České vysoké učení technické v Praze

Fakulta Elektrotechnická

Bakalářská práce

Výpočet komplementu logické funkce

Jan Kučera

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika

Obor: Informatika a výpočetní technika

Červenec 2010

Poděkování

Na tomto místě bych chtěl poděkovat vedoucímu práce Ing. Petru Fišerovi, Ph.D. za trpělivost a cenné rady.

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 Poličce dne 28. 5. 2010 ……

Abstract

The aim of this work is to implement algorithms used in Espresso minimization tool for computing complement of multiple-output Boolean functions. The implemented application should be able to process a two-level logic function given in PLA file. Another goal is the speed of mentioned algorithms, as the application is expected to process large input data. The final application will be tested on benchmark files and results will be compared with existing applications based on decision diagrams and “sharp product” algorithms.

Abstrakt

Tato práce si dává za cíl implementaci algoritmů používaných pro výpočet komplementu vícevýstupových logických funkcí v minimalizačním nástroji Espresso. Vstupem implementované aplikace má být dvouúrovňový popis funkce v soubor formátu PLA. Dalším cílem je rychlost implementovaných algoritmů, jelikož se předpokládá zpracování velkých vstupních dat. Rychlost implementovaných algoritmů bude porovnána s již exitujícími řešeními založenými na rozhodovacích diagramech a „sharp product“ algoritmech.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

1.1 Popis problému

Algoritmus pro výpočet komplementu logické funkce je součástí minimalizačních nástrojů jako je Espresso či BOOM, v nichž slouží jako součást algoritmů pro minimalizaci logických funkcí.

Výpočet komplementu logické funkce spadá do kategorie NP-těžkých algoritmických problémů – tj. problémů, které nelze řešit lépe než s exponenciální složitostí. Tento fakt je zapříčiněn tím, že počet termů logické funkce s rostoucím počtem vstupních proměnných narůstá exponenciálně rychle. To má za následek velkou časovou náročnost používaných algoritmů.

1.2 Specifikace cíle

Cílem práce by měla být implementace nástroje pro výpočet komplementu vícevýstupových logických funkcí daných dvouúrovňovým popisem ve formátu PLA (formát je blíže popsán v příloze). Tento nástroj bude implementován na základě jádra systému BOOM, které je součástí zadání práce. Kód výsledné aplikace by měl být platformě přenositelný a proto bude implementován pro běh v příkazové řádce, přičemž implementačním jazykem má být C++.

Výsledný nástroj by měl být schopen zpracovávat rozsáhlé vstupní soubory – tedy funkce s velkými počty vstupů a výstupů či funkce zadané velkým počtem vstupních termů – a to s nízkými požadavky na výpočetní čas.

Nástroj by měl být schopen přijímat na vstupu všechny typy formátu PLA a to včetně neúplně specifikovaných logických funkcí. Proto je žádoucí, aby uživatel mohl ovlivňovat způsob manipulace s „don’t care“ hodnotami – např. pomocí přepínačů.

Vzhledem k algoritmické složitosti problému a požadavku na zpracování velkých vstupních dat bude nutné umožnit uživateli zadání horní časové meze pro výpočet komplementu. Dalším požadavkem je, aby vypočtený komplement byl popsán co nejnižším počtem termů.

2 Popis problému

2.1 Booleovské funkce a jejich reprezentace

Booleovská (též logická) funkce n-proměnných je zobrazení n-tice prvků množiny booleovských hodnot {0, 1} do téže množiny booleovských hodnot {0, 1}. Označme-li tuto množinu jako B, pak libovolnou booleovskou funkci f($x_1 ... x_n$) lze popsat následujícím přepisem:

$$f : B^n \rightarrow B \quad \text{nebo} \quad f : \{0,1\}^n \rightarrow \{0,1\}$$

Booleovské funkce byly pojmenovány po britském matematikovi Georgovi Booleovi a hrají významnou roli při popisu a syntéze číslicových obvodů. Podrobnější informace o logických funkcích lze nalézt v [2] a [10]. Existuje několik možností pro vyjádření logické funkce. Následuje jejich výčet a krátký popis:

- pravdivostní tabulka
- n-dimenzionální krychle
- algebraický (booleovský) výraz

Pravdivostní tabulka je vyjádření, které ukazuje, jaké logické hodnoty booleovská funkce nabývá pro různá logická ohodnocení vstupních proměnných. Jestliže má booleovská funkce *n* proměnných, pak má pravdivostní tabulka právě 2^n řádků, neboť popisuje vždy všechny vrcholy z booleovského prostoru B^n. Pro velké počty vstupních proměnných je tudíž nevhodná. Tabulka 1 ukazuje příklad pravdivostní tabulky.

a_3	a_2	a_1	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Tabulka 1: Pravdivostní tabulka

Booleovskou funkci si lze představit také jako n-dimenzionální krychli B^n (*n* je počet vstupních proměnných), kde každý vrchol je jednoznačně určen ohodnocením vstupních proměnných a je mu přiřazena logická hodnota, kterou funkce pro dané ohodnocení nabývá. Dle tohoto ohodnocení lze vrcholy z B^n rozdělit do tří množin:

On-Set - je množina všech vrcholů (ohodnocení proměnných), pro které nabývá booleovská funkce hodnoty log. 1. Tato množina se často značí „f“.

Off-Set - je množina všech vrcholů (ohodnocení proměnných), pro které nabývá booleovská funkce hodnoty log. 0. Tato množina se často značí „r“.

DC-Set - je množina všech vrcholů (ohodnocení proměnných), pro které není hodnota booleovské funkce specifikována, neboli může nabývat hodnoty log. 1 nebo log. 0. Tato množina se často značí „d“. Funkce, které mají tuto množinu neprázdnou, nazýváme neúplně specifikované logické funkce.

Sjednocením těchto třech množin dostáváme celý B^n a dále platí, že průnikem libovolných dvou množin je prázdná množina.

$$f \cup d \cup r = B^n$$

$$f \cap d = \emptyset \quad f \cap r = \emptyset \qquad d \cap r = \emptyset$$

Algebraický výraz popisuje logickou funkci pomocí názvů proměnných a logických operací (AND, OR, XOR, NOT a další). Často se užívá tzv. „dvouúrovňový popis“, ve kterém je logická funkce vyjádřena tak, aby se operace logického součinu a logického součtu nacházely jen ve dvou úrovních.

Dále jsou zavedeny pojmy související s algebraickým vyjádřením booleovských funkcí:

literál - je proměnná nebo její negace

term - je výraz, který se skládá z literálů spojených jedinou binární logickou operací. Je-li touto operací logický součet, tak se jedná o součtový term. Je-li touto operací logický součin, tak se jedná o součinový term.

krychle - logický součin množiny literálů

implikant - krychle, pro kterou nabývá booleovská funkce hodnoty log. 1

minterm - je součinový term, který obsahuje všechny vstupní proměnné dané funkce

maxterm - je součtový term, který obsahuje všechny vstupní proměnné dané funkce

pokrytí funkce *f* - je množina krychlí reprezentující On-Set funkce *f*

Logická funkce vyjádřená algebraickým popisem se může nacházet v *úplné normální disjunktivní formě (UNDF)*, která představuje součet onsetových mintermů, nebo *úplné normální konjunktivní formě (UNKF)*, která je součinem offsetových maxtermů dané funkce. Pro funkci danou pravdivostní tabulkou Tabulka 1 jsou kanonické tvary následující:

UNDF: $f(a_3, a_2, a_1) = \overline{a_3}\,\overline{a_2}\,\overline{a_1} + \overline{a_3}\,\overline{a_2}\,a_1 + \overline{a_3}\,a_2\,a_1 + a_3\,\overline{a_2}\,\overline{a_1} + a_3\,a_2\,a_1$

UNKF: $f(a_3, a_2, a_1) = (a_3 + \overline{a_2} + a_1).(\overline{a_3} + a_2 + \overline{a_1}).(\overline{a_3} + \overline{a_2} + a_1)$

Dále se užívají *minimální normální formy*, které musejí navíc splňovat kriteria minimálního počtu termů. V případě součtu součinů se jedná o *minimální normální disjunktivní formu* (*MNDF*) a v případě součinu součtů o *minimální normální konjunktivní formu* (*MNKF*). Vyjádřením těchto forem se zabývá minimalizace logických funkcí.

2.2 Zavedení potřebných pojmů a principů

2.2.1 De Morganovy zákony

De Morganovy zákony popisují algebraické úpravy při negování termů a celých logických funkcí. Jedná se o následující pravidla:

$$\overline{a+b} = \overline{a}.\overline{b} \qquad \overline{(a.b)} = \overline{a} + \overline{b}$$

2.2.2 Maticová reprezentace pokrytí funkce

Pro následující algoritmy vyjádříme pokrytí funkce jako matici znaků „1“, „0“ a „-“. Každá krychle je jedním řádkem této matice. Každou krychli c^j z pokrytí F={c^1…c^m} logické funkce $f(x_1…x_n)$, vyjádříme následujícím způsobem:

$$c_i^j = \begin{cases} 0 & \text{značí, že se proměnná } x_i \text{ vyskytuje v krychli } c^j \text{ negovaná} \\ 1 & \text{značí, že se proměnná } x_i \text{ vyskytuje v krychli } c^j \text{ bez negace} \\ - & \text{značí, že se proměnná } x_i \text{ v krychli } c^j \text{ nevyskytuje} \end{cases}$$

Mějme logickou funkci s následujícím algebraickým popisem:

$$f(x_1, x_2, x_3, x_4) = x_1 x_2 \overline{x_3} + \overline{x_1} x_2 \overline{x_4} + x_1 x_2 x_3 x_4$$

tuto funkci můžeme popsat následující maticí:

$$F = \begin{bmatrix} 1 & 1 & 0 & - \\ 0 & 1 & - & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Obrázek 1: Maticová reprezentace pokrytí

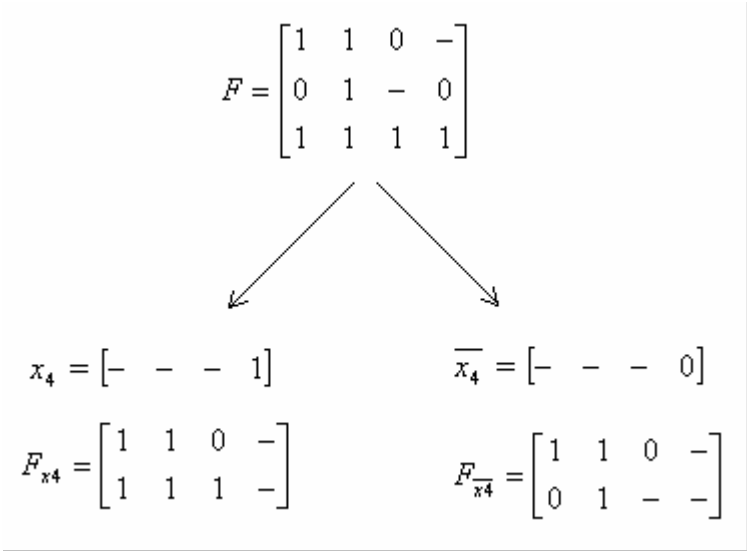
Tato reprezentace se více hodí pro operace jako je např. výpočet kofaktoru.

