České vysoké učení technické v Praze

Fakulta elektrotechnická

Katedra počítačů

Diplomová práce

Dekompozice logických funkcí založená na binárních rozhodovacích diagramech

Bc. Zdeněk Kalčík

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika, strukturovaný, Navazující magisterský

Obor: Výpočetní technika

Leden 2010

Poděkování

Děkuji vedoucímu diplomové práce Ing. Petru Fišerovi, Ph.D za jeho pomoc při tvorbě této diplomové práce.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 5. 1. 2010 ……

Abstract

This work deals with decomposition of logic function based on binary decision diagrams. The work describes the theory of logic functions and binary decision diagrams, then describes library CUDD, which is used as a tool for manipulating with binary decision diagrams.

The following part is an explanation of AND, OR, XOR decomposition, and other tools to minimize functions. This thesis also describes the implementation of tool for decomposition logic functions based on these types of decomposition.

The testing of the application itself is carried out in the last part of the thesis. Finally, the author also mentions other possibilities of extending his tool.

Abstrakt

Práce se zabývá dekompozicí logických funkcí založenou na binárních rozhodovacích diagramech. Nejprve práce popisuje teorii logických funkcí a binárních rozhodovacích diagramů, poté je popsána knihovna CUDD, která slouží jako nástroj pro manipulaci s binárními rozhodovacími diagramy.

Následuje vysvětlení dekompozic AND, OR, XOR a dalších nástrojů pro minimalizaci funkcí. V této diplomové práci je dále popsána implementace nástroje na minimalizaci logických funkcí založená na těchto typech dekompozic.

V závěru je provedeno testování nástroje a jsou zde probrány možnosti jeho dalšího rozšíření.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Tato diplomová práce vznikla na základě článku autorů Congguang Yang a Maciej Cieselski BDS: A BDD-Based Logic Optimization System, který řeší dekompozici logických funkcí založenou na binárních rozhodovacích diagramech (BDD [1]), a následnou implementací tohoto vědeckého pojednání.

Cílem dekompozice bývá (většinou současně):

1) Přizpůsobení technologii (dekompozice do hradel s omezeným počtem vstupů)
2) Zmenšení plochy výsledného obvodu

Druhů dekompozic je mnoho. V této práci je použita tzv. bi-dekompozice. Jedná se o dekompozici, která vytvoří F = X op Y (původní funkce je rozložena na dvě menší a mezi tyto funkce je vložen binární operátor (AND, OR, XOR…).

Pochopením principů dekompozice logických funkcí založené na binárních rozhodovacích diagramech je možno vytvořit nástroj pro dekompozici logických funkcí.

1.1 Vymezení požadavků a cílů

Základním cílem této diplomové práce je vytvoření nástroje pro dekompozici logických funkcí založený na dekompozici binárních rozhodovacích diagramů. Před samotnou implementací byly zvoleny následující požadavky:

- Načítání zadání pro dekompozici bude ze vstupního souboru formátu blif. Program by měl být dostatečně „robustní“ - měl by být schopný načítat i soubory, které obsahují některé nestandardní prvky (např. zakomentované řádky, prázdné řádky, dělené řádky apod.)
- Uložení struktury binárního rozhodovacího stromu z načteného souboru a jeho následné vytvoření
- Načtené a vytvořené BDD uložit do formátu blif a dot

[1] BDD - Binary decision diagram

- Implementovat dekompozice AND, OR, XOR
- Vytvořit heuristiku pro výběr vhodné dekompozice
- Navrhnout a vytvořit vhodnou strukturu pro dekompoziční strom (do něhož se budou ukládat výsledky jednotlivých dekompozicí)
- Zpracovat dekompoziční strom a uložit výsledek do souboru formátu blif
- Vytvořený nástroj otestovat na standardních zkušebních úlohách a získané výsledky porovnat s alternativními řešeními

1.2 Vývojové prostředí a programovací jazyk

Pro práci s binárními rozhodovacími stromy byla použita knihovna CUDD[2]. Tato knihovna je psaná v programovacím jazyku C a je určena pro platformu Linux. Proto jako platforma byla použita distribuce Linuxu – Ubuntu[3] ve verzi 9.04 Jaunty Jackalope. Tento operační systém je velice rychlý a jednoduchý na instalaci a pro implementaci této diplomové práce dostačující.

Jako programovací jazyk byl zvolen C++. Tento programovací jazyk je objektově orientovaný, který byl vyvinut rozšířením jazyka C. C++ byl zvolen pro svou rychlost a zpětnou kompatibilitu s C (kompatibilní s knihovnou CUDD).

Jako vývojové prostředí bylo zvoleno Eclipse[4]. Toto vývojové prostředí je sice určené pro programování v jazyce Java, ale díky různým rozšířením je možné programovat i v jazyce C++. Toto vývojové prostředí je zdarma stažitelné ze stránek výrobce a nabízí výborné debugovací[5] prostředky, které usnadňují hledání chyb v programu.

[2] CUDD - Colorado University Decision Diagram
[3] http://www.ubuntu.cz/
[4] http://www.eclipse.org/
[5] odlaďovací

1.3 Souhrn kapitol

- Ve druhé kapitole je popsána teorie booleovských funkcí a základní pojmy pro dekompozice
- Třetí kapitola stručně charakterizuje knihovnu CUDD
- Čtvrtá kapitola popisuje AND, OR a XOR dekompozice
- Pátá kapitola stručně popisuje implementaci nástroje na dekompozici logických funkcí
- V šesté kapitole je provedeno testování nástroje
- Sedmá kapitola se věnuje stručnému zhodnocení diplomové práce a jsou zde popsány možnosti budoucího rozvoje nástroje

2 Teorie

2.1 Booleovská funkce

Booleovská proměnná je proměnná, která nabývá pouze hodnot 0 nebo 1.

Úplně zadaná booleovská funkce s n vstupy a jedním výstupem je definovaná $f: B^n \rightarrow B$, kde B = {0, 1}. Tato funkce může být jednoznačně definována jejím onsetem, ON(f) = {x: $f(x)$ = 1}, a offsetem, OFF(f) = {x: $f(x)$ = 0}. Pro úplně definované booleovské funkce f a g, f pokrývá g, značeno jako $f \supseteq g$, jestliže ON(f) ⊇ ON(g).

Neúplně definovaná booleovská funkce s n vstupy a jedním výstupem je definovaná jako $f: Y^n \rightarrow Y$, kde Y = {0, 1, *} a * znamená don´t care (DC). Don´t care set (dc-set) neúplně definované booleovské funkce $f(x)$ je definován jako DC(f) = {x: $f(x)$ = *}.

Pokrytí F neúplně definované booleovské funkce f splňuje podmínku ON(f) ⊆ F ⊆ ON(f) ∪ DC(f).

Support booleovské funkce F, značeno jako supp(F), je definovaný jako množina proměnných, na kterých funkce F závisí.

2.2 Binární rozhodovací diagram

Binární rozhodovací diagram je datová struktura, která slouží k reprezentaci booleovských funkcí. Ukazuje se, že se jedná o velmi výhodnou reprezentaci těchto funkcí, především modifikované BDD - uspořádané binární rozhodovací diagramy (Ordered Binary Decision Diagrams – OBDD) a redukované upořádané binární rozhodovací diagramy (Reduced Ordered Binary Decision Diagrams – ROBDD).

2.2.1 Historie

Základní idea, ze které je tato datová struktura vytvořena, je Shanonův expanzní teorém[6]. Binární rozhodovací diagramy (BDD) poprvé představil C. Y. Lee [10].

[6] http://en.wikipedia.org/wiki/Shannon_expansion

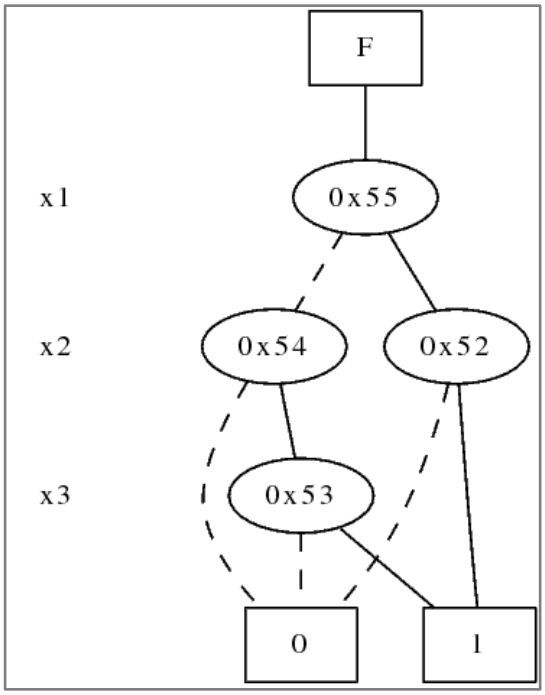
Další studie a vylepšení provedl Sheldon B. Akers[11] a Raymond T. Boute[12]. Plný potenciál pro efektivní algoritmy založené na binárních rozhodovacích diagramech přinesl Randal Bryant z univerzity Carnegie Mellon. Jeho hlavním přínosem bylo zavedení neměnného pořadí proměnných a sdílení podgrafů.

2.2.2 Definice BDD

Booleovská funkce může být reprezentována jako kořenový orientovaný acyklický graf, který se skládá z rozhodovacích uzlů a dvou terminálních uzlů zvaných 0-terminál a 1-terminál.

Každý neterminální uzel představuje binární proměnnou x z logické funkce a má dvě hrany vedoucí k potomkům. Jedna hrana je označována jako lo(x), je zakreslována do grafu dlouhými přerušovanými čarami a odpovídá případu, kdy je proměnné přiřazena hodnota 0. Druhá hrana, označována jako hi(x), je zakreslována do grafu nepřerušovanou čarou. Tato hrana odpovídá případu, kdy je proměnné přiřazena hodnota 1. Někdy jsou větve také označovány jako if-then (případ, kdy je proměnné přiřazena hodnota 1) a else (proměnné je přiřazena 0).

Na Obrázku 1 je příklad binárního rozhodovacího diagramu pro funkci $F = x_1x_2 + x_2x_3$.



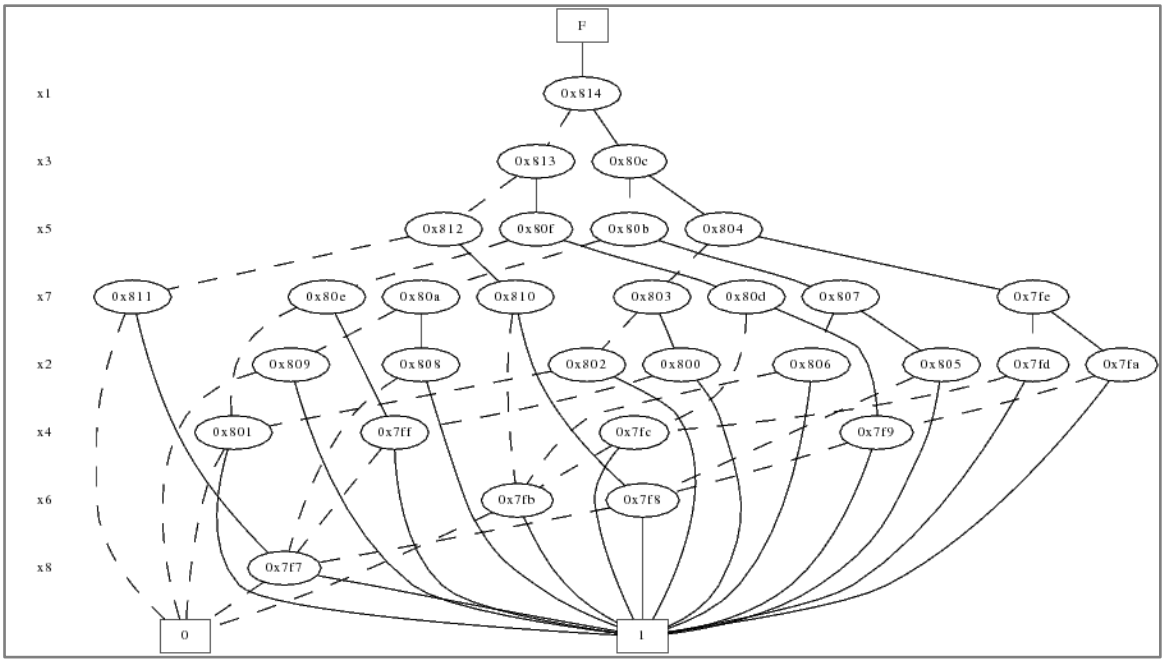
Obrázek 1: Ukázka binárního rozhodovacího diagramu pro funkci $F = x_1x_2 + x_2x_3$

Binární rozhodovací diagram může reprezentovat nekonečně mnoho logických funkcí, tzn. že vytvořením binárního rozhodovacího diagramu se ztrácí informace o syntaxi výrazu (binární rozhodovací diagram na Obrázku 1 může patřit klidně funkci $F = \overline{\overline{x1x2 + x2x3}}$).

