36BP

F. Trávnický

7. června 2005

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 dne 2. června 2005 Trávnický Filip

Abstrakt Úkolem této práce je vytvořit programový balík pro manipulaci s vícehodnotovou logikou. Za používanou reprezentaci vícehodnotových funkcí byla zvolena reprezentace progamovatelnými logickými poli (PLA). Konkrétním úkolem bylo implementovat některé operace nad termy obecně vícehodnotového PLA a těmi jsou průnik termů, sjendocení termů, nadkrychle termů a rozdíl jednoho termu od průniku tohoto termu s jiným termem. Dále jsem měl implementovat tzv. Co-singleton transformaci termů, což je převod vícehodnotových výrazů na binární obraz ke zpracování v binárních metodách optimalizace vícehodnotové logiky, a k tomu příslušnou inverzní Co-singleton transformaci. Dalším úkolem bylo implementovat operace nad dvojící funkcí, tedy nad dvěmi PLA a to konkrétně operace AND a OR. Vstupem pro zpracování operací je textový soubor s popisem PLA inspirovaný notací ESPRESSO a výstupem je rovněž takovýto soubor s výsledným PLA.

Abstract The task of this project is to make a package for manipulating with multiple-valued logic. The chosen representation for multiple-valued functions is representation by programmable logic arrays (PLA). Concrete task was to implement some operations over terms of generally multiple-valued PLAs like intersection of terms, union of terms, subercube of terms and a difference of one term with intersection of this term with another one. Then I was wanted to implement so-called Co-singleton transformation which is converting of multiple-valued expressions to binary images for manipulation using binary methods of optimization of multiple-valued logic, and also to implement corresponding inverse transformation. The next task was to implement operations over two logic functions, in this - two PLAs, concretly the operations AND and OR. The input for manipulating with is a text file with description of PLA inspirated by ESPRESSO notation, and the output is also such file with result PLA.

Obsah

1 Přístupy vícehodnotové logiky a její využití **6**

1.1 Historický pohled na vývoj a využití vícehodnotové logiky . . 7

1.2 Zobecnění boolovy algebry pro vícehodnotové proměnné . . . 8

2 Vícehodnotová logika: definice a notace **9**

3 Reprezentace funkcí s vícehodnotovými proměnnými **12**

3.1 Vícehodnotová síť 12

3.2 Vícehodnotové rozhodovací diagramy (MDD) 13

3.3 Reprezentace programovatelnými logickými poli (PLA) 14

3.3.1 Klíčová slova 14

3.3.2 Příklady 17

4 Optimalizace vícehodnotové logiky **21**

4.1 Co-singleton transformace 22

4.1.1 dopředná transformace 22

4.1.2 zpětná transformace 22

4.1.3 Příklad 22

5 Popis řešení zadaného úkolu **23**

5.1 Datové struktury 23

5.2 Popis algoritmů 25

5.2.1 Nadkrychle termů (supercube) 25

5.2.2 Průnik dvou termů 26

5.2.3 Sjednocení dvou termů 27

5.2.4 Rozdíl termu od průniku s jiným termem 27

5.2.5 Co-singleton transformace 28
5.2.6 Algoritmus pro logický součin (AND) dvou funkcí (PLA) 29
5.2.7 Algoritmus pro logický součet (OR) dvou funkcí (PLA) 29
5.3 Popis funkcí programového balíku 30

6 Závěr **36**

Seznam tabulek

2.1 Vícehodnotová funkce dvou proměnných 9
2.2 Vícehodnotová funkce dvou proměnných - poziční notace . . . 11

3.1 Příklad 3-hodnotového MDD 13

Kapitola 1

Přístupy vícehodnotové logiky a její využití

Vícehodnotová logika je zobecněním klasické booleovské logiky. Jedním z důvodů pro zobecňování je snaha po hlubším pochopení speciálních problémů. Při zobecnění nějaké struktury zjišťujeme, že některé její vlastnosti už dále obecně neplatí a my s nimi nemůžeme dále počítat, a naproti tomu některé vlastnosti se i při takovém zobecnění zachovávají.

Pokus o zobecnění pomáhá oddělit nedůležité vlastnosti od vlastností esenciálních pro daný problém.

Logický návrh se běžně popisuje s použitím binárních signálů, nicméně pro návrh vyšší úrovně je přirozené uvažovat proměnné se symbolickými hodnotami. Například, je jednodušší koncipovat procesor semaforového světla se signálem "světlo", který může nabývat tří hodnot: "červená", "žlutá" a "zelená", nežli příznaky světlo_0 = 1, světlo_1 = 0 pro stav vyjadřující, že svítí červené světlo. Proces konverze vícehodnotových proměnných na binární signály nazýváme kódováním.

Dalším významným důvodem pro zabývání se vícehodnotovou logikou je fakt, že některé booleovské problémy lze s jejím použitím řešit efektivněji. Například známý přístup k reprezentování několikanásobného výstupu booleovské funkce je považování její výstupní části za jedinou vícehodnotovou proměnnou a převod takové funkce na funkci s jedním výstupem. Toto pojetí se používá v ESPRESSO-MV [1] a MVSIS [2]. Jiné aplikace vícehodnotové logiky zahrnují návrh programovatelných logických polí (PLA) s vstupními dekodéry [3], optimalizace konečných automatů [4], testování a verifikace [5] [6].

Vícehodnotová logika také poskytuje teoretický základ pro návrh elektronických obvodů s více než dvěmi logickými úrovněmi, jako například tří- a čtyřhodnotová PLA a paměti. Vícehodnotové obvody mají několik teore-

tických výhod oproti standardním binárním obvodům. Například propojení čipu s deskou se oproti čipu s binárními signály zredukuje, mohou-li jeho signaly nabývat čtyř a více úrovní [7]. Při návrhu pamětí, možnost uložení dvou místo jednoho bitu informace na každou paměťovou buňku zdvojnásobuje hustotu paměti na její daný fyzický rozměr [8]. Pro aplikace používající aritmetické obvody se ukazuje často jako výhodné použít alternativy k binárním číslicovým systémům. Například reziduální a redundantní číslicové systémy mohou snížit nebo vyloučit postupné přenosy (carry), které se objevují při běžném binárním sčítání nebo odčítání, které je vyhodnocováno vysokorychlostními aritmetickými operacemi. Pro tyto číslicové systémy je přirozenou implementací použití vícehodnotových obvodů. [9] [10]. Nicméně praktičnost těchto potenciálních možností silně závisí na možnosti realizace zmíněných obvodů, jež musí být kompatibilní a konkurenceschopné se současnými standardními technologiemi.

Přístupy syntézy vícehodnotové logiky se stávají nyní důležitými i v nejrůznějších softwarových aplikacích. Používá se také např. v takzvané control dominated softwarové kompilaci, při níž jsou řídící proměnné počítány a testovány právě jako proměnné ve vícehodnotové logice. Zde jsou zkoumány logické vztahy mezi proměnnými při restrukturalizaci řídícího toku programu. Takovéto možnosti optimalizace obyčejně nejsou tradičními softwarovými kompilátory vůbec uvažovány.

1.1 Historický pohled na vývoj a využití vícehodnotové logiky

Vývoj vícehodnotové logiky začal prací Łukasiewicze a Posta (1920). První algebra korespondující s Postovou logikou byla poprvé formulována Rosenbloomem (1942) a potom dále vyvíjena Epsteinem. Objasnění nutných a postačujících podmínek pro úplnost množiny vícehodnotových funkcí patří Rosenbergovi (1965).

Vícehodnotové funkce byly původně studovány pro podporu návrhu m-hodnotových logických obvodů, které používají m diskrétních signálů, $m >= 2$. První pokusy o vytvoření vícehodnotových integrovaných obvodů kompatibilních s technologiemi stávajících integrovaných obvodů se datují od roku 1970, kdy se začalo pracovat na návrhu 3-hodnotových obvodů. Aplikování vícehodnotové logiky se ukázalo mít pozitivní vliv na uspořádaní obvodů v jednotlivých čipech, na operační rychlost a také spotřebu energie. Nyní jsou vícehodnotové obvody hojně užívany ve čtyřhodnotových flash a DRAM pamětech, jež jsou dnes součástí běžných komerčně dostupných integrovaných obvodů.

Aplikacemi vícehodnotové logiky v jiných vědních oborech jsou například návrh neuronových sítí, prahová logika, molekulární a optické výpočty.

1.2 Zobecnění boolovy algebry pro vícehodnotové proměnné

Boolova algebra je množina, na které jsou definované dvě operace. Boolova algebra je často spojována s binárními funkcemi binárních proměnných. Každá binární funkce je reprezentována určitou množinou hodnot - například svým On-setem (viz. definice 2.3). Uplatníme-li na dvě funkce operaci AND, počítáme tím vlastně průnik jejich On-setů. Podobně uplatníme-li takto operaci OR, počítáme tím sjednocení. Je známo, že každá boolova algebra je isomorfní k boolově algebře množin, kde průnik a sjednocení jsou tyto dvě operace. Nic však nebylo řečeno o rozměru doménového prostoru. Je možné pomocí vícehodnotových proměnných popsat bod nějakého prostoru. Například mějme dvě proměnné, x s pěti hodnotami, a y se třemi hodnotami. Potom máme 15 bodů v doménovém prostoru. Bod v prostoru (minterm) je určen, pokud známe okamžité hodnoty všech proměnných z jejich domén, např. $(x = 3, y = 1)$. Funkce je tedy nějaká podmnožina takových mintermů. Matematika boolových algeber lze tedy přímo aplikovat na binární funkce vícehodnotových proměnných. Pokud považujeme každou výstupní hodnotu jako samostatnou funkci, můžeme takto i popisovat funkci jejíž rozměr je také vícehodnotový. Například množina bodů, pro které signál "světlo" má hodnotu "červená" je On-setem jedné funkce, body pro které je světlo "žluté" je pak On-setem jiné funkce atd.

