mid V \mid = \mid Var \mid \tag{5.6}$$

$$E = \{xy \mid xy \in c \in E\} \tag{5.7}$$

kde Var označuje množinu všech proměnných formule F, xy hranu spojující proměnné x, y a c označuje kliku klauzule uvnitř grafu VIG.

Srovnání efektivity jednotlivých heuristik nad VSIDS znázorňuje následující tabulka (tabulka byla přejata ze článku „Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers“ [5]):

kategorie	mVSIDS	cVSIDS	náhodné mazání
aplikace	0.592	0.560	0.216
kombinační	0.275	0.261	0.099
náhodné	0.029	0.023	0.006

(a)

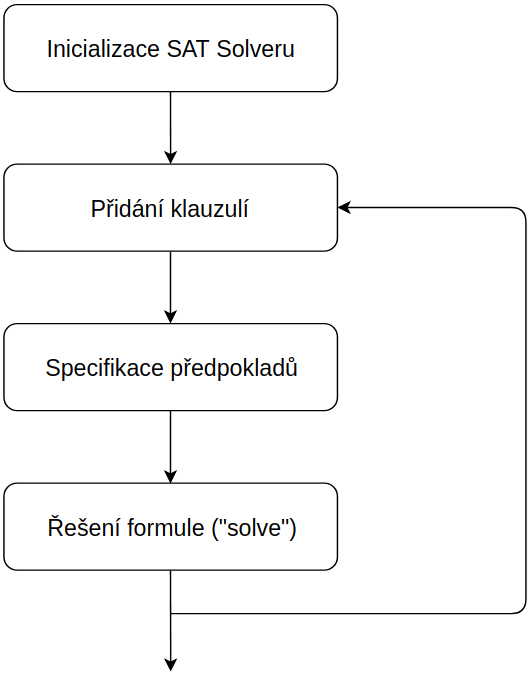
kategorie	mVSIDS	cVSIDS	náhodné mazání
aplikace	0.580	0.551	0.268
kombinační	0.505	0.473	0.265
náhodné	0.269	0.268	0.219

(b)

Tabulka 5.3: (a) prostorový experiment, hodnoty označují průměrné prostorové skóre, (b) časový experiment, hodnoty jsou průměrným časovým skóre. Podle (a) jsou VSIDS heuristiky více zaměřeny na prostorové zpracování než náhodné mazání.

5.4 Inkrementální SAT solvery

Inkrementální režim SAT solveru je speciální modifikace klasického SAT solveru sloužící k opakovanému zkoumání splnitelnosti formule s uchováním naučených konfliktních klauzulí z předchozích běhů. Aby tato funkce měla nějaký smysl, zavádí se zde předpoklady (v anglickém jazyce „assumptions“) udávající požadovanou hodnotu specifikovaných proměnných. Znovupoužití naučených klauzulí výrazně urychluje výpočet v případech, kdy je velká část formule stejná, tedy není ovlivněna zadanými předpoklady, protože již máme uložené znalosti z předchozího řešení v podobě konfliktních klauzulí. Konfliktní klauzule byly získány řešením nějaké formule, proto je vhodné použít tyto konfliktní klauzule na stejnou nebo velmi podobnou (především přidané nové klauzule, aby původně naučené konflikty stále byly platné, což by nebylo zaručeno v případě odebrání části původní formule). Ideálním případem je identická formule a změněné pouze předpoklady. Průběh takové práce inkrementálního SAT solveru s formulí ilustruje obrázek 5.8.



Obrázek 5.8: Algoritmus inkrementálního SAT solveru.

Inkrementální režim SAT solveru nachází praktické využití například při použití SAT solveru jako fitness funkce při evolučním návrhu logických obvodů. Příkladem SAT solveru podporujícího inkrementální režim může být například miniSAT.

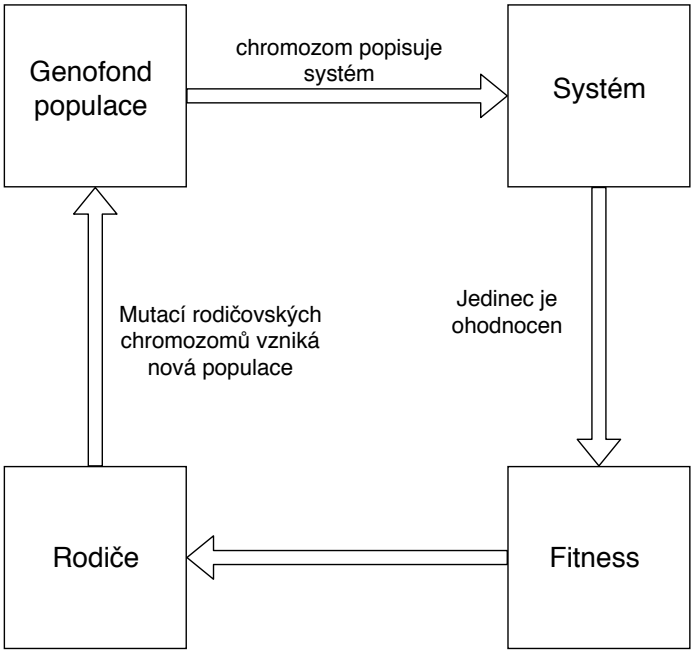
nkrementální režim SAT solveru nachází praktické využití například při použití SAT solveru jako fitness funkce při evolučním návrhu logických obvodů. Příkladem SAT solveru podporujícího inkrementální režim může být například miniSAT.

Kapitola 6

Evoluční návrh hardwaru

Klasické metody návrhu hardwaru vyžadují kompletní popis celé funkcionality požadovaného obvodu. Aby mohl být takový postup proveden, musí návrhář znát veškeré detaily zapojení jednotlivých součástek. Následně je třeba využít vývojových prostředků pro popis hardwaru k popisu celé funkcionality a struktury obvodu jako jsou například jazyky VHDL nebo Verilog. Převod do podoby srozumitelné cílovým zařízením (FPGA,...) obvykle probíhá automaticky pomocí nástrojů určených speciálně k tomuto účelu [12]. Nad takto navrženým obvodem lze dále provádět nejrůznější optimalizace. Ty ovšem pouze upravují menší části stávajícího systému.

Naproti tomu evoluční návrh hardwaru umožňuje vytvořit zcela nový obvod s kompletně novou strukturou. Tento přístup se objevil a začal šířit počátkem 90. let 20. století [12]. Evoluční návrh přináší principy postupného vývoje známé z přírody. Podobně jako při evoluci živých organismů v přírodě dochází k vytvoření generace z generace předchozí. Při vzniku nové generace dochází k mutacím. Následně na základě nejlepších jedinců aktuální generace vznikne opět další generace opět s podílem mutace. V případě návrhu hardwaru se tento postup opakuje, dokud se nenalezne alespoň jeden jedinec splňující požadované vlastnosti. Evoluční návrh také přináší možnost zavést toleranci chyby, kdy obvod nemusí poskytovat naprosto dokonalou funkcionalitu. Opakované tvoření generací ohodnocování jedinců, výběr nejlepších a generování jejich potomků znázorňuje obrázek 6.1. Aby byla evoluce hardwaru možná, musí existovat dostatečně efektivní fitness funkce.



Obrázek 6.1: Cyklus fází evolučního návrhu HW [3].

[] Jedinci z populace mohou být ohodnocováni několika různými způsoby. Mezi nejčastěji používané metody patří simulace celého obvodu a přístupy využívající převod na reprezentaci logickými rovnicemi a následným řešením problému SAT nad nimi. Vhodnou metodu je třeba zvolit podle druhu vyvíjeného obvodu. Například pro základní jednoduché logické obvody bude simulace mnohem vhodnější než fitness funkce založené na řešení SAT problému, protože v druhém případě může práce potřebná k převodu obvodu na formuli v konjunktní normální formě a vlastní přípravu řešení SAT problému převýšit množství práce nutné k provedení jednoduché simulace. Jako výslednou hodnotu fitness funkce pak lze použít například počet kombinací hodnot primárních vstupů obvodu, pro které se na výstupech obvodu objeví správná odezva.

Evoluční algoritmy návrhu hardwaru nachází využití v mnoha oblastech. Velmi dobrých výsledků dosahují zejména v oblastech vývoje analogových filtrů, regulátorů, číslicových obrazových filtrů, logické obvody, optické systémy a komunikační protokoly. V těchto odvětvích se podařilo úspěšně prokázat, že konkrétní realizace mohou dosahovat lepších výsledků (podle požadovaných kritérií), než nejlepší dosud známé realizace systémů navržených klasickým způsobem [12].

Evoluci hardwaru lze rozdělit podle dvou hlavních použití na evolucí navržený hardware a na hardware, který je schopný dynamicky měnit svoji funkcionalitu bez jakékoliv interakce se svým okolím. K tomu je třeba využít rekonfigurovatelné hardwarové prostředky jako jsou FPGA čipy. Tento přístup spojuje dohromady využití právě rekonfigurovatelného hardwaru, umělé inteligence, autonomních systémů a tolerance chyb [3]. Využití rekonfigurovatelných hardwarových prostředků při evoluci vyvíjejících se obvodů je obvykle výrazně rychlejší než simulace obvodů.

Hardware dynamicky měnící svoji funkci může být použit například v systémech schopných obnovení vlastní funkce v případě poruchy. Pokud dojde k poruše, pak funkcionalitu komponenty, která selhala, přebírá jiná část rekonfigurovatelného obvodu (například FPGA). Tento druh zařízení nachází využití v zařízeních, která jsou těžce dostupná pro

klasické způsoby řešení problému. Mezi zařízení neumožňující jednoduchý přístup za účelem vykonání údržby nebo opravy můžeme zařadit systémy pracující hluboko pod mořskou hladinou nebo ve vesmíru.

V kontextu této práce se evoluce využívá ke konstrukci logických obvodů realizující požadovanou funkci. Evoluce hledá správná logická hradla k sestavení obvodu, jejich vzájemné propojení a připojení primárních vstupů a výstupů obvodu. K tomu používá principy založené na genetickém programování. Konkrétně se zde využívá kartézské genetické programování, o kterém nalezneme více detailů v dalších kapitolách.

6.1 Principy evolučního návrhu HW

Evoluční návrh hardwaru se do značné míry spoléhá na náhodné procesy generování jedinců populace a jejich následných mutací. Cílem je dosažení obdoby Charlesem Darwinem představené evoluční teorie ve světě výpočetní techniky. Jedinci tvořící populaci jsou reprezentováni tzv. chromozomem. Chromozom obsahuje informace o propojení jednotlivých komponent vyvíjeného obvodu a informace o vlastních komponentách, například jejich vnitřní funkci. První verze chromozomu vzniká během generování počáteční populace.

