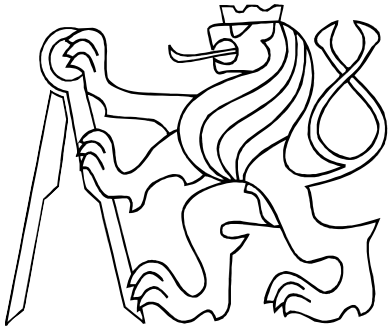
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky

KNIHOVNA PRO MINIMALIZACI LOGICKÝCH FUNKCÍ

Miroslav Šindelář

Bakalářská práce
2006



Vedoucí práce: Ing. Richard Šusta, PhD.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) 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 26. ledna 2006 …….

podpis

Abstrakt

Cíl této bakalářské práce spočívá ve vytvoření knihovny minimalizačních funkcí v jazyce C# prostředí .NET. Práce seznamuje s možnostmi zápisu logické funkce v symbolické formě, popisuje vybrané metody minimalizace, přibližuje jednotlivé postupy a algoritmy použité při realizaci knihovny. V knihovně je implementována funkce pro dekompozici rovnicově zadané funkce a heuristická metoda BOOM II.

Abstract

The aim of this work is to create the library of minimizing boolean functions in C# language for .NET platform. The work deals with possibilities of logical functions record in a symbolic form. It describes chosen methods of minimizing boolean functions, particular process and algorithms used on creating the library. In the library there are implemented functions for decomposition of equation function and heuristic method BOOM II.

Poděkování

Na tomto místě bych chtěl poděkovat Ing. Richardu Šustovi, PhD. za odborné vedení bakalářské práce. Dále pak Ing. Petru Fišerovi za poskytnutí veškerých informací a materiálů týkajících se metody BOOM.

Obsah

Kapitola 1
Úvod

Jedním z možných vyjádření logické funkce je úplný disjunktivní resp. konjunktivní normální tvar. Pro použití v praxi však tato forma obvykle není optimální, obsahuje totiž nadbytečné části, které navenek výstupní funkci nijak neovlivňují, jejich přítomnost při dalším zpracování je ovšem často spojena s vyšší náročností. Jako příklad můžeme uvést konstrukci elektronických zařízení, kde se každá booleovská funkce interpretuje soustavou logických obvodů. Každou nadbytečnou část zde tedy zaplatíme vyšší složitostí této soustavy a v důsledku toho i vyššími finančními náklady, nemluvě o velikosti zařízení. Optimalizační úpravy, které eliminují nadbytečné části logické funkce, souhrnně nazýváme minimalizací.

Následující text nás postupně seznámí s různými druhy minimalizačních metod, přiblíží některé formy zápisu logické funkce a na základě těchto poznatků osvětlí metody implementované v knihovně jazyka C# pro prostředí .NET

1.1 Definice základních pojmů

V této sekci heslovitě upřesníme některé základní pojmy, které později použijeme při popisu minimalizačních algoritmů.

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

term - v některých literaturách se o termech mluví jako o vektorech booleovské funkce
- složen z literálů mezi sebou navzájem vázaných logickým operátorem

- podle druhu operátoru jej dělíme na:

- **součinový** - neobsahuje operátor součtu “+“
  (P-term) př.: $x_1 \cdot x_2 \cdot \bar{x}_3$
- **součtový** - neobsahuje operátor součinu “.“
  (S-term) 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 generovaný proměnnými x_1, x_2, ..., x_n.
- 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.

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.: $f = (x_1 \cdot \bar{x}_2 \cdot x_3) + (\bar{x}_1 \cdot x_2 \cdot \bar{x}_3) + (x_1 \cdot \bar{x}_2 \cdot \bar{x}_3)$

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

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

OFF-set - termy, jejichž výsledná hodnota implikující funkci je rovna "0"

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

Kapitola 2

Vyjádření v symbolické formě

Pro zápis logické funkce v symbolické formě nalezneme v odborné literatuře definice mnoha formátů [1]. Při výběru zadávaného resp. výstupního formátu je proto důležité si předem ujasnit co od dané aplikace očekáváme. V následujícím textu bych chtěl přiblížit dvě možnosti vyjádření: dle formátů SLIF a PLA. Oba formáty ve své normě obsahují z důvodu univerzálnosti spoustu voleb a nastavení. Pro náš účel však postačí jen základní popis.

2.1 SLIF **(Structure Logic Intermediate Format)**

SLIF [2] je rovnicový hierarchický formát nabízející strukturální pohled na logickou funkci. K jeho výhodám patří především přehlednost zapsaných informací. Mezi nevýhody bych zařadil vyšší složitost a s tím i větší výpočetní náročnost při počítačovém zpracování. Díky své názornosti je vhodným formátem například pro konstrukci výukových programů.

2.1.1 Syntaxe (ASCII soubor)

```
.model main ;                          # definice modelu "main"
.inputs reset clock int0 int1 int2 ;   # list vstupních proměnných
.outputs out ;                         # list výstupních proměnných

out = reset' + clock * (in0' + (in1 * in2)) ;

.endmodel main ;                       # konec definice modelu "main"
```

při deklaraci operátorů jsou používány následující znaky

“ + “ ..... logický součet

“ * “ ..... logický součin

“ ' “..... negace

“ ~ “..... tzv.“don’t care“ operátor

každý výraz musí být ukončen středníkem.

2.2 PLA **(Programmable logic array)**

PLA formát slouží k dvouúrovňovému popisu vícevýstupové logické funkce. Původně byl navržen jako vstupní resp. výstupní formát pro minimalizační program ESPRESSO [3].

Funkce je zde popsána pravdivostní tabulkou a klíčovými slovy definujícími počet vstupních (.i *d*) a výstupních (.o *d*) proměnných.

Formát dále obsahuje nepovinná klíčová slova, pomocí kterých můžeme nastavovat parametry minimalizace. Jelikož jich existuje velké množství a popisovat je zde všechny není pro účel této práce nezbytné, uvedu pouze některé z nich. Podrobněji se s celým formátem můžeme seznámit v literatuře [1].