Kapitola 2

Vícehodnotová logika: definice a notace

Definice 2.1 *Vícehodnotová proměnná X_i může nabývat hodnoty z $P_i = \{\alpha_0, \alpha_1, \ldots, \alpha_{|P_i|-1}\}$.*

Protože každá symbolická hodnota může být asociována s jediným celým číslem i, budeme dále uvažovat pouze proměnné s celočíselnými hodnotami, tedy $P_i = \{0, 1, \ldots, |P_i| - 1\}$.

X_1	X_2	F
0	0	1
0	1	2
1	0	1
1	1	1
2	0	0
2	1	2

Tabulka 2.1: Vícehodnotová funkce dvou proměnných

Definice 2.2 *Vrchol je bod v prostoru $P_1 \times P_2 \times \ldots \times P_n$, kde n je počet vstupních proměnných.*

Definice 2.3 *Vícehodnotová funkce F je funkce která mapuje vrcholy v $P_1 \times P_2 \times \ldots \times P_n$ do P_F, formálně $F : P_1 \times P_2 \times \ldots \times P_n \mapsto P_F$, kde n je počet vstupních proměnných.*

Příkladem vícehodnotové funkce je funkce v tabulce 2.1. Zde máme $P_1 = \{0, 1, 2\}, P_2 = \{0, 1\}$ a $P_F = \{0, 1, 2\}$. Jestliže $P_F = \{0, 1, *\}$, potom F je vícehodnotová funkce s dvouhodnotovým (binárním) výstupem.

Jestliže vrchol (minterm) je mapován na hodnotu 1, potom říkáme, že patří do *On-set*, je-li mapován na hodnotu 0, patří do *Off-set*, a je-li mapován na *, potom patří do *Don't-care set*, což znamená, že může nabývat obou (obecně všech) hodnot daného oboru hodnot. Tedy principy z binárních funkcí jako implikanty, přímé implikanty, pokrytí a přímé pokrytí mohou být rozšířeny také na vícehodnotové funkce F s binárním výstupem.

Definice 2.4 *Vícehodnotový literál $X_i^{c_i}$ je logická funkce formy:*

$$X_i^{c_i} = (X_i = \gamma_1) + \cdots + (X_i = \gamma_k), \text{ kde } \gamma_j \in c_i \subseteq P_i$$

Definice 2.5 *Krychle $c = c_1 \times c_2 \times \ldots \times c_n$ může být chápána jako součin vícehodnotových literálů (MV-literálů) ve formě $X_1^{c_1} X_2^{c_2} \ldots X_n^{c_n}$.*

Uvědomme si, že pokud $c_i = P_i$, můžeme z výrazu krychle (term) vynechat $X_i^{P_i}$, protože $X_i^{P_i} = 1$. Jestliže proměnná X_i je dvouhodnotová, pak literál X_i můžeme zapsat v nové notaci jako $X_i^{\{1\}}$. Podobně literál $\overline{X_i}$ můžeme zapsat jako $X_i^{\{0\}}$. Jestliže proměnná nabývá obou svých hodnot (psáno ovykle jako "-"), zapíšeme jako $X_i^{\{0,1\}}$.

Definice 2.6 *Implikant je krychle c taková, že pro všechny vrcholy $v \in c$, platí $F(v) \neq 0$.*

Definice 2.7 *Přímý implikant je implikant c takový, že neexistuje implikant d takový, že $d \supset c$.*

Definice 2.8 *Pokrytí funkce F je množina implikantů, jejichž sjednocení obsahuje každý bod z* On-set *funkce F a žádný z* Off-set.

Definice 2.9 *Přímé pokrytí je pokrytí, jehož každý prvek je přímý implikant.*

Vícehodnotová funkce (MV-funkce) z tabulky 1 může být zapsána ve formě součtu krychlí pro každou ze svých hodnot. Jedno takové pokrytí pro F je:

$$\begin{aligned} F^{\{0\}} &= X_1^{\{2\}} X_2^{\{0\}} \\ F^{\{1\}} &= X_1^{\{0,1\}} X_2^{\{0\}} + X_1^{\{1\}} X_2^{\{0,1\}} \\ F^{\{2\}} &= X_1^{\{0,2\}} X_2^{\{1\}} \end{aligned}$$

Výhodnou reprezentací literálů a krychlí je poziční notace:

Definice 2.10 *Poziční notace: literál $X_i^{c_i}$ je popsán hodnotami na daných pozicích (sloupcích) $v_0, v_1, \ldots, v_{|P_i|-1}$ takových, že:*

$$v_j = \begin{cases} 1 & \text{jestliže } j \in c_i \subseteq P_i \\ 0 & \text{v opačném případě} \end{cases} \tag{2.1}$$

Například MV-funkce z tabulky 2.1 se v poziční notaci zapíše takto:

X_1	X_2	F
001	10	0
110	10	1
010	11	1
101	01	2

Tabulka 2.2: Vícehodnotová funkce dvou proměnných - poziční notace

Kapitola 3

Reprezentace funkcí s vícehodnotovými proměnnými

Vedle poziční notace existují další způsoby vyjádření mv-fukcí. Těmi jsou například vícehodnotové sítě MDD diagramy [1] a programovatelná logická pole.

3.1 Vícehodnotová síť

je reprezentace víceúrovňovým grafem, podobně jako boolovská síť s tím, že každý uzel je obecně vícehodnotová funkce. Ve vyšších úrovních abstrakce, při níž mohou proměnné nabývat symbolických hodnot, činí tato reprezentace návrh více intuitivním. Návrhář může nejprve manipulovat s vícehodnotovou sítí, optimalizovat ji, a potom provést odpovídající zakódování pro výslednou boolovskou síť. Příkladem nástroje, jehož vstup přijímá vícehodnotovou síť je Verification Interacting with Syntesis (VIS) [11], což je nástroj pro verifikaci. Obsahuje překladač Verilogu, který generuje tzv. blif-mv popis vícehodnotové sítě. Tento blif-mv popis zpracovává např. MV-SIS - nástroj pro manipulaci a optimalizaci vícehodnotové logiky. Nevýhodou síťové reprezentace je fakt, že není kanonická, to znamená, že daná funkce může mít více různých reprezentujících funkcí. Testování ekkvivalence funkcí může být poměrně složité. Tento problém je eliminován v jiné reprezentaci popsané níže.

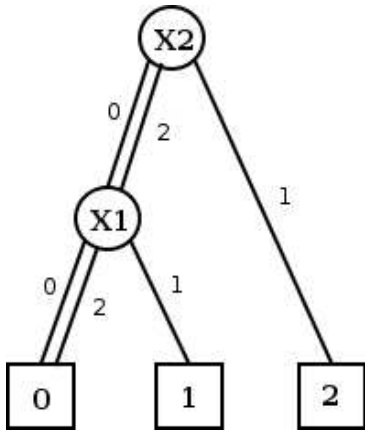
[1] multiple decision diagrams

3.2 Vícehodnotové rozhodovací diagramy (MDD)

jsou přímým zobecněním binárních rozhodovacích diagramů (BDD). Funkce jsou reprezentovány orientovanými acyklickými grafy neterminálními uzly v, označenými indexem proměnné index i, $i \in \{1, 2, \ldots, n\}$. Každý uzel v má m z něho vystupujících hran směřovaných k uzlům potomků, notovaných jako $child_j(v), j \in M$. Každý terminální uzel čili list má jako atribut hodnotu $value(v) \in M$. Pro získání redukovaného uspořádaného MDD platí podobný způsob řazení proměnných a redukce grafu jako u BDD. Příklad 3-hodnotového MDD, implementujícího funkci z tabulky 3.1 je ukázán na obrázku 3.1.

X_2/X_1	0	1	2
0	0	1	0
1	2	2	2
2	0	1	0

Tabulka 3.1: Příklad 3-hodnotového MDD



Obrázek 3.1: Příklad 3-hodnotového MDD - diagram

Jestliže v je neterminální uzel s indexem $index(v) = i, i \in \{1, 2, \ldots, n\}$, potom funkce reprezentovaná uzlem v, f_v je dána:

$$f_v(x_1, \ldots, x_n) = \sum_{j \in M} x_i^j \cdot f_{child_j(v)}(x_1, \ldots, x_n)$$

Podobně jako v BDD, hodnota funkce pro dané nastavení proměnných je determinována cestou v grafu od kořene k listům. V každém uzlu se rozhodujeme dle hodnoty proměnné spojené s daným uzlem. Pokračujeme vždy po hraně odpovídající hodnotě dané proměnné. Na konci cesty, v listu je

odpovídající výsledná hodnota funkce. Například funkce reprezentovaná diagramem z obrázku 1 má hodnotu 1 pro vstupy $x_2 = 2, x_1 = 1$.

