České vysoké učení technické v Praze
Fakulta elektrotechnická

Diplomová práce

Databáze benchmarkových obvodů

Jaroslav Pachola

Vedoucí práce: Ing. Petr Fišer

Studijní program: Informatika a výpočetní technika

leden 2007

Poděkování

Děkuji vedoucímu práce, panu Ing. Petru Fišerovi, za cenné rady a podněty.

Prohlášení

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

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

V Praze dne 10. 1. 2007 .....

Abstract

This diploma thesis is concerned with analysis of formats used for benchmarks for testing of algorithms of logic synthesis. The thesis inspects file formats that are used for notation of behaviour and structure of benchmark circuits, contains descriptions of some benchmark sets that have been created thanks to activities of research and educational institutions and some companies from the business sphere.

As the main part of the diploma thesis I have created a command line parser of benchmark formats PLA, BENCH, KISS2, BLIF and SLIF and a database of benchmark circuits with web-based user interface. The interface allows to catalogue benchmarks, search them using various criteria and download benchmark files that match those criteria - individually or all at once. The database of benchmarks utilizes database system PostgreSQL, the dynamic web-based user interface has been implemented in PHP programming language. Programs for parsing of benchmark formats and some auxiliary operations have been created using Python programming language. The thesis describes the selected solution, used means, among others from the security point of view and from the view of managability of the project and also describes testing of the application. The thesis is complemented by source code, installation guide and user guide.

Anotace

Práce se zabývá analýzou používaných formátů zkušebních obvodů pro testování algoritmů pro logickou syntézu. Zkoumá souborové formáty, které se používají pro zápis chování a struktury zkušebních obvodů, obsahuje i popis některých sad zkušebních obvodů, které vznikly díky aktivitám výzkumných a vzdělávacích institucí i komerčních subjektů.

Jako hlavní součást diplomové práce byl vypracován řádkový analyzátor formátů PLA, BENCH, KISS2, BLIF a SLIF a databáze zkušebních obvodů s webovým uživatelským rozhraním. Rozhraní umožňuje katalogizovat zkušební obvody, vyhledávat je podle různých kritérií a stahovat soubory těchto zkušebních obvodů vyhovující těmto kritériím - jednotlivě nebo najednou. Databáze zkušebních obvodů využívá databázový systém PostgreSQL, dynamické webové rozhraní bylo realizováno v jazyce PHP. K vytvoření programů pro analýzu souborů zkušebních obvodů a pro některé další pomocné akce byl použit programovací jazyk Python. Práce rámcově popisuje zvolené řešení, použité prostředky, zabývá se konkrétní implementací mimo jiné z bezpečnostního hlediska a z hlediska udržovatelnosti projektu a problematikou testování aplikace. K práci jsou přiloženy zdrojové texty a instalační a uživatelská příručka.

Obsah

Seznam obrázků

1 Úvod

Tato práce se zabývá zkušebními obvody pro testování algoritmů pro logickou syntézu, nazývanými též benchmarky. Popisuje způsoby, jakými lze zapsat různé typy zkušebních obvodů a sady zkušebních obvodů, které jsou k dispozici odborné veřejnosti. Tyto sady vznikly jako výsledek práce institucí z oblasti výzkumu a vzdělávání a obsahují zkušební obvody v různých formátech. Tyto formáty popisují zkušební obvody na různé úrovni, od popisu chování v behaviorálních modelech, až po detailní popis struktury obvodu. Některé formáty jsou lépe čitelné člověkem, jiné hůře, některé jsou snáze počítačově analyzovatelné a jiné velmi obtížně. Po prozkoumání používaných formátů a po domluvě se zadavatelem práce jsem dospěl k pětici často používaných formátů, na jejichž zpracování jsem se zaměřil - jsou to formáty PLA, KISS2, BENCH, BLIF a SLIF.

Jedním z úkolů, které přede mnou stály, bylo naprogramovat řádkový analyzátor formátů zkušebních obvodů. Pro vybranou pětici formátů jsem vytvořil program, který vypisuje základní parametry zkušebních obvodů těchto formátů.

Databáze zkušebních obvodů slouží k jejich katalogizaci, umožňuje vyhledávání těchto obvodů podle nejrůznějších kritérií, jejich stahování - jednotlivě nebo agregovaně (všechny nalezené soubory jsou zabalené v jednom archivu). Při vkládání zkušebního obvodu má uživatel dvě možnosti, jak zadat vlastnosti obvodu. Přirozenou možností je zadat vlastnosti ručně, ovšem díky vypracovanému analyzátoru souborů je možné vložit mnohé údaje s pomocí tohoto analyzátoru automaticky.

Údaje o zkušebních souborech jsou uloženy v relačním databázovém systému s podporou SQL - konkrétně byl pro implementaci použit databázový systém PostgreSQL, ale volba podpůrných knihoven a technik zohledňuje možnost nasazení jiného databázového systému.

Uživatelské rozhraní je tvořeno webovou aplikací napsanou v dynamickém programovacím jazyce PHP s využitím HTTP serveru Apache. Vývojovou platformou byl operační systém Linux, cílovou platformou pro pilotní nasazení je operační systém FreeBSD.

