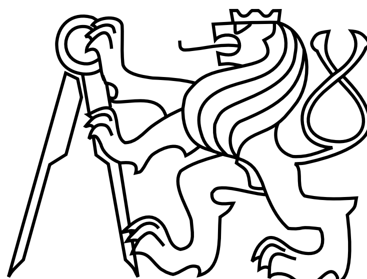
Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studiijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická



Bakalářská práce

Úprava minimalizačního nástroje ESPRESSO

Martin Miklánek

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

červen 2008

Poděkování

Děkuji všem, kteří mi pomohli a měli se mnou trpělivost.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 11.7. 2008 ....

Abstract

The main aim of this bachelor work was to create a library from already existing tool Espresso, designed for minimization of logical functions.
Except design and implementation of this library interface, the work also includes transformation of the original source code from K&R C language to C++ language.
The library was meant to be applicable under operating system MS Windows and Linux and available in static and dynamic version.

Abstrakt

Cílem bakalářské práce bylo vytvořit knihovnu z již existujícího nástroje Espresso, sloužícího pro minimalizaci logických funkcí.
Práce, kromě návrhu a implementace rozhraní knihovny, zahrnuje také transformaci původních zdrojových kódů z jazyka K&R C do jazyka C++.
Knihovna měla být použitelná pod operačními systémy Microsoft Windows a Linux, a to jak ve statické, tak i dynamické verzi.

Obsah

Seznam obrázků xiii

Seznam tabulek xv

1 Úvod 1

2 Popis problému, specifikace cíle 3
2.1 Výchozí stav 3
2.2 Cíl práce 3
2.3 Požadavky na výstup 3

3 Struktura práce 5

4 Analýza a implementace 7
4.1 Úprava do kompilovatelné formy 7
4.1.1 K&R C 7
4.1.1.1 Rozdíly oproti ANSI C 7
4.1.2 Použití vlastních knihoven 8
4.1.3 Volba nástroje na úpravu 8
4.1.4 Samotná úprava 8
4.2 Vlastní analýza Espressa 9
4.2.1 Studie manuálu Espressa 9
4.2.2 Rozbor vnitřních pochodů Espressa 9
4.2.3 Dynamická analýza 10
4.2.3.1 Volba nástroje 10
4.2.3.2 Špatná kontrola alokované paměti 10
4.2.3.3 Přístup k neinicializovaným ukazatelům 11
4.2.4 Shrnutí analýzy 11
4.3 Volba implementačního prostředí 12
4.4 Portace kódu do C++ 12
4.4.1 Změny při portaci do C++ 12
4.5 Odstranění redundantního kódu 13
4.6 Návrh a řešení interfacu 13
4.6.1 Změny provedené v původním Espressu 13
4.6.2 Řešení problémů v interfacu 15
4.6.3 Výsledek implementace 16
4.7 Kompilace knihoven 17
4.7.1 Statická knihovna 17
4.7.2 Dynamická knihovna 17
4.7.2.1 Úprava pro MS Windows 17
4.7.2.2 Úprava pro Linux 18

5 Testování 19
5.1 Testy po portaci do ANSI C 19
5.1.1 Vyhodnocení výsledků testů 19
5.2 Závěrečné testy 20

6 Závěr 23

7 Literatura **25**

A Příloha k portacím **27**
A.1 Popis balíčků původního Espressa 27
A.2 Příloha k portaci do ANSI C 27
A.3 Příloha k portaci do C++ 27

B Uživatelská příručka **29**
B.1 Popis struktury SetCost 29
B.2 popis struktury PLAStats 29
B.3 popis třídy PLA 30
B.4 Popis třídy EspressoOptions 33

C Seznam použitých zkratek **35**

D Obsah přiloženého optického média **37**

Seznam obrázků

D.1 Adresářová struktura přiloženého optického disku 37

Seznam tabulek

5.1 Nekorektní ukončení programu Espresso 20
5.2 Pády programu Espresso 20

1 Úvod

Program Espresso je heuristický nástroj pro dvouúrovňovou minimalizaci logických funkcí. Jeho autorem je Richard Rudell působící na University of California Berkeley v Kalifornii. Tento nástroj je široce používán - výjimkou není ani Katedra počítačů Fakulty elektrotechnické ČVUT v Praze.

Hlavní přínos Espressa spočívá v použití „nových“ [1] algoritmů, které zrychlují a obecně optimalizují heuristickou minimalizaci.

Další výhodou je, že formát vstupních a výstupních souborů je kompatibilní se standardním formátem pro fyzický popis PLA [2].

Espresso má nicméně i své nevýhody. Mezi největší patří použití příkazové řádky (které je podle současných standardů považováno za uživatelsky nepřátelské) a dále nemožnost použít Espresso jako modul, který by byl využíván nějakým komplexnějším programem.

Pro jeho pokrokové metody se Espresso vyplatí modernizovat, rozšířit tak jeho pole působnosti a prodloužit tím dobu používání.

Této úpravě nestojí v cestě ani licence, podle které se Espresso může volně šířit a upravovat pod podmínkou zachování copyrightu [3].

[1] upravovaná verze programu byla z roku 1988-1989

[2] PLA=Programmable logic array

[3] „Copyright ©1988, 1989, Regents of the University of California. All rights reserved.”

2 Popis problému, specifikace cíle

Tato kapitola se zaměřuje především na východiska bakalářské práce, tj. stanovuje její cíle a požadavky a popisuje stav, ve kterém byl program převzat.

2.1 Výchozí stav

V tomto odstavci je popsáno, v jakém stavu se program Espresso nacházelo před zahájením jeho změny a jaké materiály byly současně s ním autorovi práce poskytnuty.

Espresso existovalo jako spustitelný program pro operační systém MS DOS, který fungoval i pod operačním systémem MS Windows.

Samozřejmě byly k dispozici i zdrojové kódy Espressa napsané v jazyce K&R C (víc informací k nalezení zde 4.1.1.1), které se ani s přiloženým souborem *Makefile* nepovedlo zkompilovat. Program byl rozmístěn do několika adresářů představující balíčky, které plnily odlišné úkoly. Každý balíček obsahoval dokumentaci se základním popisem funkcí. (Popis balíčků k nalezení v příloze A.1

Dále byly autorovi práce poskytnuty manuálové stránky (obsahující popis přepínačů Espressa [1] a popis formátu vstupního souboru [2]) a kniha detailně popisující funkci Espresso algoritmu [6]

2.2 Cíl práce

Úkolem práce bylo upravit původní, 20 let starý program do formy použitelné pro knihovnu. Následně navrhnout a implementovat rozhraní pro knihovnu postavenou nad upraveným programem a nakonec z výsledné inovace vytvořit statickou i dynamickou knihovnu použitelnou pod operačními systémy MS Windows a Linux.

Kromě toho, že kýžená knihovna v sobě bude zahrnovat všechnu funkcionalitu původního Espressa, mají být přístupné funkce představující jednotlivé kroky Espresso algoritmu (pro víc informací viz. [6]

2.3 Požadavky na výstup

Většina primárních požadavků souvisela s tím, že se jednalo o implementaci knihovny. Knihovna by měla:

- „Po sobě vždy uklidit", tj. zajistit, aby nedocházelo k memory-leakům [1]
- poskytovat uživateli informaci o korektním průběhu vykonávaných funkcí nebo o případných chybách, které během vykonávání nastaly.
- být přenositelná

a naopak by zas neměla:

- volat funkce na ukončení běhu programu

[1] memory-leak=uniklá paměť, která měla být dealokována a nebyla

- posílat kamkoliv data, aniž by jí to bylo explicitně zadáno

Další zadavatelem určené požadavky byly následující:

- logická funkce se musí umět duplikovat (vytvořit svou kopii)
- zpřístupnit funkce představující jednotlivé kroky Espresso algoritmu

Několik dalších požadavků vyvstalo i během analýzy 4.2 a budou zmíněny v závěru analýzy 4.2.4

Volba implementačního prostředí byla ponechána na závěr analýzy (kapitola 4.2.4). Prozatím bylo určeno, že to bude C nebo C++, přičemž obě alternativy přinášejí své pro a proti.

3 Struktura práce

Jelikož kapitola 4 spojuje dva veliké celky (analýzu i implementaci), je na místě, aby bylo v následujícím odstavci stručně popsáno, o co v jednotlivých podkapitolách vlastně šlo.

Na to, aby se s programem mohlo začít skutečně pracovat, bylo zapotřebí, aby byl převeden do kompilovatelné formy (kapitola 4.1). Úspěšnou portaci mohly potvrdit jen testy nově vzniklého kódu (kapitola 5.1), které odhalily chyby, vyskytující se již v původním Espressu (kapitola 5.1.1). Až nyní se mohlo přistoupit ke skutečné analýze Espressa (kapitola 4.2). Z její výsledků se definitivně určilo implementační rozhraní (kapitola 4.3), na základě čehož se přistoupilo k poslední jazykové úpravě zdrojových kódů (kapitola 4.4), jakož i k odstranění nepoužitých funkcí a proměnných (kapitola 4.5). Následně bylo možné pokračovat k návrhu a implementaci knihovního rozhraní . Samotný závěr implementace náležel finální kompilaci knihoven ve všech požadovaných variantách pro všechny zadané platformy (kapitola 4.7).