Celý evoluční vývoj obsahuje následující kroky:

1. specifikace požadavků na výsledný obvod – nejprve je třeba specifikovat, jak se má výstupní obvod chovat, aby bylo možné správně vyčíslit fitness funkci budoucích jedinců populace a tím je ohodnotit,

2. vygeneruje se počáteční generace jedinců – celý chromozom popisující jejich vlastnosti se vygeneruje zcela náhodně (pseudonáhodně),

3. počáteční generace je ohodnocena fitness funkcí,

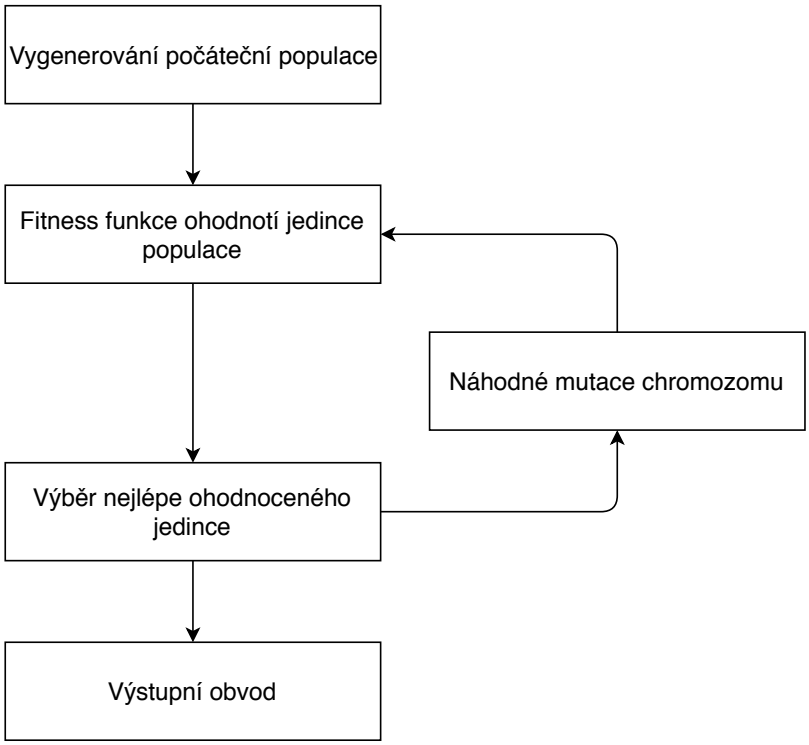
4. vybere se nejlepší jedinec z předchozí generace a pomocí náhodných mutací jeho chromozomu vznikne další generace. Chromozom nejlepšího jedince předchozí generace by měl být opět obsažen v nové generaci beze změny, aby se zaručil postup evoluce správným směrem. Pokud již byla dosažena maximální nebo uspokojivá hodnota fitness funkce, mohou se zde provádět evoluční optimalizace,

5. všichni jedinci nové generace jsou opět ohodnoceni fitness funkcí, chromozomu nejlepšího jedince z předchozí generace můžeme ponechat ohodnocení z předchozí generace, protože jeho hodnotu již známe, přispějeme tak efektivitě výpočtu omezením redundance výpočtů,

6. pokud byla dosažena maximální fitness funkce, nebo maximální počet generací evoluce algoritmus končí, jinak pokračuje krokem 4.

Z kroků evolučního algoritmu návrhu obvodu je zřejmý hlavní rozdíl mezi úpravami obvodu při klasickém návrhu, kdy se změny v obvodu provádí s nějakým konkrétním cílem a představou, jak by mělo výsledné řešení vypadat. Naproti tomu jakákoliv změna v obvodu při evolučním návrhu je náhodná a až následně ověřena její správnost fitness funkcí. Diagram

na obrázku 6.2 názorně ukazuje postup evolučních algoritmů při návrhu obvodů.



Obrázek 6.2: Kroky evolučního algoritmu.

V případě návrhu obvodů skládajících se z logických hradel lze některé hodnoty vypočítat předem například paralelním spuštěním simulace [12]. Při použití simulace však exponenciálně narůstá složitost ohodnocení jedinců populace. Pro složitost výpočtu tak platí s použitím tzv. óčkové notace:

$$O(fit(c)) = 2^n \tag{6.1}$$

kde *fit* označuje fitness funkci, c označuje ohodnocovaný chromozom a n značí počet primárních vstupů obvodu. Proto lze pro složitější obvody zavést testování fitness funkcí pouze na vybraném počtu vstupních kombinací. To ovšem přináší omezení přesnosti fitness funkce, protože nemůže ověřit všechny vstupní kombinace. To nevadí, pokud nevyžadujeme dokonalou funkčnost a tolerujeme nějakou chybu. Proto je zde vhodné použít jiné způsoby vyčíslení fitness funkce. Pokud se jedná o logický obvod může jej převést na matematickou formuli a následně zkoumat její splnitelnost pomocí SAT solverů. Ty zase na druhou stranu nejsou vhodné ke zpracování jednodušších obvodů, kde režie značně převyšuje náročnost vlastního výpočtu.

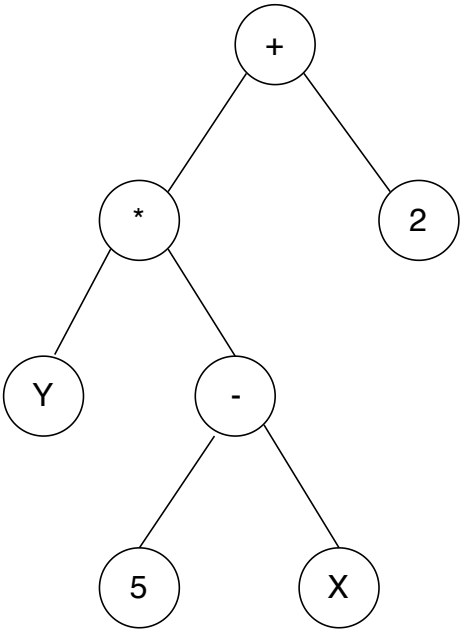
6.2 Klasické genetické programování

Pod pojmem genetické programování rozumíme techniku evolučního výpočtu, která řeší předem známý problém bez nutnosti, aby uživatel znal podrobnější informace o vnitřní struktuře zkoumaného systému [10]. Genetické programování lze využít jak při návrhu hardwaru tak i softwaru. Genetické programování používá k reprezentaci problému poměrně často stromové struktury. To však není podmínkou. Na jednoho ze zástupců používající odlišné struktury se podíváme v další kapitole věnované kartézskému genetickému programování. V případě použití genetického programování k návrhu softwaru, neterminální uzly (uzly, které nejsou listy stromu) reprezentují různé operace a listové uzly zastupují proměnné a konstanty (tedy data) použité v programu.

Mějme jednoduchý program řešící rovnici:

$$z = y \times (5 - x) + 2 \tag{6.2}$$

Příklad stromové struktury pro takový program najdeme na obrázku 6.3.



Obrázek 6.3: Ukázka stromové struktry reprezentující program řešící jednoduchou matematickou úlohu (úlohu z rovnice 6.2).

Pokud se používá stromová struktura řešení problému mutace mohou nahrazovat jednotlivé proměnné, operace nebo dokonce celé podstromy reprezentující nějaký podproblém. Cyklus tvorby nových generací aplikováním mutací na nejlepší jedince z předchozí generace vybrané pomocí fitness funkce a náhodné vygenerování populace počáteční plně koresponduje s principem popsaným v předchozí kapitole „Principy evolučního návrhu hardwaru“, avšak zde se aplikuje i na jiné problémy než pouze na návrh hardwaru.

6.3 CGP – kartézské genetické programování

Díky své schopnosti řešit grafové úlohy se tento přístup hodí zejména při návrhu vnitřní struktury a logických obvodů a zapojení jednotlivých hradel v tomto obvodu obsažených. Jedná se o variantu, kde jsou jedinci populace (též kandidátní řešení) reprezentováni orientovanými grafy. Kandidátní řešení jsou modelována jako obvykle dvojrozměrná pole (matice) programovatelných elementů (uzlů grafu označující logická hradla) o velikosti m n (počet sloupců počet řádků) [12]. Každé logické hradlo uvnitř obvodu může mít 0 až i vstupů. Každé hradlo zároveň představuje právě jednu logickou funkci. Ta se v rámci jedné generace (zároveň i jednoho jednice z populace) nemůže změnit, je tedy konstantní. Tato funkce je hradlu přiřazena náhodně z nějakého specifikovaného výběru možných funkcí. Vstupy každého hradla mohou být připojeny k primárním vstupů celého obvodu nebo k výstupům ostatních hradel, které však musí být v jednom z předchozích sloupců matice hradel. Počet sloupců, přes které lze připojit vstup hradla na výstup jiného, je omezen hodnotou s anglickým názvem „L-back“, dále značena jen L. Hodnota L nesmí být menší než 1 (nelze zapojit vstupy hradla k výstupu hradla ve stejném sloupci) a zároveň je omezena shora celkovým počtem sloupců matice hradel. Toto omezení neplatí pro výstupy celého obvodu, ty mohou být připojeny vždy k jakémukoliv hradlu nebo dokonce primárnímu vstupu obvodu. Následující příklad znázorňuje ukázku kartézského genetického programování.

Princip CGP si ukžeme na modelovém příkladu. Mějme logickou funkci *f*, kterou chceme realizovat novým specifickým logickým obvodem, který vytvoříme pomocí kartézského genetického programování. Funkce *f* má tři vstupní bity (primární vstupy) a dva bity výstupní. Vstupy jsou označeny písmeny *a*, *b*, *c* a výstupy písmeny *x*,*y*. Funkcionalitu funkce *f* pak lze popsat pravdivostní tabulkou 6.1.

a	b	c		x	y
1	1	1		0	0
1	1	0		1	0
1	0	1		1	0
1	0	0		0	0
0	1	1		1	1
0	1	0		1	1
0	0	1		0	1
0	0	0		0	1

Tabulka 6.1: Pravdivostní tabulka ukázkové logické funkce, pro kterou bude vytvořen kombinační obvod.

Evoluční algoritmus má k dispozici pouze hradla se dvěma vstupy tvořící mřížku (matici) hradel o rozměrech 3 × 3, hodnota „l-back“ je nastavena rovněž na hodnotu 3. Obrázek 6.4 ukazuje nejlepšího jedince po vytvoření několika generací a způsob jeho ohodnocení v případě, kdy je fitness funkce definována jako součet počtu správných odezev na vstupní kombinace pro jednotlivé výstupy. Lze ji tedy popsat rovnicí 6.3.

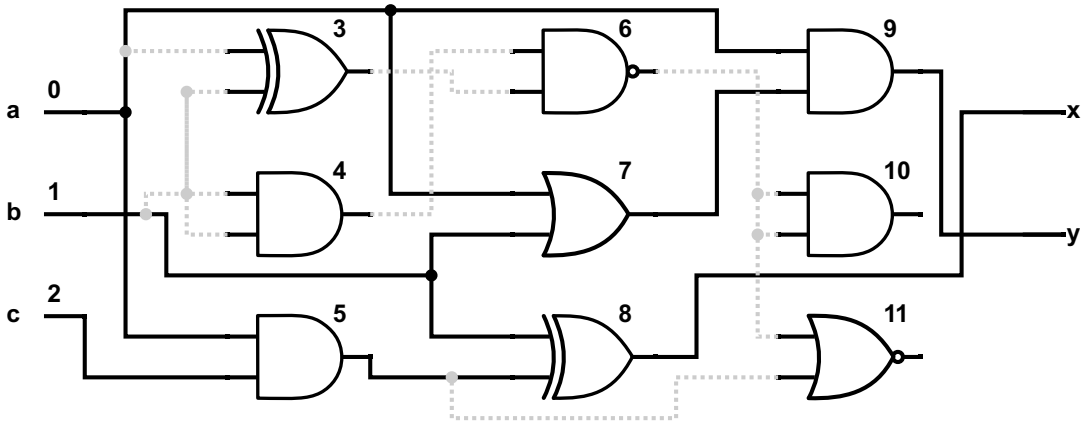
$$fitness(c) = \sum_{i=0}^{out\_cnt} \sum_{j=0}^{in\_comb\_cnt} 1 - \mid c(i,j) - r(i,j) \mid \tag{6.3}$$

kde *c* značí chromozom vygenerovaného jedince, *i* značí pořadí primárního výstupu obvodu, *j* značí index vstupní kombinace ve vektoru všech možných kombinací existující nad

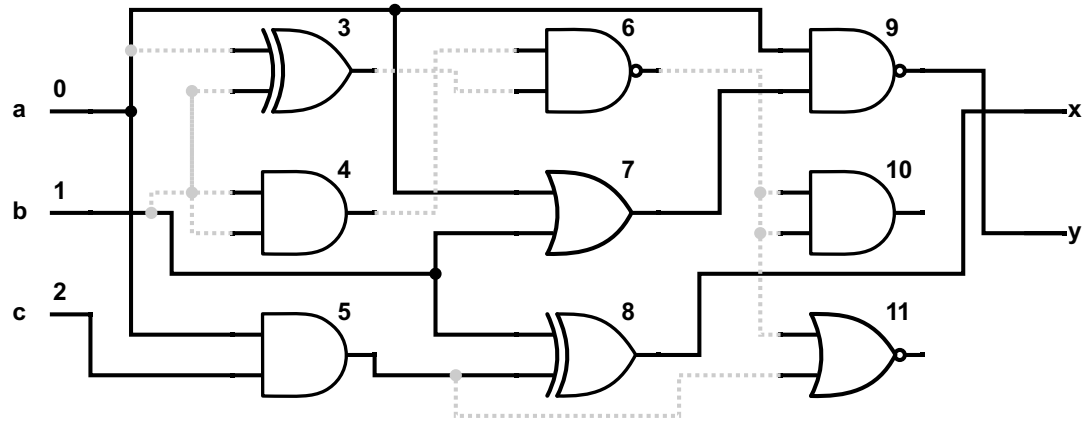
daným počtem bitů (primárních vstupů), *r(i,j)* značí korektní očekávanou odezvu na *i*-tém primárním výstupu na vstupní kombinaci pod indexem *j* . Podobně *c(i,j)* značí odezvu na *i*-tém výstupním bitu (primárním výstupu) vygenerovaného obvodu. Dále *out_cnt* značí počet primárních výstupů obvodu a *in_comb_cnt* označuje počet vstupních kombinací. Maximální hodnotu fitness funkce (dále značena *fit_max*) potom lze určit jako počet všech možných vstupních kombinací vynásobený počtem primárních výstupů obvodu. Platí tedy rovnice 6.4.

$$fit\_max = in\_cmob\_cnt \times out\_cnt \tag{6.4}$$

kde mají názvy proměnných stejný význam jako v předchozí rovnici číslo 6.3. Nyní berme v potaz jedince vzniklého během evoluce. Tento jedinec zatím neplní požadovanou funkci dokonale. Jeho podobu znázorňuje obrázek 6.4.