MDD přejímají silné výhody BDD: existuje forma pro BDD i MDD (tzv. ROBDD (ROMDD) - Reduced Ordered BDD (MDD)), což jsou maximálně zredukované BDD (MDD) s daným pořadím proměnných. Výhodou takové reprezentace je kanonicita. Dalšími výhodami jsou jednoduchost algoritmů pro manipulaci a kompaktnost pro mnoho funkcí. Nevýhodou MDD je podobně jako u BDD problém řazení proměnných. Například MDD z obrázku 3.1 by měl dvakrát tolik neterminálních uzlů, kdyby proměnné byly řazeny obráceně $< x_1, x_2 >$.

3.3 Reprezentace programovatelnými logickými poli (PLA)

Nyní uvedu notaci programovatelného logického pole dle ESPRESSO, která se používá pro účely softwarové manipulace s poli PLA při jejich návrhu, optimalizaci atd. Tuto reprezentaci používá i programový balík, jenž je součástí této bakalářské práce.

3.3.1 Klíčová slova

`[d]` značí desítkové číslo a `[s]` značí řetězec. Minimální požadovaná klíčová slova jsou `.i` a `.o` pro binární funkce nebo `.mv` pro vícehodnotové funkce.

`.i [d]`
specifikuje počet vstupních proměnných

`.o [d]`
specifikuje počet výstupních funkcí

`.mv [num_var] [num_binary_var] [d1] .... [dn]`
specifikuje počet proměnných (`num_var`), počet binárních proměnných (`num_binary_var`), a rozměr každé vícehodnotové proměnné (`d1` až `dn`)

`.ilb [s1] [s2] .... [sn]`
určuje názvy binárních proměnných. Toto klíčové slovo musí následovat až po `.i` a `.o` (nebo po `.mv`). Musí být tolik názvů, kolik je vstupních proměnných

`.ob [s1] [s2] .... [sn]`
určuje názvy výstupních funkcí. Toto klíčové slovo musí následovat až po `.i` a `.o` (nebo po `.mv`). Musí být tolik názvů, kolik je výstupních proměnných.

`.label var=[d] [s1] [s2] ....`
specifikuje názvy hodnot vícehodnotové proměnné. Toto klíčové slovo musí

následovat po `.mv`. Musí být tolik názvů, kolik je hodnot dané proměnné. Proměnné jsou číslované od 0.

`.type [s]`
nastaví logickou interpretaci charakteristické matice (viz níže). Toto klíčové slovo musí přijít před prvním součinovým termem. `[s]` je jedno z `f`, `r`, `fd`, `fr`, `dr`, nebo `fdr`.

`.phase [s] [s]`
je řetězec tolika 0 nebo 1, kolik je výstupních funkcí. Specifikuje, jaká polarita pro každý výstup má být použita pro minimalizaci (1 specifikuje, že má být minimalizován *On-set* dané výstupní funkce, 0 naproti tomu *Off-set*).

`.pair [d]`
specifikuje počet párů proměnných, které budou párovány použitím dvoubitových dekodérů. Zbytek řádku obsahuje páry čísel, které specifikují které binární proměnné z PLA mají být spárovány. Binární proměnné jsou číslovány od nuly.

`.symbolic [s0] [s1] ... [sn] ; [t0] [t1] ... [tm]`
specifikuje že binární proměnné pojmenované `[s0]` až `[sn]` jsou považovány jako jediná vícehodnotová proměnná. Proměnná `[s0]` je považována za nejvýznamnější bit a proměnná `[sn]` za nejméně významný bit. Toto vytvoří proměnnou s 2^n hodnotami. Klíčová slova `[t0]` až `[tm]` jsou názvy pro každou kombinaci `[s0]` až `[sn]`. (`[t0]` koresponduje s hodnotou $00\ldots00$, `[t1]` je hodnota $00\ldots01$, atd.). Binární proměnné mohou být identifikovány číslem sloupce nebo názvem proměnné pokud je použito `.ilb`. Binární proměnné jsou odstraněny z funkce jakmile je vytvořena vícehodnotová proměnná.

`.symbolic-output [s0] [s1] ... [sn] ; [t0] [t1] ... [tm]`
specifikuje, že výstupní funkce `[s0]` až `[sn]` mají být považovány za jediný symbolický výstup. Tím se vytvoří 2^n krát víc výstupních proměnných korespondujících s možnými hodnotami výstupů. Výstupy mohou být identifikovány číslem (číslujeme od nuly), nebo jménem proměnné, pokud je použito `.ob`. Výstupy jsou z funkce odstraněny jakmile je vytvořena nová množina výstupů.

`.kiss`
nastaví pro minimalizaci ve stylu kiss

`.p [d]`
specifikuje počet součinových termů. Součinové termy (jeden na každou řádku) následují okamžitě za tímto klíčovým slovem. Tato řádka je ignorována a `.e`, `end` indikuje konec vstupního popisu

`.e`
značí konec popisu PLA

PLA je reprezentováno jistým počtem řádků, každý řádek reprezentuje jeden součinový term. Celá funkce je chápáná jako součet těchto součinových termů. Každý řádek obsahuje vektor hodnot vstupních proměnných, jež mohou být binární (nuly, které značí, že daný literál se vyskutuje v termu ve formě negace, jedničky, která značí, že daný literál se v tomto termu vyskytuje nenegovaný, a symbol "-", který značí, že daný literál se v tomto termu nevyskytuje), vícehodnotové nebo symbolické, a vektor výstupních hodnot. Vícehodnotové proměnné obvykle následují až za binárními. Každá z mv-proměnných je reprezentována buďto vektorem nul a jedniček čili poziční notací (viz. výše) nebo řetězcem, který určuje hodnotu dané proměnné. Jednotlivé vektory vícehodnotových proměnných mohou být odděleny znakem | . Používáme-li vícehodnotové proměnné, poslední mv-proměnná je chápána jako výstup. Pokud je rozměr mv-proměnné deklarován jako menší než nula, značí to, že v daném PLA má daná proměnná své hodnoty v termech udáné ve formě řetězce, tyto symbolické názvy se pak načtou přímo z termů na místo pro informace o názvech hodnot dané proměnné. Absolutní honota tohoto čísla potom specifikuje maximální počet unikátních symbolických názvů hodnot proměnné, který předpokládáme. Při práci s konečnými automaty máme v PLA dvě symbolické proměnné "současný stav automatu" a "následující stav automatu", toto se používá při kiss-style kódování. V takovém PLA máme jako poslední proměnnou výstup, předposlední musí být "následující stav automatu" a třetí od konce musí být "současný stav automatu. Co se týče výstupů, v jejich vektorech se vyskutují symboly 0,1,- a . Jejich význam však není univerzální a závisí na tom, jak je funkce definována pomocí klíčového slova `.type`.

Máme-li typ `f`, pak platí pro každý výstup: 1 značí, že součinový term patří do On-setu, a 0 nebo -, znamená, že tento term nemá žádný význam pro danou funkci. Tento typ koresponduje s aktuálním PLA, kde je zatím implementován pouze On-set.

Typ `fd` (implicitní), pro každý výstup: 1 - On-set, 0 - žádný význam, a "-" říká, že tento term patří do Dc-setu.

Typ `fr`, pro každý výstup: 1 - On-set, 0 - Off-set, "-" - žádný význam.

Typ `fdr`, pro každý výstup: 1 - On-set, 0 - Off-set, "-" - Dc-set a $\sim$ - žádný význam.

symbol $\sim$ značí obecně "žádný význam" bez ohledu na `.type`.

Pokud je v PLA uveden znak #, znamená to, že to co následuje za ním až do konce řádku je komentář.

3.3.2 Příklady

Příklad 1

Dvoubitová sčítačka, která má dva dvoubitové operandy a výsledek je tříbitový. Možný popis kompletně v mintermech je tento:

```
# 2-bit by 2-bit binary adder (with no carry input)
.i 4
.o 3
0000  000
0001  001
0010  010
0011  011
0100  001
0101  010
0110  011
0111  100
1000  010
1001  011
1010  100
1011  101
1100  011
1101  100
1110  101
1111  110
```

je také možné specifikovat některé speciální volby jako :

```
# 2-bit by 2-bit binary adder (with no carry input)
.i 4
.o 3
.ilb a1 a0 b1 b0
.ob s2 s1 s0
.pair 2 (a1 b1) (a0 b0)
.phase 011
0000  000
0001  001
0010  010
    .
    .
    .
1111  110
.e
```

Volba `.pair` indikuje, že první binární proměnná bude spárována se třetí binární proměnnou, a že druhá bude spárována se čtvrtou. Tyto páry budou mapovány na ekvivalentní mv-proměnné.

Volba `.phase` indikuje, že pro druhý a třetí výstup bude použita positivní fáze a pro první výstup negativní fáze (viz. interpretaci `.phase` výše).