2.2.3 Kofaktor

Kofaktor f_p funkce *f*, která je dána pokrytím F={c^1…c^m}, vůči krychli *p* je dán sjednocením výsledků kofaktorů jednotlivých krychlí c^1…c^m vůči krychli *p*. Kofaktor c_p^i každé krychle c^i vůči krychli *p* určíme následujícím předpisem[1]:

$$(c_p^i)_k = \begin{cases} \emptyset & \text{pokud krychle pokud } c \cap p = \emptyset \\ - & \text{pokud } p_k=0 \text{ nebo } p_k=1 \\ c_k^i & \text{jinak} \end{cases}$$

Kofaktor představuje vytknutí proměnné z algebraické reprezentace funkce. Následuje příklad výpočtu kofaktoru funkce *f* z obrázku Obrázek 1 vůči x_4 a kofaktoru vůči $\overline{x_4}$:



Obrázek 2: Výpočet kofaktorů

Jestliže *f* a *g* jsou logické funkce, pak platí následující dvě rovnosti:

$$(f.g)_{x_j} = (f_{x_j}.g_{x_j})$$

$$(\overline{f})_{x_j} = \overline{(f_{x_j})}$$

2.2.4 Shannonův expanzní teorém

Shannonův expanzní teorém je pravidlo pro rozklad logické funkce vůči některé z jejich proměnných a má následujíc tvar:

$$f = x_i . f(x_1 ... x_{i-1}, 1, x_{i+1} ... x_n) + \overline{x_i} . f(x_1 ... x_{i-1}, 0, x_{i+1} ... x_n)$$

Pomocí kofaktorů lze tento teorém zapsat následujícím způsobem:

$$f = x_i . f_{x_i} + \overline{x_i} . f_{\overline{x_i}}$$

kde f_{x_i} resp. $f_{\overline{x_i}}$ jsou kofaktory vůči x_i resp. $\overline{x_i}$. Funkci *f* popsanou tabulkou Tabulka 1 s pokrytím F na obrázku Obrázek 1 tedy lze vyjádřit např. tímto způsobem:

$$f = x_4 . f_{x_4} + \overline{x_4} . f_{\overline{x_4}}$$

$$f = x_4 . (x_1 x_2 \overline{x_3} + x_1 x_2 x_3) + \overline{x_4} . (x_1 x_2 \overline{x_3} + \overline{x_1} x_2)$$

Expanzní teorém je jednou z klíčových myšlenek algoritmu pro výpočet negace logické funkce, neboť platí[1]:

$$\overline{f} = \overline{x_j} . (\overline{f_{\overline{x_j}}}) + x_j (\overline{f_{x_j}})$$

Důkaz této rovnosti byl z důvodu přehlednosti vynechán a lze ho najít v [1].

Tento teorém umožňuje redukovat problém výpočtu negace funkce *f* na podproblémy menších rozměrů – tedy na výpočet negace jednotlivých kofaktorů.

2.2.5 Unátní proměnná, unátní funkce

Logická funkce *f* je monotónně rostoucí v proměnné x_j, pokud se proměnná x_j ve funkci *f* nevyskytuje negovaná (všechny přímé implikanty obsahují tuto proměnnou buď bez negace nebo jako „don‘t-care“). Podobně, funkce *f* je monotónně klesající v proměnné x_j, pokud se tato proměnná ve funkci nevyskytuje bez negace (všechny přímé implikanty obsahují tuto proměnnou buď negovanou nebo jako „don‘t-care“) [1].

$$f = x_1 x_2 \overline{x_3} + x_1 \overline{x_2}\,\overline{x_4} + \overline{x_2}\,\overline{x_3} x_4$$

$$F = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 \end{matrix} \\ \begin{bmatrix} 1 & 1 & 0 & - \\ 1 & 0 & - & 0 \\ - & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Obrázek 3: Binátní funkce

Logická funkce je unátní neboli monotónní v proměnné x_j právě tehdy, když je monotónně rostoucí nebo monotónně klesající v proměnné x_j.[1] Funkce na obrázku Obrázek 4 je unátní v proměnných x_1 a x_3. V proměnné x_1 je tato funkce monotónně rostoucí, v proměnné x_3 je monotónně klesající. Logická funkce je unátní právě tehdy, když je unátní ve všech proměnných.

$$f = x_1 \overline{x_2}\,\overline{x_3} x_4 + x_1 \overline{x_2} x_4 + \overline{x_2}\,\overline{x_3} x_4$$

$$F = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 \end{matrix} \\ \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & - & 1 \\ - & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Obrázek 4: Unátní funkce

O funkci, která není unátní, říkáme že je binátní. Binátní funkce je binátní alespoň v jedné proměnné x_j – tj. tato proměnná se ve funkci vyskytuje v negované i nenegované formě.

2.2.6 Unátní rekurzivní paradigma

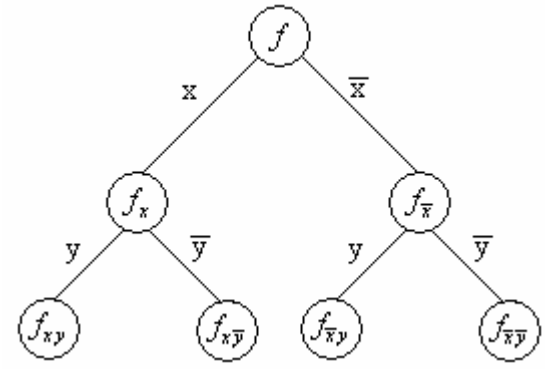
Unátní rekurzivní paradigma je princip, který je využíván v mnoha různých algoritmech pro zpracování logických funkcí – např. ověřování tautologie, průnik dvou funkcí nebo výpočet komplementu (tedy negace) logické funkce. Princip výpočtu nějaké operace (negace, kontrola tautologie, průnik dvou funkcí…) nad booleovskou funkcí spočívá v aplikaci Shannonova expanzního teorému – jedná se tedy o aplikaci této operace na kofaktory funkce a následné vyhodnocení celkového výsledku pomocí dílčích výsledků nad jednotlivými kofaktory. Jsou-li *f* a *g* logické funkce, pak rekurzivní paradigma lze popsat následovně (pro binární operaci v prvním případě, či unární operaci v případě druhém):

$$\text{operace(f, g)} = \text{kombinuj(x.operace(}\ f_x, g_x\text{), }\ \bar{x}\text{.operace(}\ f_{\bar{x}}, g_{\bar{x}}\text{))}$$

$$\text{operace(f)} = \text{kombinuj (x.operace(}\ f_x\text{), }\ \bar{x}\text{.operace(}\ f_{\bar{x}}\text{))}$$

kde operace *kombinuj()* provádí sjednocování dílčích výsledků.[1]

Rekurzivním aplikováním tohoto principu na jednotlivé kofaktory vzniká strom rekurzivního volání (Obrázek 5) v jehož listech se nacházejí unátní funkce (kofaktory). Při aplikování prováděné operace na tyto unátní funkce lze využít některých vlastností unátních funkcí a danou operaci tím urychlit.



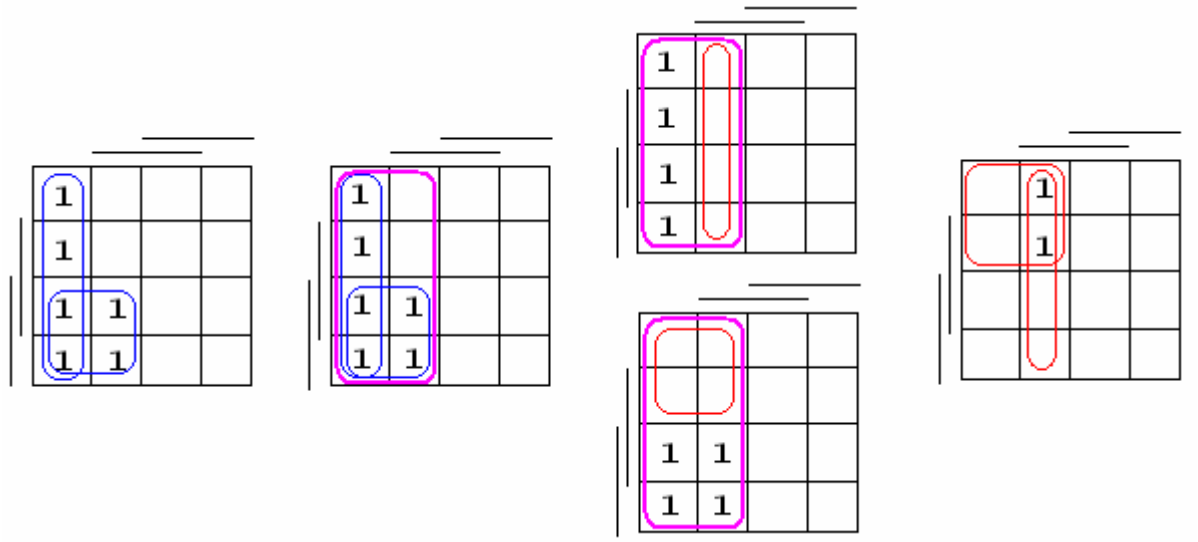
Obrázek 5: Strom rekurzivního volání

2.3 Existující řešení a jejich principy

2.3.1 Espresso

První z existujících řešení se nazývá Espresso. Jedná se o komplexní minimalizátor logických funkcí, který mimo jiné umožňuje výpočet komplementu vícevýstupových logických funkcí. Vstupním formátem je dvouúrovňový popis funkce v souboru PLA.