Obrázek 6.4: Jedinec nedosahující maximální fitness funkce, čísla vstupů a hradel označují identifikační číslo v chromozomu jedince.



Obrázek 6.5: Jedinec dosahující plného ohodnocení fitness funkcí.

Protože žádné hradlo nemá více než dva vstupy, budou nám stačit 3 čísla na uchování veškerých informací o jednom hradle (dva vstupy a jedno označení funkce). Pro tento příklad použijeme mapování funkcí na označení čísly následovně:

XOR	$\longrightarrow$	0
AND	$\longrightarrow$	1
NAND	$\longrightarrow$	2
OR	$\longrightarrow$	3
NOR	$\longrightarrow$	4

Tabulka 6.2: Ukázkové mapování logických funkcí na číselnou reprezentaci.

Dále je třeba uchovat informace o zapojení výstupů. Ty budou seřazeny na konci chromozomu. Chromozom tohoto jedince bude tedy vypadat následovně:

0,1,0|1,1,1|0,2,1|3,4,2|0,2,3|1,5,0|0,7,1|6,6,1|6,5,4|8,9.

Pravdivostní tabulka jedince s tímto chromozomem bude:

a	b	c		x	y	x'	y'
1	1	1		0	1	0	0
1	1	0		1	1	1	0
1	0	1		1	1	1	0
1	0	0		0	1	0	0
0	1	1		1	0	1	1
0	1	0		1	0	1	1
0	0	1		0	0	0	1
0	0	0		0	0	0	1

Tabulka 6.3: Pravdivostní tabulka nedokonalého kandidátního obvodu a její porovnání se správnými hodnotami označenými *x'* a *y'*.

V tabulce 6.3 jsou označeny primární výstupy generovaného obvodu označeny x, y a očekávané korektní hodnoty jako x', y'. Z tabulky je zřejmé, že hodnota prvního výstupu (x) je vždy správná, ovšem hodnota výstupu druhého (y) je opačná pro všechny vstupní kombinace. Proto bude hodnota fitness funkce tohoto jedince rovna 8 z maximální možné hodnoty 16. Proto se pokračuje další generací, která vznikne provedením mutace nad předchozím chromozomem předchozího jednice. Předpokládejme, že se podaří mutaci provést na správném místě a vznikne nový jedinec se strukturou vyobrazenou na obrázku 6.5.

V tomto případě stačila pouze jedna základní změna v chromozomu, aby bylo dosaženo kýženého výsledku. Nový chromozom vznikl aplikováním následující mutace:

0,1,0|1,1,1|0,2,1|3,4,2|0,2,3|1,5,0|0,7,**1**|6,6,1|6,5,4|8,9

$\downarrow$

0,1,0|1,1,1|0,2,1|3,4,2|0,2,3|1,5,0|0,7,**2**|6,6,1|6,5,4|8,9.

Tato mutace nám zajistí, že se výstupy obvodu budou vždy rovnat výstupům očekávaným a ohodnocení fitness funkcí bude 16, čímž obvod dosáhne její maximální hodnoty. Nyní máme plně funkční obvod a zůstávají nám tedy dvě možnosti. Buď prohlásíme obvod za hotový a použijeme jej v této podobě, nebo budeme pokračovat v evoluci s cílem optimalizovat obvod podle požadovaných kriterií, například minimalizace použitého počtu logických hradel.

6.4 Evoluční návrh logických obvodů

Toto téma již bylo nastíněno v předchozích kapitolách, nicméně nyní se podíváme cíleně na využití evolučních algoritmů při návrhu logických obvodů. V případě návrhu logických obvodů se nabízí varianta kartézského genetického programování. Kartézské genetické programování totiž umožňuje přímo ovlivnit některé parametry navrhovaného obvodu. Například upravením počtu sloupců mřížky logických hradel můžeme upravovat maximální dobu odezvy obvodu nebo efektivně optimalizovat výsledný obvod co se rozměrů a počtu použitých hradel týče. Dále lze jednoduše omezit výběr funkcí, která mohou hradla realizovat. V krajním případě můžeme navrhovat obvody složené pouze z hradel jednoho druhu, protože jak víme, například pomocí operace NAND lze popsat jakoukoliv logickou funkci. Příklad použití evoluce k návrhu logického kombinačního obvodu najdeme v předchozí kapitole.

Pro evolučními algoritmy vznikající obvody je nutná správná fitness funkce. Zde můžeme použít opět fitness funkci z příkladu v předchozí kapitole (výsledná hodnota fitness funkce je rovna součtu vstupních kombinací, vyvolávající korektní odezvu na každém primárním výstupu obvodu). Aby bylo možné dostatečně rychle generovat správné kombinační obvody, musí fitness funkce být realizace fitness funkce dostatečně efektivní. Jednou z možností je použití simulace. Avšak k vyčíslení fitness funkce lze použít také jiné metody, například převod obvodů na logické matematické formule a následně řešit jejich splnitelnost.

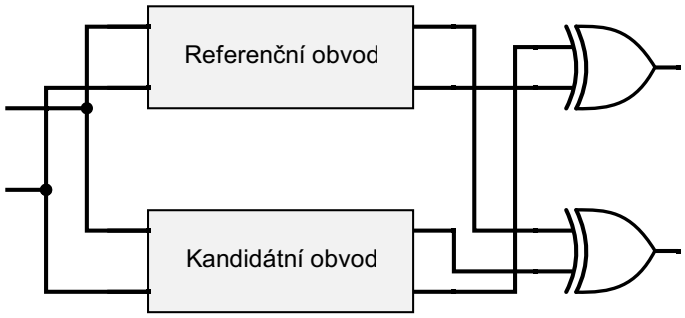
imulace je vysoce efektivní pro relativně jednoduché obvody s ne příliš velkým počet primárních vstupů. S každým dalším přidaným primárním vstupem roste časová náročnost výpočtu simulace exponenciálně. Kromě toho simulace musí postupně procházet všechny prvky simulovaného obvodu směrem od vstupů k výstupu a propočítávat hodnotu jejich výstupu. Tento postup se opakuje pro každou kombinaci, která se může objevit na primárních vstupech obvodu. Takových kombinací je zpravidla 2k, kde k označuje počet primárních vstupů obvodu. Exponenciální složitost se základem 2 vyplývá ze skutečnosti, že každý ze vstupů obvodu může nabývat dvou hodnot, logické 1 a 0.

Fitness funkci založenou na simulaci lze zrychlit zavedením paralelní simulace. Při reálném nasazení však není možné přidávat paralelně zpracovávané procesy exponenciálním tempem, aby se výkonnost vyrovnala rostoucímu počtu vstupů obvodu. Proto lze simulaci použít především pro jednoduché logické obvody. Simulace může být použita i při ohodnocování složitějších obvodů. Ty však musí být složité pouze svojí vnitřní strukturou, nikoliv počtem primárních vstupů obvodu. Ten musí zůstat relativně nízký.

Další možností je upustit od testování kandidátního obvodu na všechny existující vstupní kombinace a použít pouze vektor obsahující reprezentativní vzorek. Tímto se vyhodnocení simulace a tedy i fitness funkce výrazně zrychlí na úkor přesnosti. Protože fitness funkce již neověřuje všechny vstupní kombinace, může se zde vyskytnout chyba ve výsledném řešení. Tím je využití této optimalizace omezeno na případy, kdy jsme schopní nějakou míru chyb tolerovat. Omezení úplné správnosti výsledných obvodů omezuje rozsah jejich použití. Jednou z oblastí, kde takové obvody nachází své využití, je návrh obrazových filtrů [12]. Evoluční algoritmus s využitím simulace jako fitness funkce byl mimo jiné implementován v programové části této práce. O výstupech a efektivitě této implementace se dozvíme v kapitole 8 zabývající se dosaženými výsledky.

Kromě simulace můžeme použít metodou převodu problému na jiný matematický problém a ten následně vyřešit. Zde se nabízí problém SAT. Reprezentace fitness funkce řešením problému SAT sebou nese nutnost převádět ohodnocovaný obvod na logickou formuli, u které lze zkoumat její splnitelnost. Proto nestačí pouze převést kombinační obvod do podoby logické formule, ale musí se doplnit porovnání všech výstupních bitů obvodu s hod-

notami primárních výstupů referenčního řešení. Hledání rozdílu mezi výstupy vyvíjeného obvodu a referenčního řešení lze realizovat například zapojením korespondujících výstupních bitů do hradla s logickou funkcí XOR a následně testovat výsledek operace XOR na hodnotu 1. Hodnota bude kladná právě tehdy, když obvod ohodnocovaný fitness funkcí bude poskytovat špatnou odezvu při dané vstupní kombinaci. Takové zapojení znázorňuje obrázek 6.6.



Obrázek 6.6: Jednoduché zapojení referenčího a kandidátního řešení s detekcí chyb.

Kapitola 7

Implemetace

Tato kapitola se podrobně zabývá implementovaným řešením evolučního návrhu kombinačních obvodů. Popisuje využité algoritmy, prostředky, knihovny a metody použité při implementaci řešení. Obsahuje také popis úprav použitých již existujících řešení. Dále se zaměřuje na popis všech tří implementovaných algoritmů. A to evoluce s využitím simulace a dvou řešení využívajících sharSAT solver.

Z důvodu potřebné vysoké rychlosti, efektivity a kompatibility s použitými knihovnami (ty musí být také dostatečně efektivní) byl jako implementační jazyk zvolen c++. Kromě standardních knihoven jazyka c a c++ byly použity knihovny gmp a zlib. Knihovna gmp slouží pro práci s velkými čísly, pro která by běžné datové typy neposkytovaly dostatečně velké množství paměťových prostředků. Tato knihovna je využita pouze v rámci SAT solveru sharpSAT [16], k ukládání počtu ohodnocení, pro která je vstupní formule splnitelná. Naopak knihovna zlib se používá pouze v přípravné fázi evolučního algoritmu k dekompresi vstupního referenčního obvodu. Implementace dále využívá programu cmake při překladu programu, připojení knihoven. Kromě již zmíněných prostředků implementované řešení přebírá části kódu z implementace kartézského genetického algoritmu užitého na prvních cvičeních předmětu BIN (biologií inspirované počítače) v akademickém roce 2016/2017. Poslední použitou „knihovnou“ je rozhraní Ipasir určené ke sjednocení a zobecnění rozhraní různých implementací SAT solverů do uniformní podoby. Toto rozhraní se využívá pouze v případě dvou řešení ze tří, která používájí k výpočtu fitness funkce knihovnu sharpSAT. Tato knihovna však musela být do jisté míry upravena, aby ji bylo možné používat k tomuto specifickému účelu efektivně, a aby při takovém použití poskytovala správné výsledky.

Implementovaná řešení realizují evoluční návrh kombinačních obvodů sestávajících se z logických hradel. Jako prostředek evoluce se používá kartézské genetické programování a logická hradla jsou tedy umístě na do mřížky (dvourozměrného pole).

