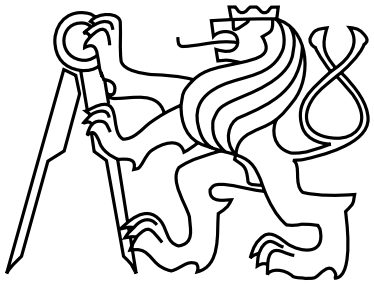
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Vizulizace dvouúrovňové minimalizace logických funkcí

Bc. Jakub Zelenka

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika, strukturovaný, Navazující magisterský

Obor: Výpočetní technika

13. května 2010

Poděkování

Rád bych poděkoval panu Petru Fišerovi za cenné rady při tvorbě aplikace a psaní textové části práce.

Také děkuji všem testujícím uživatelům za to, že si našli čas na vyzkoušení mé aplikace a na následné vyplnění dotazníku. Jsou jimi pánové Václav Hradec, Martin Poliak, Michal Šoka, Milan Kříž, Vojtěch Sedláček a Martin Sedláček.

Prohlášení

Prohlašuji, že jsem 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 13. 5. 2010 .....

Abstract

This thesis describes the methods for the minimization of logic function and the tools for their visualization, specifically the Quine–McCluskey algorithm and Espresso algorithms. The Karnaugh map and Boolean n-cube are also used as visualization tools.

In the second part of this thesis the implementation of mentioned methods in the Bmin application is described. The application architecture, its implementation, the specifics of its algorithms and the management of the application are here described in detail. The thesis is concluded with a usability test.

Abstrakt

Tato diplomová práce popisuje metody pro minimalizaci logických funkcí a prostředky pro její vizualizaci. Konkrétně se jedná o algoritmus Quine–McCluskey a sadu algoritmů Espresso. Jako vizualizační prostředky jsou popsány Karnaughova mapa a Booleovská n-krychle.

Ve druhé části práce je uvedena analýza a způsob implementace výše zmíněných metod a vizualizačních prostředků v aplikaci Bmin, která tvoří implementační část práce. Je zde podrobně rozebrána architektura, implementační specifika algoritmů a způsob ovládání. Na závěr je vyhodnocen test použitelnosti aplikace.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Minimalizace logických funkcí je součástí výuky každého předmětu, který se zabývá základy syntézy logických obvodů. Tyto předměty jsou povinné zejména na elektro-průmyslových středních a vysokých školách.

Mezi probíranou látkou je velmi často provádění minimalizace v Karnaughově mapě. Občas bývá ještě uvedena Booleovská n-krychle. Ve vyšších kurzech se někdy navíc učí fungování algoritmu Quine–McCluskey. Přestože nejsou tyto metody v praxi používány, velmi dobře se na nich minimalizace ukazuje a také jsou poměrně snadno pochopitelné.

I když je výukových materiálů hodně, je přeci jenom jednodušší a také rychlejší použít počítačový program, který vše názorně zobrazí. Bohužel, do dnešní doby neexistuje takových aplikací mnoho, a ty co existují, jsou velmi často zpoplatněny, nebo nejsou příliš kvalitní. Aplikace, která by dokázala přehledně zobrazit výše zmíněné metody a která by byla zdarma, by mohla najít široké uplatnění.

1.1 Cíl

Hlavním cílem této práce je vytvoření aplikace pro vizualizaci minimalizace logických funkcí. To konkrétně znamená vytvoření aplikace, která umí zminimalizovat zadanou logickou funkci a následně zobrazit její výslednou minimální formu v Karnaughově mapě nebo v Booleovské n-krychli. Zároveň také zobrazí průběh algoritmů Quine–McCluskey a Espresso. Součástí je také popsání jednotlivých metod a algoritmů. Nechybí ani popis způsobu implementace a dokumentace aplikace.

1.2 Postupný vývoj

Poprvé jsem se setkal s logickými obvody v roce 2007 v předmětu *Logické obvody*. Tehdy jsem při učení se Karnaughových map hledal program, který by byl zdarma a který by mi ukázal, jak se v mapě rozmisťují mintermy a jak se následně minimalizují. Bohužel jsem našel jen několik poměrně nekvalitních programů, které navíc nedokázaly zobrazit minimální pokrytí přímo v mapě. Napadla mě tedy myšlenka udělat vlastní program.

První realizace přišla o půl roku později, kdy jsem vytvořil jako svoji semestrální práci do předmětu *Programování v jazyce C* textový program pro minimalizace logických funkcí, který byl založen na Quine–McCluskey algoritmu. Název programu jsem zvolil **Bmin**, což je zkratka **B**oolean **min**imizer. Vznikla tedy první verze s označením 0.1.0, která byla asi o dva měsíce později rozšířena o generování výpisu z průběhu algoritmu – vznikla tak druhá verze 0.1.1.

V roce 2008 jsem si zapsl předmět *Technologie programování v C++* a jako semestrální práci jsem se rozhodl rozšířit aplikaci Bmin o grafické rozhraní, v němž by bylo možné zobrazit Karnaughovu mapu. Zvolil jsem k tomu grafickou knihovnu Qt. Vznikla tak verze 0.2.0, pro níž byla zvolena licence GPL 2.0. Důvodem byla svoboda a otevřenost kódu publikovaného pod touto licencí.

Následující semestr jsem si vybral předmět *Základy počítačové grafiky*, kde bylo nutné vytvořit jako semestrální práci aplikaci v OpenGL. Rozhodl jsem se pro trochu netradiční řešení a vytvořil Booleovskou n-krychli ve 3D prostoru. To se podařilo a vydal jsem tak verzi 0.3.0.

Poté jsem s vývojem aplikace na nějakou dobu přestal. Až při zvažování tématu mé diplomové práce jsem se rozhodl pro dokončení této aplikace. Od té doby byla aplikace v podstatě kompletně přepsaná a počet řádků zdrojového kódu se rozrostl na více než 17 tisíc. Zároveň došlo k přelicencování na GPL 3.0. Bylo vydáno několik meziverzí až do současné verze 0.6.0. Jejímu popisu je věnována celá kapitola 3.

Kapitola 2

Metody pro minimalizaci logických funkcí

Tato kapitola popisuje metody pro minimalizaci logických funkcí. Nejprve jsou uvedeny základní definice a značení. Poté je stručně popsána historie minimalizace a potom už následují dvě sekce, které popisují vizualizační metody. Konkrétně se jedná o Booleovskou n-krychli a Karnaughovu mapu. Poslední dvě sekce popisují algoritmy používané pro minimalizaci. Konkrétně jde o algoritmus Quine-McCluskey a sadu algoritmů Espresso.

2.1 Základní definice a značení

V této sekci jsou uvedeny základní pojmy a značení, které se budou ve zbytku práce užívat. Předpokládá se, že čtenář je znalý základů matematické logiky a logických obvodů. Proto jsou nejzákladnější pojmy vynechány.

Pokud je zvoleno $\mathbf{B} = \{0, 1\}$ a $\mathbf{Y} = \{0, 1, 2\}$ je logická funkce f definována jako

$$f : \mathbf{B}^n \to \mathbf{Y}^m$$

kde $x = [x_1, \ldots, x_n] \in \mathbf{B}^n$ je vstup, přičemž n značí počet vstupních proměnných, a $y = [y_1, \ldots, y_m] \in \mathbf{Y}^m$ je výstup. Zde je m počet výstupních proměnných. Tato práce se omezuje pouze na jednohodnotový výstup. Pak bude $m = 1$ a $y \equiv y_1$. Je třeba poznamenat, že hodnota 2 v $\mathbf{Y}$ značí neurčený nebo-li don't-care stav.

Všechna x, pro něž platí, že $f(x) = 1$, náleží do množiny **on-set** značené $\mathcal{F}$. Dále všechna x, pro něž platí, že $f(x) = 0$, náleží do množiny **off-set** značené $\mathcal{R}$. A nakonec všechna x, pro něž platí, že $f(x) = 2$, náleží do množiny **don't-care set** značené $\mathcal{D}$. Samotné x bývá často nazýváno term nebo cube. V dalším textu bude používán název **term** a někdy pouze zkráceně t. Obecná množina termů se nazývá **pokrytí** a často bude označována jako $\mathcal{C}$. Operace = pro dvě pokrytí bude značit logickou ekvivalenci. Pro i-tý term t v pokrytí $\mathcal{C}$ bude používáno značení t^i nebo $\mathcal{C}_i$, pokud nebude z kontextu jasné jakému pokrytí t náleží. Počet termů v $\mathcal{C}$ bude značen jako $|\mathcal{C}|$. Hodnoty proměnných $x_1, \ldots, x_n$ pro určitý term t budou nazývány **literály**. Literál na j-té pozici pak bude značen jako t_j nebo $\mathcal{C}_{i,j}$, pokud bude třeba zdůraznit, že se jedná o j-tý literál i-tého termu v pokrytí $\mathcal{C}$. Zároveň samozřejmě platí, že $i \in \{1, \ldots, |\mathcal{C}|\}$ a $j \in \{1, \ldots, n\}$.

Definiční obor literálu bude ještě rozšířen tak, že $\mathrm{t}_j \in \{0, 1, 2\}$. Hodnota 2 bude značit, že se daný literál v termu neobjevuje. Pokud se pak bude hovořit o počtu literálu, nebude tedy literál nabývající hodnoty 2 do tohoto součtu započítán.

Z výše uvedeného vyplývá, že pokrytí $\mathcal{F} \cup \mathcal{D} \cup \mathcal{R}$ pokrývá všechny termy funkce f, což lze zapsat jako $f = (\mathcal{F}, \mathcal{D}, \mathcal{R})$. Pro libovolné pokrytí $\mathcal{C}$ existuje doplněk do $\mathcal{F} \cup \mathcal{D} \cup \mathcal{R}$. Ten se nazývá **komplement** a bude značen $\overline{\mathcal{C}}$. Prázdné pokrytí bude značeno $\emptyset$, tedy jako prázdná množina. Znak $\emptyset$ bude používán i ve významu neexistujícího termu a někdy i neexistujícího literálu. Tedy takového literálu, který nabývá hodnoty 2.

Logická funkce, jenž pro všechna x je $f(x) = 1$, se nazývá **tautologie**. Naopak funkce, jenž pro všechna x je $f(x) = 0$, se nazývá **kontradikce**. Pro tautologii tak platí, že $\mathcal{F} = \mathcal{F} \cup \mathcal{D} \cup \mathcal{R}$, a naopak pro kontradikci platí, že $\mathcal{R} = \mathcal{F} \cup \mathcal{D} \cup \mathcal{R}$.

Mezi operace nad termy, které je třeba popsat, patří **průnik** dvou termů. Průnik termů t a u bude značen $\mathrm{t} \cap \mathrm{u}$. Výpočet se provádí pro každý literál zvlášť a řídí následujícími pravidly

		u_j		
	$\cap$	0	1	2
	0	0	$\emptyset$	0
t_j	1	$\emptyset$	1	1
	2	0	1	2

, $\quad j \in \{1, \ldots, n\}$

Pokud některý z literálů neexistuje, je produkt prázdný. V opačném případě je výsledkem nový term. Někdy bude uváděn průnik termu a pokrytí, tedy $\mathrm{t} \cap \mathcal{C}$. Výsledkem této operace je nové pokrytí složené z průniku každého termu z $\mathcal{C}$ s termem t. Počet neexistující literálů bývá nazýván vzdáleností dvou termů a pro termy t a u se značí jako $\delta(\mathrm{t}, \mathrm{u})$.

Další důležitou operací je **konsensus**. Konsensus e termů t a u je značen jako $\mathrm{e} = \mathrm{t} \odot \mathrm{u}$. Term e se určí na základě následujících pravidel

$$\mathrm{e} = \begin{cases} \mathrm{t} \cap \mathrm{u} & : \quad \delta(\mathrm{t}, \mathrm{u}) = 0 \\ \emptyset & : \quad \delta(\mathrm{t}, \mathrm{u}) \geq 2 \end{cases}$$

pokud je $\delta(\mathrm{t}, \mathrm{u}) = 1$, pak se každý literál určí jako

$$\mathrm{e}_j = \begin{cases} 2 & : \quad \mathrm{t}_j \cap \mathrm{u}_j = \emptyset \\ \mathrm{t}_j \cap \mathrm{u}_j & : \quad \mathrm{t}_j \cap \mathrm{u}_j \neq \emptyset \end{cases}$$

kde $j \in \{1, \ldots, n\}$.

Logické funkce lze zapsat několika způsoby. Asi nejpřehlednější možný způsob představuje **pravdivostní tabulka**, která vypisuje všechny možné kombinace proměnných a výstupní hodnotu daných kombinací. Například funkci o třech proměnných znázorňuje pravdivostní tabulka zobrazená v Tabulce 2.1.

x_1	x_2	x_3	y
0	0	0	1
0	0	1	2
0	1	0	2
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Tabulka 2.1: Pravdivostní tabulka

Pravděpodobně nejčastějším způsobem zápisu je **algebraický tvar**. V něm se používají dvě formy zápisu. První, a také používanější, je **DNF** (disjunktivní normální forma)[1]. Jedná se o součet součinových termů. Příkladem může být funkce

$$f(x_1, x_2, x_3) \equiv \overline{x}_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2\overline{x}_3 + x_1x_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + x_1\overline{x}_2x_3$$

kde čára nad x_j značí negovanou hodnotu literálu, tedy $\overline{x}_j = 0$ a obráceně $x_j = 1$. Druhou formou je **KNF** (konjunktivní normální forma)[2]. Jedná se součin součtových termů. Obě formy lze vzájemně převádět, přičemž lze využít De Morganových zákonů, které říkají, že

$$\overline{x_1x_2} = \overline{x}_1 + \overline{x}_2$$

Předešlý příklad DNF by se tedy v KNF zapsal jako

$$f(x_1, x_2, x_3) \equiv (x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$$

V Obou předešlých příkladech byly funkce zapsány v úplné formě. To znamená, že v každém termu byly zastoupeny všechny literály. V DNF se takový term označuje **minterm** a v KNF **maxterm**. Po provedení minimalizace mohou být některé termy expandovány a počet literálů se tak v daném termu sníží. Minimalizovaná forma funkce z předešlého příkladu vypadá v DNF následovně

$$f(x_1, x_2, x_3) \equiv \overline{x}_2 + x_1\overline{x}_3$$

a v KNF následovně

$$f(x_1, x_2, x_3) \equiv (x_1 + \overline{x}_2)(\overline{x}_2 + \overline{x}_3)$$

V dalších sekcích se bude často hovořit o mintermech. Je třeba si uvědomit, že vše by platilo obdobně i pro maxtermy, i když o nich nebude zmínka.

Před uvedením dalšího možného zápisu je třeba poznamenat, že také mintermy a maxtermy lze zapsat několika způsoby. Prvním je **algebraický** tvar, který je uveden výše. Dále je možný ještě **binární** a **dekadický** zápis. Například všechny tři níže uvedené zápisy označují tentýž minterm

$$x_1\overline{x}_2\overline{x}_3 \equiv 100 \equiv 4$$

[1] V angličtině Sum of Products
[2] V angličtině Product of Sums

Stejné by to bylo i u maxtermu, až na rozdílný algebraický zápis, který by byl $(\overline{x}_1 + x_2 + x_3)$.

Dalším typem zápisu funkce je tzv. **výčtový zápis**, který se uplatní v dekadickém zápis mintermu (resp. maxtermu). Znovu je třeba rozlišit, zda se jedná o DNF či o KNF. Pro DNF se výčet zapisuje jako suma všech mintermů z $\mathcal{F}$ plus suma všech mintermů z $\mathcal{D}$. Mintermy jsou zapisovány v už zmíněném dekadickém tvaru. Pro přehlednost je vhodné obě sumy odlišit indexem, který zdůrazní, o jakou sumu se jedná. Pokud by například pro funkci o třech proměnných bylo $\mathcal{F} = \{\overline{x}_1\overline{x}_2\overline{x}_3, x_1\overline{x}_2\overline{x}_3, x_1x_2\overline{x}_3\}$ a $\mathcal{D} = \{\overline{x}_1\overline{x}_2x_3 + x_1\overline{x}_2x_3\}$, zapsal by se výčet následovně

$$f(x_1, x_2, x_3) \equiv \sum\nolimits_{\mathcal{F}}(0, 4, 6) + \sum\nolimits_{\mathcal{D}}(1, 2)$$

V případě KNF se výčet zapisuje jako produkt všech maxtermů z $\mathcal{R}$ krát produkt všech maxtermů z $\mathcal{D}$. Po vypočítání $\mathcal{R} = \{\overline{x}_1x_2x_3, x_1\overline{x}_2x_3, x_1x_2x_3\}$ jako komplement $\mathcal{F} \cup \mathcal{D}$ z předchozího příkladu bude zápis funkce vypadat následovně

$$f(x_1, x_2, x_3) \equiv \prod\nolimits_{\mathcal{R}}(3, 5, 7) \cdot \prod\nolimits_{\mathcal{D}}(1, 2)$$

Posledním zápisem logické funkce, který zde bude zmíněn je **maticový zápis**. Ten bude často používán především při popisu algoritmů Espressa. Je proto omezen pouze na termy v DNF. Pokrytí $\mathcal{C}$ se zapíše do matice $\mathbf{M}(\mathcal{C})$, jejíž počet řádků odpovídá počtu termů v pokrytí $\mathcal{C}$ a jejíž počet sloupců odpovídá počtu proměnných. Hodnoty v matici jsou pak určeny podle následujícího pravidla

$$\mathbf{M}(\mathcal{C})_{i,j} = \begin{cases} 0 & : \quad \mathrm{t}_j^i = 0 \quad \textit{(x_j se objeví v i-tém termu v negované hodnotě)} \\ 1 & : \quad \mathrm{t}_j^i = 1 \quad \textit{(x_j se objeví v i-tém termu)} \\ 2 & : \quad \mathrm{t}_j^i = 2 \quad \textit{(x_j se neobjeví v i-tém termu)} \end{cases}$$

Například algebraický zápis funkce

$$f \equiv \overline{x}_4 + x_1\overline{x}_2 + x_1\overline{x}_3 + \overline{x}_1x_2x_3$$

by v maticovém tvaru vypadal následovně

$$\mathbf{M}(\mathcal{C}) = \begin{pmatrix} 2 & 2 & 2 & 0 \\ 1 & 0 & 2 & 2 \\ 1 & 2 & 0 & 2 \\ 0 & 1 & 1 & 2 \end{pmatrix}$$

Pro další text je třeba zmínit důležité rozdělení termů. Pokud je dána logická funkce $f = (\mathcal{F}, \mathcal{D}, \mathcal{R})$ v DNF, tak **implikant** funkce f je každý term, které má prázdný průnik se všemi termy z $\mathcal{R}$. **Přímý implikant** je pak takový implikant, který není pokryt žádným jiným implikantem. Pokud by f byla v KNF, měl by implikant funkce f prázdný průnik se všemi termy z $\mathcal{F}$.

Na závěr této sekce bude popsána speciální třída logických funkcí, které se hodně využívá v Espressu. Jedná se o tzv. **unátní funkce**. Pro jednohodnotové výstupní funkce je jejich popis velmi jednoduchý. Funkce, která je daná minimálním pokrytím $\mathcal{C}$, je unátní v proměnné x_j pokud platí, že j-tý sloupec v $\mathbf{M}(\mathcal{C})$ neobsahuje žádnou nulu (taková funkce se nazývá

monotonně rostoucí v x_j) nebo žádnou jedničku (taková funkce se nazývá monotonně klesající v x_j). Unátní funkce je pak taková funkce, která je unátní ve všech svých proměnných. Jinými slovy to znamená, že průnik všech termů v $\mathcal{C}$ je neprázdný. Příkladem takové funkce, kterou tvoří minimální pokrytí $\mathcal{C}$, je například

$$\mathbf{M}(\mathcal{C}) = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 0 & 2 \end{pmatrix}$$

Pokud funkce není unátní, tak lze najít tzv. **binátní proměnnou**. Ta se zjistí tak, že se nejdříve vypočítá pro každý sloupec v $\mathbf{M}(\mathcal{C})$ počet různých dvojic $[0_{i_1}, 1_{i_2}]$, kde i_1 a i_2 jsou čísla řádků a platí že $i_1 \neq i_2$. Pak sloupec s největším počtem těchto dvojic je označen jako j-tý sloupec a nejvíce binátní proměnnou je x_j.

2.2 Trocha Historie

Základy pro syntézu logických obvodů položil George Boole [6]. Stěžejní byla zejména jeho práce *The Laws of Thought*[3] z roku 1854 [1], v níž definoval novou algebru, která po něm zároveň byla pojmenována jako Booleova algebra. V roce 1938 pak Claude Shannon ukázal, že dvouhodnotová Booleova algebra může popsat operace v kombinačních obvodech. S jejich postupným rozvojem nabralo toto odvětví matematiky na důležitosti. Pozornost byla zaměřena zejména na minimalizace logických funkcích, protože umožnily použití menšího počtu hradel a tím i ušetření finančních prostředků a zrychlení obvodu. S prvním řešením přišel Mourice Karnaugh [7] roku 1953. Ten za použití Veitcheho diagramů vytvořil speciální mapu, jež umožňovala vizuální minimalizaci. Dostala jméno Karnaughova mapa [25]. Její přehlednost byla ale špatná už pro šest proměnných, a tak bylo třeba najít nějaké lepší řešení. S tím přišli roku 1956 pánové Willard Van Orman Quine [9] a Edward J. McCluskey [8]. Algoritmus, který vymysleli, byl po nich pojmenován a dostal tak jméno algoritmus Quine–McCluskey [27]. Dlouhou dobu byl de facto standardem. Jeho výhodou byla snadná implementace na počítači a výsledná optimální minimální funkce. Jednalo se tak o exaktní algoritmus, což mělo i své nevýhody. Jeho složitost byla totiž exponenciální a pro více jak 15 proměnných byl prakticky nepoužitelný. S masivním příchodem PLA zařízení v 70. letech bylo třeba minimalizovat funkce, které měly typicky kolem 30 proměnných. Začalo tak vznikat mnoho heuristik, které se optimálnímu řešení blížily. Asi nejznámější z nich je sada algoritmů s názvem Espresso [2]. To vzniklo počátkem 80. let minulého století a jeho autory jsou pánové Robert K. Brayton, Gary D. Hatchel, Curtis T. McMullen a Alberto L. Sangiovanni-Vincentelli. Espresso navazovalo na heuristiku MINI [3] vzniklou v IBM. Jeho výsledky však byly mnohem lepší a navíc mělo rozšíření pro minimalizaci funkcí s více výstupními hodnotami. Výzkum těchto funkcí vedl k mnoha novým heuristikám. Postupem času se vývoj v oblasti minimalizace logických funkcí přenesl do komerčních EDA (Electronic design automation) nástrojů [23], které celý proces syntézy automatizují. Mezi největší firmy vytvářející tyto nástroje patří například Synopsys [12] nebo Cadence [11].

