# Port programu BOOM pod platformu Linux

J. Šádek

26. ledna 2005

**Prohlášení** Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

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

Mé poděkování patří ing. Petru Fišerovi za odborné vedení, trpělivou pomoc a veškerý čas, který mi věnoval při zpracovávání této bakalářské práce.

V Praze dne 26. ledna 2005 Šádek Jiří

**Abstrakt** Cílem této bakalářské práce je vytvořit novou verzi programu BOOM-II, určeného k minimalizaci logických funkcí, která by byla platformě nezávislá. Součástí zadání je též zpřehlednění zdrojového kódu, popis datových struktur a jejich přepis do "čistého" jazyka C++. Za tímto účelem mi byly přiděleny zdrojové kódy přeložitelné a funkční pod operačním systémem MS Windows. Práce nejprve seznamuje s vlastním pozadím vývoje programu k aktuální verzi. Přibližuje jednotlivé postupy a algoritmy v programu použité, které bylo třeba nastudovat a pochopit jejich provázanost s původním zdrojovým kódem. Po zachycení jednotlivých problémových partií při překladu na systémech unixového typu dochází k porovnání časů minimalizace jednotlivých verzí a systémů pro zjištění, zda nedošlo k nechtěnému poklesu výkonnosti.

**Abstract** The aim of this work was to create a platform independent version of the BOOM-II (BOOlean Minimizer) program. Besides, a well-arranged C++ source code and the description of data structures is proposed here. I have been given the source code of the MS Windows compliable program. Next, I have been acquainted with a background of the program development up to the current version. It gives an explanation of the algorithms used, which has been needed to study and understand their connection to the source codes. The problems with the program compilation under the Unix systems are noticed here. At the end of the Thesis the comparison with original version is presented.

# Obsah

# Kapitola 1

# Úvod

Program BOOM (BOOlean Minimizer) je heuristický nástroj pro minimalizaci (booleovských) logických funkcí, vyvíjený katedrou výpočetní techniky fakulty elektrotechnické ČVUT. Vlastní minimalizace vychází ze známé metody Quine McCluskey [1][2], implementujíce fáze generující přímé implikanty a řešení problému pokrytí. Programová realizace této metody byla použita v minimalizačních systémech jako např. MINI [3], ESPRESSO [4] a Scherzo [5]. BOOM [6] je výjimečný ve schopnosti řešit funkce s extrémním počtem vstupních proměnných. Výpočetní schopnost těchto algoritmů však stále zaostávala v problému skupinové minimalizace funkcí s velkým počtem výstupních proměnných. Efektivní řešení poskytl nový algoritmus vytváření skupinových implikantů prezentovaný v poslední verzi programu BOOM-II.

S minimalizací booleovských logických funkcí se dnes setkáváme v mnoha oblastech moderního návrhu logických obvodů typu PLA[1], řídících systémů či zařízení BIST[2]. V následující sekci nejprve stručně popíšu stav projektu, poté nadefinuji některé termíny z logických systémů použité v textu a vlastní problém booleovské minimalizace pro pozdější pochopení funkcí programu.

## 1.1 Převzetí projektu

Originální zdrojové kódy, které jsem obdržel jako součást zadání, obsahovaly soubor knihoven zajišťujících veškerou funkci minimalizačních algoritmů. Implementace všech základních struktur měla formát výčtového datového typu `struct`, s deklarací používanou v jazyce C++, centralizované v hlavičkovém

[1] programmable logic array

[2] built-in self-test, vestavěná diagnostika

souboru `kernel.h`. Části kódu s sebou nesly pozůstatky užití programovacího jazyka C. Ostatní knihovny sloužily k definici funkcí rozdělených povětšinou podle jednotlivých minimalizačních fází. Možnosti zahrnutí prototypu funkcí do jednotlivých struktur, s nimiž funkce spolupracují, využito nebylo, naopak to bylo tímto rozmístěním datových typů znemožněno.

## 1.2 Základní pojmy

**Literál 1.2.1** *Proměnná nebo její negace.*

**Term 1.2.2** *Term je vyjádřením součtu nebo součinu literálů. Pokud se jedná o součin, říkáme, že jde o* P-term *nebo-li* součinový term*. Jedná-li se o součet, nazveme jej* S-term *nebo-li* součtový term. Minterm*, resp.* maxterm *je takový* P-term*, resp.* S-term*, který je tvořen pouze nezávislými literály; obsahuje všechny proměnné.*

**Booleovská funkce 1.2.3** *Mějme libovolný booleový výraz generovaný proměnnými $x_1, x_2, \ldots, x_n$. Každý takovýto výraz představuje nějakou funkci: $B^n \to B$ n proměnných nad* booleovskou algebrou*, když budeme proměnné $x_1, x_2, \ldots, x_n$ chápat jako proměnné, které nabývají pouze hodnot z nosiče $B$.*

*Každou logickou funkci lze vyjádřit pomocí logického součtu* mintermů *nebo logického součinu* maxtermů*. Každý* minterm*, resp.* maxterm *nabývá hodnoty logické 1, resp. 0 právě pro jediné vstupní písmeno dané logické funkce.*

*Každou logickou funkci lze zapsat také pomocí* pravdivostní tabulky*. V té se nachází* stavový index*,* vstupní proměnné *a* funkční hodnota*.* Stavový index *udává dekadický zápis binární kombinace hodnot vstupních proměnných (je zřejmé, že zde záleží na jejich pořadí).* Funkční hodnotou *nazveme výstupní hodnotu dané funkce pro danou kombinaci vstupních proměnných. Zkrácenou formou tabulkového tvaru myslíme seznam* stavových indexů*, pro které daná logická funkce nabývá buď hodnoty logické 1 nebo 0. Např. $f(a,b,c) = \Sigma_{(1)}(1,2,5) + \Pi_{(0)}(3,6) + \Sigma_{(\times)}(4)$ je úplně určená funkce $f$, kde stavy s indexem 1, 2, 5 nabývají hodnoty logické 1, stavy s indexy 3 a 6 logické 0 a na výstupní hodnotě stavu 4 nezáleží. Stavy s logickou hodnotou 1, resp. 0 nazýváme* ***on-set****, resp.* ***off-set****; stavy, na kterých daná funkce nezávisí, nazýváme* ***dc-set*** *(don't care).*

**Implikant logické funkce 1.2.4** *Jedná se o výraz ve tvaru* P-termu*, pro který platí, že danou funkci implikuje; tzn. jestliže nabývá hodnoty logické*

*1, daná funkce nabývá též hodnoty logické 1.* Implikant *nazveme* přímým implikantem *právě tehdy, když po vypuštění libovolného literálu přestává být* implikantem. Podstatný implikant *je takový implikant, který je součástí každého minimálního řešení dané logické funkce.*

