České vysoké učení technické v Praze
Fakulta elektrotechnická

Bakalářská práce

Nástroj pro manipulaci s logickými funkcemi popsanými algebraickým výrazem

David Toman

Vedoucí práce: ing. Petr Fišer

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Výpočetní technika

srpen 2007

Poděkování

Chtěl bych poděkovat především ing. Petru Fišerovi za rady a připomínky, které výrazně přispěly ke zkvalitnění této práce.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 8.8. 2007 ....

Abstract

The aim of this work is to implement a tool for manipulation with logical functions, that would allow performing operations such as modifications to varius forms (negation, DNF, CNF, ...), operations between particular functions (AND, OR, NOT, ...), minimalization of these functions and SAT solving. The primary aim is to perform operations as fast as possible even for large input data, which is achieved by minimalization during the computation. There are two types of formats that can be used for input. First of them is the PLA format and the other one is a format coming out of syntax of the VHDL language. Formats used for output are the same as those mentioned above and in addition also the BLIF format.

Abstrakt

Cílem této práce je implementace nástroje pro manipulaci s logickými funkcemi, který by umožňoval provádění operací jako jsou úpravy funkcí na různé tvary (negování, DNF, CNF, ...), logické operace mezi jednotlivými funkcemi (AND, OR, NOT, ...), minimalizaci těchto funkcí a také testy splnitelnosti. Důraz je kladen především na co nejrychlejší provádění operací pro rozsáhlá vstupní data, čehož je dosaženo minimalizací funkcí v průběhu výpočtu. Jako vstupní formát může být použit formát PLA, nebo formát vycházející ze syntaxe jazyka VHDL. Jako výstupní jsou použity oba již zmíněné formáty a navíc formát BLIF.

Obsah

Seznam obrázků xi

Seznam tabulek xiii

1 Úvod 1

1.1 Motivace a cíl práce . . . 1
1.2 Definice základních pojmů . . . 1

2 Specifikace cíle, rozbor problému 4

2.1 Požadavky na funkcionality programu . . . 4
2.2 Další specifické požadavky . . . 4
2.3 Vymezení stěžejních podproblémů . . . 4

3 Analýza a návrh řešení 5

3.1 Volba programovacího jazyka . . . 5
3.2 Způsob vnitřní reprezentace funkcí . . . 5
3.3 Návrh překladače pro načtení PLA formátu . . . 6
3.3.1 Analýza problému . . . 6
3.3.2 Návrh gramatiky . . . 7
3.4 Návrh překladače pro načtení VHDL formátu . . . 9
3.4.1 Analýza problému . . . 9
3.4.2 Návrh gramatiky . . . 10
3.5 Popis funkcí pro práci s výrazy ve vnitřní reprezentaci . . . 12
3.5.1 Negování funkcí . . . 12
3.5.2 Úprava funkcí do tvaru obsahujícího pouze základní operace . . . 13
3.5.3 Úprava funkcí do DNF a CNF . . . 15
3.5.4 Testování splnitelnosti včetně získání on-setu funkce . . . 15
3.5.5 Úprava do formy odpovídající síti NOR . . . 15
3.5.6 Úprava do formy odpovídající síti NAND . . . 17
3.5.7 Provedení libovolných logických operací mezi více funkcemi . . . 17
3.5.8 Načtení vstupních dat v požadovaném formátu (PLA, VHDL) . . . 17
3.5.9 Uložení výsledků v požadovaném formátu (PLA, VHDL, BLIF) . . . 17
3.5.10 Výpočet kofaktoru funkce . . . 17
3.6 Návrh efektivního algoritmu pro převod funkce do DNF a CNF . . . 18
3.6.1 Analýza složitosti řešení problému . . . 18
3.6.2 Popis činnosti převodního algoritmu . . . 18
3.6.3 Způsob minimalizace stromu výrazu . . . 19
3.6.4 Minimalizace ternárního stromu pomocí absorpce . . . 21

4 Realizace 22

4.1 Popis struktury pro reprezentaci stromu výrazu . . . 22
4.2 Popis struktury pro udržování informací proměnných . . . 23
4.3 Popis struktury zaobalující informace o jednotlivých výrazech . . . 23
4.4 Popis struktury pro reprezentaci ternárního stromu . . . 24
4.5 Popis interfacu programu . . . 24
4.6 Popis interfacu funkcí . . . 25

5 Testování 27

5.1 Testování převodu funkcí do DNF . . . 27

5.1.1 Test 1 27
5.1.2 Test 2 29
5.2 Testování minimalizace 30

6 Závěr **31**

7 Seznam literatury **33**

A Seznam použitých zkratek **35**

B Uživatelská příručka **37**
B.1 Překlad a spuštění programu 37
B.2 Ovládání programu 37
B.2.1 Interaktivní používání programu 37
B.2.2 Neinteraktivní používání programu 38

C Obsah přiloženého média **39**

Seznam obrázků

3.1 Strom ilustračního výrazu 5
3.2 Strom výrazu před negací 12
3.3 Strom výrazu po negaci 12
3.4 Strom před rozkladem na základní operace 14
3.5 Strom po rozkladu na základní operace 14
3.6 Ohodnocování proměnných výrazu v DNF 15
3.7 Strom výrazu před úpravou 16
3.8 Strom výrazu po úpravě 16
3.9 Příklad ternárního stromu 19
3.10 Ukázka způsobu slučování listů v ternárním stromě 19
3.11 Ternární strom po přesunutí kořene k listům 20
3.12 Ternární strom po spojení 20
3.13 Ukázka postupu absorpčního algoritmu 21

4.1 Typy uzlů použitých při implementaci 22

Seznam tabulek

3.1 Tabulka pravidel PLA . . . 7
3.2 Rozkladová tabulka PLA . . . 7
3.3 Tabulka atributů PLA . . . 8
3.4 Tabulka pravidel VHDL . . . 11
3.5 Rozkladová tabulka VHDL . . . 11
3.6 Tabulka atributů VHDL . . . 11

5.1 Přehled souborů použitých pro testování . . . 27
5.2 Naměřené hodnoty pro převod funkcí do DNF . . . 28
5.3 Časy převodu funkcí do DNF při rozdílné struktuře stromu výrazu . . . 29
5.4 Naměřené hodnoty pro srovnání minimalizačních algoritmů . . . 30

C.1 Obsah přiloženého média . . . 39

1 Úvod

1.1 Motivace a cíl práce

Základní motivací této práce bylo vytvořit co nejuniverzálnější nástroj pro práci s logickými funkcemi v různých tvarech, který by byl zároveň snadno rozšiřitelný, jelikož nic podobného zatím není k dispozici. Minimalizátorů funkcí a SAT solverů existuje sice celá řada, ale pokud potřebujeme znát například on-set funkce, která není přímo v DNF a je nutné získat exaktní řešení, nelze tyto prostředky obvykle použít. Nemáme pak jinou možnost než se pokusit provést převod na papíře, což ovšem vzhledem k exponenciální složitosti výpočtu není při větším počtu vstupních proměnných často ani realizovatelné. Podobný problém nastane, i pokud chceme obecně zapsanou funkci převést například do tvaru obsahující pouze operace NAND, což se často hodí pro realizaci logických obvodů.
Práce má tedy za cíl poskytnout uživateli celou řadu funkcionalit, jako je negování funkcí, spojování různých logických funkcí do větších celků, výpočet kofaktoru, testování splnitelnosti či minimalizace, aby měl uživatel k dispozici vše v jednom programu.

1.2 Definice základních pojmů

V této sekci jsou vysvětleny některé základní pojmy, které se vyskytují v dalším textu.

literál - proměnná logického systému nebo její negace - může nabývat pouze dvou hodnot (”0”, ”1”)

term - termy se často označují jako vektory booleovské funkce - složen z literálů mezi sebou navzájem vázaných logickým operátorem - podle druhu operátoru jej dělíme na:

- **součinový (P-term)** - neobsahuje operátor součtu ”+” př.: $x_1 * x_2 * \bar{x}_3$
- **součtový (S-term)** - neobsahuje operátor součinu ”*” př.: $x_1 + x_2 + \bar{x}_3$

vstupní písmeno - kombinace hodnot vstupních proměnných

minterm/maxterm - P-term resp. S-term obsahující všechny proměnné - je tvořen pouze nezávislými proměnnými

booleovská funkce - libovolný booleovský výraz obsahující proměnné x1, x2, ..., xn. - Každou booleovskou funkci lze vyjádřit pomocí logického součtu mintermů nebo logického součinu maxtermů. Každý minterm, resp. maxterm nabývá hodnoty ”1”, resp. ”0” právě pro jediné vstupní písmeno dané logické funkce. Dále jen *funkce.*

implikant logické funkce - Jedná se o výraz ve tvaru P-termu, pro který platí, že danou funkci implikuje, tzn. jestliže nabývá hodnoty ”1”, daná funkce nabývá též hodnoty ”1”. Implikant nazveme přímým implikantem právě tehdy, když po vypuštění libovolného literálu přestává být implikantem. Podstatný implikant je takový implikant, který je součástí každého minimálního řešení dané logické funkce.

disjunktivní normální forma - zkráceně DNF, booleovská funkce složená z disjunkcí P-termů. např.: $f = (x_1 * \bar{x}_2 * x_3) + (\bar{x}_1 * x_2 * \bar{x}_3) + (x_1 * \bar{x}_2 * \bar{x}_3)$

konjunktivní normální forma - zkráceně CNF, booleovská funkce složená z konjunkcí S-termů. např.: $f = (x_1 + \bar{x}_2 + x_3) * (\bar{x}_1 + x_2 + \bar{x}_3) * (x_1 + \bar{x}_2 + \bar{x}_3)$

ON-set - množina termů s hodnotou implikující výslednou funkci rovnou "1"

OFF-set - množina termů s hodnotou implikující výslednou funkci rovnou "0"