Přehled vybraných nepovinných klíčových slov:

.ilb *s1 s2 . . . sn* [1]	přiřazuje jména vstupním proměnným [2]
.ob *s1 s2 . . . sn*	pojmenovává výstupní proměnné [3]
.p *d*	definuje počet zadaných termů
.type *s*	volí jakým způsobem bude zpracována vstupní tabulka dané funkce
.e (end)	značí konec souboru

Možnosti zpracování dle volby *.type*

.type fd (přednastaveno) -Každý vstupní term s hodnotou výstupu “1” patří do on-setu dané funkce, “0” pak značí, že daný term nemá pro hodnotu funkce význam.

.type fr -Každý vstupní term s hodnotou výstupu “1” patří do on-setu dané funkce, “0“ term řadí do off-setu, “-” pak značí, že daný term nemá pro hodnotu funkce význam.

[1] Parametrem typu *s* za klíčovým slovem myslíme libovolný řetězec. Písmeno *d* pak značí číslo typu double.
[2] Deklarace jmen vstupních proměnných musí být uvedena až po deklaraci jejich počtu (.i *d*).
[3] Deklarace jmen výstupních proměnných musí být uvedena až po deklaraci jejich počtu (.o *d*).

2.2.1 **Syntaxe** (.pla soubor)

```
.i 5                    # def. počet vstupních prom.(povinný řádek)
.o 4                    # def. počet výstupních prom.(povinný řádek)
.p 10                   # def. počet termů (povinný řádek)
.ilb RESET q0 q1 q2 q3  # pojmenování vstupních proměnných
.ob d0 d1 d2            # pojmenování výstupních proměnných
.type fr                # volba typu zpracování


00000 0001              #****************************************
00001 0010              #
00010 0011              #
00011 0100              #
00100 0101              #             definice funkce
00101 0110              #
00110 0111              #
00111 1000              #
01000 1001              #
1---- 0000              #****************************************
.e                      # konec souboru
```

Kapitola 3

Primitivní metody minimalizace

V tomto oddíle se budu věnovat popisu základních minimalizačních metod. U každé z nich uvedu postup a možnosti jejího uplatnění.

3.1 Algebraická minimalizace

Metoda se používá především k úpravě logické funkce do standardizované formy (DNF, KNF), případně k minimalizaci funkcí s malým počtem vstupních proměnných při které využívá níže uvedených zákonů a základních logických operací.

komutativní zákony	$x+y=y+x,\ \ x\cdot y=y\cdot x$
asociativní zákony	$(x+y)+z=x+(y+z),\ (x\cdot y)\cdot z=x\cdot(y\cdot z)$
distributivní zákony	$(x+y)\cdot z=x\cdot z+y\cdot z,\ x\cdot y+z=(x+z)\cdot(y+z)$
zákon o vyloučeném třetím	$x+\overline{x}=1,\ x\cdot\overline{x}=0$
zákon o neutrálnosti nuly	$x+0=x$
zákon o neutrálnosti jedničky	$x\cdot 1=x$
zákon agresivity nuly	$x\cdot 0=0$
zákon agresivity jedničky	$x+1=1$
zákon o idempotenci prvků	$x+x=x,\ x\cdot x=x$
zákon absorpce	$x+x\cdot y=x$
zákon absorpce negace	$x+\overline{x}\cdot y=x+y,x\cdot(\overline{x}+y)=x\cdot y$
zákon dvojité negace	$\overline{(\overline{x})}=x$
De Morganovy zákony	$\overline{x}\cdot\overline{y}=\overline{(x+y)},\overline{x}+\overline{y}=\overline{(\overline{x}\cdot\overline{y})}$

Možnosti aplikace předvedeme na jednoduchém příkladu.

1. $f = \overline{((x + \bar{y}) \cdot z + y)} \cdot y$
2. $f = \overline{(x \cdot z + \bar{y} \cdot z + y)}.y$
3. $f = \overline{(x \cdot z + z + y)}.y$
4. $f = \overline{(z + y)} \cdot y$
5. $f = (\bar{z} \cdot \bar{y}) \cdot y$
6. $f = \bar{z} \cdot (\bar{y} \cdot y)$
7. $f = \bar{z}.(0)$
8. $f = 0$

Postup je následující: Nejdříve pomocí distributivního zákona odstraníme vnořenou závorku (1 ->2). Poté třetím termem absorbujeme negaci z druhého termu v uzávorkovaném výrazu (2 ->3). Pokračujeme absorpcí mezi prvním a druhým termem (3 ->4). De Morganovy zákony přemění negovaný součet na součin negovaných sčítanců (4 ->5). Následně přesuneme závorku za pomoci asociativního zákonu (5 ->6), poté použijeme zákon o vyloučeném třetím (6 ->7) a nakonec zákon agresivity nuly po kterém je již výsledek zřejmý (7 ->8).

3.2 Metoda Quine – Mc Cluskey

Metoda se používá především k minimalizaci funkcí s více než čtyřmi proměnnými. Nabízí jednoznačný algoritmus dovolující snadnou počítačovou implementaci. Bývá často základním stavebním prvkem většiny sofistikovanějších minimalizačních programů. Řeší minimalizaci funkcí zadaných v disjunktivní i konjunktivní normální formě a hodí se i pro skupinovou minimalizaci logických funkcí.

3.2.1 Princip minimalizace

Algoritmus metody Quine – Mc Cluskey [4] [5] by se dal rozdělit na dvě hlavní na sebe navazující části.