7.1 Použité knihovny a jejich modifikace

Nejdůležitější roli mezi knihovnami použitými v programové části této práce hraje sharpSAT, coby nástroj vypočítávání hodnoty fitness funkce. Také zde nalezneme modifikace provedené v knihovně sharpSAT. Dále se podíváme na rozhraní pro inkrementální SAT solvery ipasir a knihovny gmp a zlib.

7.1.1 Rozhraní Ipasir

Ipasir je rozhraní implementované v jazyce c++. Ipasir poskytuje jednotné rozhraní pro různé SAT solvery s podporou inkrementálního řešení problému SAT. Ipasir poskytuje kompletní spektrum funkcí potřebných pro běžnou práci se SAT solvery počínaje funkcí inicializace SAT solveru (nebo vytvoření jeho instance), přidání klauzulí formule, zadání předpokladů, řešení problému SAT nad zadanou formulí, vyčtení výsledku metody SAT solveru realizující výpočet řešení až po metodu rušící instanci SAT solveru. Formule se zadává postupně po jedné proměnné označené číslem, kde jsou negace proměnných identifikovány jako negativní hodnoty čísel proměnných. Protože se formule (v konjunktní normální formě) zadává po jedné proměnné, musí zde existovat způsob, jak rozdělit klauzule. V tomto případě konce klauzulí označují „proměnné“ s číslem 0. Tomuto přístupu musela být také upravena implementace knihovny sharpSAT, protože původní implementace počítá s načtením formule ze souboru ve formátu DIMACS. Tvůrci tohoto rozhraní již poskytují upravené implementace solverů MiniSAT, PicoSAT a Lawa.

7.1.2 SharpSAT

SharpSAT je SAT solver řešící problém počítání všech různých ohodnocení proměnných formulí takových, že je celá formule splnitelná. Tato knihovna je implementována v jazyce c++. Knihovna ve své původní podobě počítá s formulí uloženou v textovém souboru ve formátu DIMACS. Tento formát poskytuje předem definovaný počet klauzulí a proměnných podílejících se na konstrukci dané formule. Formule uložená v souboru musí být již z povahy tohoto formátu v konjunktní normální formě.

Formát Dimacs

Obsah souboru v tomto formátu(DIMACS) může pro naprosto základní a jednoduchou formuli skládající se ze čtyř proměnných a dvou klauzulí vypadat následovně:

```
c Komentář
c Tohle je také komentář
p cnf 4 2
1 -2 3 0
2 4 -3 0
```

Tento formát umožňuje zapisovat komentáře jako řádky začínající písmenem malé „c“. Takový soubor musí dále obsahovat řádek se specifikacemi podoby formule v konjunktní normální formě. Takový řádek musí být uvozen malým písmenem „p“. Dále musí následovat slovo „cnf“, číslo určující počet proměnných a nakonec číslo určující počet klauzulí formule. Všechny tyto hodnoty jsou odděleny mezerami. Následuje záznam vlastní formule zapsané způsobem jedna formule na jeden řádek. Proměnné jsou odděleny opět mezerami, negace značí záporná čísla proměnných a každá klauzule je ukončena číslem 0.

Algoritmus řešení SAT problému

Algorimuts užitý v knihovně SharpSAT se zakládá na klasickém algoritmu DPLL popsaném dříve. Kromě toho však přináší prvky známe z algoritmu CDCL a zavádí algoritmus autory knihovny nazvaný BCP z anglického „boolean constraint propagation“. BCP výrazně

ovlivňuje rychlost SAT a SAT solverů s použitím propagace jednotkových klauzulí [16]. SharpSAT používá také algoritmus hledání chybných literálů. Pokud tento algoritmus nalezne chybný literál, ihned se vytvoří a naučí konfliktní klauzule a negace literálu se použije jako další proměnná pro změnu větve v prohledávacím prostoru ohodnocení proměnných.

Úpravy provedené v knihovně SharpSAT

Knihovna ve své originální formě není ideální pro použití při výpočtu fitness funkce při evolučním návrhu. Hlavními problémy jsou načítání vstupní formule a správa paměti. Přístup k souborům byl upraven také co se výstupu knihovny sharpSAT týče. Další úpravy se týkaly pouze možnosti přečíst statistiky řešení splnitelnosti formule pomocí rozhraní Ipasir.

Knihovna SharpSAT počítá s formulí v konjkunktní normální formě načtenou ze souboru. Takový přístup je z hlediska rychlosti a efektivity neudržitelný v případě, kdy se řešení SAT problému musí nalézt co nejrychleji a mnohokrát opakovaně. Evoluční návrh kombinačních obvodů lze považovat za typický algoritmus, který požaduje právě tyto vlastnosti. Proto byla implementována modifikace přijímající formuli klasickými programovými prostředky, konkrétně jako parametry volaných funkcí. Přesnou podobu předávání formule popisuje rozhraní Ipasir (popsané v dřívějších kapitolách). Protože toto rozhraní zapisuje postupně jednotlivé proměnné jednu po druhé, musela být zavedena vyrovnávací paměť uchovávající formuli před provedením samotné inicializace instance solveru SharpSAT. Inicializace solveru potřebuje předem znát počet proměnných i počet klauzulí, aby mohla správně dynamicky alokovat paměť, kam se data formule ukládají. Dále musel být v inicializační metodě upraven výpočet velikosti alokované paměti pro ukládání literálů formule. V původní verzi je velikost tohoto vektoru stanovena jako velikost vstupního souboru v bytech a byla nahrazena počtem literálů tvořících formuli. Pro délku tohoto vektoru tedy platí následující rovnice 7.1.

$$l(l\_vec) =\mid f \mid \tag{7.1}$$

kde l značí délku vektoru pro ukládání literálů formule, l_vec označuje vektor, kam se literály formule ukládají a $|f|$ značí celkový počet literálů tvořících formuli.

Upravená metoda načítání vstupní formule nepotřebuje žádné čtení ze souborů na disku a dosahuje tak mnohem vyšší rychlosti. Tento efekt se dále zvýrazňuje opakovaným používáním knihovny SharpSAT.

Další modifikací se stala úprava správy paměti. Tyto úpravy lze rozdělit na dvě skupiny: oprava uvolňování dynamicky alokované paměti a úpravu užívání statických proměnných.

V rámci první skupiny úprav správy paměti byly vytvořeny chybějící destruktory a opraveno použití destruktoru, který byl použit v podobě vhodné k uvolnění paměti alokované pro jeden objekt namísto k uvolnění pole. Chybějící destruktory nezpůsobovaly znatelné úniky paměti při použití k ohodnocení formulí reprezentující jednoduché nepříliš velké logické obvody a při jednom spuštění řešení. Nicméně při použití na komplikované obvody, kde navíc evoluce prováděla mnoho generací, se neuvolněná paměť v průběhu několika desítek sekund začala pohybovat v řádech gigabytů. Tento problém se samozřejmě nevyskytuje v případě spuštění řešení SAT problému pouze jednou, protože neuvolněná paměť byla po dokončení jednoho běhu uvolněna jádrem operačního systému.

V případě úprav použití statických proměnných bylo třeba zamezit neoprávněným přístupům do paměti v případě opakovaného spuštění řešící metody knihovny SharpSAT. Zde nastával problém s alokací paměti při spuštění s formulí výrazně složitější než při běhu předchozím. V této situaci zůstaly hodnoty v některých statických proměnných uloženy z předchozího běhu a následně způsobovaly dynamickou alokaci paměti o špatné velikosti.

Úpravy umožňující čtení statistik přes rozhraní Ipasir jsou spíše kosmetické a zahrnují především mapování funkcí knihovny sharpSAT na funkce rozhraní Ipasir. Dále byly přidány funkce poskytující pokročilejší statistiky. Avšak tyto pokročilejší metody byly implementovány nad rámec rozhraní Ipasir a jsou použité pouze pro získání výsledných statistik a měření výkonu.

Zapisování do výstupních souborů bylo kompletně odebráno a nahrazeno statistikami popsanými v předchozím odstavci.

Poslední úpravou knihovny SharpSAT je zavedení možnosti ukončit výpočet ihned po nalezení specifikovaného počtu řešení a neprovádět výpočet celý. Tato úprava značně urychluje výpočet v situaci, kdy již bylo dosaženo maximální fitness funkce a postoupilo se do fáze optimalizace. Jelikož již existuje obvod splňující kompletně funkcionální požadavky, můžeme zastavit řešení SAT problému již po nalezení prvního formuli splňujícího řešení, protože takový obvod již nemůže předčit svého předchůdce.

7.2 Společná část implementace

Tato podkapitola je věnována společné části implementace. Tedy té části implementace, která se shoduje pro všechna tři implementovaná řešení. Protože různé varianty používají různé prostředky, bude každé z těchto variant věnována samostatná podkapitola. Do společné části implementace patří především načtení vstupního souboru s referenčním obvodem a informacemi o něm jako je počet primárních vstupů obvodu, počet primárních výstupů obvodu a seznam funkcí realizovatelný logickými hradly, vygenerování počáteční populace, provádění mutací mezi jednotlivými generacemi evoluce, počítání hradel použitých ke tvorbě daného obvodu a kontrola optimalizací obvodu po dosažení maximální hodnoty fitness funkce.

7.2.1 Referenční chromozom

Samotný referenční chromozom je reprezentován jako lineární pole hradel se dvěma vstupy a umožňují realizovat logické funkce AND2, NAND2, OR2, NOR2, XOR2, XNOR2, IDA (hradlo předá na svůj výstup nezměněnou hodnotu svého prvního vstupu), INV1 (hradlo realizující tuto funkci poskytuje jako svoji odezvu logickou negaci svého prvního vstupu), ZERO (výstup hradla vždy nabývá hodnoty logická 0 bez ohledu na vstupní hodnoty, ty nejsou použity) a ONE (výstup hradla je konstantní a zůstává v hodnotě logická 1). Tyto hodnoty jsou pevně mapovány na číselné reprezentace viz tabulka 7.1.

Funkce	IDA	AND2	OR2	XOR2	INVA
Číslelná reprezentace	0	1	2	3	4
Funkce	NAND2	NOR2	XNOR2	ZERO	ONE
Číslelná reprezentace	5	6	7	8	9

Tabulka 7.1: Mapování logiclkých funkcí na jejich číslené reprezentace. Toto mapování využívají implementované algoritmy.

Na pole uchovávající chromozom referenčního řešení je aplikována komprese. Proto se zde využívá knihovna zlib. Vstupní soubor musí dodržovat následující formát:

```
file: <volitelný popis souboru>
%i <seznam primárních vstupů>
%o <seznam primárních výstupů>
%f IDA,AND2,OR2,XOR2,INVA,NAND2,NOR2,XNOR2,ZERO,ONE
parametry obvodu (jedotlivá hradla ohraničená závorkami)(uspořádaný
seznam výstupnů obvodu)
```

Rozměry mřížky s hradly načtené ze souboru spolu s referenčním chromozomem se týkají především právě referenčního chromozomu. Tyto rozměry se aplikují na algoritmus kartézského genetického programování v evoluci pouze tehdy, když nejsou rozměry mřížky explicitně specifikovány při spuštění aplikace.

Referenční chromozom, stejně jako generované chromozomy používají k uložení každého hradla tři hodnoty označující číselná označení dvou vstupů a jedno číslo určující jakou funkci dané logické hradlo vykonává.

7.2.2 Specifikace parametrů evoluce

Implementace umožňuje upravovat některé parametry evoluce. Lze upravovat rozměry mřížky hradel, ve které mají vznikat nové obvody a počet generací. Ponechat evoluci stejné parametry, jaké má referenční chromozom není vhodné, protože výrazně omezuje možnosti vygenerovaného obvodu a zvyšuje pravděpodobnost,že výsledný obvod bude po všech stránkách totožný s referenčním. Zároveň omezený počet hradel snižuje rychlost konvergence evoluce ke správnému řešení. Dalším benefitem nastavení specifických rozměrů mřížky hradel je možnost upravit maximální čas odezvy celého obvodu. Možnost volby počtu generací samozřejmě nachází své užití při práci s různě složitými kombinačními obvody, kdy pro základní jednoduché obvody stačí řádově tisíce generací, což neplatí pro obvody více komplikované.