DC-set (don't care set) - termy, pro které výsledná funkce není specifikovaná

DC (don't care) - označuje, že na hodnotě dané proměnné nezáleží

základní operace - v tomto textu jsou tak označeny operace AND, OR a NOT

zákony Booleovy algebry

1. $a + b = b + a,\ a * b = b * a$ (komutativita)
2. $a + (b + c) = (a + b) + c,\ a * (b * c) = (a * b) * c$ (asociativita)
3. $a + (b * c) = (a + b) * (a + c),\ a * (b + c) = (a * b) + (a * c)$ (distributivita)
4. $a + 0 = a,\ a * 1 = a$ (neutralita 0 a 1)
5. $a + \bar{a} = 1,\ a * \bar{a} = 0$ (vlastnosti komplementu)
6. $a * 0 = 0,\ a + 1 = 1$ (agresivita 0 a 1)
7. $a * a = a,\ a + a = a$ (idempotence)
8. $\bar{\bar{a}} = a$ (dvojí negace)
9. $a + (a * b) = a,\ a * (a + b) = a$ (absorpce)
10. $a + (\bar{a} * b) = a + b,\ a * (\bar{a} + b) = a * b$ (absorpce negace)
11. $\overline{a + b} = \bar{a} * \bar{b},\ \overline{a * b} = \bar{a} + \bar{b}$ (de Morgan)
12. $(a * b) + (\bar{a} * c) + (b * c) = (a * b) + (\bar{a} * c),\ (a + b) * (\bar{a} + c) * (b + c) = (a + b) * (\bar{a} * c)$ (consensus)

logické operace

- $not\ a \Leftrightarrow \bar{a}$
- $a\ and\ b \Leftrightarrow a * b$
- $a\ or\ b \Leftrightarrow a + b$
- $a\ nand\ b \Leftrightarrow \overline{a * b}$
- $a\ nor\ b \Leftrightarrow \overline{a + b}$
- $a\ xor\ b \Leftrightarrow (a * \bar{b}) + (\bar{a} * b)$
- $a\ xnor\ b \Leftrightarrow \overline{(a * \bar{b}) + (\bar{a} * b)}$

formát PLA - formát souboru obsahující data logických funkcí ve tvaru součtu součinů

formát BLIF - formát souboru obsahující data v podobném tvaru jako PLA, který navíc umožňuje definování víceúrovňové logiky

SAT - problém splnitelnosti booleovské formule

LL(1) gramatika - gramatika v níž lze o přechodu jednoznačně rozhodnout na základě jednoho dopředu prohlíženého symbolu

lexikální element - sekvence znaků v analyzovaném kódu, která odpovídá nějakému definovanému vzoru

tabulka pravidel - udává pravidla, podle kterých se mají provádět jednotlivé přechody

rozkladová tabulka - udává čísla pravidel, která se mají použít pokud je na vstupu daný lexikální element

tabulka atributů - specifikuje dědičné (vstupní) a syntetizované (výstupní) atributy jednotlivých symbolů

2 Specifikace cíle, rozbor problému

V této kapitole budou rozebrány jednotlivé požadavky na práci a vymezeny řešené problémy.

2.1 Požadavky na funkcionality programu

Jak již bylo řečeno, jde u programu především o jeho maximální univerzálnost, proto byly vytyčeny následující požadavky na jeho funkcionality:

- Negování logické funkce
- Úprava funkce do formy obsahující pouze základní operace (AND, OR, NOT)
- Úprava funkce na CNF
- Úprava funkce na DNF
- Testování splnitelnosti funkce
- Úprava funkce do formy odpovídající síti NOR
- Úprava funkce do formy odpovídající síti NAND
- Provedení libovolných logických operací mezi více funkcemi
- Načtení vstupních dat v požadovaném formátu (PLA, VHDL)
- Uložení výsledků v požadovaném formátu (PLA, VHDL, BLIF)
- Výpočet kofaktoru funkce

2.2 Další specifické požadavky

Kromě požadavků na základní funkcionality byly stanoveny další specifické požadavky:

- Možnost interaktivního i neinteraktivního provádění operací
- Jednoduchý programový interface umožňující snadnou začlenitelnost do jiného kódu
- Snadná rozšiřitelnost programu o další funkce

2.3 Vymezení stěžejních podproblémů

Z požadavků vyplývá nutnost řešení následujících stěžejních podproblémů:

- Vhodný výběr programovacího jazyka z důvodu velké časové náročnosti některých výpočtů
- Zvolení vhodné vnitřní reprezentace funkcí pro jejich snadnou úpravu
- Návrh a implementace překladače umožňujícího načtení PLA souboru
- Návrh a implementace překladače umožňujícího načtení VHDL souboru
- Návrh a implementace funkcí pro práci s výrazy ve vnitřní reprezentaci
- Nalezení a implementace efektivního algoritmu pro převod funkcí do DNF a CNF

3 Analýza a návrh řešení

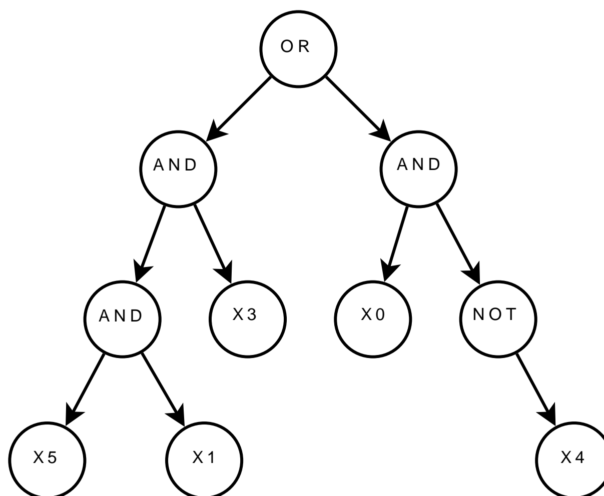
V této kapitola budou podrobně rozebrány problémy, které byly nastíněny v předchozí kapitole. Potřebná teorie byla načerpána ze zdrojů [1], [2], [7].

3.1 Volba programovacího jazyka

Při výběru programovacího jazyka hrála roli jak moje znalost daného jazyka, tak efektivita a rychlost provádění výsledného kódu. Proto pro implementaci systému připadaly v úvahu tři možnosti: Java, C a C++ . Implementace v jazyce Java byla zavrhnuta jako první, protože po prvotním rozboru problému a zjištění, že stěžejní výpočty budou mít až exponenciální složitost, bylo jasné, že je nutné usilovat o co nejmenší režii výpočtu a tu má Java oproti jiným programovacím jazykům příliš vysokou. Další možností byl tedy jazyk C, který by poskytl nejvyšší rychlost provádění kódu z uvedených variant, ale na druhou stranu to zase není příliš vhodný jazyk pro větší systémy, zvlášť co se týče snadné rozšiřitelnosti funkcí programu a přehlednosti kódu. Nakonec rozhodl způsob vnitřní reprezentace logických funkcí, protože vyžadoval objektový přístup a virtuální metody - volba tedy padla na C++.

3.2 Způsob vnitřní reprezentace funkcí

Pro reprezentaci logické funkce byla vybrána klasická stromová struktura, kde listy jsou tvořeny jednotlivými literály či logickými hodnotami a uzly jsou tvořeny operátory - binárními se dvěma potomky a unárními s jedním potomkem. Tato struktura umožní snadnou operaci se stromem výrazu pomocí dědičnosti a virtuálních metod, zároveň bude možné program snadno rozčlenit a napsat ho co nejobecněji i za cenu mírně větší režie výpočtu.
Pro ilustraci může posloužit třeba výraz $(x_5\ and\ x_1\ and\ x_3)\ or\ (x_0\ and\ not\ x_4)$, který by vypadal následovně:



Obrázek 3.1: Strom ilustračního výrazu

3.3 Návrh překladače pro načtení PLA formátu

3.3.1 Analýza problému

Cílem je navrhnout překladač umožnující načítání PLA souborů, kde je možné pojmenovat jak vstupy tak výstupy a jejichž struktura odpovídá tvaru funkce ve formátu f. Ve vstupním souboru se tedy mohou vyskytnout tato klíčová slova:

- *.i počet* - udává počet vstupních proměnných
- *.o počet* - udává počet výtupních proměnných
- *.ilb* x_0 x_1 ... x_n - pojmenování vstupních proměnných (odděleno pomocí mezer)
- *.ob* y_0 y_1 ... y_n - pojmenování výstupních proměnných (odděleno pomocí mezer)
- *.p počet* - udává počet vstupních termů
- *.e* - označuje konec souboru

Dále následují vstupní vektory, které reprezentují jednotlivé P-termy - **1** označuje výskyt proměnné v přímém tvaru, **0** výskyt proměnné v negovaném tvaru a **-** znamená, že se daná proměnná v termu nevyskytuje.

Oproti běžnému PLA souboru zde tedy chybí především definice typu funkce, protože program je navržen pro práci pouze s funkcemi typu *f*. Výstupní hodnoty tedy vždy odpovídají typu funkce *f*, což znamená, že pro hodnotu **1** patří term do on-setu dané funkce a pro hodnotu **0** nebo **-** nemá term pro hodnotu dané výstupní funkce význam.
Co se týče oddělování vstupních a výstupních vektorů, tak ty buď nemusí být odděleny vůbec, nebo lze kromě běžného whitespacu použít oddělovač | .
Příklad vstupního souboru obsahující všechna přípustná klíčová slova by vypadal třeba takto:

```
.i 4
.o 7
.ilb x0 x1 x2 x3
.ob y0 y1 y2 y3 y4 y5 y6
.p 15
0100 0000010
0101 0000100
0111 0000010
01-1 0010000
001- 0011100
0-00 1000000
100- 1011000
010- 1001000
01-0 1000000
-000 0110100
0-10 0101100
0--1 0000001
01-- 0000001
00-- 0000010
-00- 0000011
.e
```

Poznámka: Pořadí jednotlivých definic je dobré dodržovat, jak je to uvedeno v příkladu. Jinak se může stát, že se program nebude chovat korektně.

3.3.2 Návrh gramatiky

Pro implementaci bylo dále potřeba vytvořit na základě předchozího popisu gramatiku. To se podařilo relativně bez problémů, protože gramatika neobsahovala žádné konflikty a byla rovnou LL(1). Jediným problémem bylo načítání čísel, protože díky struktuře PLA nebylo možné jako lexikální element brát přímo číslo, ale pouze samostatné číslice.

Následující strany zobrazují tabulku pravidel gramatiky, tabulku přechodů a tabulku atributů jednotlivých symbolů.