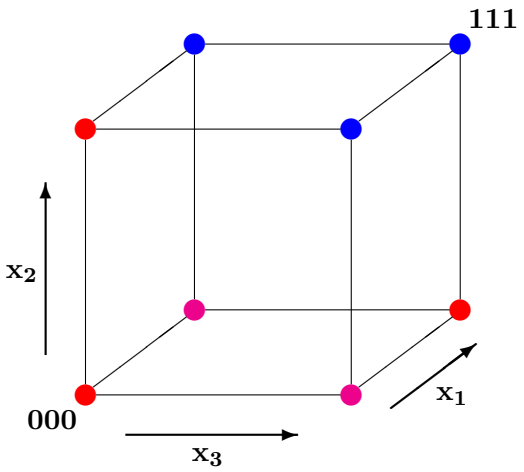
[3] Celým názvem *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and probabilities*

2.3 Booleovská n-krychle

Booleovská n-krychle značená $\mathbf{B}^n$ slouží k vizuálnímu vyjádření logické funkce. Jedná se o n-rozměrnou krychli, jejíž vrcholy představují mintermy[4] a hrany představují změnu jedné proměnné z 0 na 1 resp. z 1 na 0. Sousedící mintermy se tedy liší v jedné proměnné. Pokud by tedy n-krychle byla pokládána za graf, jednalo by se o graf s 2^n uzly a $n \cdot 2^{n-1}$ hranami, přičemž každý uzel má stupeň rovný n. Nejpřehlednější je n-krychle pro $n \leq 3$, v níž se každá proměnná mění s prostorovými souřadnicemi. Tedy například x_3 se mění s osou `x`, x_2 s osou `y` a x_1 s osou `z`. Obrázek 2.1 zobrazuje $\mathbf{B}^3$ pro funkci

$$f(x_1, x_2, x_3) \equiv \sum\nolimits_{\mathcal{F}}(0, 2, 5) + \sum\nolimits_{\mathcal{D}}(1, 4)$$

Je nutné poznamenat, že mintermy z $\mathcal{F}$ jsou zobrazen červenou barvou, mintermy z $\mathcal{R}$ modrou barvou a mintermy z $\mathcal{D}$ purpurovou barvou.



Obrázek 2.1: Booleovská n-krychle pro tři proměnné

Částečně přehledná je ještě krychle pro čtyři proměnné, která se kreslí jako menší $\mathbf{B}^3$ ve větší $\mathbf{B}^3$. přičemž jejich stejné mintermy jsou propojeny hranou, která představuje změnu čtvrté proměnné. Krychle pro větší počet proměnných se velice obtížně kreslí a je zároveň i velmi nepřehledná.

Term, který má počet literálů m je tvořen podkrychlí $\mathbf{B}^m$, přičemž $\mathbf{B}^m \subseteq \mathbf{B}^n$. Tato podkrychle pokrývá všechny mintermy, které daný term implikuje.

[4] Je uvažována pouze DNF. U KNF je vše stejné až na to, že místo o mintermy se jedná o maxtermy.

2.4 Karnaughova mapa

Karnaughova mapa nebo zkráceně pouze K-mapa je v podstatě maticový zápis výstupních hodnot všech mintermů kompletně specifikované logické funkce. Tedy jakýsi přepis pravdivostní tabulky (konkrétně jejího sloupce s výstupními hodnotami) do matice. V ní jsou mintermy uspořádány po řádcích podle Grayova kódu.

Tabulka 2.2 ukazuje strukturu K-mapy. Místo výstupních hodnot jsou v ní uvedeny pozice mintermů. Pro přehlednost jsou její sloupce a řádky číslovány od nuly. Nultý sloupec a řádek slouží k popisu toho, jakých hodnot nabývají jednotlivé proměnné. Zde byl zvolen binární popis[5], z něhož je dobře patrné, že proměnné jsou uspořádány podle Grayova kódu. Z toho vyplývá, že i jednotlivé mintermy jsou uspořádány podle Grayova kódu, jak ukazují hodnoty v tabulce, které uvádějí dekadický zápis mintermu, tedy jeho pozici v pravdivostní tabulce. Hlavní výhodou tohoto rozmístění je, že spolu sousedí vždy dva mintermy, jenž se liší pouze v jedné proměnné. Sousednost však platí i mezi některými mintermy, které nejsou přímo vedle sebe v mapě. Kolik sousedů má každý minterm závisí na počtu proměnných dané funkce. Například K-mapu pro funkci o třech proměnných si lze představit jako válec, protože spolu sousedí proměnné v prvním a posledním sloupci. K-mapa o čtyřech proměnný tvoří torus [30], tedy útvar připomínající pneumatiku. Sousedící jsou zde i první a poslední řádek. U většího počtu proměnných se situace stává ještě komplikovanější. Pro více než 6 proměnných už nemá smysl K-mapu kreslit, protože je velmi nepřehledná.

$x_1x_2\backslash x_3x_4$	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Tabulka 2.2: Rozmístění mintermů v Karnaughově mapě pro čtyři proměnné

Výhodou výše zmíněné sousednosti je možnost přehledně zobrazit termy s menším počtem literálů než je počet proměnných. Ty pokrývají v K-mapě skupiny sousedních mintermů. Tyto skupiny se vždy rozkládají na ploše, jejíž šířka a výška musí být mocninou čísla 2, přičemž jednotkou je samozřejmě políčko v K-mapě. Z toho vyplývá, že lze jednoduše určit minimální pokrytí pro danou funkci tak, že se vyhledají největší možné plochy pokrývající mintermy s výstupní hodnotou 1 nebo 2 (v případě KNF se pokrývají maxtermy s výstupní hodnotou 0 nebo 2).

Tabulka 2.3 ukazuje mapu funkce

$$f(x_1, x_2, x_3, x_4) \equiv \sum\nolimits_{\mathcal{F}}(4, 5, 6, 7, 15) + \sum\nolimits_{\mathcal{D}}(3, 13)$$

a zvýrazňuje v ní přímý implikant x_2x_4. Druhým přímým implikantem je $\overline{x}_1x_2$, který rozkládá přes všechna políčka ve druhém řádku Tabulky. Výsledná minimální funkce je tedy

$$f(x_1, x_2, x_3, x_4) \equiv x_2x_4 + \overline{x}_1x_2$$

[5]Dalším popisem bývají tzv. negační čáry, kde hodnota 0 pro danou proměnnou je znázorněna čárou a hodnota 1 je znázorněna vynecháním čáry.

$x_1x_2 \backslash x_3x_4$	00	01	11	10
00	0	0	2	0
01	1	1	1	1
11	0	2	1	0
10	0	0	0	0

Tabulka 2.3: Karnaughova mapa se zvýrazněním termu x_2x_4

2.5 Quine–McCluskey

Algoritmus **Quine–McCluskey** je jedním z prvních algoritmů pro minimalizaci logických funkcí vhodným k implementaci na počítači. Skládá se ze dvou kroků:

1. Jsou nalezeny všechny přímé implikanty.

2. Z nalezených přímých implikantů je zjištěno minimální pokrytí.

V závislosti na použité metodě pro hledání nejmenšího pokrytí se jedná buď o exaktní algoritmus nebo o heuristiku.

V prvním kroku jsou tedy hledány všechny přímé implikanty. Nejprve je třeba získat všechny mintermy (zatím je uvažována pouze DNF) z pokrytí $\mathcal{F} \cup \mathcal{D}$. Poté se vytvoří jakási matice seznamů. V ní jsou mintermy rozřazeny v řádcích podle počtu jedniček a ve sloupcích podle počtu literálů. V každém seznamu jsou tedy termy se stejným počtem jedniček a stejným počtem literálů. Na začátku je samozřejmě zaplněn jen sloupec s počtem literálů rovným počtu proměnných. Po této inicializační fázi následuje kombinování termů. Kombinují se vždy pouze seznamy termů, které jsou ve stejném sloupci a jejich řádky spolu sousedí. Tedy termy, jejichž počet jedniček se liší o hodnotu jedna. Kombinování znamená, že každý term z jednoho seznamu vytvoří dvojici s každým termem z druhého seznamu a z této dvojice se udělá konsensus. Pokud je neprázdný, tak je zřejmé, že bude mít počet literálu o jedna menší. Vloží se tedy do následujícího sloupce na stejný řádek jako je term s menším počtem jedniček z kombinované dvojice. Když se takto zkombinují všechny sousední řádky, pokračuje se kombinováním řádku ve sloupci pro termy s o jedna nižším počtem literálů. Hledání přímých implikantů končí, když už není co kombinovat. Během průběhu algoritmu se udržuje speciální seznam se všemi termy. Pokud při kombinování nastane, že $\mathrm{e} = \mathrm{t} \odot \mathrm{u} \neq \emptyset$, tak jsou termy t a u z tohoto seznamu odstraněny a je do něj přidán term e. Na konci tak zbudou pouze přímé implikanty. Těch muže být velký počet. Dá se dokázat, že horní hranice je $O(\frac{3^n}{n})$, kde n je počet proměnných.

Po získání přímých implikantů následuje druhá část algoritmu. V ní se nejprve vytvoří tabulka pokrytí mezi nalezenými přímými implikanty a původními mintermy, které náleží do $\mathcal{F}$. Přímé implikanty, které pokrývají jako jediní některý minterm, se nazývají nespornými přímými implikanty a musí být tedy ve výsledném minimálním pokrytí. Ze zbylých přímých implikantů je nutné najít minimální množinu, která bude tvořit ekvivalentní pokrytí. Jedná se o problém hledání nejmenšího pokrytí [22], který je NP úplný [26]. Ten lze řešit buď pomocí zpětného prohledávání (Backtracking [19]) nebo použít heuristiku, která se optimálnímu řešení blíží. Zde se nejvíce nabízí vybrání termu, který pokrývá nejvíce doposud nepokrytých

mintermů a poté ho vložit do minimálního prokrytí. To opakovat, dokud nejsou pokryty všechny mintermy.

Pro příklad funkce

$$f(x_1, x_2, x_3, x_4) \equiv \sum\nolimits_{\mathcal{F}}(3, 5, 6, 8, 9, 11, 15) + \sum\nolimits_{\mathcal{D}}(4, 7, 12)$$

má průběh zobrazený v Tabulce 2.4. K ní je nutné poznamenat, že místo hodnoty 2 pro chybějící literál je použita pomlčka. U přímých implikantů je uvedena hvězdička. Tabulka pokrytí je zobrazena v Tabulce 2.5. V ní jsou nesporné přímé implikanty označeni hvězdičkou.

Počet jedniček	4 literály		3 literály		2 literály	
1	m4	0100	m(4,5)	010-	m(4,5,6,7)	01--*
	m8	1000	m(4,6)	01-0		
			m(4,12)	-100*		
			m(8,9)	100-*		
			m(8,12)	1-00*		
2	m3	0011	m(3,7)	0-11	m(3,7,11,15)	--11*
	m5	0101	m(3,11)	-011		
	m6	0110	m(5,7)	01-1		
	m9	1001	m(6,7)	011-		
	m12	1100	m(9,11)	10-1*		
3	m7	0111	m(7,15)	-111		
	m11	1011	m(11,15)	1-11		
4	m15	1111				

Tabulka 2.4: Quine–McCluskey – Hledání přímých implikantů

	m3	m5	m6	m8	m9	m11	m15
m(3,7,11,15)*	×					×	×
m(4,5,6,7)*		×	×				
m(4,12)							
m(8,9)				×	×		
m(8,12)				×			
m(9,11)					×	×	

Tabulka 2.5: Quine–McCluskey – Tabulka pokrytí

Při hledání pokrytí v Tabulce 2.5 se nejprve odstraní nesporné přímé implikanty, kterými jsou m(3,7,11,15) a m(4,5,6,7), a s nimi všechny pokryté mintermy. Navíc se odstraní i term m(4,12), protože nepokrývá žádné mintermy z $\mathcal{F}$. Zbývá tak pokrýt pouze mintermy m8 a m9. Ze zbylých přímých implikantů se proto odstraní m(8,9), protože je oba pokrývá. Výsledná minimální funkce je tedy

$$f(x_1, x_2, x_3, x_4) \equiv \overline{x}_1 x_2 + x_3 x_4 + x_1 \overline{x}_2 \overline{x}_3$$

Celý předešlý popis se týká funkce v DNF. Pro KNF by vše bylo stejné, až na rozdíl, že by se jednalo o maxtermy a hodnoty by byly rozřazeny do řádků podle počtu nul.

2.6 Espresso

Espresso je sada algoritmů pro minimalizaci logických funkcí. Jedná se o heuristiku, což znamená, že výsledná minimalizovaná funkce nemusí být optimální. Výhodou ale je, že doba minimalizace je mnohem kratší.

Minimalizace se skládá z několika procedur, které se opakovaně provádějí v hlavní smyčce algoritmu. Následující podsekce podrobně popisují nejdůležitější procedury. Nejprve je však uvedena hlavní smyčka algoritmu.

2.6.1 Hlavní smyčka

Tato sekce popisuje celý průběh algoritmu Espresso. Tedy v jakém pořadí jsou volány jednotlivé procedury. Vše je pro lepší pochopení trochu zjednodušeno. Navíc je úplně vynechána procedura `MAKE-SPARSE`, která u jednovýstupových funkcí nemá význam. Celý průběh je zapsán v pseudokódu na Obrázku 2.2.

Je třeba poznamenat, že procedura `COST` je tzv. cenová funkce pro $\mathcal{F}$. Cena je dána na prvním místě počtem termů a na druhém celkovým počtem všech literálů.

```
ESPRESSO(F,D) {
  R ← COMPLEMENT(F,D)                    // Vypočte komplement a uloží ho do R.
  LOOP1:
    c1 ← c2 ← c3 ← c4 ← COST(F)          // Inicializuje ceny.
    F ← EXPAND(F,R)                      // Najde přímé implikanty.
    if (first-pass)                      // V případě prvního průchodu
      (F,D,E) ← ESSENTIAL-PRIMES(F,D)    // separuje nesporné přímé implikanty.
    if (c1 = COST(F))                    // Pokud se cena nezměnila
      goto OUT                           // skočí na poslední fázi smyčky
    c1 ← COST(F)                         // jinak cenu uloží.
    F ← IRREDUNDANT(F,D)                 // Najde minimální pokrytí.
    if (c2 = COST(F))
      goto OUT
    c2 ← COST(F)
  LOOP2:
    F ← REDUCE(F,D)                      // Dostává řešení z lokálního optima.
    if (c3 = COST(F))
      goto OUT
    c3 ← COST(F)
  OUT:
    c ← COST(F)
    if (c4 = c)
      goto QUIT
    F ← LAST-GASP(F,D,R)                 // Poslední pokus o vylepšení řešení.
    if (c = COST(F))                     // Pokud se cena během LAST-GASP nezměnila
      goto QUIT                          // cyklus je ukončen
    goto LOOP1                           // jinak skočí zpět na začátek.
  QUIT:
    F ← F ∪ E                            // Vrátí zpět nesporné přímé implikanty.
    return F
}
```

Obrázek 2.2: Hlavní smyčka Espressa

2.6.2 COFACTOR

`COFACTOR` není přímo procedurou Espressa. Je to spíše metoda pro výpočet kofaktoru. Nejedná se však přímo o kofaktor známý z lineární algebry pro výpočet determinantu [21], i když jeho význam je obdobný. Je to jakýsi algebraický doplněk termu p v daném pokrytí $\mathcal{C}$. Výsledné pokrytí se označuje jako $\mathcal{C}_{\mathrm{p}}$. Kofaktor se počítá pro každý term v $\mathcal{C}$ zvlášť podle následujícího pravidla

$$(\mathrm{t}_{\mathrm{p}}^{i})_j = \begin{cases} \emptyset & : \quad \mathrm{t}_j^i \cap \mathrm{p}_j = \emptyset \\ 2 & : \quad (p_j = 0) \vee (p_j = 1) \\ \mathrm{t}_j^k & : \quad p_j = 2 \end{cases}$$

Nejlépe vše bude vidět na příkladu. Pokud je dáno pokrytí $\mathcal{C}$, jehož zápis v maticovém tvaru vypadá následovně

$$\mathbf{M}(\mathcal{C}) = \begin{pmatrix} 1 & 1 & 0 & 2 \\ 0 & 1 & 2 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

tak kofaktor $\mathcal{C}_{\mathrm{p}}$, kde

$$\mathbf{M}(\mathrm{p}) = \begin{pmatrix} 1 & 1 & 2 & 2 \end{pmatrix}$$

je

$$\mathbf{M}(\mathcal{C}_{\mathrm{p}}) = \begin{pmatrix} 2 & 2 & 0 & 2 \\ 2 & 2 & 1 & 1 \end{pmatrix}$$

Speciální význam má kofaktor pro jednu proměnnou. Ten se značí pro i-tou proměnou jako $\mathcal{C}_{x_i}$, resp. $\mathcal{C}_{\overline{x}_i}$ pro negovanou hodnotu. Výpočet $\mathcal{C}_{x_3}$, resp. $\mathcal{C}_{\overline{x}_3}$, je stejný, jako výpočet $\mathcal{C}_{\mathrm{p}}$ pro

$$\mathbf{M}(\mathrm{p}) = \begin{pmatrix} 2 & 2 & 1 & 2 \end{pmatrix}$$

resp.

$$\mathbf{M}(\mathrm{p}) = \begin{pmatrix} 2 & 2 & 0 & 2 \end{pmatrix}$$

Výsledek tedy je

$$\mathbf{M}(\mathcal{C}_{x_3}) = \begin{pmatrix} 1 & 1 & 0 & 2 \\ 1 & 1 & 1 & 2 \end{pmatrix}; \quad \mathbf{M}(\mathcal{C}_{\overline{x}_3}) = \begin{pmatrix} 1 & 1 & 0 & 2 \\ 0 & 1 & 2 & 2 \end{pmatrix};$$

2.6.3 Shannonův rozklad

Shannonův rozklad stejně jako `COFACTOR` není procedurou Espressa. Říká, že pro jakoukoliv proměnou platí, že

$$\mathcal{C} = x_1 \mathcal{C}_{x_1} + \overline{x}_1 \mathcal{C}_{\overline{x}_1}$$

To znamená, že obě strany rovnice jsou logicky ekvivalentní a že lze tedy funkci rekurzivně rozkládat. To se uplatní například při zjišťování tautologie, jak je popsáno v následující podsekci.

2.6.4 TAUTOLOGY

Zjišťování, zda je dané pokrytí tautologií, je v Espressu velmi časté. Hlavním důvodem je, že to umožňuje zjistit, zda je určitý term pokryt daným pokrytím. Vše podstatné je uvedeno v následující větě[6].

Věta 1 *Pokud platí, že pokrytí* $\mathcal{C}_{\mathrm{p}}$ *je tautologie, tak pokrytí* $\mathcal{C}$ *pokrývá term* p.

Samotný dále popsaný algoritmus je velmi zjednodušen. Jedná o tzv. Vanilla Tautology, kde jsou vynechány testy unátnosti a další rychlostní vylepšení. Jejich popis by si vyžadoval hlubší teoretické základy a to je mimo rozsah tohoto textu.

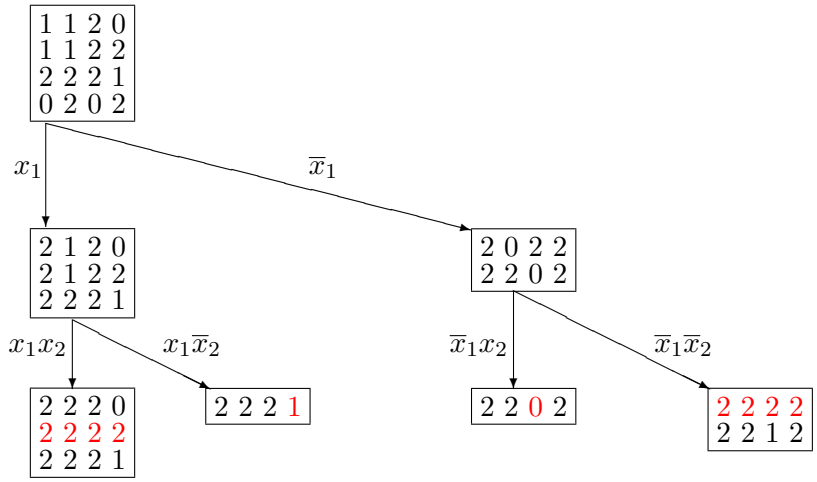
Pomocí Shannonova rozkladu se dá dokázat že platí následující věta

Věta 2 *Pokud pokrytí* $\mathcal{C}$ *je tautologie, tak musí být tautologie i* $\mathcal{C}_{x_i}$ *a* $\mathcal{C}_{\overline{x}_i}$ *pro libovolné* $i \in \{1, \ldots, n\}$

Tato věta umožňuje rekurzivní vykonávání algoritmus. Nejprve je však třeba vyhodnotit speciální případy. Ty lze jednoduše určit z $\mathbf{M}(\mathcal{C})$, kde $\mathcal{C}$ je zkoumané pokrytí. Pokud v $\mathbf{M}(\mathcal{C})$ je řádek obsahující samé dvojky, jedná se o tautologii. Pokud je v $\mathbf{M}(\mathcal{C})$ sloupec se samými jedničkami nebo nulami o tautologii se nejedná. To vyplývá z podmínky, že o tautologii se nejedná v případě, že $\mathcal{C}_{x_i}$ nebo $\mathcal{C}_{\overline{x}_i}$ je prázdné. Pokud se nejedná o speciální případ, zavolá se algoritmus rekurzivně pro $\mathcal{C}_{x_i}$ a $\mathcal{C}_{\overline{x}_i}$. Pokud jedno z nich není tautologie, tak není ani $\mathcal{C}$ tautologie. Proměnná i se mění v každém patře rekurzivního stromu. Nejjednodušeji se v prvním kroku nastaví na hodnotu 1 a pak se postupně zvyšuje[7]. Příklad průběhu algoritmu pro $\mathcal{C} = \{x_1x_2\overline{x}_4, x_1x_2, x_4, \overline{x}_1\overline{x}_3\}$ ukazuje Obrázek 2.3. Vzhledem k tomu, že v posledním potře zbylo pokrytí, kde je sloupec pouze s jedničkou, tak pokrytí $\mathcal{C}$ není tautologie.

[6] Její dokázání je mimo rozsah tohoto textu, zájemci jej mohou nalézt v [2, str. 33]

[7] Espresso využívá výběru nejvíce binátní hodnoty. To má smysl, pokud je algoritmus rozšířen o speciální vyhodnocení v případě unátního pokrytí.



Obrázek 2.3: Ukázka průběhu algoritmu `TAUTOLOGY`

2.6.5 EXPAND

Procedura `EXPAND` slouží k nalezení přímých implikantů. Toho je dosaženo tak, že se prochází přes všechny termy z $\mathcal{F}$ a každý je maximálně zvětšen, aby pokryl co nejvíce ostatních termů a počet jeho literálů byl co nejmenší. Jedná se tedy v podstatě o hladový algoritmus v němž je vážená funkce složena s počtu pokrytých termů a počtu literálů, u nichž má větší funkční hodnotu menší počet. Espresso používá heuristiku, která dává řešení blízké optimu. Rychlost řešení je zde závislá na počátečním seřazení termů. Jako nejvýhodnější se ukazuje zpracovat nejdříve termy, které jsou největší (mají nejmenší počet literálů).