7.2.3 Generování počáteční populace

Při realizaci evolučního algoritmu vznikají jedinci nové generace vždy nějakým počtem mutací z nejlepších jedinců generace předchozí. První generace však nemá žádnou generaci předcházející a musí proto být vytvořena jiným způsobem. Počáteční generace se generuje zcela náhodně a je omezena pouze rozsahem hodnot, kterých může daný prvek chromozomu nabývat. Tato omezení tvoří hodnota „l-back“ udávající počet sloupců mřížky logických hradel, přes které lze zapojit vstup logického hradla k výstupu jiného hradla. Toto omezení neplatí pro zapojení primárních výstupů obvodu a vstupů jednotlivých hradel k primárním vstupům celého obvodu. Dalším omezením náhodně generovaných hodnot uvnitř chromozomu je omezení počtu funkcí, které může jedno hradlo reprezentovat. Kromě těchto dvou omezení již žádná další nejsou třeba, protože každé logické hradlo je popsáno trojicí čísel. Dvě označující vstupy a jedno označující funkci daného hradla. Čísla vstupů se vždy vyskytují i u logických hradel, jejichž funkce vstupní hodnoty nepoužívá ke tvorbě výsledné hodnoty. Během dalšího zpracování chromozomu je taková hradla nutné identifikovat právě podle jejich funkce a následně ignorovat jejich vstupy, protože se nepodílejí na výpočtu výsledku na primárních výstupech kombinačního obvodu.

Poté, co jsou náhodně vygenerováni všichni jedinci počáteční populace, jsou ohodnoceni fitness funkcí a následuje již klasický postup evolučních algoritmů tvořící další generace na základě nejlepších jedinců z předchozí generace.

7.2.4 Ohodnocení populace a výběr nejlepšího jedince

Tento krok se liší pro každé ze tří implementovaných variant. Proto je tomuto tématu věnována zvláštní kapitola v popisu jednotlivých implementovaných řešení. Společným rysem všech fitness funkcí je způsob, kterým přiřazují výslednou hodnotu zkoumaným chromozomům jedinců populace. Výsledné ohodnocení se vždy rovná rozdílu maximální možné hodnoty fitness funkce a počtu nalezených chyb. Platí tedy: čím vyšší hodnota fitness funkce, tím méně rozdílných hodnot na primárních výstupech a tím také lepší chromozom.

Ze všech jedinců populace se vybere vždy ten nejlepší a to již v momentu, kdy je ohodnocen. Nečeká se zde tedy na ohodnocení celé populace a následný výběr. Díky tomuto přístupu se jako nejlepší jedinec vybere v případě více jedinců se stejným ohodnocením vždy ten, který na tuto hodnotu dosáhl jako první. Nejlepší jedinec je vždy předán do nové generace i v čisté podobě bez mutací, proto v případě, kdy žádný z jedinců vzniklých mutací nedosáhne lepších výsledků než původní jedinec, zachová se právě ten původní. Předáním nejlepšího jedince se zabezpečení udržení evoluce správným směrem a nemůže tak dojít ke snížení kvality výsledků generovaných kandidátních řešení.

7.2.5 Aplikace mutací

oté, co proběhne ohodnocení celé populace v dané generaci a vybere se nejlepší jedinec, je třeba s použitím jeho chromozomu vytvořit generaci novou. Jako první jedinec se použije nijak nezměněný chromozom nejlepšího jedince (bližší popis se nachází v předchozí kapitole). Každý následující jedinec se vytvoří provedením náhodným počtem mutací v rozmezí od 1 do maximálního možného počtu mutací, který je buďto stanoven uživatelem při spuštění nebo se použije výchozí hodnota.

oté, co proběhne ohodnocení celé populace v dané generaci a vybere se nejlepší jedinec, je třeba s použitím jeho chromozomu vytvořit generaci novou. Jako první jedinec se použije nijak nezměněný chromozom nejlepšího jedince (bližší popis se nachází v předchozí kapitole). Každý následující jedinec se vytvoří provedením náhodným počtem mutací v rozmezí od 1 do maximálního možného počtu mutací, který je buďto stanoven uživatelem při spuštění nebo se použije výchozí hodnota.

Provedením mutací se postupně vytvoří celá nová generace. Ta se musí následně ohodnotit jednou z fitness funkcí v závislosti na variantě implementace. Detailní popis ohodnocení (jeho části společné pro všechny tři řešení) poskytuje předchozí kapitola.

7.2.6 Nalezení počtu použitých hradel

Informace o počtu hradel účastnících se konstrukce daného obvodu nabývá důležitosti teprve v okamžiku, kdy již byla dosažena nejvyšší možná hodnota fitness funkce a přistoupilo se k optimalizování obvodu.

Jako hradla použitá ke konstrukci obvodu můžeme označit všechna hradla podílející se na tvorbě hodnot primárních výstupů kombinačního obvodu. Proto algoritmus nejprve označí jako použitá hradla připojená přímo k primárním výstupům. Následně postupně označuje jako použitá hradla sloužící jako vstupy již aktivních hradel. Zde platí omezení, že jako použitá hradla mohou být označena pouze hradla, která jsou připojena ke vstupům již aktivních hradel a zároveň se hodnota daného vstupu hradla projevuje v jeho výstupní hodnotě. Nelze tedy počítat hradla sloužící jako vstupy jiných hradel vykonávající funkci ZERO (výstup vždy 0) nebo ONE (výstup vždy 1). Navíc v případě funkcí INA (na výstup se přesměruje hodnota prvního vstupu hradla) a INVA (na výstup se přesměruje

negace hodnoty prvního vstupu hradla) nelze započítat hradla připojená jako druhý vstup. Označování dalších hradel se opakuje směrem od primárních výstupů dokud se nenarazí na primární vstupy obvodu.

Výsledný počet použitých hradel se určí jako počet hradel označených jako použitých. Konečné spočtení se provádí až po ukončení průchodu mřížkou hradel označujícím použitá hradla. Takový přístup zamezuje započítání jednoho hradla vícekrát v případě, že hradlo slouží jako vstup dvou jiných hradel použitých v daném kombinačním obvodu.

7.2.7 Optimalizace výsledného obvodu

V okamžiku, kdy evoluce nalezne kandidátní řešení dosahující maximální možné fitness funkce, přikročí se k optimalizaci obvodu podle kritéria co možná nejmenšího použitého počtu hradel.

Optimalizace probíhá jako pokračování evoluce, ovšem kromě fitness funkce, která musí stále dosahovat maximální hodnoty, dochází ke spočtení hradel použitých ke konstrukci daného kandidátního řešení. K tomu se využije algoritmus popsaný v předchozí kapitole. Pokud se nalezne takové, které využívá méně logických hradel než předchozí nejlepší řešení, je označeno za nové nejlepší řešení.

Tento postup se opakuje dokud není dosažen maximální počet generací evoluce.

7.3 CGP s využitím simulace

První implementovanou variantou je metoda využívající simulaci jako fitness funkci. V této variantě se nepoužívá žádný prostředek na principu SAT problému. Veškeré ohodnocování chromozomů jedinců populace zde provádí simulace.

Jedinci populace se vygenerují pomocí algoritmu popsaného v kapitole zabývající se částí implementace společné pro všechna tři realizovaná řešení.

Před započetím samotného ohodnocení označí všechna hradla podílející se na hodnotách primárních výstupů zkoumaného jedince. Tyto informace později poslouží ke zvýšení efektivity samotné simulace, která už nebude muset simulovat všechna hradla v mřížce hradel používané kartézským algoritmem. Simulování pouze použitých hradel výrazně omezuje zpomalení výpočtu simulace v případě, kdy pro stejný obvod budeme zvětšovat dimenze mřížky logických hradel. Dříve než začne samotná simulace, musí se připravit hodnoty primárních vstupů, se kterými bude následná simulace pracovat.

Dnešní procesory se 64 bitovou architekturou umožňují simulovat 64 vstupních kombinací najednou, proto se simulace provádí po 64 kombinacích. Jednotlivé kombinace se ukládají do bitů na odpovídajících pozicích 64 bitového datového typu. To lze provést, protože se simulují kombinační obvody pracující pouze s binárními hodnotami, které lze uložit do jednoho bitu. V případě většího počtu kombinací než 64 (navrhovaný obvod má více než 6 vstupních bitů), dojde k rozdělení vstupních kombinací na bloky velikosti 64. Takto vzniklé bloky se simulují sériově jeden za druhým. V opačném případě, kdy je vstupních kombinací méně než 64, probíhá simulace identicky, pouze kontrola výsledných hodnot je omezena na počet bitů odpovídající počtu vstupních kombinací. Počet vstupních kombinací N dostaneme z rovnice 7.2.

$$N = 2^i \tag{7.2}$$

kde i značí počet primárních vstupů obvodu.

V okamžiku, kdy jsou připravena vstupní data a příslušná hradla označena jako aktivní, začne vlastní simulace. Simulace prochází mřížku hradel zleva doprava a v každém sloupci

provede simulaci aktivních hradel (provede logickou funkci realizovanou daným hradlem a uloží výsledek jako hodnotu jeho výstupu). Tímto postupem projde postupně celou mřížku hradel. Simulace se provede jak pro nově navrhovaný obvod tak pro obvod referenční. Nyní má algoritmus k dispozici hodnoty výstupů obou obvodů a může přikročit k jejich porovnání.

Porovnání výstupních hodnot referenčního obvodu a ohodnocovaného obvodu probíhá pro každý výstupní bit zvlášť. Nad výstupními hodnotami obou obvodů se provede operace exkluzivní disjunkce (XOR). Následně spočteme počet jedniček v jejím výsledku a tím získáme počet chybných bitů. Výsledný počet chybných bitů získáme jako součet chybných bitů ze všech primárních výstupů pro všechny 64 bitů dlouhé bloky vstupních dat.

Výsledná hodnota fitness funkce je rovna rozdílu maximální možné fitness funkce a počtu chybných bitů na výstupu obvodu. Platí pro ni rovnice 7.3.

$$fitness(c) = fitness\_max - poc\_err \tag{7.3}$$

kde *fitness_max* značí maximální hodnotu fitness funkce a *poc_err* značí počet chybných bitů na primárních výstupech obvodu. Maximální hodnotu fitness funkce *fitness_max* získáme výpočtem rovnice 7.4.

$$fitness\_max = 2^{in} \times out \tag{7.4}$$

kde *in* označuje počet primárních vstupů obvodu a *out* značí počet jeho primárních výstupů.

Celý postup ohodnocení se opakuje pro všechny nové jedince v populaci. Když jsou všichni ohodnoceni pokračuje se výběrem nejlepšího jedince a vytvoření nové generace pomocí mutací. Tento postup je popsán v kapitole určené části implementace společné pro všechny varianty řešení.

7.4 CGP s využitím problému #SAT

Dvě varianty implementovaného řešení se zakládají na metodě převodu logického obvodu na matematickou formuli a následné zkoumání její splnitelnosti. Aby bylo použitelné řešení založené na SAT problému, musí být obvod převeden na formuli v konjunktní normální formě, k tomu slouží Tseitinova transformace. Protože se zde využívá problém SAT již není třeba specifikovat kombinace logických hodnot primárních vstupů obvodu. Tím odpadá úkol přípravy vstupních dat. Na druhou stranu se musí řešit správné přiřazení identifikačních čísel proměnných logickým hradlům tvořící zkoumaný obvod a obvod referenční. Tento problém řeší obě varianty odlišně a řešení bude tedy podrobně popsáno v dedikovaných kapitolách.

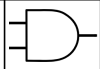
Implementované varianty se liší odlišným přístupem k vyhodnocení fitness funkce. Rozlišit je lze následovně:

1. základní porovnání – tato varianta řeší problém hledání počtu uspokojivých ohodnocení formule reprezentující vytvářený logický obvod pro každý jeho primární výstup

2. sjednocené porovnání – zde se všechny porovnávané výstupy spojují přes jednu logickou operaci disjunkce (OR). Řešení #SAT problému se tak spouští pouze jednou na ohodnocení jednoho obvodu.

Obě tyto metody využívají k řešení problému #SAT solver SharpSAT s úpravami popsanými v dřívějších kapitolách.

Logická hradla musí být převedena do reprezentace formulí v konjunktní normální formě. Proto bylo zavedeno mapování popsané na obrázku 7.1.