Popis symbolů použitých pro terminály:

- *ident* - označuje libovolný identifikátor
- *n* - označuje číslici

<table>
<tr><th colspan="3">Tabulka pravidel</th></tr>
<tr><th>Pravidlo</th><th>First</th><th>Follow</th></tr>
<tr><td>1. S $\rightarrow$ .p N</td><td>.p</td><td rowspan="5">ϵ</td></tr>
<tr><td>2. S $\rightarrow$ .i I</td><td>.i</td></tr>
<tr><td>3. S $\rightarrow$ .o O</td><td>.o</td></tr>
<tr><td>4. S $\rightarrow$ T</td><td>n, - , ϵ</td></tr>
<tr><td>5. S $\rightarrow$ .e</td><td>.e</td></tr>
<tr><td>6. N $\rightarrow$ n N</td><td>n</td><td rowspan="2">ϵ</td></tr>
<tr><td>7. N $\rightarrow$ ϵ</td><td>ϵ</td></tr>
<tr><td>8. O $\rightarrow$ ident O</td><td>ident</td><td>ϵ</td></tr>
<tr><td>9. I $\rightarrow$ ident I</td><td>ident</td><td>ϵ</td></tr>
<tr><td>10. T $\rightarrow$ n T</td><td>n</td><td rowspan="3">ϵ</td></tr>
<tr><td>11. T $\rightarrow$ - T</td><td>-</td></tr>
<tr><td>12. T $\rightarrow$ ϵ</td><td>ϵ</td></tr>
</table>

Tabulka 3.1: Tabulka pravidel PLA

Rozkladová tabulka								
	.p	.i	.o	n	-	.e	ident	ϵ
S	1	2	3	4	4	5		4
N				6				7
O							8	
I							9	
T				10	11			12

Tabulka 3.2: Rozkladová tabulka PLA

Popis symbolů použitých v tabulce atributů:

- *jm* - označuje název identifikátoru
- *hod* - označuje hodnotu číslice
- *uk* - označuje ukazatel na vzniklý strom výrazu
- *op* - označuje ukazatel na podstrom výrazu, který je operandem
- *poz* - označuje pozici ve vstupním souboru

Tabulka atributů		
Neterminály		
Symbol	**Syntet. atr.**	**Dědičné atr.**
S	/	/
N	hod	hod
O	/	poz
I	/	poz
T	uk	op, poz
Terminály		
ident	jm	/
n	hod	/

Tabulka 3.3: Tabulka atributů PLA

3.4 Návrh překladače pro načtení VHDL formátu

3.4.1 Analýza problému

Cílem této části je navrhnout překladač umožňující načítání souborů vycházejících svou syntaxí z jazyka VHDL. Takový soubor bude obsahovat deklarace proměnných a výrazy, ve kterých se mohou vyskytnout všechny logické operace: AND, OR, NOT, NAND, NOR, XOR, XNOR.
Kromě proměnných a závorek se mohou ve vstupu vyskytnout také logické vektory, což jsou binární čísla uzavřená v dvojitých uvozovkách. Jednotlivé výrazy se pak zapisují tímto způsobem: x <= a and (b or c); Obecně tedy: id <= V;
Všechny binární operátory zde mají stejnou prioritu, nejvyšší má unární operátor NOT.
Dále se ve vstupním souboru se mohou vyskytovat i běžná desítková čísla, čímž se tento formát odlišuje od jazyka VHDL. Tato čísla slouží ke spojování jednotlivých výstupů do jiných funkcí. Každé desítkově zadané číslo je při analýze nahrazeno výstupní funkcí odpovídající její pozicí v souboru. Stejným způsobem je nahrazen i identifikátor příslušející nějaké výstupní proměnné.

Formát tedy vyžaduje definici těchto klíčových slov:

- *variable* $x_0, x_1, ..., x_n$ - udává názvy vstupních proměnných
- *and* - logická operace AND
- *or* - logická operace OR
- *not* - logická operace NOT
- *nand* - logická operace NAND
- *nor* - logická operace NOR
- *xor* - logická operace XOR
- *xnor* - logická operace XNOR

Jako ukázka formátu může posloužit například tento kód:

```
variable x0,x1,x2,x3;
y0 <= (not x0 and x1) xor (x2 or not x3);
y1 <= not ((x0 nand x1) or (x2 xor x3));
y2 <= (not "0" xor x1) and (not ( "1" and not x3) );
y3 <= 1 or x0;
-- zde proběhne náhrada výrazem příslušejícím funkci na pozici 1 (tedy y1)
y4 <= y1 and x0; -- jiný způsob zapsání téhož
```

Po uložení by tedy výsledek vypadal takto:

```
variable x0,x1,x2,x3;
y0 <= (not x0 and x1) xor (x2 or not x3);
y1 <= not ((x0 nand x1) or (x2 xor x3));
y2 <= (not "0" xor x1) and (not ( "1" and not x3) );
y3 <= (not ((x0 nand x1) or (x2 xor x3)) ) or x0;
y4 <= (not ((x0 nand x1) or (x2 xor x3)) ) or x0;
```

Poznámka: Pokud ve chvíli sestavování výrazu není výstupní funkce s daným označením k dispozici, je automaticky nahrazena logickou **0**.

Definování proměnných klíčovým slovem *variable* není nutné, pokud chceme provádět pouze operace, kde neprobíhá ohodnocování proměnných ve výrazech. Při operacích kde ohodnocování probíhá (minimalizace a test splnitelnosti), se každá nedefinovaná proměnná bere automaticky jako logická **0**.

3.4.2 Návrh gramatiky

Také zde bylo nutné navrhnout gramatiku, která byla tentokrát mírně složitější v tom, že obsahovala levou rekurzi. Po jejím odstranění již vzniklá gramatika byla LL(1).
Z gramatiky je vidět, že podle daných pravidel budou vznikat stromy s levou asociativitou operandů.

Následující strany zobrazují tabulku pravidel gramatiky, tabulku přechodů a tabulku atributů jednotlivých symbolů.

Popis symbolů použitých pro terminály:

- *id* - označuje libovolný identifikátor
- *var* - označuje klíčové slovo *variable*
- *val* - označuje logický vektor nebo desítkové číslo
- *uop* - označuje libovolný unární operátor
- *bop* - označuje libovolný binární operátor

Popis symbolů použitých v tabulce atributů:

- *jm* - označuje název identifikátoru
- *hod* - označuje hodnotu čísla/vektoru
- *uk* - označuje ukazatel na vzniklý strom výrazu
- *op* - označuje ukazatel na podstrom výrazu, který je operandem

Tabulka pravidel		
Pravidlo	**First**	**Follow**
1. S → P;	id	ϵ
2. S → var D;	var	
3. P → id <= V	id	;
4. V → T V'	(, id, uop	), ;
5. V' → bop T V'	bop	), ;
6. V' → ϵ	ϵ	
7. T → uop T	uop	), ;, bop
8. T → (V)	(	
9. T → val	val	
10. D → id D'	id	","
11. D' → , id D'	","	;
12. D' → ϵ	ϵ	

Tabulka 3.4: Tabulka pravidel VHDL

Rozkladová tabulka											
	id	bop	uop	(	)	;	,	var	vect	cis	ϵ
S	1							2			
P	3										
V	4		4	4							
V'		5			6	6					
T	9		7	8					9	9	
D	10										
D'						12	11				

Tabulka 3.5: Rozkladová tabulka VHDL

Tabulka atributů		
Neterminály		
Symbol	**Syntet. atr.**	**Dědičné atr.**
S	uk	/
P	uk	/
V	uk	/
V'	uk	op
T	uk	/
D	/	/
D'	/	/
Terminály		
id	jm	/
vect	hod	/
cis	hod	/

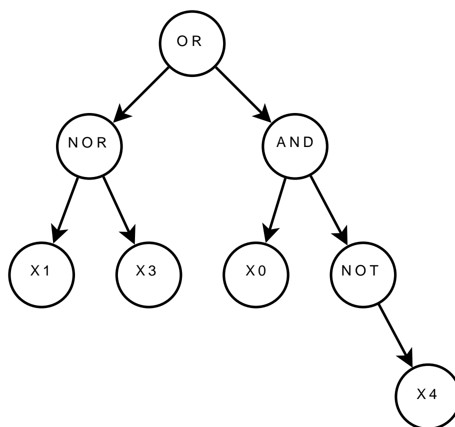
Tabulka 3.6: Tabulka atributů VHDL

3.5 Popis funkcí pro práci s výrazy ve vnitřní reprezentaci

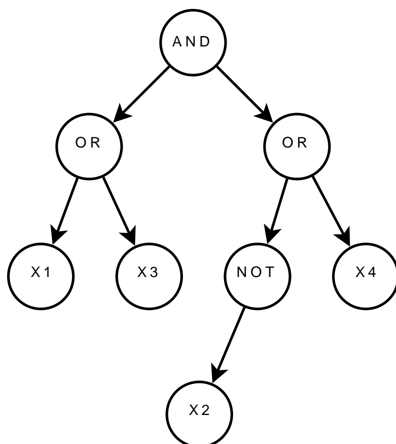
V této sekci bude popsán způsob činnosti algoritmů řešících jednotlivé operace se stromem výrazu.

3.5.1 Negování funkcí

Postup při negování funkce je poměrně jednoduchý, vždy se postupuje od kořene stromu rekurzivně k listů. Pokud je nalezen operátor NOT, tak se ze stromu vymaže a rekurze se v té části podstromu ukončí (úprava dle zákona o dvojí negaci).
Pokud jsou nalezeny operátory NAND, NOR, XOR, respektive XNOR, změní se pouze jejich typ na AND, OR, XNOR respektive XOR a rekurze je ukončena.
V případě, že se ve stromu vyskytnou operátory AND, respektive OR, proběhne změna typu na OR, respektive AND a dále je znegován i celý zbytek podstromu (úprava dle de Morganova zákona).
Co se týče negace listů stromu, tak k těm je předřazen nový uzel NOT a připojen na původní místo, kde se nacházel list.
Vše ilustrují obrázky 3.2 a 3.3.