Řádkový analyzátor a některé pomocné programy byly naprogramovány v jazyce Python. Výhodou nasazení tohoto jazyka je vysoká flexibilita, dobře čitelný kód a podpora mnoha platforem.

Práce obsahuje analýzu řešených problémů, vysvětluje, proč byla zvolena konkrétní řešení a popisuje implementaci aplikační části z hlediska bezpečnosti a udržovatelnosti kódu a zabývá se i problematikou testování aplikace.

Kapitola 2 se zabývá analýzou cílů práce, na které budou postaveny další kapitoly.

Kapitola 3 podrobněji probírá funkcionalitu aplikační části bez řešení konkrétní implementace.

Kapitola 4 stručně pojedná o základních podporovaných formátech zkušebních obvodů, zjišťuje, jaké informace je možné v daných formátech nalézt, díky čemuž bude možné lépe rozhodnout o struktuře databáze a povaze uložených dat.

V kapitole 5 se zabývám různými možnostmi řešení aplikační části diplomové práce a volbou konkrétních řešení. Krátce pojednám i o používaných technologiích a programech pro lepší

uvedení do problematiky samotné implementace.

Provoz každé aplikace je ohrožen bezpečnostními riziky. Bezpečnostní rizika ohrožují provoz aplikace, ukládaná data i systém, na kterém aplikace běží. O možných bezpečnostních problémech a jejich řešení pojednává kapitola 6.

V kapitole 7 postoupím k popisu implementace a jejích jednotlivých částí. Konkrétně se zastavím u problematiky automatické analýzy jednotlivých benchmarkových formátů a výpisu získaných parametrů, potom vysvětlím strukturu dat uložených v databázi, implementaci webového rozhraní databáze benchmarků, dávkové a interaktivní plnění databáze a agregované stahování dat.

Úspěšná implementace zadané funkcionality aplikace je hlavním a na první pohled i jediným cílem softwarového projektu. Ve skutečnosti je situace složitější. Projekt je obvykle zapotřebí udržovat a rozvíjet, což vyžaduje dodržování určitých zásad při vytváření kódu. V kapitole 8 se krátce zmíním o některých zásadách psaní udržovatelného kódu a o efektivní správě zdrojového kódu.

Kapitola 9 se zabývá testováním funkčnosti aplikační části. Důraz je kladen na testování systémové a akceptační, ale ukážu i praktickou ukázku jednotkového testování specifické operace a zmíním se i o automatizované kontrole korektnosti kódu bez vazby na znalost požadované funkce aplikace.

Závěr hodnotí dosažené výsledky a zamýšlím se v něm nad možným pokračováním.

Struktura, rozhraní, chování, parametry zkušebního obvodu apod. jsou uloženy v souborech několika různých formátů. Tyto formáty mají různé vyjadřovací možnosti, které se snaží tato práce popsat. Příloha A se bude podrobněji zabývat těmito formáty, jejich strukturou a možnostmi.

V současné době existuje vícero různých sad benchmarků, které vznikaly v různých dobách, na jejich vzniku se podílely nejrůznější výzkumné a vzdělávací instituce, používají různé benchmarkové formáty apod. Příloha B se stručně zmíní o významných sadách benchmarků, jejich struktuře, používaných formátech a dalších charakteristikách.

Instalační a uživatelskou příručku, stejně jako další materiály, jsem umístil na CD. Jeho obsah je předmětem přílohy C.

2 Analýza cílů práce

V této kapitole se zmíním o základních cílech databáze, podrobnější informace o zvolených přístupech a řešeních se objeví v dalším textu

Tato práce má několik dílčích cílů:

- Analyzovat dostupné zkušební obvody pro testování algoritmů pro logickou syntézu.
- Analyzovat jejich dostupné specifikace a určit informace, které lze zjistit ze zdrojového souboru nebo analýzou obvodu.
- Vytvořit program pro analýzu některých formátů zkušebních obvodů.
- Vytvořit interaktivní databázi benchmarků.

2.1 Dostupné zkušební obvody pro logickou syntézu

Obvody, které jsem měl k dispozici, se nacházejí v rozličných sadách benchmarků, které mi poskytl vedoucí práce. Zahrnují širokou škálu obvodů kombinačních i sekvenčních, mají různou složitost i způsob popisu. O některých zkušebních obvodech se zmíním v kapitole o sadách benchmarků.

2.2 Informace, které lze zjistit ze zkušebních obvodů

V závislosti na typu specifikace souboru lze zjistit nejrůznější typy parametrů. Konkrétnější popisy uvedu v kapitole o formátech benchmarků. Některé benchmarkové soubory obsahují komentáře, které mohou obsahovat užitečné informace o benchmarku, tyto komentáře ovšem, pro jejich nejednoznačnost z hlediska syntaxe, nebudu uvažovat.

2.3 Zpracování aplikační části

Vytvořené programy by měly být pokud možno provozovatelné na různých platformách a důraz by měl být kladen i na čitelnost jejich kódu a snadnou rozšiřitelnost a udržovatelnost.