Samotná expanze termu je prováděna pro každý term, který nebyl ještě pokryt jiným termem. Nejprve se z $\mathcal{R}$ a expandovaného termu (dále označovaného jako t) vytvoří tzv. blokovací matice (Blocking matrix) $\mathbf{B}$. Ta je vytvořena podle následujícího vzorce

$$\mathbf{B}_{i,j} = \begin{cases} 1 & : \quad ((\mathrm{t}_j = 1) \wedge (\mathcal{R}_{i,j} = 0)) \vee ((\mathrm{t}_j = 0) \wedge (\mathcal{R}_{i,j} = 1)) \\ 0 & : \quad \text{v ostatních případech} \end{cases}$$

To znamená, že matice $\mathbf{B}$ obsahuje jedničku na i-tém řádku a j-tém sloupci, pokud je zaručena disjunktnost j-té proměnné t a i-tého termu z $\mathcal{R}$. To zaručuje, že t nebude pokrývat i-tý term z $\mathcal{R}$. K tomu aby byla zaručena úplná disjunktnost se všemi termy z $\mathcal{R}$ je třeba najít takové sloupcové pokrytí $\mathbf{B}$, aby množina sloupců pokryla v každém řádku alespoň jednu jedničku. Tato množina sloupců se nazývá Lowering Set a dále bude označována jako $\mathbf{L}$. Název Lowering Set značí, že každý k-tý literál z t, kde $k \in \mathbf{L}$, si zachová svojí původní hodnotu a nebude tedy expandován (zvětšen na hodnotu 2). $\mathbf{L}$ dává zároveň finální podobu expanze t. K dosažení největšího snížení počtu literálů je tedy třeba, aby $\mathbf{L}$ byla co nejmenší.

Jako příklad je zadáno $\mathcal{F}$, $\mathcal{R}$ a $\mathrm{t} \equiv \mathcal{F}_3$ v maticovém tvaru

$$\mathbf{M}(\mathcal{F}) = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 2 \\ 0 & 1 & 1 & 2 & 1 \\ 1 & 2 & 0 & 0 & 2 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

$$\mathbf{M}(\mathcal{R}) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 2 & 1 \end{pmatrix}$$

$$\mathbf{M}(t) = \begin{pmatrix} 0 & 1 & 0 & 2 & 1 \end{pmatrix}$$

pak matice $\mathbf{B}$ vypočítaná z $\mathcal{R}$ a t bude vypadat následovně

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Snížení počtu literálů ale není jediný předpoklad. Tím druhým je pokrytí co možná největšího počtu termů. K tomu je třeba vytvořit tzv. matici pokrytí (Covering Matrix) $\mathbf{C}$. Ta se vytváří z t a $\mathcal{F}$ podle následujícího vzorce

$$\mathbf{C}_{i,j} = \begin{cases} 1 & : \quad ((\mathrm{t}_j = 1) \wedge (\mathcal{F}_{i,j} \neq 1)) \vee ((\mathrm{t}_j = 0) \wedge (\mathcal{F}_{i,j} \neq 0)) \\ 0 & : \quad \text{v ostatních případech} \end{cases}$$

To značí, že jednička v matici $\mathbf{C}$ určuje, že t nepokrývá i-tý term z $\mathcal{F}$, protože existuje proměnná, jejíž hodnota je pro oba termy rozdílná. K tomu, aby byl pokryt i-tý term z $\mathcal{F}$ je třeba, aby pro všechny $k \in \mathbf{L}$ bylo $\mathbf{C}_{i,k} = 0$.

Matice $\mathbf{C}$ získaná z t a $\mathcal{F}$ z předchozího příkladu bude vypadat následovně

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Samotný průběh algoritmu začíná nalezením všech sloupců z $\mathbf{B}$, které musí být v $\mathbf{L}$. Je zřejmé, že pokud v $\mathbf{B}$ existuje řádek pouze s jednou jedničkou, tak pozice této jedničku musí být v $\mathbf{L}$, protože jinak by daný řádek nemohl být pokryt. Nejprve jsou tedy nalezeny všechny řádky s jednou jedničkou a pozice této jedničky je přidána do $\mathbf{L}$. Jakmile je tak učiněno, mohou být všechny tyto sloupce a řádky, které jsou jimi pokryty, odstraněny z $\mathbf{B}$

i **C**. Nyní je buď **B** prázdná nebo jsou v ní pouze řádky, jež mají dvě a více jedniček. Lze tedy vybrat libovolný sloupec a ten odstranit. Je nejvýhodnější vybrat sloupec z **C**, který pokrývá nejméně termů (obsahuje nejvíce jedniček). Tento sloupec se vloží do množiny **R**. Jedná se o tzv. Raising Set, což značí, že každý k-tý literál z t, kde $k \in \mathbf{R}$, bude expandován, tedy $\mathrm{t}_k = 2$. Poté se ještě do **R** přidají sloupce se samými nulami v **B**. Celý proces se pak opakuje, dokud nejsou odstraněny všechny sloupce, nebo dokud **B** a **C** jsou neprázdné. Pokud **B** je po skončení hlavního cyklu neprázdná, tak je třeba ještě najít minimální sloupcové pokrytí **B**. K tomu se používá speciální heuristika, která najde řešení blízké optimu. Její popis je však mimo rozsah tohoto textu.

V úplném závěru algoritmu se expanduje t podle **L**. Všem literálům, které nejsou v **L**, je hodnota zvýšena na 2, což znamená, že jsou s t odstraněny. To lze matematicky zapsat takto (nově expandovaný term je označen jako t^+)

$$\mathrm{t}_j^+ = \begin{cases} \mathrm{t}_j & : \quad j \in \mathbf{L} \\ 2 & : \quad \text{v ostatních případech} \end{cases}$$

Nakonec jsou z $\mathcal{F}$ odstraněny všechny termy, které t pokrývá.

Celý postup bude ukázán na předchozím příkladu. Zde se nejprve vloží do **L** hodnota 2, protože čtvrtý sloupec obsahuje pouze jednu jedničku a ta je ve druhém sloupci. Není se z **B** odstraní druhý sloupec a třetí a čtvrtý řádek, které jsou jím pokryt. Z **C** se také odstraní druhý sloupec a první, třetí a čtvrtý řádek, který jsou jím pokryty. To, že bude odstraněn třetí řádek je zcela zřejmé už před započetím algoritmu, protože je zkoumán třetí term z $\mathcal{F}$ a tím pádem je jasné, že bude vždy pokrývat sám sebe. Redukované matice vypadají nyní následovně (pro lepší pochopení jsou v indexech uvedeny původní pozice v maticích)

$$\mathbf{B} = \begin{pmatrix} 1_{1,1} & 1_{1,3} & 0_{1,4} & 0_{1,4} \\ 0_{2,1} & 1_{2,3} & 0_{2,4} & 1_{2,5} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} 1_{2,1} & 1_{2,3} & 0_{2,4} & 1_{2,5} \\ 1_{5,1} & 0_{5,3} & 0_{5,4} & 1_{5,5} \\ 0_{6,1} & 1_{6,3} & 0_{6,4} & 1_{6,5} \end{pmatrix}$$

V dalším kroku lze odstranit libovolný sloupec, který se přidá do **R**. Vybere se tedy sloupec s největším počtem jedniček v **C**, což je pátý sloupec. Navíc se do **R** také přidá čtvrtý sloupec, protože je to sloupec se samými nulami v **B**. Do **R** se tedy vloží hodnoty 4 a 5. Tyto sloupce se zároveň odstraní a v **B** i **C** zůstanou pouze dva sloupce. Nyní se znovu vybere sloupec, který musí být v **L**, což je beze sporu třetí sloupec, protože v šestém řádku je nyní jen jedna jednička. Tento sloupec zároveň pokrývá oba zbylé řádky v **B**, takže po jejich odstranění je **B** prázdná a cyklus je ukončen.

Po skončení cyklu je $\mathbf{L} = \{2, 3\}$, což znamená, že se zkoumaný term expanduje následovně

$$\begin{pmatrix} 0 & 1 & 0 & 2 & 1 \end{pmatrix} \longrightarrow \begin{pmatrix} 2 & 1 & 0 & 2 & 2 \end{pmatrix}$$

Nakonec se ještě odstraní $\mathcal{F}_1$ z $\mathcal{F}$, který je jediný pokryt t^+. Jako jediný má totiž v druhém a třetím sloupci v **C** nuly.

2.6.6 IRREDUNDANT

Smyslem procedury `IRREDUNDANT` je odstranit redundantní termy z $\mathcal{F}$, které byly získány procedurou `EXPAND`. Celý průběh algoritmu se dá rozdělit na tři subprocedury. Každá bude detailněji popsána v následujících podsekcích. Nyní bude uveden pouze jejich obecný význam a návaznost na další subprocedury.

První subprocedurou je `REDUNDANT`. Ta slouží k rozdělení termů z $\mathcal{F}$ na ty, po jejichž vynechání se pokrytí nezmění (ty budou označovány $\mathbf{R}$ jako Redundantní) a na ty, po jejichž vynechání se pokrytí změní (ty budou označeny $\mathbf{E}$ jako Essentials tedy nesporné[8]).

Druhá subprocedura `PARTIALY-REDUNDANT` dostane jako parametr $\mathbf{E}$, $\mathbf{R}$ a $\mathcal{F}$. Jejím účelem je najít množinu $\mathbf{R_p} \subseteq \mathbf{R}$ takovou, že $\mathbf{R_p}$ neobsahuje žádné termy, které jsou pokryty $\mathbf{E}$.

Třetí subprocedura `MINIMAL-IRREDUNDANT` najde minimální množinu $\mathbf{R_c} \subseteq \mathbf{R_c}$ takovou, aby pokrytí $\mathbf{R_c} \cup \mathbf{E}$ bylo ekvivalentní $\mathcal{F}$ a počet termů v $\mathbf{R_c}$ byl co nejmenší. Jedná se tedy o problém hledání nejmenšího pokrytí. $\mathbf{R_c} \cup \mathbf{E}$ je zároveň výsledné pokrytí, které procedura `IRREDUNDANT` vrací.

2.6.6.1 REDUNDANT

Subprocedura `REDUNDANT` rozděluje $\mathcal{F}$ na dvě disjunktní podmnožiny $\mathbf{E}$ a $\mathbf{R}$. To tedy znamená, že platí $\mathbf{E} \cup \mathbf{R} = \mathcal{F}$ a zároveň $\mathbf{E} \cap \mathbf{R} = \emptyset$. V množině $\mathbf{R}$ jsou všechny termy, po jejichž odebrání se pokrytí nezmění, takže musí platit, že

$$\mathcal{F} \cup \mathcal{D} - \{\mathrm{t}\} = \mathcal{F} \cup \mathcal{D}, \quad \forall \mathrm{t} \in \mathbf{R}$$

a pro $\mathbf{E}$ musí tedy naopak platit, že

$$\mathcal{F} \cup \mathcal{D} - \{\mathrm{t}\} \neq \mathcal{F} \cup \mathcal{D}, \quad \forall \mathrm{t} \in \mathbf{E}$$

Pokud je tedy term t v $\mathbf{R}$, tak je zřejmé, že musí být pokryt pokrytím $\mathcal{F} \cup \mathcal{D} - \{\mathrm{t}\}$. Stačí tedy pouze zjistit, zda je kofaktor $(\mathcal{F} \cup \mathcal{D} - \{\mathrm{t}\})_\mathrm{t}$ tautologií, což se snadno zjistí pomocí procedury `TAUTOLOGY`. Pokud je tautologií, tak je term t pokryt a vloží se do $\mathbf{R}$. V opačném případě se přidá do $\mathbf{E}$. Takto se rozdělí všechny termy z $\mathcal{F}$.

2.6.6.2 PARTIALY–REDUNDANT

Subprocedura `PARTIALY-REDUNDANT` zjišťuje pro každý term z $\mathbf{R}$, zda je pokryt $\mathbf{E}$. Pokud není je vložen do množiny $\mathbf{R_p}$. Tedy jinak řečeno pro každý term $\mathrm{t} \in \mathbf{R}$ je zjišťováno zda kofaktor $\mathbf{E}_\mathrm{t}$ je tautologie. Pokud není, je term t vložen do $\mathbf{R_p}$. Ostatní termy, které jsou pokryty $\mathbf{E}$, jsou přebytečné a mohou být tedy odstraněny.

[8] Pozor, nejedná se však o nesporné přímé implikanty, ale pouze o relativně nesporné přímé implikanty

2.6.6.3 MINIMAL–IRREDUNDANT

Subprocedura `MINIMAL-IRREDUNDANT` je pravděpodobně nejsložitější částí celého Espressa. Aplikuje speciální heuristiku pro získání minimálního pokrytí z $\mathbf{R_p}$. K jejímu bližšímu popsání by bylo třeba uvést mnoho nových definic včetně rozšíření algoritmu pro zjištění tautologie. To je mimo rozsah tohoto textu.

Stručně by se `MINIMAL-IRREDUNDANT` dala popsat tak, že hledá nejmenší možné pokrytí $\mathbf{R_c}$, pro něž platí že $\mathbf{R_c} \subseteq \mathbf{R_p}$ a zároveň

$$\mathcal{F} = \mathbf{R_p} \cup \mathbf{E} = \mathbf{R_c} \cup \mathbf{E}$$

Všechny tři pokrytí tedy musí být logicky ekvivalentní.

2.6.7 REDUCE

Procedura `REDUCE` zkouší každý term $\mathrm{t} \in \mathcal{F}$ nahradit menším termem, tedy termem s větším počtem literálů. Ten bude značen $\tilde{\mathrm{t}}$ a musí platí, že je pokryt t. Zároveň musí $\mathcal{F}$ po nahrazení termu zůstat zachováno. Z toho je zřejmé, že některé termy nelze zmenšit a v $\mathcal{F}$ zůstanou nezměněny. Důležité je, že nově vzniklé termy nemusí být přímé implikanty. Hlavní význam `REDUCE` je v tom, že může dostat řešení z lokálního optima. Je to tedy krok, který významně vylepšuje celou heuristiku.

Algoritmus začíná seřazením termů. To se dělá z důvodu, že termy jsou postupně nahrazovány. Počáteční seřazení má tedy vliv na konečný výsledek a zároveň i na rychlost provádění algoritmu. Jako nejvýhodnější se ukazuje vložit na počátek největší term a všechny zbylé termy seřadit podle delta vzdálenosti, kterou mají vzhledem k počátečnímu, tedy vybranému největšímu termu.

Hlavním cílem `REDUCE` je, jak už bylo uvedeno, co nejvíce zmenšit každý term z $\mathcal{F}$, přičemž pokrytí musí zůstat zachováno. To znamená, že nově vzniklý term $\tilde{\mathrm{t}}$ vznikne jako průnik původního termu t s termem $\hat{\mathrm{t}}$, který je komplementem všech termů z pokrytí $(\mathcal{F} \cup \mathcal{D} - \{\mathrm{t}\}) \cap \mathrm{t}$. Výpočet $\hat{\mathrm{t}}$ je podrobně popsán v podsekci 2.6.7.1. Pokud je $\hat{\mathrm{t}}$ prázdný, tak $\tilde{\mathrm{t}} = \mathrm{t}$. Algoritmus tedy funguje tak, že se projdou všechny termy z $\mathcal{F}$a každý term t^i je nahrazen nově vzniklým termem $\tilde{\mathrm{t}}^i$.

2.6.7.1 SCCC: Smallest Cube Containing Complement

`SCCC` je subprocedura Espressa pro výpočet termu $\hat{\mathrm{t}}$ z pokrytí $\mathcal{C} \equiv \mathcal{F} \cup \mathcal{D} - \{\mathrm{t}\}$. K tomu je nejprve třeba spočítat kofaktor $\mathcal{C}_\mathrm{t}$, který bude dále označen jako $\mathcal{K}$. Pokud je $\mathcal{K}$ tautologií, je $\hat{\mathrm{t}} = \emptyset$. V opačném případě je třeba zjistit, zda je $\mathcal{K}$ unátní. Pokud ano, tak $\hat{\mathrm{t}}$ lze velmi snadno zjistit. A to tak, že nejprve se najde term p, který je průnikem všech termů z $\mathcal{K}$, a poté se testuje každý literál $\mathrm{p}_j \neq 2$, zda pro některý term $\mathrm{c} \in \mathcal{K}$ platí, že kofaktor $\mathrm{c}_{\mathrm{p}_j}$ je tautologií. Pokud je, tak to znamená, že proměnná x_j je kompletně pokryta pokrytím $\mathcal{C}$ a literál komplementárního termu $\hat{\mathrm{t}}_j$ tak bude nabývat inverzní hodnoty p_j. Pokud $\mathrm{c}_{\mathrm{p}_j}$ není tautologií, tak proměnná x_j není pokryta $\mathcal{C}$ a $\mathrm{p}_j = 2$.

Pokud funkce není unátní, lze použít rekurzivní rozklad. V případě, že je

$$\mathrm{SCC}(\mathcal{K}) = \mathrm{SCCC}(\overline{\mathcal{K}}) \ ,$$

tak platí, že

$$\mathrm{SCCC}(\mathcal{K}) = \mathrm{SCC}(x_j\,\mathrm{SCCC}(\mathcal{K}_{x_j}) + \overline{x}_j\,\mathrm{SCCC}(\mathcal{K}_{\overline{x}_j}))$$

To vychází ze Shannonova kofaktoru.[9].

Výsledný rozklad se udělá tak, že se nejprve vybere nejvíce binátní proměnná, která bude dále označena jako x_j. Výsledný term $\hat{\mathrm{t}}$ je pak určen jako

$$\hat{\mathrm{t}} = x_j\,\mathrm{SCCC}(\mathcal{K}_{x_j}) \sqcup \overline{x}_j\,\mathrm{SCCC}(\mathcal{K}_{\overline{x}_j})$$

Operace $\sqcup$ znamená speciální sjednocení, které je pro termy u a v definováno následovně

u_j \ $\sqcup$	v_j: 0	1	2	$\emptyset$
0	0	2	2	0
1	2	1	2	1
2	2	2	2	2
$\emptyset$	0	1	2	$\emptyset$

, $\quad j \in \{1, \ldots, n\}$

2.6.7.2 Příklad

Jako příklad je dána funkce o čtyřech proměnných, kde $\mathcal{F} = \{\overline{x}_2\overline{x}_4, \overline{x}_3\overline{x}_4, x_1\overline{x}_2x_3\}$, $\mathcal{D} = \emptyset$ a prvním zkoumaným termem je $\mathrm{t} = \mathcal{F}_1 = \overline{x}_2\overline{x}_4$. Pak

$$\mathcal{C} \equiv (\mathcal{F} \cup \mathcal{D} - \{\mathrm{t}\} \cap \mathrm{t} = \{\overline{x}_3\overline{x}_4, x_1\overline{x}_2x_3\} \cap \overline{x}_2\overline{x}_4 = \{\overline{x}_2\overline{x}_3\overline{x}_4, x_1\overline{x}_2x_3\overline{x}_4\}$$

což v maticovém zápisu vypadá následovně

$$\mathbf{M}(\mathcal{C}) = \begin{pmatrix} 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

Kofaktor $\mathcal{K} \equiv \mathcal{C}_{\mathrm{t}}$ je pak

$$\mathbf{M}(\mathcal{K}) = \begin{pmatrix} 2 & 0 & 2 & 2 \\ 2 & 1 & 2 & 1 \end{pmatrix}$$

Z maticového zápisu je dobře vidět, že $\mathcal{K}$ není unátní. Jedinou binátní proměnnou je x_3. Po rozkladu je $\mathbf{M}(\mathcal{K}_{\overline{x}_3}) = \begin{pmatrix} 2 & 2 & 2 & 2 \end{pmatrix}$, tedy tautologie, a $\mathbf{M}(\mathcal{K}_{x_3}) = \begin{pmatrix} 2 & 2 & 2 & 1 \end{pmatrix}$, jež je unátní a v tomto kroku je tedy $\mathbf{M}(\hat{\mathrm{t}}_{x_3}) = \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix}$. Výsledný rozklad tedy vypadá následovně

$$\hat{\mathrm{t}} \;=\; \overline{x}_3 \cdot \emptyset \sqcup x_3\overline{x}_1 \;=\; \emptyset \sqcup \overline{x}_1x_3 \;=\; \overline{x}_1x_3$$

Nahrazující term $\tilde{\mathrm{t}}$ bude mít tvar

$$\tilde{\mathrm{t}} \;=\; \mathrm{t} \cap \hat{\mathrm{t}} = \overline{x}_2\overline{x}_4 \cap \overline{x}_1x_3 = \overline{x}_1\overline{x}_2x_3\overline{x}_4$$

Po jeho nahrazení jsou zároveň všechny termy v $\mathcal{F}$ disjunktní a po proběhnutí `REDUCE` je $\mathcal{F} = \{\overline{x}_1\overline{x}_2x_3\overline{x}_4, \overline{x}_3\overline{x}_4, x_1\overline{x}_2x_3\}$.

[9] Zájemci si mohou důkaz najít v [2, str. 119-120]

2.6.8 LAST–GASP

Procedura `LAST-GASP` je posledním krokem heuristiky v hlavní smyčce. Je to v podstatě poslední pokus o dostání se z lokálního optima. K tomu používá upravenou proceduru `REDUCE`. Změna spočívá v tom, že oproti klasické `REDUCE` nejsou termy $\tilde{t}$ nahrazovány v $\mathcal{F}$, ale místo toho se pouze uloží do nového pokrytí $\tilde{\mathcal{F}}$. V dalším provádění tak nově vzniklé termy nemají vliv na redukci dalších termů z $\mathcal{F}$. To může vést k tomu, že pokrytí $\mathcal{F}$ a $\tilde{\mathcal{F}}$ nemusí být ekvivalentní. Nápravu přinese volání procedury `EXPAND` na nově vzniklé termy. Nakonec je na výsledné pokrytí zavolána procedura `IRREDUNDANT-COVERS`, která může snížit výsledný počet termů.

2.6.9 Ostatní procedury

Espresso obsahuje ještě některé další procedury, které zatím nebyly popsány. V této podsekci bude jenom stručně uveden jejich význam. Většinou se totiž jedná pouze o doplňkové procedury, které slouží ke zrychlení nebo k drobnému vylepšení celé heuristiky. Nejsou proto zcela zásadní.

2.6.9.1 COMPLEMENT

Procedura `COMPLEMENT` slouží k výpočtu komplementu. Algoritmus je založen na tom, že výpočet komplementu pro unátní pokrytí je velmi snadný. Pokud pokrytí unátní není, lze použít Shannonův rozklad a výsledná pokrytí spojit. Základ je tedy podobný jako při výpočtu SCCC v procedure `REDUCE` popsaném v podsekci 2.6.7.1. Mimochodem stejný základ využívá i vylepšený algoritmus pro výpočet tautologie.