**Booleovská minimalizace 1.2.5** *Mějme množinu mohutnosti m booleovských funkcí o počtu n vstupních proměnných* $\mathcal{F}_1(x_1, x_2, \ldots, x_n)$, $\mathcal{F}_2(x1, x_2, \ldots, x_n)$, $\ldots$, $\mathcal{F}_m(x_1, x_2, \ldots, x_n)$. *Jejich výstupní hodnoty jsou dány pravdivostními tabulkami, které definují on-set* $F_i(x_1, x_2, ..., x_n)$ *a off-set* $R_i(x_1, x_2, ..., x_n)$ *pro každou funkci* $\mathcal{F}_i$. *Všechny termy, které nejsou prvky pravdivostní tabulky, jsou implicitně zařazeny do skupiny dc-set* $D_i(x_1, x_2, ..., x_n)$. *Tyto hodnoty mohou být zahrnuty také explicitně ve vstupní pravdivostní tabulce. Je zřejmé, že pro úplné definování logické funkce je postačující specifikovat pouze dvě z těchto množin. Nejčastější použití definující on-set a off-set dané funkce je výhodné pro problémy s velkým počtem vstupních proměnných a navíc specifikací off-setu zjednodušíme proces ověřování, zda je daný term implikantem či nikoli. Zde jen poznamenám, že BOOM umožňuje definovat vstupní funkci za pomoci dvojice on-set off-set. Vlastní minimalizace je tedy algoritmus, který pro každou výstupní funkci* $\mathcal{F}_i$ *spočte (SOP) výraz* $G_i = g_{1i} + g_{2i} + .. + g_{ti}$, *kde* $F_i \subseteq G_i$ *a* $G_i \cap R_i = \emptyset$. *Výraz* $T = \sum_{i=1}^{m} t_i$ *by měl být minimální.*

# Kapitola 2

# Princip činnosti

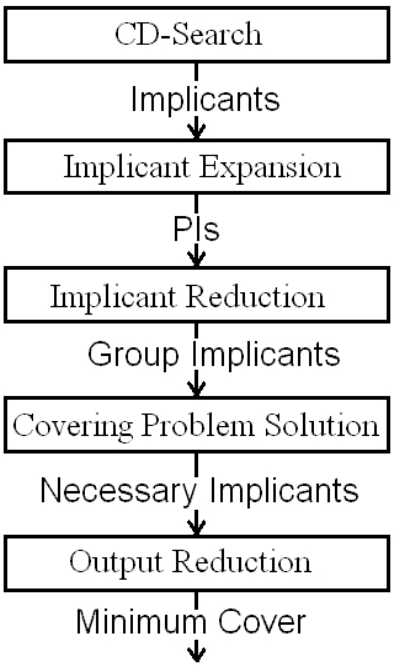
V tomto oddíle stručně shrnu vývoj programu BOOM. Zaměřím se především na teoretickou stránku minimalizace, proberu jednotlivé použité metody a postupy algoritmu.

## 2.1 První přiblížení - BOOM

Minimalizační algoritmus sestává tedy ze dvou fází. Generování přímých implikantů pro funkce jedné výstupní proměnné probíhá ve dvou krocích - *CD-Search* (Coverage-Directed Search), vytvářející množinu implikantů potřebnou pro pokrytí on-setu vstupní funkce, a následném *Implicant Expansion* (IE), která z nich vytvoří přímé implikanty. Pro funkce více výstupních proměnných se tyto dvě fáze aplikují na každou výstupní funkci odděleně. Z takto získaných přímých implikantů se následně tvoří skupinové implikanty - *Implicant Reduction* (IR). Vyřešením problému pokrytí ve fázi druhé získáme implikanty, které jsou součástí minimálního řešení. Jelikož algoritmus *CD-Search* není deterministický (je řízen náhodnou událostí), je možno jej s výhodou použít iterativně pro získání lepších výsledků. Průběh minimalizace je naznačen na obr. 2.1

### 2.1.1 CD-Search

Princip generování implikantů touto metodou spočívá ve výběru nejvíce vyhovujícího literálu, který by mohl být přidán k některému dříve vytvořenému termu. Přidáváním literálů dochází k redukování $n$-rozměrné hyperkrychle,




Obrázek 2.1: Průběh minimalizace - BOOM

které se provádí do té doby, než je daný term implikantem $\mathcal{F}_i$ (tedy do okamžiku, kdy daná hyperkrychle neprotíná žádný 0-term). Hledání vhodného literálu, možného rozšířit nějaký term, je směrováno přímo k nalezení implikantu pokrývajícího co největší počet 1-termů. Abychom toho dosáhli, začneme výběrem nejfrekventovanějšího literálu z on-setu vstupní funkce. Takto získaná hyperkrychle s rozměrem $(n-1)$ je implikantem za podmínky, že neprotíná žádný 0-term. Pokud ano, přidáme další literál (druhý nejfrekventovanější) a testujme znovu. Pokračujeme do té doby, než je implikant vygenerován a uložíme jej. Odstraníme 1-termy, které jsou již pokryty tímto implikantem, a pokračujeme v generování dalšího. Jelikož výstupem algoritmu nemusí být implikant přímý, musí být později rozšířeny. Funkce implementující tuto fázi se nacházejí v souborech `cd.h` a `cd.cpp`.

### 2.1.2 Implicant expansion

Ke snížení počtu implikantů potřebných k pokrytí všech 1-termů je potřeba je ještě rozšířit. To je zajištěno postupným odebíráním literálů z jejich termů (odebráním literálu z termu dojde k zdvojnásobení mintermů pokrytých daným termem); pokud již žádný literál nemůže být odebrán z žádného termu, jedná se o přímý implikant. Odebírání literálů je postupné a po každé expanzi

se provádí kontrola, zda nový term neprotíná off-set dané funkce. Kontrola je implementována jednoduchým porovnáváním daného termu se všemi termy off-setu. K odebírání literálů zde existují tři přístupy: zkoušení všech různých sekvencí literálů - *Exhaustive IE*, odebírání ze všech termů postupně jeden po druhém - *Sequential IE* a *Multiple Sequential IE* - zde se zkouší všechny možnosti postupného odebírání pokaždé s jinou startovní pozicí; každý implikant tak může být expandován do několika přímých. Tato fáze používá knihovny `ie.h`, v níž se nalézá funkce `ImplExpand`, jež je implementována spolu s jednotlivými metodami v `ie.cpp`. Typ použité metody se řídí proměnnou `IE_type`, obsaženou ve struktuře `MinimizeOptions`.

### 2.1.3 Problém pokrytí

Na algoritmu problému pokrytí výrazně závisí kvalita výsledné minimalizované funkce. Jelikož počet získaných přímých implikantů může být velký, je zapotřebí užití nějaké heuristiky [9]. Zde jsou použity dvě. První známá jako LCMC[1] - do řešení jsou preferované takové implikanty, které pokrývají minterm s nejmenším počtem pokrytí ostatními implikanty. Druhá metoda je založena na počítání skóre pro každý term jako kritéria pro zahrnutí do řešení. Typ použité metody je v programu nastaven pomocí proměnné `COV_type` ve struktuře `MinimizeOptions`. Funkce řešící problém pokrytí nalezneme v souborech `cover.h` a `cover.cpp`.