První fáze vychází z pravdivostní tabulky, ze které nás zajímají, v případě minimalizace do disjunktivní resp. konjunktivní normální formy, pouze termy definující "1" resp. "0" a všechny[4] nedefinované stavy výstupní funkce. Získaný soubor implikantů dále roztřídíme do skupin podle počtu lišících se vstupních proměnných. Každý prvek jedné množiny následně porovnáváme se všemi implikanty ze skupiny s jednou diferencí oproti porovnávanému prvku. V případě, že se porovnávané implikanty liší v právě jedné proměnné, lze konstatovat, že na této proměnné nezávisí. Můžeme tedy dané implikanty sloučit. Obecně lze napsat:

$$x_n \cdot x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_{i+1} \cdot x_i \cdot x_{i-1} \cdot \ldots \cdot x_1 \cdot x_0 + x_n \cdot x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_{i+1} \cdot \overline{x}_i \cdot x_{i-1} \cdot \ldots \cdot x_1 \cdot x_0 =$$
$$x_n \cdot x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_{i+1} \cdot (x_i + \overline{x}_i) \cdot x_{i-1} \cdot \ldots \cdot x_1 \cdot x_0 = x_n \cdot x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_{i+1} \cdot x_{i-1} \cdot \ldots \cdot x_1 \cdot x_0$$

Po sloučení je nutno nově vzniklý implikant vhodně označit. Celý proces se opakuje dokud nevznikne skupina implikantů již dále neslučitelných (přímých implikantů).

Některý přímý implikant může ve funkci pokrývat více implikantů. Pro minimální tvar výsledné funkce je tedy nutné vybrat pouze ty, které jsou nezbytné k sestavení minimálního výrazu odpovídajícího dané funkci (provést kontrolu pokrytí).

[4] Zahrnutí implikantů v nichž funkce není definována není nutné, přijdeme však o možnost jejich užití při zjednodušování. Z tohoto důvodu jsou do řešení zahrnuty a o jejich možném vypuštění se rozhodne později při kontrole pokrytí.

3.3 Karnaughovy mapy

Karnaughovy mapy [6] vycházejí z algoritmu Quine-McCluskey, pro hledání sousedních implikantů však používají grafickou metodu. Jsou vizuálně přehledné a proto vhodné pro rychlé zpracovaní bez použití výpočetní techniky.

Řeší minimalizaci funkcí zadaných v disjunktivní i konjunktivní normální formě. Metoda může být použita pro minimalizaci funkce s více než jedním výstupem, v takovém případě je ale třeba pro každý z nich vytvořit samostatnou mapu.

3.3.1 Postup při minimalizaci

Hodnotami výstupní funkce z pravdivostní tabulky vyplníme podle stavových indexů příslušná políčka v Karnaughově mapě (každé políčko v mapě představuje minterm resp. maxterm). Velikost tabulky tedy závisí na počtu kombinací vstupních proměnných (2^n, kde n je počet vstupních proměnných).

Vlastní minimalizaci provádíme sdružováním jedniček (DNF) resp. nul (KNF) v mapě do skupin, tzv. smyček.

Při tvorbě smyček dodržujeme tato pravidla:

1. Do smyčky přiřazujeme pouze vzájemně sousedící políčka stejné hodnoty (pro DNF “1”, pro KNF pak “0”). Přitom první a poslední sloupec (resp. řádek) mapy se také považují za vzájemně sousedící.
2. Počet políček ve smyčce musí být roven násobku 2 .
3. Smyčka musí mít tvar obdélníku nebo čtverce.
4. Každé políčko může být součástí více smyček (smyčky se mohou překrývat).
5. Snažíme se vytvářet co nejméně co největších smyček.
6. Neurčitý stav lze považovat za “1” u DNF (resp. “0” pro KNF) pokud tím dosáhneme větší smyčky. V ostatních případech neurčitý stav považujeme za “0” (resp. “1” u KNF).
7. Každá “1” (resp. “0” pro KNF) musí být uzavřena ve smyčce. Pokud ji nelze sloučit s jinou považuje se za smyčku obsahující jedinou buňku.

Při vyhodnocování mapy postupujeme po jednotlivých smyčkách. Pro každou z nich napíšeme logický výraz podle následujících pravidel:

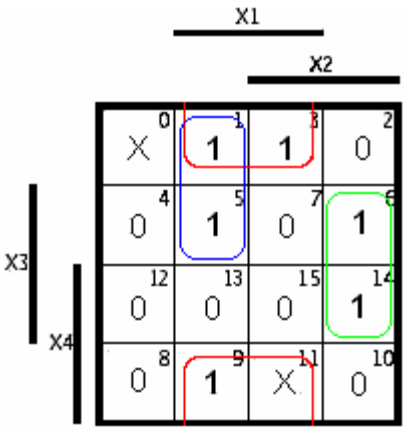
1. Pokud se proměnná v termech v dané smyčce nemění a její hodnota odpovídá hodnotě “1” pro DNF (resp. “0” pro KNF), pak proměnnou do výsledného výrazu zapíšeme v přímé formě.
2. Pokud se proměnná v termech v dané smyčce nemění a její hodnota odpovídá hodnotě “1” pro DNF (resp. “0” pro KNF), pak proměnnou do výsledného výrazu zapíšeme v negované formě.
3. V případě, že se hodnota proměnné v termech dané smyčky mění, danou proměnnou do výsledného výrazu neuvádíme.
4. Jednotlivé proměnné zapsané do výrazu mezi sebou logicky násobíme (DNF) resp. sčítáme (KNF).

Výsledné výrazy nakonec vypíšeme ve formě (DNF resp. KNF), pro kterou jsme danou mapu řešili.

Pro lepší pochopení zde uvedu jednoduchý příklad. Řešení bude uvedeno pro DNF.