Procedura `COMPLEMENT` je však podle všeho použita pouze před začátkem hlavní smyčky pro výpočet $\mathcal{R}$. Pokud však $\mathcal{F}$ a $\mathcal{D}$ obsahuje pouze mintermy, lze pro zjednodušení vypočítat $\mathcal{R}$ jako doplněk do kompletně specifikované funkce.

2.6.9.2 ESSENTIAL–PRIMES

Procedura `ESSENTIAL-PRIMES` může výrazně zrychlit celou heuristiku. Slouží k nalezení nesporných přímých implikantů. Ty pak uloží zvlášť a odstraní je z $\mathcal{F}$. Další procedury tedy s nimi nemusí počítat a mohou tak být prováděny rychleji.

2.6.9.3 MAKE–SPARSE

Procedura `MAKE-SPARSE` má význam jen pro vícevýstupové funkce. Zde je uvedena pouze pro doplnění, protože je součástí Espressa.

Kapitola 3

Bmin – Boolean minimizer

Bmin je implementační částí této diplomové práce. V následující sekci je popsána analýza výběru prostředku použitých v implementaci. Ve zbytku kapitoly jsou pak popsány jednotlivé části aplikace.

3.1 Analýza

Zadané požadavky na aplikaci byly následující:

- Vytvořit aplikaci pro vizualizaci běhu algoritmů minimalizace logických funkcí, konkrétně algoritmů Quine–McCluskey a Espresso.
- Zobrazit průběh výše zmíněných algoritmů v Karnaughově mapě a 3-rozměrné krychli.
- Oddělit výpočetní jádro od grafického rozhraní.
- Nezávislost na platformě.

Nejprve tedy bylo třeba vybrat vhodný programovací jazyk. Z důvodu rychlosti a mých programovacích znalostí jsem zvolil jazyk C++. Navíc jsem se rozhodl pro použití knihovny STL [5]. Ta je standardní součástí programového vybavení každého operační systému a splňuje tak podmínku nezávislosti na platformě. Ostatní podpůrné knihovny, kromě grafické (viz dále), nebyly použity, protože by bylo těžší zaručit zmíněnou nezávislost na platformě.

Podporované platformy byly vybrány pouze dvě, a to Windows a Linux. K testování na ostatních platformách, především Mac OS X, jsem bohužel neměl prostředky. Na každé platformě je trochu odlišný způsob instalace. Více informací, včetně instalačních pokynů, je možné nálezt v dodatku B.

Pro grafickou část aplikace byla zvolena knihovna Qt [17] . Hlavní důvod jejího výběru je spíše historický (viz sekce 1.2). Přesto bych ale knihovnu zvolil, i kdybych s aplikací začínal znovu. Důvodem je především její nezávislost na platformě, otevřenost, obsáhlost a neustálé zdokonalování. Qt totiž patří společnosti Nokia, která do jejího vývoje investuje velké prostředky. Navíc nabízí i svoje vývojové prostředí Qt Creator, které velice usnadňuje práci.

Bylo zároveň třeba vybrat vhodný grafický způsob zobrazení 3-rozměrné krychle. V tomto ohledu bylo možné zvolit pouze knihovnu OpenGL [4], protože knihovna DirectX [10] není multiplatformní. OpenGL je navíc součástí Qt.

Při požadavku na platformě nezávislou aplikaci a použití OpenGL existuje pouze jedna alternativní knihovny místo Qt. Jedná se o wxWidgets [31]. S touto knihovnou mám osobní zkušenosti a i když není špatná, zdaleka nedosahuje kvalit Qt. Především systém událostí ve wxWidgets je poněkud těžkopádný. Navíc chybí *model and view framework*, který je velmi užitečný.

Aby došlo k oddělení výpočetního jádra, byla aplikace rozdělena na tři části:

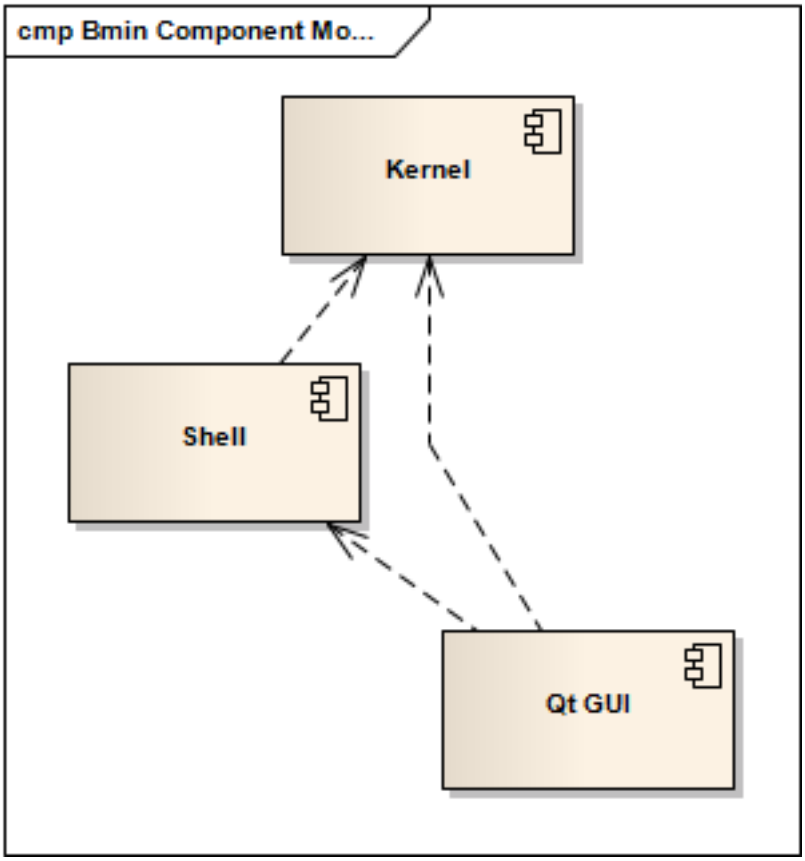
- Hlavní částí je jádro aplikace nebo-li **Kernel**. V něm jsou implementovány oba výše zmíněné minimalizační algoritmy. Dále jsou v něm třídy pro reprezentaci termu a logické funkce včetně speciálního kontejneru pro množinu termů. Jádro nabízí i poměrně propracovaný systém událostí. Součástí jsou struktury pro reprezentaci Karnaughovy mapy, n-krychle a informací z průběhu algoritmu Quine-McCluskey a Espressa.

- Další částí je tzv. **Shell**. Jak název napovídá, umožňuje použití aplikace v příkazovém řádku. Příkazový řádek je však podporován pouze v Linuxu. Důvodem je platformě specifické chování, které je detailněji rozebráno v sekci 3.3.2. Konzolové spouštění však není jediný účel části **Shell**. Jsou v ní totiž zpracovávány uživatelem zadané textové vstupy. Těmi jsou různé příkazy a také zadání funkce ve výčtovém nebo PLA formátu.

- Poslední a nejrozsáhlejší částí je **Qt GUI**. Jedná se o grafické rozhraní, které se stará o veškerou vizuální komunikaci s uživatelem[1]. Je použita knihovna Qt s podporou OpenGL [4], jenž slouží pro vykreslení 3D krychle. Grafické rozhraní má mnoho nejrůznějších funkcí, které budou všechny popsány v sekci 3.4.

Vzájemné závislosti mezi všemi částmi v **Bmin** ukazuje obrázek 3.1. Na něm je vidět, že část **Kernel** je zcela nezávislá. **Shell** závisí pouze na části **Kernel**. Může s ní být zároveň samostatně použita jako konzolová aplikace (samozřejmě pouze v Linuxu). Část **Qt GUI** závisí jak na **Kernel**, tak na **Shell**.

[1] Samozřejmě pokud není brána v úvahu konzolová část.



Obrázek 3.1: Závislost částí **Bmin**

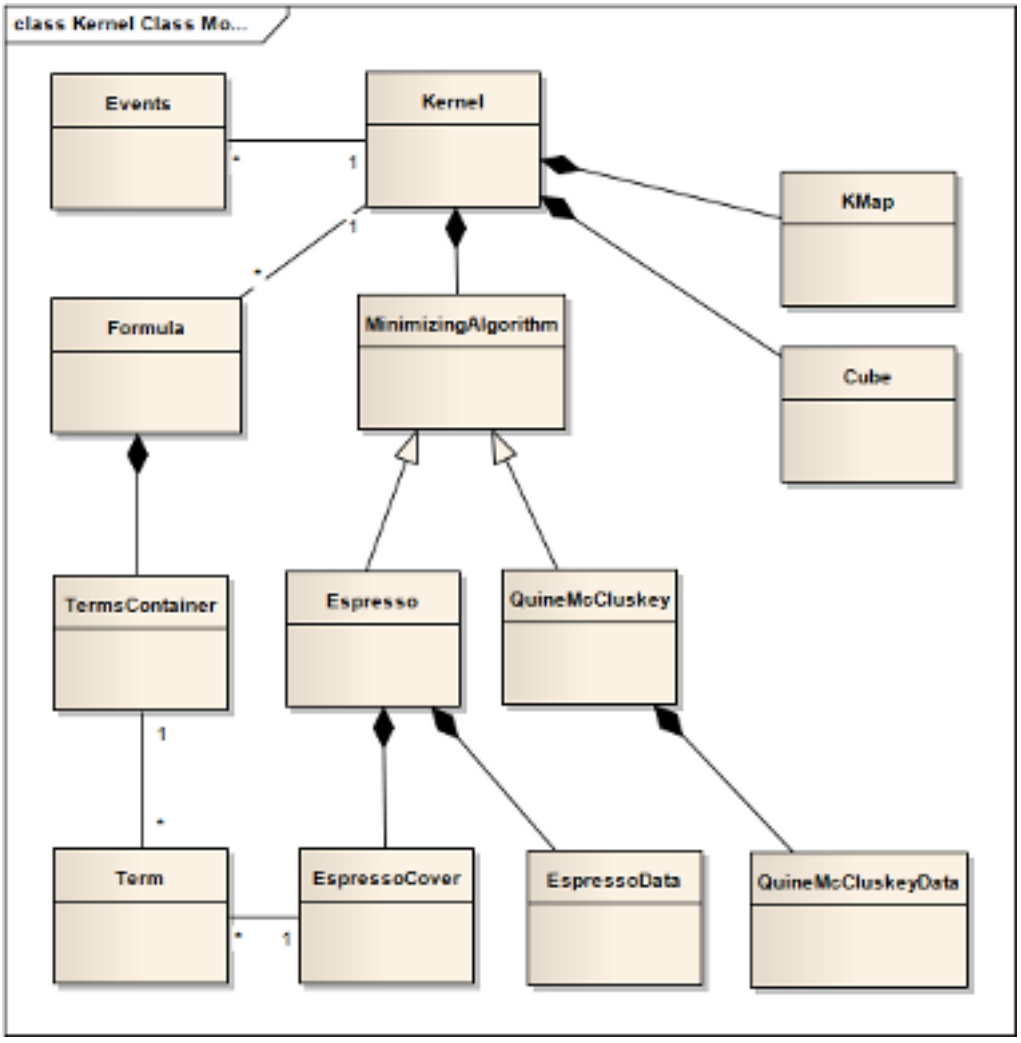
Následující sekce podrobně popisují jednotlivé části.

3.2 Kernel

Kernel je hlavní částí aplikace. Obsahuje všechny zabudované algoritmy pro minimalizace, různé kontejnery, speciální struktury a také vlastní systém událostí. Cílem této kapitoly je popsat architekturu jádra a některé důležité části (především minimalizační algoritmy). Jejich teoretický základ je popsán v kapitole 2. Tato sekce popíše jejich implementační specifika v **Bmin**.

3.2.1 Architektura

Architektura části **Kernel** je poměrně komplikovaná. Zjednodušené rozložení hlavních tříd ukazuje Obrázek 3.2. V tomto UML diagramu nejsou obsaženy některé vedlejší třídy a jsou také vynechány některé nedůležité propojení, které slouží ke zjednodušení implementace.



Obrázek 3.2: Rozložení tříd v části **Kernel**

Hlavní třídou je `Kernel`. Ta existuje pouze v jediné instanci[2] a je zodpovědná za propojení všech ostatních tříd. Slouží tak jako přístupový bod pro ostatní části **Bmin**. Ty mohou buď přímo volat některé metody `Kernel`, nebo mohou podědit třídu `Events` a předefinovat některé její virtuální metody, které pak `Kernel` volá automaticky, když nastane některá událost. Více o událostech je uvedeno v sekci 3.2.2. Třída `Kernel` také obsahuje kontejner pro logické funkce, přičemž vždy pouze jedna je zvolena jako aktuální. Na ní jsou vykonávány všechny požadované operace.

Logická funkce je reprezentována třídou `Formula`. Ta je zodpovědná především za manipulaci s proměnnými. Současný limit je nastaven na **10 proměnných**. Důvodem je hlavně nevyužitelnost většího počtu proměnných ve vizualizačních metodách. Pro uchování termů `Formula` využívá třídu `TermsContainer`. Jedná se o obecný kontejner, který umožňuje přidávání, odebírání a zpřístupňování termů. Navíc má důležitou úlohu při změně reprezentace s DNF na KNF (resp. s KNF na DNF). Automaticky dopočítává $\mathcal{R}$ (resp. $\mathcal{F}$) jako komplement $\mathcal{F} \cup \mathcal{D}$ (resp. $\mathcal{R} \cup \mathcal{D}$). To provádí pouze s mintermy (resp. maxtermy). Implementačně je `TermsContainer` postaven na třídě `std::vector`, ale je napsán tak, aby bylo možné interní kontejner v budoucnu vyměnit. Pro lepší výkonnost se nabízí třída `std::list` nebo vlastní kontejner v podobě ternárního stromu [29]. Samotný term je reprezentován třídou `Term` a bude podrobně popsán v sekci 3.2.3.

Oba minimalizační algoritmy jsou v samostatných třídách. Jedná se o třídy `Espresso` a `QuineMcCluskey`. Obě jsou potomky abstraktní třídy `MinimizingAlgorithm`, která definuje abstraktní metodu `minimize(bool debug)`. Ta je volána při požadavku na minimalizaci. Parametr `debug` určuje, zda je třeba generovat dodatečná data z průběhu algoritmu, která jsou využívána při jeho zkoumání. Třída `MinimizingAlgorithm` zároveň specifikuje, že po skončení minimalizace bude logická funkce uložena jak v úplné, tak v minimalizované formě. Uchovávají se tedy dvě funkce. O výběr a volání algoritmu se stará třída `Kernel`. Detailní popis třídy `Espresso` je v sekci 3.2.5 a třídy `QuineMcCluskey` v sekci 3.2.4

Poslední dvě třídy, které zaslouží bližší popis jsou `Cube` a `KMap`. První jmenovaná slouží, jak už název napovídá, jako interní struktura pro n-krychli. Kromě toho, že zastřešuje aktuální logickou funkci, nabízí také speciální kontejner pro označení vybraných termů jak z minimální, tak z úplné funkce. Třída `KMap` má podobný význam jako `Cube` s rozdílem, že je to interní struktura pro Karnaughovu mapu. Oproti `Cube` je však mnohem propracovanější a obsahuje některé zajímavé prvky. Je jí proto věnována samostatná sekce 3.2.6.

[2] Jedná se tedy o Singleton pattern [28].

3.2.2 Události

Architektura událostí v **Bmin** je postavena na modelu dědičnosti. Pro tyto účely je vytvořena speciální třída `Events`, která obsahuje pouze virtuální metody. Ty slouží jen k předefinování a nic tedy nedělají. Pokud některá z tříd mimo část `Kernel` chce být zapojena do systému událostí, tak stačí, aby podědila třídu `Events` a předefinovala metody, které potřebuje. Napojení na třídu `Kernel` je zajištěno tak, že konstruktor třídy `Events` vloží odkaz na svou instanci do kontejneru s událostmi v `Kernel`. Když pak nastane nějaká událost, `Kernel` volá dané metody všech instancí v kontejneru s událostmi.

Tento způsob je výhodný zejména pokud neexistuje mnoho tříd, které potřebují registrovat pouze jednu nebo několik málo událostí. Potom by docházelo ke zbytečnému volání mnoha prázdných metod. Pro tyto účely je vhodnější architektura založená na callback funkcích [20]. **Bmin** je však vytvořen tak, že dědičnost událostí je pro něj nejvýhodnější.

Část **Kernel** generuje celkem 14 událostí. Následující seznam vypisuje prototypy a popis funkcí pro každou událost.

`virtual void evtFormulaCanged(Formula *f);`
Událost je vyvolána při změně aktuální logické funkce. Ta je také předána jako argument `f`.

`virtual void evtFormulaRemoved();`
Událost je vyvolána při odstranění aktuální logické funkce. Po této události je aktuální funkce prázdná.

`virtual void evtFormulaMinimized(Formula *f, MinimizeEvent &evt);`
Událost je vyvolána po minimalizaci logické funkce. Ta je zároveň předána jako argument `f`. *Druhým argumentem* `evt` *je instance třídy* `MinimizeEvent`, *která obsahuje informace o provedené minimalizaci. Konkrétně jaký algoritmus byl použit, zda se ukládaly dodatečné informace o minimalizaci a některé další informace.*

`virtual void evtMinimalFormulaChanged(Formula *mf);`
Událost je vyvolána po změně minimální funkce, což se děje v podstatě pouze při krokování Espressa. Nová funkce je zároveň předána jako argument `mf`.

`virtual void evtFormulasSet(unsigned idx);`
Událost je vyvolána, pokud byla vybrána nová aktuální logická funkce z kontejneru všech uložených funkcí. Zároveň je jako argument předáno její číslo.

`virtual void evtAlgorithmChanged(Kernel::Algorithm alg);`
Událost je vyvolána při změně algoritmu, jehož typ je zároveň předán jako argument `alg`

`virtual void evtError(std::exception &exc);`
Událost je vyvolána, pokud uživatel provede nějakou nepřípustnou operaci. Těch je poměrně velký počet. Například pokud zadá špatný textový formát funkce, požaduje větší počet proměnných než je povoleno a mnoho dalších.

`virtual void evtExit();`
Událost je vyvolána těsně před ukončením aplikace.

`virtual void evtHelp();`
Událost je vyvolána, pokud uživatel žádá nápovědu.

`virtual void evtShowEspresso(EspressoData *);`
Událost je vyvolána při požadavku na krokování průběhu algoritmu Espresso. Data o průběhu algoritmu jsou předána jako argument `data`.

`virtual void evtShowQm(QuineMcCluskeyData *data);`
Událost je vyvolána při požadavku na zobrazení průběhu algoritmu Quine–McCluskey. Dodatečná data o průběhu algoritmu jsou předána jako argument `data`.

`virtual void evtShowKmap(KMap *kmap);`
Událost je vyvolána při požadavku na zobrazení Karnaughovy mapy. Argument `kmap` *obsahuje interní reprezentaci K-mapy.*

`virtual void evtShowCube(Cube *cube);`
Událost je vyvolána při požadavku na zobrazení Booleovské n-krychle. Argument `cube` *obsahuje interní reprezentaci n-krychle.*

`virtual void evtShowFce(Formula *of, Formula *mf);`
Událost je vyvolána při požadavku na vypsání logické funkce. Argument `of` *představuje funkci v úplném tvaru a* `mf` *v minimalizovaném tvaru. V případě, že minimalizace neproběhla, je* `mf` *rovna* `NULL`.

3.2.3 Term

Třída `Term` slouží k reprezentaci termu. Obsahuje velký počet užitečných metod pro práci s termem včetně operací pro porovnání, produkt, komplement a inverzi. Jsou zde i metody pro výpočet kofaktoru a delta vzdálenosti vzhledem k jinému termu. Lze také zjistit, zda daný term pokrývá některý jiný a případně i vygenerovat všechny mintermy, které daný term pokrývá. Samozřejmě nechybí ani metody pro převod do textového tvaru. Na výběr je jak binární výpis, tak výčtový výpis, v němž jsou uvedeny indexy všech mintermů, které daný term pokrývá.

Vnitřní reprezentace je postavena na dvou číselných maskách. Obě jsou typu `term_t`, což je `typedef` pro `int`. Každé proměnné z logické funkce náleží jeden bit v každé masce. Teoretický maximální počet proměnných je tedy na většině počítačů 31, což o 21 proměnných převyšuje limit daný třídou `Formula`. První maska určuje literály s hodnotou 1. Pozice, kde má bit v masce hodnotu 1 znamená, že literál na stejné pozici má hodnotu 1. Druhá maska je pro literály s hodnotou 2, tedy chybějící literály. Například term $x_1\overline{x}_2x_4$ z funkce o čtyřech proměnných by měl první masku rovnu hodnotě $1001_2 = 9_{10}$. Druhá maska by pak byla rovna $0100_2 = 4_{10}$, protože chybí proměnná x_3, která je na třetí pozici.

Pro reprezenataci literálu je zároveň vytvořena třída `LiteralValue`, která usnadňuje testování jeho hodnot. Dále také definuje jejich textovou podobu. Metody pro získání či uložení hodnoty literálů v třídě `Term` využívají právě třídy `LiteralValue`. Její obdobou pro výstupní hodnoty logické funkce je třída `OutputValue`, která má velmi podobnou strukturu.

Třída `Term` má navíc speciální rozšíření pro **Espresso**. Kromě výše zmíněného výpočtu kofaktoru umožňuje spočítat výsledek operace $\sqcup$ popsané v sekci 2.6.7.1. Významně také usnadňuje expanzi termu podle množiny **L**. Zároveň má metody pro transformaci termu podle v argumentu zadaného termu na řádek blokovací matice nebo matice pokrytí, které jsou použity v proceduře `EXPAND` popsané v sekci 2.6.5.

`Term` má mnoho příznaků. Všechny uvádí Tabulka 3.1. Většina, ale ne všechny, slouží k vykonávání algoritmu **Espressa**. U těch, které slouží pouze pro tyto účely je uvedeno kurzivou v závorce *(Espresso)*.

Příznak	**Význam**
`ONE`	Term náleží do $\mathcal{F}$. Význam má především ve výpisech termu. Rozlišuje se tak, zda se jedná o DNF či KNF. V algoritmech je však většinou ignorován.
`DC`	Term náleží do $\mathcal{D}$. Často používán, pokud se pracuje s jednou množinou termů vzniklou sjednocením $\mathcal{F}$ a $\mathcal{D}$, tedy pokud se pracuje s pokrytím $\mathcal{C} \equiv \mathcal{F} \cup \mathcal{D}$. Umožňuje tak odlišit termy z $\mathcal{D}$.
`INVALID`	Term je neplatný nebo prázdný (neexistuje). Často používán v operacích produkt a kofaktor pro návrat prázdného termu. Term s tímto příznakem tedy představuje symbol $\emptyset$.
`PRIME`	Term je přímým implikantem. *(Espresso)*
`NONESSEN`	Term není nesporný přímý implikant. *(Espresso)*
`ACTIVE`	Term je aktivní. Jedná se o pomocný příznak k odlišení určitého termu. *(Espresso)*
`REDUND`	Term je redundantní. *(Espresso)*
`COVERED`	Term je pokryt nějakým jiným termem v pokrytí. *(Espresso)*
`RELESSEN`	Term je relativní nesporný přímý implikant (viz sekce 2.6.5). *(Espresso)*

Tabulka 3.1: Příznaky třídy `Term`

3.2.4 QuineMcCluskey

Implementace algoritmu Quine–McCluskey je pravděpodobně nejstarší částí v celém **Bmin**. Původně pařila do třídy `Formula`. Postupem času se však osamostatnila a byla pro ni vytvořena nová třída `QuineMcCluskey`, která je potomkem `MinimizingAlgorithm`.