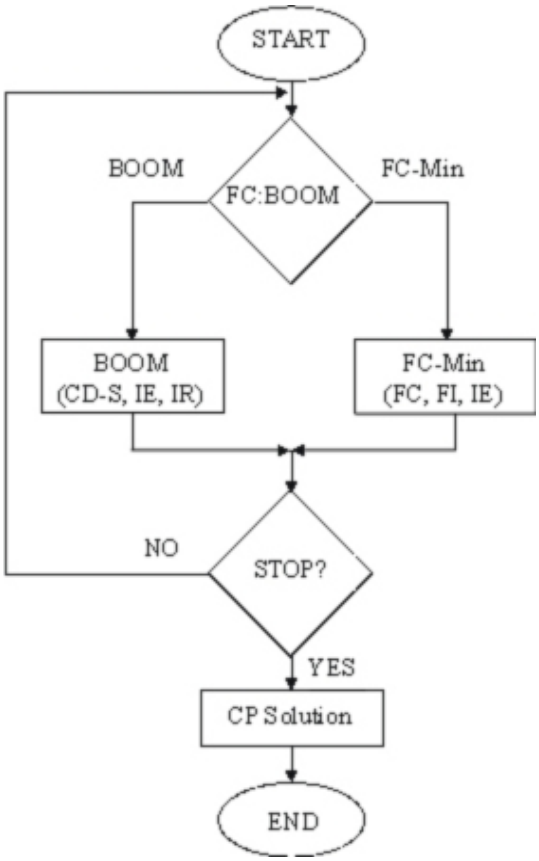
### 2.1.4 Implicant reduction

V této fázi, při skupinové minimalizaci, je snaha všechny získané přímé implikanty redukovat přidáním literálu, aby se stal implikantem více než jedné výstupní funkce. Literály se přidávají sekvenčně až do doby, kdy je zřejmé, že daný implikant se nepoužije pro více výstupních funkcí. Pokud další redukce nevede k lepším výsledkům, je zastavena, term je uložen a jemu přiřazena výstupní funkce. Pokud term neprotíná off-set nějaké funkce, stává se jejím implikantem. Implementace je zajištěna pomocí jediné funkce `MoReduce` uložené v `ir.cpp`.

[1] Least Covered, Most Covering

## 2.2 Druhé přiblížení - BOOM-II

BOOM-II je založený na kombinaci osvědčeného algoritmu *CD-Search* ze své předchozí verze a nově vyvinutého *FC-Min*. Jak bylo zmíněno, *CD-Search* je vhodný pro funkce s velkým počtem vstupních proměnných a malým počtem výstupů. To je dáno tím, že je založen na generování přímých implikantů a tudíž je použitelný v situacích, kde větší část řešení sestává právě z nich. Právě pro funkce s vysokým počtem výstupních proměnných byl vyvinut algoritmus *FC-Min*. Z důvodu, že IR fáze generování skupinových implikantů algoritmu *CD-Search* je tak časově náročná a produkuje mnoho nepodstatných implikantů, nastoluje *FC-Min* úplně jinou cestu k jejich nalezení. Způsob propojení obou algoritmů popisuje obr. 2.2.




Obrázek 2.2: Průběh minimalizace - BOOM-II

### 2.2.1 FC-Min

Jádro minimalizačního algoritmu, funkce *FC-Min*, generuje implikanty postupem úplně opačným, než je tomu u klasických minimalizačních algoritmů. Nejprve je nalezeno pokrytí on-setu dané funkce nezávislé na jeho zdrojových termech, poté jsou implikanty odvozovány z tohoto pokrytí. A právě proto, že *FC-min* neprodukuje žádné přímé implikanty, ale pouze potřebné skupinové, je to velice silný nástroj pro skupinovou minimalizaci s malými nároky na paměť.

Vlastní algoritmus *FC-Min* je rozdělen na dvě hlavní fáze - *Find Coverage*, ve které je získáno pokrytí on-setu, a *Implicant Generation*, generující implikanty z tohoto pokrytí. Funkce tohoto algoritmu obsahuje hlavičkový soubor `fcmin.h` spolu s jejich definicí v `fcmin.cpp`.

### 2.2.2 Find Coverage

Získáním pokrytí on-setu dané funkce je vlastně myšleno nalezení potenciálních implikantů, které by mohly být součástí řešení, splní-li podmínky pro stanovené pokrytí. Důležitost této fáze spočívá v tom, že určí počet implikantů obsažených ve výsledném řešení. Nalezení nejvýhodnějšího pokrytí je NP-těžký problém[2], a proto přichází na řadu heuristika. Zde použitá heuristika je založena na postupném hledání pokrývajících elementů sestávajících z maximálního počtu "1". Po nalezení takovéto části pokrytí jsou všechny "1", obsažené v něm, označeny a pokračuje se v hledání dalších pokrytí, dokud nejsou pokryty všechny "1".

### 2.2.3 Implicant generation

Poté, co je vygenerováno pokrytí (jednotlivé elementy a jejich "masky"), následuje odvození implikantů z tohoto pokrytí a zdrojových termů vstupní funkce. Je zřejmé, že pokud by měl nějaký term pokrývat konkrétní výstupní vektor, odpovídající vstupní vektor musí být v něm obsažen, jelikož vstupní vektor implikuje výstupní vektor. Proto minimální term, odpovídající danému pokrytí, je konstruován jako minimální nadkrychle ze všech vstupních vektorů odpovídajících dané množině termů tvořících pokrytí implikantu. Současně nesmí dojít k průniku s žádným vstupním termem, který není součástí dané množiny (jinak by mohl pokrývat také "0").

Více informací o algoritmech zde použitých lze získat v [7] a [8].

[2] neřešitelný v polynomiálním čase

## 2.3 Iterativní minimalizace

Oba algoritmy byly vyvinuty v iterativních verzích. Zde to znamená, že část minimalizačního procesu je řízena náhodnou událostí, což má za následek, že pokud spustíme daný algoritmus na stejné zadání vícekrát, nemusíme nutně dostat shodné výsledky. Navíc, kombinací vygenerovaných implikantů z jednotlivých běhů iteračního procesu minimalizace, lze dosáhnou výsledků lepších. K tomuto účelu je k dispozici společný buffer k ukládání vygenerovaných implikantů, z nichž je spočteno výsledné řešení. Výklad jednotlivých typů bufferů a jejich programovou reprezentaci diskutuje následující kapitola, oddíl 3.4.