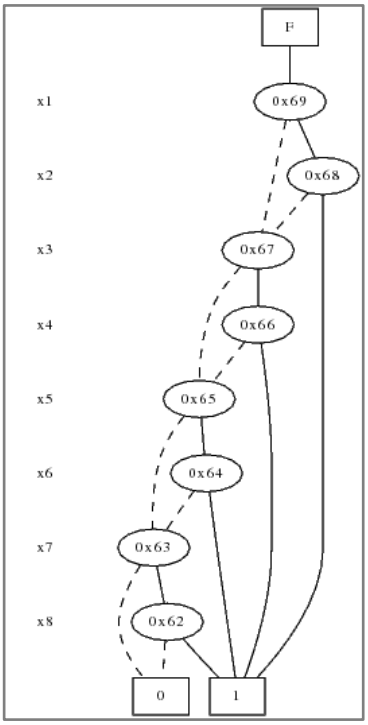
2.2.3 OBDD

Velikost binárního rozhodovacího diagramu závisí na funkci, kterou reprezentuje, a na zvoleném pořadí proměnných. Rozhodovací binární diagram nazýváme uspořádaný, jestliže pro každé dvě proměnné, pro které platí uspořádání x1 < x2, platí, že při všech možných průchodech (cestách) od kořene k listům binárního rozhodovacího diagramu narazíme na uzel, který je označený jménem proměnné x1, dříve než na uzel označený jménem proměnné x2.

Na Obrázku 2 a Obrázku 3 jsou znázorněny binární rozhodovací diagramy pro funkci $f = x_1x_2 + x_3x_4 + x_5x_6 + x_7x_8$ lišící se pouze volbou pořadí proměnných.



Obrázek 2: Binární rozhodovací diagram se špatným uspořádáním proměnných



Obrázek 3: Binární rozhodovací diagram s dobrým uspořádáním proměnných

Jak je vidět z Obrázku 2 a z Obrázku 3, tak vhodnou volbou pořadí proměnných klesl počet neterminálních uzlů z 30 na 8.

Obecně neexistuje pravidlo, podle kterého bychom mohli určit pořadí proměnných tak, abychom sestrojili diagram s minimálním počtem vnitřních uzlů. Pro booleovskou funkci můžeme sestrojit graf, jehož počet neterminálních uzlů bude v nejlepším případě lineární a v nejhorším případě exponenciální.

Existuje pravidlo pro booleovskou funkci $f(x_1, \ldots, x_n)$ zadanou ve tvaru $f(x_1, \ldots, x_n) = x_1x_2 + x_3x_4 + \ldots + x_{2n-1}x_{2n}$: lineárního počtu uzlů dosáhneme uspořádáním $x_1 < x_2 < x_3 < x_4 < \ldots < x_{2n-1} < x_{2n}$ a exponenciálního počtu uzlů dosáhneme uspořádáním $x_1 < x_3 < \ldots < x_{2n-1} < x_2 < x_4 < \ldots < x_{2n}$.

2.2.4 ROBDD

Jestliže je logická funkce reprezentována jako binární rozhodovací diagram, pak v této datové struktuře může docházet k redundancím (některé části diagramu se opakují). Toto je nežádoucí zejména z hlediska provádění operací na binárním rozhodovacím diagramu a z hlediska velikosti datové struktury.

V BDD mohou vznikat tyto redundance:

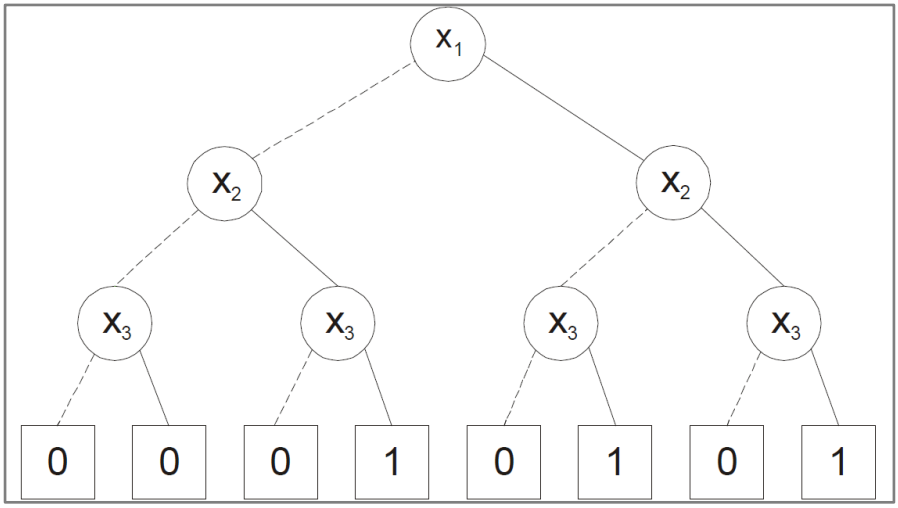
1) Opakující se terminální uzel se stejnou hodnotou
2) Jestliže pro některý uzel x si hi(x) a lo(x) odpovídají, pak z hlediska sémantiky nemá tento uzel žádný význam
3) Uvnitř stromové struktury BDD se vyskytují podstromy, které nesou stejnou informaci – jsou identické

Definujeme tři transformační pravidla, která vedou k odstranění těchto redundancí a zároveň nemění sémantiku funkce:

A. **Odstranění opakujících se terminálů** – odstranění všech terminálních uzlů a ponechání pouze jednoho 0-terminálu a 1-terminálu a hrany vedoucí do odstraněných uzlů se přesměrují na tyto ponechané terminály
B. **Odstranění nadbytečných vnitřních uzlů** – jestliže si oba potomci daného uzlu x odpovídají (hi(x) = lo(x)), pak se tento uzel odstraní a všechny hrany vedoucí do tohoto uzlu se přesměrují do jednoho z potomků (je jedno do kterého, jelikož se jedná o stejný uzel)

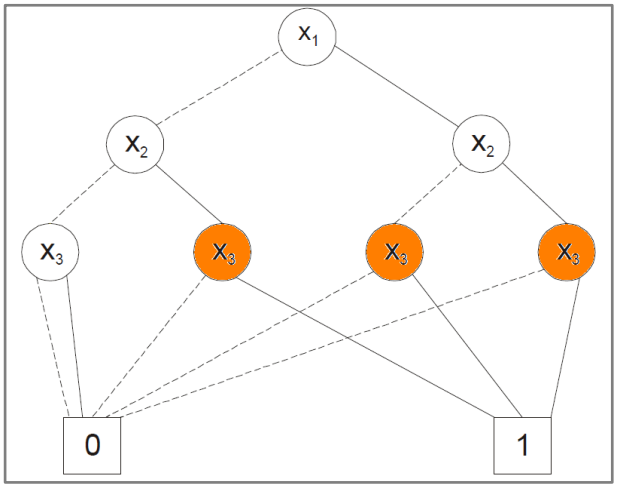
C. **Odstranění opakujících se vnitřních uzlů** – jestliže existují dva identické podstromy – tzn. pro dva uzly x a y platí, že levý potomek uzlu x je stejný jako levý potomek uzlu y a zároveň pravý potomek uzlu x je stejný jako pravý potomek uzlu y (hi(x) = hi(y) a zároveň lo(x) = lo(y)), pak se odstraní jeden z těchto uzlů a hrany vedoucí do odstraněného uzlu se přesměrují do neodstraněného

Tato tři pravidla se aplikují do té doby, dokud se dané BDD redukuje a dokud je možné aplikovat některé z těchto pravidel.



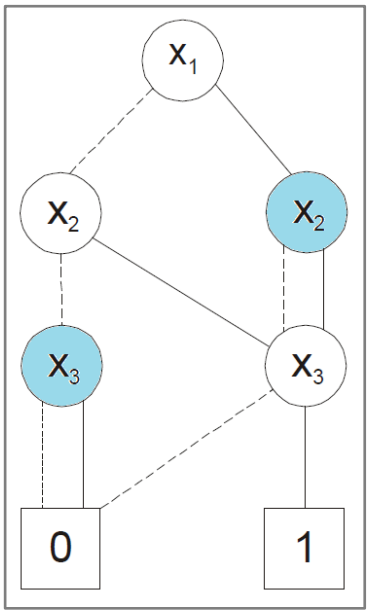
Obrázek 4: neredukované OBDD

Na Obrázku 4 je zobrazeno neredukované OBDD s uspořádáním $x_1 < x_2 < x_3$. Aplikací všech redukčních pravidel vznikne ROBDD.



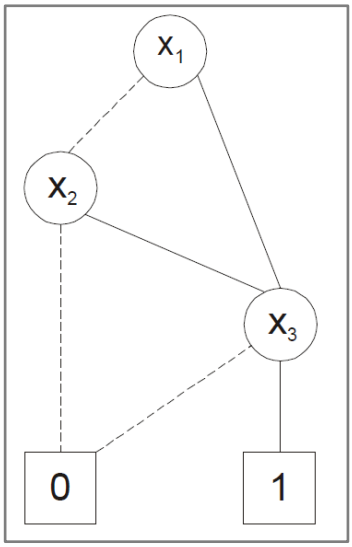
Obrázek 5: Odstranění opakujících se terminálů

Obrázek 5 ukazuje výsledek aplikace pravidla odstranění opakujících se terminálů. Z původních osmi terminálů zbyly jen dva (0-terminál a 1-terminál) a hrany vedoucí do odstraněných terminálů byly přesměrovány do zbylých terminálů.



Obrázek 6: Odstranění opakujících se vnitřních uzlů

Obrázek 6 ukazuje výsledek aplikace pravidla odstranění opakujících se vnitřních uzlů. Z Obrázku 5 je vidět, že tři neterminální uzly proměnné x_3 (na obrázku vyznačené oranžovou barvou) mají stejné podstromy. Proto byl jeden uzel zachován a hrany vedoucí na odstraněné uzly byly přesměrovány do tohoto uzlu.



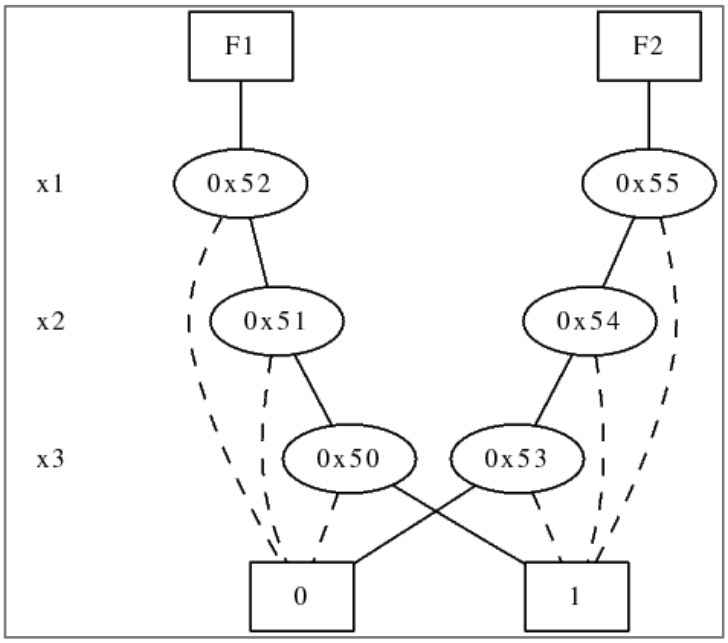
Obrázek 7: Odstranění nadbytečných vnitřních uzlů

Obrázek 7 ukazuje výsledek aplikace pravidla odstranění nadbytečných vnitřních uzlů. Na Obrázku 6 jsou vidět dva uzly (vyznačeny modrou barvou), jejichž potomci vedou do stejného uzlu. Tyto uzly byly odstraněny a hrany vedoucí do těchto uzlů byly přesměrovány do jejich potomků.