Výpočet komplementu v Espressu je založen na technice zvané *Sharp Product*. Základní operací je operace „sharp“, která operuje nad dvěma termy následujícím způsobem. Nejdříve je určen minimální term obsahující oba tyto operandy. Poté jsou určeny termy představující doplňky operandů k tomuto termu a jejich průnikem je získán hledaný výsledek operace *sharp*. Tento princip ilustruje Obrázek 6. Tato operace je prováděna na jednotlivých termech dané funkce a výsledky jsou průběžně minimalizovány. Více o tomto minimalizačním nástroji je možné nalézt v [7].



Obrázek 6: Sharp Product

2.3.2 Binární rozhodovací diagramy

Další nástroj byl implementován na základě binárních rozhodovacích diagramů (BDD – Binary Decision Diagrams). Tento nástroj vzniknul na Fakultě Elektrotechnické ČVUT v Praze jako diplomová práce Jana Bílka [11].

Binární rozhodovací diagram je stromová struktura, která má v kořeni a každém z vnitřních uzlů vstupní proměnnou. Z každého takového uzlu vedou hrany představující ohodnocení proměnné v uzlu, ze které tato hrana vychází. Listy tohoto stromu jsou ohodnocení této funkce pro jednotlivá vstupní ohodnocení, která dostaneme průchodem od kořene k danému listu. Více na toto téma je možné najít v [11].

3 Analýza a návrh řešení

3.1 Analýza a návrh řešení – popis užitých algoritmů

3.1.1 Výpočet komplementu

Jak již bylo zmíněno, vstupem pro implementovaný nástroj je vícevýstupová logická funkce. Funkce jsou zadány dvouúrovňovým popisem v disjunktivní normální formě. Tomuto vyjádření odpovídá formát PLA. Výpočet komplementu je implementován jako iterativní výpočet komplementů jednotlivých výstupní funkcí, které jsou následně zřetězeny. Termy, které implikují několik výstupních funkcí jsou sjednoceny. Popsaný postup je dán následujícím pseudokódem, který vychází z algoritmu popsaném v [1]:

```
// F – množina termů On-Setů všech výstupních funkcí
// D – množina termů DC-Setů všech výstupních funkcí
// m – počet výstupních funkcí
// Fi – On-Set i-té výstupní funkce
// Di - DC-Set i-té výstupní funkce

Procedure Complement (F, D)
{
        R = ø ;
        for (i = 1…m)
        {
           (Fi, Di) = ExtractOnsetDCset(F, D, i);
           Ri = SingleOutputComplement(Fi ∪ Di);
           R = R ∪ Ri ;
        }
        return R;
}
```

Pro každou výstupní funkci je určen On-Set a DC-Set funkcí *ExtractOnsetDCSet* a komplement je počítán jako doplněk k jejich sjednocení funkcí *SingleOutputComplement,* která počítá komplement k jedno-výstupové booleovské funkci.

3.1.2 Funkce SingleOutputComplement

Funkce *SingleOutputComplement* je tedy stěžejním kamenem výpočtu komplementu a je založena na *unátním rekurzivním paradigmatu* (angl. *unate recursive paradigm*). Tento princip využívá Shannonovu expanzi (2.2.4), při které je funkce *SingleOutputComplement* rekurzivně volána a při každém volání je vybrána tzv. štěpící proměnná, vůči které je pokrytí funkce děleno na stéle menší kofaktory. Tento princip je aplikován do té doby, než se jednotlivé kofaktory stanou unátními (2.2.5). Jejich komplement je poté vypočítán pomocí rychlejšího algoritmu pro unátní funkce. Následuje pseudokód této funkce, který vychází z algoritmu z [1]:

```
// Fc je kofaktor F vůči c
// Fxj je kofaktor F vůči j-té vstupní proměnné
// F(-xj) je kofaktor F vůči znegované j-té vstupní proměnné
Procedure SingleOutputComplement (F)
{
    R = ø ;
    if (F covers all subspace) return R;
    if (F is unate) return UnateComplement(F);
    if (F contains variable c only as 1 or only as 0)
    {
        F = c.Fc ;
        F' = c.Fc' = c' + Fc' ;
        R = R + c' ;
        F = Fc
    }
    j = BinateSelect(F);
    R1 = SingleOutputComplement( Fxj );
    R2 = SingleOutputComplement( F(-xj) ) ;
    return MergeWithContainment(R1, R2);
}
```

Při zavolání této funkce se nejdříve vyžívá speciálních případů, ve kterých je možné komplement vypočítat bez dalšího rekurzivního volání této funkce. Tyto případy jsou kontrolovány prvními třemi podmínkami ve výše uvedeném pseudokódu. Prvním případem je pokrytí obsahující krychli samých don't-care hodnot. Takovou funkci lze považovat za tautologii, neboť nabývá hodnoty log. 1 pro libovolné ohodnocení vstupních proměnných a jejím komplementem je prázdné pokrytí. Druhým speciálním případem je pokrytí, které je unátní. V tomto případě je komplement vypočítán funkcí *UnateComplement*. Třetím speciálním případem je pokrytí obsahující i-tou proměnnou buď jen znegovanou nebo jen bez negace – tj. ve sloupci matice jsou stejné hodnoty. V takové případě je možné vyjádřit F jako

$$F = c \cap F_c$$

a komplement určit De Morganovými pravidly následovně:

$$\overline{F} = \overline{c \cap F_c} = \overline{c} \cup \overline{F}_c$$

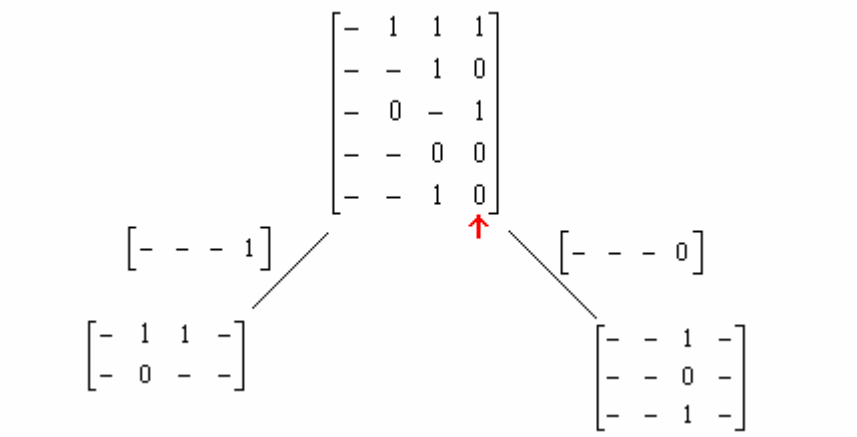
Tím se problém výpočtu komplementu pokrytí F redukuje na výpočet komplementu kofaktoru vůči této proměnné.

Nedojde-li však k žádnému ze speciálních případů, je užita metoda „rozděl a panuj“. Je zvolena štěpící proměnná pomocí funkce *BinateSelect*, vypočítají se kofaktory vůči zvolené proměnné a na každý kofaktor se rekurzivně zavolá funkce *SingleOutputComplement*. Tím je pokrytí F děleno na menší kofaktory, které se dalším dělením stávají více a více unátní a strom rekurzivního volání funkce *SingleOutputComplement* je v každém případě zakončen výpočtem komplementu pomocí jednoho ze speciálních případů.

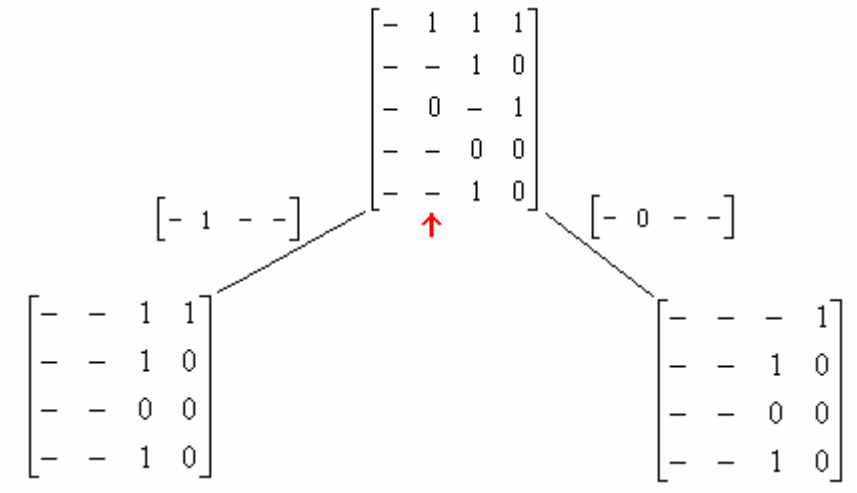
3.1.3 Heuristika BinateSelect

Z výše popsaného principu vyplývá, že volba štěpící proměnné, je jedním z hlavních faktorů ovlivňujících rychlost celého výpočtu, neboť ovlivňuje mohutnost kofaktorů vzniklých štěpením. V této části se tedy zaměřím na bližší popis heuristických funkcí, které by bylo možné užít, na popis jejich vlastností a vybrání vhodné heuristiky pro implementaci.

Následující obrázky ukazují správnou a špatnou volbu štěpící proměnné:



Obrázek 7: Správná volba štěpící proměnné



Obrázek 8: Špatná volba štěpící proměnné

Z uvedených příkladů vyplývá, že volba této proměnné bude silně záviset na počtech jedniček a nul v jednotlivých sloupcích, neboli na počtech negovaných a nenegovaných výskytů každé proměnné.