Algoritmus Quine–McCluskey byl podrobně popsán v sekci 2.5. V této sekci bude popsán způsob jeho implementace a dále také způsob uchování informací o jeho průběhu, k čemuž slouží třída `QuineMcCluskeyData`.

3.2.4.1 Implementace algoritmu

Implementace je rozdělena do dvou kroků. Nejprve jsou nalezeny přímé implikanty a poté z nich získáno minimální pokrytí. Ještě před tím jsou vytvořeny dvě instance logické funkce v úplné formě. První se jmenuje `of` a slouží pro uchování originální logické funkce. Druhá se jmenuje `mf` a představuje novou minimální funkci.

Hledání přímých implikantů začíná vytvořením pole polí nebo-li matice STL listů, která představuje matici seznamů popsanou v sekci 2.5. Do seznamů v prvním sloupci matice jsou následně rozděleny minterm (resp. maxtermy) podle počtu jedniček (resp. nul). K průchodu tabulkou slouží dvouúrovňový cyklus, jehož vnější cyklus prochází všechny sloupce a vnitřní všechny řádky, se kterých se vždy vyberou dva sousední seznamy. V nich se termy kombinují každý s každým. Pokud existuje jejich konsensus, tak se přidá do seznamu v sousedním sloupci. Zároveň se z `mf` odstraní oba kombinované termy a přidá se jejich konsensus. Po skončení cyklu jsou v `mf` pouze přímé implikanty. Složitost hledání přímých implikantů je v této implementaci $O(n \cdot m^2)$, kde n je počet proměnných a m je počet mintermů (resp. maxtermů).

Získání minimálního pokrytí je zahájeno vytvořením tabulky pokrytí. Jedná se o matici `bool` hodnot, přičemž každý sloupec reprezentuje jeden minterm (resp. maxterm) a každý řádek jeden přímý implikant. Na pozici, kde přímý implikant implikuje minterm je hodnota nastavena na `true`. Všude jinde je hodnota `false`. Nejprve se z tabulky odstraní nesporné přímé implikanty. To jsou ty, které mají jako jediné v některém sloupci hodnotu `true`. Odstranění se provádí tak, že se nastaví na `false` celý řádek reprezentující odstraňovaný přímý implikant a zároveň se na `false` nastaví i všechny sloupce, ve kterých byla v tomto řádku hodnota `true`. To značí, že jsou dané mintermy pokryty a nepotřebují tedy být pokryty jinými přímými implikanty. Po odstranění nesporných přímých implikantů je třeba vybrat minimální množinu zbylých přímých implikantů. K tomuto účelu je zvolena **heuristika**, která postupně odstraňuje přímé implikanty podle počtu hodnot `true` ve svém řádku. Nejprve jsou odstraněny ty, které mají nejvíce hodnot `true`. To znamená ty, které pokrývají nejvíce ještě nepokrytých mintermů (resp. maxtermů). Při odstraňování se každý přímý implikant uloží do samostatného seznamu, který reprezentuje finální minimální pokrytí. Vhledem k použití heuristiky se ale nemusí jednat o optimální řešení. Pro funkce do 10 proměnných řešení v naprosté většině případů optimální je. To je také důvod výběru heuristiky namísto exaktního řešení.

3.2.4.2 QuineMcCluskeyData

V případě že je metoda `minimize` volána s parametrem `debug` nastaveným na `true`, tak jsou během provádění algoritmu ukládána ještě dodatečná data. K jejich uložení slouží třída `QuineMcCluskeyData`. Jedná se o speciální kontejner. Nejprve dojde k jeho inicializaci. Uloží se počet proměnných a určí se, zda se jedná o DNF či o KNF. V průběhu hledání přímých implikantů se do něj nejprve vloží všechny mintermy (resp. maxtermy) a během smyčky se do něj vkládají všechny nově vzniklé implikanty. K jejich uložení slouží STL vektor obsahující STL listy s termy, což se v C++ zapíše jako `std::vector<std::list<Term> >`. Není tedy již použita matice, místo toho se sloupce a řádky mapují pomocí mapovací funkce, která je zároveň použita i pro tabulku pokrytí. Ta je reprezentována množinou `std::set`. Zároveň jsou do samostatných kontejnerů typu `std::vector` uloženy všechny přímé implikanty a původní mintermy.

Výhodou třídy `QuineMcCluskeyData` je jednoduchý způsob ukládání dat v průběhu minimalizace. Zároveň je možné získaná data rychle a přehledně zpřístupnit jiným třídám, čehož také využívají části **Shell** a **Qt GUI**.

3.2.5 Espresso

Espresso, jehož algoritmus je vysvětlen v sekci 2.6, je v **Bmin** implementováno v poměrně zjednodušené podobě. Úplně chybí všechny procedury uvedené v sekci 2.6.9, které slouží především k rychlostnímu vylepšení heuristiky, což v **Bmin** není příliš podstatné. Zároveň není implementována ani procedura `LAST-GASP`. Pro funkce o malém počtu proměnných by se její účinek vůbec neprojevil. To ale platí i pro proceduru `REDUCE`, která ovšem součástí **Bmin** je. Důvodem je lepší ozřejmění chování algoritmu uživateli, který si může její výsledek prohlédnou při krokování algoritmu. Dalšími implementovanými procedurami jsou `EXPAND` a `IRREDUNDANT`, které jsou nezbytné pro správnou minimalizaci. Následující podsekce uvádějí podrobnější popis jednotlivých částí Espressa v **Bmin**.

3.2.5.1 Hlavní smyčka

Hlavní smyčka je vzhledem k chybějícím procedurám velmi zjednodušena. Protože procedura `LAST-GASP` není implementována, stačí pouze jednoúrovňový cyklus. Ještě před tím než je započat, získají se $\mathcal{F}$, $\mathcal{D}$, $\mathcal{R}$. Ty jsou ve formě mintermů. Poté se ještě inicializují ceny a začíná cyklus. V něm jsou postupně prováděny procedury `EXPAND`, `IRREDUNDANT` a `REDUCE`. Po každé je testována, zda se změnila cena od jejich posledního vykonání. Pokud ano, je cyklus ukončen. Jinak se pokračuje. Po skončení cyklu se z $\mathcal{F}$ vytvoří nová instance třídy `Formula`, která představuje novou minimální funkci. Vše je pro lepší přehlednost zapsáno formou pseudokódu na obrázku 3.3. Z něj je patrné, že smyčka v **Bmin** se o proti smyčce uvedené v sekci 2.6.1 výrazně liší. Možná by šlo říci, že se vlastně nejedná o Espresso. Je ale třeba podotknout, že výsledná sekvence procedur bude pro velkou většinu funkcí do 10 proměnných stejná (samozřejmě s vynecháním `LAST-GASP`).

```
ESPRESSO(F,D) {
  R ← COMPLEMENT(F,D)      // Vypočte komplement (počítáno z mintermů).
  c1 ← c2 ← c3 ← COST(F)   // Inicializuje ceny.
  LOOP:
    F ← EXPAND(F,R)        // Najde přímé implikanty.
    if (c1 = COST(F))      // Pokud se cena nezměnila
      goto QUIT            // ukončí průběh
    c1 ← COST(F)           // jinak cenu uloží.
    F ← IRREDUNDANT(F,D)   // Najde minimální pokrytí.
    if (c2 = COST(F))
      goto QUIT
    c2 ← COST(F)
    F ← REDUCE(F,D)        // Dostává řešení z lokálního optima.
    if (c3 = COST(F))
      goto QUIT
    c3 ← COST(F)
    goto LOOP
  QUIT:
    return Formula(F)
}
```

Obrázek 3.3: Smyčka Espressa v **Bmin**

Pro reprezentaci pokrytí je vytvořena speciální třída `EspressoCover`, která slouží jako kontejner pro termy, k čemuž interně používá `std::list<Term>`. Kromě základních metod pro zjištění počtu termů, přidání a odebrání termu, nabízí ještě speciální iterační makro `foreach_cube`, které velmi zpřehledňuje zápis při iteraci přes všechny termy v pokrytí. Definuje také specifické metody pro třídění a mazání podmnožiny termů, které mají zapnutý konkrétní příznak. Má také metodu pro zjištění unátnosti pokrytí a následné vybrání nejvíce binátní proměnné. Samozřejmě nechybí ani výpočet ceny daného pokrytí.

Pro krokování algoritmu je vytvořena třída `EspressoData`. Ta je propojena s `Kernel`, která umožňuje dočasně změnit aktuální minimální funkci. Toho také přesně `EspressoData` využívá a při každém kroku je aktuální minimální funkce nahrazena funkcí, která byla aktuální v daném kroku minimalizace. Každý krok je dán vykonáním jedné procedury. Vše funguje tak, že během provádění hlavní smyčky se po každém vykonání nějaké procedury Espressa testuje, zda je zapnuta volba `debug`. Pokud ano, tak se z aktuálního pokrytí $\mathcal{F}$ vytvoří nová instance třídy `Formula`, která se uloží do kontejneru v `EspressoData`. Při krokování jsou pak už jen vybírány uložené instance z kontejneru.

3.2.5.2 EXPAND

Procedura `EXPAND` je vykonávána metodou `expand`. Ta prochází pomocí makra `foreach_cube` všechny termy z $\mathcal{F}$, které nejsou přímými implikanty a zároveň nejsou pokryty. To se určuje pomocí příznaků, které jsou při práci s pokrytím často využívány. Na každý term se pak volá metoda `expand1`, která expanduje term přesně jak bylo popsáno v sekci 2.6.5. Matice **B** a **C** jsou vytvořeny jako pokrytí `EspressoCover`. Využívají tak rozšíření třídy `Term`, která může sloužit i jako řádek těchto matic. Množiny **L** a **R** jsou implementovány jako bitové masky typu `term_t`, což velmi usnadňuje práci. Navíc třída `Term` má v sobě zabudovanou metodu, která podle masky `term_t` automaticky expanduje term.

Pokud zůstanou v **B** některé sloupce a je třeba najít minimální sloupcové pokrytí, tak je zvolena zjednodušená heuristika, která není součástí Espressa. Jedná se o postupné odstraňování sloupců s největším počtem jedniček. To se dělá tak dlouho, dokud odstraněné sloupce nepokrývají všechny zbylé řádky v **B**.

3.2.5.3 IRREDUNDANT

Procedura `IRREDUNDANT`, jež je popsána v sekci 2.6.6, se vykonává v metodě `irredundant`. Ta po sobě volá metody `redundant`, `partialyRedundant` a `minimalIrredundant`, které přesně odpovídají procedurám `REDUNDANT`, `PARTIALY-REDUNDANT` a `MINIMAL-IRREDUNDANT`.

Pro odlišení jednotlivých množin **E**, **R**, $\mathbf{R_c}$ a $\mathbf{R_p}$ jsou používány příznaky termů. To má velkou výhodu, protože není třeba neustále kopírovat termy mezi různými instancemi třídy `EspressoCover`. Místo toho stačí pouze jedna instance `EspressoCover`, která zároveň reprezentuje pokrytí $\mathcal{F}$.

V metodách `redundand` a `partialyRedundant` se pro zjišťování, zda je pokrytí tautologií, používá metoda `tautology`. Ta funguje přesně, jak je popsáno v sekci 2.6.4. Není tedy rozšířena na speciální krok pro unátní funkce.

Implementace procedury `MINIMAL-IRREDUNDANT` v metodě `minimalIrredundant` je velmi zjednodušena. S algoritmem, který je součástí Espressa nemá nic společného. Jedná se o velmi primitivní heuristiku, která nedává příliš optimální výsledek. Funguje tak, že postupně odstraňuje termy v pořadí v jakém jdou zrovna po sobě. To děla tak dlouho, dokud není nové minimální pokrytí vzniklé z odstraněných termů ekvivalentní původnímu pokrytí. Hlavním důvodem takto neoptimálního řešení je především složitost procedury `MINIMAL-IRREDUNDANT` a nedostatek času pro její implementaci. Dalším důvodem je však také to, že pro funkce o menším počtu proměnných se ve většině případu neoptimálnost `MINIMAL-IRREDUNDANT` neprojeví. Často na ní buď vůbec nedojde (nezbudou žádné redundantní termy), nebo je třeba rozhodovat mezi stejně velkými termy, což konečný výsledek nijak neovlivní.

3.2.5.4 REDUCE

Implementace procedury `REDUCE` odpovídá, až na drobné změny, popisu v sekci 2.6.7. Hlavní část se nachází v metodě `reduce`. Vše začíná seřazením termů v $\mathcal{F}$ a dočasným připojením $\mathcal{D}$ k $\mathcal{F}$. Poté jsou nastaveny počáteční hodnoty příznaků pro všechny termy. Těch je znovu využíváno pro usnadnění práce s pokrytím. Následné procházení termů funguje přesně tak, jak je popsáno v sekci 2.6.7.

Pozornost si zaslouží akorát subprocedura `SCCC`. Konkrétně její implementace pro případ unátnosti zkoumaného pokrytí $\mathcal{C}$. Ta je oproti popisu v sekci 2.6.7.1 trochu upravena. Nejdříve je spočítán produkt všech termů v $\mathcal{C}$ (ten bude dále značen jako p). Pro každou pozici j se pak testuje, zda existuje v $\mathcal{C}$ nějaký term t^i, který po nastavení $t^i_j = 2$ bude tutologií[3]. Pro j však zároveň musí platit, že $p_j \neq 2$. Oproti popisu v sekci 2.6.7.1 se tedy nehledá kofaktor pro každé $p_j \neq 2$, místo toho se pozice pouze doplňuje do každého termu. Ve výsledku je to ale to samé.

[3] Všechny jeho literály tedy budou mít hodnotu 2

3.2.6 KMap

Třída `KMap` slouží pro reprezentaci Karnaughovy mapy až pro 6 proměnných. Hlavním účelem `KMap` je definovat rozhraní, které umožní mapování jednotlivých termů do mapy. A to jak základních mintermů[4], které představují jedno pole v mapě, tak i termů o menším počtu literálů, které pokrývají více polí.

Mintermy jsou mapovány pomocí Grayova kódu [24]. Pro něj je vytvořena speciální třída `GrayCode`, která obsahuje napevno daných 8 hodnot. To je zcela dostačující, protože mapa může v každém sloupci a řádku pokrývat 3 proměnné. Tím pádem je maximální počet hodnot pro určení kombinací všech proměnných v řádku i sloupci $2^3 = 8$. Výhodou Grayova kódu je jeho symetričnost, která umožňuje jednoduchý převod z řádkového mapování. Pro pozici p na souřadnicích $[x, y]$ v mapě platí, že

$$p \;=\; x + y \cdot m \;=\; (\mathrm{GC}(x)\ \mathbf{shl}\ \log_2(m))\ \mathbf{or}\ \mathrm{GC}(y) \quad ,$$

kde m je počet sloupců, **or** je logický součet, **shl** je bitový posun doleva a GC je funkce pro převod do Grayova kódu. Výpočet pozice každého mintermu v mapě je tedy poměrně rychlý a jednoduchý. Jediný problém nastává při vykreslování mapy pro 6 proměnných. První proměnná se totiž pro přehlednost a lepší vizuální dojem vpisuje do řádku s poslední a předposlední proměnnou. Je tedy třeba první proměnnou přehodit a ve vzorci tak vzniknou určité úpravy, což ve výsledku vypadá následovně

$$p \;=\; x + y \cdot 8 \;=\; (\mathrm{GC}(x)\ \mathbf{shl}\ 3)\ \ \mathbf{or}\ \ (\mathrm{GC}(y)\ \mathbf{and}\ 3)\ \ \mathbf{or}\ \ ((\mathrm{GC}(y)\ \mathbf{and}\ 4)\ \mathbf{shl}\ 3) \quad ,$$

kde **and** je logický součin a slouží tedy jako odmaskování určitých částí. Ve vzorci je navíc nahrazeno m hodnotou 8, protože mapa o 6 proměnných má právě 8 sloupců.

Každé pole v mapě je reprezentováno třídou `KMapCell`, která v sobě obsahuje kromě své pozice také instanci mintermu, který představuje. Pro termy o menším počtu literálů je určena třída `KMapCover`, která slouží jako kontejner pro instance třídy `KMapCell`. Term o menším počtu literálů totiž pokrývá více polí v mapě a právě ty má `KMapCover` ve svém kontejneru. Při jejich vykreslování v části **Qt GUI** se používají barevné obdélníky. V každém poli je tak třeba určit, zda a případně kudy přes něj vede úsečka či úsečky tvořící obdélník. To se dá určit podle poziční sousednosti mintermů v mapě. Kromě pozic zmíněných úseček obsahuje `KMapCell` i další užitečné informace jako je například výstupní hodnota mintermu.

[4] V dalším textu v této sekci bude hovořeno pouze o mintermech, i když úplně stejně by vše platilo i pro maxtermy.

3.3 Shell

Část **Shell** slouží k několika hlavním účelům:

- Zpracování voleb zadaných při spouštění aplikace
- Zpracování zadání logické funkce ve formátu PLA nebo ve vlastním výčtovém formátu
- Vytvoření interaktivního konzolového prostředí s předem určenou skupinou příkazů

V následujících podsekcích bude popsán způsob implementace jednotlivých účelů.

3.3.1 Volby aplikace

Při spouštění aplikace lze zadat několik voleb. Všechny jsou uvedeny v Tabulce 3.2. Volby `-h` a `-v` nebo alternativně `--help` a `--version` mají spíše informační význam. Slouží hlavně k dodržení GNU standardu [15] pro zadávání voleb. Volba `-s` nebo případně `--shell` spouští konzolové prostředí, které bude podrobně popsáno v následující podsekci.

Volbal	Alternativa	Význam
`-h`	`--help`	Zobrazení nápovědy.
`-v`	`--version`	Zobrazení verze včetně informací o licenci.
`-s`	`--shell`	Spuštění konzolového prostředí místo grafického prostředí.

Tabulka 3.2: Volby aplikace

3.3.2 Konzolové prostředí

Konzolové prostředí představuje jakousi terminálovou aplikaci. V současné době je podporována jen pro operační systém Linux. Ve Windows není možné aplikaci spustit, protože se automaticky otvírá v novém procesu. Důvodem je pravděpodobně načtení grafické knihovny, což způsobí vyvolaní nového procesu. Ten nemá připojen vstup a výstup ke konzolovému rozhraní, ze kterého byl vyvolán. Tento problém nebyl řešen z důvodu malé oblíbenosti práce v konzolovém prostředí mezi uživateli Windows.

Po spuštění konzolového prostředí se vypíše několik řádků týkajících se práv a licence. Po nich se vytiskne řádek začínající znakem `>`. Ten uvozuje příkazovou řádku, do níž lze psát příkazy nebo zadat logickou funkci ve specifickém výčtovém tvaru. Každý příkaz nebo zadání funkce se předá ke zpracování po stisknutí klávesy `Enter`.

3.3.2.1 Výčtový zápis funkce

Logickou funkci lze zadat v zápisu, který je v podstatě shodný s klasickým výčtovým zápisem uvedeným v sekci 2.1. Rozdíl spočívá v omezení názvu proměnných pouze na jednopísmenný název. To platí zároveň i pro jméno funkce. Dále jsou nahrazeny matematické symboly za názvy složené z běžných znaků (tzn. symboly vyskytující se na klávesnici). Tabulka 3.3 uvádí všechna nahrazení.

Matematický zápis	Programový zápis	Popis
$\sum_{\mathcal{F}}$	`sum m`	Suma mintermů z $\mathcal{F}$
$\sum_{\mathcal{D}}$	`sum d`	Suma mintermů z $\mathcal{D}$
$\prod_{\mathcal{R}}$	`prod m`	Produkt maxtermů z $\mathcal{R}$
$\prod_{\mathcal{D}}$	`prod d`	Produkt maxtermů z $\mathcal{D}$
$\cdot$	`*`	Spojení výčtů $\prod_{\mathcal{R}}$ a $\prod_{\mathcal{D}}$

Tabulka 3.3: Nahrazení matematických symbolů ve výčtovém zápisu funkce

Funkce v DNF reprezentaci zadaná v matematickém zápisu jako

$$f(c,b,a) = \sum_{\mathcal{F}}(0,2,3) + \sum_{\mathcal{D}}(1,6)$$

se programově zapíše jako

```
f(c,b,a) = sum m(0,2,3) + sum d(1,6)
```

Potom je její KNF reprezentace v matematickém tvaru

$$f(c,b,a) = \prod_{\mathcal{R}}(4,5,7) \cdot \prod_{\mathcal{D}}(1,6)$$

a programově se zapíše jako

```
f(c,b,a) = prod m(4,5,7) * prod d(1,6)
```

Je třeba poznamenat, že výčet pro $\mathcal{D}$ je volitelný a v zápisu se objevit nemusí. První výčet pro $\mathcal{F}$, resp. $\mathcal{R}$, je však povinný a musí být tedy vždy uveden, a to i v případě, že je prázdný.

Kontradikce by se v DNF pro funkci o třech proměnných zapsala jako

```
f(c,b,a) = sum m()
```

Naopak tautologie se v KNF zapíše jako

```
f(c,b,a) = prod m()
```

3.3.2.2 Příkazy

Konzolové prostředí rozeznává několik příkazů. Všechny je uvádí Tabulka 3.4. Speciální formu má příkaz `show`, který bez argumentu zobrazí úplnou formu aktuální funkce. Pokud je zminimalizována, tak ještě zobrazí její minimální formu. V případě, že je příkazu `show` předán jeden ze čtyř v tabulce uvedených argumentů, provede se daná akce. Pro argument `espresso` jsou zobrazena výsledná pokrytí logické funkce po vykonání jednotlivých procedur. Pro ostatní argumenty, kterými jsou `qm`, `kmap` a `cube`, se zobrazí konkrétní akce v podobě obrázku vytvořeným v ASCII Art, což je podrobně popsáno v podsekci 3.3.2.3. Další příkazy, které přebírají argument, jsou `load` a `save`. Je jim předána cesta k souboru, ze kterého bude funkce načtena, resp. do něj uložena v PLA formátu (viz sekce 3.3.3). Cesta musí být zadána ve dvojitých uvozovkách, tedy například `"/home/kuba/fce.pla"`

Příkaz	Popis
`exit`	Ukončí program.
`minimize`	Spustí minimalizaci aktuální logické funkce.
`qm`	Přepne algoritmus na Quine-McCluskey.
`espresso`	Přepne algoritmus na Espresso.
`sop`	Nastaví reprezentaci funkce na DNF.
`pos`	Nastaví reprezentaci funkce na KNF.
`load` *`path`*	Načte funkci ze souboru ve formátu PLA na cestě *`path`*.
`save` *`path`*	Uloží aktuální funkci do souboru *`path`* ve formátu PLA.
`show`	Zobrazí aktuální funkci v úplné a minimální formě.
`show qm`	Zobrazí průběh Quine-McCluskey algoritmu v ASCII Art.
`show espresso`	Zobrazí průběh Espressa.
`show kmap`	Zobrazí Karnaughovu mapu v ASCII Art.
`show cube`	Zobrazí Booleovskou n-krychli v ASCII Art.