# Kapitola 3

# Programové struktury

Součástí práce bylo též zpřehlednění zdrojového kódu a přepsání základních datových struktur. Šlo tedy o ucelení kódu do "čistého" jazyka C++, s přihlednutím respektování určitých standardů pro vyhnutí se některým problémům při pozdějším portu pod další platformy. Dalším neméně důležitým kritériem bylo, aby se nesnížila rychlosti výpočtu.

V této části proberu přepis nejdůležitějších datových struktur s odkazy do vlastních částí programu a uvedu stručný popis vlastní minimalizace v pseudokódu.

## 3.1 Strategie

Kvůli přehlednosti zdrojových kódů došlo k přepracování nejen datových struktur, ale též jejich uspořádání do jednotlivých knihoven. V rámci přehlednosti jsem většinu hlavních struktur přepsal do datového typu třídy, umožňující specifikovat nejen přístup oprávnění k jednotlivým položkám, ale také sjednotit metody s ní spolupracující. Zde jen poznamenám, že v rámci oddělení jednotlivých fází minimalizace zůstaly funkce je implementující v samostatných knihovnách.

Držel jsem se standardu ukládání deklarace datových typů do souborů s příponou `.h` ještě spolu s prototypy funkcí, které následně definuji v souboru `.cpp` stejného názvu. Pokud je zapotřebí použít někde jistý datový typ, bude nutně také potřeba používat funkce s ním spojené (stačí tedy do daného kódu zahrnout hlavičkový soubor - `#include<.h>`). V tomto duchu jsem přeuspořádal jednotlivé knihovny programu. V souboru `kernel.h` tak zůstaly pouze struktury voleb programu a služeb na měření času. Současně je tato knihovna využita pro přenos adres globálně viditelných proměnných.

K přepisu do datového typu třídy došlo u struktur booleovské funkce `TBoolFnct`, P-termu `TPfnct` a ternárního stromu `TermTree`. Zároveň byla vytvořena knihovna pro novou třídu sjednocující adresy jednotlivých typů bufferů, které byly původně nepřehledně zařazeny do struktury `MinimizeOptions`. Zbylé struktury pro problém pokrytí a případnou dekompozici se po dohodě s vedoucím práce nepřepisovaly z důvodu nutných zásadních oprav.

Použití pojmenování `NULL` nulového ukazatele (ukazatele "nikam") jsem eliminoval a použil hodnotu `0` (nula) dle přístupu jazyka C++. Jedním z důvodů je, že hodnota `NULL` je přes standardní makro přepisována na zmíněnou hodnotu `0`. Nic však nebrání toto makro předefinovat na hodnotu jinou, a tím mohou vzniknout potíže.

Dále jsem ze všech knihoven odstranil použití direktivy preprocesoru `#pragma`, která má sloužit k provedení akcí specifických pro danou implementaci jazyka C či C++. Nutno podotknout, že překladač *gcc* je většinou ignoruje. Vkládání těchto direktiv do zdrojového kódu v prostředí Borland C++ je však automatické.

Specifikace tříd, uvedené v následujících odstavcích, nejsou úplné a mají pouze ilustrativní charakter při popisu jednotlivých datových typů. Úplný popis lze nalézt v příloze A.

## 3.2 TBoolFnct

```
class TBoolFnct {
public:
  int terms;
  int inpvars;
  int outvars;
  TBoolFnct * parent;
  char ** inames;
  char ** onames;
  int type;
  TPfnct ** onset;
  TPfnct ** offset;
  TPfnct ** dcset;
  TPfnct * fnct;
  TPfnct * last;
  FnctInfo info;
  int * onterms;
  int * offterms;
  int * dcterms;
  int maxonterms;
  TPfnct ** index;
private:
  int split;
  int * ffbinds;
  PartInfo pinfo;
  DFTInfo dinfo;
};
```

Hlavní strukturou reprezentující vlastní booleovskou funkci je třída `TBoolFnct`. Obsahuje informace o počtu vstupů a výstupů logické funkce v celočíselných proměnných `inpvars` a `outvars`. Jejich názvy, které umí formát PLA také specifikovat, jsou pod položkami dvourozměrných polí znaků `inames` a `onames`. Typ dané funkce, tzn. informace o tom, zda je zadaná jako dvojice on-set off-set (typ fr) nebo on-set dc-set (typ fd), je dán hodnotou proměnné `type` (možnost zadat funkci typu fd není však prozatím implementována). Při nahrání dat funkce ze souboru se nejprve uloží jednotlivé P-termy ve formě spojového seznamu s adresou prvního prvku ve `fnct` a proměnná `split` se nastaví na nulovou hodnotu. Poté se za pomoci daného spojového seznamu vytvoří další tři reprezentující on-set, off-set a dc-set dané ukazateli `onset`, `offset` a `dcset`, hodnota `split` se poté změní na hodnotu

1 indikující toto rozdělení. V proměnných `onterms`, `offterms` a `dcterms` jsou pak uloženy počty vektorů s danými výstupy. Proměnné datových typů `FnctInfo`, `PartInfo` a `DFTInfo` slouží k ukládání informací o funkci při minimalizaci. Ukazatel `parent` udává adresu zdrojové funkce (u vstupní je roven 0). Třída `TBoolfnct` je definována v souboru `tboolfnct.h` a její metody v `tboolfnct.cpp`.

## 3.3 TPfnct

```
class TPfnct {
public:
  char * iv;
  char * ov;
  int dim;
  TPfnct * next;
  int valid;
  TCovs * cov;
  int covlen;
  int * covfield;
  int prime;
  int origin;
  int index;
public:
  TPfnct& operator=(TPfnct&);
};
```

Pro práci se strukturou P-termu, zmiňovanou v předchozím odstavci, je pomocí `tpfnct.h` a `tpfnct.cpp` definován datový typ třídy `TPfnct` a její metody. Hodnoty (0, - a 1), definující vstup a výstup daného termu, jsou uloženy jako pole znaků na adresách `iv` a `ov`. Indikace, zda je daný term přímým implikantem, je dána hodnotou proměnné `prime`. Původ daného termu, tzn. zda je vygenerován metodou *CD-Search* či *FC-Min*, udává celočíselná proměnná `origin`. Tři proměnné `cov`, `covlen` a `covfield` slouží pro potřeby fáze realizující pokrytí funkce. Jak již bylo zmíněno, používá se tato struktura ve formě spojového seznamu, obsahuje tedy ještě ukazatel na následující prvek - `next`.