Funkce BinateSelect nejdříve vypočte kolik „1“ a „0“ se nachází na j–té pozici (odpovídá j-té proměnné) a pro každou proměnou vypočte minimum z těchto dvou hodnot. Pokud jsou všechna minima nulová, pak je funkce F unátní a štěpící se nevybírá. V opačném případě je vybrána štěpící proměnná pomocí heuristické funkce H. Volba této funkce je popsána dále. Následuje pseudokód funkce *BinateSelect*[1]:

```
// F je pokrytí funkce
// n je počet vstupních proměnných
// H je heuristická funkce
// p0[j] je počet negovaných výskytů j-té proměnné
// p1[j] je počet nenegovaných výskytů j-té proměnné

Procedure BinateSelect (F)
{
    for ( j = od 1 do n )
    {
        p0[j] = |{c^i ∈ F : c^i_j = 0}|;
        p1[j] = |{c^i ∈ F : c^i_j = 1}|;
        min01[j] = min(p0[j], p1[j]);
    }

    m = max_j(min01[j]);
    if (m == 0)
        return (-1);

    return H(p0[j], p1[j]);
}
```

V pseudokódu je vidět výpočet neznegovaných resp. znegovaných výskytů j-té proměnné – tyto hodnoty jsou ukládány do $p_0[j]$ resp. $p_1[j]$. Otázkou zůstává volba heuristiky H, která podle těchto počtů určí štěpící proměnnou. První možností, která se nabízí, je funkce H_1:

$$H_1 = \arg\max_j (p_0[j] * p_1[j])$$

Tato funkce bere v úvahu počty negovaných a nenegovaných výskytů jednotlivých proměnných, pro jednotlivé proměnné tyto počty násobí a vybírá proměnnou s maximální hodnotou.

Pro tuto heuristiku bylo provedeno měření na benchmarkových souborech [4] s výsledky v tabulce Tabulka 2. Měření bylo provedeno na třech kombinacích přepínačů, pomocí kterých může uživatel ovlivňovat průběh výpočtu. První kombinace představuje časově nejméně

náročnou možnost výpočtu – nedochází však k minimalizování počtu termů výsledného komplementu. Třetí možnost naopak počet termů redukuje za cenu vyššího výpočetního času. Druhá možnost představuje kompromis mezi předchozími dvěma možnostmi. Více o přepínačích je možné nalézt v kapitole 5.5 a dále v uživatelském manuálu (v příloze).

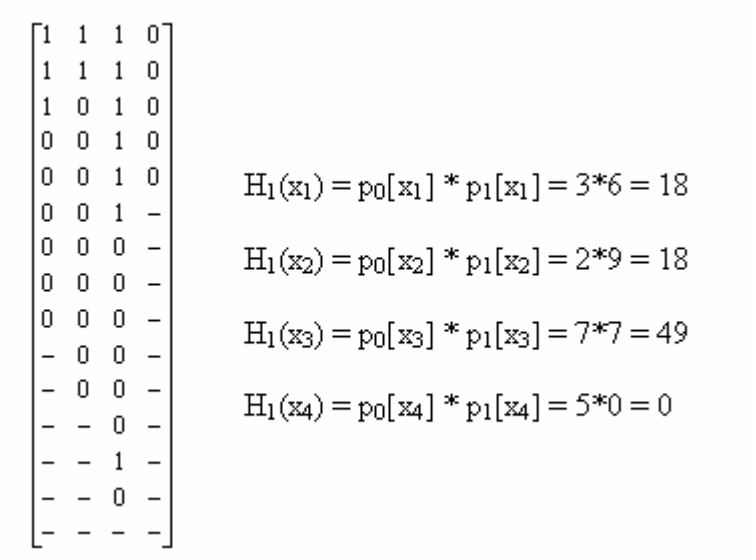
Význam sloupců tabulek je následující:

.type	typ PLA v daném souboru
.i	počet vstupních proměnných
.p	počet termů vstupního PLA
.o	počet výstupních funkcí
[kB]	velikost souboru v kB
t_1	doba výpočtu pro 1. kombinaci přepínačů (sekundy)
t_2	doba výpočtu pro 2. kombinaci přepínačů (sekundy)
t_3	doba výpočtu pro 3. kombinaci přepínačů (sekundy)
p_1	počet termů komplementu (1. kombinace přepínačů)
p_2	počet termů komplementu (2. kombinace přepínačů)
p_3	počet termů komplementu (3. kombinace přepínačů)

Soubor	type	i	o	p	p1	p2	p3	t1[s]	t2[s]	t3 [s]
b12.pla	fd	15	9	431	38	38	33	0,031	0,047	0,063
ex5.pla	fd	8	63	256	399	393	237	0,266	0,313	0,359
bcd.pla	fd	26	38	243	2810	2752	2010	0,094	0,187	0,813
max512.pla	fd	9	6	512	387	366	352	0,094	0,094	0,094
max128.pla	fd	7	24	128	266	261	225	0,063	0,047	0,063
max1024.pla	fd	10	6	1024	745	714	697	0,171	0,188	0,234

Tabulka 2: Měření pro heuristiku H1

Při aplikaci této heuristiky však zjistíme, že volba štěpící proměnné není vždy nejlepší. To ilustruje následující obrázek:



Obrázek 9: Nedostatek heuristiky H1

Jak je vidět na obrázku Obrázek 9, heuristika H_1 sice vybírá proměnou, která je nejvíce binátní (tj. nejčastěji se objevuje jak negovaná tak i neznegovaná), ale neuvažuje vždy vliv

„don't-care" hodnot dané proměnné – např. proměnné x_1 a x_2 mají stejné ohodnocení, ale ve sloupci proměnné x_2 je méně „don't-care" hodnot než u x_1, a proto by x_2 byla pro štěpení vhodnější než x_1. Vyšší počet těchto hodnot ve sloupci reprezentujícím štěpící proměnou zvyšuje mohutnost kofaktorů vůči této proměnné a tím se zpomaluje výpočet. Proto je nutné zvolit lepší heuristiku, která zváží i tento faktor. Takovou heuristickou funkcí by mohla být funkce H_2:

$$H_2 = \arg\max_j (p_0[j] + p_1[j])$$

Tato funkce sčítá znegované a neznegované výskyty pro jednotlivé proměnné a tím eliminuje nedostatek výše uvedené heuristiky H_1. Pro tuto heuristiku bylo provedeno měření, aby bylo možné tyto heuristiky porovnat:

Soubor	type	i	o	p	p1	p2	p3	t1[s]	t2[s]	t3 [s]
b12.pla	fd	15	9	431	37	37	32	0,031	0,046	0,046
ex5.pla	fd	8	63	256	371	362	218	0,265	0,250	0,312
bcd.pla	fd	26	38	243	2687	2581	1637	0,063	0,171	0,593
max512.pla	fd	9	6	512	201	205	197	0,093	0,093	0,109
max128.pla	fd	7	24	128	204	209	175	0,063	0,063	0,093
max1024.pla	fd	10	6	1024	406	411	401	0,156	0,171	0,203

Tabulka 3: Měření pro heuristiku H2

V porovnání s heuristikou H1 dává tato heuristika lepší výsledky – časová náročnost výpočtu komplementu je ve většině případů nižší a výsledné funkce jsou minimálnější (mají nižší počet termů).

Heuristika H_2 však přináší jiný nedostatek - funkce může vrátit stejnou hodnotu dvěma proměnným, pro které platí, že nejsou stejně binátní, ale součet jejich hodnot z polí $p_0[]$ a $p_1[]$ je stejný. Zmíněný problém je znázorněn na obrázku Obrázek 10:

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & - \\ 0 & 0 & 0 & - \\ 0 & 0 & 0 & - \\ 0 & 0 & 0 & - \\ 0 & 0 & 0 & - \\ 0 & 0 & 0 & - \\ 0 & - & 0 & - \\ 0 & - & 1 & - \\ 0 & - & 0 & - \\ - & - & - & - \end{bmatrix}$$

$$H_2(x_1) = p_0[x_1] + p_1[x_1] = 11+3 = 14$$

$$H_2(x_2) = p_0[x_2] + p_1[x_2] = 2+9 = 11$$

$$H_2(x_3) = p_0[x_3] + p_1[x_3] = 7+7 = 14$$

$$H_2(x_4) = p_0[x_4] + p_1[x_4] = 5+0 = 5$$

Obrázek 10: Nedostatek heuristiky H2

Za účelem eliminace této nedokonalosti byla vyzkoušena třetí heuristická funkce H_3, která kombinuje obě metody z heuristik H_1 a H_2. Nejvhodnější volba pro štěpící proměnnou je užita

heuristika H_2 a pokud je nalezeno více možností (tj. několik proměnných má stejné ohodnocení) je užita heuristika H_1 pro posouzení binátnosti těchto proměnných. Následují výsledky měření pro tuto heuristiku:

Soubor	type	i	o	p	p1	p2	p3	t1[s]	t2[s]	t3 [s]
b12.pla	fd	15	9	431	40	40	35	0,047	0,031	0,031
ex5.pla	fd	8	63	256	399	393	237	0,343	0,281	0,281
bcd.pla	fd	26	38	243	2687	2581	1637	0,094	0,141	0,813
max512.pla	fd	9	6	512	745	714	697	0,094	0,049	0,125
max128.pla	fd	7	24	128	387	366	352	0,094	0,078	0,063
max1024.pla	fd	10	6	1024	266	261	225	0,172	0,203	0,250

Tabulka 4: Měření pro heuristiku H3

Tímto měřením bylo zjištěno, že heuristika H3 však ve většině případů dosahuje horších výsledků než heuristika H_2. Zejména se jedná o generování větších počtů termů výsledných funkcí, což by mohlo zejména při zpracování velkých objemů dat negativně ovlivnit i čas potřebný pro výpočet. Z toho důvodu byla pro výslednou implementaci vybrána heuristika H_2.