Tabulka 3.4: Příkazy pro konzolové prostředí

3.3.2.3 ASCII Art

ASCII Art je název pro obrázek nakreslený pouze pomocí ASCII znaků. Konzolové prostředí používá ASCII art pro vykreslení průběhu Quine–McCluskey algoritmu a také pro zobrazení Karnaughovy mapy a Booleovské n-krychle.

Zobrazení Karnaughovy mapy
K-mapa zobrazuje pouze své rozložení. Není v ní tedy vidět minimální pokrytí. Proměnných může být až 6 a jejich popis je uveden v binárním zápisu. Příklad ukazuje Obrázek 3.4.

```
> f(d,c,b,a) = sum m(0,1,2,3,10,12) + sum d(7,14)
Function was set
> show kmap
   \ ba
  dc\  00  01  11  10
     \---------------
     |   |   |   |   |
  00 | 1 | 1 | 1 | 1 |
     |---|---|---|---|
     |   |   |   |   |
  01 | 0 | 0 | X | 0 |
     |---|---|---|---|
     |   |   |   |   |
  10 | 1 | 0 | 0 | X |
     |---|---|---|---|
     |   |   |   |   |
  11 | 0 | 0 | 0 | 1 |
     |---|---|---|---|
```

Obrázek 3.4: Konzolové zobrazení Karnaughovy mapy

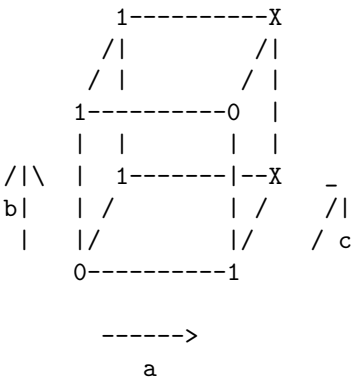
Zobrazení průběhu algoritmu Quine–McCluskey
Při zobrazení algoritmu Quine–McCluskey se nejdříve vygeneruje tabulka obsahující průběh hledání přímých implikantů, která se následně zobrazí. Po ní je zobrazena tabulka pokrytí. Celý průběh i se zadáním funkce ukazuje Obrázek 3.5.

```
> f(c,b,a) = sum m(1,2,4,6) + sum d(5,7)
Function was set
> show qm
Finding Prime Implicants

  Size 1 Implicants                 | Size 2 Implicants | Size 4 Implicants
 -----------------------------------|-------------------|---------------------
  Number of 1s   Minterm   0-Cube   | Minterm   1-Cube  | Minterm      2-Cube
 -----------------------------------|-------------------|---------------------
  1              m1        001      | m(4,6)    1-0     | m(4,5,6,7)   1--*
                 m2        010      | m(4,5)    10-     |
                 m4        100      | m(2,6)    -10*    |
 -----------------------------------| m(1,5)    -01*    |
  2              m5        101      |-------------------|
                 m6        110      | m(6,7)    11-     |
 -----------------------------------| m(5,7)    1-1     |
  3              m7        111      |                   |

Prime Implicants Covering Table

              |1  |2  |4  |6  |
 -------------|---|---|---|---|
  m(1,5)      | X |   |   |   |
 -------------|---|---|---|---|
  m(2,6)      |   | X |   | X |
 -------------|---|---|---|---|
  m(4,5,6,7)  |   |   | X | X |
 -------------|---|---|---|---|
```

Obrázek 3.5: Konzolové zobrazení průběhu algoritmu Quine–McCluskey

Zobrazení Booleovské n-krychle
Booleovská n-krychle může být zobrazena maximálně pro 3 proměnné. Stejně jako K-mapa zobrazuje pouze své rozložení a nejsou v ní zvýrazněna minimální pokrytí. Příklad ukazuje Obrázek 3.6.

```
> f(c,b,a) = sum m(1,2,4,6) + sum d(5,7)
Function was set
> show cube
```



Obrázek 3.6: Konzolové zobrazení Booleovské n-krychle

3.3.3 PLA formát

Berkeley standard PLA format [13] je formát pro zadávání logických funkcí v programu Espresso II[5]. Umožňuje zadat vícevýstupovou funkci pouze v DNF, přičemž má mnoho voleb, které se specifikují v hlavičce. Nicméně **Bmin** podporuje pouze určitou skupinu těchto voleb. Všechny je uvádí Tabulka 3.5

Všechny tyto volby (kromě `.e`) musí být uvedeny v hlavičce, to znamená před zadáním produktových termů (viz dále). Volby `.p`, `.type` a `.e` jsou zcela přebytečné a slouží jen k zachování kompatibility s klasickým PLA formátem. Počet produktových termů se určí automaticky podle počtu uvedených termů a konec je určen jako konec souboru. Logická implementace je podporována pouze pro typ `fd`, který je v klasickém PLA formátu implicitní, takže ho není třeba uvádět.

Přestože **Bmin** podporuje pouze jednovýstupové funkce, je v PLA formátu možné zadat více výstupních hodnot (tedy přesněji více výstupních funkcí). **Bmin** pak dá uživateli na výběr, kterou ze zadaných funkcí zvolí jako aktuální.

Typ `fd` určuje charakter výstupních hodnot. Významy jednotlivých výstupních hodnot uvádí Tabulka 3.6.

Každý produktový term je zadán na jednom řádku. Nejprve jsou zadány vstupní literály, které jsou uvedeny v binárním zápisu rozšířeným o znak `-` pro chybějící literál. Po zadání literálů je třeba vložit alespoň jednu mezeru či tabulátor a poté zadat výstupní hodnoty.

[5]Espresso II je oficiální implementace algoritmů Espressa. Pochází z roku 1983 a jejím autorem je Richard Rudell. Zdrojové kódy jsou k dispozici na [14].

Volba	Význam
`.i` *n*	Počet vstupních proměnných, kde *n* je číslo.
`.o` *n*	Počet výstupních funkcí, kde *n* je číslo.
`.ilb` *seznam*	Názvy vstupních proměnných, kde *seznam* je výčet mezerou oddělených jednopísmenných ASCII znaků abecedy.
`.ob` *seznam*	Názvy výstupních funkcí, kde *seznam* je výčet mezerou oddělených jednopísmenných ASCII znaků abecedy.
`.p` *n*	Počet produktových termů uvedených ve výčtu, kde *n* je číslo.
`.type` *typ*	Nastavuje logickou interpretaci zadaných termů. Parametr *typ* může nabývat pouze hodnoty `fd`.
`.e (.end)`	Volitelně značí konec skupiny zadaných termů

Tabulka 3.5: Volby pro PLA formát

Výstupní hodnota	Význam
`1`	Produktový term náleží do $\mathcal{F}$.
`-`	Produktový term náleží do $\mathcal{D}$.
`0`	Produktový term nemá význam pro danou výstupní funkci.

Tabulka 3.6: Význam výstupních hodnot pro typ `fd`

Ty jsou uvedeny ve stejném zápisu jako literály, přičemž každá hodnota představuje výstup jedné funkce.

Obrázek 3.7 ukazuje příklad zadání funkcí

$$\begin{aligned} f(c,b,a) &\equiv \textstyle\sum_{\mathcal{F}}(2,3) + \sum_{\mathcal{D}}(0,1,7) \\ g(c,b,a) &\equiv \textstyle\sum_{\mathcal{F}}(1,2,3,7) \end{aligned}$$

```
.i 3
.o 2
.ilb e b a
.ob f g
.p 5
000 -0
001 -1
010 11
011 11
111 -1
.e
```

Obrázek 3.7: Ukázka zadání logické funkce v PLA formátu

3.3.4 Vnitřní implementace

Implementačně je část **Shell** složena z několika málo tříd, z nichž některé jsou závislé na části **Kernel**. Závislost na jiné než STL knihovně neexistuje.

Pro konsolové rozhraní slouží třída `Konsole`, která se stará o získání vstupů a předání výstupů uživateli. Je zároveň potomkem třídy `Events`, takže zpracovává všechny události vyvolané jádrem. Veškeré uživatelské vstupy ale předává třídě `Parser`, která podle dané LL gramatiky provádí rekurzivní sestup. Pro lexikální analýzu používá třídu `LexicalAnalyzer`.

Pro vykreslování obrázků v ASCII Art slouží třída `AsciiArt`, která je volána instancí `Konsole` při událostech `evtShowQm`, `evtShowKMap` a `evtShowCube`. Vykreslování n-krychle a K-mapy je poměrně triviální a není třeba ho rozebírat. Vykreslování průběhu algoritmu Quine–McCluskey, konkrétně hledání přímých implikantů, je o něco složitější. Pro tyto účely byla vytvořena třída `Implicants`, která slouží k rozvržení tabulky tak, aby mohla být vykreslena po řádcích a přitom zůstala zachována šířka jednotlivých sloupců.

Zcela nezávislou třídou je `Options`, která slouží k obecnému zpracování voleb. Je to jakási obdoba knihovny `getopt`. Na základě zadaných požadavků zpracovává programové volby.

3.4 Qt GUI

Část **Qt GUI** tvoří grafické rozhraní aplikace **Bmin**. Ke svému fungování potřebuje jak **Kernel** tak **Shell**. Navíc je závislá na grafické knihovně Qt.

Qt GUI je spouštěna automaticky, pokud nejsou programu zadány žádné argumenty. Po jejím spuštění se zobrazí hlavní okno s menu barem, hlavním layoutem a status barem (viz Obrázek 3.8)[6]. Hlavní layout je rozčleněn na dvě části – **horní** a **dolní**:

Horní část slouží k zadání Booleovské funkce v úplné formě a k jejímu zobrazení ve výsledné minimální formě. Funkci lze zadat buď přímo do textového pole ve formátu popsaném v sekci 3.3.2.1 nebo pomocí průvodce pro vytvoření (viz sekce 3.4.1), který se zobrazí po kliknutí na tlačítko `New...`. Pokud je funkce zadána, změní se tlačítko z `New...` na `Edit...` a funkci lze po kliknutí na něj editovat. Dále je možné vybrat reprezentaci logické funkce a minimalizační algoritmus. K výběru reprezentace slouží výběrové pole `Representation of logic function`, kde je na výběr ze `Sum of Products`, což je DNF, a `Products of Sums`, což je KNF. Pro výběr algoritmu je vytvořeno výběrové pole `Algorithm`, které umožňuje vybrat mezi `Quine-McCluskey` a `Espresso`. Ke spuštění minimalizace slouží tlačítko `Minimize`, které zminimalizuje zadanou funkci a výsledek zobrazí do textového pole `Minimal form`. To je pouze pro čtení (nejde do něj tedy zapisovat).

Dolní část obsahuje na levé straně čtyři tlačítka pro přepnutí módu, který je zobrazen vedle panelu s tlačítky. Na každém tlačítku je uveden název módu bílým písmem na šedém pozadí. Pouze tlačítko s aktuálně zobrazeným módem má černé písmo na bílém pozadí. Následující výčet uvádí význam jednotlivých módů

Welcome umožňuje vybrat jiný mód nebo vyvolat About dialog, který zobrazuje základní informace o programu a licenci. Vzhled **Welcome** módu je zobrazen na Obrázku 3.8.

K-Map zobrazuje Karnaughovu mapu (viz sekce 3.4.2).

Boolean n-Cube zobrazuje Booleovskou n-krychli ve 3D (viz sekce 3.4.3).

Quine-McCluskey zobrazuje průběh algoritmu Quine–McCluskey (viz sekce 3.4.5).

V následující podsekcích jsou důkladněji popsány jednotlivé módy (kromě **Welcome**) a také některé zajímavé části.

3.4.1 Dialog pro zadání logické funkce

Logické funkce lze vytvářet a editovat v dialogu, který je zobrazen na Obrázku 3.9. V něm je možné zadat název funkce, její reprezentaci a počet proměnných. U proměnných lze nastavit i jejich názvy, které jsou při každé změně jejich počtu automaticky vygenerovány jako abecední posloupnost začínající písmenem `a`.

Při změně počtu je zároveň vygenerována pravdivostní tabulka, která uvádí v prvním sloupci index mintermu, po němž jsou v následující sloupcích zobrazeny hodnoty jednotlivých

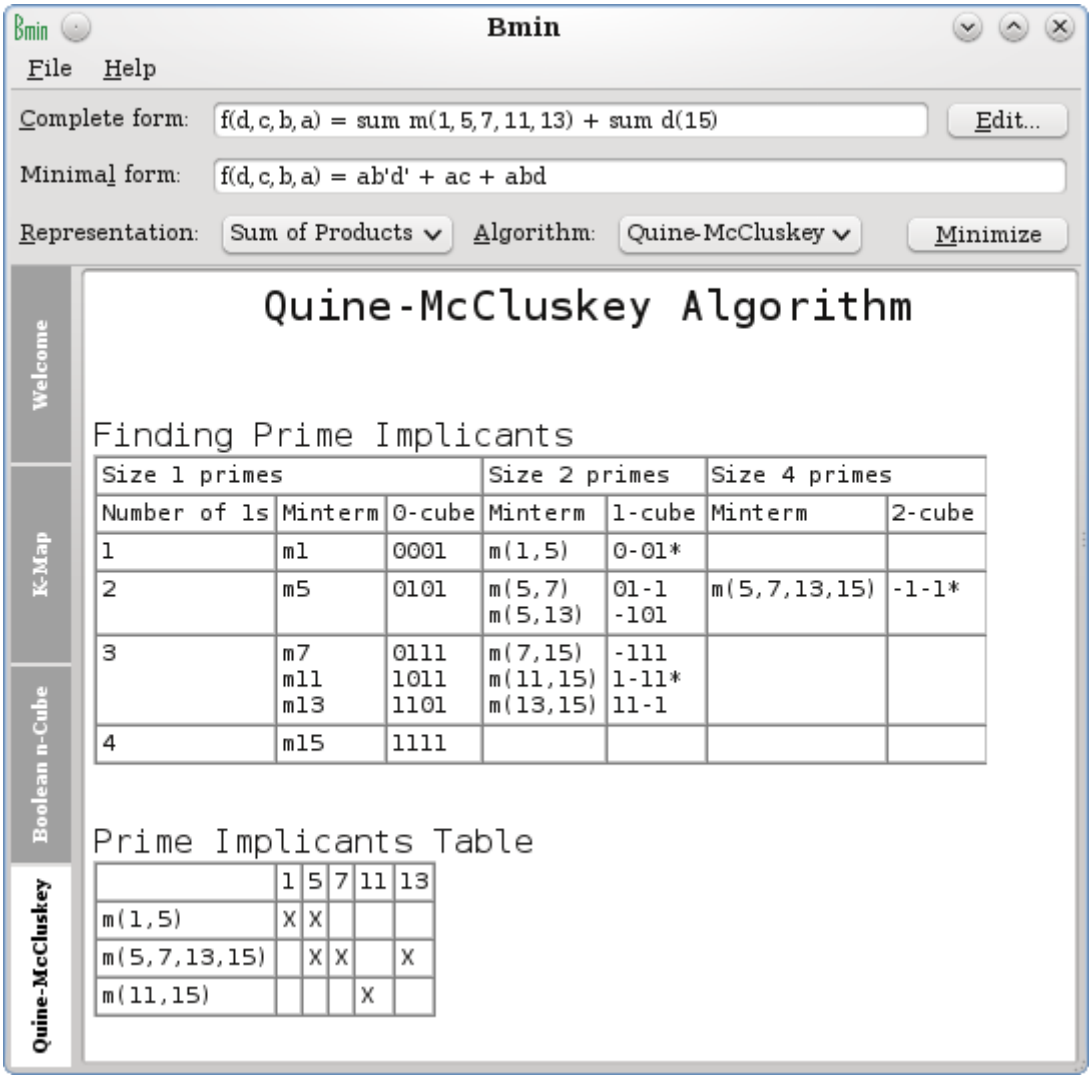
[6] Na obrázku je z důvodu jeho velikosti vynechán status bar.

Obrázek 3.8: **Qt GUI** část po svém spuštění s aktivním **Welcome módem**

proměnných. V posledním sloupci jsou pak uvedeny výstupní hodnoty mintermů. Ty lze měnit dvojím kliknutím[7] na příslušné pole a zvolením výstupní hodnoty z výběrového pole. Další způsob, jak lze měnit výstupní hodnotu, je pomocí kontextového menu vyvolaného kliknutím pravým tlačítkem myši nad vybranou skupinou polí v tabulce a následným zvolením položky `Selected output values to->1`. Lze tak změnit více výstupních hodnot najednou. Kontextové menu obsahuje ještě položku `All output values to`, která umožňuje změnit výstupní hodnotu všem položkám v pravdivostní tabulce najednou.

Pro uložení funkce stačí zmáčknout klávesu `Enter` nebo případně kliknout na tlačítko `Ok`. Pro zrušení dialogu bez uložení funkce naopak stačí zmáčknout klávesu `Esc` nebo kliknout na tlačítko `Cancel`.

[7] Stejného účinku se dosáhne i pouze jedním kliknutím a zmáčknutím klávesy `F2` nebo napsání hodnoty, na kterou má být změněno (např. `1`).

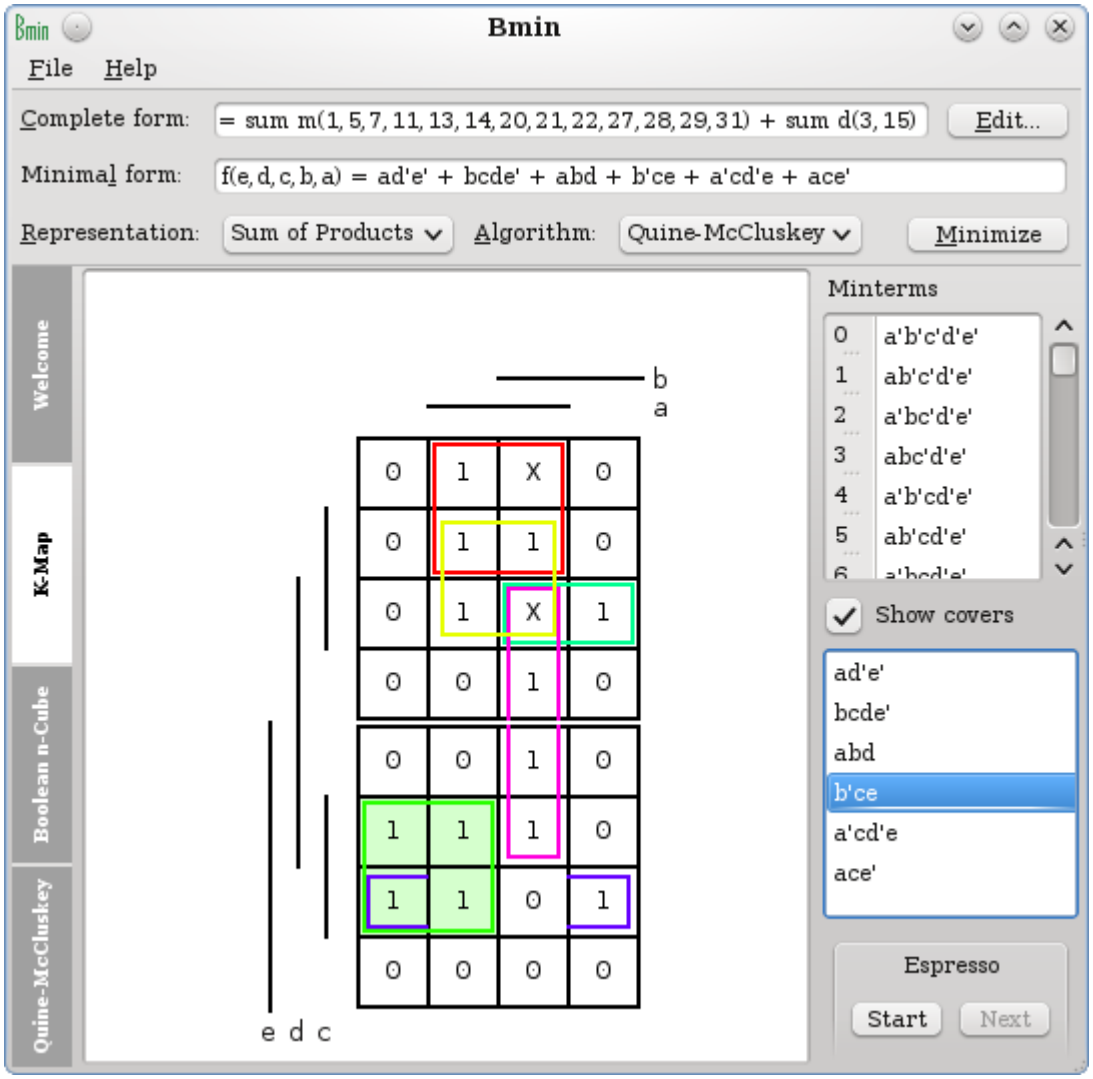


Obrázek 3.9: Dialog pro zadání logické funkce

3.4.2 K-Map

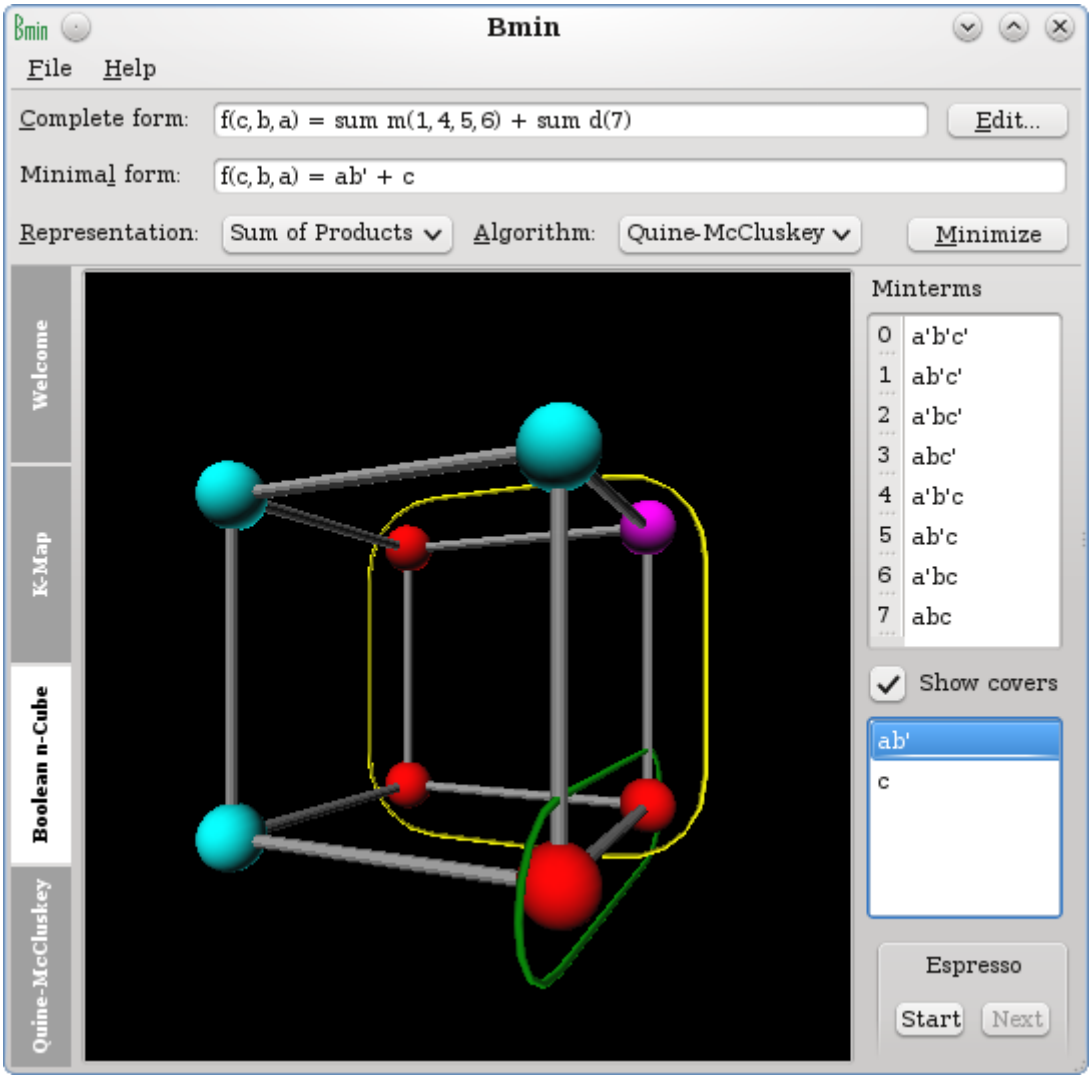
Karnaughova mapa se vykresluje až pro 6 proměnných. Layout je horizontálně rozdělen na dvě části. V levé je mapa vykreslována a v pravé se zobrazují seznamy pro výpis mintermů[8] a pokrytí[9] (ta pouze v případě, že je zaškrtnuto políčko `Show covers`). Mapa se automaticky vykreslí, jakmile je zadána funkce. K popisu proměnných existují dva módy. První ukazuje rozložení v binární formě a druhý jako čáry představující negaci. Módy lze měnit vyvoláním kontextového menu (kliknutím pravým tlačítkem) na popis proměnných. Lze také měnit výstupní hodnoty mintermů zmáčknutím levého tlačítka myši nad příslušným polem v mapě. Změny jednotlivých mintermů se přímo zobrazí v zadání funkce v textovém poli. Mapa je zároveň propojena s výpisem mintermů a pokrytí na pravé straně.