Dále bylo pro tuto strukturu nutné vytvořit metodu, která zajistí "hlubokou" kopii instance této třídy. Je to z důvodu, že při použití operátoru přiřazení se předávají pouze adresy dané instance. Pokud tedy dojde k smazání první instance, dealokuje se obsazená paměť. Druhá instance ukazuje na část paměti, která je uvolněná, nebo na nesmyslná data (programu nic

nebrání ukládat na uvolněnou adresu nová data). Jazyk *C++* pro tento problém nabízí elegantní řešení pomocí přetěžování operátorů. Nadefinujeme si, jakým způsobem se má daný objekt kopírovat tak, aby nedocházelo k přiřazení pouhé adresy. Přetížený operátor `TPfnct& operator=(TPfnct&)` má tedy následující funkci. V přiřazení statických proměnných k problému nedochází a tudíž jsou přiřazeny standardně. Avšak u dynamicky alokovaných je nutno vytvořit novou instanci (proměnnou stejného typu s novou adresou) a překopírovat obsahy. Pokud i daná dynamicky alokovaná proměnná obsahuje dynamické proměnné, je při jejich kopírování třeba použít stejný postup. Zde je třeba kopírovat dva spojové seznamy, což je uskutečněno postupným procházením v cyklu.

## 3.4 Buffery

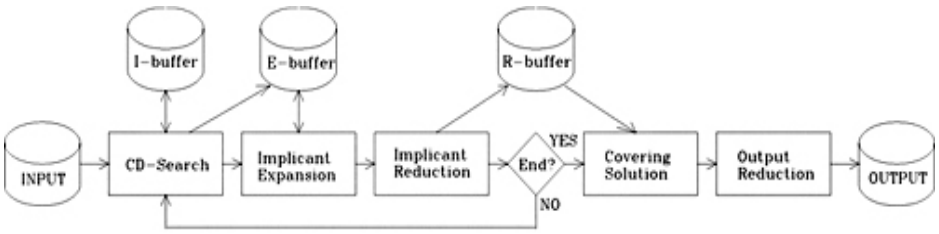
V předchozí kapitole jsem hovořil, převážně teoreticky, o generování termů, se zmínkou o jejich uložení či zapamatování. Zde specifikuji konkrétní struktury sloužící k těmto účelům.

### 3.4.1 Boom

```
class TBuffers {
private:
  TBoolFnct *source;
public:
  TermTree ** i_buffer;
  TBoolFnct * e_buffer;
  TBoolFnct * r_buffer;
};
```

Část výpočetního algoritmu, založená na první verzi programu, využívá tři typy bufferů, z toho však dva jsou reprezentovány stejným datovým typem. Všechny implikanty, které vyprodukuje fáze CD-Search, jsou ukládány do *I-bufferu* (Implicant buffer). Ten je z důvodu rychlého vyhledávání v něm implementován jako ternární strom hloubky $n$, kde $n$ je počet proměnných v termu. Hledání probíhá následovně: na $k$-té úrovni je výběr dán $k$-tou proměnnou v termu (0, -, 1). Pokud není možno výběr uskutečnit, implikant není ve stromu obsažen a je dynamicky přidán. Přítomnost termu ve stromu je indikována existencí korespondujícího listu pro daný term. Zároveň, pokud nebyl v *I-bufferu* nalezen, je vložen do *E-bufferu* (Expansion buffer), kde jsou ukládány implikanty, které jsou kandidáty k expandování do přímých.

*E-buffer* je reprezentován typem booleovské funkce `TBoolFnct`, která je odvozena od zdrojové. Implikanty tedy ukládají do spojového seznamu s prvním prvkem na adrese uloženou v `TBoolFnct->fnct`. Po expanzi jsou z *E-bufferu* odebrány a redukovány na skupinové. Po kontrole na duplicitu a dominanci jsou nově vytvořené skupinové implikanty ukládány do *R-bufferu* (Reduced implicants buffer). Struktura *R-bufferu* je shodná jako u *E-bufferu*. Použití bufferů je přehledně zakresleno na obr. 3.1.




Obrázek 3.1: Přehled použití bufferů při minimalizaci - BOOM

Vlastní implementace doznala jistých zásadních změn. Původně umístěné proměnné obsahující adresy jednotlivých bufferů, ve struktuře minimalizačních voleb `MinimizeOptions`, byly přesunuty do nově vytvořené třídy speciálně pro buffery, deklarované ve `buffers.h`. Díky tomu mohla být i část kódu ve vlastním minimalizačním algoritmu věnovaná právě jejich inicializaci, resp. dealokaci, přesunuta do konstruktoru, resp. destruktoru této třídy definovaným v `buffers.cpp`.

Pro strukturu ternárního stromu, původně nešikovně označenou jako historie, byla vytvořena třída `TermTree` (`bufferstruct.h`), ke které byly sjednoceny spolupracující metody. Byly sem přesunuty také funkce zajišťující ukládání implikantů do spojových seznamů *E-bufferu* a *R-bufferu*.

### 3.4.2 Boom-II

Při minimalizaci pomocí metody *FC-min* se nepoužívá žádných speciálních typů bufferů. Všechny vygenerované implikanty funkcí `FCMinimize` (`fcmin.cpp`) se ukládají pouze do struktury booleovské funkce `TBoolFnct`, opět ve formě spojového seznamu. Ta je potom jediným výstupem funkce `FCMinimize`.

## 3.5 Volby programu

Z předchozího teoretického výkladu je již známo, že činnost programu ovlivňuje sada voleb (options). Ty byly rozděleny podle pole působnosti na minimalizační - struktura `MinimizeOptions` a na obecné - struktura `GeneralOptions`. O minimalizačních volbách jsem se již zmínil; zobecním tedy, že se jedná o proměnné mající vliv na průběh jednotlivých fází minimalizace a také na její dobu. Obecné volby definují oddělovače pro násobení a součet, symbol pro negaci a fakt, zda se bude výpočetní čas udávat jako čas reálný, nebo jako čas spotřebovaný procesorem. Obě jsou uloženy v knihovně `kernel.h` spolu s informacemi o délce minimalizačního procesu ve struktuře `MinInfo`. Současně tato knihovna slouží k přenosu adres proměnných, které je třeba mít viditelné globálně.

## 3.6 Minimalizace

Na závěr této kapitoly uvedu popis minimalizace pomocí části pseudokódu. Měla by sloužit hlavně k provázání teoretických poznatků spojení obou algoritmů s vlastním zdrojovým kódem.

```
BoolFunction *Minimize( BoolFunction *source ) {
  random_generator_init();
  buffers->init(source);
  source->splitfnct();

  do {
       if (random_ratio > MinimizeOptions->CD_FC)
            buffers->r_buffer = One_Step_CD-Search(source);
       else
            buffers->r_buffer = One_Step_FC-Min(source);
       min_function = make_cover(buffers->r_buffer);
       }
  while ( stop );

  buffers->destroy();
  return min_function;
}
```

Na počátku minimalizačního procesu nejprve dojde k inicializaci bufferů sloužících k ukládání vygenerovaných implikantů. Zdrojová funkce rozdělí

vstupní P-termy podle daného výstupu na on-set, off-set a dc-set. `Do ... while` cyklus zajišťuje iterativní volání jednotlivých algoritmů *CD-Search* nebo *FC-Min*. Rozhodování je dáno koeficientem `CD_FC`, jenž udává poměr implikantů vytvořených v té či oné metodě.