Obrázek 3.2: Strom výrazu před negací



Obrázek 3.3: Strom výrazu po negaci

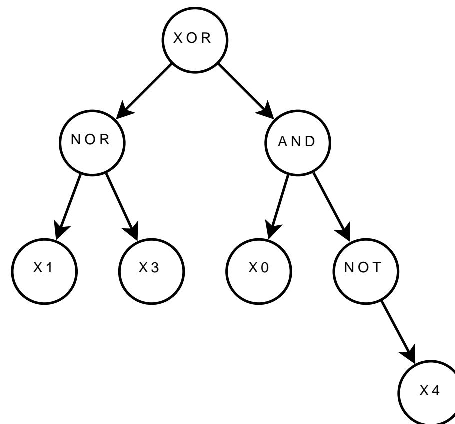
3.5.2 Úprava funkcí do tvaru obsahujícího pouze základní operace

Účelem této úpravy je transformace funkce do formy obsahující pouze operace AND, OR a NOT. Při úpravě se postupuje také od kořeni k listům a všechny operace nespadající mezi základní se nahrazují svým ekvivalentem z operací základních. Všechny úpravy probíhají dle pravidel o logických operacích definovaných v 1.2.

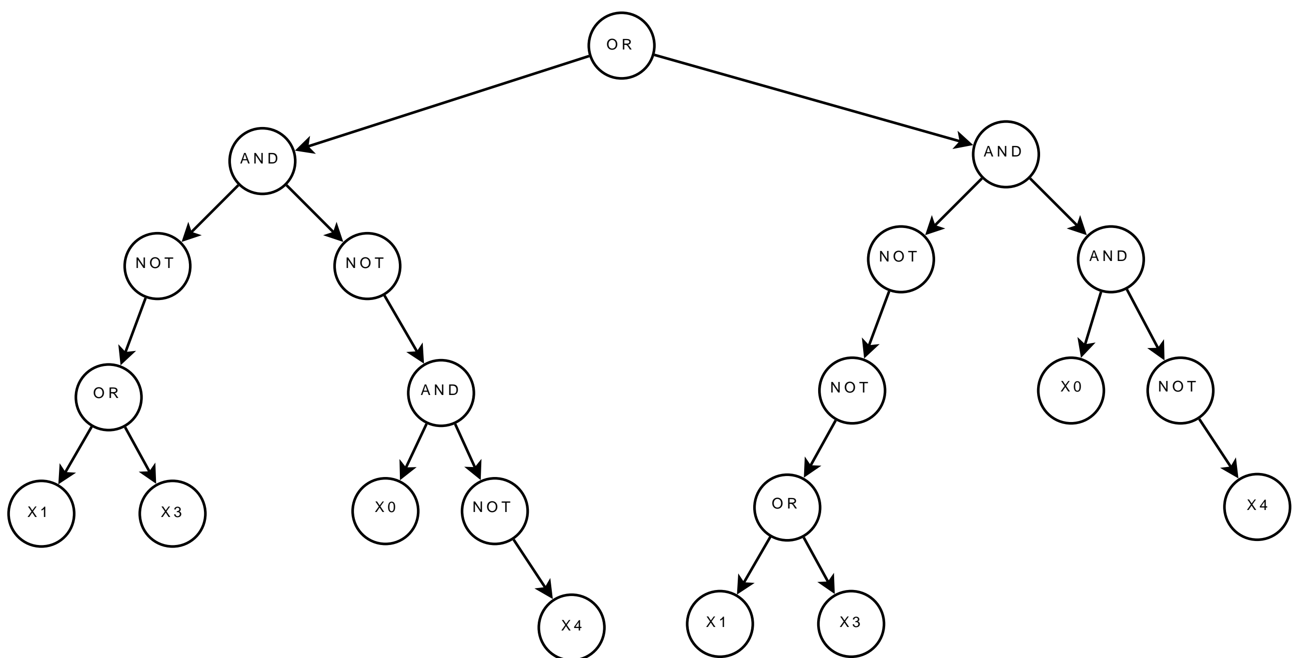
Způsob náhrady se odvíjí od typu nalezené operace:

- NAND - vytvoří se nový uzel NOT, který se předřadí nahrazovanému uzlu a jeho typ se změní na AND
- NOR - vytvoří se nový uzel NOT, který se předřadí nahrazovanému uzlu a jeho typ se změní na OR
- XOR - vytvoří se dva nové podstromy, z nichž první obsahuje levý podstrom spojený pomocí AND s negovaným pravým podstromem a druhý obsahuje negovaný levý podstrom spojený pomocí AND s pravým podstromem. Tyto nové podstromy jsou vloženy na místo těch původních a typ operace v nahrazovaném uzlu je změněn na OR
- XNOR - stejné jako v předchozím případě, akorát je celému podstromu předřazen ještě operátor NOT

Efekt úpravy je zachycen na obrázcích 3.4 a 3.5.



Obrázek 3.4: Strom před rozkladem na základní operace



Obrázek 3.5: Strom po rozkladu na základní operace

3.5.3 Úprava funkcí do DNF a CNF

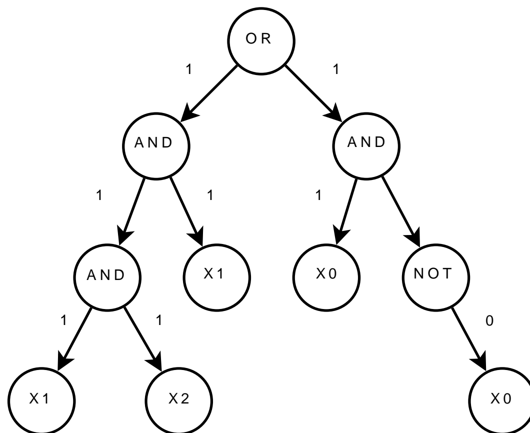
Úprava probíhá dle algoritmu popsaného v sekci 3.6. Výsledkem je minimalizovaná funkce v požadovaném tvaru. V nejhorším případě může mít exponenciální složitost.

3.5.4 Testování splnitelnosti včetně získání on-setu funkce

Při testování splnitelnosti je nutné nejdříve funkci upravit do DNF. Pak již lze v lineárním čase určit, zda jednotlivé P-termy mohou nabývat logické hodnoty **1**. To je zajištěno postupným dosazováním hodnot za jednotlivé proměnné termu a detekcí konfliktů mezi hodnotami proměnných (zde se totiž v jednotlivých P-termech mohou opakovat proměnné, a tudíž nelze automaticky prohlásit, že term do on-setu patří, jakmile je různý od logické **0**). V případě, že ke konfliktu nedojde, patří daný term do on-setu funkce.

Při ohodnocování se postupně posílá logická **1** z kořene stromu směrem k listům. Cesta vedoucí přes uzel OR značí, že se má levý a pravý podstrom ohodnotit odděleně (tedy že se jedná o různé P-termy, kde každý má své vlastní ohodnocení). Pokud cesta vede přes uzel obsahující NOT, je aktuální hodnota znegována a poslána dále. Hodnota, která dorazí k listu s proměnnou, je uložena jako ohodnocení dané proměnné. Každá neohodnocená proměnná je pak brána jako DC.

Způsob ohodnocování zachycuje obrázek 3.7, kde je ukázáno ohodnocení výrazu
$(x_1\ and\ x_2\ and\ x_1)\ or\ (x_0\ and\ not\ x_0)$.



Obrázek 3.6: Ohodnocování proměnných výrazu v DNF

Z obrázku je patrné, že u pravého P-termu došlo ke konfliktu mezi hodnotami proměnných, a tudíž do on-setu funkce nepatří. Naopak levý P-term žádné konflikty neobsahuje a celá funkce nabývá hodnoty **1** pro vektor $(x_0, x_1, x_2) = (-, 1, 1)$.

3.5.5 Úprava do formy odpovídající síti NOR

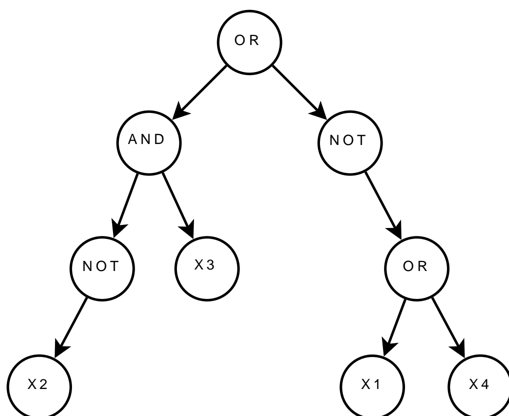
Tato úprava transformuje funkci do formy odpovídající síti dvouvstupových hradel. Před vlastní úpravou je strom nejdříve rozbit na základní operace.

Celá operace pak probíhá rekurzivně tímto způsobem:

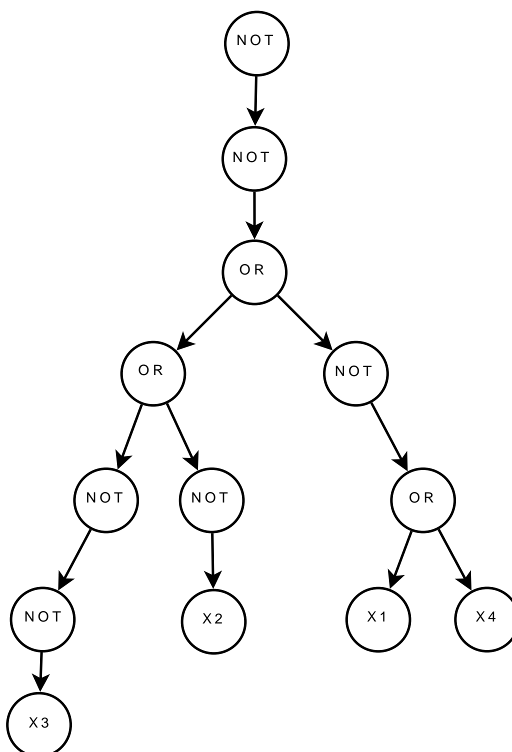
- Pokud se při postupu od kořene níže nalezne operace NOT následovaná operací OR, tak se daná část stromu přeskočí a pokračuje se v úpravě stromu.

- Pokud je nalezen samostatný uzel obsahující OR, vytvoří se nad ním dva uzly NOT a také se pokračuje dále.
- V případě nalezení operátoru AND se uzel zamění za OR a předřadí se uzlu samotnému i jeho potomkům negace. Algoritmus pak pokračuje v činnosti v úrovni od těchto potomků.