3.1.4 Výpočet komplementu unátní funkce

Při výpočtu komplementu unátní funkce je využito vlastnosti, že se každá proměnná ve funkci f vyskytuje jen v jedné polaritě. To umožňuje převést maticovou reprezentaci pokrytí této funkce na matici booleovských hodnot M (tzv. personality matrix). Tyto hodnoty poté říkají jen to, zda se daná proměnná v dané krychli vyskytuje (hodnota „1“) nebo nevyskytuje (hodnota „0“). Je však potřeba uchovat informaci o tom, zda se proměnná vyskytuje v této unátní funkci negovaná či nikoli. Za tímto účelem je vypočtena tzv. překladová matice T (ang. *translation matrix*), která pro každou proměnnou uchovává informaci o její polaritě (hodnota „0“ značí monotónně klesající proměnnou, hodnota „1“ značí monotónně rostoucí proměnnou). Všechny další výpočty jsou prováděny nad těmito maticemi. Výsledkem je matice booleovských hodnot, která reprezentuje unátní funkci jakožto vypočtený komplement původní unátní funkce. Tato matice je přeložena pomocí překladové matice na běžnou maticovou reprezentaci pokrytí funkce $\overline{f}$. Následuje příklad výpočtu překladové a booleovské matice:

$$F = \begin{bmatrix} 1 & 1 & - & 0 \\ 1 & 1 & - & - \\ - & 1 & 0 & 0 \\ - & - & 0 & - \end{bmatrix} \longrightarrow M = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$
$$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$$
$$T = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$$

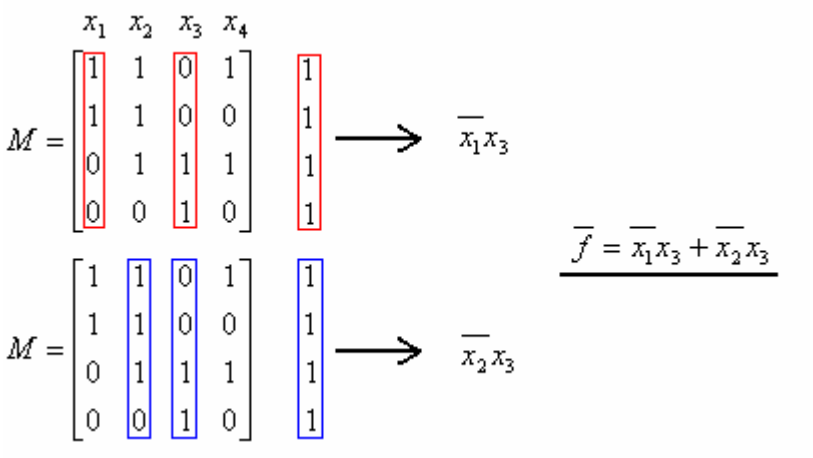
Obrázek 11: Výpočet matice M a matice T

Algoritmus výpočtu komplementu unátní funkce tedy probíhá nad popsanou maticí M.

První možností jak komplement vypočítat je tzv. *problém unátního pokrytí*, jehož principem je nalezení všech minimálních sloupcových pokrytí matice M – tj. chceme najít všechny minimální množiny sloupců *J*, aby pro každý *i*-tý řádek matice M platilo:

$$\exists j \in J : B_{ij} = 1$$

Tedy aby každý řádek matice byl pokryt alespoň jednou „1“ z vybraných sloupců, přičemž počet vybraných sloupců musí být minimální. Následuje příklad pro matici na obrázku Obrázek 11 :



Obrázek 12: Minimální sloupcové pokrytí

Druhou možností pro výpočet komplementu unátní funkce, je algoritmus založený na Shannonově expanzním teorému, který lze v případě unátní funkce upravit následujícím způsobem:

pro *f* monotónně rostoucí v x_j:

$$\overline{f} = \overline{x_j}.(\overline{f_{\overline{x_j}}}) + (\overline{f_{x_j}})$$

pro *f* monotónně klesající v x_j:

$$\overline{f} = x_j(\overline{f_{x_j}}) + (\overline{f_{\overline{x_j}}})$$

Jak již vzorce naznačují, tento algoritmus je opět rekurzivní a spočívá ve zvolení jedné ze vstupních proměnných, následného výpočtu komplementu pokrytí neobsahujícího tuto proměnnou a výpočtu komplementu pokrytí, z něhož je tato proměnná vytknuta. Poté je touto proměnnou vynásoben první z těchto výsledků a výsledky jsou nakonec sloučeny, čímž je získán komplement původní funkce. To vše probíhá nad maticí M. Strom rekurzivního volání má v listech následující speciální případy komplementace unárních funkcí:

a) Matice M obsahuje řádek samých „0“. To odpovídá řádku samých „-“ v pokrytí F – tedy celému universu. Komplementem je v tomto případě prázdná matice.

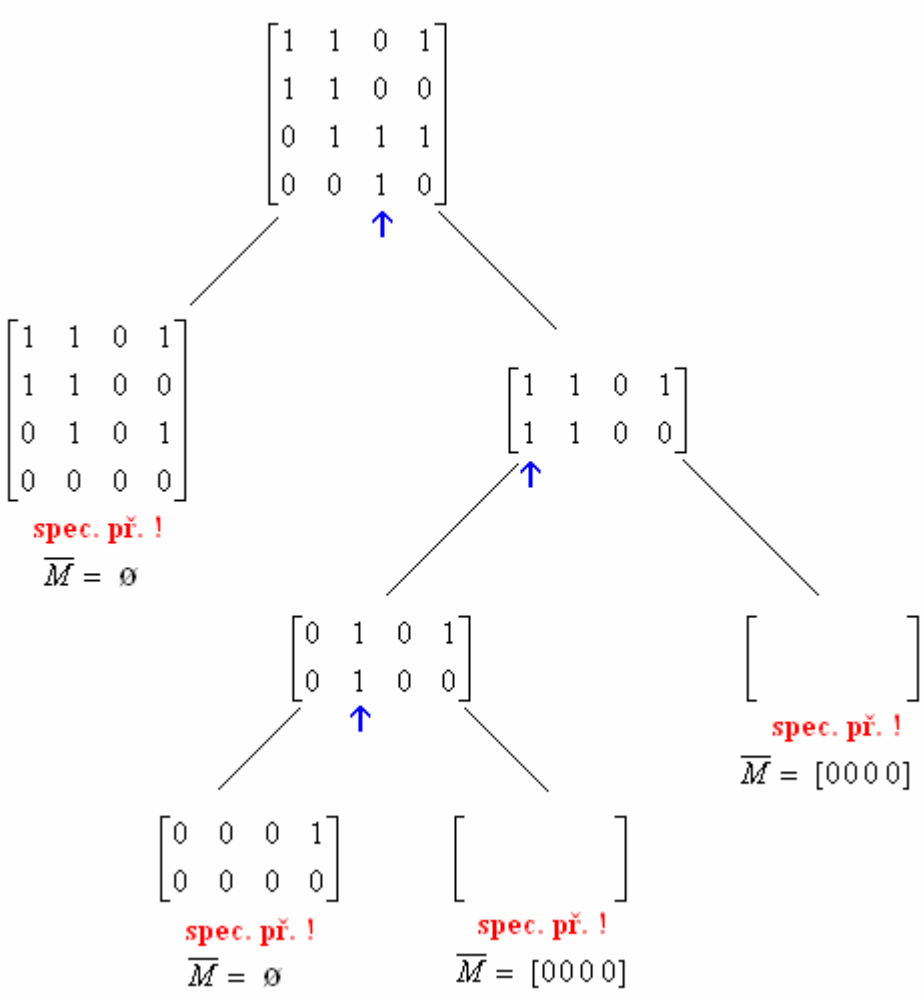
b) Matice M obsahuje jediný řádek. Komplement je vypočítá pomocí De Morganových zákonů.

c) Matice M je prázdná. Pak je výsledkem celé universum reprezentované řádkem nul.

Z těchto dvou popsaných řešení je dále pro implementaci zvolena možnost druhá (tedy rekurzivní výpočet pomocí Shannonovy expanze) a to kvůli nižší výpočetní náročnosti. Princip tohoto výpočtu je odpovídá algoritmu v [1].

Pro výběr proměnné je užita funkce *UnateVarSelect*. Tato funkce vybírá proměnnou tak, aby jedním rekurzivním průchodem bylo eliminováno co nejvíce krychlí. Nejdříve je nalezena největší krychle v celé matici M (tj. řádek s největším počtem „0“) a z proměnných, které se v této krychli nacházejí je vybrána ta, která se nejčastěji nachází v krychlích ostatních. To má za následek rychlou terminaci stromu rekurzivního volání pomocí speciálních případů a tím i rychlé provádění algoritmu.

Následuje ilustrace rozkladu pokrytí unátní funkce (přičemž její komplement je shodný s výsledkem první metody výpočtu):



Obrázek 13: Výpočet komplementu unátní funkce

3.1.5 Funkce MergeWithContainment

Funkce *MergeWithContainment* slouží ke slučování výsledků, které jsou vráceny rekurzivním voláním funkce *SingleOutputComplement*. Těmito výsledky jsou komplementy dvou kofaktorů vůči proměnné x_j. Slučování probíhá tak, že jsou nejdříve oba komplementy prohledány na shodné krychle a ke každý z nich je vynásoben proměnnou, vůči které jsou kofaktory vytvořeny. Návratovou hodnotou je sjednocení takto upravených kofaktorů.

Tato funkce provádí porovnání každé dvojice termů, kde první term náleží o prvního mezivýsledku a druhý term do mezivýsledku druhého. Při porovnání se testuje shoda termů i to, zda jeden z nich není obsažen ve druhém. Tento postup je prováděn s kvadratickou složitostí.

4 Realizace

4.1 Výběr jazyka a prostředí pro implementaci

Vzhledem k tomu, že implementovaný nástroj využívá součástí jádra systému BOOM, které bylo součástí zadání této práce, byl jazyk předem stanoven. Implementačním jazykem je C++, které poskytuje rychlost a současně přenositelnost kódu na jiné platformy. Pro implementaci je také využito knihoven STL, které poskytují šablony kontejnerových tříd (seznamy, vektory, fronty, …) a dále šablony algoritmů. Nástroj pro výpočet komplementu byl implementován v prostření Visual Studio Express Edition.

4.2 Datové struktury

4.2.1 Jádro BOOMu

Součástí zadání práce bylo jádro systému BOOM obsahující třídy, které reprezentují logické funkce a jejich elementy a zapouzdřují některé jejich operace. První z těchto tříd je třída *Cube*, která reprezentuje logickou krychli – tedy součinový term. Tato třída zapouzdřuje – mimo jiné - metody pro výpočet průniku s jinou krychlí, dále metodu pro porovnání a metody pro modifikaci vstupního ohodnocení.

Další třídou jádra BOOMu je třída SoPLA. Tato třída reprezentuje logickou jednovýstupovou logickou funkci ve tvaru „Sum-of-products“ – tedy jako součet součinových termů. Třída uchovává kolekci termů a navíc umožňuje pojmenovávat vstupní proměnné. Dále poskytuje metody pro přidávání a ubírání termů, zřetězení dvou funkcí a další metody.