2.4 Testování aplikační části

Je třeba vyvinout sadu testovacích scénářů, které ověří bezchybný chod programů a jejich správnou instalaci a konfiguraci.

2.5 Dokumentace aplikační části

Správce aplikace potřebuje pro usnadnění instalace mít k dispozici instalační příručku, uživateli pro základní orientaci poslouží uživatelská příručka.

2.6 Použitý jazyk aplikační části

Ve všech programech budou, z důvodu čitelnosti lidmi pocházejícími z různých zemí, používány anglické identifikátory i komentáře. Toto je zásada, která se v praxi dobře osvědčila a vzhledem k mobilizaci pracovní síly a otevírání naší země světu její význam dále poroste. Po domluvě se zadavatelem bude i uživatelské rozhraní programů v anglickém jazyce.

2.7 Program pro analýzu benchmarků

Tento program má za úkol extrahovat informace z benchmarkového souboru a zobrazit je, případně je dále zpracovat. Program musí podporovat základní typy formátů benchmarků - podle dohody se zadavatelem práce se bude jednat o formáty PLA, KISS2, BENCH, BLIF a SLIF.

2.8 Interaktivní databáze benchmarků

Databáze benchmarků musí umožňovat zadávat a vyhledávat informace o dostupných benchmarcích. Benchmarky vyhledané podle zadaných kritérií musí databáze umožňovat stáhnout jednotlivě nebo agregovaně. K databázi je třeba definovat i přístupovou politiku.

Z povahy zadání plyne, že aplikace musí být dynamická, interagovat s uživatelem a poskytovat mu v každé chvíli pokud možno vyčerpávající relevantní informace.

3 Požadovaná funkcionalita aplikační části

V této kapitole rozeberu potřebnou funkcionalitu aplikační části. Rozbor umožní vybrat prostředky pro konkrétní implementaci a tuto implementaci realizovat.

3.1 Automatická analýza formátů testovacích obvodů

Vychází z potřeby uživatele zjistit co nejvíce informací o daném obvodu. Formát může obsahovat nejrůznější informace, na základě kterých si uživatel může udělat základní obrázek o složitosti obvodu a jeho struktuře. Typickými informacemi, které lze zjistit analýzou obvodu, jsou počty vstupů a výstupů nebo počty hradel určitého typu.

Analyzátor přijme jeden nebo více souborů obsahujících testovací obvody a vypíše informace o každém obvodu. Analýzu provede na základě znalosti fomátu souboru. Formát souboru vyhodnotí ze znalosti přípony souboru nebo na základě explicitního zadání přípony uživatelem.

3.2 Základní požadavky na databázi testovacích obvodů

Databáze testovacích obvodů obsahuje informace o jednotlivých obvodech, ale i samotné soubory testovacích obvodů v podporovaných formátech. Informace v databázi musí být možné modifikovat, prohlížet a vyhledávat, soubory bude třeba mít možnost stáhnout do adresáře uživatele pro prohlédnutí souboru nebo další zpracování.

3.3 Informace ukládané v databázi

Základní jednotkou dat uloženou v databázi je samotný testovací obvod - benchmark. Data benchmarku rozdělíme na popisná a strukturní. Strukturní data jsou v podobě souboru v podporovaném formátu (např. BENCH) - konkrétní podporované formáty proberu v následující kapitole.

Testovací obvody jsou seskupeny v sadách a tuto skutečnost musí respektovat i naše databáze.

Testovací obvod se stejnou funkcí a názvem může být vyjádřen různými způsoby i v rámci jedné sady. To znamená, že sada může obsahovat onen obvod v několika formátech nebo i v několika instancích v rámci jednoho formátu - např. v optimalizované a neoptimalizované podobě. Těmto různým vyjádřením jednoho obvodu budu dále říkat instance testovacího obvodu (benchmarku). V praxi to znamená, že je možné mít základní sdílená data vyjádřená pro obvod jako takový, které nazvu data benchmarku a dále data platná pouze pro danou instanci.

Data benchmarku zahrnují název benchmarku a jeho obecný popis.

Popisná data instance se rozdělují do dvou skupin. První skupinou jsou obecná data, která je možno přiřadit jakémukoliv testovacímu obvodu, ve druhé skupině jsou data, která jsou přiřazována dynamicky na základě automatické analýzy nebo doplněním uživatelem.

Mezi obecná data instance patří:

- Formát instance (např. BLIF).
- Konkrétní popis instance (např. informace o provedení optimalizace).
- Sada benchmarků, do níž instance náleží.

Dynamicky přidělovaná data nazvu atributy instance. Tyto atributy se budou připojovat k instanci až v případě, že budou jejich hodnoty zadány do systému a vymazání hodnoty povede k odpojení atributu.

Informace o sadě benchmarků musí obsahovat název sady a její popis.

Informace o formátu musí obsahovat název formátu a jeho popis.

3.4 Vytvoření struktury a základních dat databáze

Struktura databáze musí být v rámci instalace vytvořena automaticky, je nutné vypracovat prostředek, který naplní základní data, např. data podporovaných formátů, do databáze, po vytvoření struktury.