4 Analýza a implementace

4.1 Úprava do kompilovatelné formy

Jelikož se zdrojové kódy Espressa nepovedlo zkompilovat, bylo potřeba je před započetím samotné analýzy převést do funkční podoby. Tomu bránilo jednak použití staré „normy“ jazyka a jednak použití výše zmíněných balíčků, ve kterých byly implementovány i knihovní funkce. V následujících odstavcích budou popsány oba zmíněné problémy.

4.1.1 K&R C

Pojmem K&R C se označuje jazyk C, který byl popsán Brianem Kernighanem a Dennisem Ritchiem v roce 1987 v knize „The C Programming Language” [4]. Jednalo se o její první vydání a pro mnohé programátory se stala učebnicí. O deset let později – tedy v roce 1988 – byl jazyk standardizován a doznal několik změn. Tento standard se označuje jako ANSI [1] C nebo někdy i ISO [2] C. K tomuto počinu byla vydána druhá edice knihy „The C Programming Language” [5], ve které jazyk C odpovídal nově vzniknuté normě.

V následující kapitole je uvedeno několik podstatných rozdílů mezi těmito edicemi.

4.1.1.1 Rozdíly oproti ANSI C

Některé rozdíly se týkají vizáže, jiné preprocesoru a další rozšíření, případně omezení jazyka samotného.

Zde je uvedeno několik příkladů týkajících se:

- **definic funkcí** – v K&R C jsou typy parametrů určeny až za „hlavičkou“ funkce, tedy v prostoru mezi kulatými závorkami a složenými závorkami. V ANSI C musí být parametry i jejich typy určeny už v kulatých závorkách.
- **deklarací funkcí** – v K&R C se funkce deklarovaly bez parametrů. ANSI C se snaží o typovou bezpečnost, a tak počet a typ parametrů musí být stejný v definici i deklaraci funkce.
- **implicitního typu** *int* – v K&R C se při neuvedení typu proměnné nebo návratové hodnoty funkce doplnil implicitní typ *int*. ANSI něco takového nedovoluje a ohlásí to jako chybu.
- **zániku některých klíčových slov** – původně klíčová slova *asm* a *fortran* v ANSI C již klíčovými slovy nejsou a jejich použití je zakázáné.
- **vzniku nových klíčových slov** – norma přidala tři nová klíčová slova: *const*, *volatile* a *signed*.
- **přidání sufixů za číselné konstanty** – ANSI C umožňuje určit typ číselné konstanty přidáním jednoho ze sufixů: u, U, l, L, f, F (značící postupně *int, long, float*)
- **přeproceroru** – přeprocesor K&R C byl původně rekurzivní. Tato jeho vlastnost se v normě odstranila. Další rozdíly jsou už jen ve prospěch normy a přidávají vlastnosti, které původně nebyly implementovány, např. spájení řetězců,. . .

[1]American National Standardization Institute
[2]International Standardization Organization

Úplný výčet všech rozdílů lze nalézt na stránce [3].

4.1.2 Použití vlastních knihoven

V době vzniku programu ještě neexistovala standardní knihovna, jak je tomu nyní, a autor si většinu jejích funkcí implementoval do zmíněných balíčků (příloha A.1) sám. Nyní jsou ale jeho funkce na obtíž, protože jsou při překladu v konfliktu s funkcemi deklarovanými ve standardních hlavičkových souborech. Překladač ohlásí chybu a kompilace skončí. Proto je potřeba všechny tyto funkce z projektu odstranit.

4.1.3 Volba nástroje na úpravu

Zdrojové kódy bylo potřeba pozměnit. První tři zmíněné rozdíly (kap. 4.1.1.1) oproti normě jsou jednoduše nalezitelné a upravitelné. Nevýhodou je monotónnost a zdlouhavost této úpravy. (původně se Espresso skládalo z 91 souborů se zdrojovými kódy a 19 hlavičkových souborů). Proto bylo jedním z cílů nalezení nebo vytvoření programu, který by provedl změnu definic a deklarací funkcí.

Hledání na internetu odhalilo několik programů, které byly schopny doplnit typy parametrů do deklarací a také přepsat hlavičky funkcí podle normy. Jak se později ukázalo, jediný použitelný byl nástroj *protoize,* který je určen pro operační systém Linux. Ke své činnosti nicméně potřeboval kompilovatelný kód, který v tuto chvíli nebyl k dispozici.

Původní idea byla „překlopit" kód do normalizované formy a až pak se zabývat odstraňováním vlastnoručně implementovaných funkcí. Tato myšlenka nakonec neuspěla, protože v semestrálním projektu, který si stanovil tento cíl, se velice obtížně povedlo implementovat překladač z „čistého" [3] K&R C do ANSI C. Po konzultaci s Ing. Nešvěrou se došlo k závěru, že implementace preprocesoru, který dokáže navrátit provedené změny je velice obtížná. Tato vlastnost byla ale klíčová, protože zdrojový kód po zpracování preprocesorem je méně přehledný a bez komentářů. Následnou analýzu by to učinilo obtížnou a zdlouhavou. Proto se přistoupilo k alternativě využívající nástroj *protoize.*

4.1.4 Samotná úprava

K úpravě zdrojových kódů byl vybrán nástroj *protoize,* který je určen pro operační systémy Linux a většinou bývá přímou součástí systému, neboť je pevně spjat s GCC [4]. Výjimkou nebyla ani stanice *sunray1* nacházející se na akademické půdě Fakulty elektrotechnické ČVUT na Karlově náměstí, na které byl celý port proveden.

Před začátkem samotného zpracovávání se soubory ze všech balíčků umístily do jednoho adresáře, aby se nemusely upravovat cesty pro vkládané hlavičkové soubory. Také se odhadovalo, že dojde k značnému zredukování počtu souborů, a rozdělení do balíčků bude tím pádem zbytečné.

Port byl dělán iterativně, tj. při každém spuštění program modifikoval část zdrojových kódů, které dokázal zpracovat, a následně vypsal seznam chyb (kvůli kterým nemohl pokračovat) a jejich výskyt. Pochopení příčiny a následné odstranění těchto chyb bylo ryze na lidské obsluze, jejích zkušenostech a chápání programovacích záležitostí. V příloze A.2 jsou uvedeny příklady některých chyb a jejich korekce. Jedná se jen o útržek změn, kompletní výpis je dlouhý a pro čitatele nenese důležitou informaci.

[3] Myšleno forma bez direktiv preprocesoru

[4] GNU C Compiler

Na závěr nutno podotknou, že v tomto kroku byly učiněny oba potřebné kroky: převedení do ANSI C a zároveň odstranění nežádoucích funkcí. Bylo toho docíleno právě díky tomu, že *protoize* vyžadovalo kompilovatelný kód. A jelikož konfliktní funkce, které se měly původně odstranit později, byly odstraněny již nyní, byl výsledný kód kompilovatelný. Pro další postup bylo potřeba provést testy (kapitola 5.1), které potvrdí korektnost úprav.

4.2 Vlastní analýza Espressa

Analýza teď již funkčního Espressa se musela provést postupně. Nejdříve bylo potřeba prostudovat manuál Espressa a obeznámit se s druhy a funkcí přepínačů. Pak se mohlo přistoupit k rozboru samotného kódu a na závěr se hledaly příčiny pádů odhalených při testování (kapitova 5.1).

4.2.1 Studie manuálu Espressa

Ovládání Espressa je vcelku jednoduché. Do příkazové řádky se nejdříve zadají přepínače a následně jméno souboru obsahující logickou funkci. Když se jméno nezadá, program očekává zadání logické funkce přímo z klávesnice.

Přepínače Espressa lze rozdělit do následujících skupin:

- **funkční** – určují, co se bude s logickou funkcí provádět
- **upřesňující** – ovlivňují průběh minimalizace
- **debugovací** – určují, kolik přebytečných informací uživatel obdrží
- **výstupní** – určují formát výstupní logické funkce

Jednotlivé skupiny mají vliv v různých fázích programu, což je potřeba zohlednit při návrhu knihovny.

4.2.2 Rozbor vnitřních pochodů Espressa

Následoval rozbor hlavní funkce *main()*. Bylo zjištěno, že program ve veliké míře pracuje se řadou globálních proměnných a dokonce i s globálními strukturami. Proto musela být provedena hloubková analýza všech 42 hlavních větví funkce *main()*. Každá větev představovala jeden funkční přepínač (viz. předcházející odstavec). Manuál Espressa [1] popisoval přibližně 20 těchto přepínačů, operace vykonány zbylými přepínači musely být dohledány v zdrojovém kódu. Bylo zjištěno, že některé vykonávají izolované kroky Espresso algoritmu, což vedlo k částečnému splnění požadavku na zpřístupnění těchto funkcí (viz. kapitola 2.3).

Protože popis všech větví je dlouhý a není důvod, proč s ním čitatele zatěžovat, nebude uveden ani v příloze.

Při této analýze je nutné zdůraznit, že dělat ručně rozbor všech zmíněných větví a o každé si vytvářet záznam volání je pracné a nenázorné. Pro tento typ úkolu bylo vhodné použít nějaký nástroj na statickou analýzu a vizualizaci zdrojových kódů. Šlo hlavně o zobrazení vzájemného volání mezi funkcemi, což se označuje jako *callgraph*. Po prohledání internetu se jako nejlepší favorité jevili programy *CodeViz* (využívající program *Graphviz* pro vizualizaci grafů) a *Flow for C*. Svobodný projekt *Codeviz* určený pro Linux se z neznámého důvodu nepovedlo zprovoznit. Byla proto použita 10-denní trialverze komerčního nástroje *Flow for C*. Mezi jeho nevýhody

patří kromě limitovaného použití neplacené verze neschopnost generovat tak přehledné grafy jaké jsou zobrazeny na stránkách projektu *Graphviz* . Na druhou stranu poskytuje tento nástroj o analyzovaném zdrojovém kódu víc informací než projekt *CodeViz* a proto byl opětovně použit v kapitole 4.4.