Od třídy Cube dědí třída Term, která ke vstupnímu termu přidává řetězec výstupních ohodnocení pro jednotlivé logické funkce a dále metody pro manipulaci s tímto řetězcem.

Třída PLA poté reprezentuje formát PLA – jedná se tedy o množinu termů vícevýstupové logické funkce, přičemž každému termu je přiřazena jedna výstupní hodnota pro každou výstupní funkci. Ve třídě PLA jsou množiny On-Setu, DC-Setu a OFF-Setu jednotlivých funkcí uchovávány v kolekci *vector*. Třída rovněž pojmenovává vstupní proměnné i výstupní funkce a poskytuje metody pro načítání a ukládání do souboru.

4.2.2 Třída Complement

Všechny algoritmy pro výpočet komplementu byly implementovány v jako metody třídy Complement, která dále obsahuje atributy pro ovlivňování průběhu výpočtu. Tyto atributy se nastavují buď v konstruktoru nebo pomocí k tomu určených metod. Následuje výčet atributů třídy Complement a jejich význam.

Atribut *containment* určuje, zda bude v metodě MergeWithContainment popsané v kapitole 3.1.5 provedena i ta část algoritmu, ve které se hledají termy obsažené v termech jiných.

Atribut *merge_shared* říká, zda se po vypočtení komplementů jednotlivých výstupních funkcí má jejich zřetězení minimalizovat pomocí slučování sdílených termů.

Atribut *verbose* slouží k nastavení, zda bude během výpočtu vypisováno v jakém stavu se výpočet nachází. To se může hodit zejména při zpracování větších objemů dat, které je časově náročné.

Atributy *time_limit* a *time_limit_exceeded* slouží k omezení doby výpočtu limitem, který zadává uživatel.

Hodnoty těchto atributů - a tím i průběh výpočtu komplementu – může uživatel ovlivňovat z příkazové řádky pomocí přepínačů (blíže popsáno v příloze).

4.3 Implementované algoritmy

4.3.1 Metoda PLAComplement

Metoda *PLAComplement* slouží jako vstupní metoda celého implementovaného algoritmu. Jejím vstupem je instance třídy PLA (popsána v předchozí kapitole) a informace o způsobu manipulace s neurčenými stavy. Výstupem je instance třídy PLA, která přestavuje výsledný komplement.

Funkce byla implementována podle popisu v kapitole 3.1.1. Komplement PLA je vypočítán iterativně pro jednotlivé výstupní funkce a výsledky jsou zřetězeny.

4.3.2 Metoda SingleOutputComplement

Metoda SingleOutputComplement rekurzivně počítá komplement jednovýstupové logické funkce. Strom rekurzivního volání je zakončen speciálními případy, které jsou popsány společně s pseudokódem tohoto algoritmu v kapitole 3.1.2.

Rozdílem mezi zmíněným pseudokódem a implementovanou metodou je způsob, kterým jsou speciální případy zjišťovány. Pseudokód navrhuje postupné ověřování jednotlivých speciálních případů (nalezení sloupce stejných hodnot, zjištění zda je funkce unátní), z nichž každé ověření představuje průchod přes všechny termy v daném SoPLA. V případě že nenastane žádný z těchto případů, je nutné vybrat štěpící proměnnou, což představuje opětovné prohledání všech termů. Z tohoto důvodu bylo ověřování speciálních případů včetně výpočtu štěpící proměnné implementováno do metody *CheckSpecialCases*, která na jedno zavolání vrátí strukturu *MultiResult*, která obsahuje příznaky speciálních případů a dále štěpící proměnnou.

Na počátku je tedy volána metoda *CheckSpecialCases*, která svou návratovou hodnotou podává informaci o tom, zda došlo k některému ze speciálních případů. Pokud ano, určí, o který případ se jedná. Pokud ne, určí štěpící proměnnou a podle ní se dané pokrytí rozštěpí na kofaktory a dochází k rekurzivnímu volání.

4.3.3 Metoda UnateComplement

Strom rekurzivního volání funkce *SingleOutputComplement*, která byla popsána v předchozí kapitole je zakončen několika speciálními případy. Jeden z nich nastává, když je funkce v listu toho stromu unátní. Komplement této unátní funkce je vypočítán voláním *UnateComplement*.

Unátní funkce je nejdříve přeložena na tzv. „personality matrix“, na který je spuštěn rekurzivní výpočet založený na podobném principu jako funkce *SingleOutputComplement*. Takto získaný komplement je přeložen zpět a navrácen volající funkci.

4.3.4 Metoda Pers_Unate_Complement

Metoda *Pers_Unate_Complement* slouží k rekurzivnímu výpočtu komplementu unátní funkce. Vstupem však není její maticová reprezentace ale tzv. personality matrix (viz. kapitola 3.1.4). Princip je podobný jako u *SingleOutputComplement*. Funkce je rekurzivně volána nad stále zmenšujícími se kofaktory. Zakončení opět zajistí speciální případy (algoritmus je popsán v kapitole 3.1.4).

5 Testování

V této kapitole bude implementovaný nástroj testován na souborech benchmarku MCNC [8]. Testování je zaměřeno na rychlost výpočtu a minimálnost výsledných funkcí. Dále následuje porovnání s již existujícími nástroji popsanými v kapitole 2.3. Jediným rozdílem je, že pro porovnání časové náročnosti vůči technice „Sharp Product“ není užito originální Espresso [7], ale stejnojmenný nástroj [13], který vznikl jako bakalářská práce Martina Miklánka [12]. Důvodem je, že originální Espresso neumožňuje měřit čas běhu výpočtu. Oba nástroje jsou však založeny na stejných algoritmech.

5.1 Výsledky měření

V tabulce Tabulka 6 jsou shrnuty výsledky měření pro různé hodnoty přepínačů – tyto hodnoty jsou popsány v tabulce Tabulka 5.

Výsledné komplementy obdržené při testování byly porovnány s komplementy, které vypočítal minimalizační nástroj Espresso, následujícím způsobem:

Espresso.exe –Dverify file1 file2

Logickou ekvivalenci takto obdržených komplementů potvrzují sloupce -Dver1, -Dver2 a -Dver3 (opět pro jednotlivé kombinace přepínačů uvedené v tabulce).

První kombinace přepínačů provádí nejrychlejší výpočet – nedochází totiž k prohledání mezivýsledků na tzv. *containment*, tedy hledání a vyloučení termů obsažených v jiných termech. To vše ale na úkor minimality výsledného komplementu.

Druhá kombinace přepínačů již obsažené termy vyhledává minimalizuje výsledný komplement. Počet termů výsledného komplementu je nižší, ale časová náročnost je vyšší.

Třetí kombinace přepínačů provádí stejnou minimalizaci jako kombinace druhá, k tomu navíc prohledává výsledný komplement a vyhledává termy sdílené mezi výstupními funkcemi komplementu. Tyto termy pak slučuje. Toto prohledávání má však kvadratickou složitost, a tudíž jsou naměřené časy pro tuto kombinaci přepínačů značně vyšší než časy u předchozí kombinace. Bližší informace o přepínačích, jejich významech a použití je možné nalézt v uživatelském manuálu.

Následuje tabulka popisující přepínače použité pro jednotlivá měření a dále samotná tabulka naměřených hodnot.

(*) t_1:	Complement.exe -d so -n název_soubou -m off -c off -p off -v
(*) t_2:	Complement.exe -d so -n název_soubou -m off -c on -p off -v
(*) t_3:	Complement.exe -d so -n název_soubou -m on -c on -p off -v

Tabulka 5: Nastavení přepínačů při měření

Význam sloupců tabulek je stejný jako u tabulek v kapitole 3.1.3, a proto již není znovu popisován.

Complementer															Espresso	
Soubor	**.type**	**.i**	**.o**	**.p**	**[kB]**	**-Dver1**	**-Dver2**	**-Dver3**	**p1**	**p2**	**p3**	**t1[s] (*)**	**t2[s] (*)**	**t3 [s] (*)**	**merging [%]**	**p**	**čas [s]**
alu1.pla	fd	12	8	19	1	OK	OK	OK	20	20	20	0,015	0,031	0,031	0,00	20	0,000
alu4.pla	fr	14	8	1184	30	OK	OK	OK	667	667	609	0,047	0,047	0,063	25,40	609	0,031
apex1.pla	fd	45	45	206	19	OK	OK	OK	1941	1947	1591	0,156	0,156	1,230	87,32	1234	0.500
apex3.pla	fd	54	50	280	30	OK	OK	OK	1157	1158	928	0,093	0,203	0,578	64,88	756	0,359
apex4.pla	fd	9	19	438	14	OK	OK	OK	1815	1809	1619	0,078	0,171	0,453	62,25	1471	0,218
apex5.pla	fd	117	88	1227	250	OK	OK	OK	2547	2166	2084	0,421	0,546	2,328	76,55	1622	0,953
b12.pla	fd	15	9	431	12	OK	OK	OK	37	37	32	0,031	0,046	0,046	0,00	29	0,000
bca.pla	fd	26	46	301	23	OK	OK	OK	4257	4185	2561	0,109	0,203	1,406	85,56	2583	0,796
bcb.pla	fd	26	39	299	20	OK	OK	OK	3301	3188	2065	0,140	0,171	0,968	82,33	1929	0,625
bcc.pla	fd	26	45	245	18	OK	OK	OK	3412	3300	2031	0,093	0,203	0,984	79,37	1855	0,750
bcd.pla	fd	26	38	243	16	OK	OK	OK	2687	2581	1637	0,063	0,171	0,593	71,16	1473	0,359
ex5.pla	fd	8	63	256	19	OK	OK	OK	371	362	218	0,265	0,250	0,312	19,87	134	0,109
ex7.pla	fd	16	5	123	3	OK	OK	OK	176	176	174	0,016	0,031	0,031	0,00	173	0,015
f51m.pla	fd	8	8	256	5	OK	OK	OK	79	79	79	0,047	0,063	0,078	19,87	101	0,063
in0.pla	fd	15	11	138	4	OK	OK	OK	415	379	328	0,016	0,046	0,046	0,00	269	0,046
in1.pla	fd	16	17	110	5	OK	OK	OK	1262	1224	788	0,109	0,171	0,312	45,19	403	0,093
in2.pla	fd	19	10	137	5	OK	OK	OK	368	372	333	0,046	0,062	0,093	33,33	263	0,031
max1024.pla	fd	10	6	1024	20	OK	OK	OK	406	411	401	0,156	0,171	0,203	15,76	470	0,078
max128.pla	fd	7	24	128	5	OK	OK	OK	204	209	175	0,063	0,063	0,093	32,80	173	0,093
max512.pla	fd	9	6	512	10	OK	OK	OK	201	205	197	0,093	0,093	0,109	14,68	237	0,062