Úpravu do dané formy zobrazují obrázky 3.7 a 3.8.



Obrázek 3.7: Strom výrazu před úpravou



Obrázek 3.8: Strom výrazu po úpravě

3.5.6 Úprava do formy odpovídající síti NAND

Algoritmus je prakticky identický s tím v předchozí sekci, stačí zaměnit v textu operaci OR za operaci AND a naopak.

3.5.7 Provedení libovolných logických operací mezi více funkcemi

Algoritmus provádění této operace je stejný jako při načítání VHDL souboru. Operace může proběhnout na základě externího skriptu nebo zadáním příkazu přímo z klávesnice.

3.5.8 Načtení vstupních dat v požadovaném formátu (PLA, VHDL)

Při načítání vstupních dat se postupně vytvářejí jednotlivé stromy daných výrazů. Rozdíl mezi PLA a VHDL je pouze v tom, že z PLA souboru se díky jeho struktuře budují všechny stromy najednou, zatímco u VHDL je vždy nejdříve vybudován kompletně N-tý strom a teprve potom ten další.
Tento rozdíl je způsoben tím, že u PLA formátu je každý term následován množinou funkcí, ke kterým patří, a proto je nutné vytvořit jeho reprezentaci pro všechny funkce najednou. U VHDL formátu jsou jednotlivé funkce specifikovány svými vlastními výrazy, a tudíž je možné vybudovat celý strom výrazu nezávisle na ostatních.
Celá operace proběhne v lineárním čase.

3.5.9 Uložení výsledků v požadovaném formátu (PLA, VHDL, BLIF)

Zde je mezi formáty podstatný rozdíl. Při ukládání VDHL a BLIFu totiž není nutné provádět s výrazem žádné další úpravy, protože díky jejich obecnosti lze uložit strom výrazu v jakémkoli tvaru. Do formátu PLA je však možné uložit pouze výraz v DNF a musí tedy nejdříve proběhnout potřebná modifikace výrazu, která se skládá z aplikací distributivních zákonů a minimalizace jak je popsáno v 3.6.
Uložení VHDL a BLIFu proběhne tedy v lineárním čase, zatímco u PLA jde v nejhorším případě o složitost exponenciální.

3.5.10 Výpočet kofaktoru funkce

U této operace jde o výpočet kofaktoru funkce vůči nějaké proměnné. Operace probíhá na základě Shannnonova teorému o rozkladu, který říká, že každou funkci lze rozložit podle následujícího vzorce:

$$f(x_0, \bar{x}_0, x_1, \bar{x}_1, ..., x_n) = x_0 * f(1, 0, x_1, \bar{x}_1, ..., x_n) + \bar{x}_0 * f(0, 1, x_1, \bar{x}_1, ..., x_n)$$

Tento teorém říká, že každou funkci lze rozdělit na dvě. Před první funkci vytkneme přímou formu proměnné a její výskyty ve funkci nahradíme jedničkou. Negovanou formu proměnné potom nahradíme nulou. Stejným způsobem postupujeme i s druhou funkcí, pouze vytkneme negovanou proměnnou.
Definice kofaktoru například vůči x_1 by vypadala takto:
$f(x_0, \bar{x}_0, x_1, \bar{x}_1, ..., x_n)_{(x_1)} = x_1 * f(x_0, \bar{x}_0, 1, 0, ..., x_n)$

Při vlastním výpočtu proběhne náhrada tímto způsobem: Pokud se počítá kofaktor vůči přímé formě proměnné je tato nahrazena v celém výrazu logickou **1**, v případě výpočtu kofaktoru vůči negaci proměnné je tato nahrazena logickou **0**.
Kofaktor vůči termu lze vypočítat opakovaným vyvoláním této operace.

3.6 Návrh efektivního algoritmu pro převod funkce do DNF a CNF

Ačkoli se původně zdálo, že půjde o problém obtížností srovnatelný s předchozími, tvoří nakonec řešení tohoto problému majoritní část celé práce.

3.6.1 Analýza složitosti řešení problému

Uvažujme funkci obsahující disjunkci k P-termů, z nichž každý obsahuje M literálů.
Celková délka výrazu tedy bude: $N = k * M$
Pokud budeme aplikovat distributivní zákony, bude potřeba na převod do DNF celkem M^k operací.

Po dosazení za k a úpravě získá výraz reprezentující počet operací následující tvar: $(\sqrt[M]{M})^N$
Z tohoto tvaru již vidíme, že pro $N \to \infty$ poroste výraz nejvíce právě tehdy, když je podvýraz v závorce maximální, což platí pro $M = 2$.
To potvrzuje předpoklad, že složitost výpočtu převodu výrazu do DNF je největší právě tehdy, když je převáděný výraz ve tvaru CNF a každý P-term obsahuje právě 2 literály. Výpočetní i paměťová složitost je tedy v nejhorším případě $O(\sqrt{2}^N)$, kde N je počet literálů ve výrazu.

Převod funkce do DNF je ovšem nezbytný pro získání on-setu funkce, který umožňuje uložení výsledku ve formátu PLA, proto bylo nutné nalézt efektivní způsob, jak tento převod uskutečnit.

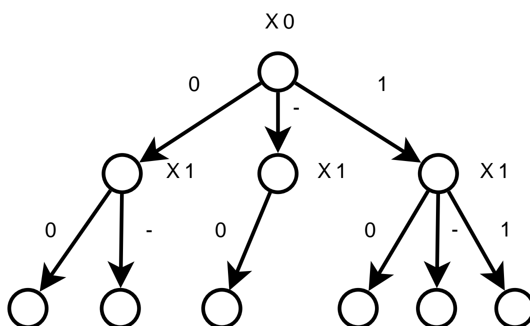
3.6.2 Popis činnosti převodního algoritmu

Prvním krokem algoritmu je převedení výrazu do formy složené pouze ze základních operací a přesunutí všech negací až k listům stromu. Tím získáme tvar, na který lze již použít distributivní zákony, jejichž postupnou opakovanou aplikací ve stromu se lze dostat až k požadovanému tvaru funkce. Z výsledné funkce v DNF již není problém získat její on-set dle postupu popsaného v sekci 3.5.2. K převodu do CNF je použit naprosto stejný algoritmus, jen je před začátkem výpočtu i po něm funkce znegována.
Původní verze programu fungovala pouze na tomto principu a je zřejmé, že si díky složitosti výpočtu nebyla schopna poradit s výrazy v CNF majícími více než 20 termů a výsledek navíc obsahoval i velké množství duplicitních termů.

Pro zrychlení výpočtu byla tedy nutná minimalizace výrazu v průběhu výpočtu. K tomu je zapotřebí převádět na DNF jednotlivé podstromy výrazu směrem od listů stromu ke kořeni a tyto vždy samostatně minimalizovat. Jinak řečeno - minimalizace se musí provádět mezi každou operací ve stromě a postupuje se od výrazů s nevyšší prioritou operace až k těm s nejnižší.

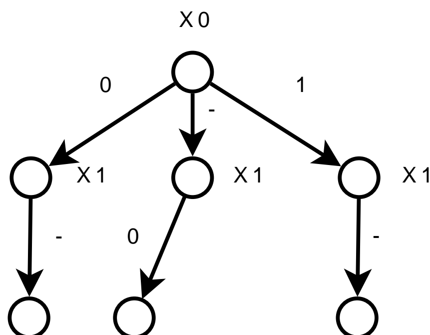
3.6.3 Způsob minimalizace stromu výrazu

Dalším problémem bylo nalézt způsob jak vůbec samotnou minimalizaci provádět, aby strom dostatečně zredukovala a zároveň netrvala příliš dlouho, protože se provádí opakovaně. Vhodnou strukturou pro řešení tohoto problému se ukázal ternární strom uvedený v práci [6]. Příklad ternárního stromu výrazu $(x_0, x_1) = (\bar{x}_0 * \bar{x}_1) + \bar{x}_0 + \bar{x}_1 + (x_0 * \bar{x}_1) + x_0 + (x_0 * x_1)$ zachycuje obrázek 3.9 (**0** označuje negovanou proměnnou, **-** znamená, že na hodnotě dané proměnné nezáleží a **1** označuje proměnnou v přímé formě).



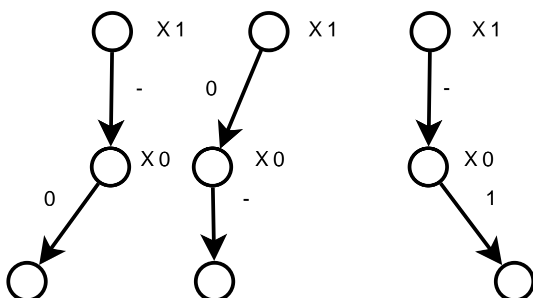
Obrázek 3.9: Příklad ternárního stromu

Jak probíhá vložení termu do stromu je zřejmé; procházíme term po proměnných, zároveň procházíme strom a pokud je třeba (pro dané ohodnocení proměnné ještě ve stromu neexistuje uzel), vytváříme nové uzly.
Z vlastností ternárního stromu vyplývá, že jsou automaticky eliminovány všechny duplicitní termy a poměrně snadno lze eliminovat i proměnné podle zákona o komplementu za použití algoritmu na minimalizaci ternárního stromu uvedeného taktéž v [6].
Tento algoritmus pracuje na principu rotace stromu a slučování listů stromu, které mají společného předchůdce (tzn. liší se pouze v jedné proměnné). Strom z příkladu 3.9 po sloučení listů na základě tohoto algoritmu zachycuje obrázek 3.10.



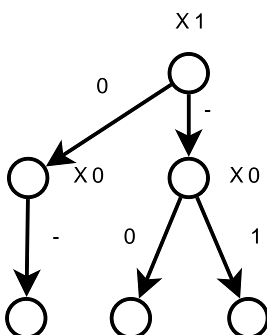
Obrázek 3.10: Ukázka způsobu slučování listů v ternárním stromě

Sloučení listů se provádí před každou rotací stromu. Samotná rotace stromu spočívá v přesunutí uzlu v kořeni až k listům stromu a spojení takto nově vzniklých stromů zpět do jednoho (postup spojování je stejný, jako při vkládání nových termů). Předchozí strom po provedení přesunu kořene ukazuje obrázek 3.11.