4.2.3 Dynamická analýza

V předchozí podkapitole jsme se spokojili se statickou analýzou programu – ke zjištění pochodů Espressa nám stačilo následovat jednotlivá volání mezi funkcemi a tím jsme získali přehled o vzájemných vztazích mezi funkcemi i rolemi globálních proměnných.

Nyní bylo potřeba zjistit, proč dochází k pádům Espressa u specifických případů uvedených v tabulce 5.2. Na to je nutné znát hodnoty proměnných bezprostředně před a v okamžiku pádu.

Tento druh analýzy je bez použití debugovacího nástroje prakticky nemožný, proto je zde odstavec věnovaný výběru tohoto nástroje.

4.2.3.1 Volba nástroje

Při porovnání komerčních a freewarových nástrojů se zřetelem k jejich možnostem a spolehlivosti vyšlo vítězně Visual Studio od firmy Microsoft. Jeho integrovaný debugger disponuje všemi potřebnými funkcemi (sledování proměnných, zobrazení zásobníku volání, synchronizace instrukcí v assembleru s příkazy vyššího programovacího jazyka) a také dokáže detekovat neuvolnění paměti na konci programu – tzv. *memory-leaky*. Některé freewarové programovací prostředí mají také debugger, ale ten buď nemá všechny potřebné funkce, nebo je jeho fungování více než záhadné (testováno *DevC++*, *Code::Blocks*, *Ultimate++*)

Po výběru vhodného nástroje pro dynamickou analýzu programu se přistoupilo k samotnému rozboru. Jelikož jsme narazili na více různých příčin pádu, člení se následující text na podkapitoly věnující se jednotlivým chybám.

4.2.3.2 Špatná kontrola alokované paměti

Budeme se snažit zjistit, proč byla logická funkce *o*64 jediná, u které došlo v základním nastavení Espressa k chybě (rozdělení testovacích skupin viz kapitola 5.1). Podezření padlo na nekontrolování návratové hodnoty při alokaci paměti. Po prohlédnutí zdrojového kódu programu se zjistilo, že paměť se v programu alokuje dvěma způsoby:

- **Přímým voláním** funkcí *malloc()* a *realloc()*. U tohoto způsobu alokace paměti byla návratová hodnota vždy kontrolována.
- **Nepřímo** – funkce *malloc()* a *realloc()* byly součástí maker. Návratová hodnota se u tohoto způsobu alokace nekontrolovala a tudíž zde mohlo dojít k nezachycení vzniknuté chyby.

Do těchto podezřelých maker bylo vloženo volání kontrolní funkce *valid()*, která ověřila nenulovost návratové hodnoty alokační funkce. V případě, že návratová hodnota byla nulová, funkce *valid()* ukončila program s chybovým oznámením. Funkce *valid()* uspěla – potvrdila naše podezření na vrácení nulového ukazatele.

Na zjištění konkrétního místa pádu programu bylo použito vestavěného debuggeru vývojového prostředí *MS Visual Studio 2005*. S jeho pomocí bylo odhaleno, že chyba nastane uvnitř

funkce *complement()*. Jak bylo při konzultaci se zadavatelem zjištěno, tato funkce může mít veliké paměťové požadavky, zvláště když zpracovává takovou rozsáhlou logickou funkci jakou je zmíněna funkce *o64.pla*.

Podezření na špatnou alokaci paměti se tudíž potvrdilo a s pomocí vhodných nástrojů se odhalilo konkrétní místo vzniku problému.

Dalším krokem bylo zjistit, proč dojde k „pádu“ programu, aniž by byla vyčerpána dostupná operační paměť.

4.2.3.3 Přístup k neinicializovaným ukazatelům

Další debugování odhalilo, že zbylé pády (pády v komplexním testu byly způsobeny přístupem k neinicializovanému ukazateli. Jak již bylo shrnuto v tabulce 5.2, k pádům docházelo jen v určité kombinaci vstupního nastavení Espressa a „druhu“ logické funkce. Debugování odhalilo, že dojde k přístupu k neinicializovaným položkám. Tato situace je důsledkem chybně zadaného vstupu od uživatele, na základě čehož program považuje dané položky za inicializované.

4.2.4 Shrnutí analýzy

Po obeznámení se s vnitřními pochody původního Espressa se analýza zaměřila na hledání nedostatků a použitých konstrukcí, které by bránily implementaci knihovny.

Následující odstavce popisují největší problémy a jejich důsledky v následné implementaci, které byly analýzou objeveny. Odstavce jsou označeny podle záležitosti, která je v nich blíže specifikována.

Špatná kontrola alokace paměti: Jedná se o nejhorší možné chyby, které se v programu mohou vyskytnout. Jako přítěž působí i fakt, že se nacházejí v rekurzivních funkcích, z čehož plyne velice obtížná dealokace paměti při návratu. Jelikož jsou alokační makra použita ve většině případů, nebude možno ošetřit všechny volání. Množství a složitost provedených změn by rozsahem postačovalo na samostatnou bakalářskou práci.

Neinicializované ukazatele: Ošetření těchto chyb v implementaci by nemělo být tak problematické jako je tomu u předchozího případu, jelikož se tyto chyby nacházejí v nerekurzivních funkcích. Na tomto místě bude postačovat přidání kontroly kritických ukazatelů a ukončení prováděné funkce doprovázené návratovou hodnotou indikující chybu.

Vnitřní reprezentace logických funkcí: Logická funkce je v programu reprezentována třemi hlavními strukturami. Data jsou v těchto strukturách dynamicky alokována a neodborné zacházení s jejich obsahem může mít fatální důsledky pro funkci celé navrhované knihovny. Je tudíž nevyhnutné ochránit data před neoprávněným přístupem. Jinými slovy – je potřeba ochránit data před jakýmkoliv přístupem mimo přístup funkcí původního Espressa a navrhované knihovní nadstavby.

Další výsledek analýzy se opětovně týká výše zmíněných tří struktur popisujících logickou funkci. Jen jedna z těchto struktur je mezi funkcemi programu předávána pomocí ukazatele, přičemž zbylé dvě vystupují jako globální proměnné. Jelikož jedním z požadavků je, aby se načítané logické funkce mohly uchovávat, bude potřeba zajistit, aby nedocházelo k přepisování těchto struktur při zpracování jiné logické funkce.

Globální proměnné: Většinu globálních proměnných Espressa představují upřesňující a debugovací přepínače (rozdělení přepínačů viz kapitola 4.2.1, které uživatel zadal při spuštění programu. Zkoumání odhalilo, že některé z nich jsou modifikovány i uvnitř funkcí Espressa, aniž by byly po skončení dané funkce obnoveny jejich původní hodnoty. Při vnějším pohledu

na knihovnu se očekává, že se zvolená nastavení nebudou samovolně měnit, a proto je potřeba i tento nový požadavek zohlednit při návrhu zvolené knihovny.

4.3 Volba implementačního prostředí

Doposud byl návrh knihovny takový, že její rozhraní bude implementováno pomocí funkcí a struktur jazyka C. Důvod byl ten, aby byla výsledná knihovna použitelná jak v programech psaných v jazyce C, tak v jazyce C++.

Z výsledků analýzy ovšem plyne potřeba zaměřit se v navrhované knihovně na bezpečnost, protože původní Espresso je náchylné jak na chybné údaje získané od uživatele, tak na neúspěšnou alokaci paměti.

Bezpečnost dat je jazykem C potlačená do největší možné míry, protože přístup k položkám struktur se nedá omezit a kdokoliv může modifikovat jejich obsah. Kdyby uživatel schválně (nebo omylem) modifikoval některou položku těchto struktur, data by se stala nekonzistentní a následně by došlo k pádu nebo v nejlepším případě ke špatnému fungování programu.

Jelikož okázalé chování uživatele se nedá zaručit, rozhodlo se, že rozhraní bude implementováno třídami v jazyce C++.

Kromě již zmiňovaných výhod (privátní data, použití výjimek) přináší použití jazyka C++ jedinou nevýhodu – nemožnost použít tuto knihovnu v programech psaných v jazyce C.

4.4 Portace kódu do C++

Zvolení jazyka C++ jako implementačního rozhraní navrhované knihovny umožnilo transformaci samotného základu knihovny (původní funkce Espressa) do jazyka C++.

Jazyk C se prakticky chová jako podmnožina jazyka C++. Problém při přechodu by mohl nastat u knihoven, které Espresso používalo. Naštěstí to byly jen standardní knihovny, které mají své ekvivalenty v jazyce C++. Změna jazyku by znamenala přepsání názvů vkládaných hlavičkových souborů, hlubší úpravy zdrojového kódu by neměly být zapotřebí.

Kromě programovacích výhod, jako je např. možnost použití výjimek i v samotném základu vytvářené knihovny, by tato transformace nepřinesla žádné nevýhody a proto byla navrhovaná změna vykonána.