K	x_4	x_3	x_2	x_1	f
0	0	0	0	0	x
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	x
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	0

Tabulka 3.1: Pravdivostní tabulka



Obrázek 3.1: Karnaughova mapa

$$f = x_1 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_2 \cdot x_3 + x_1 \cdot \overline{x}_2 \cdot \overline{x}_4$$

Kapitola 4

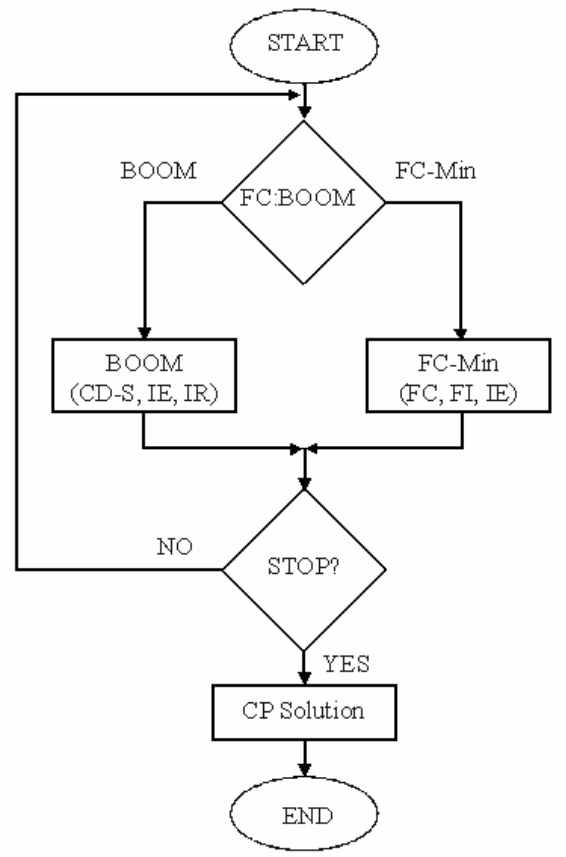
Sofistikované minimalizační programy

Jak sem již zmínil v předchozí kapitole, většina sofistikovaných minimalizačních programů vychází z metody Quine-McCluskey. Programová realizace této metody byla použita v minimalizačních systémech jako např. MINI [7], ESPRESSO a Scherzo [8]. V této kapitole se budu blíže věnovat programu BOOM II [9], který vyvíjí katedra výpočetní techniky fakulty elektrotechnické ČVUT.

4.1 BOOM II

BOOM II je velmi rychlý heuristický nástroj se schopností minimalizace funkcí velkého počtu vstupních i výstupních proměnných. Podporuje iterativní minimalizaci. Má malé paměťové nároky. S výhodou minimalizuje neúplně definované funkce. Jeho vstup je definován standardem PLA.

Princip této metody je založen na kombinaci starší verze BOOMu s nově vyvinutým algoritmem FC-Min. BOOM je vhodný pro řešení funkcí s jedním výstupem a velkým počtem vstupních proměnných. Algoritmus FC-Min jeho stavba předurčuje pro minimalizaci vícevýstupových funkcí. Jejich propojení popisuje vývojový diagram na následujícím obrázku.



Obrázek 4.1: Průběh minimalizace - BOOM-II

4.1.1 BOOM

Algoritmus můžeme rozdělit do dvou základních fází.

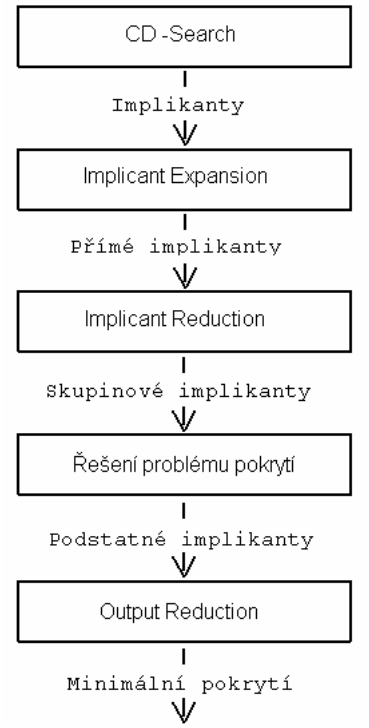
První fáze zajišťuje vytvoření skupiny přímých implikantů z funkce jedné výstupní proměnné. Probíhá ve dvou na sebe navazujících krocích.

Prvním je *CD-Search (Coverage - Directed Search),* generující implikanty potřebné k pokrytí on-setu logické funkce. Druhým pak *Implicant Expansion,* který z nich vytváří implikanty přímé.

V případě minimalizace funkce s větším počtem výstupních proměnných použijeme první fázi na každou výstupní funkci odděleně.

V druhé fázi vytvoříme z předchozích výsledků skupinové implikanty, což zajišťuje část *implicant reduction* a následně vyřešíme *skupinový problém pokrytí*. Pro dosažení konečného výsledku pak vzniklou matici znovu redukujeme funkcí *Output Reduction*.

Zjednodušený princip BOOMu je zachycen na následujícím obrázku.



Obrázek 4.2: Princip BOOMu

Z postupu použitého v první fázi vidíme, že efektivnost BOOMu se odvíjí od počtu výstupních proměnných. Z tohoto důvodu se hodí spíše pro minimalizaci jednovýstupových funkcí s velkým počtem vstupních proměnných. Se zvyšujícím se počtem výstupů se algoritmus stává časově náročným.

4.1.1.1 CD-Search

Princip tohoto kroku spočívá ve výběru nejvíce vyhovujícího literálu, který by mohl být přidán k některému dříve vytvořenému termu. Zvyšováním počtu literálů redukujeme n-rozměrnou hyperkrychli. Literály přidáváme, dokud se z daného termu nevytvoří implikant, což nastane v okamžiku kdy daná hyperkrychle neprotíná 0-term. Algoritmus se při hledání zároveň snaží, aby vygenerované implikanty pokryly co největší počet 1-termů. Abychom toho dosáhli, začínáme vždy od nejfrekventovanějšího literálu z on-setu vstupní funkce. Zvolený literál tvoří hyperkrychli s rozměrem *n-1*. Ta je implikantem za podmínky, že neprotíná 0-term. Pokud tomu tak není, celý proces opakujeme s dalším nejfrekventovanějším literálem v pořadí. Po vygenerování implikantu odstraníme všech-

ny 1-termy které jsou jím pokryty. Vznikne nám redukovaný on-set. Celá metoda se opakuje, dokud nemáme skupinu implikantů pokrývající on-set minimalizované funkce.