Příklad 2

Tento příklad ukazuje popis vícehodnotové funkce s 5 binárními proměnnými a se třemi mv-proměnnými (celkem 8 proměnných), přičemž mv-proměnné mají rozměry 4, 27 a 10 (poslední proměnná je výstup a kóduje informaci o On-setu, Off-setu a DC-setu).

```
.mv 8 5 4 27 10
.ilb in1 in2 in3 in4 in5
.label var=5 part1 part2 part3 part4
.label var=6 a b c d e f g h i j k l m n
             o p q r s t u v w x y z a1
.label var=7 out1 out2 out3 out4 out5 out6
             out7 out8 out9 out10
0-010|1000|1000000000000000000000000000|0010000000
10-10|1000|0100000000000000000000000000|1000000000
0-111|1000|0010000000000000000000000000|0001000000
0-10-|1000|0001000000000000000000000000|0001000000
00000|1000|0000100000000000000000000000|1000000000
00010|1000|0000010000000000000000000000|0010000000
01001|1000|0000001000000000000000000000|0000000010
0101-|1000|0000000100000000000000000000|0000000000
0-0-0|1000|0000000010000000000000000000|1000000000
10000|1000|0000000001000000000000000000|0000000000
11100|1000|0000000000100000000000000000|0010000000
10-10|1000|0000000000010000000000000000|0000000000
11111|1000|0000000000001000000000000000|0010000000
                        .
                        .
                        .
11111|0001|0000000000000000000000000001|0000000000
```

Příklad 3

Tento příklad ukazuje popis mv-funkce nastavené pro kiss-style minimalizaci. Máme zde 5 binárních proměnných, dvě symbolické proměnné (současný a následný stav konečného automatu) a výstup (celkem 8 proměnných).

```
.mv 8 5 -10 -10 6
.ilb io1 io0 init swr mack
.ob wait minit mrd sack mwr dli
.type fr
.kiss
--1--     -        init0     110000
--1--     init0    init0     110000
--0--     init0    init1     110000
--00-     init1    init1     110000
--01-     init1    init2     110001
--0--     init2    init4     110100
--01-     init4    init4     110100
--00-     init4    iowait    000000
0000-     iowait   iowait    000000
1000-     iowait   init1     110000
01000     iowait   read0     101000
11000     iowait   write0    100010
01001     iowait   rmack     100000
11001     iowait   wmack     100000
--01-     iowait   init2     110001
--0-0     rmack    rmack     100000
--0-1     rmack    read0     101000
--0-0     wmack    wmack     100000
--0-1     wmack    write0    100010
--0--     read0    read1     101001
--0--     read1    iowait    000000
--0--     write0   iowait    000000
```

Příklad 4

Tento příklad ukazuje použití klíčového slova `.symbolic`.

```
.i 15
.o 4
.ilb SeqActive<0> CacheOp<6> CacheOp<5> CacheOp<4>
     CacheOp<3> CacheOp<2> CacheOp<1> CacheOp<0>
     userKernel<0> Protection<1> Protection<0>
     cacheState<1> cacheState<0> PageDirty<0>
     WriteCycleIn<0>

.ob CacheBusy<0> dataMayBeValid<0> dataIsValid<0>
    WriteCycleOut<0>

.symbolic CacheOp<6> CacheOp<5> CacheOp<4> CacheOp<3>
```

```
          CacheOp<2> CacheOp<1> CacheOp<0> ;
          FET NA PHY_FET PR32 PRE_FET PW32 RA32 RD32
          RD64 RDCACHE RFO32 RFO64 TS32 WR32 WR64 WRCACHE ;

.symbolic Protection<1> Protection<0> ;
          PROT_KRO_UNA PROT_KRW_UNA PROT_KRW_URO PROT_KRW_URW ;

.symbolic cacheState<1> cacheState<0> ;
          CS_Invalid CS_OwnPrivate CS_OwnShared CS_UnOwned ;

.p 22
0000001--010110 0001
0000001-1-00110 0001
00001011-01011- 0100
000010111-0011- 0100
0000--001--01-- 0100
0000-10--0-1--- 0100
0000-10-1--1--- 0100
00000-0--0-1--- 0100
00000-0-1--1--- 0100
0000-10--0--1-- 0100
0000-10-1---1-- 0100
00000-0--0--1-- 0100
00000-0-1---1-- 0100
---1----------- 1000
--1------------ 1000
-1------------- 1000
1-------------- 1000
-------0------- 1000
----1---------- 1000
-----0--------- 1000
------0-------- 1000
--------------1 1110
.e
```

Kapitola 4

Optimalizace vícehodnotové logiky

V této kapitole stručně uvedu různé přístupy optimalizace vícehodnotové logiky a minimalizace mv-funkcí.

Pro minimalizaci mv-funkcí existují různé metody. Jedním z nich je využití zobecněných principů boolovské logiky, aplikované obdobným způsobem jako pro dvouhodnotovou logiky [12]. Jedná se o minimalizaci počtu termů ve vyjádření funkce jako součet součinových termů (Sum Of Products). Využívá se zde zobecněný Shannonův expanzní teorém. Velmi podstatný je vliv kódování vícehodnotových proměnných na efektivitu výpočtu minimalizace a vlastně i vliv na uspokojivý výsledek. Jednou z prvních vyvinutých metod je tzv. one-hot kódování, což je vlastně reprezentace mv-proměnné dle poziční notace uvedené v kapitole 2. Pokud proměnná nabývá dané hodnoty, je nastaven daný odpovídající signál na 1, ostatní jsou 0. Výhodou tohoto kódování je fakt, že mapuje problém do oblasti binární logiky a je tedy možné uplatnit vysoce vyvinuté binární algoritmy. Nevýhodou je vysoký počet signálů ke zpracování a také vznik velkého počtu prvků Don't-care-set. Vývoj efektivního programového balíku s inteligentním kódováním pro přímou manipulaci s vícehodnotovou logikou a její optimalizaci, následovanou binární optimalizací je výzva pro budoucnost celkového vývoje v oblasti logických systémů. Jiným přístupem jsou algebraické a semi-algebraické metody optimalizace [13]. Semi-algebraické metody kombinují čistě algebraické metody s metodami binárními. Algebraické metody pracují s funkcemi jako s polynomy, ignorují boolovské zákony jako např. $x \cdot \overline{x} = 0, x + \overline{x} = 1, x \cdot x = x$. Algebraické metody jsou např. hledání společného podvýrazu nebo společného dělitele, vytýkání společný term před závorku atd. Dalším přístupem jsou čistě binární metody. Zde se převádí výraz s mv-proměnnými na výraz obsahující pouze binární proměnné. Existuje bijektivní zobrazení mezi mv-výrazem a binárním výrazem zprostředkované tzv. Co-singleton trans-

formací [14]. Tato transformace však nesmí být chápána jako kódování vícehodnotové funkce, neboť vzniklé "kódy" pro hodnoty mv-proměnných nejsou disjunktní. Pouze se převede vždy jeden mv-výraz na binární výraz, upraví se binárními metodami a pak je zpětnou transformací převeden opět mv-výraz, čímž dosahujeme zjednodušení původního výrazu. Nyní popíšu samotnou Co-singleton transformaci, protože ji také implementuji v programovém balíku, jenž je součástí této práce.

4.1 Co-singleton transformace

4.1.1 dopředná transformace

vstup: mv-výraz E

výstup: odpovídající binární výraz E'.

Každý literál $X^S \in E$ nahradíme $\prod_{i \in S} x_i$. ($x_i \equiv X^{\{0,\ldots,i-1,i+1,\ldots n\}}$ je co-singleton literál).

4.1.2 zpětná transformace

vstup: binární výraz E'

výstup: mv-výraz E upravený, avšak ekvivalentní s původním výrazem E před vstupem do dopředné co-singleton transformace

Každý součin $\prod_{i \in T} x_i \in E$ nahradíme $X^{\{j|j \notin T\}}$.

4.1.3 Příklad

$$
\begin{aligned}
E &\xrightarrow{\text{dopředná}} a^{\{2,3\}}\, b^{\{0,1\}} + a^{\{0,3\}}\, b^{\{1,2\}} + a^{\{1,2\}}\, b^{\{0,3\}} + a^{\{0,1\}}\, b^{\{2,3\}} \\
E' &\xrightarrow{\text{faktorizace}} a_0a_1b_2b_3 + a_1a_2b_0b_3 + a_0a_3b_1b_2 + a_2a_3b_0b_1 \\
E' &\xrightarrow{\text{zpětná}} (a_1b_3 + a_3b_1)(a_0b_2 + a_2b_0) \\
E &= (a^{\{0,2,3\}}b^{\{0,1,2\}} + a^{\{0,1,2\}}b^{\{0,2,3\}})(a^{\{1,2,3\}}b^{\{0,1,3\}} + a^{\{0,1,3\}}b^{\{1,2,3\}})
\end{aligned}
$$

Kapitola 5

Popis řešení zadaného úkolu

Dané úkoly jsem implementoval v programovacím jazyce C.