4.4.1 Změny při portaci do C++

Jak se předtím odhadovalo, během úprav se nevyskytly žádné větší potíže. Kromě přepsání názvů hlavičkových souborů bylo potřeba řešit jmenné konflikty u nových maker, funkcí a klíčových slov, které byly zavedeny jazykem C++ nebo v jím použitých knihovnách. Podrobnější popis a rozdělení vykonaných úprav lze nalézt v příloze A.3

Nová verze programu byla posléze otestována a výsledky testů dopadly bez chyb.

Při této úpravě bylo využito freewarového nástroje *TextCrawler* pro vyhledávání a náhradu textových řetězců ve více souborech současně. Nástroj se velice osvědčil a lze jej doporučit jako alternativu ke komerčním programům v této skupině nástrojů.

4.5 Odstranění redundantního kódu

Poslední učiněný krok před řešením a implementováním interfacu knihovny bylo odstranění nepoužitých funkcí a globálních proměnných ve zdrojových kódech Espressa.

Tato úprava nebyla nezbytně nutná, ale měla za následek zjednodušení implementace. Největším přínosem bylo odstranění globálních záložních struktur, které nebyly nikdy použity a implementace se jimi nemusela zabývat. Eliminací redundantního kódu se rovněž zmenšila konečná velikost knihovny.

Na analýzu zdrojových kódů a hledání redundantních funkcí byl použit nástroj *Flow for C*, který již byl použit v kapitole 4.2.2.

Na závěr je možno dodat, že během všech úprav byly zdrojové soubory původního Espressa značně zredukovány. Na začátku úprav byl program tvořen 91 implementačními soubory a 19 hlavičkovými soubory. Na konci úprav klesl tento počet na 50 definičných a 10 hlavičkových souborů, což potvrdilo původní odhady.

Výčet všech změn provedených ve zdrojových kódech od začátku práce nebude uveden z důvodu značné rozsáhlosti.

4.6 Návrh a řešení interfacu

Původní návrh byl oddělit funkcionalitu Espressa od nastavení. K tomu účelu byly vytvořeny 2 třídy:

- **PLA** – představuje samotnou logickou funkci a její metody nahrazují funkční a výstupní přepínače původního Espressa.
- **EspressoOptions** – koncentruje v sobě upřesňující a debugovací informace. Důvod jeho vzniku byl požadavek, aby se objektu třídy PLA nemusely nastavení zadávat po jednotlivých parametrech, nýbrž najednou prostřednictvím objektu této třídy.

Tento návrh se postupně upřesňoval a přizpůsoboval nově objeveným problémům a požadavkům, avšak zásadních změn nebylo učiněno.

Během implementace se řešili problémy dvou druhů:

1. Jak uzpůsobit původní Espresso, aby bylo použitelné pro rozhraní
2. Jak implementovat některé složité výstupy Espressa, aby byly přístupné v jiné než jen původní textové formě

4.6.1 Změny provedené v původním Espressu

Globální struktury

Jak již bylo zmíněno, logická funkce je v původním Espressu popsána třemi strukturami. Kromě samotné struktury *PLA_t*, ve které jsou uložené informace načtené ze vstupu, používá ještě pomocné struktury *cube_struct* a *cdata_struct*. Při zpracování se struktura PLA_t nebo její část předává funkcím pomoci ukazatele. Zbylé dvě struktury vystupují jako globální proměnné s názvem *cube* a *cdata*. Muselo se zabezpečit, aby tyto dvě struktury náležely právě zpracovávané logické funkci, přičemž mezi načtením logické funkce a provedením nějaké Espresso operace nad touto logickou funkcí se mohl vyskytovat libovolný počet načítání a zpracování jiných logických

funkcí. Aby se tyto struktury vzájemně nepřepisovaly, bylo potřeba upravit stávající kód. Na výběr bylo několik možností:

1. Neupravovat vnitřek původních Espresso funkcí a jen z nich udělat privátní metody třídy PLA, přičemž bude tato třída obsahovat zmíněné struktury pod stejným názvem, jak tomu bylo v původním Espressu. Tato možnost se následně zavrhla, protože Espresso používá spoustu struktur, které jsou hierarchicky uspořádány a jejich společným znakem je právě přístup k zmíněným strukturám. Touto úpravou by se všechen zdrojový kód znepřehlednil, což by bylo jen na škodu při případných dalších úpravách.

2. Neupravovat vnitřek původních Espresso funkcí a zajistit, aby se tyto struktury před každou operací zkopírovaly z privátní zálohy právě zpracovávané logické funkce a po skončení této operace se privátní záloha zpětně aktualizovala z globálních struktur původního Espressa. Jak je jistě vidět, toto řešení má zcela zbytečně velikou režii a není zcela elegantní.

3. Odstranění zbytečné režie předchozího příkladu by se docílilo, kdyby funkce Espressa pracovaly ne s *globálními strukturami*, ale pouze s *globálními ukazateli na struktury*. Neustálé kopírování celých struktur by bylo nahrazeno přirazením adres do těchto ukazatelů. To by obnášelo pozměnění zdrojových kódů původního Espressa – zaměnit přímý přístup k položkám struktur za přístup nepřímý (pomocí dereference ukazatele a následného přímého přístupu). Zabezpečit inicializaci těchto ukazatelů bylo úkolem nadřazené třídy PLA, která jediná spouští funkce původního Espressa. Tato možnost byla nakonec vybrána.

Kontrola alokace paměti

Testy provedené po portaci zdrojových kódů do jazyka C odhalily nezachycené chyby při alokaci paměti. Jak již bylo zmíněno, k chybě došlo během provádění funkce, která představuje část Espresso algoritmu majícího vysoké paměťové požadavky. Protože zasvěcení čitatele do konkrétního případu by zabralo spoustu vysvětlování, bude zde uveden částečně obecný postup, jak v takovém případě postupovat. V takovéto situaci se požadavky na opravu dají shrnout do těchto bodů:

- Chyba musí být programem zachycena
- Iniciátor akce (uživatel) se o chybě musí dozvědět, tj. informace o výskytu chyby se musí distribuovat od místa vzniku až k uživateli
- každá funkce, přes kterou informace o výskytu chyby prochází musí provést dealokaci dat, která jsou chybou zasažena, případně dat, která kvůli předčasnému ukončení funkce nebudou již upotřebena.

Tyto požadavky byly aplikovány na všechny zainteresované funkce – od těch nejspodnějších, které alokují paměť, až po ty nejvyšší, které tvoří uživatelské rozhraní. Jelikož většina funkcí měla návratový typ *ukazatel*, byla informace o výskytu chyby předána pomocí hodnoty *NULL* (metody uživatelského rozhraní většinou oznamují tuto událost hodnotou -1). Tato změna si vyžádala pečlivě promyšlený zásah do řady funkcí, přičemž největší obtíž bylo určit zasažená a neupotřebitelná data.

Další změny provedené v původní zdrojovém kódu Espressa se týkaly přizpůsobení novým požadavkům (např. práce s novými druhy výstupních proudů,...) a nemá smysl tady opisovat všechny. Byly uvedeny jen ty nejdůležitější.

4.6.2 Řešení problémů v interfacu

Následující část popisuje řešení problémů, se kterými bylo potřeba se vypořádat při návrhu a implementaci samotného knihovního rozhraní.

Použití třídy EspressoOptions
Třída EspressoOptions vznikla kvůli požadavku, aby bylo možno spouštět Espresso s před-připravenými nastaveními a tyto nastavení kdykoliv pohodlně měnit. Původně se zamýšlelo, že tato třída bude parametrem každé funkce Espressa. Tím pádem dochází ke kopírování nastavení, aniž je to zapotřebí. V starších verzích návrhu byl konstruktor parametrizován a docházelo k načtení logické funkce už při inicializaci objektu. EspressoOptions se měli zadávat jako parametr metody konkrétního objektu. Konflikt vznikal v tom, že některé nastavení byla zapotřebí už při načítání logické funkce. Tento i problém zbytečného přepisování nastavení se vyřešil použitím *statické* metody *setOptions()* třídy *PLA*.

Konstruktor bez parametrů
Původní idea byla, že umístění logické funkce, kterou chce uživatel načíst (ze souboru, proudu nebo řetězce) bude parametrem konstruktoru třídy PLA. Jenže při zpracovávání vstupu může dojít k množství závažných chyb a inicializace objektu se nemusí povést. Uživatel by se o vyskytnuté chybě neměl možnost dozvědět, protože konstruktor nemá návratovou hodnotu. Může jen hodit výjimku, co by nebylo zrovna nejlepší řešení. Proto bylo rozhodnuto, že se vytvoří konstruktor bez parametrů a uživatel bude muset pro načtení logické funkce zažádat metodami *loadFromFile()* a *loadFromString()*

Metody pro načtení
Požadavkem na knihovnu bylo, aby se logická funkce mohla načíst jak ze souboru (zadaného jménem nebo deskriptorem), tak z řetězce jazyka C (typ *char**). Jelikož řetězec a jméno souboru jsou stejného typu (*char**), mohlo by být pro uživatele matoucí, že stejná metoda je schopna načíst logickou funkci jak ze souboru, tak z řetězce. Proto se přistoupilo k mnemotechnické změně a vytvořily se metody *loadFromFile()* a *LoadFromString()*, u kterých si je uživatel jist jejich funkcí a vkládaným parametrem.