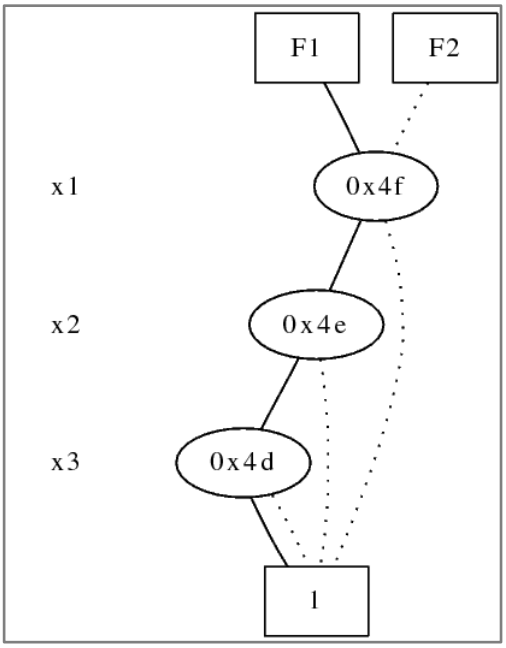
Z původních patnácti uzlů (terminálů i neterminálů) byl diagram redukován na redukovaný uspořádaný rozhodovací binární diagram s pěti uzly.

2.2.5 Komplementované (negované) hrany

Dalším velmi efektivním vylepšením je použití tzv. negovaných hran. Tato technika je založena na faktu, že redukované uspořádané binární diagramy funkce f a její negace $\bar{f}$ se liší jen v tom, že hodnoty jejich uzlů jsou navzájem vyměněné. Přidáním nové jednobitové informace o každé hraně docílíme značné úspory paměti. Pokud tento bit není nastaven, je podgraf ROBDD interpretován normálně (jako f). Pokud tento bit je nastaven, pak je podgraf ROBDD interpretován negovaně (jako $\bar{f}$). Díky tomu mohou být funkce f a $\bar{f}$ reprezentovány pomocí jednoho ROBDD. Také stačí pouze jeden terminální uzel (1-terminál).



Obrázek 8: Binární rozhodovací diagram bez použití negovaných hran



Obrázek 9: Binární rozhodovací diagram s použitím negovaných hran

Na Obrázku 8 a Obrázku 9 jsou zobrazeny stejné funkce F1 = $x_1x_2x_3$ a F2 = $\overline{F1}$. Obrázek 8 ukazuje binární rozhodovací diagram těchto funkcí bez použití negovaných hran a Obrázek 9 ukazuje binární rozhodovací diagram těchto funkcí s použitím negovaných hran. Úspora místa je značná – z původních 6 uzlů a 2 terminálů vznikly 3 uzly a 1 terminál.

Hlavním problémem použití této reprezentace je, že při výskytu negovaných hran v ROBDD ztrácíme kanonickou reprezentaci. K zachování kanonicity musí být dodrženo pravidlo, že negovaná hrana musí být přiřazena vždy else větvi (větev, do které se jde, když proměnná je ohodnocena nulou). Více v [4] a [5]. V grafech je negovaná hrana zobrazena jako tečkovaná čára.

2.2.6 Použití BDD

Binární rozhodovací diagramy mají v praxi široké využití. Pomocí ROBDD bylo vyřešeno mnoho problémů například v oblasti návrhu číslicových systémů, v oblasti umělé inteligence, v oblasti konečných automatů, matematické logice a dalších oblastech.

V oblasti číslicových systémů se ROBDD používají například při ověřování, testuje se ekvivalence dvou kombinačních logických obvodů. ROBDD se také využívají při opravování konstrukčních chyb, v citlivostních analýzách, pravděpodobnostních analýzách atd.

2.3 Terminologie a základy pro dekompozice

BDD

BDD je orientovaný acyklický graf reprezentující booleovskou funkci. Je jednoznačně definován jako čtveřice: BDD = (Φ, V, E, {0, 1}), kde Φ je kořen (root), V je množina uzlů, E je množina hran a 0 a 1 jsou terminální uzly.

Listové hrany

Listová hrana je hrana e ∈ E, která je přímo spojená s terminálním uzlem binárního rozhodovacího diagramu. Množina všech listových hran je značena Σ a může být rozdělena na Σ_0 (množina listových hran vedoucí do 0) a Σ_1 (množina listových hran vedoucí do 1). Všechny zbylé hrany se nazývají vnitřní.

Cesty

Cesta vedoucí od kořene do terminálního uzlu 0 (1) se nazývá 0-cesta (1-cesta). Množina všech 0-cest (1-cest) se značí Π_0 (Π_1). Množina všech cest binárního rozhodovacího diagramu se značí Π ($\Pi = \Pi_0 \cup \Pi_1$).

Řez

Řez binárního rozhodovacího diagramu je množina hran, které rozdělují uzly V na dvě disjunktní množiny – D a (V – D). Kořen patří do D ($\Phi \in$ D) a terminály 0, 1 patří do (V – D). Horizontální řez je takový řez, který rozdělí proměnné na dvě disjunktní množiny.

AND dekompozice

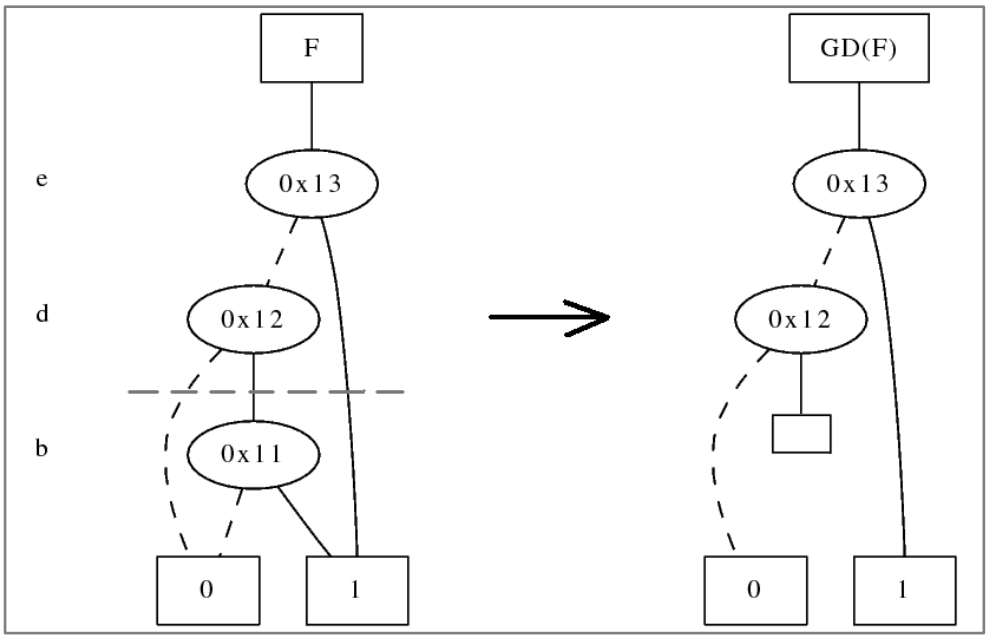
Booleovská funkce F má AND dekompozici, jestliže může být reprezentována jako F = D * Q. Funkce D se nazývá booleovský dělitel (boolean divisor) a Q se nazývá kvocient (quotient).

OR dekompozice

Booleovská funkce F má OR dekompozici, jestliže může být reprezentována jako F = G + H.

Zobecněný dominátor (Generalized dominator)

Je dán řez funkce F, který dělí binární rozhodovací diagram na dvě části. Zobecněný dominátor vznikne tak, že hrany řezu (hrany, které protíná řez) vedoucí do terminálu se ponechají a hrany řezu vedoucí do neterminálních uzlů se nechají volné (free edges). Na Obrázku 10 je ukázána tvorba zobecněného dominátoru pro funkci F = e + db a daný řez.



Obrázek 10: Tvorba zobecněného dominátoru pro daný graf a řez

3 Knihovna CUDD

3.1 Seznámení

Knihovna CUDD (Colorado University Decision Diagram) poskytuje funkce pro manipulaci s binárními rozhodovacími diagramy (BDD – binary decision diagram), algebraickými rozhodovacími diagramy (ADD – algebraic decision diagram) a rozhodovacími diagramy s potlačenou nulou (ZDD - zero-suppressed binary decision diagrams). Dále umožňuje přeměnit jeden typ diagramu na druhý. Obsahuje rovněž velkou kolekci algoritmů pro uspořádávání proměnných.

Knihovna CUDD je vyvíjena od roku 1996, kdy ji napsal Fabio Somenzi na katedře elektrotechniky a výpočetní techniky na univerzitě Colorado. Aktuální verze knihovny je 2.4.2.

CUDD je možné používat třemi způsoby:

1) **Jako černou skříňku** (black box) – v tomto případě aplikace, která potřebuje manipulovat s rozhodovacími diagramy, používá pouze exportované funkce knihovny CUDD. Značné množství funkcí obsažené v CUDDu dovoluje, aby mnoho aplikací bylo možné psát tímto způsobem. Samotná aplikace se nemusí starat o detaily uspořádání proměnných – to se děje automaticky na pozadí. Na adrese http://vlsi.colorado.edu/~fabio/CUDD/cuddExtAbs.html je možné najít seznam exportovaných funkcí
2) **Jako průhlednou skříňku** (clear box) – při psaní složitějších aplikací založených na binárních rozhodovacích diagramech je někdy potřeba naimplementovat vlastní funkce a ty pak přidat do CUDDu. Na adrese http://vlsi.colorado.edu/~fabio/CUDD/cuddAllAbs.html je možné najít seznam exportovaných a vnitřních funkcí
3) **Prostřednictvím interface** (through an interface) - objektově orientované jazyky jako C++ a Perl5 umožňují osvobodit programátora od správy paměti. C++ rozhraní je zahrnuto přímo v distribuci – toto rozhraní

automaticky uvolňuje rozhodovací diagramy, které již nejsou potřeba. Perl5 rozhraní existuje samostatně

3.2 Základní datové struktury

Základní datové struktury pro BDD, ADD a ZDD jsou uzly (DdNode), manager (DdManager) a cache tabulka. Tyto struktury jsou detailněji popsány v příloze A.

3.3 Příklad použití

V následujícím odstavci je ukázáno použití knihovny CUDD k vytvoření funkce $F = x_1x_2x_3x_4$.

```
#include <iostream>
using namespace std;

#include "cudd.h"

int main(int argc, char* argv[]) {
  DdNode *f, *var, *tmp;

  Cudd_PrintVersion(stdout);

  DdManager *manager = Cudd_Init(0, 0, CUDD_UNIQUE_SLOTS,
  CUDD_CACHE_SLOTS, 0);

  f = Cudd_ReadOne(manager);
  Cudd_Ref(f);
  for (int i = 3; i >= 0; i--) {
    var = Cudd_bddIthVar(manager, i);
    tmp = Cudd_bddAnd(manager, var, f);
    Cudd_Ref(tmp);
    Cudd_RecursiveDeref(manager, f);
    f = tmp;
  }
  FILE *file = fopen("graf.dot", "w");
  DdNode **outputs = new DdNode*[1];
  outputs[0] = f;
```

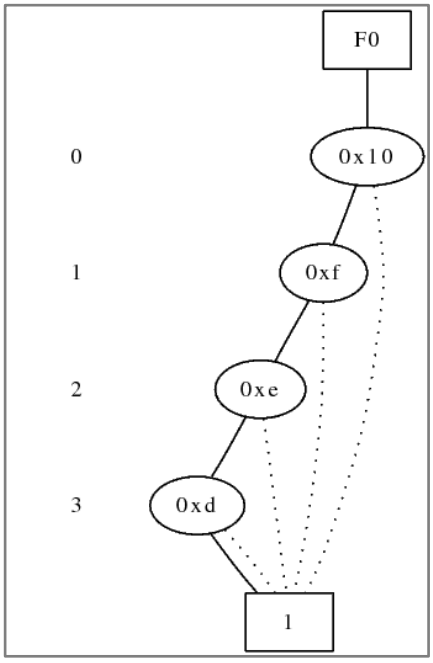
```
    Cudd_DumpDot(manager, 1, outputs, NULL, NULL, file);

    Cudd_Quit(manager);
}
```

Nejdříve je nutné naimportovat knihovnu CUDD. To se dělá direktivou #include "cudd.h". Dále je volána funkce z knihovny pro vytisknutí verze CUDDu (Cudd_PrintVersion). Poté je inicializován manager s defaultními parametry (Cudd_Init).