5.1 Datové struktury

Vstupem a výstupem programu pro zpracování obecně vícehodnotových funkcí je, jak již bylo uvedeno v úvodu, popis PLA dle notace ESPRESSO v textovém souboru. Vstupní informace je třeba beze zbytku načíst do vnitřní reprezentace PLA. Vnitřní reprezantace datové struktury PLA vypadá takto:

```
typedef struct PLA        {
 int i,o,mvnum,cs,inam,onam,kiss;
 char *type,*phase,*pair,*symbolic,*symb_out;
 char * i_names [I],* o_names [O];
 mv_desc mvinfo [MV];
 term * first_term;
 };
```

Struktura PLA obsahuje spojový seznam součinových termů daného PLA (ukazatel na první term je zde položka `first_term`), celočíselná proměnná `i` udává počet vstupních binárních proměnných, proměnná `o` udává počet výstupních proměnných a proměnná `mvnum` udává počet vícehodnotových proměnných, které nejsou považovány za výstupní. Proměnné `cs`, `inam`, `onam`, `kiss` jsou příznakové proměnné, které nabývají pouze hodnot 0 a 1, charakterizují specifickým způsobem dané PLA. Proměnná `cs`, pokud je nastavena na 1, říká, že dané PLA obsahuje všechny termy v podobě výstupních výrazů Co-singleton transformace. Proměnná `inam`, pokud je nastavena na 1, udává, že dané PLA má specifikovány názvy vstupních binárních proměnných. Analogicky příznak `onam` říká, zda jsou specifikovány názvy výstupních

proměnných. Příznak `kiss`, pokud je nastaven na 1, říká, že PLA je zadáno v kiss notaci. Řetězcové proměnné `type`, `pair`, `symbolic` a `symb_out` obsahují průvodní informace z načteného PLA v podobě řetězců, tak jak jsou napsány v textovém souboru vstupního PLA, aby je ve stejné podobě bylo možné kopírovat na do výstupních PLA. Statické pole řetězců `i_names` obsahuje názvy binárních vstupních proměnných, podobně pole `o_names` obsahuje názvy výstupních fukcí. Pole `mvinfo` obsahuje popis jednotlivých vícehodnotových proměnných v podobě záznamů (typ `mv_desc`).

Datový typ `mv_desc` vypadá takto :

```
typedef struct mv_desc        {
 int size;
 int exp_size;
 int is_symbolic;
 int mvnam;
 char * values [N];
 };
```

Proměnná `size` udává rozměr vícehodnotové proměnné. Pokud byl ve vstupním PLA rozměr nějaké mv-proměnné zadán jako záporné číslo, pak `size` je nastaven na 0 a do proměnné `exp_size` se uloží absolutní hodnota zadaného rozměru. Zároveň se nastaví příznak `is_symbolic` na 1, což říká, že hodnoty dané proměnné v termech jsou očekávány ve formě řetězců. Proměnná `size` se pak mění dynamicky podle toho, kolik různých symbolických názvů dané proměnné bylo nalezeno v popisu PLA. Proměnná `size` nesmí být větší než `exp_size`. Příznak `is_symbolic`, pokud je nastaven na 1, také říka, že hodnoty dané proměnné mají být v PLA popsány symbolickými hodnotami i ve výstupním PLA, a nikoliv číselnými hodnotami (to má smysl, pouze pokud nabývá daná proměnná jednoznačné hodnoty, nelze u částečtého či úplného don't care). Další příznak `mvnam` říká, zda jsou specifikovány symbolické názvy jednotlivých hodnot dané proměnné. Ano, pokud je nastaven na 1, ne pokud na 0. Pole řetězců `values` pak případně obsahuje tyto názvy hodnot.

Jak bylo uvedeno, struktura PLA obsahuje spojový seznam součinových termů. Struktura term vypadá takto :

```
typedef struct term     {
 term * next_term;
 int j;
 char binvars [I];
 char MVvar [MV] [N];
 char outputs [O];
 int marks [2];
 };
```

Proměnná `next_term` je ukazatel na další term, proměnná `j` udává pořadí termu v PLA; s touto hodnotou se v tomto balíku nepracuje, nicméně může být užitečná při revizích výpočtu. Pole znaků `binvars` obsahuje hodnoty vstupních binárních proměnných "0", "1". nebo "-", dvourozměrné pole `MVvar` osahuje hodnoty všech vícehodnotových proměnných, které nejsou považovány za výstupní, každá mv-proměnná je reprezentována polem znaků "0" a "1" dle poziční notace ESPRESSO, a pole znaků `outputs` obsahuje hodnoty výstupních proměnných "0", "1", nebo "-". Pole dvou příznaků `marks` obsahuje hodnoty které se nastavují a mají smysl při algoritmu funkce OR, při inicializaci jsou nastaveny na 0.

Při výpočtech nad několika PLA, musí být všechna PLA tzn. vstupní, vypočítaná i dočasná uložena v paměti tak, aby bylo možné s nimi jednoduše manipulovat, dělat výpočty nejen s načtenými vstupními PLA, ale i s vypočítanými PLA, která mohou sloužit jako mezivýsledky. Tato všechna PLA musí být možné vypisovat na výstup a také je jednoduše mazat. PLA jsou tedy uspořádána ve spojové struktuře PLAs :

```
typedef struct PLAs                {
 PLAs * next;
 int i;
 PLA * ppla;
 };
```

kde `next` je ukazatel na další prvek, `ppla` je ukazatel na strukturu PLA a proměnná `i` je pořadí daného PLA, na nějž je v tomto prvku reference, což slouží pro snadnou manipulaci uživatelem: vstupní i vypočtená PLA jsou identifikována celými čísly.

Globální proměnná `PLAs * firstPLA` je ukazatel na první prvek spojového seznamu `PLAs`.

5.2 Popis algoritmů

5.2.1 Nadkrychle termů (supercube)

Výsledkem nadkrychle několika termů, bez ohledu na výstupní hodnoty obsažených mintermů (ty jsou ignorovány), je nejmenší krychle (term), která obsahuje všechny zadané termy. Všechny zadané termy si musí odpovídat počtem vstupních proměnných a jejich rozměry. Nadkrychli počítáme postupně, pro každou proměnnou x takto: výsledný term bude mít v proměnné x všechny hodnoty, které se vyskytují v zadaných termech v proměnné x, a žádné jiné. U binárních proměnných, pokud mají všechny zadané termy v proměnné x 0, nebo všechny 1, bude ve výsledném termu také $x = 0$, resp. $x = 1$. Pokud se v proměnné x alespoň jeden term svou hodnotou liší od

ostatních, bude mít ve výsledném termu proměnná x hodnotu Don't care. Nadkrychle u vícehodnotových proměnných, které jsou zadány poziční notací, se počítá takto: u každé mv-proměnné x počítám pro každou pozici logický součet hodnot zadaných termů v dané pozici, protože pokud má alespoň jeden zadaný term na dané pozici 1, je třeba, aby tuto hodnotu v dané proměnné obsahoval i výsledný term.

Příklady

termy mají 5 binárních proměnných, 1 tříhodnotovou a 1 čtyřhodnotovou proměnnou

```
                              term 1 : -101- | 010 | 0101
                              term 2 : 1100- | 100 | 1100

nadkrychle termu 1 a termu 2 je term : -10-- | 110 | 1101

                              term 1 : 10-1- | 011 | 0110
                              term 2 : 010-1 | 101 | 1011

nadkrychle termu 1 a termu 2 je term : ----- | 111 | 1111
```

5.2.2 Průnik dvou termů

Výsledkem průniku dvou zadaných termů bez ohledu na výstupní hodnoty obsažených mintermů (ty jsou ignorovány) je term, který obsahuje právě ty mintermy, které obsahují oba zadané termy. Pokud nemají žádný společný minterm, průnik neexistuje. Oba zadané termy si musí odpovídat počtem vstupních proměnných a jejich rozměry. Průnik se počítá postupně, pro každou proměnnou hledáme, zda mají oba zadané termy nějaké společné hodnoty. Narazíme-li na proměnnou, ve které oba zadané termy nemají ani jednu společnou hodnotu, výpočet končí a průnik těchto termů neexistuje. U binárních proměnných je podmínka existence průniku splněna, když pro každou proměnnou x platí : oba termy mají v proměnné x buďto stejnou hodnotu, nebo alespoň jeden term má v proměnné x hodnotu Don't care. Pokud má jeden term $x = 0$ a druhý $x = 1$, průnik neexistuje, pokud mají oba termy stejnou hodnotu x, výsledný term má tutéž hodnotu x, pokud jeden z termů má hodnotu x Don't care, výsledný term má hodnotu x rovnou hodnotě x druhého termu. Průnik u vícehodnotových proměnných, které jsou zadány poziční notací, se počítá takto: u každé mv-proměnné x počítám pro každou pozici logický součin hodnot obou zadaných termů v dané pozici, protože pokud mají oba zadané termy na dané pozici 1, znamená to, že oba obsahují danou hodnotu. Pokud má výsledný term v dané mv-proměnné na všech pozicích 0, znamená to, že průnik neexistuje a výpočet končí.