Dopočet chybějících dat
Logická funkce je úplně popsána třemi množinami hodnot [5]. Její zápis v této úplné formě zbytečně zabírá mnoho místa a protože jsou tyto množiny disjunktní [6], dají se vzájemně dopočítat. Potřeba jednotlivých množin závisí na prováděné operaci a výstupním formátu logické funkce. Původní Espresso poznalo oba tyto parametry již při spuštění. Tím pádem nebyl problém dopočítat potřebné množiny hned po úspěšném načtení logické funkce. V našem případě je tomu jinak – znalost uživatelem požadované operace již při načítaní logické funkce je vyloučena, a proto musí knihovna reagovat na potřebu dopočtu konkrétní množiny až v okamžiku spuštění požadované operace. Protože tuto kontrolu a případný dopočet musí provádět všechny metody pracující s množinami, byla vytvořená privátní metoda *computeSets()*, která tuto činnost obstarávala.

Řešení parametru –Dcheck
Výstupem této funkce v původním Espressu byli 4 řádky textu informující o disjunktnosti množin tvořící danou logickou funkci. Jelikož se každé tvrzení dalo vyjádřit hodnotami *pravda/nepravda*, nabízelo se řešení pomocí dekompozice tohoto výpisu na množinu metod vracejících *bool*. Metoda *check(FILE* target)* vypíše původní výpis a současně vrátí hodnotu true nebo false v závislosti na tom, jestli se během ověřování, které provádí, vyskytla nějaká chyba. Dekompozicí byly

[5]Čitatel znalý problematiky jistě ví, že řeč je o ON-setu, OFF-setu a DC-setu. Pro tento výklad je však zacházení do detailů nežádoucí a neznalý čitatel by se mohl nechat zmást.

[6]jejich průnik je prázdná množina

vytvořeny metody *ONDCdisjoint()*, *ONOFFdisjoint()*, *DCOFFdisjoint()* a *universe()* vracející *bool* a informující o pravdivosti konkrétního výroku.

Bližší popis jednotlivých metod naleznete v příloze B.3

Řešení parametru –Dstats
Obdobný problém jako u *check()* se vyskytl i tady. Řešení už nebylo tak jednoduché a dekompozice na integrální typy se nedala použít na všechny poskytované informace, protože výstup byl v některých případech proměnlivě dlouhý výpis číselných hodnot. Tento výpis závisel na velikosti zpracovávané funkce a také na jejím „typu". Požadavkem zadavatele bylo, aby byly pomocí rozhraní knihovny přístupné všechny informace – nepostačovalo je pouze vypsat do výstupního proudu. Pro tento účel se vytvořila struktura *PLAStats*, která obsahuje integrální typy, vektory a struktury *SetCost* odpovídající všem požadovaným položkám. Nutno podotknout, že tato struktura se předává pomocí návratové hodnoty *ukazatel* a ne *referencí*. Použitím reference by se musela celá struktura zkopírovat, což u tak složité struktury stojí určitou režii. Proto se dynamicky vytvoří nová struktura na haldě a ukazatel na ní se předá uživateli. Plyne z toho jediná nevýhoda – uživatel musí tuto strukturu smazat sám. (Popis PLAStats se nachází v příloze B.2)

Zpracování dvou logických funkcí
Espresso mělo několik funkcí, které vykonaly nějakou operaci nad dvěma vstupními logickými funkcemi (např. průnik, sjednocení,. . .) a výsledkem byla třetí. Původně výsledek přepsal jednu ze vstupních funkcí – nebyl důvod ponechávat je nezměněny, protože program se po vypsáni výsledku stejně ukončil. Po diskuzi se zadavatelem vznikl požadavek, aby vstupní funkce zůstaly nezměněny a výsledná třetí byla předána pomocí návratové hodnoty.

Přirazení výstupů
Espresso poskytovalo uživateli o svém běhu spoustu výpisů. Ne všechny se daly potlačit a žádnému se nedalo určit, kam má být poslán. Knihovna má naproti tomu poskytovat úplnou kontrolu nad zobrazovanými daty. Uživatel může určit *kam* a *jaká data* budou zapsána. K tomuto účelu byly výstupy Espressa rozděleny do šesti skupin, přičemž každé jedné skupině se dá explicitně určit, zda a kam se má vypisovat. Toto opatření obnášelo úpravu všech funkcí pro výpis. Popis jednotlivých skupin a jim přislouchajícím metodám lze nalézt v příloze B.4

Návratové hodnoty funkcí
Jelikož bylo potřeba uživatele informovat o správnosti průběhu volaných metod, byla většině těchto metod určena návratová hodnota typu *int* (celočíselný typ). Výjimku tvořily pouze metody vracející typ *ukazatel na strukturu* nebo typ *bool*. Výskyt chyby při provádění dané metody je indikován návratovou hodnotou -1 u celočíselného typu,u typu *bool* hodnotou *false* a u typu *ukazatel na strukturu* hodnotou *NULL*. Návratová hodnota typu *void* byla použita jen u některých metod třídy *EspressoOptions*

Settry a gettry
Metody pro nastavení parametrů původního Espressa jsou většnou ve tvaru *nazev_parametru()* pro získání hodnoty daného parametru a *setnazev_parametru()* pro jeho nastavení. *Nazev_parametru* byl volen tak, aby byl stejný, případně nejvíc podobný parametru v původním Espressu.

4.6.3 Výsledek implementace

Knihovní rozhraní se úspěšně propojilo s původními zdrojovými kódy Espressa. V podstatě se jednalo o „obalení" Espressa navrhovaným rozhraním, přičemž původní zdrojové kódy se přizpůsobovaly tlaku rozhraní a rozhraní samotné bylo formováno zadanými a nově vzniklými požadavky.

Kromě toho, že přibyly dvě struktury pro předávání dat uživateli, se konečná verze knihovny od navrhované výrazně neliší. Implementovaly se všechny požadavky zadavatele a v rámci možností byly odstraněny i chyby v původním Espressu.

Popis metod jednotlivých tříd a položek struktur lze nalézt v přílohách:

1. třída PLA v příloze číslo B.3
2. třída EspressoOptions v příloze číslo B.4
3. struktury PLAStats a SetCost v příloze číslo B.2 a B.1

4.7 Kompilace knihoven

Výsledkem implementace jsou již kompletní zdrojové kódy knihovny. Ty čeká ještě poslední krok a tím je kompilace za účelem vytvoření knihovny. Jelikož bylo původní Espresso multiplatformní a během úprav se nepoužily žádné platformně závislé knihovny jsou zdrojové kódy knihovny přenositelné a tudíž není potřeba je před kompilací na různých operačních systémech upravovat.

4.7.1 Statická knihovna

Vytvoření statické knihovny pod Linuxem i MS Windows je stejné. Implementační soubory je potřeba nejdřív zkompilovat pomocí programu g++ s parametrem –c. Vzniknou tak soubory s koncovkou *.o*, které se následně spojí pomocí programu *ar* s parametrem –r do jediného souboru s koncovkou *.a* – výsledné statické knihovny. Použití knihovny je následující – do svého zdrojového kódu uživatel vloží hlavičkový soubor (EspressoLib.h) a při kompilaci přidá do záznamu knihoven soubor EspressoLib.a.

4.7.2 Dynamická knihovna

Kompilace dynamické knihovny pro MS Windows a Linux je rozdílná, společná je jen konvence – soubory s koncovkou *.a* mají prefix *lib*, samotná dynamická knihovna tento prefix nemá a má koncovku *.dll* (MS Windows) nebo *.so* (Linux). Následně je uveden postup kompilace pro obě platformy.

4.7.2.1 Úprava pro MS Windows

Pro dynamickou knihovnu vytvářenou pod MS Windows je potřebné zdrojový kód knihovny mírně upravit. MS Windows na rozdíl od Linuxu vyžaduje, aby se funkce a třídy určené pro export explicitně označily. Na vytvoření dynamické knihovny se použilo programovací prostředí *Dev-C++* a v něm vytvořená šablona projektu pro dynamickou knihovnu. V ní už byly definovány potřebné makra a také byly názorně použity. Makro *BUILD_DLL* je definováno pouze při kompilaci knihovny, takže není nutné vytvářet dvě verze hlavičkového souboru. Pro zajímavost jsou zde uvedeny změny, jaké je potřeba ve zdrojovém kódu provést:

```
#include <windows.h>

#ifdef BUILD_DLL
    #define DLL_EXPORT __declspec(dllexport)
#else
```

```
    #define DLL_EXPORT __declspec(dllimport)
#endif

class DLL_EXPORT MyClass{
/* deklarace Vaší třídy */
};

void DLL_EXPORT funkce(); /* deklarace Vaší funkce */
```

4.7.2.2 Úprava pro Linux

Kompilace dynamické knihovny pod Linux nevyžaduje žádné dodatečné úpravy kódu. Tento přístup je pohodlný, na druhou stranu autor nemá možnost funkce nebo třídy skrýt – při vytváření knihovny jsou vyexportovány všechny. Kompilace knihovny byla ponechána na programovacím prostředí *Code::Blocks*, které mělo v sobě připraveny šablony pro kompilaci *dynamicky vázané knihovny* (Dynamic Link Library) a také *sdílené knihovny* (Shared Library)

Zdrojové kódy k jednotlivým verzím jsou v adresáři *src* na přiloženém optickém médiu. Zkompilované knihovny s ukázkami použití jsou v adresáři *exe*.