Obrázek 3.11: Ternární strom po přesunutí kořene k listům

Nyní tedy následuje spojení jednotlivých stromů, které vidíme na obrázku 3.12.



Obrázek 3.12: Ternární strom po spojení

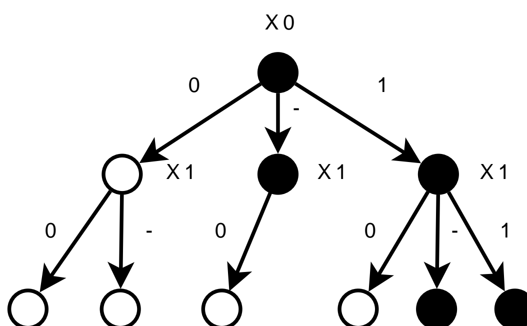
Po provedení této operace může být na listy opět aplikováno sloučení. Opakováním tohoto postupu se nakonec zbavíme většiny komplementů ve stromě.
Dále lze na základě ternárního stromu snadno generovat termy, které mají rovnou seřazené proměnné, což například při uložení ve VHDL formátu značně zvyšuje přehlednost.
Složitost celé operace včetně transformace výrazu na ternární strom a zpět je dle [6] jen $O(N * P^2)$, kde P je počet vstupních proměnných a N je počet vstupních termů.

Pouhá eliminace duplicit a komplementů se však nakonec ukázala nedostatečná, protože velkou část dat mohou tvořit termy, které jsou samy podmnožinou jiných termů. Navíc samotná minimalizace zase vytváří termy, které mohou být nadmnožinami ostatních termů. Proto bylo nutné nějakým způsobem odstranit všechny nadbytečné termy na základě absorpce. Algoritmus eliminace těchto termů pomocí udržované skupiny potencionálních nadmnožin popsaný v [6], mě nepřesvědčil o své kvalitě, protože vyžadoval alokaci další paměti a nevedl k úplné absorpci všech nadbytečných termů. To vedlo k vytvoření nového algoritmu popsaného v sekci 3.6.4.

3.6.4 Minimalizace ternárního stromu pomocí absorpce

Tento algoritmus pracuje na principu porovnávání termů v ternárním stromě mezi sebou. Do stromu jsou termy přidávány standardním způsobem, ale při transformaci ternárního stromu zpět na strom výrazu se zjišťuje, zda daný term není podmnožinou nějakého jiného termu obsaženého ve stromě - v takovém případě se do výstupu vůbec nezahrne.
Postup této operace je následovný: Nejdříve se z ternárního stromu vytvoří kopie podstromu reprezentující daný výstupní term. Potom se paralelně ok kořene porovnávají jednotlivé uzly původního stromu a výstupního termu. Pokud má uzel v původním stromě nějakého následníka shodujícího s následníkem ve zkoumaném termu, nebo je jedním z následníků ve stromě uzel reprezentující DC, znamená to, že tato část termu je podmnožinou původního ternárního stromu a má smysl pokračovat dál. Následně dojde v obou stromech posun o hladinu níž a opakuje se předchozí postup. Pokud se tímto způsobem podaří dostat až k listům stromu, znamená to, že zkoumaný term je skutečně podmnožinou nějakého jiného termu. Každý term je samozřejmě podmnožinou sebe sama, což nesmí zabránit jeho umístění do výstupu - term je tedy pohlcen, jen pokud se liší aspoň v jednom uzlu.

Postup algoritmu pro term $(x_0, x_1) = (1, 1)$ zachycuje obrázek 3.13, kde jsou všechny navštívené uzly označeny černě. Z obrázku vidíme, že tento term by do výstupu zahrnut nebyl protože je podmnožinou termu $(1, -)$.



Obrázek 3.13: Ukázka postupu absorpčního algoritmu

Důležitou otázkou samozřejmě je, jakou má tento postup časovou složitost. Horním odhadem složitosti je výraz $P{*}N^2$ (kde P označuje počet proměnných a N počet termů), neboť porovnání jednoho termu s celým stromem proběhne nejhůře v čase $P * N$ a porovnání každý s každým tedy nejhůře v čase $P * N^2$. Tento odhad je však až příliš pesimistický. K porovnávání totiž dochází jen do doby, kdy je nějaká část termu shodná s částí původního stromu. Vždycky tedy odpadne minimálně třetina porovnávaného podstromu.

Představme si nyní ternární strom obsahující úplně všechny uzly (a tím pádem i termy), což je nejhorší případ, který může nastat. Celkový počet termů je pak roven počtu listů, což je $N = 3^P$.
Při srovnávání termu se bude tedy nejhůře bude pokračovat v každém kroku dvojnásobným počtem cest. Počet srovnání v každém uzlu je vždy roven třem. To nám dává pro každý zkoumaný term maximálně $3 * (2^P - 1)$ porovnání, pro celý strom tedy $3 * (2^P - 1) * N$.

Výsledná časová složitost po dosazení $N = 3^P$ a vytknutí N^2 je tedy: $O((\frac{2}{3})^P * N^2)$,
vyjádřeno pouze pomocí N: $O((\frac{2}{3})^{\log_3 N} * N^2)$

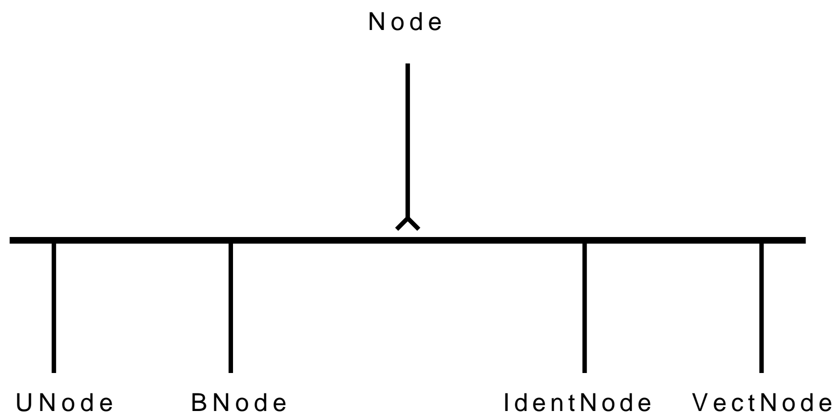
4 Realizace

V této kapitole budou popsány datové struktury a třídy použité při řešení jednotlivých problémů. Také zde bude popsán základní interface programu a některých funkcí.

Při realizaci byla využita jen standardní knihovna, takže je program přeložitelný nezávisle na operačním systému.
Kvůli paměťovým nárokům některých výpočtů je vhodné mít alespoň 512 MB RAM.

4.1 Popis struktury pro reprezentaci stromu výrazu

Struktura výrazu vychází z abstraktního předka **Node**, z něhož jsou odvozeny všechny ostatní typy uzlů. Celý strom pak vznikne propojením jednotlivých uzlů pomocí ukazatelů. Přehled jednotlivých typů uzlů ukazuje obrázek 4.1.



Obrázek 4.1: Typy uzlů použitých při implementaci

Nyní rozebereme jednotlivé typy uzlů podrobněji:

- **UNode** - reprezentuje unární uzel (zde pouze operace NOT), má vždy jednoho následníka
- **BNode** - reprezentuje binární uzel (operace AND, OR, NAND, NOR, XOR, XNOR), má vždy dva následníky
- **IdentNode** - reprezentuje identifikátor, obsahuje řetězec s názvem, nemá žádného následníka
- **VectNode** - reprezentuje logický vektor, obsahuje hodnotu a délku, nemá žádného následníka

Všechny uzly navíc obsahují položku se svým typem, aby je bylo možné odlišit. Jako ukázka může posloužit třeba binární uzel:

```
class BNode : public Node
{
  public:
  Node * Left,* Right; //ukazatele na potomky
  unsigned char Type;  //typ uzlu

  ...
}
```

4.2 Popis struktury pro udržování informací proměnných

Další použitou strukturou je tabulka symbolů, kde jsou zaznamenány všechny vstupní proměnné. Při čtení vstupního souboru jsou do ní jednotlivé proměnné postupně přidávány. Tabulka je nezbytná pro ohodnocování výrazů - tedy při minimalizaci a testování splnitelnosti. Je realizována spojovým seznamem složeným z jednotlivých symbolů. U každého symbolu je udržován jeho název, hodnota a proměnná označující, zda je hodnota symbolu platná.

```
class Symbol
{
  public;
  string name;        //název proměnné
  unsigned int value; //hodnota proměnné
  bool valid;         //platnost hodnoty proměnné
  Symbol * next;      //ukazatel na další symbol

  ...
}

class STable
{
  Symbol * first, *last, *act;
  //ukazatele na začátek, konec a aktuální symbol v seznamu

  public:
  unsigned int count; //počet vložených položek

  ...
}
```

4.3 Popis struktury zaobalující informace o jednotlivých výrazech

Protože informace o výrazu jsou kromě tabulky symbolů a stromem výrazu tvořeny ještě dalšími daty, bylo vhodné vytvořit další zaobalující strukturu. Tato struktura tedy obsahuje kromě ukazatele na strom výrazu a tabulku symbolů ještě jméno výstupní proměnné a za ukazatel na další prvek. Tyto jednotlivé elementy jsou stavebními prvky finální struktury, která vše sjednocuje. Jde opět o spojový seznam, jenž navíc obsahuje informace načtené ze vstupu, které jsou společné pro všechny elementy.

```
class Element
{
  public:
  Node * root;         //ukazatel na strom výrazu
  STable * symbTable;  //ukazatel na tabulku symbolů
  string name;         //název výstupní proměnné
  unsigned int terms;  //počet termů výrazu
  Element * next;      //ukazatel na další element

  ...
}
```

```
class TreeArray
{
  Element * first,*last; //ukazatele na začátek a konec seznamu

  public:
  unsigned int terms; //počet termů
  int invars;         //počet vstupních proměnných
  int outvars;        //počet výstupních proměnných
  bool namedi;        //informace o pojmenování vstupů
  bool namedo;        //informace o pojmenování výstupů

  STable * symbNames; //ukazatel na společnou tabulku symbolů

...
}
```

4.4 Popis struktury pro reprezentaci ternárního stromu

Struktura pro reprezentaci ternárního stromu je velmi jednoduchá, protože se v ní vyskytuje pouze jeden typ uzlu. Takový uzel obsahuje pouze tři položky - ukazatel na levého, prostředního a pravého následníka. V případě, že některý z těchto následníků neexistuje, je ukazatel nastaven na NULL.

```
class tNode
{
  protected:
  tNode * left, * mid, * right;
  //ukazatele na levého (0), prostředního (-) a pravého (1) následníka

  ...
}
```

4.5 Popis interfacu programu

Program je možné používat dvěma způsoby, buď interaktivně - zadáváním jednotlivých požadavků z klávesnice, nebo neinteraktivně - zadáním operací pomocí externího skriptu. Skript umožňuje pouze vytváření nových výstupních funkcí, ostatní operace je nutné zadávat vždy interaktivně. Příkazy ve skriptu mají stejnou syntaxi jako příkazy VHDL formátu.
Bližší popis bude uveden v uživatelské příručce.