# Kapitola 4

# Port a jeho problémy

Současně s prací na datových strukturách jsem se snažil o vlastní port pod operační systém *Linux*. Jednalo se v podstatě o odhalení problematických pasáží v původním zdrojovém kódu a jejich opravy, které bych později aplikoval na nově vytvořenou verzi. Zároveň jsem se snažil, aby tato verze nebyla specifická pro zadaný systém, nýbrž byla přeložitelná pod co možná nejvíce platformami. Na konci provedu testy a porovnání rychlostí minimalizace.

## 4.1 Použité programové vybavení

Testování a veškeré úpravy programu jsem prováděl na následujících verzích programových nástrojů:

- Distribuce Debian GNU/Linux: Sarge
- Linuxové jádro: 2.6.6
- Borland C++ BuilderX: 1.0.0.1786
- Kompilátor gcc: 3.3.4
- Knihovna libc6: 2.3.2.ds1-13

## 4.2 Problémové funkce

Zdrojové kódy verze programu mně přidělené byly napsány v prostředí Borland C++. Při prvotním pokusu o překlad pod operačním systémem Linux

jsem se setkal s problémy plynoucími z nedodržování ANSI/ISO standardů. Faktem je, že většina komerčních vývojových prostředí obsahuje knihovny pomocných funkcí, které však nejsou zahrnuty do standardních knihoven jiných operačních systémů. Některé z nich jsou sice obsaženy, ale zahrnuty do jiných hlavičkových souborů.

Vytvořil jsem tedy hlavičkový soubor `conf.h`, kde jsem za pomoci direktiv preprocesoru deklaroval použití jednotlivých problémových knihoven a funkcí.

Konkrétně se tedy jednalo o funkci **strcmpi** s prototypem:

```
#include <string.h>
int strcmpi(const char *s1, const char *s2);
```

která porovnává řetězce `s1` a `s2` bez ohledu na rozlišení velkých a malých znaků (non case sensitive). Použití je stejné jako u makra `stricmp`. Právě toto makro překládá volání `strcmpi` do `stricmp`, čímž je zajištěna kompatibilita s jinými překladači. Podobný problém nastal u funkce **strncmpi**, která však na rozdíl od předešlé dokáže porovnávat i části řetězců:

```
#include <string.h>
int strncmpi(const char *s1, const char *s2,
size_t n);
```

Další použitá funkce **setmem**, spadající do množiny funkcí pracujících s pamětí, byla třeba nahradit.

```
#include <mem.h>
void setmem(void *dest, unsigned length, char
value);
```

Tato funkce nastaví blok délky `length` bytů, v paměti počínající adresou danou ukazatelem *`dest`, na hodnotu `value`. Jelikož daná funkce nesplňuje normu ANSI C a neshledal jsem ji součástí standardních knihoven, nahradil jsem ji makrem, které její funkci předává jiné - **memset** s následujícím prototypem:

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Funkce memset ukládá hodnotu `(unsigned char) c` do všech buněk pole dané ukazatelem `*s` o velikosti `n`. Vrací hodnotu `s`. Je zřejmé, že obě funkce jsou v podstatě totožné, jelikož mezi datovými typy `int` a `char`, udávajícími hodnotu výplně, existuje implicitní konverze. Hodnota udávající délku jednoho elementu `size_t` je definovaná makrem, které ji převádí do množiny

hodnot `unsigned int`, a jedná se o hodnotu výstupu z operátoru `sizeof`. Problém v záměně právě těchto dvou parametrů a přetypování návratové hodnoty jsem vyřešil zmíněným makrem:

```
#define setmem(d,l,v) (void)memset(d,v,l)
```

S funkcí pro generování náhodných čísel v rozsahu $0\ldots(n-1)$ byl také problém.

```
#include <stdlib.h>
int random(int n);

#define random(n) (n ? (int)(rand()%(n)) : 0)
```

Ta je rovněž definována jako makro v `stdlib.h`, ale vyskytuje se pouze v knihovnách pod systémem Windows. Jelikož většina heuristik, na kterých algoritmus minimalizace pracuje, je řízena právě náhodnou událostí, věnoval jsem tomuto problému větší čas. Generátor náhodných čísel v rozsahu $0\ldots(n-1)$ definovaný makrem je uskutečněn jednoduše jako modulo dělení pseudonáhodného čísla, vygenerovaném `rand()` v rozsahu 0...RAND_MAX, požadovanou horní mezí $n$. Číslo RAND_MAX, definováno rovněž v `stdlib.h`, je však systémově závislé. Na různý platformách tak dostaneme "různě náhodná" čísla. Bylo by tedy zajímavé pokusit se napsat nějaký obecný generátor pseudonáhodných čísel a otestovat, zda má podstatný vliv na kvalitu výsledného řešení minimalizace.

## 4.3 Překlad pod dalšími systémy

Po úspěšném zprovoznění programu na systému, na němž jsem pracoval (viz. výše), jsem se jal vyzkoušet překlad na dalších systémech. Na systému **SunOS 5.9** (*gcc*: 2.95.3) společnosti *Sun Microsystems* nedošlo k žádným dalším potížím a kód byl přeložitelný ve stejné verzi jako pod operačním systémem **Linux**. Z dalších systémů unixového typu, na které mám přístup, se jednalo o **Fedora Core 3** (*gcc*: 3.4.2) a **FreeBSD 4.10** (*gcc*: 2.95.4). Na výše zmíněných systémech došlo ještě k drobnému problému s nalezením knihovny s funkcí **powl**:

```
#include <math.h>
long double powl(long double x, long double y);
```

zajišťující `y`-tou mocninu základu `x`. Ta je vyžadována standardem C99, avšak v C++ je nahrazena klasickou funkcí `pow` se stejným prototypem. Po nahrazení této funkce se odstranily všechny potíže s překladem pod systémem

**Fedora Core 3**. Ve **FreeBSD 4.10** vyvstaly problémy s knihovní funkcí pro měření času **ftime()**, která má odlišný prototyp od ostatních zde zmíněných systémů.

```
#include <sys/types.h>
#include <sys/timeb.h>

int ftime(struct timeb *tp);
```

Poměrně banální odlišnost spočívající v přidání knihovny `sys/types.h`, která je jinak součástí knihovny `sys/timeb.h`, jsem vyřešil snadno. Avšak poté, při úspěšném překladu všech knihoven programu *BOOM*, se nepodařilo vytvořit samotný spustitelný soubor a kompilátor ohlásil chybu `undefined reference to 'ftime'`. To znamená, že nemohl nalézt funkci `ftime()`, přestože se jí podařilo zahrnout do hlavičkových souborů s ní spolupracujících. Po testovacím pokusu se zrušeným počítáním času minimalizačního procesu se program povedlo přeložit a úspěšně spustit i pod tímto systémem. Po studiu manuálových stránek systému FreeBSD a diskuzních konferencí na internetu jsem došel k zjištění, že plná podpora této funkce je od verze systému 4.20.