[8] Je uvažována pouze DNF, pro KNF by to bylo stejné, akorát by se jednalo o maxtermy.
[9] Zde je pokrytím myšlen každý term z minimálního pokrytí funkce.



Obrázek 3.10: Karnaughova mapa

Každý vybraný minterm v seznamu změní barvu pole v mapě, jakmile je vybrán, a stejně je tomu i u pokrytí, které zvýrazní všechny mintermy, které jsou jím pokryté. Pokrytí je zároveň v mapě vyznačeno pomocí barevných čtyřúhelníků. Způsob zápisu mintermů a termů minimální funkce (pokrytí) závisí na zvolené reprezentaci. V pravém části pod seznamy jsou ještě dvě tlačítka pro krokování Espressa. Ta budou podrobně popsána v sekci 3.4.4. K-mapu v **Qt GUI** ukazuje Obrázek 3.10.



Obrázek 3.11: Booleaovská n-krychle

3.4.3 Boolean n-Cube

Booleovská n-krychle na Obrázku 3.11 je vytvořena pomocí knihovny OpenGL. Je tedy zobrazena ve 3D prostoru a nabízí dva pohledy. První je klasický 3D pohled, jenž umožňuje procházet vnitřkem krychle. Druhý je tzv. rotační pohled a umožňuje krychlí různě natáčet. Je možné zobrazit maximálně n-krychli pro 3 proměnné. Vrcholy jsou zobrazeny pomocí koulí, přičemž výstupní hodnotu každého mintermu[10] určuje barva koule:

azurová ≡ 0
červená ≡ 1
purpurová ≡ 2

[10] Znovu je uvažována pouze DNF, pro KNF by to bylo stejné, akorát by se jednalo o maxtermy.

Stejně jako u Karnaughovy mapy je layout horizontálně rozdělen na dvě části. V levé je zobrazena výše popsána 3D krychle a v pravé jsou seznamy s mintermy a termy z minimálního pokrytí. Pod nimi jsou ještě dvě tlačítka pro krokování Espressa. Tedy vše je úplně stejné jako u K-mapy. Mintermy se v krychli zvýrazňují pomocí ztmavnutí koulí symbolizujících vybraný minterm. Pokrytí se zobrazují pomocí žlutých smyček obíhajících pokryté mintermy. Vybrané pokrytí změní barvu smyčky na zelenou.

3.4.3.1 Ovládání

Ovládání n-krychle je poměrně obsáhlé, a proto je uzavřeno do samostatné podsekce. Většina akcí se provádí pomocí klávesnice. Navíc lze použít pro některé akce i kontextové menu a myš.

Změna výstupních hodnot jednotlivých mintermů se provádí kliknutím pravým tlačítkem myši. Tím se zároveň změní barva příslušné koule.

Klávesových voleb je velké množství. Je třeba rozlišovat mezi volbami společnými pro oba módy, které jsou uvedeny v Tabulce 3.7, a volbami specifickými pro konkrétní mód, které jsou uvedeny v Tabulkách 3.8, resp. 3.9.

Klávesa	**Význam**
`1`	Přepnutí do módu pro 3D průchod krychlí.
`2`	Přepnutí do rotačního módu.
`L`	Automatický pohyb světel.
`C`	Zobrazení nebo Zakrytí minimalizovaných termů (pokrytí).

Tabulka 3.7: Volby pro Booleovskou n-krychli společné v obou módech

Klávesa	**Alternativa**	**Význam**
`W`	šipka dopředu	Pohyb dopředu.
`S`	šipka dozadu	Pohyb dozadu.
`A`	šipka doleva	Natočení doleva.
`D`	šipka doprava	Natočení doprava.
`E`	`Page Up`	Natočení nahoru.
`X`	`Page Down`	Natočení dolů.
`T`		Pohyb nahoru.
`G`		Pohyb dolů.
`F`		Pohyb doleva.
`H`		Pohyb doprava.
`R`		Zapnutí reflektoru.

Tabulka 3.8: Volby pro Booleovskou n-krychli ve 3D módu

Kontextové menu má dvě položky. První je `Camera`, jenž umožňuje vybrat pohled mezi klasickým 3D pohledem pro průchod krychlí označením `Camer->3D` a rotačním pohledem označením `Camera->Rotated`. Druhou položkou je `Lights`, jenž umožňuje určit, která světla budou svítit. Pro 3D pohled je možné zapnout nebo vypnout zdrojové světlo, kterým je reflektor, označením `Lights->Reflector`. Dále jsou k dispozici dvě směrová světla – přední a zadní. Pro ně jsou v menu volby `Lights->Light 1`, resp. `Lights->Light 2`.

Klávesa	Význam
šipky	Natáčení krychlí.
`+`	Přiblížení krychle.
`-`	Oddálení krychle.
`R`	Automatická rotace.

Tabulka 3.9: Volby pro Booleovskou n-krychli v rotačním módu

3.4.4 Espresso

Espresso je možné krokovat po jednotlivých procedurách. K tomu účelu slouží dvě tlačítka:

`Start` Zahájí krokování algoritmu. Po spuštění se automaticky změní na `Finish`, které slouží k ukončení krokování a zobrazení výsledné minimální funkce. Po ukončení se změní `Finish` znovu na `Start` a je možné spustit krokování od začátku.

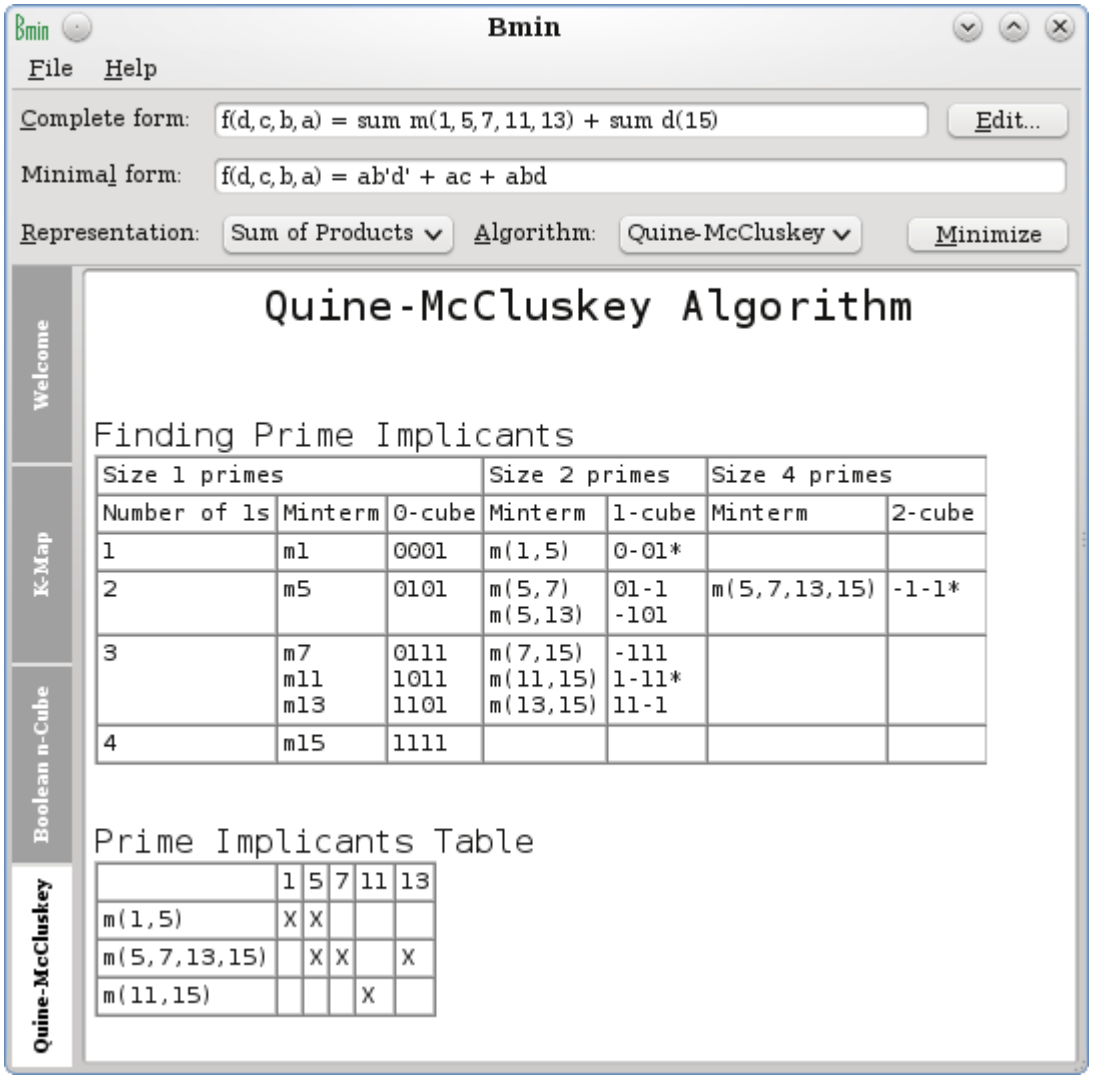
`Next` Zobrazí výslednou funkci po proběhnutí jedné procedury. Vykoná se tedy jedna procedura v pořadí určeném hlavní smyčkou Espressa (viz sekce 2.6) a poté je zobrazena výsledná funkce. Zároveň se ve status baru objeví informace o tom, jaká procedura právě proběhla.

Krokování je možné spustit jak v Karnaughově mapě, tak v Booleovské n-krychli. Průběh je pak nejlépe vidět, pokud je zapnuta volba `Show covers`.

3.4.5 Quine-McCluskey

Mód Quine-McCluskey zobrazuje průběh algoritmu Quine–McCluskey. Nejprve je vypsán průběh hledání přímých implikantů, kde jsou uvedeny termy jak v binárním zápisu, tak ve výčtu pokrytých mintermů. Termy označené hvězdičkou jsou přímými implikanty. Dále je pak uvedena tabulka pokrytí, která je ve standardním formátu. V řádcích jsou tedy uvedeny přímé implikanty a ve sloupcích mintermy z $\mathcal{F}$. Při změně funkce se automaticky mění i obě tabulky. Příklad ukazuje Obrázek 3.12.

Pokud je zvolena KNF reprezentace, jsou zobrazeny výčty maxtermů, které symbolizuje velké písmeno `M` oproti malému `m` pro mintermy. Zároveň jsou samozřejmě ve sloupcích tabulky maxtermy z $\mathcal{R}$.



Obrázek 3.12: Průběh algoritmu Quine–McCluskey v **Qt GUI**

3.4.6 Hlavní menu bar

Hlavní menu bar obsahuje dvě menu, a to `File` a `Help`. Druhé jmenované obsahuje pouze jednu akci, kterou je `About Bmin`. Ta způsobí zobrazení About dialogu, který obsahuje, jak už bylo uvedeno, základní informace o programu a licenci. Zajímavější je první menu, kterým je `File`. To obsahuje několik akcí, které uvádí Tabulka 3.10. Kromě názvu položky a jejího popisu uvádí Tabulka i klávesovou zkratku pro vyvolání dané akce.

Je třeba poznamenat, že při načítaní aktuální logické funkce z PLA souboru může být vytvořeno více výstupních funkcí. V tom případě je uživateli zobrazen dialog, ze kterého si vybere jednu, která se stane aktuální funkcí.

Dále je ještě třeba poznamenat, že uživatelské nastavení, které je využíváno pro akce `Store` a `Load`, je nejčastěji skrytý adresář `config` v uživatelově domácím adresáři, kam se ukládají programová nastavení.

Položka	Klávesová zkratka	Popis
`New...`	`Ctrl+N`	Vyvolá dialog pro vytvoření nové logické funkce.
`Edit...`	`Ctrl+E`	Vyvolá dialog pro editování aktuální logické funkce. Lze provést pouze v případě, že aktuální logická funkce existuje.
`Open...`	`Ctrl+O`	Načte logickou funkci funkci z PLA souboru, který je zvolen pomocí vyvolaného dialogu.
`Save As...`	`Ctrl+S`	Uloží aktuální logickou funkci ve formátu PLA do souboru zvoleného pomocí vyvolaného dialogu.
`Load`	`Ctrl+L`	Načte aktuální logickou funkci uloženou, která je uložena v uživatelském nastavení.
`Store`	`Ctrl+W`	Uloží aktuální logickou funkci do uživatelského nastavení.
`Exit`	`Ctrl+Q`	Ukončí program.

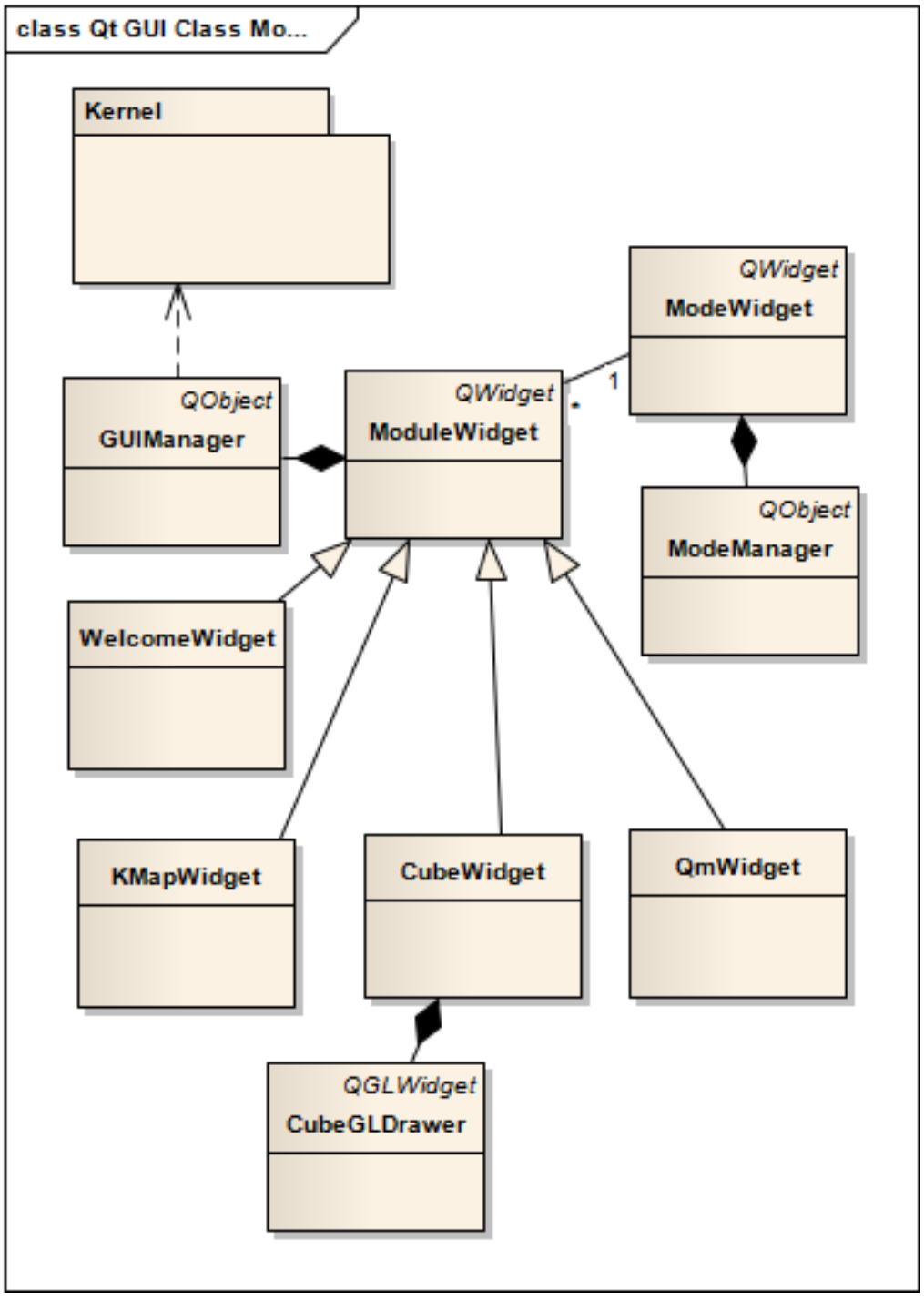
Tabulka 3.10: Položky ve `File` menu

3.4.7 Vnitřní implementace

Vnitřní implementace **Qt GUI** je poměrně komplikovaná. Většina tříd dědí od základních Qt tříd. Komunikace je pak postavena na *signal and slots* frameworku, který v Qt nahrazuje klasický model událostí. V mnoha třídách je také používán *Model and View* framework. Veškeré vykreslování je samozřejmě založeno na Qt třídách. Pro bližší popis by tedy bylo třeba důkladně popsat knihovnu Qt, což je mimo rozsah této práce. Lze však uvést alespoň základní rozdělení tříd.

O veškorou komunikaci s jádrem se stará třída `GUIManager`, která zároveň slouží i jako propojovací vrstva mezi všemi třídami v **Qt GUI**. Existuje proto pouze v jediné instanci.

O spravování módů se stará třída `ModeManager`, která existuje také pouze v jediné instanci. Každý mód je potomkem třídy `ModuleWidget`, která obsahuje základní informace jako jméno módu a jeho ID. Jednotlivé módy jsou pak reprezentovány třídami `WelcomeWidget`, `KMapWidget`, `CubeWidget` a `QmWidget`. Třída `CubeWidget` používá pro OpenGL vykreslování třídu `CubeGLDrawer`, která je zároveň i nejrozsáhlejší třídou v celé aplikaci. Vše je pro přehlednost znázorněno na obrázku 3.13.



Obrázek 3.13: Rozložení tříd v části **Qt GUI**

Kapitola 4

Testování

Bmin je aplikací, jejímž cílem není optimálně a rychle minimalizovat logické funkce. Proto nemá smysl dělat výkonnostní testy. Místo toho je udělán **usability test**, nebo-li test použitelnosti. Jeho účelem je otestovat použitelnost aplikace na uživatelích. Více informací o podstatě usability testu lze najít například v [18].

4.1 Usability test

4.1.1 Cíle testu

Cílů testu je několik:

1. Zjistit, zda uživatel dokáže plně ovládat aplikaci bez použití manuálů.
2. Zjistit, zda uživatel dokáže v aplikaci optimálně používat pravdivostní tabulku.
3. Zjistit, zda uživatel dokáže v aplikaci optimálně používat Karnaughovu mapu.
4. Zjistit, zda uživatel dokáže v aplikaci optimálně používat Booleovskou n-krychli.
5. Zjistit, zda je uživatel schopný pochopit Quine–McCluskey algoritmus z výpisu, jenž je zobrazen v aplikaci.
6. Zjistit, zda je uživatel schopný pochopit Espresso při jeho krokování.

4.1.2 Výběr účastníků

Bylo vybráno celkem 6 testovacích uživatelů. V dalším textu budou označeni jako U1 až U6. Podmínkou výběru byla alespoň základní znalost logických funkcí a jejich minimalizací. Všichni uživatelé prošli nějakým kurzem zaměřeným na logické obvody, a tak každý znal representace logických funkcí a uměl je minimalizovat pomocí Karnaughových map. Pouze uživatelé U2 a U3 znali navíc ještě fungování algoritmu Quine–Mccluskey. O Espressu nikdo z uživatelů nikdy neslyšel.

4.1.3 Parametry testu

Test byl připraven formou dotazníku. Každému uživateli byl poslán emailem jeden dotazník, jehož součástí bylo i zadání čtyř úkolů uvedených v sekci 4.1.4. Uživatelé poté doma, u svého počítače, dotazník vyplnili a poslali zpět. Při dělání úkolů 1 až 3 zároveň měřili, jak dlouho jim trvalo, než daný úkol dokončili. Otázky byly rozděleny do tří skupin:

1. Otázky týkající se znalostí o logických funkcích a minimalizací – odpovědi pouze **ano** nebo **ne**.
2. Otázky týkající se jednotlivých úkolů – výběr jedné z odpovědí.
3. Otázky týkající se celkového dojmu z testu – hodnocení jednotlivých částí, vlastní návrhy a popsání svých dojmů.

Test byl koncipován tak, aby zabral uživateli přibližně půl hodiny.

4.1.4 Úkoly

Pro uživatele byly připraveny čtyři navazující úkoly. Následující seznam uvádí jejich znění. U každého úkolu jsou navíc ještě v závorce vypsány čísla výše uvedených cílů, které daný úkol pokrývá.