Následuje vytvoření konstantní funkce 1. Jelikož tvorba diagramu je efektivnější odspodu, tak for cyklus jde od 3 do 0 (od x_4 do x_1). Při každé iteraci for cyklu se inicializuje proměnná (Cudd_bddIthVar), přidává se k funkci f (Cudd_bddAnd), referuje nově vzniklá funkce (Cudd_Ref) a dereferuje stará funkce (Cudd_RecursiveDeref).

Na závěr je výsledná funkce zapsána do souboru graf.dot ve formátu dot (Cudd_DumpDot) a poté je ukončen manager (Cudd_Quit). Výsledná funkce je vidět na Obrázku 11.



Obrázek 11: BDD pro funkci F = $x_1x_2x_3x_4$

4 Dekompozice

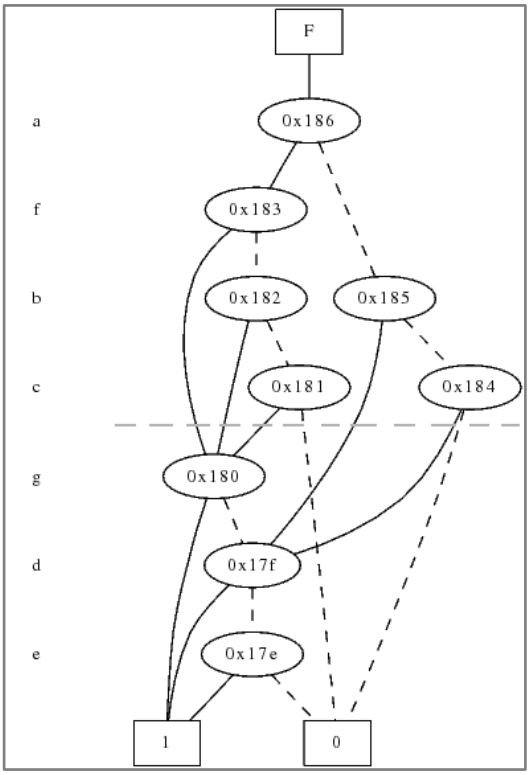
4.1 AND a OR dekompozice

4.1.1 AND dekompozice

Jestliže je zkonstruovaný zobecněný dominátor GD(F) funkce F, tak booleovský dělitel D se získá tak, že volné hrany se přesměrují do terminálu 1. Kvocient Q se získá tak, že hrany uzlů nad řezem vedoucí do 0-terminálu se přesměrují do don´t care (DC) uzlu. Důkaz tohoto tvrzení je napsán v [1].

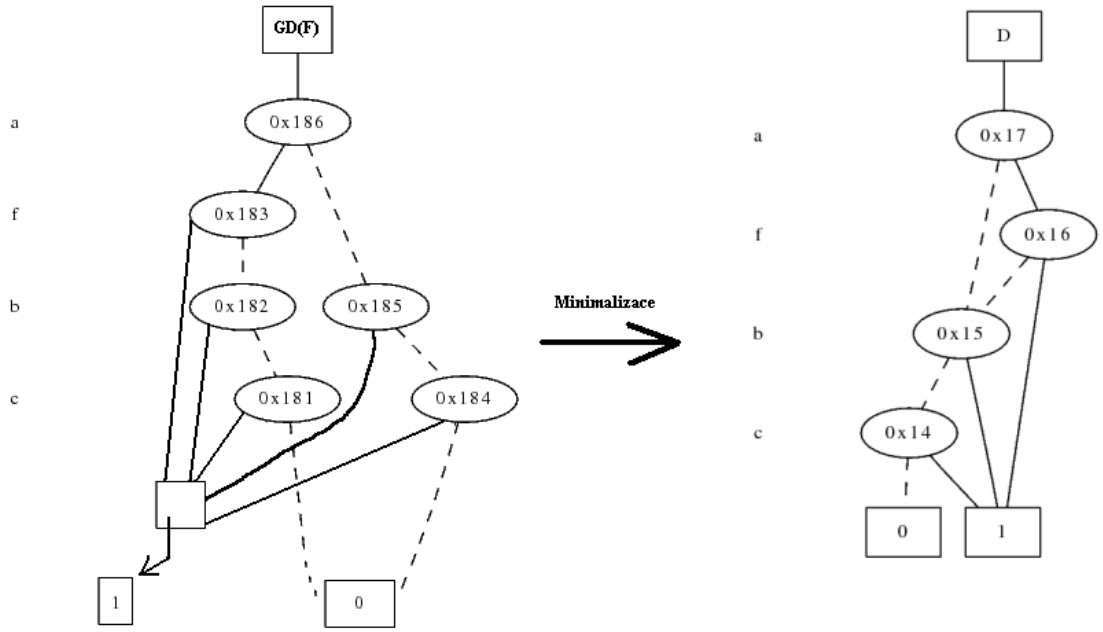
Výsledná funkce pak vznikne jako F = D * Q.

Na následujících třech obrázcích bude ukázána AND dekompozice pro funkci F a daný řez. Na Obrázku 12 je zobrazena funkce F a její řez.



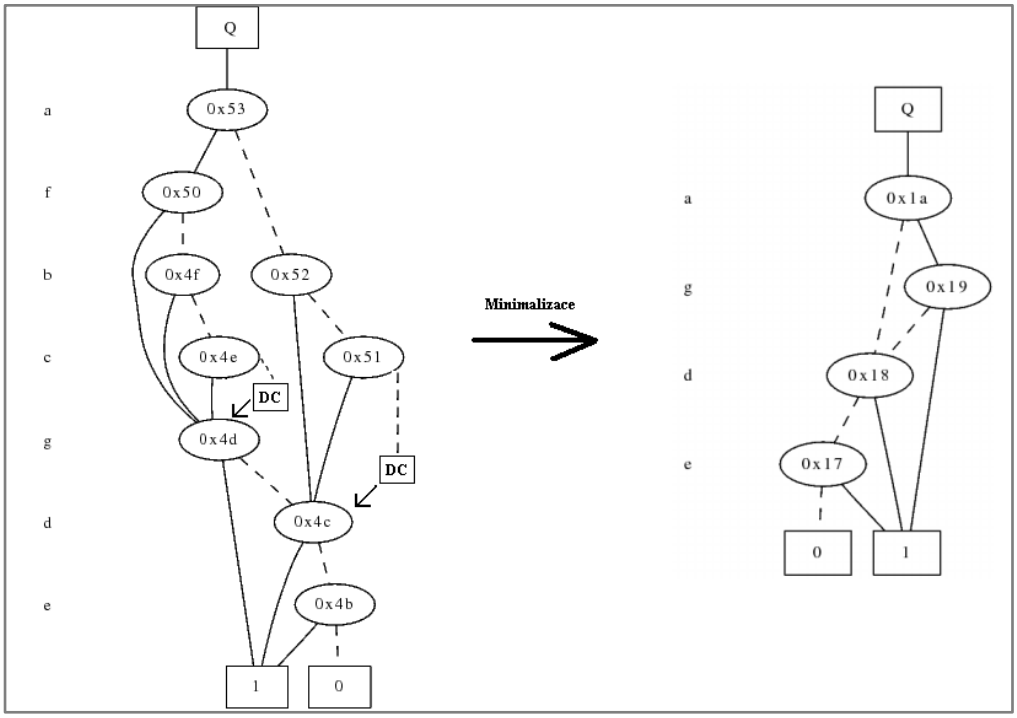
Obrázek 12: Funkce F a řez pro AND dekompozici

Nejprve je vytvořen zobecněný dominátor GD (hrany řezu vedoucí do neterminálního uzlu se nechají volné) a z něho je poté vytvořen booleovský dělitel D (tyto volné hrany se přesměrují do 1-terminálu). Vše je vidět na Obrázku 13.



Obrázek 13: Tvorba zobecněného dominátoru a booleovského dělitele

Dále se pak vytvoří kvocient Q, který vznikne tak, že hrany nad řezem jdoucí do 0-terminálu se přesměrují do don´t care DC. Následnou minimalizací vnikne kvocient Q pro danou funkci F. Vše je vidět na Obrázku 14.



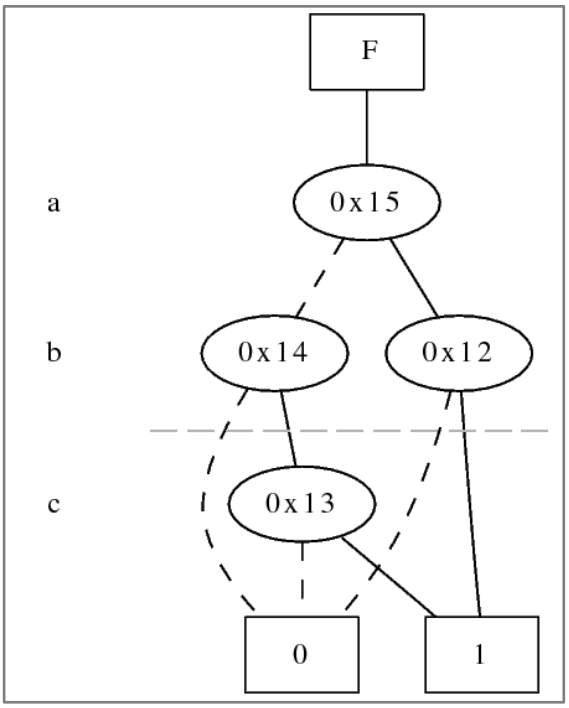
Obrázek 14: Tvorba kvocientu Q

4.1.2 OR dekompozice

OR dekompozice je podobná AND dekompozici. Pro zkonstruovaný GD(F) funkce F se funkce G získá tak, že volné hrany se přesměrují do terminálu 0. Funkce H se získá tak, že hrany uzlů nad řezem vedoucí do 1-terminálu se přesměrují do don´t care (DC) uzlu. Důkaz tohoto tvrzení je napsán v [1].

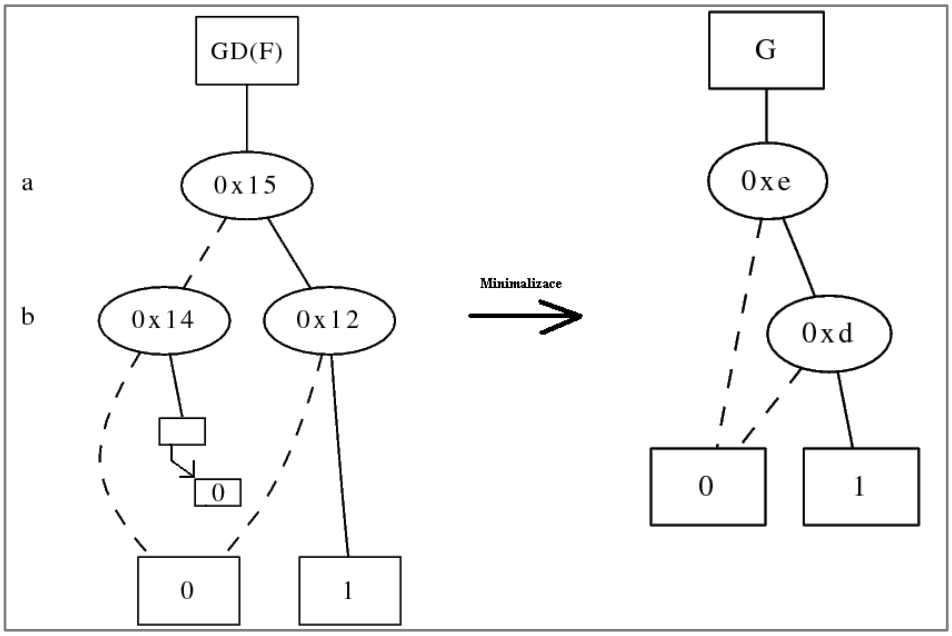
Výsledná funkce pak vznikne jako F = G + H.