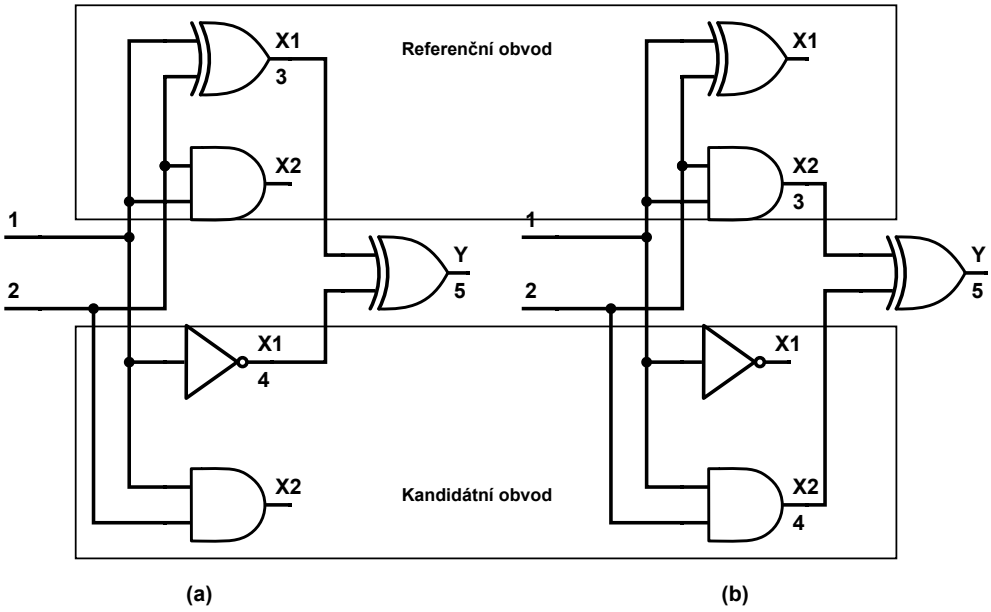
Název	Hradlo	Funkce	Ekvivalentní formule v CNF
not A		C = ¬A	(A ∨ C) ∧ (¬A ∨ ¬C)
A		C = A	(¬A ∨ C) ∧ (A ∨ ¬C)
AND		C = A & B	(¬A ∨ ¬B ∨ C) ∧ (A ∨ ¬C) ∧ (B ∨ ¬C)
NAND		C = ¬(A & B)	(¬A ∨ ¬B ∨ ¬C) ∧ (A ∨ C) ∧ (B ∨ C)
OR		C = A | B	(A ∨ B ∨ ¬C) ∧ (¬A ∨ C) ∧ (¬B ∨ C)
NOR		C = ¬(A | B)	(A ∨ B ∨ C) ∧ (¬A ∨ ¬C) ∧ (¬B ∨ ¬C)
XOR		C = (A ^ B)	(¬A ∨ ¬B ∨ ¬C) ∧ (A ∨ B ∨ ¬C) ∧ (A ∨ ¬B ∨ C) ∧ (¬A ∨ B ∨ C)
XNOR		C = ¬(A ^ B)	(¬A ∨ ¬B ∨ C) ∧ (A ∨ B ∨ C) ∧ (A ∨ ¬B ∨ ¬C) ∧ (¬A ∨ B ∨ ¬C)
0		C = 0	(¬C)
1		C = 1	(C)

Obrázek 7.1: Použité překódování hradel do formule v CNF.

7.4.1 Varianta základního porovnání

Tato varianta hledá počet vstupních kombinací, pro které se liší odezva nového a referenčního obvodu, postupně pro jednotlivé primární výstupy. To znamená, že se musí během ohodnocení jednoho kandidátního obvodu řešení SAT problému spustit tolikrát, kolik primárních výstupů má daný obvod.

Porovnání odezvy referenčního a nově vyvíjeného obvodu se realizuje zapojením korespondující výstupů do dalšího hradla jehož funkcí je exkluzivní disjunkce (XOR). Výsledné zapojení pro jednobitovou sčítačku bez přenosu a ne zcela správné kandidátní řešení pak bude zapojení během ohodnocování odpovídat vyobrazení na obrázku 7.2, kde a) ukazuje ohodnocení prvního a b) druhého primárního výstupu.



Obrázek 7.2: Zapojení hradla XOR pro porovnání referenčního a navrhovaného obvodu a)kontorola prvniho primárního výstupu, b) kontrola druheho primárního výstupu.

Při převodu obvodu, respektive jeho části využité pro daný výstupní bit, vyvstává nutnost správně volit číselné identifikace proměnných, protože problém SAT vyžaduje souvislou posloupnost čísel, aby mohl správně vyhodnocovat počet řešení. Kdyby došlo k nevyužití nějaké proměnné, počet řešení by se zdvojnásobil. Proto je zavedeno postupné číslování každé nově vytvořené proměnné. Tato posloupnost však nemůže začínat od nuly, neboť vstupy referenčního a ohodnocovaného obvodu jsou identické a musí tak mít pro oba obvody stejné číslo. Proto se začíná číslovat až od čísla o jedničku vyšší než je počet primárních vstupů obvodu a menší čísla jsou ponechána rezervovaná pro vstupy. Jak můžeme vidět na obrázku 7.2, číslování se liší podle výstupu, který se právě ohodnocuje. Čísla jsou přidělována směrem od primárního výstupu směrem ke vstupům pouze hradlům aktivním vzhledem ke zkoumanému výstupu. Zanedbání tohoto faktoru by opět znamenalo narušení správnosti fitness funkce (nekontrolovatelný nárůst počtu řešení). Výstup hradla XOR sloužícího k hledání rozdílných bitů, má vždy číslo nejvyšší. Protože se číslování liší pro různé výstupní bity, musí se přidělení čísel provádět dynamicky pro každý výstupní bit.

V případě tohoto zapojení se hodnota fitness funkce počítá identicky s verzí využívající simulaci. Počet řešení SAT problému pro každý výstupní bit odpovídá počtu vstupních kombinací, na které poskytl daný obvod špatnou odezvu na aktuálním výstupním bitu. Sečteme-li počet chyb pro všechny primární výstupy a odečteme tuto hodnotu od maximální hodnoty fitness funkce, dostaneme výsledné ohodnocení. Konkrétní zápis rovnic (7.4 a 7.3) viz předchozí kapitola zabývající se použitím simulace coby fitness funkce.

Po ohodnocení všech jedinců opět následují úkony spojené s tvorbou nové generace popsané v kapitole na téma společná část implementace.

Oproti řešení vyčíslující problém SAT nad všemi výstupními bity najednou získává výhodu toto jednoduché řešení ve větším rozsahu a přesnosti hodnot fitness funkce. Na druhou stranu se zde musí spustit řešení SAT problému tolikrát, kolik má vyvíjený obvod primárních výstupů oproti jednomu spuštění na celý obvod ve variantě druhé.

7.4.2 Varianta sjednoceného porovnávání

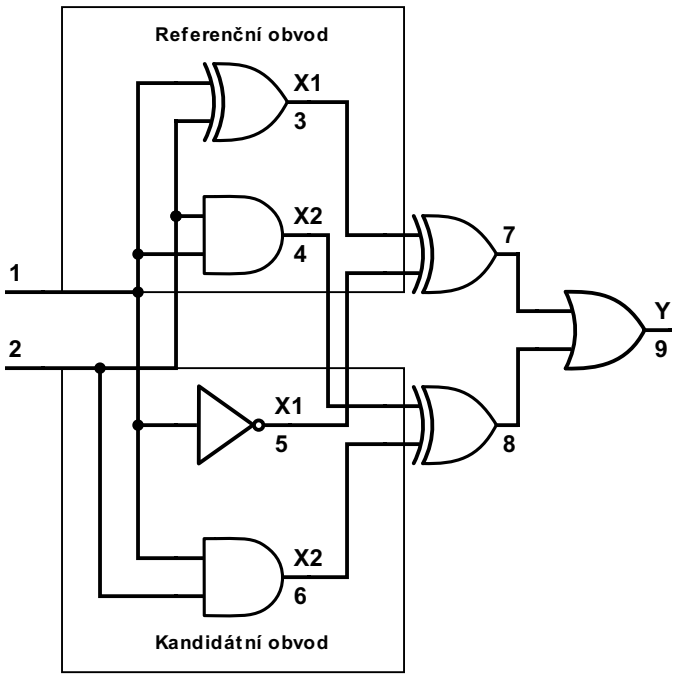
Tento přístup používá odlišný způsob hledání vstupních kombinací, na které ohodnocovaný obvod reaguje špatnou odezvou zjednodušuje na pouhé jedno spuštění řešení SAT problému na celé ohodnocení jednoho obvodu. Využívá se zde podobě jako u jednoduché varianty popsané v předchozí kapitole zapojení korespondujících primárních výstupů do přidaného hradla XOR.

Tato metoda hodnocení kandidátních řešení však zapojení výstupů neprovádí postupně, ale každá dvojice primárních výstupů dostane jedno nové hradlo XOR. Výstupy těchto přidaných hradel je následně nutné spojit jedním dalším novým hradlem, tentokrát hradlem OR. Poslední přidané hradlo má stejný počet vstupů jako zkoumaný obvod primárních výstupů. Zde si takový přístup můžeme dovolit, protože výstup tohoto hradla již nikdy není použit jako vstup dalšího hradla a tedy není nutné provádět převod do konjunktní normální formy pomocí Tseitinovy transformace. Místo toho stačí přidat do SAT solveru následující klauzuli:

$$y_1 \vee ... \vee y_n - 1$$

kde yX označuje výstup s indexem X a n označuje celkový počet primárních výstupů obvodu.

Zapojení všech dvojic výstupních bitů přes hradla XOR do jednoho hradla OR ukazuje obrázek 7.3.



Obrázek 7.3: Zapojení do společného OR hradla pro všechny výstupy.

Výsledkem spuštění řešení SAT problému nad takto vzniklou formulí poskytne rovnou počet vstupních kombinací, pro které je alespoň jeden výstupní bit špatný. Tuto hodnotu opět odečteme od maximální hodnoty fitness funkce, abychom dostali ohodnocení jednice. Maximální hodnota fitness funkce se však snižuje, neboť v momentu, kdy se špatná hodnota objeví na více než jednom výstupu obvodu, výsledkem bude stále pouze jedno řešení.

Maximální hodnota fitness funkce pak odpovídat rovnici 7.5.

$$fitness\_max = 2^{in} \tag{7.5}$$

kde *in* značí počet primárních vstupů navrhovaného obvodu.

Snížení rozsahu a přesnosti fitness funkce se může jevit jako nevýhoda ovlivňující rychlost, jakou evoluční algoritmus konverguje ke správnému řešení. Na druhou stranu se pro větší obvody dramaticky (tolikrát, kolik má navrhovaný obvod výstupů) snižuje počet spuštění řešení SAT problému, ovšem nad výrazně složitější formulí. Podrobný dopad rozdílných přístupů k ohodnocování kandidátních řešení nalezneme v kapitole dosažených výsledků.

Číslování proměnných reprezentující hradla obvodu se zde přidělují staticky. To snižuje náročnost výpočtu o nutnost kontrolovat, které hradlo již bylo aktivováno a byla mu přidělena proměnná. Nevýhodou tohoto přístupu je nutnost po přidání všech klauzulí reprezentujících referenční a evolučně navrhovaný obvod, přidat také všechna nepoužitá hradla z obou obvodů coby jednotkových klauzulí. Tím se zamezí možnosti měnit jejich hodnotu a tím také ovlivnit výslednou hodnotu fitness funkce. Ukázkové přidělení čísel proměnných znázorňuje obrázek 7.3.

Kapitola 8

Experimentální výsledky

Tato kapitola se zaměřuje na popis dosažených výsledků řešení. Výsledky jednotlivých řešení lze porovnat podle různých kritérií, jako je počet vstupních bitů obvodu nebo počet hradel realizující tato hradla. Nejprve budou představeny výsledky jednotlivých řešení, jejich výhody, nevýhody a nakonec jejich porovnání. Dosažené výsledky budou náležitě zdůvodněny.