Příklady

termy mají 5 binárních proměnných, 1 tříhodnotovou a 1 čtyřhodnotovou proměnnou

```
                term 1 : -1--1 | 011 | 1100
                term 2 : 0-0-1 | 101 | 0110

průnik termu 1 a termu 2 :  010-1 | 001 | 0100

                term 1 : -01-0 | 010 | 1010
                term 2 : 111-0 | 111 | 1011
```

průnik neexistuje, protože v druhé binární proměnné má term 1 hodnotu 0, kdežto term 2 hodnotu 1.

```
                term 1 : -1001 | 011 | 1100
                term 2 : 11001 | 110 | 0010
```

průnik neexistuje, protože ve čtyřhodnotové proměnné nemají oba termy žádnou společnou hodnotu (společnou jedničku na některé pozici).

5.2.3 Sjednocení dvou termů

Výsledkem operace sjednocení dvou termů není obecně jeden term. Tak by tomu bylo kdyby jeden term byl podmnožinou druhého, potom by výsledkem byl tento druhý term. Vzhledem k tomu, že minimalizace logických výrazů není součástí této práce, bude pro mě uspokojivým výsledkem sjednocení několika termů PLA, které obsahuje všechny tyto termy. Algoritmus pro sjednocení termů tedy spočívá pouze v přemístění daných termů do cílového PLA. Případné redundance pak může odstranit nějaký modul pro minimalizaci PLA.

5.2.4 Rozdíl termu od průniku s jiným termem

Výsledkem této operace je několik termů, tedy PLA s výslednými termy. Výstupy termů jsou v této operaci ignorovány. Zadány jsou dva termy. Účelem operace je oddělit od prvního termu takovou jeho část, která je sdílená také druhým termem. Výsledné termy, jsou takové termy, které jsou podmnožinou prvního termu, ale nemají průnik s druhým termem. Nejprve se spočítá průnik obou termů. Potom se hledá u každé proměnné, jestli u prvního termu daná proměnná nabývá více hodnot než u termu průniku. Potom výsledný

term bude mít v dané proměnné ty hodnoty, které má v dané proměnné první term, ale term průniku je v dané proměnné nemá. Ostatní proměnné budou mít hodnoty nezměněny tj. jako první term. Výsledkem bude tedy tolik termů, v kolika proměnných se liší první term od termu průniku.

Příklad

```
               term 1 : 0-1-0 | 011 | 1001
               term 2 : -1100 | 101 | 0101

spočítám jejich průnik : 01100 | 001 | 0001
```

a počítám rozdíl termu 1 od termu průniku :

```
               term 1 :  0-1-0 | 011 | 1001
               průnik :  01100 | 001 | 0001
                         ------------------
         výsled.termy :  001-0 | 011 | 1001
                         0-110 | 011 | 1001
                         0-1-0 | 010 | 1000
```

5.2.5 Co-singleton transformace

Princip transformace je popsán v kapitole 4. Jedné binární proměnné odpovídají dvě binární proměnné, n-hodnotové proměnné odpovídá n binárních proměnných.

Příklad

původnímu term s binárními proměnnými a, b, c, tříhodnotovou proměnnou d a čtyřhodnotovou proměnnou e odpovídá term s binárními proměnnými $a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, d_2, e_0, e_1, e_2, e_3$.

```
               abc   d     e
původní term : 0-1 | 011 | 1011

odpovídající tranformovaný term :

               a0 a1 b0 b1 c0 c1 d0 d1 d2 e0 e1 e2 e3
              -  1  -  -  1  -  1  -  -  -  1  -  -
```

Inverzní Co-singleton transformace převede transformovaný term na původní term.

5.2.6 Algoritmus pro logický součin (AND) dvou funkcí (PLA)

Tato operace je implementována pouze pro PLA typu `fd`. Vstupem pro operaci jsou dvě PLA, výstupem je jedno PLA. On-set výsledného PLA je průnikem On-setů vstupních PLA. Don't care-set výsledného PLA je průnikem On-setu prvního PLA s Don't care-setem druhého PLA, průnikem Don't care-setu prvního PLA s On-setem druhého PLA a průnikem Don't care-setu prvního PLA s Don't care-setem druhého PLA. Výpočet probíhá takto: pro každý term prvního PLA hledám průnik se všemi termy druhého PLA. Potom u každé takové dvojice termů, která má průnik kontroluju výstupní proměnné. Pokud má jeden term danou výstupní proměnnou rovnou 0 (význam 0 v typu `fd` viz kapitola 3.3.1), bude mít výsledný term v dané proměnné také hodnotu 0. Pokud má jeden term danou výstupní proměnnou rovnou 1 a druhý term - (Don't care), bude mít výsledný term v dané proměnné hodnotu - . Pokud mají oba termy v dané proměnné 1, výsledný term bude mít v dané proměnné také 1. Termy výsledného PLA tedy sestávají z termů, jejichž vstupní proměnné jsou výsledkem průniku dvojic termů z prvního a druhého vstupního PLA, a mají dopočítané výstupní proměnné právě zmíněným způsobem. Pokud ve výstupním PLA vznikne term, který má všechny výstupní proměnné rovny 0, bude odstraněn. Ve výstupním PLA se také můžou objevit dva termy, které mají totožnou vstupní část a jejich výstupní část se liší pouze tím, že v některé proměnné má jeden term 0, kdežto druhý term nějakou jinou hodnotu. Protože 0 nemá v PLA typu `fd` žádný význam, jsou oba termy vlastně totožné a správná výstupní hodnota dané proměnné je obsažena v druhém termu. Pokud je taková dvojice detekována, vznikne z ní jeden term, který přejímá vstupní část původních termů a výstupní část je upravena tak že nahradí nuly nenulovými hodnotami (pokud neobsahují nulovou hodnotu oba termy).

5.2.7 Algoritmus pro logický součet (OR) dvou funkcí (PLA)

Tato operace je implementována pouze pro PLA typu `fd`. Funkce reprezentovaná pomocí PLA je vlastně logický součet termů daného PLA. V prvním přiblížení bychom tedy mohli konstatovat, že logický součet dvou PLA vznikne shromážděním termů obou PLA v jednom výsledném PLA. Přitom by mělo platit, že minterm, který patří v jednom PLA do On-setu a v druhém do On-setu nebo Don't care-setu bude ve výsledném PLA patřit do On-setu (podle zákona logického součtu $1 + a = 1$). Pro typ PLA `fd` však platí: pokud v jednom PLA patří nějaký minterm zároveň do On-setu i do Don't care-setu, je chápán, že patří do Don't care-setu. Tedy pokud pracujeme s nějakým izolovaným termem, který má v nějakém výstupu hodnotu 1, není ještě jisté, zda jeho skutečná hodnota není - . To je jasné až po analýze celého PLA. Než tedy přistoupíme k vlastní operaci logického součtu dvou

PLA, je třeba zajistit v obou PLA jednoznačnost On-setu a Don't care-setu ve všech termech.

Je tedy třeba vždy vyhledat dvojici termů, která má neprázdný průnik a takových, že jeden má některý výstup roven 1 a druhý má tentýž výstup roven - . Term, který má v daném výstupu hodnotu - můžeme ponechat (ten má jistě celý v daném výstupu hodnotu -) a term, kteý má v daném výstupu hodnotu 1 musíme dekomponovat pomocí výše uvedené operace rozdílu. Výsledné termy po dekompozici neobsahují žádné mintermy patřící do průniku s Don't care-setem a u nich tedy můžeme v daném výstupu ponechat hodnotu 1. Toto je tzv. odstranění jednoduché kolize, kdy jeden term má v některých výstupech 1 a druhý má ve stejných výstupech hodnotu - . Může však také nastat tzv. dvojitá kolize, kdy jeden term má výstupy v On-setu tam, kde má druhý term tyto výstupy v Don't care-setu a zároveň první term má jiné výstupy v Don't care-setu, tam kde druhý term má výstupy v On-setu. Potom je potřeba ještě větší dekompozice, kdy provádíme operace rozdílu dvakrát - rozdíl prvního termu od průniku s druhým termem a rozdíl druhého termu od průniku s prvním termem. Výsledkem je term průniku, který má v původně kolidujících výstupech vždy - , a u termů vzniklých oddělením od průniku zůstávají výstupy nezměněny.

Poté, co je takto provedena jednoznačnost výstupů každého termu pro obě PLA, je možné zkopírovat termy obou vstupních PLA do výstupního PLA. Ve výsledném PLA se však znovu můžou objevit kolize On-setu a Don't care-setu, tyto kolize však musí být interpretovány jako výsledek operace OR, a tedy podle pravidel pro logický součin takových, že minterm vzniklý po operaci OR nad dvěma funkcemi takovými, že v první funkci patří do On-setu a v druhé funkci do Don't care-setu má ve výsledné funkci patřit do On-setu. Interpretace takového mintermu v jednom PLA však je, jak bylo již uvedeno právě opačná tj. že patří do Don't care-setu. Je tedy třeba opět provést jednoznačnost všech termů výsledného PLA analogickým způsobem jako to bylo provedeno ve vstupních PLA, avšak s opačnou podmínkou - tentokrát je kolize 1 a - chápána jako 1.

Algoritmus řeší také odstraňování duplicitních termů stejným způsobem jako algoritmus pro logický součin uvedený výše.