5 Testování

Nejen na samém konci, ale i po portacích zdrojových kódů byly provedeny kompletní testy, které měly ověřit funkčnost programu. Testovací vstupy jsou na přiloženém optickém médiu v adresáři **/TEST**

5.1 Testy po portaci do ANSI C

Korektní provedení všech změn učiněných v předchozím kroku mohly potvrdit pouze podrobné testy provedené nad co největší množinou úloh.

Testovací sadou byla sbírka standardních funkcí poskytnuta zadavatelem. Testovalo se metodikou *black box* FOOTNOTE(černá skříňka = testovací metodika, která nepožaduje znalost vnitřních pochodů testovaného objektu a jen kontroluje správnost výstupů na určité vstupy), přičemž jako referenční hodnoty byly vzaty výstupy spustitelného Espressa dodaného na začátku práce.

Testy se daly rozdělit do dvou skupin podle nastavení programu a velikosti testované množiny:

1. *základní nastavení* Espressa (spouštěno s jediným parametrem - názvem vstupního souboru) pro *všechny funkce* z testovací sady.
2. postupně *všechny typy nastavení* Espressa (rozuměj všechny možné parametry) pro cíleně zvolené funkce (vstupní soubory)

První skupina sloužila k ověření korektnosti minimalizační metody. Proto se použila na všechny dostupné funkce. Druhá skupina měla ověřit všechny ostatní funkce. Správně by se měly provést komplexní testy nad všemi dostupnými logickými funkcemi. Některé funkce Espressa jsou výpočetně náročné a vyžádaly by si spoustu času a energie. Proto byl z každého „druhu“ funkce vybrán jeden zástupce, na kterém byly provedeny všechny operace. Výklad týkající se zmíněných „druhů“ logických funkcí není pro náš případ nezbytný. Důležité je pouze to, že se to při testování neopomnělo. Jestli má čitatel o toto téma zájem, může si jej přečíst na manuálové stránce zabývající se vstupním formátem pro Espreso [2]. Po testech nad první skupinou byla do druhé skupiny (skupina pro testování všech parametrů) dodatečně vložena další funkce uložená v souboru s názvem *o64.pla*. Touto funkcí se budou zabývat i následující kapitoly, neboť jako jediná ze 165 logických funkcí způsobila pád programu v základním nastavení.

Testovací soubory a skripty jsou umístěny na přiloženém médiu v adresáři **/TEST/ANSI**. Stejná testovaní skupina a nastavení byla použita i v dalších testech.

5.1.1 Vyhodnocení výsledků testů

V základním nastavení se výstupy testované verze shodovaly s výstupy referenčního Espressa. Při zadání vstupní funkce uložené v souboru *o64.pla* „havarovaly“ oba programy. Po opětovném spuštění a sledování využívaných systémových prostředků bylo zjištěno, že program postupně alokuje paměť po čím dál větších blocích. V okamžiku pádu měl přiděleno přes 1GB operační paměti! Z toho se usuzuje, že mohlo dojít k vyčerpání volné paměti pro tento proces, operační systém tím pádem nevyhověl dalšímu požadavku na paměť, program nekontroloval návratovou hodnotu alokační funkce (což je obvyklé zjednodušení při programování) a následným přístupem mimo dovolené meze byl operačním systémem zastaven. Potvrzení nebo vyvrácení této domněnky bude jedním z cílů analýzy.

V druhé testovací skupině dopadlo porovnání ve většině případů bez rozdílů. V některých případech byly porovnávané soubory velikostí stejné, obsahem však odlišné (porovnáváno příkazem

comp na MS Windows XP SP2 CZ). Při použití specializovanějšího nástroje na porovnávání obsahů dvou souborů (jaký obsahuje např. PSPad nebo TotalCommander) vyšlo najevo, že soubory jsou i obsahově stejné, jen jsou v nich zpřeházené řádky. Toto rozličné uskupení nemá vliv na správnost výsledné funkce. Tento rozdíl se vyskytl jen v nastavení využívajícím *náhodné* uspořádání (parametr *-erandom* viz. Popis parametrů Espressa **??**) a tudíž to není chyba, ale jen důsledek využívané náhody. Při zmíněné funkci o64.pla bylo chování obou programů totožné a byly ukončeny operačním systémem téměř ve všech případech. Nestalo se tak jen v některých nastaveních.

Funkce ze souboru *o64.pla* nebyla jediná problémová. Následující tabulka udává název funkce (souboru) a parametr, který způsobil problém. Jsou zde zahrnuty i dva případy pro funkci *o*64, kdy došlo k odlišnému způsobu pádu – bez alokace velkého množství paměti.

Funkce (soubor)	**Nastavení**
io.pla	-ekiss
phasepair.pla	-Dpair
phasepair.pla	-Dpairall
type.pla	-ekiss
o64.pla	-ekiss
o64.pla	-Dmap

Tabulka 5.1: Nekorektní ukončení programu Espresso

Následující tabulka uvádí případy, kdy program vypsal chybovou hlášku a následně se sám ukončil. Uvedeny jsou zde pro zdůraznění potřeby indikace a určení chyby v navrhované knihovně.

Funkce (soubor)	**Nastavení**	**Hláška**
kiss.pla, label.pla, symbolic.pla	-oeqn	espresso Must have binary-valued function for EQNTOTT output mode
kiss.pla	-Dgasp	espresso empty reduction in reduce_ gasp, shouldn't happen
kiss.pla	-Dsuper_ gasp	espresso empty reduction in reduce_ gasp, shouldn't happen
symbolic.pla	-Dpairall	espresso ON-set and OFF-set are not orthogonal
o64.pla	-Dminterms	espresso unreasonable expansion in unravel

Tabulka 5.2: Pády programu Espresso

Ukončení programu Espresso s vypsáním chybové hlášky

Všechny chyby bylo nutné brát na vědomí a v analýze jim věnovat zvláštní pozornost. Odhalily totiž nedostatky původního Espressa, které je žádoucí opravit nebo alespoň minimalizovat jejich následky.

5.2 Závěrečné testy

Závěrečné testy byly provedeny nad stejnou testovací množinou jako Testy po portaci do ANSI C. Tentokrát se testovala již hotová knihovna. Ověření funkčnosti původních algoritmů Espressa,

nově přidaných parametrů a funkcí bylo provedeno na dynamické verzi knihovny pro MS Windows. Tyto testy odhalily drobné chyby a nedodělky, které byly po odhalení opraveny. Na ostatních verzích se ověřovala pouze možnost vložit a následně použit knihovnu ve vlastním zdrojovém kódu. Hloubkové testy každé verze by byly zbytečné, protože ve zdrojovém kódu byly použity pouze knihovní funkce a k úpravám zdrojových kódů vyjma případu dynamické knihovny pro MS Windows nedocházelo.

6 Závěr

Práce splnila své cíle. Hlavní – implementaci knihovního rozhraní, i vedlejší – opravení odhalených chyb v původním kódu. Nutno dodat, že splnění vedlejšího cíle si vyžádalo mnohem víc času a úsilí, než se při zadávaní předpokládalo. Byla potřebná mnohem hlubší znalost a pochopení původního kódu. Jelikož dokumentace Espressa nezacházela až tak hluboko, muselo se těchto znalostí pracně nadobýt pomocí nástrojů FLOW for C a MS Visual Studio 2005. Rovněž jakýmkoliv jiným změnám v původním kódu byla věnována podrobná studie mající za cíl nalézt všechny překážky a případné negativní důsledky této změny.

Rozhodně se nepovedlo odhalit a opravit všechny chyby. Taková úprava by byla časově náročná, vyžádala by si značnou refaktorizaci zdrojových kódů a výsledek by dle mého názoru za to nestál. Osobně bych se přiklonil k vytvoření zcela nového projektu, který by si z původního Espressa odnesl jenom algoritmy a poučení, že by se rozhodně neměla podceňovat kontrola alokování paměti.

Práce na projektu mne obohatila v mnohých směrech a plyne z ní několik cenných ponaučení a rad do budoucna:

- analýzu není dobré podceňovat
- kontrolovat návratovou hodnotu funkcí
- operační paměti není nikdy dost a může dojít
- nepředpokládat, že jakýkoliv (ne jen 20 let starý) program je bez chyb

Na závěr jen doufám, že moje práce nebyla zbytečná a výsledná knihovna se bude používat nejen na Katedře počítačů Elektrotechnické fakulty ČVUT v Praze, ale na všech místech, kde použití původního Espressa nebude možné anebo mnou implementované rozhraní bude shledáno přívětivější než je příkazová řádka.

7 Literatura

[1] Espresso - Boolean minimizer.
http://www.ece.duke.edu/~dwyer/courses/ece52/espresso.1.html.

[2] Espresso - Input file format for Espresso.
http://www.ece.duke.edu/~dwyer/courses/ece52/espresso.5.html.

[3] Sun - The Differences Between K&R Sun C and Sun ANSI/ISO C.
http://docs.sun.com/source/806-3567/compat.html.

[4] Brian Kernighan, Dennis Ritchie. *The C Programming Language.* Prentice Hall, 1978.

[5] Brian Kernighan, Dennis Ritchie. *The C Programming Language.* Prentice Hall, 1988.