Měření probíhalo na školních serverech s následujícími parametry spuštění:

- maximální počet generací – počet generací byl nastaven na hodnotu 1000000000, avšak tento počet nemusel být vždy dosazen, protože délka běhu evolučního algoritmu byla omezena,
- maximální doba běhu – tato hodnota omezuje maximální možnou dobu běhu evoluce. Hodnota byla nastavena na 3600 sekund, tedy jednu hodinu,
- velikost populace – hodnota tohoto parametru byla stanovena na 5 jedinců populace,
- všechny ostatní parametry byli určovány dynamicky pro jednotlivé testované obvody.

Výsledné hodnoty byly získány jako průměr hodnot naměřených během deseti spuštění.

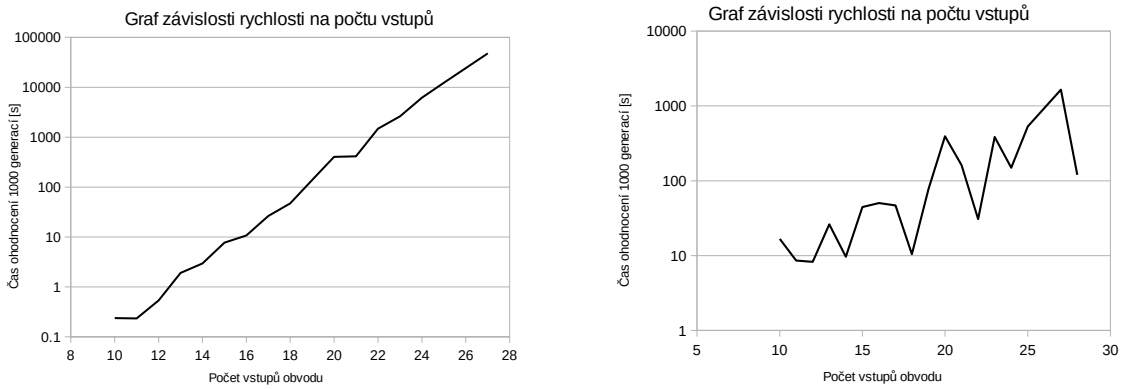
8.1 Závislost rychlosti na počtu vstupů

Prvním kritériem zkoumaným u implementovaných řešení je závislost rychlost hodnocení jedinců populace na počtu vstupních obvodů. Zabývat se těmito informacemi má význam především za účelem ukázat rozdíl, který přináší použití řešení SAT problému coby fitness funkce namísto simulace. Z grafů na obrázku 8.1 lze jednoduše vyvodit, že exponenciální složitost simulace všech vstupních kombinací obvodu, se již netýká zbylých dvou řešení využívající problém #SAT.

Doba nutná k ohodnocení stejného počtu generací sice roste také pro fitness funkce řešící SAT problém, nicméně tento růst je mnohem méně dramatický a je způsoben především větší složitostí vnitřní struktury obvodů zpracovávajících více vstupů. Nikoliv primárně počtem vstupů obvodu, nicméně složité obvody mají větší pravděpodobnost vyššího počtu vstupů. To ovšem není pravidlem a graf tak není příliš přesný.

8.2 Složitost obvodu a rychlost hodnocení populace

V případě ohodnocování obvodů s použitím SAT problému závisí potřebný čas především na vnitřní struktuře obvodu a logické funkci, kterou obvod realizuje. Proto bylo nutné



(a) (b)

Obrázek 8.1: Obrázek ukazuje dvojici grafů, odpovídající implementovaným řešením (se SAT problémem a bez něj). Graf řešení s využitím simulace se nachází v části obrázku a). V části obrázku b) nalezneme závislost času potřebného k ohodnocení populace s pomocí SAT problému, zde však počet vstupů nehraje příliš velkou roli. Zobrazené hodnoty odpovídají času ohodnocení 1000 generací.

zavést výpočet složitosti obvodu na základě struktury zapojení hradel uvnitř obvodu. Tento výpočet lze provést průchodem skrze zkoumaný obvod směrem od vstupů k výstupům. Během průchodu se postupně ohodnocují všechna hradla uvnitř obvodu podle jejich funkce a již spočtené složitosti hradel (nebo primárních vstupů obvodu) zapojených ke vstupům hradla aktuálního. Primárním vstupům obvodu se přiřadí hodnota na počátku výpočtu a bude vždy rovna 1. Podobný koncept můžeme nalézt v pracích „Digital Hamming Weight and Distance Analyzers for Binary Vectors and Matrices“ [13] a „Hardware and Software: Verification and Testing“ [1], ovšem ke zkoumání kompletně jiných než logických složitostí obvodu.cha

Pro složitost $sloz(h)$ každého hradla h platí rovnice 8.1.

$$sloz(h) = [sloz(in_1) + sloz(in_2)] \times w_i \tag{8.1}$$

kde h označuje aktuální hradlo, in_1 první vstup hradla, in_2 druhý vstup hradla a wh určuje váhu aktuálního hradla. Váha každého hradla závisí na jeho logické funkci, a platí pro ni tabulka 8.1.

Logická funkce	Váha
XOR, XNOR	4
INA, NEGA	1.3
AND, NAND, OR, NOR	1.9
0,1	1

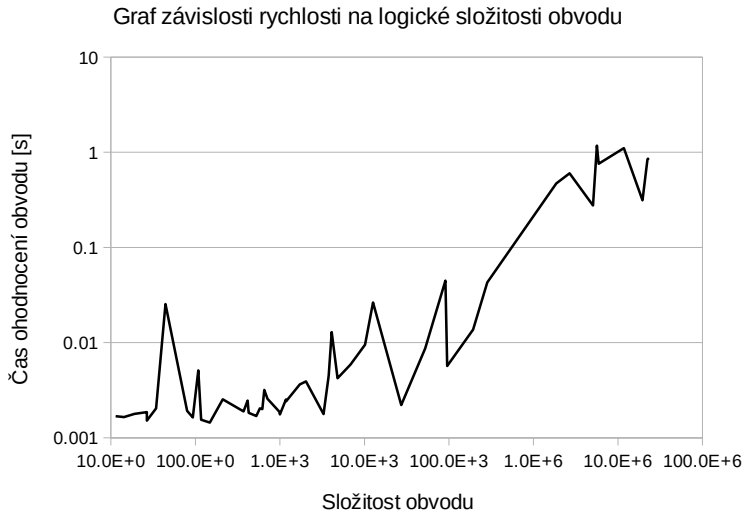
Tabulka 8.1: Váhy jednotlivých logických operací při výpočtu logické složitosti obvodu.

Tyto hodnoty v tabulce8.1 byly získány zkoumáním závislosti implementovaných řešení na vlastnostech obvodů a experimentováním s různými váhami logických funkcí. Právě tyto hodnoty nejlépe vystihují složitost hodnocení obvodů naměřenou při reálném použití na reálné logické obvody. Hradla XOR a XNOR ovlivňují dobu výpočtu nejvýrazněji, neboť rozdělují řešení podproblému pro jeho vstupní hodnoty na dvě symetrické větve.

Výslednou hodnotu pak určíme pro obě implementované varianty využívající SAT problém rozdílně. Pro variantu spouštějící řešení tohoto problému se logicky hodí součet složitostí všech hradel připojených k primárním výstupům obvodu. Pak pro složitost celého obvodu platí rovnice 8.2.

$$sloz(c) = \sum_{i=0}^{out\_cnt} sloz(h_i) \tag{8.2}$$

kde, c označuje aktuální logický obvod a h_i hradlo připojené k primárnímu výstupu obvodu na pozici i. Závislost naměřených časů hodnocení obvodu na jeho složitosti znázorňuje graf na obrázku 8.2.



Obrázek 8.2: Graf závislosti rychlosti porovnávání výstupů referenčního a kandidátního obvodu na logické složitosti obvodu. Tento graf platí pro řešení počítající každý výstup obvodu odděleně.

Pro variantu řešící problém SAT pro všechny primární výstupy obvodu najednou se více hodí výběr maximální hodnoty složitosti ze všech výstupů. Výslednou závislost můžeme pozorovat na obrázku 8.3.

Obrázek 8.3: Graf závislosti rychlosti porovnávání výstupů referenčního a kandidátního obvodu na logické složitosti obvodu. Tento graf platí pro řešení počítající všechny výstupy najednou.

Výběr správné hodnoty, respektive kombinace hodnot, z hodnot složitostí primárních výstupů byly opět zvoleny na základě analýzy reálného spuštění implementace na testovacích logických obvodech.

Čas výpočtu dále ovlivňuje například Hammingova vzdálenost kombinací na výstupech kandidátního a referenčního obvodu, protože v závislosti na ní dochází k různému počtu konfliktů během řešení #SAT problému. Dalším faktorem ovlivňujícím rychlost výpočtu je symetričnost logické funkce, kterou obvod realizuje. Další odchylky byly způsobeny nepřesností při měření způsobené vnějšími vlivy jako například proměnlivou rychlostí procesoru, atd. K odchylkám přispěl také fakt, že se všechny běhy evoluce pro různé soubory nespouštěly na procesorech s identickou architekturou.

Z grafů na obrazcích 8.2 a 8.3 vyplývá exponenciální složitost výpočtu SAT problému při ohodnocování logických obvodů v závislosti na složitosti jejich logické funkce. Prováděním experimentů byla ověřena teorie, že doba výpočtu nezávisí přímo na žádném základním parametru obvodu ani přímo na počtech hradel realizujících jednotlivé logické operace, nýbrž na kompletní vnitřní struktuře obvodu zohledňující jak funkci jednotlivých hradel, tak strukturu jejich vzájemného propojení. Časová složitost obou implementovaných řešení je již z podstaty exponenciální složitosti problému SAT také exponenciální vzhledem k náročnosti řešeného problému. Zde ovšem existuje nejhorší případ (pro řešení počítající SAT problém pro každý výstup obvodu), kdy toto neplatí a složitá struktura se bude muset řešit pro každý výstup obvodu. Proto je vhodné při použití k návrhu obvodu zvolit implementaci podle řešeného problému.

Ačkoliv je složitost řešení počítajícího problém závislá na součtu složitosti všech výstupů, což je vždy vyšší hodnota než pouze výběr maximální logické složitosti pro jednotlivé výstupy, řešení počítající problém #SAT dosahuje pro některé obvody lepších výsledků než řešení s pouze jedním spuštěním #SAT problému pro celý obvod. To se děje, protože pro jednotlivé výstupní bity mohou vzniknout méně komplikované problémy. V tomto případě roste složitost řešení exponenciálně pouze pro jednotlivé výstupní bity a pro jejich kombinaci již lineárně s počtem výstupů obvodu. Oproti tomu při aplikování #SAT problému na všechny výstupní bity současně, složitost roste stále exponenciálně. Na druhou stranu

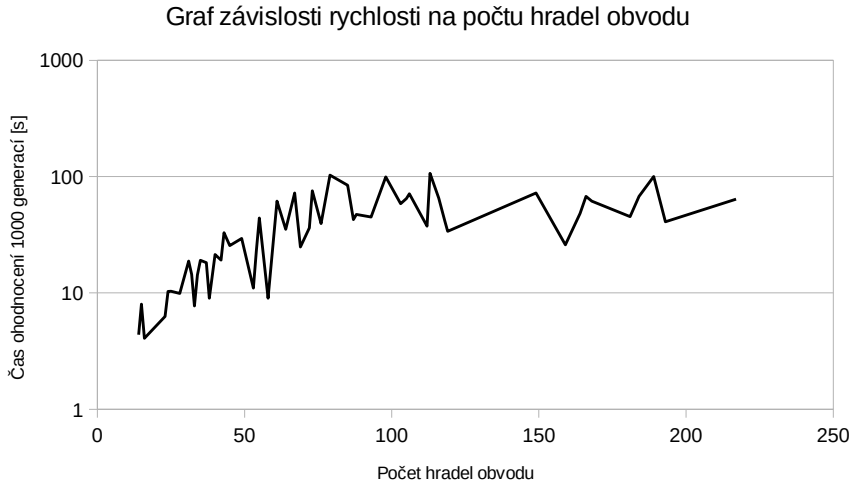
v momentu, kde se musí řešit složitý problém pro každý výstup, počítání každého výstupu zvlášť selhává.

koumat závislost rychlosti hodnocení kandidátních řešení pomocí simulace na logické složitosti obvodu nemá žádný potenciál, jelikož procesor během simulace provede všechny logické operace stejně rychle. Proto se složitost simulace zakládá především na počtu primárních vstupů, protože se musí odsimulovat všechny možné vstupní kombinace v počtu $2^{\text{počet vstupů}}$.