Na následujících třech obrázcích bude ukázána OR dekompozice pro funkci F a daný řez. Na Obrázku 15 je zobrazena funkce F (F = ab + bc) a její řez.



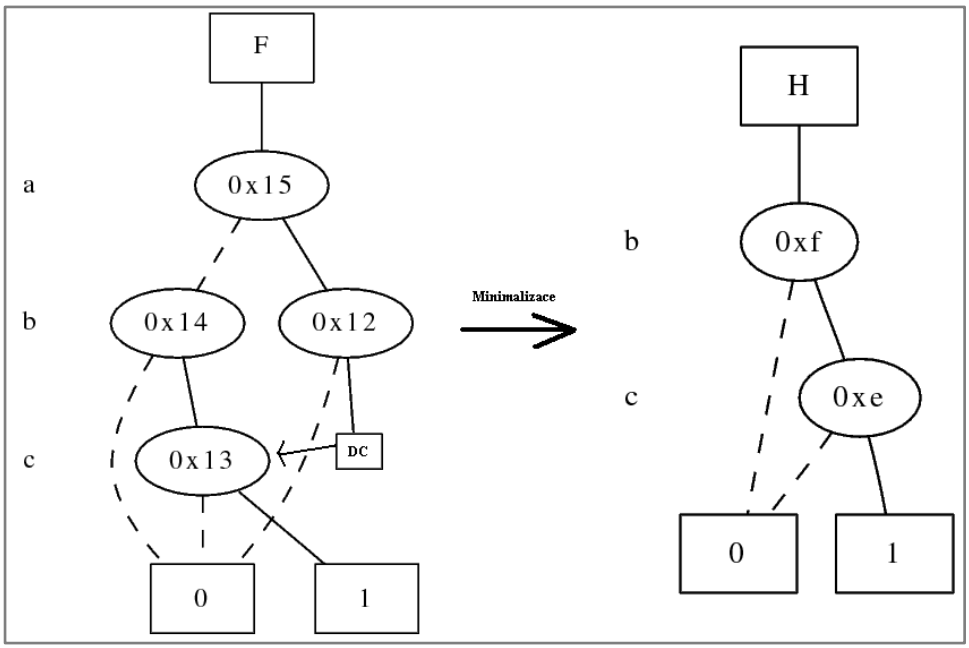
Obrázek 15: Funkce F a její řez pro OR dekompozici

Nejprve je vytvořen zobecněný dominátor GD (hrany řezu vedoucí do neterminálního uzlu se nechají volné) a z něho je poté vytvořena funkce G (tyto volné hrany se přesměrují do 0-terminálu). Vše je vidět na Obrázku 16.



Obrázek 16: Tvorba zobecněného dominátoru a funkce G

Dále se pak vytvoří funkce H, která vznikne tak, že hrany nad řezem jdoucí do 1-terminálu se přesměrují do don´t care DC. Následnou minimalizací vnikne funkce H pro danou funkci F. Vše je vidět na Obrázku 17.

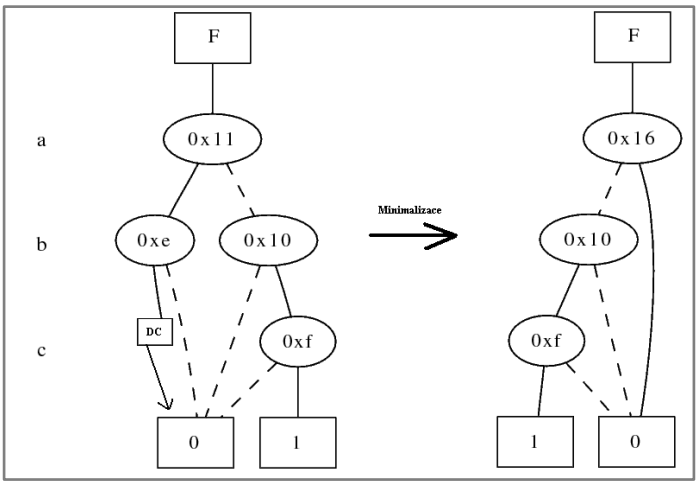


Obrázek 17 Tvorba funkce H

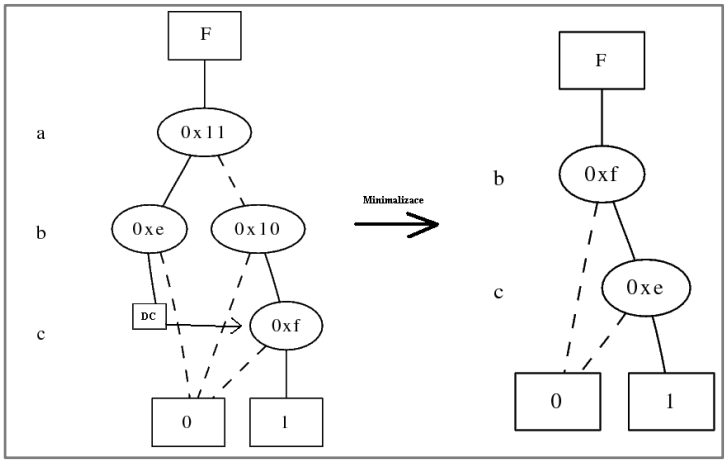
4.1.3 Minimalizace pomocí don´t care

V AND (OR) dekompozici při tvorbě Q (H) je použita minimalizace pomocí don´t care. Tato minimalizace je nezbytná část dekompozic AND i OR. Minimalizace pomocí don´t care se ukázala jako NP-úplný problém [7], [8]. Existuje jen několik heuristik, které tento problém řeší.

V této diplomové práci řeším tuto minimalizaci pouze tak, že potomka uzlu směřujícího do DC přesměruji na druhého potomka daného uzlu, což způsobí odstranění nadbytečného uzlu, viz. kapitola 2.2.4 ROBDD.



Obrázek 18: Minimalizace pomocí DC – přesměrování na druhého potomka



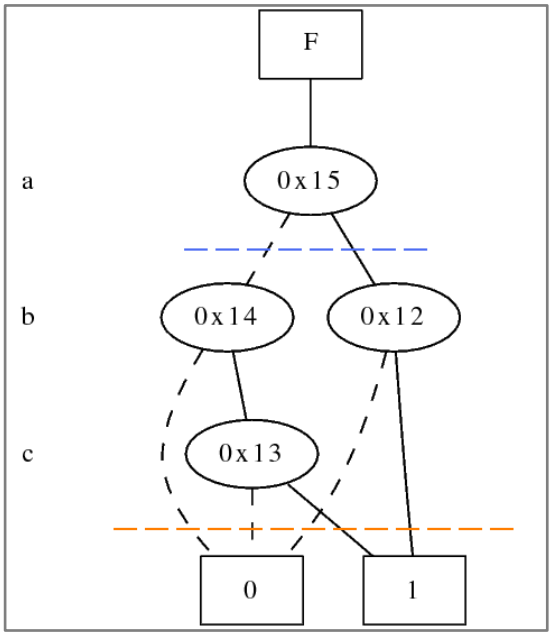
Obrázek 19: Minimalizace pomocí DC – lepší varianta

Z Obrázku 18 a Obrázku 19 je patrné, že DC minimalizace pomocí přesměrování na druhého potomka nedává nejlepší minimalizaci. Na druhou stranu tato minimalizace není výpočetně vůbec náročná.

4.1.4 Užitečné řezy

Množství řezů v diagramu může být značné i pro středně velké BDD. Proto byl vynalezen mechanismus, který některé řezy zneplatní.

V [9] je dokázáno, že řezy, které obsahují alespoň jednu listovou hranu, mohou vést k netriviální dekompozici. Označují se jako užitečné řezy. Dále je nutné zajistit, aby řez neobsahoval všechny terminální hrany, tzn. aby řez AND dekompozice neobsahoval všechny hrany, které vedou do 0-terminálu, a řez OR dekompozice neobsahoval všechny hrany, které vedou do 1-terminálu.



Obrázek 20: Extrémní případy řezů

Na Obrázku 20 jsou vidět dva extrémní případy řezů. Modrý řez neobsahuje ani jednu hranu vedoucí do terminálu. Tzn. že booleovský dělitel pro AND dekompozici bude roven D = 1, jelikož volné hrany ze zobecněného dominátoru se přesměrují do 1-terminálu (pro OR dekompozici vznikne G = 0) a kvocient bude roven Q = F (pro OR

dekompozici H = F). Výsledná funkce bude F = D * Q = 1 * F = F (pro OR dekompozici F = G + H = 0 + F = F).

Oranžový řez obsahuje všechny hrany, které vedou do terminálů. Booleovský dělitel pro AND dekompozici bude D = F, jelikož graf neobsahuje žádnou volnou hranu (pro OR dekompozici G = F) a kvocient bude Q = 1 (pro OR dekompozici H = 0). Ve výsledku bude F = F jako pro modrý řez.

V této diplomové práci se omezuji pouze na horizontální řezy. Nehorizontální řezy mohou vést k lepším výsledkům, ale značně zvýší výpočtovou složitost. V případě horizontálních řezů je maximální počet řezů roven |V|, kde V je počet proměnných (úrovně binárního rozhodovacího diagramu).

4.2 XOR dekompozice

BDD dekompozice založená na zobecněném dominátoru (kapitola 4.1) závisí na hranách, které vedou do terminálů (listové hrany). Zatímco AND a OR intenzivní funkce mívají mnoho listových hran, XOR intenzivní funkce mají velmi málo listových hran. Je zřejmé, že dekompozice závislá na listových hranách nebude pro tyto XOR intenzivní funkce fungovat.

XOR dekompozice je schopná si poradit s XOR intenzivními funkcemi. Bylo zjištěno [1], že XOR dekompozice je spojená s přítomností komplementovaných hran v binárním rozhodovacím diagramu. Proto se jako reprezentace BDD používají grafy s komplementovanými hranami. Namísto XOR funkce bude používána XNOR funkce, která je přímočařejší na vývoj.

Po aplikaci XOR dekompozice vzniknou dvě nové funkce G a H. Jestliže support těchto funkcí (supp(G) a supp (H)) jsou disjunktní množiny, pak se jedná o algebraickou dekompozici, v opačném případě se jedná o booleovskou dekompozici.

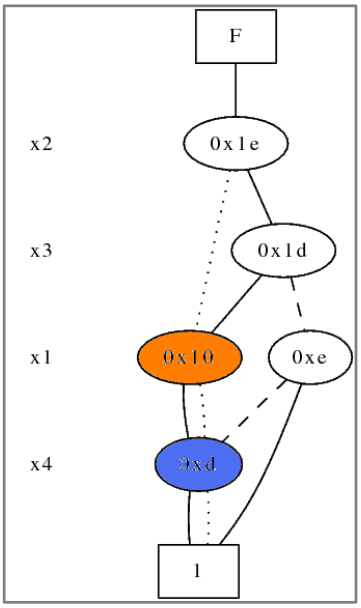
4.2.1 Booleovská

Podobně jako se v AND a OR dekompozici hledá zobecněný dominátor, tak v booleovské XOR dekompozici se hledá zobecněný x-dominátor. Zobecněný x-dominátor

je uzel, na který ukazuje alespoň jedna komplementovaná hrana a jedna normální hrana (je jedno jestli hrana then nebo else).

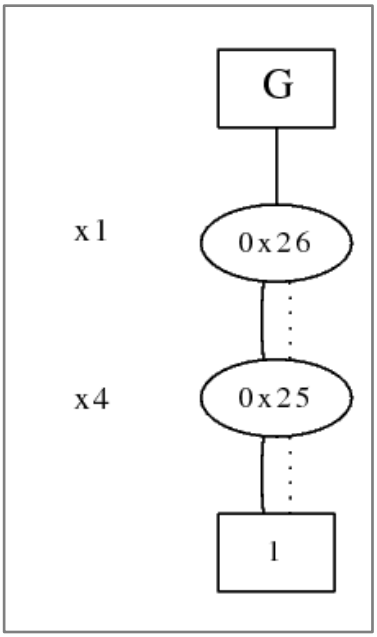
Jestliže je možné nalézt zobecněný x-dominátor, tak funkce, která má kořen v tomto zobecněném x-dominátoru, je označena jako G. Jestliže provedeme G $\overline{\oplus}$ F, dostaneme funkci H. Výsledná funkce F je pak F = G $\overline{\oplus}$ H (důkaz v [1]).