4.1.1.2 Implicant expansion (IE)

Implikanty generované v průběhu kroku CD-search nejsou přímé. K zvýšení šance, že bude k pokrytí všech 1-termů zadané funkce potřeba méně implikantů, musíme zvětšit jejich velikost. To jest odstranit literály z jejich termů. Pokud již neexistuje literál, který by mohl být odebrán z termu, jedná se o přímý implikant. Odebírání literálů je postupné a po každé expanzi se provádí kontrola, zda nový term neprotíná off-set dané funkce. Kontrola probíhá jednoduchým porovnáváním daného termu se všemi termy off-setu. O IE se v BOOMu starají tři metody, liší se složitostí a kvalitou získaných výsledků.

Exhausive IE – zkouší různé sekvence literálů.

Sequential IE - odebírá literály ze všech termů postupně jeden po druhém.

Multiple Sequential IE - zkouší všechny možnosti postupného odebírání pokaždé s jinou startovní pozicí.

Z předchozího vyplývá, že expanzí jednoho implikantu může vzniknout několik rozdílných přímých implikantů.

4.1.1.3 Implicant reduction (IR)

Všechny zjištěné přímé implikanty zkoušíme redukovat přidáváním literálů tak, abychom z nich vytvořili implikanty více než jedné výstupní funkce. Redukci provádíme dokud není zřejmé, že již daný implikant nelze použít pro více výstupních funkcí. Poté term uložíme a přiřadíme mu výstupní funkci. Pokud term neprotíná off-set žádné z výstupních funkcí, považujeme jej za přímý implikant.

4.1.1.4 Řešení problému pokrytí

Jelikož je BOOM určen především pro minimalizaci funkcí s velkým počtem vstupních proměnných, je pravděpodobné, že při řešení problému pokrytí budeme pracovat s velkým množstvím přímých implikantů. Použití exaktní metody pokrytí by v tomto případě bylo časově nefektivní. Z tohoto důvodu má BOOM v sobě pro řešení pokrytí implementované tři heuristické metody. První z nich - *LCMC* [10] do řešení preferuje

term pokrytý nejmenším počtem ostatních implikantů. Druhá - *Contributive Selection* opakovaně volí termy s maximem hodnotící funkce spočítané pro každý term. Poslední metoda - *Contributive Removal* k řešení používá stejnou hodnotící funkci jako metoda předchozí, vybírá však termy s nejnižším skóre. Toto je však vhodné jen pro malý počet implikantů.

4.1.1.5 Output Reduction

Vyřešením skupinového pokrytí získáme množinu implikantů, které budou součástí konečného řešení. K vytvoření požadovaného výstupu však nemusí být některé implikanty nezbytné (jsou nutné pro ostatní výstupy). Tyto implikanty mohou vytvářet nadbytečné mintermy ve výsledném řešení. V některých případech mohou být tyto mintermy prospěšné (např. snižují možnost výskytu hazardních stavů), pokud ale potřebujeme minimální řešení (například pro pozdější hardwarovou implementaci), můžeme je z řešení vyloučit. Toto se děje na konci minimalizace řešením problému pokrytí nezávisle pro každou výstupní funkci.

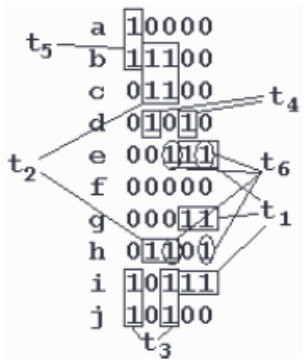
4.1.2 FC-Min

Zásadní odlišnost tohoto algoritmu od klasických minimalizačních metod je v jeho opačné strategii. Nejdříve vytvoří pokrytí on-setu dané funkce nezávisle na termech deklarujících vstupní funkci. Z tohoto pokrytí pak jsou odvozovány skupinové implikanty. Ke klasickému generování přímých implikantů zde tedy vůbec nedojde, což značně snižuje výkonové i paměťové nároky na hardware provádějící skupinovou minimalizaci.
Algoritmus pracuje ve třech následujících fázích.

4.1.2.1 Find Cover

V této fázi zkoušíme najít pokrytí on-setu obdélníkovým pokrytím všech “1” hodnot výstupní matice funkce. Pokrytím on-setu myslíme nalezení potenciálních implikantů, které by při splnění podmínek pro stanovené pokrytí mohly být součástí řešení. Hlavním cílem této fáze je učení počtu implikantů obsažených ve výsledném řešení. Metoda je heuristická, jelikož exaktní řešení by pro multivýstupové funkce nebylo možné nalézt v rozumném čase. Postup implementované heuristiky je obdobný řešení Karnaughovy mapy pro získání DNF, to jest hledáme elementy sestávající se z maximálního počtu “1”,

přičemž při každém nalezení elementu uvnitř obsažené “1” označíme. Ve vyhledávání pokračujeme dokud nejsou pokryty všechny “1”. Příklad pokrytí výstupní matice můžeme vidět na následujícím obrázku.



Obrázek 4.3: Příklad pokrytí matice

4.1.2.2 Find Implicants

V následující fázi odvozujeme implikanty ze získaného pokrytí a vstupní matice dané funkce. Je zřejmé, že v případě pokrytí konkrétního výstupního vektoru musí být v termu pokrývajícím tento vektor obsažen odpovídající vstupní vektor (vstupní vektor je implikantem výstupního vektoru). Minimální term, odpovídající danému pokrytí, je tedy konstruován jako minimální nadkrychle všech vstupních vektorů odpovídajících dané množině termů (termů tvořících pokrytí implikantů). Při tvorbě minimálního termu ovšem nesmí dojít k průniku s žádným vstupním termem, který není součástí dané množiny, jelikož by mohlo dojít k nechtěnému pokrytí “0”.

4.1.2.3 Input Expansion

Výstupní matice získaná z předchozího kroku již reprezentuje platné řešení. Avšak tyto výsledky je možno dále zjednodušit redukcí počtu literálů (expanzí vstupní matice výsledků).

Algoritmus této úpravy je založen na expanzi termů za pomoci heuristické “*pecking out*” metody, která se stará o vyváženou expanzi všech termů (odstraňování literálů probíhá v náhodném pořadí). Pokud expandovaný term nepokrývá žádnou “0” můžeme jej zařadit do výsledného řešení, v druhém případě odstraněný literál vrátíme zpět. Tento postup opakujeme dokud takto nejsou vyzkoušeny všechny literály.