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

A Příloha k portacím

A.1 Popis balíčků původního Espressa

Informativní popis balíčků:

- ERRTRAP - obsluha závažných chyb
- ESPRESSO - funkce samotného Espressa
- MM - správa paměti (funkce malloc, free,...)
- PORT - balík funkcí umožňující přenositelnost mezi různými operačními systémy
- ST - balík implementující tabulku řetězců
- UPRINTF - umožňuje uživatelovi vypsat formátovaný řetězec na určený výstup
- UTILITY - balík běžně používaných funkcí a maker

A.2 Příloha k portaci do ANSI C

Zde je uveden jenom útržek zmíněných problémů

Konfliktní typy – šlo o deklaraci funkce se stejným jménem, jako má knihovní funkce, kterou autor deklaroval s odlišným typem – v minulosti se namísto typu *void** používal typ *char**. Řešilo se to přepsáním prototypu a použití přetypování na příslušný typ při každém volání této funkce. Např. knihovní funkce *qsort()*,...

Redefinice – knihovní funkce a autorem definovaná funkce měli stejný prototyp a u všech případů měli i stejnou funkci. To nastalo v případe, kdy si autor sám implementoval funkci, která se později stala součástí standardní knihovny. Tato chyba se řešila odstraněním autorovy funkce a použití knihovní.

A.3 Příloha k portaci do C++

Při změně zdrojového kódu z jazyka C do jazyka C++ se vyskytly tři typy problémů:

1. **přepsání názvů knihoven**

    - hlavičkové soubory v C mají tvar: *nazev*.h
    - ekvivalentní knihovny v C++ mají tvar: c*nazev*

    kde *nazev* je: stdio, stdlib, time, math, assert, string, limits.

2. **Konflikt nových klíčových slov**
    Jazyk C++ byl rozšířen o několik klíčových slov. Tato slova naneštěstí kolidovala s:

    - názvy některých funkcí, např. funkce delete a new, byly přejmenovány na delete_ (s ní párová funkce insert_) a new_

- autor si definoval booleovskou proměnnou pro tříhodnotovou logiku a nazval ji výstižně *bool*. To je v konfliktu s definicí *bool*-u v C++. Autorův *bool* byl proto nazván *bool3* a konflikt byl vyřešen.

3. **konflikt maker**
   C++ definuje několik maker navíc, které byla v konfliktu s již definovanými makrami a funkcemi. Problémové byly makra *false*, *or*, *and*, které autor Espressa použil jako názvy návěští nebo proměnných. Opět to bylo vyřešeno náhradou původních názvů za *false_*, *or_*, a *and_*.

B Uživatelská příručka

B.1 Popis struktury SetCost

```
struct SetCost{
    int cubes; /* number of cubes in the cover */
    int primes; /* number of prime cubes */
    int in; /* transistor count, binary-valued variables */
    int out; /* transistor count, output part */
    int mv; /* transistor count, multiple-valued vars */
    int total; /* total number of transistors */
};
```

Struktura *SetCost* reprezentuje cenu jednoho pokrytí. Tato struktura je třikrát použitá v struktuře PLAStats je a určuje cenu ON-setu, OFF-setu a DC-setu objektu třídy *PLA*, nad kterou byla zavolána metoda *getStats()*. Následující popis položek by měl poskytnout dostatek informací o jejich významu:

- *cubes* – udává počet krychlí(setů) v daném pokrytí
- *primes* – udává počet přímých implikantú v daném pokrytí
- *in* – udává počet tranzistorů (nul) pro všechny dvouhodnotové proměnné (vstupy) /* Count transistors (zeros) for each binary variable (inputs) */
- *out* – udává počet tranzistorů (jedniček) pro výstupní proměnnou /* Count the transistors (ones) for the output variable */
- *mv* – udává počet tranzistorů pro všechny vícehodnotové proměnné /* Count transistors for each mv variable based on sparse/dense */
- *total* – udává celkový počet literálů (in + out + mv)

B.2 popis struktury PLAStats

```
struct PLAStats{
    std::string source;
    int variables;
    int binary_variables;
    int multiplevalued_variables;
    std::vector<$int> mv_sizes;
    int outputs;
    SetCost ON_set;
    SetCost OFF_set;
    SetCost DC_set;
    std::string phase
    std::vector< std::pair<int, int> > two_bit_decoders;
    std::vector< std::vector<int> > symbolic;
};
```

Struktura *PLAStats* slouží pro předání kompletních statistik o logické funkci. Hodnoty její položek odpovídají hodnotám proměnných ve výpisu obdrženém po zavolání metody *printStats(FILE* target)*.

Následující popis položek by měl poskytnout dostatek informací o jejich významu:

- *source* – C++ řetězec s názvem zdroje, odkud byla logická funkce načtena. Liší se podle metody použité při načtení logické funkce:
  - *loadFromFile(const char*)* - řetězec shodný s názvem souboru, který byl použit jako parametr
  - *loadFromFile(FILE*)* – řetězec má hodnotu „(stream)“
  - *loadFromString(FILE*)* – řetězec má hodnotu „(string)“
- *variables* – udává počet proměnných v logické funkci
- *binary_variables* – udává počet dvouhodnotových proměnných
- *multiplevalued_variables* – udává počet vícehodnotových proměnných
- *mv_sizes* – je to vektor velikostí vícehodnotových proměnných
- *outputs* – udává počet výstupů
- *ON_set*, *OFF_set*, *DC_set* – ceny pro jednotlivá pokrytí. Pro víc informací viz. struktura SetCost
- *phase* – C++ řetězec, ve kterém je zapsána fáze logické funkce
- *two_bit_decoders* – vektor dvojic dvoubitového dekodéru
- *symbolic* – vektor vektorů hodnot symbolických proměnných

B.3 popis třídy PLA

Třída *PLA* reprezentuje logickou funkci. Její metody provádějí operace nad logickou funkcí (nebo dvěma funkcemi), dá se jim určit výstupní formát logické funkce. Další metody slouží pro vypsání logické funkce do zadaného streamu. Pomocí této třídy se určují původní nastavení Espressa (upřesňující informace pro některé vykonávané funkce, přesměrování výstupů, debugovací výpisy,. . .)

Metody pro načtení logické funkce

- *loadFromFile(const char* filename)* – načte funkci ze souboru zadaného jménem
- *loadFromFile(FILE* fdescr)* – načte funkci ze streamu (ze souboru ale i z klávesnice)
- *loadFromString(const char *function)* – načte funkci se zadaného řetězce

Metody vykonávající Espresso algoritmy

- *espresso()* – provede Espresso minimalizaci logické funkce
- *exact()* – vykoná exaktní minimalizační algoritmus

- *qm()* – vykoná rychlejší exaktní minimalizační algoritmus (vypočítává jenom jedno pokrytí)
- *so(int strategy = 0)* – každou funkci minimalizuje jako single-output funkci. Parametr *strategy* může nabývat hodnot: 0 – provede Espresso minimalizaci, 1 – provede exaktní minimalizaci
- *so_both(int strategy = 0)* – každou funkci minimalizuje jako single-output přičemž vždy vybírá tu menší funkci z ON-setu a OFF-setu. Parametr *strategy* může nabívat hodnot: 0 – provede minimalizaci, 1 – provede exaktní minimalizaci
- *simplify()* – provede rychlé zjednodušení ON-setu
- *signature()* – provede „signature cubes“ algoritmus
- *opo(bool exact_minimize = false, bool repeated = false, bool skip_sparse = false)* – provede optimalizaci výstupní fázy. Parametr *exact_minimize* určí, jestli se použije exaktní minimalizace, parametr *repeated* určí jestli se bude provádět opakované přirazení fáze a parametr *skip_sparse* určí, jestli se přeskočí „rozptýlení“ (*sparse* krok Espresso algoritmu)
- *opoall(int first = -1, int last = -1, int strategy = 0)* – minimalizuje funkci se všemi možnými přirazeními fáze. Parametry *first* a *last* určují rozsah výstupních funkcí, nad kterými se má minimalizace provést, hodnoty -1 a -1 určují celý rozsah. Parametr *strategy* může nabívat hodnot: 0 – pro Espresso minimalizaci, 1 – pro exaktní minimalizaci
- *pair(int strategy = 0)* – provede párování (hledá, které binární proměnné by měly být spárovány, aby došlo k největší redukci termů). Parametr *strategy* může nabívat hodnot: 0 – algebraické dělení, 1 – silné dělení, 2 – Espresso minimalizace, 3 – exaktní minimalizace
- *pairall(int strategy = 0)* – provede párování na všechny páry. Parametr strategy mlže nabívat hodnot: 0 – Espresso minimalizace, 1 – exaktní minimalizace, 2 – přirazení fáže

Metody provádějící operace nad dvěma logickými funkcemi

- *_union_(PLA* second)* – provede sjednocení logických funkcí
- *_xor_(PLA* second)* – provede exkluzivní součet logických funkcí
- *sharp(PLA* second)* – spočte sharp produkt logických funkcí
- *dsharp(PLA* sekond)* – spočte disjunktní sharp produkt logických funkcí
- *intersect(PLA* second)* – provede průnik logických funkcí
- *verify(PLA* second)* – ověří, jestli jsou logické funkce ekvivalentní
- *PLAverify(PLA* second)* – ověří, jestli jsou logické funkce ekvivalentní, přičemž provádí permutaci sloupců a řádků na základě jmen

Metody vykonávající jednotlivé kroky Espresso algoritmu

- *essen()* – zkontroluje přímé implikanty
- *expand()* – provede expanzi
- *gasp()*
- *super_gasp()*

- *irred()* – extrahuje minimální neredundantní subret
- *make_sparse()* – provede „rozptýlení“ (redukuje celkový počet literálů v pokrytí
- *reduce()* – provede redukci (každou krychli nahradí její redukovanou formou)
- *test()* – provede nad danou logickou funkcí jednu iteraci Espresso algoritmu