Funkčnost překladu pod systémem **Windows** byla samozřejmě zachována.

## 4.4 Benchmarks

Pro zajímavost jsem provedl srovnání rychlostí výpočtu s původní verzí a verzemi pro operační systémy *Windows* a *Linux*. Testování jsem prováděl na klasickém PC s procesorem AMD AthlonXP 1700+ a 256MB RAM. V tabulkách používám následující konvenci zkratek názvů sloupců: sloupec "i/o/p" udává počet vstupů, výstupů a termů příslušného testu; sloupec "l/o/t" popisuje řešení kvalitativně, informuje o počtu literálů v termech SOP[1] formy řešení, ceně výstupu a o počtu produkovaných termů.

K testování jsem použil výběr funkcí ze sady standardních *MCNC* testů [10].

Porovnáme-li dosažené výsledky jednak shrnuté v tabulkách 4.1 a 4.2 nebo s výsledky měření v [8], doznáme zvýšení rychlosti minimalizace se srovnatelnou kvalitou výsledného řešení. Srovnáním časů nově vytvořené verze pod systémy *Windows* a *Linux* (tabulky 4.1 a 4.3) zjistíme značné výkonnostní rozdíly. Důvodem toho může být použití odlišného překladače, jenž má podstatný vliv na výsledné rychlosti běhu programu. Testováním rychlosti na

[1] Sum-of-Product

systémech *Fedora Core 3* a *SunOS 5.9*, které již tabulkami časů nedokládám, jsem docházel k obdobným výsledkům.

| BOOM-II nová verze | | | | | |
|---|---|---|---|---|---|
| vstup | | BOOM | | FC-Min | |
| test | i/o/p | čas [s] | l/o/t | čas [s] | l/o/t |
| cps | 24/109/855 | 8.547 | 2110/758/184 | 6.937 | 1890/946/163 |
| soar | 83/94/779 | 26.062 | 2568/509/378 | 11.609 | 2445/549/353 |
| cordic | 23/2/2105 | 1.657 | 13825/914/914 | 12.562 | 13825/914/914 |
| apex1 | 45/45/1440 | 27.641 | 1915/1025/229 | 8.875 | 1739/1103/206 |

Tabulka 4.1: Časy minimalizace, překladač bcc32, MS Windows XP

| BOOM-II původní verze | | | | | |
|---|---|---|---|---|---|
| vstup | | BOOM | | FC-Min | |
| test | i/o/p | čas [s] | l/o/t | čas [s] | l/o/t |
| cps | 24/109/855 | 10.250 | 2110/758/184 | 9.500 | 1890/946/163 |
| soar | 83/94/779 | 33.250 | 2568/509/378 | 13.796 | 2445/549/353 |
| cordic | 23/2/2105 | 1.768 | 13825/914/914 | 12.859 | 13825/914/914 |
| apex1 | 45/45/1440 | 31.751 | 1910/1030/228 | 10.594 | 1739/1103/206 |

Tabulka 4.2: Časy minimalizace, překladač bcc32, MS Windows XP

| BOOM-II nová verze | | | | | |
|---|---|---|---|---|---|
| vstup | | BOOM | | FC-Min | |
| test | i/o/p | čas [s] | l/o/t | čas [s] | l/o/t |
| cps | 24/109/855 | 10.621 | 2096/775/182 | 10.223 | 1890/946/163 |
| soar | 83/94/779 | 34.175 | 2568/509/378 | 21.438 | 2445/549/353 |
| cordic | 23/2/2105 | 2.154 | 13825/914/914 | 18.275 | 13825/914/914 |
| apex1 | 45/45/1440 | 31.698 | 1910/1030/228 | 14.362 | 1739/1103/206 |

Tabulka 4.3: Časy minimalizace, překladač gcc, Debian GNU/Linux

# Kapitola 5

# Závěr

Práce zdokumentovala hlavní úskalí přepisu programu *BOOM-II* pod systémy unixového typu. Tento program, jak již bylo zmíněno, je stále ve stádiu vývoje a myslím, že trend platformě nezávislých zdrojových kódů by měl být zachován.

Podařilo se mi zajistit množinu funkcí, které způsobovaly nemožnost kompilace programu pod systémem Linux, a jejich přímé přepsání nebo opravu pomocí definice maker preprocesoru jazyka C++. Důsledným testováním a krokováním fází minimalizace jsem také opravil funkce pracující přímo s pamětí a zařídil korektní kopírování instancí datových objektů.

Výsledkem jsou tak zdrojové kódy tohoto programu kompilovatelné nejen pod jednotlivými distribucemi systému Linux, konkrétně testované *Debian* a *Fedora Core 3*, ale také dalšími systémy unixového typu jako je *SunOS* či *FreeBSD*. Výhodou je tedy i fakt, že se nejedná o verze specifické pro jednotlivé dané systémy, ale stále tytéž zdrojové kódy. Změna datových struktur kód zpřehledňuje, ale především udává směr, kterým by se měl program vyvíjet. Přepsání kódu do plně objektového, s možností využití standardních šablon jazyka C++, stojí jistě za zmínku. Problém by však vyžadoval mnohem větší analýzu, aby se použitím těchto moderních technik nedostavil pokles výkonnosti jinak sofistikovaného minimalizačního nástroje.

Ze závěrečného provedení testů výkonnosti vyplývá, že ke srovnatelné kvalitě výsledného řešení docházíme v kratším výpočetním čase. Zajímavé je i porovnání výsledných časů minimalizace pod operačními systémy *Windows* a *Linux* při použití překladačů *bcc32* a *gcc*.

Mezi obecné poznatky při psaní programů pro více operačních systémů by mělo patřit důsledné dodržování ANSI/ISO standardů, nepoužívání funkcí

dostupných pouze v komerčních verzích vývojových prostředí či funkcí specifických pro určitou platformu. Velice se mi osvědčilo použití multiplatformního vývojového prostředí Borland C++ BuilderX, které je k dispozici i ve své volně šiřitelné verzi.

Právě pro široké možnosti programu *BOOM-II* by bylo dobré jej rozšířit o kvalitní a přehledné GUI[1], nebo lépe na zdrojových kódech nezávislý FrontEnd, který zjednodušuje práci s programem určeným pro příkazový řádek. Dalším krokem, zakomponováním jistých vizualizačních prvků do jednotlivých fází minimalizačního procesu, by mohl být tento program uplatněn v edukativním prostředí.

[1]Graphic User Interface = Grafické uživatelské rozhraní.

# Literatura


[1] W.V. Quine: The problem of simplifying truth functions, Amer. Math. Monthly, 59, No.8, 1952, pp. 521-531

[2] E.J. McCluskey: Minimization of Boolean functions, The Bell System Technical Journal, 35, No.5, Nov. 1956, pp. 1417-1444

[3] S.J. Hong, R.G. Cain and D.L. Ostapko, "MINI: A heuristic approach for logic minimization", IBM Journal of Res. & Dev., Sept. 1974, pp. 443-458

[4] R.K. Brayton et al., "Logic minimization algorithms for VLSI synthesis", Boston, MA, Kluwer Academic Publishers, 1984, 192 pp.