4.1.3 Iterativní minimalizace

Oba algoritmy (BOOM a FC-Min) byly vyvinuty v iterativních verzích. Iterativní minimalizace je řízena náhodnou událostí a její výsledky tak mohou být při každém spuštění jiné. Kombinací implikantů z různých výsledků můžeme dosáhnout lepšího řešení. Proto jsou oba algoritmy spouštěny vícekrát a jejich výsledky ukládány do společného bufferu. Z tohoto bufferu je pak vytvořeno výsledné pokrytí.

Kapitola 5
Knihovna

V knihovně jsou implementovány dvě odlišné metody. První je kombinací algoritmu Quine-McCluskey s algebraickou minimalizací. Druhou je heuristický nástroj BOOM II. V této části stručně popíšu, jakým způsobem byly obě implementovány.

5.1 1. METODA – MinimizeString

Metoda je určena pro minimalizaci logické funkce popsané matematickým výrazem.

5.1.1 Syntaxe

Syntaxe je odvozena z formátu SLIF. Změny oproti standardu usnadňují zadání minimalizovaného výrazu v českém prostředí OS Windows. Minimální tvar funkce je v DNF.

Při deklaraci operátorů jsou používány následující znaky

“ + “ ..... logický součet

“ . “ ..... logický součin

“ ! “..... negace

Příklad zadání:

a.b+!d+!(e.!f.g+W).alfa odpovídá funkci $a \cdot b + \bar{d} + \overline{(e \cdot \bar{f} \cdot g + W)} \cdot alfa$

5.1.2 Implementace

Implementace metody je rozdělena do čtyř tříd. Všechny jsou obsaženy v souboru *MinimizeStringMethod.cs*. Jelikož jsou funkce jednotlivých tříd navzájem složitě provázány a jejich jednotlivé popisování by tím bylo velmi nepřehledné, omezím se pouze na blokový popis postupu minimalizačního algoritmu.

5.1.2.1 Detekce proměnných

Při postupném průchodu daným řetězcem separujeme jednotlivé proměnné, jejichž názvy pak funkce vrací ve formě pole řetězců. Dochází zde rovněž k syntaktické kontrole. Informace o případných chybách jsou předávány vyvoláním výjimky s textovým parametrem specifikujícím chybu. Vše je implementováno ve funkci *Variables (Třída LogOp).*

5.1.2.2 Převod do DNF

Srdcem této fáze je rekurzivní funkce *Translate (Třída ClassTranslate),* která zajišťuje překlad řetězce do pole P-termů. Její první instance je spuštěna vždy. K dalšímu spuštění dochází v momentě, kdy předchozí instance nalezne uzávorkovaný výraz. Logické operace, kterými jsou výsledky instance svázány s předchozí instancí, jsou implementovány ve funkcích *LogAnd(Třída LogOp)* a *Neg (Třída LogOp).*.

Funkce LogAnd definuje součin mezi dvěma poli P-termů.

Funkce Neg řeší negaci pole P-termů. Implementace negace je založena na De Morganových zákonech. To znamená, že každý P-term s více než jedním literálem je rozložen na součet negací těchto literálů. Všechny takto rozložené P-termy pak mezi sebou násobíme opakovaným voláním funkce *LogAnd*. Termy, které vystupují jako mezivýsledek násobení, minimalizujeme, abychom při zpracování většího počtu P-termů snížili nároky na operační paměť.

5.1.2.3 Minimalizace

Principem implementované minimalizace je vyhledávání dvou P-termů, které se liší pouze v jedné proměnné nebo na ně lze aplikovat zákon absorpce. V programu je zastoupena funkcí *MinimizeTerms (Třída MinimizeClass).*

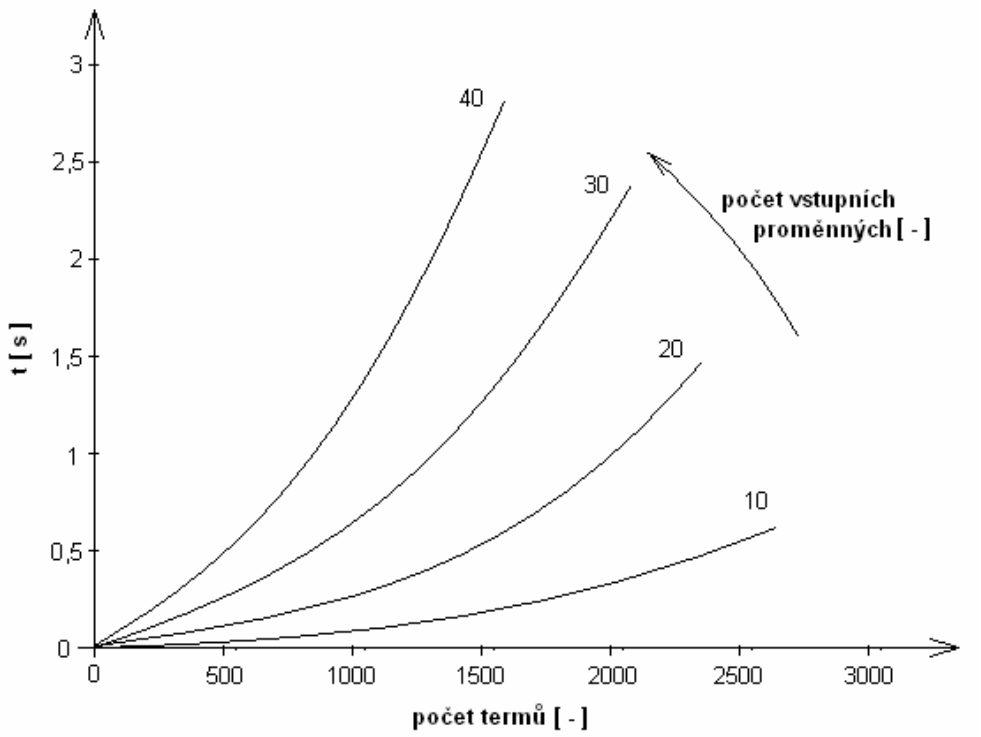
5.1.2.4 Sestavení výsledného řetězce

Minimální tvar získaný z předchozí fáze má podobu pole struktur *Term*. Převod na výsledný řetězec zajišťuje funkce *result (Třída LogOp).*

5.1.3 Výkonnost

Doba minimalizace metodou MinimizeString je závislá na složitosti zadané logické funkce. Hlavními ovlivňujícími faktory jsou zde počet vstupních proměnných, množství vnořených výrazů a jejich složitost.