Tabulka 6: Výsledky měření na souborech MCNC

Ve sloupci s názvem *merge* tabulky Tabulka 6 je vidět přibližný procentuální podíl mergování na celkové době výpočtu, v případě zapojení mergování do výpočtu (tj. třetí kombinace výše uvedených přepínačů). Tato procedura prochází všechny termy vypočteného komplementu a hledá termy sdílené mezi jednotlivými výstupními funkcemi – což je časově náročné, jak ukazuje výše uvedená tabulka. Obecně lze říci, že čas spotřebovaný tímto výpočtem rychle vzrůstá s narůstajícím počtem termů komplementu. Porovnáme-li však tyto procentuální hodnoty u dvou komplementů s relativně blízkým počtem termů, zjistíme, že u komplementu s vyšším počtem termů nemusí nutně být tato hodnota vyšší. To ukazuje, že průběh mergování nezávisí jen na počtech termů komplementu, ale také na konkrétních termech tohoto komplementu.

Dalším cílem testování bylo změřit počty generovaných termů a ověřit, zda je vyjádření výsledného komplementu minimální, a provést porovnání s Espressem. Vzhledem k veliké časové náročnosti hledání minimálního pokrytí logické funkce bylo toto měření provedeno na nižší počtu souborů než měření předchozí. Výsledky jsou shrnuty v následující tabulce:

Complementer									**Espresso**	
Název souboru	**type**	**i**	**o**	**p**	**[kB]**	**p1**	**p2**	**p3**	**p**	**p -Dexact**
alu1.pla	fd	12	8	19	1	20	20	20	20	20
alu4.pla	fr	14	8	1184	30	667	667	609	609	393
b12.pla	fd	15	9	431	12	37	37	32	29	27
ex5.pla	fd	8	63	256	19	371	362	218	134	72
ex7.pla	fd	16	5	123	3	176	176	174	173	169
f51m.pla	fd	8	8	256	5	79	79	79	101	76
in0.pla	fd	15	11	138	4	415	379	328	269	107
in1.pla	fd	16	17	110	5	1262	1224	788	403	139
in2.pla	fd	19	10	137	5	368	372	333	263	118
max1024.pla	fd	10	6	1024	20	406	411	401	470	250
max128.pla	fd	7	24	128	5	204	209	175	173	78
max512.pla	fd	9	6	512	10	201	205	197	237	134

Tabulka 7: Výsledky měření na souborech MCNC

Výsledkem měření pro Espresso jsou dvě hodnoty, z nichž první (tj. sloupec *.p*) udává počet termů při standardním výpočtu komplementu s přepínači *Espresso.exe -o r* a druhý sloupec udává počet termů, které vrátí Espresso při spuštění s parametrem *–Dexact*. V tomto případě Espresso provádí exaktní minimalizaci, jejíž výsledkem je minimální vyjádření dané logické funkce. Tímto měřením tedy byly zjištěny minimální počty termů, které budou dále sloužit pro porovnávání dříve zmíněných nástrojů pro výpočet komplementu.

5.2 Porovnání s Espressem

Naměřené hodnoty, které jsou shrnuty v tabulce Tabulka 6, ukazují časy výpočtů měřených na jednotlivých benchmarkových souborech pro moji aplikaci i pro Espresso. Z časů měřených pro tyto dva nástroje vyplývá, že můj nástroj podává horší výsledky než Espresso – k tomu dochází zejména při zpracování rozsáhlých vstupních dat. Důvodem lepších výsledků Espressa může být použití kvalitnějších heuristických funkcí pro výběr štěpících proměnných nebo užití rychlého minimalizačního algoritmu v průběhu výpočtu. V některých případech je však moje aplikace rychlejší – zejména při zpracování menších vstupních dat s vypnutou funkcí pro mergování. Při zapnutí mergovací funkce za účelem hledání menšího pokrytí komplementu pak dochází k velkému nárůstu spotřebovaného času a můj nástroj podává horší časové výsledky než Espresso.

Hodnoty z tabulky Tabulka 7 porovnávají počty termů generovaných komplementů s jejich minimálním pokrytím. Tato tabulka ukazuje, že ke generování minimálního pokrytí obecně nedochází (resp. dochází jen někdy – nejčastěji u malých objemů vstupních dat). Při testování však bylo zjištěno, že minimální pokrytí není generováno ani minimalizačním nástrojem Espresso.

Při porovnání minimálnosti výsledků s výsledky Espressa lze prohlásit, že můj nástroj v některých případech generuje komplement, který má nižší počet termů než komplement Espressa – např. soubory *max512.pla*, *max1024.pla* a *f51m.pla* ve výše uvedené tabulce. V jiných případech je tomu však naopak. Rozdílnost výsledků lze zdůvodnit např. použitím jiných heuristických funkcí.

5.3 Porovnání s BDD

Následující tabulka ukazuje naměřené hodnoty pro výpočet komplementu založený na principu BDD (blíže popsán v kapitole 2.3.2). Význam sloupců je stejný jako u předchozích tabulek a proto není opět popisován. K měření na BDD byly užity stejné soubory jako při měření na Espressu.

Complementer								BDD	
Název souboru	**.type**	**.p1**	**.p2**	**.p3**	**t1[s]**	**t2[s]**	**t3 [s]**	**.p**	**t[s]**
alu1.pla	fd	20	20	20	0,015	0,031	0,031	**15**	**0.20**
alu4.pla	fr	667	667	609	0,047	0,047	0,063	**546**	**0.22**
b12.pla	fd	37	37	32	0,031	0,046	0,046	**32**	**0.20**
ex5.pla	fd	371	362	218	0,265	0,250	0,312	**165**	**0.20**
ex7.pla	fd	176	176	174	0,016	0,031	0,031	**123**	**0.20**
f51m.pla	fd	79	79	79	0,047	0,063	0,078	**77**	**0.23**
in0.pla	fd	415	379	328	0,016	0,046	0,046	**195**	**0.22**
in1.pla	fd	1262	1224	788	0,109	0,171	0,312	**727**	**0.27**
in2.pla	fd	368	372	333	0,046	0,062	0,093	**229**	**0.23**
max1024.pla	fd	406	411	401	0,156	0,171	0,203	**322**	**0.20**
max512.pla	fd	201	205	197	0,093	0,093	0,109	**165**	**0.20**

Tabulka 8: Měření na BDD

Z tabulky je vidět, že na většině použitých benchmarkových souborů je moje aplikace rychlejší než BDD. Při zadání velkých vstupních souborů však dává lepší výsledky aplikace

BDD, neboť u mojí aplikace dochází s rostoucím objemem vstupních dat k rychlejšímu nárůstu spotřebovaného výpočetního času.

5.4 Celkové srovnání

Pro přehlednost je dále uvedena tabulka porovnávající všechny výše zmíněné aplikace pro výpočet komplementu – tedy Espresso, BDD a moji aplikaci.

Complementer								Espresso		BDD	
Soubor	**.type**	**p1**	**p2**	**p3**	**t1[s]**	**t2[s]**	**t3 [s]**	**p**	**t[s]**	**p**	**t[s]**
alu1.pla	fd	20	20	20	0,015	0,031	0,031	20	0,000	**15**	**0.20**
alu4.pla	fr	667	667	609	0,047	0,047	0,063	609	0,031	**546**	**0.22**
b12.pla	fd	37	37	32	0,031	0,046	0,046	29	0,000	**32**	**0.20**
ex5.pla	fd	371	362	218	0,265	0,250	0,312	134	0,109	**165**	**0.20**
ex7.pla	fd	176	176	174	0,016	0,031	0,031	173	0,015	**123**	**0.20**
f51m.pla	fd	79	79	79	0,047	0,063	0,078	101	0,063	**77**	**0.23**
in0.pla	fd	415	379	328	0,016	0,046	0,046	269	0,046	**195**	**0.22**
in1.pla	fd	1262	1224	788	0,109	0,171	0,312	403	0,093	**727**	**0.27**
in2.pla	fd	368	372	333	0,046	0,062	0,093	263	0,031	**229**	**0.23**
max1024.pla	fd	406	411	401	0,156	0,171	0,203	470	0,078	**322**	**0.20**
max512.pla	fd	201	205	197	0,093	0,093	0,109	237	0,062	**165**	**0.20**

Tabulka 9: Porovnání aplikací

5.5 Využití přepínačů

Naměřené hodnoty shrnuté v Tabulka 6 a Tabulka 7 také ukazují vliv jednotlivých přepínačů na rychlost výpočtu a minimálnost vypočteného komplementu.

První kombinace přepínačů z tabulky Tabulka 5, je vhodná zejména ve chvíli, kdy očekáváme relativně rychlý výpočet, přičemž větší počet termů komplementu nevadí. Druhá kombinace způsobuje (ve většině případů) generování nižšího počtu termů za cenu vyšší časové náročnosti. Třetí kombinace se od druhé liší nastavením přepínače –m na hodnotu *on*, což má silný vliv jak na počet termů tak i na čas potřebný k výpočtu. V tomto případě dochází ke generování nejnižšího počtu termů, přičemž výpočetní čas se v mnoha případech značně. Užití tohoto nastavení se hodí, když požadujeme komplement s co nejnižším počtem termů - cenou je však vyšší výpočetní čas.

6 Závěr

Hlavní cíl práce byl splněn. Byly naimplementovány algoritmy pro výpočet komplementu vícevýstupových logických funkcí. Vznikla aplikace pro příkazovou řádku, jejíž vstupem je logická funkce v soubor formátu PLA a výstupem je její komplement v tomtéž formátu.