3.5 Plnění dat do databáze

Data do databáze je třeba mít možnost zadávat jak ručně, tak i automatizovaně. Automatizované plnění dat využije vytvořený analyzátor benchmarků a dovoluje naplnit najednou data celé sady, jsou-li uložena v rámci společného adresáře. Automatické plnění dat bude probíhat tak, že program rekurzivně prochází zadaný adresář a při nalezení benchmarkového souboru známého formátu naplní jeho data do databáze a přiřadí jim zadanou sadu benchmarků.

3.6 Zobrazovaná data databáze

Uživatelské rozhraní databáze musí umožňovat zobrazovat veškerá popisná data konkrétních instancí. Dále je třeba umožnit výpis podporovaných formátů, popisných informací o těchto formátech a zobrazení seznamu sad, které konkrétní formát obsahuji. U benchmarkových sad je nutno rovněž zobrazit název a popis a dále seznam formátů, které sada obsahuje a odkaz na vyhledání instancí obsažených v sadě.

3.7 Vyhledávání instancí

Instance benchmarků potřebuje uživatel vyhledávat podle několika kritérií:

- Podřetězec názvu benchmarku,
- formát,
- sada,
- hodnota atributu.

Je vhodné moci zadat v rámci vyhledávacích kritérií referenční hodnoty více atributů, u každé hodnoty je třeba mít možnost zadat i relaci (>, <, =, <>, <=, >=). Všechna vyhledávací kritéria budou spojena logickou operací AND.

Protože zadaným kritériím může vyhovovat větší množství instancí, je třeba vymyslet způsob, jak omezit počet najednou zobrazovaných výsledků a jak listovat mezi podmnožinami výsledků.

3.8 Stahování benchmarků

Uživatel musí mít možnost stáhnout si vyhledané instance benchmarků, resp. jejich strukturální data ve tvaru benchmarkového souboru, do určeného adresáře. Stahování může být jednotlivé nebo agregované. Agregované stahování znamená stažení archivu obsahujícího všechny benchmarky vyhovující zadaným kritériím.

3.9 Modifikace dat v databázi

Modifikace dat v databázi zahrnuje změnu popisu formátu nebo sady. Nejdůležitější je však modifikace instancí benchmarků.

Modifikace instancí benchmarků musí umožňovat změnit popis instance, obecný popis benchmarku a hodnoty atributů (u neexistujících to znamená připojení atributu k instanci, je nutné též moci smazat hodnotu atributu). Zadáním nového jména dojde k připojení instance k jinému benchmarku nebo jeho vytvoření. Instanci musí být rovněž možno přesunout do jiné sady.

3.10 Omezení přístupu k datům databáze

Správce systému určí, zda prohlížení databáze bude dostupné všem zájemcům nebo zda bude k prohlížení nutno mít založen uživatelský účet. Modifikace dat v databázi benchmarků je vázána na uživatelský účet s dostatečným oprávněním.

3.11 Správa uživatelských účtů

Součástí rozhraní databáze benchmarků bude i správa uživatelských účtů, tedy jejich vytváření, změna přístupových práv, změna hesla a zrušení uživatelského účtu. Pro správu uživatelských účtů musí mít uživatel zvláštní úroveň oprávnění. Při vytváření struktury databáze bude vytvořen jeden uživatelský účet, který bude disponovat právy k administraci uživatelských účtů.

4 Podporované formáty zkušebních obvodů

V této kapitole stručně pojednám o pěti formátech testovacích souborů, které aplikační část, zejména pak analyzátor souborů, podporuje. Tato kapitola si neklade za cíl poskytnout vyčerpávající informace o těchto formátech - podrobnější popis těchto formátů se nachází v příloze A.

4.1 Formát PLA / Espresso

Tento formát je dvouúrovňový popis logické funkce. Formát je řádkově orientovaný, každý významný řádek představuje řádek PLA matice nebo deklaraci uvozenou klíčovým slovem. Soubor může obsahovat komentáře uvozené znakem # na začátku řádku (objevil se ale i soubor s komentářem za samotným obsahem řádku) - vše za tímto znakem je při zpracování ignorováno. Přebytečné mezery a jiné bílé znaky mezi elementy jsou ignorovány, stejně jako řádky obsahující pouze bílé znaky nebo komentář.

Identifikátory (vstupů a výstupů) mohou obsahovat znaky alfanumerické znaky (a-z, A-Z, 0-9), dále znaky _~-[]<>+.

Ze souboru v tomto formátu lze snadno automaticky vyčíst:

- Počet vstupů,
- počet výstupů,
- počet termů.

4.2 Formát KISS2

Formát KISS2 obsahuje definici sekvenčního obvodu ve formě konečného automatu. Skládá se z hlavičky a seznamu přechodů. Formát je řádkově orientovaný - každý řádek hlavičky obsahuje maximálně jeden údaj a každý řádek seznamu přechodů obsahuje jeden přechod.

Ze souboru je možné automatizovaně zjistit:

- Počet vstupů,
- počet výstupů,
- počet přechodů,
- počet stavů,
- resetovací stav.