[5] O. Coudert, "Doing two-level logic minimization 100 times faster", Proc. of the sixth annual ACM-SIAM symposium on Discrete algorithms, 1995, pp. 112-121

[6] J. Hlavička and P. Fišer, "BOOM - a Heuristic Boolean Minimizer", Proc. ICCAD-2001, San Jose, Cal. (USA), 4.-8.11.2001, 439-442

[7] P. Fišer, J. Hlavička a H. Kubátová, "FC-Min: A Fast Multi-Output Boolean Minimizer", Proc. Euromicro Symposium on Digital Systems Design (DSD'03), Antalya (TR), 3.-5.9.2003, pp. 451-451

[8] P. Fišer, H. Kubátová, "Two-Level Boolean Minimizer BOOM-II", Proc. 6th Int. Workshop on Boolean Problems (IWSBP'04), Freiberg, Germany, 23.-24.9.2004, pp. 221-228

[9] Coudert, O.: Two-Level Logic Minimization: An Overview, Integration. The VLSI Journal, 17-2, pp. 97-140, Oct. 1994.

[10] ftp://ic.eecs.berkeley.edu

[11] M. Virius, "Programování v C++", skriptum ČVUT Praha, 1999

[12] H. Kubátová, Z. Blažek, "Logické systémy", skriptum ČVUT Praha, 1996

[13] http://www-ccs.ucsd.edu/c/

# Příloha A

# Datové struktury

## A.1 TermTree

```
class TermTree {
private:
  struct TreeNode
  {
    TreeNode * on;
    TreeNode * off;
    TreeNode * dc;
    int value;
    TPfnct * ff;

    int SearchInterNode(char * iv, int pos, int n);
  };
private:
  TreeNode * root;
  int inpvars;
  long nodes;
  long items;

  TreeNode * AddTreeNode();
  void DestroyTreeNode(TreeNode * tn);
  TreeNode * ValAddToTreeHistory(char * t, int val);
public:
  TermTree(TBoolFnct * fn);
  TermTree(int iv);
  ~TermTree() { DestroyTreeNode(this->root );};
  TreeNode * ValAddToTreeHistory(TPfnct * ff, int val);
```

```
  int AddToTreeHistory(TPfnct * ff);
  int AddToTreeHistory(char * t);
  int IsInTreeHistory(TPfnct * ff);
  TPfnct * IsTermInTreeHistory2(char * t);
  void DeleteTermFromTreeHistory(char * t);
  int IsInTreeHistory(char * t);
  int AddToTreeHistory2(char * t);
  TreeNode * GetRoot() {return root;};
  TPfnct * AddTermToTreeHistory(TPfnct * ff);
};
```

## A.2 TBoolFnct

```
class TBoolFnct {
public:
  int terms;
  int inpvars;
  int outvars;
  TBoolFnct * parent;
  char ** inames;
  char ** onames;
  int type;

  TPfnct ** onset;
  TPfnct ** offset;
  TPfnct ** dcset;

  TPfnct * fnct;
  TPfnct * last;

  FnctInfo info;
  int firstpass;

  int hits;
  int tries;

  DecompAss * decomp;
  int optval;

  int * onterms;
  int * offterms;
  int * dcterms;
  int maxonterms;
```

```
  TPfnct ** index;

private:
  int split;
  int tmp;
  int * ffbinds;

  PartInfo pinfo;
  DFTInfo dinfo;

  void DuplNames();
  void NameOVars();
  void NameIVars();

public:
  TBoolFnct& operator=(TBoolFnct &_fn);
  TBoolFnct(int iv, int ov, int vct, int tp);
  ~TBoolFnct();
  TBoolFnct * LoadFnct(char * filn);
  TBoolFnct * CreateChildFnct();
  TBoolFnct * DuplFnct();
  TBoolFnct * DumpOnset();
  TPfnct * AddEmptyTermToEnd();
  TPfnct * AddTermToBeginning(char *ivr, char *ovr, int tdim);
  TPfnct * AddTermToEnd(char * ivr, char * ovr, int tdim);
  TPfnct * DeleteTerm(TPfnct * ff, TPfnct * pff);
  void DeleteTerm(int t);
  int SaveFnct(char * filn);
  void SaveFnctAsSymbols(char * filn);
  void SaveFnctAsHTML(char * filn);
  void SaveFnctAsVHDL(char * filn);
  void SaveFnctAsVerilog(char * filn);
  void SaveFnctAsVerilog2(char * filn);
  void InitSourceFnct();
  void CloseSourceFnct();
  void NameVars() {this->NameIVars(); this->NameOVars();};
  void JoinFnct(TBoolFnct * srcf);
  void FillEmptyFnct();
  void GetInfo();
  PartInfo GetPartinfo();
  DFTInfo GetDFTinfo();
  char * FunctToSymbols(int ovr);
```

```
  void DeleteInvalidTerms();
  void SplitFnct();
  int GetDim(char * ivr);
  int AdvCountPrimes(int mask);
  int CountFnctHamming(char * c);
  void NullFFBinds();
  void IsolateFunction() {
    this->DuplNames();
    this->parent = 0;
    };
  void DeleteInpVariable(int v);
  void DeleteOutVariable(int v);
  void InsertTerm(int t);
  void InsertInpVariable(int v);
  void InsertOutVariable(int v);
  void MakeCovField(TPfnct * ff);
  void SubstOutDCs();
  void DeleteSplit();
  void DeleteNames();
  void GetOptVal();
  void Reindex();
  char * IsPrime(TPfnct * ff);
};
```

## A.3 TPfnct

```
class TPfnct { public:
  char * iv;
  char * ov;
  int dim;
  TPfnct * next;
  int valid;
  TCovs * cov;
  int covlen;
  int * covfield;
  int dvar;
  int essential;
  int tmp;
  int tmp2;
  int cvd;
  int prime;
  int origin;
  int index;
```

```
public:
  TPfnct();
  TPfnct& operator=(TPfnct&);
  TPfnct(TBoolFnct *_parent, char *_iv, char *_ov, int _tdim);
  ~TPfnct();
  void DeleteCovs();
  int CountPrimes();
  int CountFCs();
  int CountCommons();
  int CountUnknowns();
};
```

## A.4 Buffers

```
class TBuffers {
private:
  TBoolFnct *source;
public:
  TermTree ** i_buffer;
  TBoolFnct * e_buffer;
  TBoolFnct * r_buffer;
  TBoolFnct * minf;
  TBoolFnct * ess;
  TermTree * primes_ie;
  TermTree ** offtree;
  TermTree ** imp_test;
  TBuffers(TBoolFnct *srcf);
  ~TBuffers();
};
```