Na následujících třech obrázcích bude ukázána XOR dekompozice pro funkci F. Na Obrázku 21 je zobrazena funkce F pro XOR dekompozici.



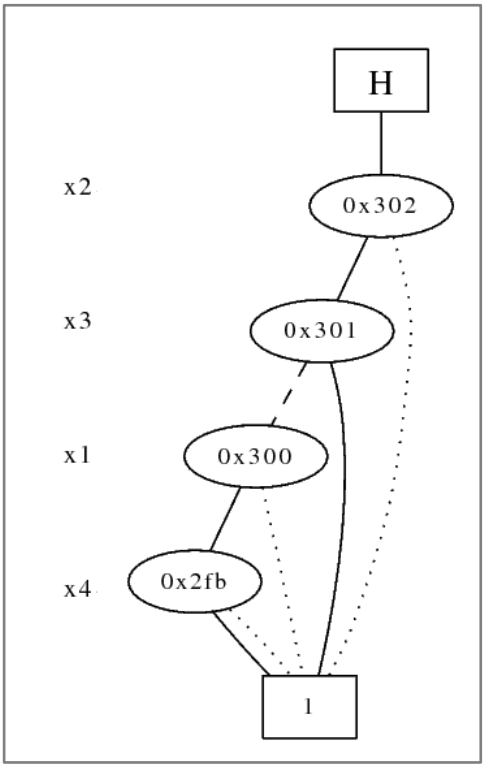
Obrázek 21: Funkce F pro XOR dekompozici

Jak je vidět z Obrázku 21, tak tato funkce má dva zobecněné x-dominátory (oranžový a modrý uzel). XOR dekompozici provedeme podle oranžového uzlu. Funkce G je funkce, která má kořen v tomto zobecněném x-dominátoru. Na Obrázku 22 je zobrazena tato funkce.



Obrázek 22: Funkce G pro XOR dekompozici

Funkce H pro XOR dekompozici vznikne tak, že se spočítá H = G $\overline{\oplus}$ F. Na Obrázku 23 je zobrazena tato funkce.



Obrázek 23: Funkce H pro XOR dekompozici

4.2.2 Algebraická

V algebraické XOR dekompozici se hledá x-dominátor. X-dominátor je uzel, přes který vedou všechny cesty Π. Pro nalezený x-dominátor může být funkce F rozložena na $F = G \overline{\oplus} H$, kde G je BDD s kořenem v x-dominátoru a H je získáno z F tak, že normální hrany (nenegované) vedoucí do x-dominátoru jsou přesměrovány do 1-terminálu a negované hrany jsou přesměrované do 0-terminálu.

Funkce, která nemá algebraickou XOR dekompozici, může mít mnoho booleovských dekompozic. Z Obrázku 21 je vidět, že funkce nemá žádný x-dominátor (nemá algebraickou dekompozici) a má 2 zobecněné x-dominátory (má booleovskou XOR dekompozici).

5 Popis implementace

5.1 Struktura souboru ve formátu blif

Formát blif (Berkeley Logic Interchange Format) byl vyvinut z potřeby popsat hierarchické obvody textovou formou. Tento formát byl vyvinut na Kalifornské univerzitě v Berkeley.

Zjednodušená struktura blif formátu, která je použita v této diplomové práci, je následující:

.model <decl-model-name>

.inputs <decl-input-list>

.outputs <decl-output-list>

<.names directives>

.

.

.

.end

- **decl-model-name** je jméno modelu. Toto jméno je nepovinné.
- **decl-input-list** je list jmen vstupních proměnných, které jsou oddělené mezerou a zakončené koncem řádky. Víceřádková definice vstupních proměnných je povolena, řádka musí být zakončena dvěma zpětnými lomítky a list vstupních proměnných pokračovat na další řádce.
- **decl-output-list** je list jmen výstupních proměnných, které jsou oddělené mezerou a zakončené koncem řádky. Víceřádková definice výstupních proměnných je povolena, řádka musí být zakončena dvěma zpětnými lomítky a list výstupních proměnných pokračovat na další řádce.
- **.names direktiva** je ohodnocení výstupu pro jednu konkrétní funkci daného modelu. Za .names se nachází jména vstupních proměnných a poslední jméno značí výstupní proměnnou (může se jednat o výstupní

proměnnou modelu nebo tzv. vnitřní proměnnou – proměnná, která pak slouží k definici výstupní proměnné). Na dalších řádcích je pak ohodnocení pro danou direktivu, kde výstup je oddělen mezerou.

Následuje jednoduchá ukázka funkce F = x1 * x2:

```
.model ukazka
.inputs x1 x2
.outputs F
.names x1 x2 Inner
11 1
.names Inner F
1 1
.end
```

Model má jméno ukazka, vyskytují se v něm dvě vstupní proměnné x1 a x2 a jedna výstupní proměnná F. Dále se zde vyskytují dvě direktivy .names, první definuje vnitřní proměnnou Inner a druhá definuje výstupní proměnnou F. Vnitřní proměnná Inner zde není vůbec potřebná, je tu pouze z důvodu ukázky. Na pořadí jednotlivých .names direktiv vůbec nezáleží. Mohla by klidně být nejdříve definována výstupní proměnná F z vnitřní proměnné Inner a až poté definovat Inner z vstupních proměnných x1 a x2.

5.2 Práce se soubory

Program umí načítat soubory ve formátu blif (viz. předchozí kapitola). Načtený model umožňuje zapsat do souboru blif a dot[7] (tento formát umožňuje zobrazit funkci pomocí grafu).

[7] http://www.graphviz.org/doc/info/lang.html

5.2.1 Načtení souboru

Ze souboru se nejdříve načte jméno modelu (jestliže existuje). Poté jsou načteny všechny vstupní a výstupní proměnné. Následuje cyklus, který prochází zbytek souboru do té doby, než najde .end, nebo se vyskytne nějaká chyba v zadání. Každá direktiva .names je uložena do speciální třídy Entry. Třída Entry je definována následovně:

```
class Entry {
private:
        string output;
        vector<string> inputs;
        vector<string> settings;
        bool singularity;
        bool canInit;

public:
        Entry();
        Entry(string outputName);
        virtual ~Entry();
        void addInput(string input);
        void addSetting(string setting);
        string getOutput();
        void toString();
        bool canInitialize(set<string>& initializedNodes);
        int lineCount();
        string getLine(int number);
        string getInput(int number);
        bool getSingularity();
};
```

Tato třída obsahuje několik polí:

- output – jméno výstupní proměnné pro daný záznam
- inputs – jména vstupních proměnných pro daný záznam
- settings – jednotlivá ohodnocení výstupní funkce
- singularity – říká, zda daná funkce má výstup 1 nebo 0
- canInit – říká, zda daný záznam je možné inicializovat
- public metody - metody, slouží pro práci s private poli

Před uložením všech načtených tříd Entry proběhne jejich uspořádání tak, aby všechny záznamy pro danou výstupní (vnitřní) proměnou byly u sebe (je to z důvodu ušetření času při stavbě BDD, viz. Kapitola 5.3).

5.2.2 Uložení do souboru

V programu je možné uložit postavené BDD do souboru formátu blif a dot.

K ukládání do souboru formátu blif slouží metoda writeBliff (DdManager* manager, fileStructureNew structure, char* fileName). Tato metoda uloží do listu vstupních proměnných i ty proměnné, které nebyly v BDD použity. Tato metoda přijímá parametry:

- manager – ukazatel na DdManager, ve kterém je BDD postavené
- fileStructureNew - struktura, ve které jsou uložené záznamy o modelu a ukazatele na výstupní funkce
- filename – jméno souboru, do kterého se má BDD uložit

K uložení do souboru formátu dot slouží metoda writeDot (DdManager *manager, fileStructureNew structure, char* fileName), která přijímá stejné parametry jako metoda writeBliff.

5.3 Vytvoření BDD

Před tvorbou BDD se nejdříve inicializují všechny vstupní proměnné a výstupní proměnné. V následujícím pseudokódu je ukázáno inicializování záznamů typu Entry.

```
unsigned int entriesFinished = 0;
for (unsigned int i = 0; i < pocet promenych; i++) {
 if (entriesFinished == pocet zaznamu entry) {
  //vsechno inicializovano
  break;
 }
 for (unsigned int ii = 0; ii < pocet zaznamu; ii++) {
  if (je mozne inicializovat zaznam ii && není inicializovan zaznam ii) {
   inicializuj zaznam ii();
   entriesFinished++;
  } else {
   continue;
  }
 }
}
```

Vnější for cyklus se projíždí maximálně tolikrát, kolik je počet proměnných. V nejlepším případě se projede jen jednou, protože všechny záznamy typu Entry se inicializují ve vnitřním for cyklu (to je případ, kdy vnitřní proměnné jsou definované dříve, než jsou použity). V nejhorším případě se vnější for cyklus provede tolikrát, kolik činí počet proměnných, protože ve vnitřním for cyklu se při každém průchodu inicializuje jen jedna proměnná (to je případ, kdy vnitřní proměnné jsou použity dříve, než jsou definované).

Po dokončení těchto dvou for cyklů se provede kontrola, zda byly všechny záznamy typu Entry a všechny výstupní proměnné inicializovány. Po dokončení této kontroly je postavené BDD pro daný model.

5.4 Dekompozice

Před samotnou dekompozicí se provede kopírování výstupních funkcí do vlastních managerů. K udržení informací o dekompozici slouží třída Decomp:

```
#define AND     1
#define OR      2
#define XNOR    3
```

```
class Decomp {
public:
  bool list;

  DdNode *function;
  DdManager *manag;

  int oper;
  Decomp *L;
  Decomp *R;
};
```

Tato třída obsahuje několik polí:

- **list** – označuje, zda tato třída je list (tzn. nemá potomky ani operaci a má ukazatel na funkci)
- **function** – ukazatel na funkci typu DdNode*
- **manag** – ukazatel na manager typu DdManager*
- **oper** – operace, která spojuje potomky dekompozice
- **L** – ukazatel na jednoho potomka dekompozice
- **R** – ukazatel na druhého potomka dekompozice

Po zkopírování výstupních funkcí vznikne list instancí Decomp. Počet těchto instancí je roven počtu výstupních funkcí. Každá takto vytvořená instance má v sobě ukazatel na vlastního managera, ukazatel na výstupní funkci a příznak list je vyplněn na true. Zbývající pole jsou NULL.

5.4.1 AND

Nejdříve je nutné nastavit množinu uzlů BDD, které se nacházejí nad řezem. To se dělá metodou initCut(set<int> cut2), která přijímá jako parametr set indexů uzlů nad řezem.

Poté se rekurzivně prochází funkce a staví se nová funkce D (booleovský dělitel) s novým managerem odspodu s tím, že uzly, které leží pod řezem, jsou nahrazeny 1-terminálem. Funkce Q (kvocient) je získána z původní funkce tak, že hrany nad řezem jdoucí do 0-terminálu se přesměrují do don't care DC (hrana se přesměruje na druhého potomka).

Takto vzniknou dvě nové funkce – Q z původní funkce a D jako nová funkce. Poté se vytvoří dvě nové instance třídy Decomp pro tyto funkce a v původní instanci třídy Decomp se uloží odkazy na tyto dvě nové instance a do pole function se uloží AND operace.

5.4.2 OR

OR dekompozice je téměř stejná jako AND dekompozice. Také se musí nejdříve definovat řez. Poté se rekurzivně prochází funkce a staví se nová funkce G a z původní funkce se vytvoří funkce H. Dále se vytvoří nové instance třídy Decomp pro tyto funkce a do původní instance se uloží odkazy na tyto instance a do pole function se uloží OR operace.

5.4.3 XOR booleovská

Nejprve je nutné nastavit uzel, podle kterého se provádí XOR dekompozice (zobecněný x-dominátor). To se dělá metodou initGNX(DdNode *gnx2), která přijímá ukazatel na tento uzel. Funkce G vznikne tak, že tento uzel se stane kořenem pro funkci G. Následuje zkopírování této funkce do vlastního manageru.

Dále se provede $G \overline{\oplus} F$ a tím se dostane funkce H. Pro funkce G a H se vytvoří dvě nové instance Decomp a v původní instanci třídy Decomp se uloží odkazy na tyto dvě nové instance a do pole function se uloží XOR operace.