4.3 Formát BENCH

Jedná se o víceúrovňový popis kombinačního nebo sekvenčního obvodu - jde o přímé propojení hradel a klopných obvodů (tzv. netlist).

Ze souboru je možné automatizovaně vyčíst

- Počet vstupů,

- počet výstupů,
- počet hradel podporovaných typů: OR, AND, NOR, NAND, XOR, XNOR, BUF, NOT,
- počet klopných obvodů typu D.

4.4 Formát BLIF

Formát BLIF se používá k víceúrovňovému popisu logického obvodu v textové podobě. Popisovaný obvod může obsahovat odkazy na externí definice hradel a KO (.mlatch, .gate) Může obsahovat popis konečného automatu vložením .kiss bloku a popis hradel pomocí PLA matice.

Ze souboru lze automaticky určit počet hradel, některé typy hradel je možné určit analýzou PLA matic, zejména hradla typu BUF, NOT, AND, NAND, OR, NOR.

4.5 Formát SLIF

SLIF je formát s poměrně volnými pravidly, příkazy mohou začínat kdekoliv na řádku a mohou obsahovat libovolný počet mezer, tabelátorů a mohou být rozděleny do několika řádků. Příkaz formátu SLIF je ukončen středníkem.

Identifikátory mohou obsahovat písmena, číslice a znaky :^%[]_./-

Proměnné, názvy modelů i instancí jsou identifikátory. "0" i "1" jsou proměnné se speciálním významem a reprezentují logické hodnoty True a False.

Logické výrazy používají operátory + (OR), * (AND, může být vynechán), ' (negace) a části výrazů mouhou být uzavřeny do kulatých závorek. Příklad deklarace obsahující logický výraz:

```
out = (aa + (bb cc)')';
```

Ze souboru lze snadno automaticky vyčíst:

- Počet vstupů,
- počet výstupů,
- počet klopných obvodů typu D.

Převedením těchto logických výrazů na hradla lze určit počet hradel, analýzou těchto logických výrazů je možné určit běžné typy hradel - BUF, NOT, AND, NAND, OR, NOR.

5 Volba řešení aplikační části

5.1 Uživatelské rozhraní

Lze uvažovat tři základní typy uživatelského rozhraní:

1. Příkazová řádka. Program je spuštěn s parametry z příkazové řádky. Pro většinu uživatelů je tato možnost příliš strohá a klade na ně příliš vysoké nároky.

2. Klasické GUI rozhraní založené na oknech v systému typu Microsoft Windows nebo X-Window. Pro uživatele je příjemné, nicméně vyžaduje stažení speciální aplikace na počítač klienta a pro každou podporovanou platformu je nutné zajistit funkčnost klienta. Možnou výhodou by byl běh celé aplikace na klientském počítači. To ovšem představuje také významnou nevýhodu - data by bylo třeba uchovávat na každém počítači zvlášť, pokud bychom chtěli dosáhnout opravdové samostatnosti. A dalším argumentem proti tomuto řešení je omezená možnost kooperace mezi uživateli a distribuce změn v jednotlivých instalacích databáze.

3. Rozhraní založené na technologii WWW (webové rozhraní). Uživatelská přívětivost je srovnatelná s okenním GUI rozhraním, ale má několik podstatných výhod. Především klientská aplikace je prakticky na všech potenciálních platformách k dispozici. Je jí webový browser. Ten je dostupný jak na osobních počítačích různých typů s různými operačními systémy, tak i na terminálech, průmyslových počítačích, PDA a dokonce i ve většině dnešních mobilních telefonů.

Zvoleno bylo webové rozhraní. Dnešní technologie pro tvorbu dynamických webů se liší mimo jiné v nárocích kladených na klientský software, mnohé weby vyžadují podporu JavaScriptu, technologie Macromedia Flash apod. U webového rozhraní databáze benchmarků jsem se rozhodnul těmto technologiích zcela vyhnout a využívat pouze možností klasického HTML.

5.2 Výběr prostředků řešení

5.2.1 Ukládání dat

Ukládat data je možné v různých formátech. Od primitivních textových souborů nebo binárních souborů s pevnou délkou záznamu přes jednoduché databázové formáty typu DBF nebo Berkeley DB až po relační nebo objektové databáze.

1. Volba jednoduchého formátu souboru je výhodná v tom, že si aplikace vystačí s prostředky standardního programovacího jazyka a nemusí spoléhat na dodatečný software a knihovny. Na druhou stranu ovšem programátor musí řešit jak přístup k jednotlivým záznamům, jejich vytváření, mazání, případnou indexaci, kontrolu integritních omezení a podobně. Dále není bez dalšího úsilí možné pracovat nad daty jinými prostředky, než pomocí dané aplikace. Vytváření vazeb mezi daty jednotlivých souborů je opět plně v režii programátora.

2. Řešení typu Berkeley DB umožňuje poměrně komfortní a rychlou práci se záznamy, přístup k záznamům pomocí klíčů (asociativní pole), rozhraní je standardizované a relativně snadno dostupné pro přístup z jiných programů, nicméně problém provázání dat apod. zůstává.