Během implementace bylo nutné řešit problémy týkající se zejména rychlosti implementovaných algoritmů. Jedním z nich byla volba vhodné heuristické funkce. Tento problém byl vyřešen na základě výsledků měření na nabízejících se možných heuristikách. Dalším problémem pak byla složitost algoritmu pro slučování dvou mezivýsledků a jejich prohledávání za účelem minimalizace. Tento problém zůstal otevřen s tím, že algoritmus byl naimplementován s kvadratickou složitostí. Vyřešení tohoto problému by vedlo ke zrychlení celé aplikace.

Vzniklá aplikace byla porovnána s minimalizačním nástrojem Espresso a dále s nástrojem založeným na rozhodovacích diagramech. Dle dosažených výsledků lze prohlásit, že komplementy mojí aplikace a Espressa jsou logicky ekvivalentní. Z hlediska časové náročnosti dává implementovaná aplikace výsledky horší než Espresso, a to zejména pro velká vstupní data. Při porovnání minimálnosti generovaných komplementů dosahuje moje aplikace řádově stejných výsledků jako Espresso, přičemž bylo zjištěno, že ani jeden z nástrojů minimální vyjádření komplementu negeneruje. Během měření byly nalezeny i funkce, pro které dává moje aplikace komplement minimálnější než nástroj Espresso.

Závěrem lze tedy říci, že moje aplikace podává obecně horší výsledky než Espresso, přestože jsou založeny na stejných principech. Odůvodněním je pravděpodobně fakt, že Espresso navíc užívá nějaké postupy, které nebyly v dokumentaci zveřejněny.

Literatura

[1] R. K. Baryton, A. L. Sangiovanni-Vincentelli, C. T. McMullen, G. D. Hachtel: Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publisher, 1984

[2] Z. Blažek, H. Kubátová: Logické systémy. Vydavatelství ČVUT, 1996

[3] J. Liberty: Naučte se C++ za 21 dní. Computer Press, 2002

[4] N. M. Josuttis: C++ Standardní knihovna a STL. Computer Press, 2005

[5] P. Wróblewski: Algoritmy, datové struktury a programovací techniky. Computer Press, 2004

[6] K336 Info – pokyny pro psaní bakalářských prací.
https://info336.felk.cvut.cz/, stav z 1. 4. 2010

[7] Espresso – minimalizační nástroj
http://vlsi.felk.cvut.cz/Download/espresso_dos.zip

[8] S. Yang, "Logic Synthesis and Optimization Benchmarks, Version 3.0," Tech. Report, Microelectronics Center of North Carolina, 1991.

[9] Booleova algebra
http://cs.wikipedia.org/wiki/Booleova_algebra, stav z 1. 4. 2010

[10] J. Bílek: Minimalizace neúplně určených logických funkcí pomocí modifikovaných binárních rozhodovacích diagramů, Diplomová práce na FEL ČVUT v Praze, 2007.

[11] M. Miklánek: Úprava minimalizačního nástroje ESPRESSO, Bakalářská práce na FEL ČVUT v Praze, 2008

[12] Espresso – minimalizační nástroj vyvinutý jako bakalářská práce M. Miklánka [12]

Popis formátu PLA

Vstupem implementovaného nástroje je vícevýstupová logická funkce uložená v souboru ve formátu PLA. Soubor tohoto formátu se skládá ze záhlaví, ve kterém jsou specifikovány základní informace o logické funkci (počet vstupních proměnných, názvy vstupních proměnných, počet výstupních funkcí a jejich názvy, typ PLA, …), a dále vstupní a výstupní matice.

Záhlaví souboru PLA může obsahovat klíčová slova s následujícím významem:

.i [N]	specifikuje počet vstupních proměnných
.o [M]	specifikuje počet výstupních funkcí
.ilb [s_1] … [s_N]	určuje názvy vstupních proměnných
.ob [s_1] … [s_M]	určuje názvy výstupních funkcí
.p [K]	určuje počet termů v souboru
.type [typ]	určuje typ PLA.

Po specifikaci zmíněných parametrů následuje vstupní a výstupní matice. Vstupní matice velikosti NxK je tvořena K součinovými termy, z nichž každý má N vstupních proměnných. Každý z K řádků výstupní matice velikosti MxK pak náleží právě jednomu termu a hodnota v M-tém sloupci této výstupní matice určuje, zda je daný součinový term implikantem M-té výstupní funkce.

Každý řádek vstupní matice představuje jedno ohodnocení vstupních proměnných. Hodnota „0“ na i-té pozici j-tého řádku vstupní matice určuje, že se i-tá proměnná v dané termu vyskytuje znegovaná, zatímco hodnota „1“ říká, že daná proměnná znegovaná není. Je-li na této pozici znak „-“, pak se daná proměnná v příslušném součinovém termu nevyskytuje, neboli term je implikantem nezávisle na hodnotě této proměnné. Následuje ukázka souboru PLA:

```
.i 4
.o 5
.type f
-1-1    10000
100-    11000
10-0    10100
0-10    01000
10-1    01000
-001    01000
--11    00100
-01-    00100
10--    00010
-1-0    00010
11--    00001
-011    00001
.e
```

Obrázek 14:Ukázka souboru PLA

V této ukázce je vícevýstupová logická funkce ve formátu PLA typu f, která má čtyři vstupní proměnné a pět výstupních funkcí. Jména proměnných ani výstupních funkcí nejsou specifikována. Z výstupní matice je patrné, že první výstupní funkce je implikována prvními třemi termy. Pro tuto výstupní funkci tedy lze napsat následující algebraický předpis:

$$f_1 = bd + a\bar{b}\bar{c} + a\bar{b}\bar{d}$$

Typ souboru PLA určuje jakými množinami součinových termů je funkce popsána (On-Set, Off-Set, DC-Set) a tím i reprezentaci znakových hodnot vstupní a výstupní matice logickými hodnotami.

Přípustné jsou typy f a r, dále jejich kombinace s typem d, který nemůže být samostatný (tedy fd, fr, dr a fdr). Dále jsou popsány jednotlivé typy a způsob výpočtu komplementu pro každý typ.

V PLA typu „f“ jsou všechny výstupní funkce popsány výhradně On-Setem. Znaková hodnota „1“ u i-té výstupní funkce symbolizuje, že daný součinový term patří do On-setu i-té výstupní funkce a je tedy jejím implikantem. Jiné znakové hodnoty ve výstupní matici nemají žádnou vypovídací hodnotu. Při výpočtu Off-Setu je DC-Set považován za prázdnou množinu a je vypočítán komplement On-Setu vzhledem k univerzu.

V PLA typu „r“ jsou všechny výstupní funkce popsány jen množinou Off-Setových termů. Znaková hodnota „0“ u i-té výstupní funkce symbolizuje, že daný součinový term patří do Off-Setu i-té výstupní funkce a tudíž implikuje její logickou hodnotu „0“.

V PLA typu „fd“ jsou všechny výstupní funkce popsány pomocí dvou množin: On-Setu a DC-Setu. Ve výstupní matici mají vypovídací hodnotu jen znaky „1“ (pro On-Set) a „~“ (pro DC-Set). Off-Set se určí jako komplement sjednocení On-Setu a DC-Setu k univerzu. Tento typ je výchozím typem (pro případ, že v souboru PLA není typ specifikován).

V PLA typu „dr“ je funkce popsána DC-Setem a Off-Setem. Ve výstupní matici mají vypovídací hodnotu jen znaky „0“ (pro Off-Set) a „~“ (pro DC-Set).

V PLA typu „fr“ je logická funkce popsána pomoci On-Setu a Off-Setu. Ve výstupní matici mají vypovídací hodnotu jen znaky „0“ (pro Off-Set) a „1“ (pro On-Set).

V PLA typu „fdr“ jsou všechny tři množiny, kterými je možné popsat logickou funkci. Ve výstupní matici mají vypovídací hodnotu znaky „1“ (pro On-Set), „0“ (pro Off-Set) a „~“ (pro DC-Set).

Uživatelský manuál

Implementovaný nástroj je určen pro běh v příkazové řádce a jeho ovládání probíhá pomocí přepínačů. Následuje příklad spuštění aplikace a popis jednotlivých přepínačů.

Complement.exe –n *file* [-o file] [-d so|sc] [-m on|off] [-p on|off] [-c on|off] [-t *time*] [-v]

Přepínače:

-n ..... určuje vstupní soubor formátu PLA.
Přepínač je povinný.

-d ..... určuje manipulaci s DC hodnotami při výpočtu komplementu.
Přepínač není povinný.
Hodnoty:
- so DC-set je přidán k onsetu vstupní funkce
- sc DC-set je přidán ke komplementu (výchozí hodnota)

-o ..... určuje výstupní soubor formátu PLA
Přepínač není povinný.

-m ..... určuje, zda se po vypočtení komplementu mají sloučit termy, které jsou sdíleny mezi jednotlivými výstupními funkcemi
Přepínač není povinný.
Hodnoty:
- on zapne funkci (výchozí hodnota)
- off vypne funkci

Tato funkce je časově náročná a výpočet tedy bude trvat déle!

-p ..... Zapíná a vypíná tisk výsledného komplementu na příkazovou řádku.
Přepínač není povinný.
Hodnoty:
- on zapne funkci (výchozí hodnota)
- off vypne funkci

-c ..... zapíná a vypíná vnitřní hledání duplicitních termů.
Přepínač není povinný.
Hodnoty:
- on zapne funkci (default)
- off vypne funkci

Tato funkce je časově náročná a výpočet tedy bude trvat déle!

-t ..... nastavuje časový limit pro výpočet – v sekundách
Přepínač není povinný.
Hodnota musí být větši než nula.
Nulová hodnota znamená, že limit není stanoven.

-v ..... zapne výpis informací o průběhu výpočtu.

Obsah CD

Adresář, podadresář	Obsah
/	kořenový adresář – obsahuje složky **text** a **komplement**
/text/	Text práce
/text/doc	Text práce ve zdrojové podobě – MS Word
/text/pdf	Text práce ve formátu PDF
komplement	implementovaná aplikace a zdrojové kódy
komplement/exe	zkompilovaná aplikace
komplement/source	zdrojové kódy
komplement/tests	soubory užité pro testování a měření
komplement/manual	Manuál ve formátu PDF

Tabulka 10: Obsah přiloženého CD