5.5 Faktorizační strom

Struktura provádění jednotlivých dekompozic je uložená ve faktorizačním stromě. Faktorizační strom tvoří instance třídy Decomp.

5.5.1 Vytvoření

Na začátku je pouze jedna instance Decomp pro jednu výstupní proměnnou. Jestliže je provedena dekompozice, tak z původní instance vzniknou dvě nové a odkazy na tyto instance se uloží do původní instance společně s operací, která tyto dvě instance spojuje.

Stejným způsobem se pokračuje s potomky původní instance až do té doby, dokud je možné provádět dekompozice. Výsledný faktorizační strom je pak uložen ve struktuře potomků původní instance třídy Decomp pro výstupní funkci.

5.5.2 Uložení

Nejprve je do souboru zapsáno jméno modelu, jména vstupních proměnných a jména výstupních proměnných. Poté se prochází celá struktura faktorizačního stromu a jednotlivé mezivýsledky se ukládají následovně:

1. jestliže se jedná o operaci, která již nešla dále dekomponovat, tak se tato operace uloží
2. jestliže se jedná o dekompozici, tak se do souboru uloží daná dekompozice a na potomky se dále volá stejná metoda na uložení faktorizačního stromu

5.6 Problémy v implementační části

5.6.1 Stavba BDD

První problémy nastaly hned při načítání souboru a tvorbě binárního rozhodovacího diagramu. Nejprve byly načítání a tvorba prováděny současně. Postupně

se procházel soubor a inicializovalo se BDD podle načteného řádku. Pro jednoduché funkce toto fungovalo správně.

Pro funkce, které měly ve vstupním souboru formátu blif definovanou vnitřní proměnnou, tento postup nefungoval. Musela být přidána mapa vnitřních proměnných, ve které byly tyto proměnné definovány. Když byla pro inicializaci jiné proměnné potřeba vnitřní proměnná, tak se použila proměnná z této mapy. Pro funkce definované s vnitřními proměnnými toto fungovalo správně.

Další problém přineslo načítání složitějších funkcí ze souborů formátu blif. Některé proměnné (vnitřní i výstupní) byly použity dříve, než tyto proměnné byly definované. Musela se opustit myšlenka společného načítání a tvoření BDD a tyto metody se musely rozdělit.

Jak již bylo psáno, nejdříve byla načtena celá struktura vstupního souboru do tříd Entry a až poté bylo stavěno BDD. Toto umožnilo nejdříve definovat proměnné a poté je použít při stavbě BDD.

5.6.2 Řez

Jak již bylo psáno v kapitole 4.1.4, v této diplomové práci jsem se zaměřil pouze na horizontální řezy. To znamená, že povedu-li řez, tak index vstupní proměnné se nachází buď nad řezem, nebo pod řezem. V tomto případě nemůže nastat situace, kdy by index jedné proměnné byl nad řezem i pod řezem současně.

Před volbou dekompozice se prochází celá struktura daného BDD a zjišťují se údaje k volbě dekompozice a řezu (popsáno v kapitole 5.6.4). Jestliže je vybrána AND nebo OR dekompozice, je potřeba definovat indexy vstupních proměnných, které se nacházejí nad řezem. Tento údaj je potřebný pro provádění vybrané dekompozice.

Jelikož se volá metoda na změnu pořadí proměnných po každé dekompozici (z důvodu minimalizace nově vzniklé funkce po dekompozici), tak se pořadí proměnných mění, a proto není možné vybrat ty indexy proměnných, které jsou menší než index proměnné řezu. Před první dekompozicí by to ještě bylo možné, protože indexy

proměnných jsou seřazeny od nuly do n, kde n je počet vstupních proměnných. Ale hned po první dekompozici již toto pořadí nemusí platit.

Naštěstí existuje metoda int Cudd_ReadInvPerm(DdManager * dd, int i), která jako parametry přijímá ukazatel na manager a index (úroveň BDD). Tato metoda vrací index vstupní proměnné, která se nachází na dané úrovni. Pak stačí jednoduchý for cyklus na naplnění setu indexy proměnných, které se nacházejí nad řezem:

```
for (int i = 0; i < Cudd_ReadSize(decomp -> manag); i++) {
  cut2.insert(Cudd_ReadInvPerm(decomp -> manag, i));
  if (Cudd_ReadInvPerm(decomp -> manag, i) == cutLevel) {
    break;
  }
}
```

Prochází se for cyklem od nuly do počtu proměnných daného manageru a vkládají se do pole indexy proměnných, které se nacházejí nad řezem (úroveň nula je kořen). Jestliže se narazí na index proměnné řezu, tak tento for cyklus končí. Po tomto for cyklu je naplněn set cut2 indexy proměnných, které se nachází nad řezem. Díky tomuto definování řezu je možné používat změnu pořadí proměnných v daném BDD, aniž by se musely uchovávat nějaké informace o změně pořadí proměnných.

5.6.3 Kopírování funkcí

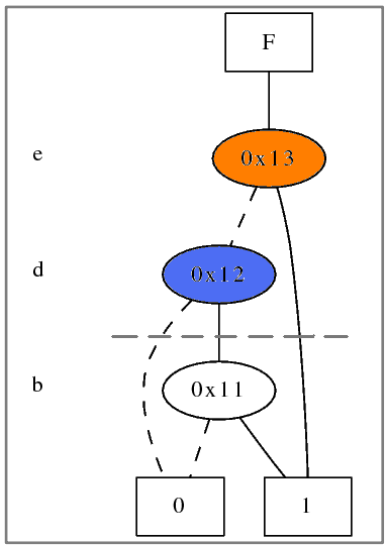
5.6.3.1 AND a OR dekompozice

V AND dekompozici (platí i pro OR dekompozici) obě funkce (booleovský dělitel a kvocient) vycházejí ze stejné původní funkce F a změní se aplikací pravidel pro tvorbu AND dekompozice.

Nešlo však udělat to, že původní funkce F by se nakopírovala pomocí volání metody DdNode* Cudd_bddAnd(DdManager * dd, DdNode * f, DdNode * g), která by jako argumenty přijala ukazatel na manager funkce F, ukazatel na kořen funkce F a ukazatel na 1-terminál. Vzniklý uzel je sice funkce F, ale je to ukazatel na stejný uzel – kořen funkce F. Následnou úpravou BDD pro danou funkci (ať už tvorbou booleovského

dělitele nebo kvocientu) se změnila i druhá funkce (to je pochopitelné, neboť obě funkce mají ukazatele na stejný kořen – stejné BDD), což bylo nežádoucí.

Bylo nutné zajistit, aby obě BDD pro nově vzniklé funkce po dekompozici byly nezávislé. Před provedením dekompozice byl inicializován nový manager. Při tvorbě kvocientu je funkce procházena rekurzivně a hledají se všechny listové hrany nad řezem, které vedou do 0-terminálu. Tyto hrany se hledají pouze pro uzly nad řezem. Jelikož pro tvorbu booleovského dělitele není potřeba znát strukturu BDD pod řezem, je možné při hledání hran kopírovat funkci do nově vytvořeného manageru.



Obrázek 24: BDD funkce F = e + db

Na obrázku 24 je zobrazen BDD pro funkci F = e + db a daný řez. Při kopírování funkce pro booleovský dělitel se začíná odspoda a první uzel, který vznikne, je pro proměnnou d (na obrázku modře). Vznikne pomocí ITE operace, která vypočte f*g + $\bar{f}$*h. V tomto případě se zavolá metoda Cudd_bddIte(DdManager * dd, DdNode * f, DdNode * g, DdNode * h), která jako parametry přijímá:

- dd je ukazatel na nový manager
- f je ukazatel na uzel proměnné, získá se zavoláním metody Cudd_bddIthVar(DdManager * dd, int i) s ukazatelem na manager a příslušným indexem (v tomto případě 1, což je index proměnné d, v původním BDD)

- g je ukazatel na uzel hrany then (v tomto případě je to 1-terminál, protože uzel proměnné b je pod řezem, a proto je v AND dekompozici přesměrován na 1-terminál)
- h je ukazatel na uzel hrany else (v tomto případě 0-terminál)

Takto vznikne uzel d v novém manageru a může se použít pro tvorbu uzlu e (na obrázku oranžově) zavoláním operace ITE, kde f je ukazatel na uzel proměnné s indexem 0, g je 1-terminál a h je ukazatel na uzel d vzniklý v minulém kroku. Tím se zkopírovala původní funkce do nového manageru a zároveň se vytvořil booleovský dělitel původní funkce.

5.6.3.2 Booleovská XOR dekompozice

V XOR dekompozici vznikají dvě nové funkce G a H. G je funkce, jejíž kořen je ukazatel na uzel (zobecněný x-dominátor) v původním BDD. Jestliže se provede $G \oplus \overline{F}$, tak se funkce G zruší. Proto je nutné tuto funkci zkopírovat do nového manageru.

Postup kopírování je téměř stejný jako v případě AND (OR) dekompozice. Před dekompozicí se vytvoří nový manager. Rekurzivně se prochází funkce od uzlu, na který ukazuje funkce G (jedná se o zobecněný x-dominátor), a v novém manageru se vytváří kopie odspoda voláním ITE operací.

Tímto vznikne funkce G v novém manageru a tato funkce je nezávislá na původní funkci.

5.6.4 Řízení dekompozice

K dekompozici logické funkce je potřeba iterativní dekompozice jejího BDD, kdy z velkého BDD vznikají menší. O iterativní dekompozici se stará metoda doDecomposition v Manager.cpp, která přijme množinu instancí Decomp (pro každou výstupní proměnnou je jedna instance). Postupně provádí dekompozice nad touto množinou.

Nejdříve je vyjmuta jedna instance Decomp a na ní provedena analýza k zjištění údajů k volbě dekompozice a řezu. Je zjištěn počet listových hran pro jednotlivé proměnné a počet zobecněných x-dominátorů.

Jestliže existuje listová hrana pro nějakou proměnnou, volá se AND (OR) dekompozice, je zvolena dekompozice podle toho, kterých listových hran je pro danou proměnnou více. Jestliže je více hran vedoucích do 0-terminálu pro jednu proměnnou, tak se volí AND dekompozice, jestliže je více hran vedoucích do 1-terminálu, tak se volí OR dekompozice. Jako řez se volí index proměnné s maximem těchto listových hran.

Jestliže neexistuje žádná listová hrana, volá se booleovská XOR dekompozice a jako zobecněný x-dominátor se zvolí první nalezený.

Po provedení dekompozice na instanci Decomp vznikají dvě nové instance Decomp, které jsou vloženy do množiny všech instancí Decomp.

Jestliže již nelze provézt žádnou dekompozici, nic nového se do množiny instancí Decomp nevkládá. Jestliže je množina instancí Decomp prázdná, dekompozice končí.

6 Testování a srovnání

Testování nástroje na dekompozici logických funkcí bylo provedeno pouze na čtyřech logických funkcích. V průběhu finálního testování nástroje bylo totiž zjištěno, že nástroj bez dont´t care DC minimalizace (kapitola 4.1.3) a vhodnou volbou řezu (kapitola 4.1.4) je nepoužitelný na větší logické funkce. Z důvodů časové náročnosti implementace (hlavně problémy s knihovnou CUDD) nebylo možné tyto heuristiky naprogramovat (DC minimalizace je sama o sobě velice náročná, rozsahem by se spíše jednalo o další diplomovou práci).

Další problém, který se objevil při testování, je špatný návrh implementace provádění jednotlivých dekompozic. Při každé dekompozici jsou vytvářeny dvě nové funkce, jedna zůstává v původním DdManagerovi a druhá se kopíruje do nového DdManagera. DdManager zabírá v paměti necelý jeden megabyte. Pro malé funkce, na kterých byl zpočátku nástroj testován, toto vůbec nevadilo. Ale pro větší funkce, kde se vytvoří mnoho managerů, je toto kritické (2000 managerů zabere v paměti skoro 2GB) a stává se, že paměť dojde.

Implementací heuristik na minimalizaci DC a výběrem vhodného řezu, by se počet managerů zmenšil, čím by se docílilo funkčnosti i pro velké logické funkce, ale úplně by tento problém odstranit nešel. Pro odstranění této paměťové náročnosti by musel být zaveden jeden hlavní manager a všechny funkce by se musely držet v něm. To by ale znamenalo kompletní předělání implementace řízení dekompozice a provádění jednotlivých dílčích dekompozic.