Úkol 1: *Zapni program a vytvoř funkci o třech proměnných, která bude mít výstupní hodnotu 1 pro mintermy s indexem 1, 2 a 3. Tedy funkci* $f(c, b, a) = \sum_{\mathcal{F}}(1, 2, 3)$. *Funkci zminamalizuj a změň její reprezentaci na Product of Sums.* (cíl 1 a 2)

Úkol 2: *Nyní zobraz Booleovskou n-krychli. V ní najdi maxterm s indexem 0 (a + b + c) a pak zkus změnit jeho výstupní hodnotu na 1. Poté se projdi skrz krychli, otoč se a podívej se nahoru. Nakonec změň pohled (Camera) na rotační a zarotuj s krychlí.* (cíl 1 a 4)

Úkol 3: *Nyní změň počet proměnných v logické funkci na čtyři, nastav reprezentaci na Sum of Products a všechny výstupní hodnoty na 1. Dále zobraz Karnaughovu mapu. Změň výstupní hodnotu pro mintermy 0 (a'b'c'd'), 1 (ab'c'd'), 3 (abc'd'). Následně zobraz pokrytí a jedno z nich zvýrazni. Dále se pokus změnit popis mapy (legendu, která označuje negace proměnných) na binární zápis proměnných.* (cíl 1, 2 a 3)

Úkol 4: *Nejprve zadej do pole* `Complete Form` *tuto funkci:* `f(d,c,b,a) = sum m(1,5,7,8,9,10,11,14,15)` *(zkopíruj tam toto zadání). Potom přepni na Quine-McCluskey a koukni na průběh algoritmu. Pak přepni na Karnaughovu mapu, zaškrtni* `Show covers` *a spusť Espresso krokování (tlačítko* `Start`*). Poté mačkej tlačítko* `Next` *a pozoruj, jak se mění pokrytí.* (cíl 5 a 6)

4.1.5 Analýza výsledků

V následující analýze jsou výsledky rozděleny do skupin podle zadaných úkolů. Navíc je přidána analýza testu z celkového dojmu. Na závěr jsou všechny výsledky shrnuty.

4.1.5.1 Úkol 1

Při provádění Úkolu 1 postupovali téměř všichni uživatelé stejně. Nejprve klikli na tlačítko `New...` poté výstupní hodnotu změnili tak, že dvakrát klikli na pole s výstupní hodnotou a z výběrového pole vybrali hodnotu 1. Nakonec klikli na tlačítko `Minimize` a ve výběrovém poli v hlavním layoutu změnili reprezentaci funkce. Jedinou výjimku tvořil uživatel U4, který změnil výstupní hodnotu v pravdivostní tabulce pomocí kontextového menu.

Splnění všech úkolů tedy proběhlo optimální cestou a všichni vše splnili, což znamená, že k vytvoření funkce není třeba manuál.

Naměřené časy ale nebyly optimální pro polovinu uživatelů, kterým trvalo splnění úkolu déle jak 2 minuty. Uživatelům U1 a U2 trvalo splnění kolem 3 minut. Uživatel U4 dokonce strávil nad úkolem déle než 5 minut, což podle jeho slov bylo způsobeno hledáním způsobu změny výstupní hodnoty v pravdivostní tabulce.

4.1.5.2 Úkol 2

V Úkolu 2, který byl zaměřen na Booleovskou n-krychli, všichni uživatelé našli požadovaný maxterm pomocí postranního seznamu. Pohyb po krychli zvládli bez problémů uživatelé U1, U2, U3. Uživatelé U4 a U5 se akorát nedokázali podívat nahoru a U6 vůbec netušil, jak se v krychli pohybovat. Zároveň jako jediný nedokázal změnit hodnotu maxtermu přímo v krychli kliknutím na příslušnou kouli. Vše zvládl až po použití manuálu. Následné přepnutí do rotačního pohledu nedělalo nikomu problém. Všichni použili kontextové menu. Rotování s krychlí zvládli všichni pomocí šipek. Jedinou výjimkou byl znovu uživatel U6, který musel použít manuál.

Časy se téměř u všech uživatelů pohybovaly pod 3 minuty. Jedinou výjimkou byli U5, jemuž úkol trval 4 minuty, a U6, který vzhledem k použití manuálu čas neuvedl.

4.1.5.3 Úkol 3

V Úkolu 3 nejprve uživatelé měnili všechny hodnoty v logické funkci. Pouze polovina z nich použila optimální řešení, kterým bylo vybrání akce `All output values to` v kontextovém menu pravdivostní tabulky. Uživatel U1 měnil hodnoty po jedné a uživatelé U3 a U4 vše

změnili až Karnaughově mapě, což byla zároveň další část úkolu, kterou zvládli všichni bez problémů. I zobrazení pokrytí a změna popisu mapy proběhla bez jakýkoliv potíží.

Uživatelé U2 a U5 zvládli úkol vypracovat v čase pod 3 minuty. Zbylí uživatelé to zvládli do 4 minut. To jsou poměrně optimální časy.

4.1.5.4 Úkol 4

Úkol 4 byl zaměřen na minimalizační algoritmy. Čas zde měřen nebyl. Šlo především o zjištění, zda je uživatel schopen pochopit algoritmus, jehož průběh je v aplikaci zobrazen.

U algoritmu Quine–McCluskey byly odpovědi jednoznačné. Uživatelé U2 a U3, kteří algoritmus alespoň částečně znali, napsali, že jim přišel přehledný. Ostatní pak uvedli, že z jho průběhu vůbec nepochopili, jak funguje.

Uživatelé U2, U3, U4 a U6 při krokování Espressa alespoň trochu pochopili podstatu jednotlivých procedur. Ostatní, tedy U1 a U5, uvedli, že průběh krokování Espressa vůbec nepochopili.

4.1.5.5 Hodnocení a dojmy uživatelů

V poslední části testu měli uživatelé ohodnotit určité části aplikaci na stupnici od 1 do 5, kde 1 znamená *nejlepší* a 5 *nejhorší*. Výsledná hodnocení uvádí Tabulka 4.1. Zpracování průběhu algoritmu Quine–McCluskey hodnotily pouze uživatelé, kteří ho znali.

	U1	U2	U3	U4	U5	U6	Průměr
Grafický vzhled	1	2	2	1	2	1	1.5
Layout aplikace	1	1	2	2	2	1	1.5
Ovladatelnost	2	1	1	2	2	2	1.7
Zpracování pravdivostní tabulky	1	3	3	2	3	2	2.3
Zpracování Karnaughovy mapy	1	1	1	1	1	1	1.0
Zpracování Booleovská n-krychle	2	1	1	1	1	2	1.3
Zpracování průběhu Quine–McCluske	-	2	3	-	-	-	2.5
Celkový dojem	1	2	2	1	2	1	1.5

Tabulka 4.1: Hodnocení jednotlivých částí aplikace

Uživatelé na závěr ještě popsali své dojmy a navrhly některá vylepšení. Všem se aplikaci líbila a každý vyzdvihl především zpracování Karnaughovy mapy. Uživatelé U2, U3, U4 a U5 napsali, že by vylepšili zpracování pravdivostní tabulky. Některá navrhovaná vylepšení uvádí následující seznam:

- Zvýrazňovat mintermy v postranním seznamu při změně jejich hodnot v K-mapě a Booleovské n-krychli.
- Pravdivostní tabulka by měla jasně oddělit a zvýraznit, která pole jsou editovatelná a která informační. Sloupeček indexu by měl být lépe oddělený od sloupců nul a jedniček.
- Espresso i Quine Mccluskey algoritmy by měly mít asociovanou nápovědu (třeba okno s popisem), která by byla jednoduše vyvolatelná.

- Přidat do krokování Espressa tlačítko zpět.
- Zkusit zakomponovat nějaký background image.
- Zpřehlednit přidáním ikonek.
- Rotaci u krychle pomocí myši.
- Při minimalizaci počítat s hazardy.

4.1.5.6 Shrnutí

Vzhledem k získaným výsledkům se zdá, že aplikaci lze poměrně bez problému ovládat bez použití manuálu. Jedinou výjimku tvoří Booleovská 3D n-krychle, kde ne každý dokázal odhalit klávesové zkratky.

Použitelnost pravděpodobnostní tabulky není příliš optimální. Většině uživatelů nebyl jasný způsob ovládání, který pro ně nebyl na první pohled zřejmý.

Karnaughova mapa byla ovládána zcela optimálně. Všichni na ní dokázali provést úkoly poměrně rychle a bez problému.

Booleovská n-krychle byla poměrně optimálně používána. Někteří uživatelé však nebyli schopni zvládnout všechny pohyby bez nápovědy.

Co se týče průběhu algoritmu Quine–McCluskey, tak ten byl pro všechny uživatele naprosto nepochopitelný. Espresso dopadlo lépe. Většina uživatelů ho při jeho krokování alespoň trochu pochopila.

4.1.6 Zjištěné problémy a návrh jejich řešení

Hlavním problémem je zpracovaní pravdivostní tabulky. Měla by být zpřehledněna a jasně určit, které hodnoty lze měnit. To bohužel není příliš jednoduchý úkol. Je třeba třeba podědit Qt třídu pro zobrazení tabulky a předefinovat její vykreslování. Další možností je tabulku vykreslovat pomocí jiných tříd, což by mohlo být jednodušší, ale ne tak efektivní.

Dalším problémem je ovládání Booleovské n-krychle. Nejlepším řešením by mohla být rychlá nápověda při prvním zobrazení krychle. Následně by byla vyvolatelná z ovládacího panelu.

Nápověda by byla vhodná i u zpracování obou algoritmů, které byly pro některé uživatele málo pochopitelné.

Kapitola 5

Závěr

Pro splnění zadání této diplomové práce bylo třeba napsat poměrně obsáhlou aplikaci. Vizualizace minimalizace logických funkcí je implementačně velmi náročná. Je nutné implementovat minimalizační algoritmy a poté ještě vytvořit grafické rozhraní. Grafické knihovny ale nenabízí příliš vhodné prostředky pro vykreslení vizualizačních metod. Bylo tedy nezbytné vše postavit na základních třídách a ty poté upravit. Nejsložitější a časově nejnáročnější částí byla jednoznačně implementace Booleovské n-krychle ve 3D prostoru. Zaručit kompatibilitu mezi platformami nebyl jednoduchý úkol, a to i přesto, že OpenGL je standard, který by měl fungovat všude stejně. Realita ovšem taková není. Liší se nejen mezi platformami, ale i mezi některými instalacemi Windows. Krychli bylo tedy potřeba několikrát značně upravit a nakonec byly pro lepší stabilitu úplně vypuštěny textury.

Celkově myslím, že aplikace splnila všechny požadavky zadání. Přesto jsem si vědom, že některé části mohli být splněny lépe. Především nejsem úplně spokojen se zobrazením průběhu obou algoritmů. Generování průběhu Quine–McCluskey je pomalé a vzhledově nevyhovující. To je způsobeno výběrem základní třídy, na které je vykreslování postaveno. Co se týče Espressa, mělo by asi dostat větší prostor. V rozvržením však dostaly přednost uživatelsky více použitelné součásti jako jsou seznamy pro výběr termů. Na druhou stranu jsem velmi spokojen s výsledným zpracováním Karnaughovy mapy, kterou všichni testovací uživatelé ohodnotili velice kladně. I zpracování Booleovské n-krychle se dle mého názoru povedlo. I když by možná bylo dobré trochu vylepšit její ovládání. Co se týče konzolové části, je třeba uvést, že je to pouze rozšíření aplikace, které není součástí zadání. To je také důvod, proč jsem se hlouběji nezabýval jejím zprovozněním ve Windows. Osobně si nemyslím, že by ji někdy nějaký uživatel Windows spustil, proto by to bylo asi i zbytečné úsilí.

Textová část práce se zaměřila především na minimalizační metody a jejich popis. Veškeré teoretické základy uvádí kapitola 2. V kapitole 3 byl pak podrobně popsán program, který byl v následující kapitole 4 otestován na uživatelích. To bylo velmi přínosné, protože bylo ozřejmeno mnoho drobných nedostatků. Textová část zároveň může sloužit i jako manuál k aplikaci, což byl jeden z mých cílů.

Hlavním mým cílem bylo dostat původní aplikaci, která fungovala jen pod Linuxem, do stavu použitelnosti na všech počítačích. To se myslím povedlo, i když bude třeba ještě udělat mnohá vylepšení, než bude aplikace rozšířena mezi uživatele. Plán budoucího vývoje uvádí následující sekce.

5.1 Budoucí vývoj

První věc, kterou bude třeba udělat je instalátor. V Linuxu bude stačit udělat soubor `configure`, který by měl ověřit všechny závislosti. Ve Windows bude třeba najít vhodný instalační program, který automaticky provede instalaci za uživatele. Po zabudování instalátoru by okamžitě měla vyjít verze 1.0.0, která by se měla objevit na Google Code [16].

Ve verzi 1.1.0 by měla být kompletně změněna pravdivostní tabulka a vykreslení průběhu algoritmu Quine–McCluskey. To by mělo být společně s nově přidanou nápovědou postaveno na knihovně QtWebkit. V plánu je také vytvoření rozšiřovatelného plovoucího layoutu a přidání obrázků pro lepší grafický vzhled. Poslední věcí je ukládání sezení (sessions), které by mohlo být načítáno při dalším spuštění.

V budoucnu by pak mohlo dojít ještě k rozšíření a zabudování shellu do grafické aplikace, tedy k vytvoření nějakého nového okna s příkazovou řádku. Mohlo by dojít i k rozšíření na vícevýstupové funkce. To by si ale vyžádalo poměrně mnoho velkých změn v jádru, což v blízké době není moc pravděpodobné.

Další vývoj bude především záležet na tom, zda se aplikace rozšíří mezi velký počet uživatelů. Také bude dost podstatné, zda se někdo zapojí do jejího vývoje, protože sám už pravděpodobně nebudu mít tolik času.

Literatura

[1] G. Boole. *An Investigation of the Laws of Thought.* Project Gutenberg: http://www.gutenberg.org/etext/15114, 1854.

[2] R. K. Brayton, G. D. Hatchel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publisher, 1984.

[3] S. J. Hong, R. G. Cain, and D. L. Ostapko. *MINI: A heuristic approach for logic minimazition*, volume 18. IBM J. of Res. and Dev., 1974.

[4] M. Segal and K. Akeley. *The OpenGL Graphics System*, 4.0 edition, Mar. 2010.

[5] A. Stepanov and M. Lee. *The Standard Template Library*, Nov. 1995.

[6] Biografie: George Boole. http://plato.stanford.edu/entries/boole/, stav z 22. 4. 2010.

[7] Biografie: Mourice Karnaugh. http://en.wikipedia.org/wiki/Maurice_Karnaugh, stav z 22. 4. 2010.

[8] Biografie: Edward J. McCluskey. http://en.wikipedia.org/wiki/Edward_J._McCluskey, stav z 22. 4. 2010.

[9] Biografie: Willard Van Orman Quine. http://www.wvquine.org, stav z 22. 4. 2010.

[10] Microsoft directx. http://www.microsoft.com/windows/directx/, stav z 22. 4. 2010.

[11] Cadence Design Systems. http://www.cadence.com/, stav z 22. 4. 2010.

[12] Synopsys, Inc. http://www.synopsys.com/, stav z 22. 4. 2010.

[13] Kompletní specifikace berkeley pla formátu. http://service.felk.cvut.cz/vlsi/prj/BOOM/pla_c.html, stav z 22. 4. 2010.

[14] Zdrojové kódy espresso ii. http://embedded.eecs.berkeley.edu/pubs/downloads/espresso, stav z 22. 4. 2010.

[15] GNU standard pro volby v příkazové řádce.
http://www.gnu.org/prep/standards/standards.html, stav z 22. 4. 2010.

[16] Google code.
http://code.google.com/, stav z 22. 4. 2010.

[17] Qt – a cross-platform application and ui framework.
http://www.qt.nokia.com, stav z 28. 4. 2010.

[18] Usabilitynet: usability resources for practitioners and managers.
http://www.usabilitynet.org/home.htm, stav z 22. 4. 2010.

[19] Wikepedia: Zpětné prohledávání.
http://en.wikipedia.org/wiki/Backtracking, stav z 22. 4. 2010.

[20] Wikipedia: Callback (computer science).
http://en.wikipedia.org/wiki/Callback_(computer_science), stav z 22. 4. 2010.

[21] Wikepedia: Cofactor.
http://en.wikipedia.org/wiki/Cofactor_(linear_algebra), stav z 22. 4. 2010.

[22] Wikepedia: Exact cover.
http://en.wikipedia.org/wiki/Exact_cover, stav z 22. 4. 2010.

[23] Wikipedia: EDA (Electronic design automation).
http://en.wikipedia.org/wiki/Electronic_design_automation, stav z 22. 4. 2010.

[24] Wikipedia: Gray code.
http://en.wikipedia.org/wiki/Gray_code, stav z 22. 4. 2010.

[25] Wikepedia: Karnaugh map.
http://en.wikipedia.org/wiki/Karnaugh_map, stav z 22. 4. 2010.

[26] Wikepedia: NP-complete problem.
http://en.wikipedia.org/wiki/NP-complete, stav z 22. 4. 2010.

[27] Wikipedia: Quine-McCluskey algorithm.
http://en.wikipedia.org/wiki/Quine-McCluskey, stav z 22. 4. 2010.

[28] Wikipedia: Singleton pattern.
http://en.wikipedia.org/wiki/Singleton_pattern, stav z 22. 4. 2010.

[29] Wikipedia: Ternary tree.
http://en.wikipedia.org/wiki/Ternary_tree, stav z 22. 4. 2010.

[30] Wikepedia: Torus.
http://en.wikipedia.org/wiki/Torus, stav z 22. 4. 2010.

[31] wxWidgets.
http://www.wxwidgets.org/, stav z 22. 4. 2010.

Příloha A

Přiložené CD

Součástí této diplomové práce je i přiložené CD. To obsahuje následující adresáře a soubory:

`/bmin.tgz`
– *Zdrojové soubory aplikace Bmin.*

`/bmin-win.zip`
– *Spustitelné soubory pro Windows.*

`/text`
– *Textová část diplomové práce.*

`/text/dp.pdf`
– *Elektronická verze diplomové práce ve formátu pdf.*

`/text/dp.dvi`
– *Elektronická verze diplomové práce ve formátu dvi.*

`/text/tex`
– *Zdrojové soubory textové části diplomové práce.*

`/text/tex/bmin.tex`
`/text/tex/cd.tex`
`/text/tex/dp.tex`
`/text/tex/hyphen.tex`
`/text/tex/instalace.tex`
`/text/tex/mini.tex`
`/text/tex/k336_thesis_macros.sty`
`/text/tex/reference.bib`
`/text/tex/test.tex`
`/text/tex/uvod.tex`
`/text/tex/zaver.tex`

`/text/tex/figures`
– *Obrázky pro textovou část.*

- `/text/tex/figures/ascii-cube.txt`
- `/text/tex/figures/ascii-kmap.txt`
- `/text/tex/figures/ascii-qm.txt`
- `/text/tex/figures/BminComponentModel.eps`
- `/text/tex/figures/BminComponentModel.pdf`
- `/text/tex/figures/bmin-cube.eps`
- `/text/tex/figures/bmin-cube.pdf`
- `/text/tex/figures/bmin-dialog.eps`
- `/text/tex/figures/bmin-dialog.pdf`
- `/text/tex/figures/bmin-kmap.eps`
- `/text/tex/figures/bmin-kmap.pdf`
- `/text/tex/figures/bmin-qm.eps`
- `/text/tex/figures/bmin-qm.pdf`
- `/text/tex/figures/bmin-welcome.eps`
- `/text/tex/figures/bmin-welcome.pdf`
- `/text/tex/figures/fce.pla`
- `/text/tex/figures/KernelClassModel.eps`
- `/text/tex/figures/KernelClassModel.pdf`
- `/text/tex/figures/LogoCVUT.eps`
- `/text/tex/figures/LogoCVUT.pdf`
- `/text/tex/figures/QtGUIClassModel.eps`
- `/text/tex/figures/QtGUIClassModel.pdf`

Příloha B

Instalace

Bmin je možné instalovat na operačních systémech Windows a Linux. Způsob instalace se liší v závislosti na platformě.

B.1 Windows

Pro Windows je k dispozici binární spustitelný soubor `bmin.exe`. Ten se nachází na přiloženém CD v archivu `bmin-win.zip`, ve kterém jsou i všechny knihovny potřebné ke spuštění `bmin.exe`.

Instalace je tedy velmi jednoduchá. Je třeba pouze rozbalit výše zmíněný archiv a aplikaci spustit.

B.1.1 Vlastní překlad

Pro vlastní překlad souborů je třeba uložit na disk archiv `bmin.tgz` a následně ho rozbalit. Poté existuje několik způsobů jak vše přeložit. Zde bude popsán pouze jeden, který je velice jednoduchý a rychlý.

Nejprve je třeba stáhnout Qt SDK, které je možné zdarma stáhnout z oficiálního webových stránek Qt – `http://qt.nokia.com/downloads`. Součástí instalace je i vývojové prostředí Qt Creator, které usnadňuje práci při vývoji Qt aplikací. Po nainstalování je tedy třeba Qt Creator spustit a následně v něm otevřít soubor `bmin.pro`, který je ve složce `bmin` z rozbaleného archivu `bmin.tgz`. Po otevření `bmin.pro` se projekt načte. Pak už stačí zmáčknout tlačítko `Run` a vše se automaticky přeloží a spustí.

B.2 Linux

V Linuxu je třeba provést překlad. K přeložení je nutné mít nainstalovánu knihovnu Qt. Konkrétně její knihovny QtCore, QtGUI a QtOpenGL. Doporučená verze je alespoň 4.5 (verze starší než 4.5 nepůjdou pravděpodobně přeložit). Qt knihovnu je zatím třeba mít nainstalovanou i v případě, že je spouštěn pouze shell. Dále je třeba mít nainstalovanou Mesa 3D knihovnu, a to verzi alespoň 7.8. Ve starších verzích se vyskytovala chyba se zamrznutím OpenGL. Také je třeba mít nainstalován xorg-server verze alespoň 1.7 (starší totiž nepodporují Mesa knihovnu verze 7.8).

Pro překlad stačí zkopírovat a rozbalit archiv `bmin.tgz`. Překlad se provádí ve dvou krocích. Nejprve se v adresáři s rozbalenými zdrojovými kódy zavolá příkaz `qmake` (ten vytvoří `Makefile`) a poté se zavolá příkaz `make`, který přeloží zdrojové kódy. Aplikace se spustí pomocí binárního souboru `bmin`. Možný postup ukazuje Obrázek B.1.

```
$ tar -xzf bmin.tgz
$ cd bmin
$ qmake
$ make
$ ./bmin
```

Obrázek B.1: Průběh instalace v Linuxu

B.3 Webová podpora

Všechny informace o aplikaci lze nalézt také na internetu. Projekt má svojí webovou stránku:

http://bukka.eu/projects/bmin

Tam je také možné nalézt instalační soubory k současné a k minulým verzím.