Smíšené metody

- *copy()* – vrátí ukazatel na kopii logické funkce
- *print(FILE* target)* – funkce zapíše logickou do daného výstupu
- *seto(const char*)* – slouží na nastavení výstupního formátu logické funkce. Přípustné řetězce jsou: "f", "r","d", "fd", "fr", "dr", "fdr", "fc", "rc", "dc", "fdc", "frc", "drc", "fdrc", "pleasure", "eqn", "eqntott", "kiss", "cons" a "scons", přičemž „f“ znamená ON-set, „d“ znamená DC-set, „r“ znamená OFF-set, „c“ znamená constraints_type, „pleasure“ znamená pleasure_type, „eqn“ a „eqntott“ znamenají eqntott_type, „kiss“ znamená kiss_type a „scons“ znamená symbolic_constraints_type
- *pos()* – vymění ON-set a OFF-set
- *lexsort()* – seřadí krychle standardním lexikálním způsobem

Metody poskytující informace o logické funkci

- *check(FILE* target)* – do zadaného výstupu vypíše informace o ověření konzistence logické funkce (disjunktnost všech pokrytí a jestli pokrytí tvoří univerzum)
- *ONDCdisjoint()* – vrací true/false v závislosti na tom, jestli je ON-set a DC-set disjunktní
- *ONOFFdisjoint()* – vrací true/false v závislosti na tom, jestli je ON-set a OFF-set disjunktní
- *DCOFFdisjoint()* - vrací true/false v závislosti na tom, jestli je DC-set a OFF-set disjunktní
- *universe()* – vrací true/false v závislosti na tom, jestli sjednocení ON-setu, OFF-setu a DC-setu tvoří univerzum
- *printStats(FILE* target)* – do zvoleného streamu vypíše všechny informace o funkci
- *getStats()* – vrátí ukazatel na nově alokovanou strukturu obsahující všechny informace o logické funkci (viz popis struktury PLAStats)
- *taut()* – zjistí, jestli je logická funkce tautologií
- *outputs()* – vrátí počet výstupů. Pro použití s *opoall()*
- *inputs()* – vrátí počet proměnných v logické funkci. Pro použití s *minterms()* a *d1merge()*

Statické metody pro nastavení EspressoOptions

- *setOptions(EspressoOptions &opt)* – aktualizuje nastavení podle zadaného *opt*

- *setDefaultOptions()* – nastaví výstupy a ostatní nastavení tak, jak byly v původním Espressu

Takzvané „hacky“

- *equiv()* – vrátí ukazatel na vektor obsahující dvojice ekvivalentních výstupů
- *map(FILE* target)* – do zadaného výstupu vypíše Karnaughovu mapu funkie
- *map_ dcset()* – spočte DC-set logické funkce
- *contain()* – smaže sety, které jsou obsaženy v nějakém větším setu
- *d1merge(int first = -1, int last = -1)* – provede efektivní distance-1 slučovací algoritmus. Parametry *first* a *last* určují rozsah proměnných, nad kterými se má minimalizace provést, hodnoty -1 a -1 určují celý rozsah.
- *d1merge_ in()* – provede efektivní distance-1 slučovací algoritmus, ale jenom pro vstupy
- *disjoint()* – upraví ON-set na disjunktní pokrytí
- *minterms(int first, int last)* – provede expanzi do mintermů. Parametry *first* a *last* určují rozsah proměnných, nad kterými se má minimalizace provést, hodnoty -1 a -1 určují celý rozsah
- *primes()* – generuje všechny přímé implikanty (pomocí konsenzu)
- *separate()* – odstrání DC-set z ON-setu

B.4 Popis třídy EspressoOptions

Třída *EspressoOptions* reprezentuje „úschovnu“ upřesňujících informací pro Espresso algoritmy. Dále obsahuje debugovací nastavení a určení výstupů pro výpisy Espresso algoritmů. Úlohou této třídy je udržovat nastavení zadaná uživatelem v konzistentním stavu. Dokud není použita jako argument statické metody *setOptions(EspressoOptions&)* třídy *PLA*, nemají její nastavení žádný vliv na prováděné operace třídy *PLA*. V okamžiku, kdy je použita jako parametr zmíněné metody *setOptions()*, dojde ke zkopírování jejích nastavení do globálních proměnných Espressa. Následně může být objekt této třídy modifikován, aniž by se tyto změny projevily při provádění následujících operací třídy *PLA*.

Nastavení zahrnují: určení a povolení různých druhů výstupů a upřesňující informace pro minimalizační algoritmy třídy *PLA*.

Metody na automatické nastavení hodnot

- reset() – nastaví všechny hodnoty na false a výstupy do NULL
- setEspressoDefaults() – nastaví výstupy tak, jak byly původně v Espressu

Metody na povolení výstupů

- setEat(bool value) – nastaví vypisování komentářů nalezených při načítaní logické funkce
- setEatdots(bool value) – nastaví vypisování neznámých příkazů nalezených při načítaní logické funkce

- setTrace(bool value) – nastaví „trasování“ (
- setSummary(bool value) – nastaví vypisování shrnutí po každé úspěšné operaci
- setDebug(bool value) – nastaví vypisování debugovacích informací. Vypnutím těchto výpisů budou zastaveny i výpisy *verboseDebug*
- setVerboseDebug(bool value) – nastaví vypisování podrobných debugovacích informací. Toto nastavení ovlivňuje i nastavení *debug*
- setVerboseDebug(const char * function) – nastaví vypisování podorbných informací u zvolené funkce. Název funkce musí být jedním z: "compl", "essen", "expand", "expand1", "irred", "irred1", "reduce", "reduce1", "mincov", "mincov1", "sparse", "sharp", "taut", "gasp", "exact".

Metody na nastavení výstupů Tyto metody nastavují výstupní proud, do kterého budou hlášky přesměrovány. Nastavení proudu na *NULL* zakáže jejich vypisování. Po určení výstupného proudu je potřeba výpis příslušné skupiny potvrdit – nastavit jeho příznak na hodnotu *true*

- *setTraceSummaryOutput(FILE* fdescr)* – nastaví výstup pro trace a summary
- *setDebugOutput(FILE* fdescr)* – nastaví výstup pro debut a verboseDebug
- *setErrorOutput(FILE* fdescr)* – nastaví výstup pro chybové hlášky
- *setEat(FILE* fdescr)* – nastaví výstup pro komentáře nalezené při načítaní logické funkce
- *setEatdots(FILE* fdescr)* – nastaví výstup pro neznámé příkazy nalezené při načítání logické funkce
- *setStandardOutput(FILE* fdescr)* – nastaví výstup pro oznamovací hlášky

Metody na nastavení přepínačů týkajících se algoritmů Espressa

- setFast(bool value) – nastaví příznak, aby se expanze vykonala jenom jednou
- setKiss(bool value) – nastaví „kiss-style“ minimalizační problém
- setNess(bool value) – nastaví příznak, aby se odstranili přímé implikanty
- setNirr(bool value) – nastaví příznak, aby v posledním kroku nebyly odstraněny redundantní literály
- setNunwrap(bool value) – nastaví příznak, aby se ON-set před započetím minimalizace „nerozbalil“
- setOnset(bool value) – nastaví příznak, aby se ON-set opětovně dopočítal
- setRandom(bool value) – nastaví příznak pro použití náhodného uspořádání
- setStrong(bool value) – nastaví příznak, aby se použila metoda super_gasp

Metody na získaní hodnot všech přepínačů Metody vracejí true/false v závislosti na hodnotě daného přepínače. Názvy jsou voleny tak, aby bylo zřejmé, která metoda na nastavení přísluší k dané metodě na získání hodnoty

- *eat(), eatdots(), fast(), kiss(),ness(), nirr(), nunwrap(), onset(), random(), strong(), trace(), summary(), debug(), verboseDebug()*

C Seznam použitých zkratek

ANSI American National Standardization Institute

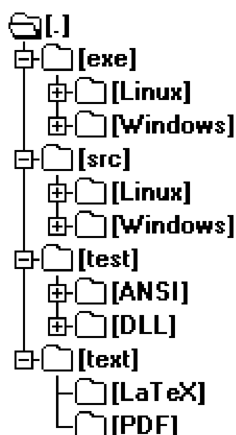
GCC GNU C Compiler

ISO International Standardization Organization

K&R Kernighan and Ritchie

PLA Programmable logic array

D Obsah přiloženého optického média



Obrázek D.1: Adresářová struktura přiloženého optického disku

V kořenovém adresáři se nachází soubor **READ.ME**, který obsahuje základní informace o obsahu všech hlavních adresářů na optickém médiu. Jednotlivé adresáře rovněž obsahují soubor READ.ME, který je specifický pro každý adresář zvlášť a obsahuje bližši informace, je li to potřebné. Základní popis adresářů:

- **exe** – obsahuje dva adresáře Linux a Windows, a v kterých se nacházejí zkompilované knihovny a také názorné ukázky použití těchto knihoven.
- **src** – obsahu zdrojové kódy knihovny připravené ke kompilování pod požadovaným operačním systémem a v požadované verzi
- **test** – adresář obsahující testovací data a specielní kód, který používá výslednou knihovnu a zkouší její funkčnost na různých vstupních funkcích
- **text** – tento adresář obsahuje textovou formu této bakalářské práce a to jak v TeX tak v PDF formátu.