4.6 Popis interfacu funkcí

Jedním z požadavků na program bylo také vytvoření interfacu pro snadnou použitelnost a začlenitelnost funkcí, které provádějí jednotlivé operace. Všechny tyto funkce jsou definovány v souboru *interface.h*.

Nyní si zmíněné funkce rozeberme:

- **void init ()** - inicializuje potřebné proměnné a vytvoří objekty, nutné zavolat jednou na začátku programu
- **void readpla (istream * input)** - načte PLA, parametrem je ukazatel na vstupní stream
- **void readvhdl (istream * input)** - načte VHDL, parametrem je ukazatel na vstupní stream
- **void performOp (int option, ostream * output, int pos, int num, bool val)** - provede požadovanou operaci, lze volat opakovaně, parametr option udává typ operace. Další parametry jsou již nepovinné - parametr output reprezentuje výstupní stream pro uložení výsledku, ostatní tři parametry slouží k výpočtu kofaktoru funkce - první z nich udává číslo výstupní funkce, druhý číslo vstupní proměnné a třetí přiřazovanou hodnotu.
- **void performScript (istream * script)** - provede obsah zadaného skriptu
- **void clean ()** - dealokuje vytvořené objekty, vhodné zavolat na konci programu

Typy operací, proveditelných zavoláním funkce *performOp* ukazuje následující kód:

```
#define NEG 1       //negace
#define AON 2       //forma obsahující pouze základní operace
#define CNF 3       //CNF
#define DNF 4       //DNF
#define SAT 5       //test splnitelnosti
#define NANDF 6     //NAND forma
#define NORF 7      //NOR forma
#define ADDF 8      //přidání logické funkce
#define SAVEPLA 9   //uložení jako PLA
#define SAVEVHDL 10 //uložení jako VHDL
#define SAVEBLIF 11 //uložení jako BLIF
#define ABS 12      //vypnutí/zapnutí absorpce
#define COF 13      //výpočet kofaktoru
```

Poznámka: Všechny tyto operace jsou definovány také v souboru *interface.h*.

Jednoduchý program pro provedení skriptu, znegování všech výrazů, výpočtu kofaktoru třetí funkce vůči první proměnné v přímé formě a uložení výsledku by mohl vypadat takto:

```
#include <fstream>
#include <sstream>
#include "interface.h"

int main (int argc, char * argv[])
{
  ifstream * input = new ifstream("input.pla");
  ofstream * output = new ofstream("output.pla");

  ifstream * script = new ifstream("script.txt");
  //istringstream * script = new istringstream("y3 <= 0 xor 1 xor 2;");
  //skript může být reprezentován jakýmkoli vstupním streamem

  init();
  autoprint = true; //nastaví ukládání bez interakce

  readpla(input);

  performScript(script);
  performOp(NEG);
  performOp(COF, NULL, 3, 0, 1);
  performOp(SAVEPLA, output);

  clean();
}
```

5 Testování

V této kapitole budou otestovány navržené algoritmy jak z hlediska rychlosti, tak z hlediska kvality řešení a následně porovnány s jinými existujícími řešeními. Při testování byly použity příklady z [3] a náhodně generované funkce. Testovací sestava byla Core 2 duo 2 GHz, 2 GB RAM.

5.1 Testování převodu funkcí do DNF

5.1.1 Test 1

Cílem tohoto testu je zjistit, jak dlouho budou trvat velmi složité převody u reálných problémů a jak se projeví na době výpočtu nedostatečná minimalizace výrazu. Proto zde byly použity pouze příklady z [3] (sada MCNC, příklady bez specifikace typu funkce). Specifikace jednotlivých benchmarků, seřazených dle obtížnosti, jsou zobrazeny v tabulce 5.1.
Potřebné složitosti je dosaženo znegováním všech výrazů ve vstupu, protože negace vede u PLA souborů vždy na tvar CNF a převod zpět na DNF může mít pak až exponenciální složitost. Rozdílná míra minimalizace při jednotlivých sadách měření je zajištěna postupným zapojováním navržených minimalizačních algoritmů do výpočtu (vždy je přidán další způsob minimalizace). Výsledky měření jsou zobrazeny v tabulce 5.2.

Tabulka benchmarků				
benchmark	**vstupů**	**výstupů**	**termů**	**podíl DC**
ibm.pla	48	17	173	velký
soar.pla	83	94	529	velký
ti.pla	47	72	241	velký
ex4.pla	128	28	620	velký
ex1010.pla	10	10	1024	nulový
test3.pla	10	35	1024	nulový
pdc.pla	16	40	2810	malý
x7dn.pla	66	15	622	velký
test2.pla	11	35	2048	nulový
xparc.pla	41	73	551	střední

Tabulka 5.1: Přehled souborů použitých pro testování

Tabulka výsledků pro různé způsoby minimalizace						
	identitou		**komplementem**		**absorpcí**	
benchmark	**čas** [s]	**termy** [-]	**čas** [s]	**termy** [-]	**čas** [s]	**termy** [-]
ibm.pla	-	-	15,7	5201	4,1	878
soar.pla	-	-	96,7	5306	19,0	1064
ti.pla	-	-	-	-	103,5	7372
ex4.pla	-	-	606,6	3665	111,9	579
ex1010.pla	-	-	279,6	13662	139,7	11196
test3.pla	-	-	692,3	52901	294,8	37862
pdc.pla	-	-	-	-	432,5	2694
x7dn.pla	-	-	-	-	1295,4	23481
test2.pla	-	-	-	-	1985,4	100689
xparc.pla	-	-	-	-	3143,0	22675

Tabulka 5.2: Naměřené hodnoty pro převod funkcí do DNF

Z výsledků je vidět, že vliv minimalizace na dobu výpočtu je skutečně obrovský. Pro úplně vypnutou minimalizaci (není uvedeno v tabulce) dojde prakticky okamžitě k vyčerpání veškeré dostupné paměti a výsledku se nedočkáme u žádného z těchto benchmarků. Při přidání minimalizace pomocí identity (tedy jen základě vkládání a vybírání termů z ternárního stromu) je situace prakticky stejná - pouze dojde k vyčerpání paměti až po delší době a také nedostaneme žádný výsledek.
Abychom získali alespoň nějaké výsledky, je tedy nutné přidat i minimalizaci pomocí komplementů - zde se již pro některé benchmarky výpočet dokončí, ale výsledky jsou stále neuspokojivé, protože se daří vyřešit jen polovinu příkladů testovací množiny.
Pro získání lepších výsledků musíme tedy zapojit i minimalizaci na základě absorpce - ta si je již schopna úspěšně poradit se všemi testovacími příklady, a to v časech o mnoho kratších než v předchozím případě.
Jasně se tedy prokázalo, že efektivní minimalizace je pro tyto převody naprosto nezbytná.

Podle výsledných hodnot se dále ukazuje, že nejlépe si algoritmus poradí s benchmarky, kde je velký podíl DC a malý počet termů (tedy obecně s benchmarky s celkově malým počtem literálů), zatímco na počtu vstupů tolik nezáleží.
To není zas tak překvapující, neboť při úpravách pomocí distributivních zákonů nemá počet vstupních proměnných přímý vliv - záleží jen na aktuálním počtu literálů. Tento vliv se projeví, až když se funkci nedaří dostatečně minimalizovat a počet literálů narůstá. To může být případ benchmarku `x7dn.pla`, který oproti předpokladům trval velmi dlouho.
Dále je viditelný velmi malý vliv počtu výstupů na rychlost algoritmu - počet výstupů totiž ovlivňuje dobu provádění pouze lineárně.

5.1.2 Test 2

V tomto testu se pokusím zjistit, jaký má vliv struktura stromu převáděného výrazu na rychlost výpočtu a jestli je dobré výrazy před prováděnou úpravou minimalizovat.
Stromy výrazů načtených přímo z PLA souborů mají totiž vždy levou asociativitu, zatímco stromy vytvořené minimalizačním algoritmem nemají asociativitu nijak určenu.
Rozdílné struktury stromu za stejných podmínek lze dosáhnout uložením dat v PLA formátu a jejich opětovným načtením. Variantu stromů bez asociativity lze realizovat převodem všech vstupních výrazů do DNF.
Naměřené hodnoty jsou zobrazeny v tabulce 5.3.

Tabulka výsledků		
benchmark	**čas bez asociativity [s]**	**čas s levou asociativitou [s]**
ibm.pla	3,3	4,3
soar.pla	13,3	22,4
ti.pla	24,9	49,7
ex4.pla	31,3	95,0
ex1010.pla	122,4	94,0
test3.pla	284,5	204,5
pdc.pla	7,5	16,3
x7dn.pla	-	1632,1
test2.pla	2460,3	1389,7
xparc.pla	629,2	745,7

Tabulka 5.3: Časy převodu funkcí do DNF při rozdílné struktuře stromu výrazu

Výsledky tohoto testu jsou vcelku překvapivé. Například benchmark `x7dn.pla` se při struktuře bez dané asociativity vůbec nedokončil z důvodu nedostatku paměti, ale druhý výpočet proběhl bez problémů. V případě benchmarku `ex4.pla` zas trval výpočet při levé asociativitě třikrát tak dlouho než bez asociativity.
Vysvětlení tohoto jevu spočívá v tom, že při rozdílných strukturách výrazů se také jinak provádějí jednotlivé minimalizace a to může mít na celkový průběh výpočtu fatální dopad. Z testu tedy vyplývá, že ani jedna varianta není obecně rychlejší a je dobré je zkoušet případně obměňovat.
Stejně tak nelze obecně říci, že zminimalizování výrazů před úpravou zaručí rychlejší řešení. Pokud totiž porovnáme časy výpočtů v tabulce 5.2 s těmi v tabulce 5.3, zjistíme, že některé operace proběhly bez úvodní minimalizace rychleji než s ní. Ve většině případů však urychlení přinesla.