3. Objektová databáze je vhodná pro ukládání komplexně strukturovaných objektových dat. Pro spoustu účelů však logiku programu zbytečně komplikuje a její nasazení nemá výrazné výhody proti nasazení relační databáze.

4. Relační databáze jsou v dnešní době zřejmě nejčastějším úložištěm strukturovaných dat. Umožňují standardním způsobem mapovat data z jednotlivých domén a relace mezi nimi do tabulek. Moderní relační databázové systémy umožňují definovat tabulky s daty nejpoužívanějších typů, vytvářet indexy pro rychlý přístup, stanovit automaticky hlídané integritní omezení, zajistit atomicitu operací apod. Standardním jazykem pro práci s daty v relační databázi je SQL. Relační databáze byla logickou volbou pro dostupnost, rychlost, komfort práce a možnost administrovat data pomocí standardních nástrojů včetně možnosti exportu dat z relační databáze a importu do jiné databáze.

Pro menší projekty typu databáze benchmarků existuje několik vhodných databázových systémů, které jsou platformě nezávislé, dostatečně výkonné a disponují potřebnými funkcemi. Známé jsou například databáze MySQL, PostgreSQL, SQLite apod. Databáze benchmarků je navržena tak, že je v zásadě možné ji po zanedbatelných změnách provozovat nad různými databázovými stroji. Vzhledem k původnímu zadání a potřebám byla zvolena databáze PostgreSQL. Tato databáze je volně šiřitelná, má dostupné zdrojové kódy, získala si dobrou pověst díky svojí stabilitě a pokročilým možnostem. Lze ji provozovat pod různými systémy typu Unix i pod MS Windows.

5.2.2 Webová aplikace

Pro provozování webové aplikace potřebujeme server protokolu HTTP, který dokáže komunikovat s WWW browsery a zpřístupnit jim data uložená na lokálním nebo vzdáleném počítači. Existuje velké množství HTTP serverů, které nabízejí různé možnosti, jak z hlediska rychlosti, tak podpory dynamických stránek, bezpečnosti apod. Zvolené webové rozhraní je možné realizovat různými prostředky a detailní řešení zahrnuje technologii, programovací jazyk, HTTP server i operační systém.

1. Statické HTML stránky nejsou dostatečné pro vkládání dat ani pro jejich parametrizované vyhledávání apod.

2. Klasický CGI přístup. Data jsou zpracovávána programem, který pracuje na klasickém principu programu pro příkazovou řádku. Čte data ze standardního vstupu a výstupu. Tento přístup je relativně univerzální a při použití technologií jako FastCGI vykazuje i rozumných rychlostí. Zvolená abstrakce je ale relativně nízká a tento přístup není v dnešní době příliš oblíběn.

3. Specializovaný jazyk nebo webový framework. Mezi nejpoužívanější technologie patří PHP, ASP a JSP.

Zvolen byl jazyk PHP, který je dynamickým jazykem specializovaným na tvorbu dynamických webových aplikací. Může běžet nad různými HTTP servery a operačními systémy. Má prostředky pro komfortní práci s daty v relačních databázích s využitím jazyka SQL, mnoho různých knihoven dodávaných přímo v základní instalaci nebo dostupných pro rozšíření možností. Jazyk je interpretovaný, ale dosahuje dostatečné rychlosti, kterou je možné dále zvýšit pomocí komerčního nástroje Zend Optimizer. Výhodou PHP je rychlý vývoj a možnost vkládání dynamických prvků přímo do stránky. Existují i řešení snažící se o přísné oddělení aplikační logiky a prezentační vrstvy v PHP aplikacích, ale pro náš projekt toto není příliš zajímavé.

Důležitými argumenty pro volbu PHP bylo využívání jazyka v jiných školních projektech, jeho snadná zvládnutelnost apod.

PHP může běžet i nad jinými servery, ovšem zvolil jsem kombinaci PHP + Apache + Linux, která se osvědčila (nejen) při řešení jiných projektů školy. Pilotní instalace ovšem běží pod operačním systémem FreeBSD a ukázalo se, že z hlediska funkčnosti mojí aplikace jsou oba operační systémy v zásadě ekvivalentní.

5.2.3 Práce se zdrojovými soubory benchmarků

Pro zpracování zdrojových benchmarkových souborů jsem zvolil jazyk Python. Python je dynamický, silně typový, objektově orientovaný jazyk s funkcionálními prvky. Při jeho návrhu byl kladen důraz na rychlý vývoj aplikací, dobrou čitelnost programů třetími osobami a vhodností nasazení i pro řešení konkrétních problémů neprogramátory (např. vědeckými pracovníky různých oborů). Python je interpretovaný jazyk a jeho interpreter je dnes dostupný pro všechny moderní serverové i desktopové výpočetní platformy. Existují rovněž interpretery pracující nad virtuálními stroji JRE a .NET a připravuje se podpora dalších virtuálních strojů, jako jsou Parrot a LLVM. Svoji životaschopnost prokázal jazyk jakožto základ projektů typu Zope, jako skriptovací rozšíření mnohých programů i při budování infrastruktury komerčních firem - Python je masivně nasazen ve firmě Google, která se zabývá vyhledáváním WWW i dalšími internetovými aktivitami.