Zahrnout všechny dříve zmíněné faktory do vstupních funkcí při testu, tak aby se výsledky daly jednoduše popsat (např. formou grafu), nelze. Omezil jsem se tedy jen na vstupní funkce v DNF, kde jsem volil množství mintermů a počet vstupních proměnných. Testování jsem provedl na notebooku s procesorem Intel Celeron Mobile 2,4 GHz a 738 MB RAM.



Obrázek 5.1: Výkonnostní graf funkce MinimizeString

5.2 2. METODA - BOOM II

O tomto nástroji již bylo mnoho řečeno v předchozí kapitole. V této sekci se proto zaměřím pouze na strukturální popis.

5.2.1 Implementace

Struktura BOOM II je rozdělena do systému souborů základních tříd, struktur a globálních funkcí. Jejich vzájemné provázání zajišťují především následující třídy.

5.2.1.1 Tpfnct

Základní stavební jednotkou je třída Tpfnct představující část logické funkce (term), tzn. kombinaci bitů (nul, jedniček či nedefinovaných hodnot), včetně výstupních hodnot funkce pro dané vstupní hodnoty. Strom jednotlivých termů tvoří právě logickou funkci, v našem případě strukturu třídy TboolFnct (popsanou níže).

```
public class  Tpfnct
{
        public char[] iv;          pole bitů vstupních hodnot
        public char[] ov;          pole bitů výstupních hodnot
        public int dim;            určuje počet aktuálních  nenulových prvků

pomocí následujících tří prvků se realizuje metoda pokrytí logické funkce
        public TCovs  cov;
        public int covlen;
        public int[] covfield;

pomocné proměnné:
        public int dvar;
        public int essential;
        public int tmp;
        public int tmp2;
        public int cvd;

        public int prime;          určuje, zda-li term  je přímým  implikantem
        public int origin;         pomocná informace o předchozích krocích během mini-
                                     malizace v rámci třídy  TboolFnct
        public int index;          index určující polohu v logické funkci inicializované v
                                     TboolFnct třídě
}
```

5.2.1.2 TboolFnct

Stěžejní a také nejobsáhlejší třídou celé knihovny je třída TboolFnct, která reprezentuje vlastní logickou funkci načtenou ze souboru formátu PLA. Kromě funkcí pro načtení, připojení, vymazání či různé manipulace s termy během minimalizace obsahuje funkce pro ukládání logické funkce do různých formátů včetně HTML.
Její nejpodstatnější datové prvky jsou:

```
public class  TboolFnct
{
        public int terms;
        public int inpvars;             počet vstupních  proměnných
        public int outvars;             počet výstupních proměnných
        public char[][] inames;         pole názvů vstupních proměnných
        public char[][] onames;         pole názvů výstupních proměnných

následující prvky charakterizují pole termu v mezikrocích během minimalizace
        public TPfnct[] onset; // onsets
        public TPfnct[] offset; // offsets
        public TPfnct[] dcset; // dc-sets

        public FnctInfo info;           informační struktura
        public DecompAss decomp;        pomocná struktura

pole indexů pro termy udávající specifickou vlastnost (nastaveno na 0, 1 nebo  - )
        public int[] onterms;
        public int[] offterms;
        public int[] dcterms;
        public int maxonterms;          čítač termů
        public TPfnct[] index;          termy logické funkce
        int split;                      identifikátor rozdělení logické funkce na termy
}
```

5.2.1.3 TermTree

Další třídu TermTree, mající na starosti správu jednotlivých termu a pomocných informací, reprezentuje především její hlavní datový prvek – třída TreeNode, která propojuje termy společných vlastností (1-termy, 0-termy a dc-termy z výše zmíněné TboolFnct třídy) a ještě je dále specifikuje.

```
public class TermTree
{
        public TreeNode root;
        int inpvars;
        long nodes;
        long items;
}
public class TreeNode
{
        public TreeNode on;
        public TreeNode off;
        public TreeNode dc;
        public int _value;
        public TPfnct ff;
}
```

5.2.1.4 Boom

Nejvíce nadřazenou třídou cele metody je třída Boom, která ovládá chod minimalizace, provazuje hodnoty termů s vlastnostmi a průběžnými informacemi, vytváří komplexní pojetí cele struktury. Po inicializaci booleovské funkce dochází k vnějšímu řízení a volání funkcí konkrétních tříd pro chronologicky správný běh programu v rámci minimalizace.

```
public class Boom
{
```

informační datové prvky obsahující většinou řídící příznaky:

```
        public MinimizeOptions mo;
        public TGauge gauge;
        public MinInfo info;
```

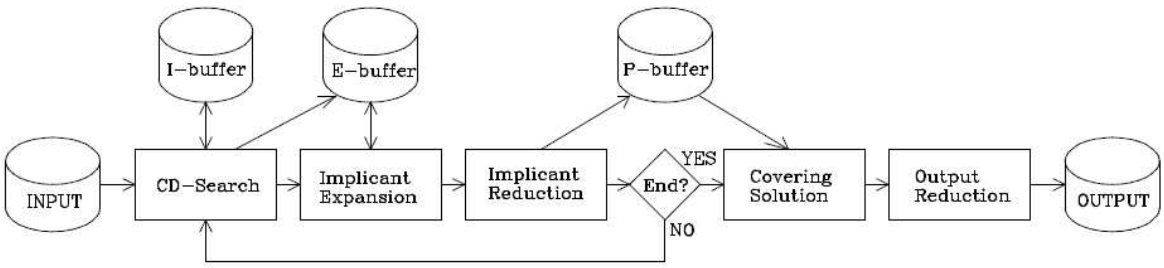
buffery sloužící k ukládání mezivýsledků:

(jejich význam dobře popisuje obr.5.2)

```
        TermTree[] i_buffer;
        TBoolFnct e_buffer;
        TBoolFnct r_buffer;
```

pomocné prvky:

```
        TermTree primes_ie;
        TermTree[] offtree;
        TermTree[] imp_test;
}
```



Obrázek 5.2: Způsob využití bufferů při minimalizaci

Jednotlivé kroky uvedené v popisu funkce v předchozí kapitole jsou implementovány v souborech s obdobnými názvy (např. *CD-search* najdeme v cd.cs).