5.3 Popis funkcí programového balíku

`Funkce  term * MakeNewTerm (PLA * tempPLA)`

vytvoří nový prázdný term v některém PLA. Parametr `tempPLA` je ukazatel na PLA, ve kterém se má term vytvořit, funkce vrací ukazatel na nově vytvořený term. Nový term je připojen na konec spojového seznamu termů, jeho hodnota pořadí termu v daném PLA je nastavena na odpovídající pořadí

termu ve spojovém seznamu. Všechny proměnné daného termu jsou inicializovány hodnotou "!". Celé pole marks termu je inicializováno hodnotami 0.

Funkce `void CopyTerm (term * src, term * dest)`

zkopíruje všechny datové položky jednoho termu do termu jiného. Parametr `src` je ukazatel na zdrojový term, parametr `dest` je ukazatel na cílový term.

Funkce `term * FindTerm (int n, PLA * tempPLA)`

vyhledá term daného pořadí v daném PLA. Parametr `n` je pořadí daného termu, parametr `tempPLA` je ukazatel na dané PLA, funkce vrací ukazatel na nalezený term.

Funkce `void DeleteOneTerm (PLA * tempPLA, term * term_to_delete)`

odstraní term ze spojového seznamu některého PLA. Parametr `term_to_delete` je ukazatel na term, který má být odstraněn, parametr `tempPLA` je ukazatel na PLA obsahující daný term.

Funkce `int CountTerms (PLA * tempPLA)`

vrací počet termů nějakého PLA. Parametr `tempPLA` je ukazatel na dané PLA.

Funkce `int CompareTerm (term * origin, term * t)`

zjišťuje ekvivalentnost dvou termů. Parametr `origin` je ukazatel na vzorový term, s nímž má být porovnáván term na nějž ukazuje parametr `t`. Pokud jsou v obou termech totožné hodnoty všech vstupních proměnných a hodnoty výstupních proměnných si odpovídají s tím, že pokud některé výstupy mají u jednoho termu hodnotu 0 a u druhého jinou hodnotu jsou ekvivalentní, pouze první term nenese o dané proměnné v typu `fd` žádnou informaci, pak funkce vrací hodnotu 1. V proměnných kde má jeden term hodnotu 0 a druhý jinou se hodnota v termu na nějž ukazuje parametr `origin` nastaví na konkrétní nenulovou hodnotu. Pokud termy nejsou ekvivalentní, funkce vrací 0.

Funkce `void RemoveSameTerm (term * origin, PLA * tempPLA)`

odstraní pomocí funkce `DeleteOneTerm` z PLA všechny termy ekvivalentní s termem na nějž ukazuje parametr `origin`. Ekvivalence je zjišťována funkcí `CompareTerm`. Parametr `tempPLA` je ukazatel na PLA, v němž mají být takto odstraněny duplicity.

Funkce `void DeleteTerms (PLA * tempPLA)`

odstraní všechny termy daného PLA. Parametr `tempPLA` je ukazatel na dané PLA.

```
Funkce    PLA * MakeNewPLA (void)
```

vytvoří nové prázdné PLA a připojí ho na konec spojového seznamu všech PLA. Vrací ukazatel na nově vytvořené PLA.

```
Funkce    void RenumPLAs (void)
```

očísluje všechny položky spojového seznamu všech PLA dle pořadí v tomto seznamu. Tyto položky totiž mohou být očíslovány způsobem neodpovídajícím pořadí PLA vlivem přidávání nových PLA a mazání libovolných PLA ve spojovém seznamu.

```
Funkce    PLA * FindPLA (int n)
```

vyhledá PLA daného pořadového čísla ve spojovém seznam všech PLA. Parametr `n` je dané pořadové číslo, funkce vrací ukazatel na toto PLA.

```
Funkce    void CopyMainAttributes
              (PLA * src, PLA * dest)
```

zkopíruje esenciální informace z jednoho PLA do jiného. Těmito informacemi jsou počet všech proměnných a rozměry případných mv-proměnných. Parametr `src` je ukazatel na zdrojové PLA, parametr `dest` je ukazatel na cílové PLA.

```
Funkce    void CopySideAttributes
              (PLA * src, PLA * dest)
```

zkopíruje datové položky z jednoho PLA do jiného PLA, takové, které nekopíruje funkce `CopyMainAttributes`, kromě termů a příznaku mv-proměnné `is_symbolic`. Příznak `is_symbolic` je relevantní, pokud vypisujeme na výstup PLA, nad kterým nebyly prováděny žádné operace, neboť vypsaní hodnot některé mv-proměnné v symbolických názvech je nemožné, poku daná proměnná nabývá více než jedné hodnoty současně. Parametr `src` je ukazatel na zdrojové PLA, parametr `dest` je ukazatel na cílové PLA.

```
Funkce    void DeletePLA (PLAs * PLA_to_del)
```

smaže jedno PLA. Parametr `PLA_to_del` je ukazatel na PLA, které má být smazáno.

```
Funkce    void InitSpace (void)
```

vytvoří prázdný spojový seznam pro PLA.

```
Funkce    void DeleteSpace (void)
```

smaže celý spojový seznam všech PLA.

```
Funkce    term * SuperCube
          (term * sterm1, term * sterm2, term * dterm)
```

je algoritmus pro vypočítání nadkrychle dvou zadaných termů. Parametry `sterm1` a `sterm2` jsou ukazatele na vstupní termy, ukazatel `dterm` je ukazatel na výsledný term. Funkce vrací ukazatel na výsledný term.

```
Funkce    term * InterSection
          (term * sterm1, term * sterm2, term * dterm)
```

je algoritmus pro výpočet průniku dvou zadaných termů. Parametry `sterm1` a `sterm2` jsou ukazatele na vstupní termy, ukazatel `dterm` je ukazatel na výsledný term. Funkce vrací ukazatel na výsledný term nebo `NULL` v případě, že průnik neexistuje.

```
Funkce    int Separate
          (term * sterm1, term * sterm2, PLA * dPLA)
```

je algoritmus pro výpočet rozdílu prvního zadaného termu od průniku s druhým termem. Parametry `sterm1` a `sterm2` jsou ukazatele na vstupní termy, ukazatel `dPLA` je ukazatel na PLA, kam se mají uložit výsledné termy. Funkce vytváří nové termy pomocí funkce `MakeNewTerm`, počítá průnik pomocí funkce `InterSection`, maže redundantní termy pomocí funkce `RemoveSameTerm` a na konci výpočtu odstraní první term výsledného PLA, ve kterém je výsledek průniku vstupních termů, kterýžto do výsledného PLA nepatří. Funkce vrací 1, pokud vstupní termy mají průnik, v opačném případě výpočet rozdílu neproběhne a funkce vrací 0.

```
Funkce    int OutForAND
          (term * sterm1, term * sterm2, term * dterm)
```

nastaví výstupy termu na korektní hodnoty v PLA, které je výsledkem operace AND dvou PLA. Parametry `sterm1` a `sterm2` jsou ukazatele na vstupní termy, ukazatel `dterm` je ukazatel na výsledný term. Pokud jsou všechny výsledné výstupy daného PLA rovny nule a term je neplatný, funkce vrací 0. V opačném případě vrací 1.

```
Funkce    int ColisionOfSets (term * t1, term * t2)
```

detekuje kolize On-setu a Don't care-setu dvou termů. Parametry `t1` a `t2` jsou ukazatele na dané termy. Funkce vrací 0, pokud nebyla detekována kolize. Funkce vrací 1, pokud termy mají neprázdný průnik některé výstupy mají v prvním termu hodnotu 1, kdežto v druhém termu tytéž výstupy mají hodnotu - . Funkce vrací 2, pokud termy mají neprázdný průnik některé výstupy mají v prvním termu hodnotu - , kdežto v druhém termu tytéž výstupy mají hodnotu 1. Funkce vrací 3, pokud byla detekována tzv. dvojitá kolize (viz popis algoritmu OR výše).

Funkce **void** IgnoreOutputs (term * t)

nastaví výstupní hodnoty termu na 0, což je v typu PLA `fd` nulová informace. Toto může být použito při triviálních operacích nad termy, kdy hodnoty výstupů jsou irelevantní. Parametr `t` je ukazatel na daný term.

Funkce **void** DoubleColision
(term * sterm1, term * sterm2, term * dterm, **int** n)

upravuje výstupní hodnoty termu jenž je průnikem dvou termů, u kterých byla detekována dvojitá kolize On-setu a Don't care-setu (viz popis algoritmu OR výše). Parametr `n`, pokud má hodnotu 1, udává, že v dané kolizi má být hodnota interpretována jako - , pokud má hodnotu 2, v dané kolizi má být hodnota interpretována jako 1. Parametry `sterm1` a `sterm2` jsou ukazatele na vstupní termy, ukazatel `dterm` je ukazatel na výsledný term.

Funkce PLA * CoSingTrans (PLA * PLA_to_trans)

transformuje dané PLA CO-singleton transformací. Pokud je transformované PLA zobrazeno na výstup, objeví se v jeho popisu klíčové slovo `.cs`. Parametr `PLA_to_trans` je ukazatel na PLA, které má být transformováno, funkce vrací ukazatel na nově vzniklé transformované PLA.

Funkce PLA * InverseCoSing (PLA * PLA_to_trans)

provádí zpětnou Co-singleton transformaci daného PLA. V nově vzniklém PLA potom zaniká klíčový příznak `.cs`. Parametr `PLA_to_trans` je ukazatel na PLA, které má být transformováno, funkce vrací ukazatel na nově vzniklé transformované PLA.