Python jsem zvolil proto, že umožňuje snadný převod myšlenek do kódu, disponuje slušnou výpočetní rychlostí, je snadno dostupný a obsahuje mnoho nástrojů pro rychlý vývoj a analýzu a testování fungování aplikace.

5.3 Stručný popis základních používaných technologií

Následuje krátký popis použitých technologií a programů, který uvede do problému případného čtenáře, který s nimi není zcela obeznámen.

5.3.1 Linux a FreeBSD

Operační systém Linux je systém unixového typu založený z velké části na základech a předpokladech unixové odnože SYSTEM V. Tento operační systém je velmi populární pro svoji dostupnost zdarma, ale firemní sféra s oblibou používá tzv. enterprise řešení od velkých firem typu RedHat, SUSE nebo IBM. Linux (konkrétně distribuci Gentoo) jsem používal jako vývojovou a referenční platformu. Výhodou je, že distribuce Linuxu obsahují přímo po instalaci nebo po snadném doplnění díky tzv. balíčkovacím systémům všechny programy a knihovny, které jsem k vývoji aplikace potřeboval.

Operační systém FreeBSD patří do unixové větve BSD. Tento operační systém je rovněž k dispozici zdarma, ale za poněkud odlišných licenčních podmínek. Systém má menší podporu dodavatelů hardware a software než Linux, je ale ceněný pro svoji stabilitu a kvalitní návrh. Operační systém FreeBSD spuštěný ve virtuálním prostředí, byl cílovým systémem nasazení mé aplikace. Ukázalo se, že z hlediska aplikačního programátora jsou rozdíly oproti Linuxu zanedbatelné.

5.3.2 Databázový systém PostgreSQL

PostgreSQL je volně dostupný relační databázový systém s vysokým výkonem v náročnějších aplikacích, kvalitní podporou SQL standardů a vysokou flexibilitou. Za zmínku stojí:

Podpora transakcí Transakce jsou plně podporovány - vyhovují požadavkům ACID (atomicita, konzistence, izolace, izolace, trvalost)

Rozšiřitelnost Funkcionalitu databáze lze rozšiřovat pomocí široké škály podporovaných jazyků. PostgreSQL obsahuje jazyk PL/pgSQL, který vychází z PL/SQL firmy Oracle. Široce používané jazyky jako C, C++ nebo Java jsou podporovány rovněž. Ze skriptovacích jazyků lze použít například Python, Perl nebo Scheme.

Podpora triggerů a uložených procedur Všechny jazyky použitelné pro rozšíření funkcionality PostgreSQL mohou být použity pro použití ve vnořených procedurách a triggerech.

Dobrá spolupráce s operačním systémem Databáze umožňuje využívat uživatelské účty operačního systému; kromě klasického internetového TCP/IP spojení je možné na lokálním počítači využít unixové sockety.

5.3.3 HTTP server Apache

Databázový server Apache je v současné době nejpoužívanější HTTP server. Podle serveru netcraft.com používaly koncem roku 2006 servery dostupné pod více než 60% doménových jmen právě server Apache.

Umožňuje snadnou konfiguraci, poskytuje vysoký výkon a podporuje celou řadu technologií pro vytváření dynamických webových aplikací.

Apache umožňuje konkurenční obsluhu klientských požadavků, založenou na systému více vláken nebo více procesů.

Pomocí různých modulů je možno přidávat různé metody autentizace (např. pomocí MD5 hashování, pomocí databáze), podporu dalších protokolů (WebDav), podporu šifrovaného spojení (SSL, TLS), podporu vyvažování zátěže, přepisování URL apod. Moduly slouží i pro podporu dynamických technologií - protokol CGI, dynamické stránky tvořené jazyky PHP, Perl, Python apod.

5.3.4 Jazyk PHP

Jazyk PHP byl navržen speciálně pro vytváření dynamických webových stránek a jeho nasazení je skutečně nejčastější v této oblasti.

PHP má širokou podporu databázových systémů, umožňuje snadnou práci s řetězci, soubory, umožňuje spouštění různých programů a mnoho jiných užitečných činností.

Z pohledu vývojáře webové aplikace mezi nejzajímavější vlastnosti patří:

- Efektivní možnosti vkládání obsahu proměnných do řetězců.
- Snadný přístup k datům poslaných z formulářů - tzv. superglobální proměnné $_GET, $_POST.
- Snadná práce s cookies - cookies patří k hlavním prostředkům, které umožňují překonat problém nestavovosti HTTP protokolu.
- Podpora relací - relace nabízejí možnost, jak zajistit kontinuitu identifikace uživatele na serveru.

PHP umožňuje psát objektově orientované programy založené na klasickém modelu třída - instance třídy, má podporu dědičnosti, kontrolu přístupu k atributům a metodám objektů.