5.2 Testování minimalizace

Cílem tohoto testu je srovnat algoritmus **pupik** použitý v [6] s mnou implementovaným algoritmem (nazvěme ho **btmin**) a sofistikovaným minimalizátorem logických funkcí **espresso** [4]. Aby byly výsledky porovnatelné, je nutné měření provést pouze na jednovýstupových funkcích. K jejich vytvoření byl použit generátor náhodných funkcí z práce [5]. Protože algoritmy **pupik** a **btmin** pracují na podobném principu, ukáže se dobře efekt rozdílného přístupu k minimalizaci absorpcí.
Co se týče čísel v názvech benchmarků, udává číslo za písmenem i počet vstupů, číslo za písmenem p počet termů v tisících a číslo za písmenem d procentuální podíl DC v termech. Jediný výsledný term po minimalizaci značí, že daná funkce je tautologie.

Tabulka výsledků						
	čas [s]			**termy [-]**		
benchmark	**pupik**	**btmin**	**espresso**	**pupik**	**btmin**	**espresso**
i20p1d10	0,078	0,422	0,51	1000	1000	1000
i20p2d10	0,140	0,889	1,71	1998	1995	1995
i20p5d10	0,234	2,028	9,32	4985	4960	4958
i20p10d10	0,390	3,869	38,56	9949	9856	9833
i20p20d10	0,951	7,878	235,12	19790	19491	19308
i20p1d25	0,078	0,468	1,21	1000	999	999
i20p2d25	0,156	0,811	5,63	1993	1986	1983
i20p5d25	0,234	1,887	45,17	4953	4925	4908
i20p10d25	0,421	3,744	156,18	9879	9706	9511
i20p20d25	1,076	7,815	949,08	19480	18935	16361
i20p1d50	0,063	0,343	4,39	973	958	911
i20p2d50	0,140	0,655	36,19	1905	1854	1036
i20p5d50	0,234	1,294	3,04	3642	3327	20
i20p10d50	0,405	2,199	0,57	6000	5328	1
i20p20d50	0,967	3,978	0,62	7243	6292	1
i20p1d65	0,062	0,219	0,06	475	421	1
i20p2d65	0,093	0,343	0,07	620	483	1
i20p5d65	0,203	0,670	0,06	408	320	1
i20p10d65	0,327	0,670	0,04	1	1	1
i20p20d65	0,670	0,171	0,07	1	1	1

Tabulka 5.4: Naměřené hodnoty pro srovnání minimalizačních algoritmů

Z naměřených výsledků je vidět, že nejrychlejší je ve většině případů algoritmus **pupik** a nejkvalitnější řešení dává vždy **espresso**, což odpovídá předpokladům. Dosti překvapivé jsou ovšem časy minimalizace funkcí, které příliš minimalizovat nejdou. Zatímco **pupik** i **btmin** mají jen lineární nárůst času výpočtu s počtem termů, roste doba výpočtu **espressa** přinejmenším kvadraticky a pro více než 20000 termů se už výsledku často nedočkáme.
Dále se potvrdil můj předpoklad, že minimalizace absorpcí na základě nadmnožin použitá v algoritmu **pupik** nebude tak efektivní jako ta mnou implementovaná v algoritmu **btmin**. U benchmarků s málo termy a řídkými DC jsou sice rozdíly jen nepatrné, ale s jejich rostoucím počtem se již výsledky liší i v desítkách procent.

6 Závěr

V průběhu práce se vyskytlo mnoho různých problémů, se kterými si bylo nutné poradit, z nichž nejsložitější se ukázal být problém minimalizace logických funkcí. Proto byl zdokonalen a použit nový slibný minimalizační algoritmus založený na manipulaci s ternárními stromy.

Tento algoritmus byl porovnán jak se svým předchůdcem, ze kterého vycházel, tak se sofistikovaným minimalizátorem logických funkcí **espresso**. Nedosahuje sice tak dobrých výsledků jako **espresso**, ale z hlediska minimalizace rozsáhlých vstupních dat by mohl najít uplatnění díky své rychlosti. Pokud by se podařilo nalézt způsob, jak v ternárních stromech provádět efektivně absorpce negace a detekovat consensus, mohl by se z této metody vyvinout plnohodnotný minimalizační algoritmus dosahující kvalit dnešních sofistikovaných minimalizátorů.

Dále byl patřičně otestován navržený algoritmus na úpravu logických funkcí do DNF a bylo prokázáno, že si navzdory své exponenciální časové a paměťové složitosti dokáže poradit i s velmi rozsáhlými vstupními daty.

Také byly navrženy a implementovány další požadované funkce pro práci se vstupními daty, umožňující provádění různých logických operací a upravování logických funkcí do různých tvarů.

Všechny cíle vytyčené v úvodu práce se tedy podařilo úspěšně splnit.

Mírnou odchylkou od původního zadání je pouze vypuštění CNF formátu, který byl po dohodě s vedoucím práce nahrazen univerzálnějším formátem VHDL.

7 Seznam literatury

[1] `http://service.felk.cvut.cz/courses/X36PAA/`.

[2] `http://lob.felk.cvut.cz/x36lob/`.

[3] `http://service.felk.cvut.cz/vlsi/prj/Benchmarks/`.

[4] Espresso - two-level boolean minimizer. University of California, Berkely.

[5] T. Měchura. *Parametrizovaný generátor náhodných booleovských funkcí.* BP ČVUT, FEL, 2006.

[6] P. Rucký. *Převod víceúrovňové logické sítě na dvouúrovňovou pomocí BDD.* DP ČVUT, FEL, 2007.

[7] M. Virius. *Programování v C++.* ČVUT, 2004.

A Seznam použitých zkratek

PLA Programmable logic array

VHDL Very high speed integrated circuit hardware description language

BLIF Berkeley logic interchange format

DC Don't care

DNF Disjunktivní normální forma

CNF Konjunktivní normální forma

B Uživatelská příručka

B.1 Překlad a spuštění programu

Program využívá pouze standardní knihovnu, takže je přeložitelný a spustitelný nezávisle na operačním systému. Pro překlad pod systémem Windows je možné použít makefile z přiloženého média.

B.2 Ovládání programu

B.2.1 Interaktivní používání programu

Pokud je program spuštěn bez parametru, vyžádá si od uživatele postupně:

1. Název vstupního souboru
2. Název výstupního souboru
3. Typ vstupního souboru
4. Typ požadované operace

Pokud název vstupního souboru bude obsahovat řetězec *.pla* , automaticky se nastaví typ vstupního souboru na PLA, v případě výskytu řetězce *.vhd* se typ vstupního souboru nastaví na VHDL.
Jednotlivé operace se vybírají zadáním příslušného čísla a lze je libovolně kombinovat.

Při operaci přidání nové výstupní funkce (*add function*) je nutné zadat vstupní řetězec ve tvaru `id <= V;` kde **id** udává pojmenování výstupní proměnné a **V** libovolný výraz. Čísla ve výrazu udávají čísla výstupních funkcí, kterými mají být nahrazeny. Tímto způsobem je možné zadat jedním vstupním řetězcem i několik příkazů za sebou (stačí, když je každý z nich korektně zakončen středníkem), ale v takovém případě je již lepší použít skript.

Příklad použití: `y4 <= (0 and 1) xor (2 or 3);`

Po zadání tohoto řetězce by se vytvořila nová výstupní proměnná **y4** a do vytvořeného výrazu by se dosadily výrazy příslušející funkcím s pořadovým číslem 0, 1, 2 a 3.

Při závěrečném ukládání výsledku je pak nutné zvolit, zda se mají uložit všechny, nebo jen nově přidané funkce.
Pokud je výsledek ukládán ve formátu VHDL, tak se v případě, že poslední operace před uložením byla test splnitelnosti, uloží do souboru také komentář se všemi ohodnoceními, pro které jsou jednotlivé funkce splnitelné.

Pro urychlení práce s programem lze zadat název vstupního i výstupního souboru parametrem. V případě, že je program spuštěn s jedním parametrem, bere se tento parametr jako název vstupního souboru. Přidání dalšího parametru interpretuje program jako název výstupního souboru.

B.2.2 Neinteraktivní používání programu

Tento způsob použití umožňuje vykonání libovolného skriptu zadaného parametrem.

Operace se vyvolá zadáním třech parametrů. První parametr udává název vstupního souboru, druhý parametr název výstupního souboru a třetí název souboru se skriptem.

Příklad volání programu: `booltool vstup.pla vystup.pla skript.txt`

Syntaxe skriptu je stejná jako u výrazů popsaných v předchozí sekci, ale je možné zadat více výstupních funkcí najednou.

Příklad obsahu skriptu:
```
y4 <= (0 or 1) and 3;
y5 <= 2 xor 3;
y6 <= 4 nand 5;
```

Tento skript by vytvořil nové výstupní proměnné **y4**, **y5** a **y6** a provedl by náhradu podle stejného principu, jak je popsáno v předchozí sekci.
Uložení výsledku proběhne automaticky.

Na ještě nezadefinované funkce se nelze dopředu odkazovat, pokud tak uživatel učiní, je programem upozorněn na chybu.

Poznámka: Vstupní i výstupní proměnné jsou číslovány od nuly.

C Obsah přiloženého média

adresář/soubor	popis
index.html	výchozí stránka projektu
readme.txt	postup instalace a spuštění programu
text	adresář s vlastním textem BP
exe	adresář s přeloženým programem
data	adresář s benchmarky pro testování
src	adresář se zdrojovými kódy programu

Tabulka C.1: Obsah přiloženého média