Funkce **int** NoMarkedTerms (PLA * tempPLA)

zjišťuje zda je ukončen algoritmus provedení jednoznačnosti On-setu a Don't care-setu v daném PLA při funkci OR. Funkce OR totiž nalezené kolidující termy označí tak, že do příznak obou termů `marks [0]` nastaví na 1. Pokud je detekováno, že žádný z termů není takto označen, znamená to, že algoritmus odstraňování kolizí je dokončen a funce `NoMarkedTerms` vrací 1. V opačném případě vrací 0. Parametr `tempPLA` je ukazatel na dané PLA.

Funkce PLA * RemoveColisions (PLA * sourPLA, **int** n)

odstraňuje kolize On-setu a Don't care-setu při funkci OR. Hledá kolize a zjišťuje druh dané kolize pomocí funkcí `ColisionOfSets` a `InterSection`, termy, které kolidují označí tak, že jejich příznak `marks [0]` nastaví na 1, podle druhu kolize dekomponuje termy a umísťuje je do nového PLA, pomocí funkce `RemoveSameTerm` odstraňuje duplicitní termy. Když jsou ve výsledném PLA pomocí funkce `NoMarkedTerms` detekovány stále kolize, celý algoritmus `RemoveColisions` se rekurzivně volá na toto výsledné PLA a

vzniká nové PLA pro uložení výsledku pro tento nový výpočet. Toto se děje dokud funkce NoMarkedTerms nevrátí 1. Všechna mezivýsledková PLA jsou mazána (původní vstupní PLA nikoliv). Parametr `n`, pokud má hodnotu 1, udává, že v dané kolizi má být hodnota interpretována jako - , pokud má hodnotu 2, v dané kolizi má být hodnota interpretována jako 1. Parametr `sourPLA` je ukazatel na vstupní PLA, funkce vrací ukazatel na výsledné PLA bez kolizí.

```
Funkce    PLA * AND (PLA * pla1 , PLA * pla2 )
```

je algoritmus pro výpočet logického součinu dvou PLA. Používá funkci `InterSection` pro výpočet průniku termů, funkci `OutForAND` pro korektní nastavení výstupu a funkci `RemoveSameTerm` na odstranění duplicitních termů. Parametry `pla1` a `pla2` jsou ukazatele na vstupní PLA, funkce vrací ukazatel na výstupní PLA.

```
Funkce    PLA * OR (PLA * pla1 , PLA * pla2 )
```

je algoritmus pro logický součet dvou PLA. Nejprve odstraní kolize On-setu a Don't care-setu v obou vstupních PLA pomocí funkce `RemoveColisions` s parametrem $n = 1$. Potom označí termy z prvního PLA a z druhého PLA tak, že nastaví příznak $marks[1]$ u termů prvního PLA na 1 a u druhého na 2. Potom sesype všechny termy z obou PLA do výsledného PLA, přičemž odstraňuje duplicitní termy. Nakonec zavolá funkci `RemoveColisions` avšak s parametrem $n = 2$. Kolize se v tomto případě hledají jen mezi termy, které nejsou z téhož vstupního PLA, což je zajištěno růzností příznaků `marks [1]` a interpretováno ve funkci `RemoveColisions`. Opět se odstraňují duplicity. Parametry `pla1` a `pla2` jsou ukazatele na vstupní PLA, funkce vrací ukazatel na výstupní PLA.

```
Funkce Message1 ža Message12
```

jsou neimplementované výjimky, volané v případě načítání PLA ze souboru, které není korektně zadané.

```
Funkce    PLA * ReadPLA (char * filename )
```

načítá PLA ze vstupního souboru. Parametr `filename` je název vstupního souboru. Funkce vrací ukazatel na takto nově vzniklé PLA, v případě že se nepodařilo otevřít vstupní soubor, funkce vrací `NULL`.

```
Funkce    int WritePLA
              (char * filename ,PLA * PLA_to_write)
```

zapisuje PLA do výstupního souboru. Parametr `filename` je název výstupního souboru. Funkce vrací 0, pokud se podařilo otevřít soubor pro zápis PLA, v opačném případě vrací 1.

Kapitola 6

Závěr

V rámci této práce byly implementovány některé užitečné operace pro manipulaci s logickými, obecně vícehodnotovými funkcemi, reprezentovanými programovatelnými logickými poli (PLA).

Cílem bylo vytvořit takový programový balík, jehož algoritmy by byly účelné, takové, které jsou často potřeba při nejrůznějších aplikacích při analýze a syntéze logických systémů. V tomto balíku jsou implementovány algoritmy *průniku termů*, *sjednocení termů*, *nadkrychle termů*, *rozdíl termů*, *transformace na binární obraz* a odpovídající *zpětná transformace*, a *logický součin* a *součet funkcí*. Tyto algoritmy používají elementárnější funkce pro práci s vnitřní reprezentací jednotlivých PLA a jejich termů, které mohou být užitečné při případném obohacení balíku o další operace s logickými funkcemi.

Důležité bylo také vytvořit notaci vstupních a výstupních funkcí, která je běžně používaná. Proto jsem zvolil notaci ESPRESSO. Při načítání PLA do vnitřní datové struktury, program podporuje všechny informace, které se mohou vyskytnout v popisu PLA v dané notaci. Tyto informace slouží jako vstupní informace pro nejrůznější aplikace v analýze a syntéze logických obvodů.

Literatura

[1] R.Rudell and A.Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization", IEEE Trans. on CAD/ICAS, Vol.CAD-5, No.9, pp. 727-750, Sept. 1987

[2] M.Gao, J.-H.Jiang, Y.Jiang, Y.Li, S.Sinha and R.Brayton, "MVSIS," in Proc. Int. Workshop on Logic Synthesis, pp. 138-144, June 2001

[3] T.Sasao, "Multiple-valued logic and optimization of programable logic arrays," IEEE Computer, Vol.21, pp. 71-80, 1988

[4] G.D. De Micheli, R.Brayton and A.Sangiovanni-Vincentelli, "Optimal state assignment for finete state machines," IEEE Trans. on CAD/ICAS, Vol. CAD-4, No.3, pp. 269-284 July 1985

[5] S.L.Hurst, "Multiple-Valued Logic - its status and its future," IEEE Trans. on Computers, Vol. C-33, No. 12, pp. 1160-1179, Dec. 1984

[6] E.Dubrova and H.Sack, "Probablistic verification of multiple valued functions," in Proc. 30th Int. Symp. on Multiple-Valued Logic, pp.461-466, May 2000

[7] I.Ben Dhaou, E.Dubrova and H.Tenhunen 2001 "Power efficient inter-module communication for digit-serial architectures in deep-submicron technology," in Proc. 31st Int. Symp. on Multiple-Valued Logic, pp. 61-67, May 2001

[8] B.Ricco, G.Torell, M.Lanzoni, A.Manstretta, H.E.Maes, D.Montanari and A.Modelli, "Non-volatile multilevel memories for digitalapplications," in Proc IEEE, Vol.86, No.12, pp.2399-2421, Dec 1998

[9] T.Hanyu and M.Kameyama, "A 200 MHz pipelined multiplier using 1.5 V-supply multiple-valued MOS current-mode circuits with dual-rail source-coupled logic," IEEE Journal of Solid-State Circuits, Vol.30 No.11, pp.1239-1245, Nov. 1995

[10] K.Shimabukuro and C.Zukeran, "Reconfigurable current-mode multiple-valued residue arithmetic circuits," in Proc. 28th Int. Symp. on Multiple-Valued Logic, pp.282-287, May 1998

[11] The VIS Group, "VIS: A system for Verification and Synthesis," in Proc. 8th Int. Conf. on computer Aided Verification, Springer Lecture Notes in Computer Science, ed: Alur and Henzinger, Vol.1102, New Brunswick, NJ, pp.428-432, 1996

[12] Multi-valued Logic Synthesis, R. K. Brayton, and S. P. Khatri, International Conference on VLSI Design, Goa, India, Jan 1999.

[13] Semi-Algebraic Methods for Multi-Valued Logic, M. Gao, R. K. Brayton, IEEE IWLS 2000, International Workshop on Logic Synthesis 2000, Dana Point, CA, Workshop Notes 73-80, May 31 - June 2, 2000.

[14] A. Mishchenko, R. Brayton, and T. Sasao, "Exploring multi-valued minimization using binary methods", Proc. IWLS '03, pp. 278-285

[15] Don't Cares and Multi-Valued Logic Network Minimization, Y. Jiang, and R. K. Brayton, IEEE/ACM International Conference on CAD, ICCAD'00, Santa Clara, November 2000.

[16] J.-H. R. Jiang, A. Mishchenko, and R. K. Brayton, "Reducing multi-valued algebraic operations to binary", Proc. DATE '03, pp. 752-757

[17] A Boolean Paradigm in Multi-Valued Logic Synthesis, Alan Mishchenko and Robert K. Brayton, In the Notes of the International Workshop on Logic Synthesis, New Orleans, June 2002

[18] ttp://ece-www.colorado.edu/