5.2.2 Výkonnost

V době dokončování tohoto textu byl bohužel BOOM II ve fázi ladění. Proto zde uvádím výsledky verze implementované v jazyce C++ [11].

BOOM II					
vstup		BOOM		FC-Min	
test	i/o/p[5]	čas [s]	l/o/t[6]	čas [s]	l/o/t
cps	24/109/855	8.547	2110/758/184	6.937	1890/946/163
soar	83/94/779	26.062	2568/509/378	11.609	2445/549/353
cordic	23/2/2105	1.657	13825/914/914	12.562	13825/914/914
apex1	45/45/1440	27.641	1915/1025/229	8.875	1739/1103/206

Tabulka 5.1: Výsledky výkonnostního testu - BOOM II

Testy byly provedeny na klasickém PC s procesorem AMD AthlonXP 1700+ a 256MB RAM. Vstupní funkce byly čerpány ze sady standardních MCNC testů [12].

Rychlost mé verze bude zřejmě o něco nižší jelikož je napsána pro platformu .NET [13] .

[5] sloupec “i/o/p” udává počet vstupů, výstupů a termů příslušného testu
[6] sloupec “l/o/t” informuje o počtu literálů v termech řešení , ceně výstupu a o počtu produkovaných termů.

Kapitola 6
Závěr

Práce zdokumentovala vývoj knihovny pro minimalizaci logických funkcí.

První metoda byla navržena tak, aby mohla být velmi snadno implementována. Vstupní a výstupní struktura je velmi jednoduchá. Metoda se proto hodí např. pro využití v edukativním prostředí. Její vstupní syntaxe nesplňuje žádný standardní formát a tak nemohla být otestována žádnou ze standardních testovacích sad. Přestože jsou výsledky testů hůře porovnatelné s jinými minimalizačními metodami, výkonnost metody dobře popisují.

Metoda BOOM II byla do knihovny implementována pro její vysoký výkon a univerzálnost. Díky jejím vlastnostem a faktu, že je napsána v jazyce podporujícím platformu .NET se může dobře uplatnit jako síťový minimalizační nástroj. V současné době je metoda stále ve vývoji. Struktura knihovny je proto vytvořena tak, aby se případné změny daly snadno zanést do stávajícího řešení.

Pokračováním této práce by mohlo být například rozšíření knihovny o funkce které by zajišťovaly konverzi mezi použitými syntaktickými formáty, což by rozšířilo možnosti uplatnění implementovaných metod.

Závěrem mohu konstatovat, že jsem zadání této práce splnil bezezbytku.

Literatura

[1] http://embedded.eecs.berkeley.edu/pubs/

[2] G. Micheli, “Synchronous Logic Synthesis” ,Stanford University,1989.

[3] R.K. Brayton et al., “Logic minimization algorithms for VLSI synthesis”, Boston, MA, Kluwer Academic Publishers, 1984, 192 pp.

[4] W.V. Quine, “The problem of simplifying truth functions”, Amer. Math. Monthly, 59, No.8, 1952, pp. 521-531

[5] E.J. McCluskey, “Minimization of Boolean functions”, The Bell SystemTechnical Journal, 35, No.5, Nov. 1956, pp. 1417-1444

[6] Karnaugh, M .“The Map Method for Synthesis of Combinational Logic Circuits”, Transactions of the AIEE, Part I, Vol . 72, No . 9 ,pp . 593-599, 1953.

[7] S.J. Hong, R.G. Cain and D.L. Ostapko, “MINI: A heuristic approach for logic minimization”, IBM Journal of Res. & Dev., Sept. 1974, pp.443-458

[8] O. Coudert, “Doing two-level logic minimization 100 times faster”, Proc. of the sixth annual ACM-SIAM symposium on Discrete algorithms, 1995, pp.112-121

[9] P. Fišer, H. Kubátová, “Two-Level Boolean Minimizer BOOM-II”, Proc. 6th Int. Workshop on Boolean Problems (IWSBP'04), Freiberg, Germany, 23.-24.9.2004, pp. 221-228

[10] O. Coudert, “Two-Level Logic Minimization An Overview, Integration” The VLSI Journal, 17-2, pp. 97-140, Oct. 1994.

[11] J.Šádek, “Port programu BOOM pod platformu Linux”, Bakalářská práce - katedra počítačů FEL ČVUT, 2005.

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

[13] T. Archer, “ Myslíme v jazyku C# ”, Grada, Praha 2002.

[14] J. Bayer, Z. Hanzálek, R. Šusta, “Logické systémy pro řízení”, skriptum ČVUT Praha, 2000.

[15] Z. Diviš, Z. Chmelíková, I. Petříková, “Logické systémy pro kombinované a distanční studium”, skriptum VŠB Ostrava, 2003.

[16] J. Hlavička, P. Fišer, “BOOM - a Heuristic Boolean Minimizer”, Proc. ICCAD-2001, San Jose, Cal. (USA), 4.-8.11.2001, 439-442

[17] P. Fišer, J. Hlavička a H. Kubátová, “FC-Min: A Fast Multi-Output Boolean Minimizer”, Proc. Euromicro Symposium on Digital Systems Design (DSD'03), Antalya (TR), 3.-5.9.2003, pp. 451-451

[18] http://service.felk.cvut.cz/vlsi/prj/BOOM/

[19] http://portal.acm.org