8.3 Dopad počtu hradel obvodu na rychlost

Z povahy implementace jednotlivých řešení tato vlastnost obvodu se v případě simulace prakticky neprojeví. Jediným řešením, kde lze spekulovat o závislosti rychlosti ohodnocování populace, je řešení, kde se problém #SAT řeší pro každý výstup zvlášť. To však není zapříčiněno přímou závislostí na této veličině, nýbrž stále logickou složitostí obvodu popsanou v předchozí podkapitole.

Při spuštění řešení #SAT problému se zde objevuje určitá tendence jen díky tomu, že se výpočet #SAT problému spustí pro každý výstup obvodu a počet hradel v čisté podobě se projeví více. Tyto tendence můžeme pozorovat na obrázku 8.4. Naopak při jednom spuštění pro celý obvod dojde pouze k jednomu spuštění, a tedy význam samotného počtu hradel ztrácí význam.



Obrázek 8.4: Graf dopadu počtu hradel obvodu na rychlost jeho ohodnocení fitness funkcí využívající #SAT problém.

8.4 Porovnání experimentálních výsledků

Tato podkapitola popisuje rozdíly mezi výsledky implementovaných řešení co se rychlosti týče.

Tabulka 8.2 obsahuje čas potřebný k ohodnocení jednoho tisíce generací o počtu pět jedinců. Tabulka znázorňuje údaje o rychlosti pro všechna tři implementovaná řešení.

Obvod	**Simulace**	**#SAT povýstupech**	**Jednotný #SAT**
x9dn-27-7-168x1-18l	51282.917	430.458	1034.03
10-adder-21-11-61x1-21l	479.446	676.104	315.026
s526-24-21-105x1-9l	6692.195	1362.347	192.305
cm150a-21-1-58x1-9l	349.003	9.006	8.115
newtpla-15-5-53x1-7l	5.461	54.975	7.893
mark1-20-19-193x1-9l	454.943	778.079	1116.51
cm151a-12-2-28x1-7l	0.422	19.804	4.783
b12-15-9-69x1-0l	6.618	48.867	17.155
parity-16-1-15x1-4l	5.778	5.159	6.115

Tabulka 8.2: Porovnání rychlosti všech tří řešešní pro vybrané testovací obvody.

Z tabulky 8.2 lze snadno odvodit, že pro malé málo náročné obvody dosahuje nejlepších výsledků právě simulace. Příčinou je zde převýšení náročnosti samotného výpočtu režií přípravy dat pro SharpSAT solver a následného řešení formule.

Pro velké komplikované obvody s vysokým počtem primárních vstupů naopak dominuje řešení jednoho #SAT problému pro všechny primární výstupy obvodu. Zde již simulace musí provádět výpočet pro velmi mnoho vstupních kombinací. Řešení počítající #SAT problém pro každý výstupní byt odděleně zde již také ztrácí na efektivitě. Je sice stále rychlejší než simulace, ale nedosahuje tak dobrých výsledků jako varianta řešící problém #SAT pouze jednou nad celým obvodem.

Rozhodnutí mezi efektivitami dvou variant založených na řešení #SAT problému není jednoduché. Zde se výrazně projevuje vnitřní struktura obvodu. Pokud dekompozicí obvodu na části ovlivňující vždy pouze jeden výstup dojde ke zjednodušení problému, který se musí řešit, pak dosahuje lepších výsledků metoda řešící #SAT problém pro jednotlivé výstupy obvodu. Toto pravidlo platí i opačným směrem, pokud tedy rozdělením podle primárních výstupů nedojde k výraznému zjednodušení dílčích problémů, pak se metoda řešení #SAT problému pro každý výstup stává výrazně pomalejší variantou.

Kapitola 9

Závěr

Cílem této práce bylo vytvořit studii na téma využití SAT solverů v oblasti optimalizace kombinačních obvodů, vytvořit implementaci evolučního návrhu těchto obvodů s využitím SAT solverů a zpracovat a vyhodnotit výsledky experimentů s implementovaným řešením. Do této problematiky spadá řešení splnitelnosti matematických logických formulí, generování, ohodnocení a následné mutace chromozomů během evolučního návrhu logických obvodů a principy převodu kombinačních obvodů na ekvivalentní logické formule. Dále se tato práce zabývá klasickými metodami simulování logických obvodů za účelem srovnání s výsledky hlavního cíle této práce. Bylo třeba zvolit vhodné prostředky k řešení jednotlivých částí zadaného problému.

Tato práce nejdříve seznámí čtenáře s nezbytnými teoretickými znalostmi k pochopení praktických částí této práce, metodami používanými moderními SAT solvery, použité technologie a knihovny. Následně tato práce popisuje vlastní implementované řešení. Nakonec se zaměřuje na dosažené výsledky, výsledky experimentů, a zhodnocení efektivity implementovaného řešení.

V rámci praktické části byly implementovány 3 aplikace realizující evoluční návrh kombinačních obvodů. Aplikace se liší přístupem k ohodnocování kandidátních řešení. Jedna aplikace využívá klasickou simulaci a byla implementována za účelem porovnání efektivit řešení využívajících k tomuto účelu SAT solvery. Jako SAT solver použitý v implementaci byl zvolen SharpSAT [16]. Algoritmus tedy nepoužívá běžný problém SAT, ale problém #SAT. Použitím #SAT problému bylo dosaženo nalezení počtu vstupních kombinací, pro které obvod poskytuje špatnou odezvu v jednom kroku. Tuto knihovnu však bylo třeba upravit, aby ji bylo možné používat v rámci jednoho programu opakovaně. Kromě toho byla spojena s rozhraním Ipasir. V tomto kroku bylo nutné implementovat podporu načítání formulí z jiných zdrojů než je soubor na disku v knihovně SharpSAT.

Dalším předmětem implementace byl převod kombinačního obvodu na ekvivalentní logickou formuli. K tomu byla použita Tseitinova transformace. Právě tato část implementace poskytuje vstupní data nově implementovanému rozhraní solveru SharpSAT.

Problém #SAT se využívá ve dvou implementacích lišících se zapojením výstupů referenčního a vznikajícího obvodu pro kontrolu správnosti navrhovaného obvodu. Jedna implementace provádí porovnání pro každý výstup obvodu odděleně a druhá pro všechny najednou. První varianta poskytuje přesnější hodnoty fitness funkce, avšak na úkor výkonu. Přidání inkrementálního režimu solveru ke knihovně SharpSAT nebyla implementována z důvodu nulového přínosu oproti verzi, kde se všechny výstupy kontrolují najednou.

Implementované řešení umožňuje používat princip evolučního návrhu v případě obvodů, kde kvůli příliš velkému počtu vstupních kombinací selhává simulace. Implementované ře-

šení pro větší a složitější obvody dosahuje výrazně lepších výsledků, než klasický přístup s využitím simulace.

Rychlost implementovaného řešení již není závislá na základních parametrech obvodu, ale na jeho vnitřní struktuře. Proto byla ve fázi vyhodnocování výsledků zavedena metoda počítání složitosti obvodu na základě jeho vnitřní struktury a logické funkce. Postupnou analýzou statistik ze spuštění evoluce nad množinou testovacích obvodů, byly nalezeny správné koeficienty pro výpočet náročnosti obvodu. Výsledky hledání této náročnosti a závislost rychlosti na nich je zdokumentována v kapitole výsledků experimentů.

Poslední kapitola rovněž obsahuje srovnání rychlosti a efektivity řešení založených na řešení #SAT problému oproti běžné simulaci. Z nich je patrné, že implementované řešení dominuje převážně pro velké obvody, protože u těch menších převáží režie nad vlastním výpočtem.

Literatura

[1] Bertacco, V.; Legay, A.: *Hardware and Software: Verification and Testing*. Springer, 2013, ISBN 9783319030777.

[2] Biere, A.: *HANDBOOK of satisfiability*. Washington, DC: IOS Press, 2009, ISBN 978-1-58603-929-5.

[3] Greenwood, G. W.; Tyrrell, A. M.: *Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems*. IEEE series on computational intelligence, Hoboken: Wiley Interscience, 2007, ISBN 9780470049716.

[4] Larrabee, T.: *Test pattern generation using Boolean satisfiability. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, ročník 11, 1 1992.

[5] Liang, J. H.; Ganesh, V.; Zulkoski, E.; aj.: *Understanding VSIDS Branching Heuristicsin Conflict-Driven Clause-Learning SAT Solvers*. [Online; navštíveno 20.05.2019].
URL https://arxiv.org/pdf/1506.08905.pdf

[6] M., W. M.; C., F. M.; Zhao, Y.; aj.: *Chaff: engineering an efficient SAT solver*. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, Las Vegas, NV, USA: IEEE, 6 1999.

[7] Marques-Silva, J.; Lynce, I.; Malik, S.: *HANDBOOK of satisfiability*, kapitola *Conflict-Driven Clause LearningSAT Solvers*. IOS Press, 2009.

[8] Marques-Silva, J.; Sakallah, K.: *GRASP: a search algorithm for propositional satisfiability. IEEE Transactions on Computers*, ročník 48, 5 1999.

[9] Mendelson, E.: *Introduction to Mathematical Logic*. Chapman Hall, 1997, ISBN 0412808307.

[10] Poli, R.; Langdon, W. B.; McPhee, N. F.: *A Field Guide toGenetic Programming*. S.l.: Lulu Press, 2008, ISBN 978-1-4092-0073-4.

[11] Raclavský, J.: *Úvod do logiky: klasická výroková logika*. Brno: Masarykova univerzita, 2015, ISBN 978-80-210-7790-4.

[12] Sekanina, L.: *Evoluční návrh hardware*. FIT VUT v Brně, [Online; navštíveno 20.05.2019].
URL http://www.fit.vutbr.cz/~sekanina/pubs.php?file=%2Fpub%2F9234%2Fsekanina.pdf&id=9234

[13] Skylarov, V.; Skliarova, I.: *Digital Hamming Weight and Distance Analyzers for Binary Vectors and Matrices. Internation Journal of Innovative Computingm Information and Control*, ročník 9, 12 2013.

[14] Sörensson, N.; Een, N.: *MiniSat v1.13 – A SAT Solver with Conflict-ClauseMinimization.* [Online; navštíveno 20.05.2019].
URL http://minisat.se/downloads/MiniSat_v1.13_short.pdf

[15] Tafertshofer, P.; Ganz, A.: *SAT Based ATPG Using Fast Justification and Propagation in the ImplicationGraph.* In *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers*, San Jose, CA, USA: IEEE, 11 1999.

[16] Thurley, M.: *sharpSAT - Counting Models with AdvancedComponent Caching and Implicit BCP.* [Online; navštíveno 20.05.2019].
URL https://a0e05165-a-62cb3a1a-s-sites.googlegroups.com/site/marcthurley/papers/sharpSAT06.pdf?attachauth=ANoY7cooUxcNiu_ta1oBy45-7aFZvOpBiDGqBl_xqudi36O3UMxHmfEqwee1pKhhYSVweoGN7p0vXUM83TP4ESgKA1cqh5M2PvTvwdgQwL5sn-zO79fNbyh8qKPNQ0p38E1A3uYwWTNaaJrUKg-DuikyvXUjF43hJ0u_QVQU2p6kukMbB_kOYvJXDXf40OpLgbt_4jp9ZSZuz9UCqN22_2z7tuQgxQD5heBNLP3D&attredirects=1

[17] Zhang, H.: *SATO: A Solver for Propositional Satisfiability.* [Online; navštíveno 20.05.2019].
URL http://homepage.divms.uiowa.edu/~hzhang/sato/

Příloha A

Obsah CD

Přiložené CD obsahuje položky:

- zdrojové kódy implementovaného řešení a návod k jeho použití ve složce `/src`,
- text této diplomové práce ve formátu pdf ve složce `/text`
- text práce v podobě zdrojových kódů ve formátu LaTeX ve složce `/text/src`
- výbrané testovací obvody ve složce `/test-data`