Jak již bylo psáno, byly otestovány čtyři logické funkce (tyto funkce lze nalézt na přiloženém CD). První test spočíval v ověření, zda funkce načtená ze vstupního souboru a výstupní funkce (dekomponovaná funkce) si odpovídají. Tento test byl proveden pomocí nástroje ABC[8]. Všechny čtyři funkce prošly.

Další test probíhal jako srovnání naprogramovaného nástroje pro dekompozici logických funkcí a programem BDS[9], jehož autorem je Congguang (Anda) Yang. Pomocí

[8] A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/~alanmi/abc/
[9] BDD Based Logic Synthesis System

nástroje SIS[10] byl zjištěn údaj o počtu uzlů a počtu literálů (uzly značí .names výrazy a literály značí z kolika jedniček a nul jsou složeny, platí že čím menší počet literálů, tím lepší dekompozice).

Logická funkce	Počet uzlů	Počet literalů
func1	1	61
func1.final	7	16
func1.dekomp	7	14
func2	6	12
func2.final	6	12
func2.dekomp	16	34
func3	102	194
func3.final	49	144
func3.dekomp	149	382
func4	58	351
func4.final	244	472
func4.dekomp	1689	4250

Tabulka 1: Test logických funkcí

V tabulce 1 jsou funkce dekomponované pomocí implementovaného nástroje označené „dekomp“ a funkce dekomponované pomocí nástroje BDS jsou označeny „final“. Z tabulky jsou patrné chybějící heuristiky pro dekompozice větších logických funkcí. V případě func2 a func3 je třikrát větší počet literálů a v případě func4 dokonce téměř desetkrát větší. Jedině v případě func1 je počet literálů nepatrně menší.

[10] Synthesis of both synchronous and asynchronous sequential circuits, http://embedded.eecs.berkeley.edu/pubs/downloads/sis/index.htm

7 Závěr

7.1 Další možné rozšíření nástroje

V nástroji pro dekompozici logických funkcí je použita velmi jednoduchá don´t care minimalizace (kapitola 4.1.3). V dnešní době existuje několik heuristik, které tento problém řeší. Použití některé z těchto heuristik by vedlo k lepší minimalizaci logických funkcí.

Dále se v nástroji používají pouze horizontální řezy. Bylo by vhodné prozkoumat možnost použití i nehorizontálních řezů. Tyto nehorizontální řezy by tak dále mohly vést k lepším výsledkům dekompozice.

Dalšího vylepšení nástroje by se dosáhlo hledáním společných faktorizačních podstromů pro více výstupních funkcí (pokud mají dva výstupy společný faktorizační podstrom, mohl by být jeden odstraněn a mohl by být použit jen společný faktorizační podstrom).

V každé iteraci dekompozice je vybrán pouze jediný typ dekompozice pomocí jednoduché heuristiky. Dalšího zlepšení by se dosáhlo vylepšením heuristiky pro volbu dekompozice a následnou volbu řezu nebo zobecněného x-dominátoru. Jinou možností je provedení všech dekompozic (pro všechny řezy nebo zobecněné x-dominátory) a vybrat tu, která má za následek nejmenší počet uzlů ve výsledných dvou BDD.

Dále by bylo potřeba přepracovat nástroj na použití pouze jediného managera, aby nedocházelo k přetékání paměti, při velkých počtech managerů.

7.2 Shrnutí diplomové práce

Díky implementaci nástroje pro dekompozici logických funkcí jsem se podrobně seznámil s knihovnou CUDD, která je určená pro práci s binárními rozhodovacími diagramy – její strukturou, použitím a funkcemi. Dále jsem se seznámil s problematikou dekompozice logických funkcí založené na binárních rozhodovacích diagramech.

Podařil se vytvořit nástroj pro dekompozici logických funkcí postavený na knihovně CUDD. Tento nástroj umí načítat logické funkce ze souboru ve formátu blif a načtené funkce umí zapisovat do formátu blif a dot.

Dále nástroj umožňuje provést dekompozici jednoduchých logických funkcí pomocí iterativního volání dekompozice AND, OR nebo XOR a výsledky jednotlivých dekompozic ukládat do faktorizačního stromu. Tento faktorizační strom je poté uložen do souboru ve formátu blif. Pro dekompozici velkých funkcí by bylo potřeba předělat nástroj na použití pouze jediného managera.

Literatura

[1] YANG, Congguang, CIESIELSKI, Maciej. BDS: A BDD-Based Logic Optimization System. In IEEE Trans. Computers. 21th edition. [s.l.] : [s.n.], 2002. s. 866-876.

[2] Wikipedia : The Free Encyklopedia [online]. 2009 [cit. 2009-12-25]. Dostupný z WWW: <http://www.wikipedia.org/>.

[3] KOLOŠ, Ondřej. Port programového balíku CUDD pod platformu Windows. [s.l.], 2006. 69 s. Vedoucí bakalářské práce Petr Fišer.

[4] AKERS, S. B. Functional testing with binary decision diagrams. In Eighth Annual Conf. Fault-Tolerant Computing. [s.l.] : [s.n.], 1978. s. 75-82.

[5] BRYANT, R. Graph-based algorithms for boolean function manipulation. In IEEE Trans. Computers. 35th edition. [s.l.] : [s.n.], 1986. s. 677-691.

[6] SOMENZI, Fabio. CUDD: CU Decision Diagram Packag [online]. 2009 [cit. 2009-12-25]. Dostupný z WWW: <http://vlsi.colorado.edu/~fabio/CUDD/>.

[7] SAUERHOFF, M., WEGENER, I. On the complexity of minimizing the OBDD size for incompletely specified functions. In IEEE Trans. Computers. 15th edition. [s.l.] : [s.n.], 1996. s. 1435-1437.

[8] OLIVEIRA, A. L., et al. Exact minimization of binary decision diagrams using implicit techniques. In IEEE Trans. Computers. 47th edition. [s.l.] : [s.n.], 1998. s. 1282-1296.

[9] Yang, C., Ciesielski, M., “BDS: BDD-Based logic optimization system,” Dept. Electrical and Computer Engineering, Univ. Massachusetts Amherst, 2000.

[10] C. Y. Lee. "Representation of Switching Circuits by Binary-Decision Programs". Bell Systems Technical Journal, 38:985–999, 1959.

[11] Sheldon B. Akers. Binary Decision Diagrams, IEEE Transactions on Computers, C-27(6):509–516, June 1978.

[12] Raymond T. Boute, "The Binary Decision Machine as a programmable controller". EUROMICRO Newsletter, Vol. 1(2):16–22, January 1976.

Příloha A. Základní datové struktury knihovny CUDD

a. Uzly (DdNode)

BDD, ADD i ZDD jsou tvořeny z DdNode (zkráceně uzel). DdNode je struktura s několika poli:

```
struct DdNode {
   DdHalfWord index;
   DdHalfWord ref;
   DdNode *next;
   union {
         CUDD_VALUE_TYPE value;
         DdChildren kids;
   } type;
};

typedef struct DdChildren {
   struct DdNode *T;
   struct DdNode *E;
} DdChildren;
```

Pole index drží jméno proměnné pro tento uzel. Index proměnné je permanentní atribut, který vyjadřuje pořadí, ve kterém byl uzel vytvořen. Index 0 vyjadřuje uzel proměnné, která byla vytvořena jako první. Na 32-bitovém počítači je maximum proměnných, které je možné držet v paměti, roven unsigned short integer mínus 1. Poslední (největší) index je vyhrazen pro konstantní uzel. Na 64-bitovém počítači je tato velikost rovna unsigned integer mínus 1.

Proměnná ref udržuje informaci o počtu referencí na tento uzel. Je to informace pro garbage collector, jestli tento uzel může odstranit či nikoliv.

Next je proměnná typu DdNode a ukazuje na další uzel v hashovací tabulce – unique table.

Type je výčtový typ, který nese informaci pro uzly. Jestliže je daný uzel neterminál, tak nese informaci o jeho následovnících (viz. struktura DdChildren, kde jsou dva

ukazatele na DdNode – na uzel then a na uzel else). Jestliže je daný uzel terminál, tak tato struktura nese informaci o jeho hodnotě.

b. Manager (DdManager)

Všechny uzly v BDD, ADD a ZDD diagramech jsou drženy ve speciálních hashovacích tabulkách zvaných unique tables. BDD a ADD sdílejí společnou unique table a ZDD má unique table vlastní. Hlavní účel unique tables je zaručit, aby každý uzel byl unikátní, to znamená, aby neexistoval jiný uzel pro danou proměnnou se stejnými potomky.

Unique table a další pomocné struktury tvoří DdManager (zkráceně manager). Aplikace, které používají pouze exportované funkce CUDDu, se nemusí zajímat o detaily manageru. Jediné, co musí aplikace udělat, je inicializace manageru zavoláním příslušné metody. Následně se musí předávat ukazatel na tento manager všem funkcím, které pracují s příslušným diagramem.

V knihovně CUDD nejsou žádné globální proměnné (až na počitadla určená pro statistiky). Proto je možné inicializovat mnoho managerů v jedné aplikaci. Ukazatel na manager říká funkci, nad kterými daty má pracovat.

Pro využívání funkcí z knihovny CUDD je nejdříve potřeba inicializovat manager voláním Cudd_init. Tato funkce přijímá 5 parametrů:

```
DdManager *Cudd_Init(
  unsigned int numVars,
  unsigned int numVarsZ,
  unsigned int numSlots,
  unsigned int cacheSize,
  unsigned long maxMemory
);
```

- **numVars:** počáteční počet proměnných pro BDD a ADD diagramy. Jestliže je počet proměnných předem určen pro aplikaci, tak je efektivnější vytvořit

manager s určeným počtem proměnných. Jestliže tento počet není dopředu známý, tak se nastavuje 0 nebo nějaká jiná spodní hranice. Nastavení menšího počtu není špatné, ale je to neefektivní.

- **numVarsZ:** stejné jako numVars, akorát pro ZDD proměnné.
- **numSlots:** určuje počáteční velikosti všech podtabulek v unique table. Pro každou proměnnou je jedna podtabulka. Velikost podtabulky se dynamicky mění s ohledem na počet uzlů. Defaultně se nastavuje parametr CUDD_UNIQUE_SLOTS.
- **cacheSize:** počáteční velikost (počet záznamů) cache. Defaultní parametr je CUDD_CACHE_SLOTS.
- **maxMemory:** maximální velikost manageru v paměti (v bytech). Knihovna CUDD používá tento parametr pro:
    - určení maximální velikosti cache (bez ohledu na „hit rate“ nebo velikost unique table)
    - velikost unique table, při které se provede garbage collection

  Jestliže je tento parametr nastaven na nulu, tak se knihovna CUDD sama pokusí určit tento parametr na základě volné paměti.

Typické volání inicializace manageru může vypadat následovně:

```
Cudd_Init(0, 0, CUDD_UNIQUE_SLOTS, CUDD_CACHE_SLOTS, 0);
```

K odstranění veškeré paměti spojené s managerem musí aplikace zavolat metodu Cudd_Quit.

c. Cache

K efektivní manipulaci s rozhodovacími diagramy je zapotřebí tabulka k uložení vypočtených výsledků. Taková tabulka se v CUDDu označuje cache. Na začátku je tato cache malá a postupně se zvětšuje až do té doby, dokud už její zvětšení nepřinese žádný

užitek nebo již dosáhla maximální kapacity. Uživatel má možnost nastavit výchozí a maximální velikost cache.

Příliš malá cache způsobí časté přepisování užitečných výsledků. Příliš velká cache způsobí značné zatížení, jelikož při spuštění garbage collection se musí prozkoumat celá cache. Optimální parametr velikosti cache závisí na dané aplikaci. Defaultní parametr je vhodný pro většinu aplikací.

Cache je používána většinou v rekurzivních funkcích v knihovně CUDD. Může být také použita ve funkcích dodaných uživatelem.

Příloha B.Obsah přiloženého CD

- Diplomova_prace
  - Zde je umístěna diplomová práce ve formátu pdf.
- Binary
  - Zde se nachází spustitelná verze programu.
- Src
  - Tato složka obsahuje všechny zdrojové kódy.
- Funkce
  - Logické funkce použité v testování