5.3.5 Jazyk Python

Jazyk Python umožňuje psát multiplatformní dynamické programy založené na principech objektového a funkcionálního programování. Programátor je nucen do jisté míry dodržovat základní kódovací styl - odsazování kódu má sémantické důsledky. K výhodám Pythonu patří:

- Podpora násobné dědičnosti.
- Efektivní metody zpracování dat typu iterátorů a generátorů.
- Vysoká dynamičnost vycházející mimo jiné z možnosti dynamického importu modulů, jejich aktualizace za běhu, možnosti získávat pomocí řetězce obsahujícího jméno atributy modulů, tříd, instancí apod.
- Možnost definování standardních operací pro uživatelsky definované třídy.
- Velké možnosti reflexe - zkoumání existujících objektů a jejich vazeb za běhu programu.
- Možnost ladění programu v interaktivním interpreteru.
- Množství knihoven pro nejrůznější oblasti použití - mnohé z nich jsou k dispozici v základní instalaci.

V dnešní době je jazyk Python nasazován v širokém spektru oblastí od náhrady složitějších skriptů unixového shellu přes GUI aplikace až po dynamické webové stránky.

Ve své práci používám programy v jazyce Python běžící z příkazové řádky. Z tohoto hlediska bych chtěl vyzvednout tyto vlastnosti:

- Snadný přístup k parametrům příkazové řádky.
- Snadné zpracování voleb příkazové řádky.
- Snadná práce se soubory.
- Podpora klasických posixových konstrukcí - práce s procesy apod.

6 Možné bezpečnostní problémy zvoleného řešení

Každá aplikace a data, která zpřístupňuje, jsou v reálném světě vystaveny bezpečnostním rizikům. Tato rizika jsou tím závažnější, čím důležitější je bezproblémový chod aplikace a souvisejících dat. Aplikace se ale může stát i vstupní branou pro ovládnutí celého počítače, což může mít ještě daleko závažnější následky.

6.1 Základní druhy ohrožení

Ohrožení bezpečnosti se týká několika oblastí.

- Porušení důvěrnosti dat.
- Ztráta dat.
- Neoprávněná modifikace dat.
- Získání přístupových práv k počítači.
- Přerušení dostupnosti služby.

V praxi se lze setkat s různými pokročilými požadavky z oblasti bezpečnosti, jako je vedení záznamů o provedených změnách a jejich původcích. Tuto problematiku však v mé aplikaci, po konzultaci se zadavatelem, neřeším a nebudu se jí zabývat ani v této kapitole.

Zachování důvěrnosti je založeno na předpokladu, že informace mohou číst pouze osoby oprávněné. Tento předpoklad je splněn tenkrát, když:

- Systém dokáže rozlišovat mezi osobami oprávněnými k přístupu a osobami neoprávněnými.
- Systém zamezí přístupu osobám neoprávněným.

Důvěrnost dat v souborovém systému je dána rozumně nastavenými přístupovými právy, v případě webové aplikace je třeba s tímto požadavkem počítat v návrhu aplikace. O přijatých opatřeních v databázi benchmarků se zmíním v dalším textu.

Ztráta dat může být způsobena nejenom zásahem útočníka, ale i omylem obsluhy, chybou hardware a jinými příčinami. Proto je třeba nejenom bránit útočníkům zvenčí přistupovat k datům, ale je třeba zachovat pravidlo co nejmenších práv pro oprávněné osoby a data pravidelně zálohovat. Zálohování databáze benchmarků spočívá v zálohování uložených souborů a zálohování samotné databáze - například formou tzv. SQL dumpu.

Neoprávněné modifikaci lze opět zabránit kontrolou přístupu a v případě, že situace nastane, obnovením dat ze zálohy.

Získání přístupových práv k počítači je možné obvykle díky chybám v software třetích stran nebo špatnému návrhu programu. O obou možných problémech se zmíním v dalším textu.

Přerušení dostupnosti služby (DoS) je komplexní problém, který lze dobrým návrhem pouze omezit, ale zabránit mu zcela nelze. Obranou před přerušením dostupnosti služby, pokud vyloučíme záměrný útok, je efektivně navržená aplikace.

6.2 Bezpečnostní rizika řádkových skriptů

Řádkové skripty psané v jazyce Python jsou z bezpečnostního hlediska poměrně malou hrozbou.

Parser benchmarků je spouštěn z lokálního počítače a obyčejně i s právy, která odpovídají běžnému uživateli. Jazyk Python je odolný proti chybám typu buffer overflow a neobsahuje žádná volání externích programů, takže nebezpečí jeho zneužití při spuštění z příkazové řádky je minimální.

Určitým problémem by mohlo být jeho spuštění ze serveru s cílem vyvolat DoS útok, který by měl za cíl způsobit zpomalení serveru a zahlcení jeho paměti.

Skript pro přípravu archivu pro agregované stahování by teoreticky mohl být dalším cílem útoku. Uvědomíme-li si ale, že soubory se do archivu přidávají na základě SQL dotazu vytvořeného naším PHP skriptem, je riziko zneužití minimální.

6.2.1 Kontrola přístupu

Kontrola přístupu byla rozdělena na čtyři oblasti:

- Přístup k prohlížení dat benchmarků.
- Přístup k modifikaci dat benchmarků.
- Přístup k editaci uživatelských účtů.
- Možnost změny vlastního hesla.

Prohlížet data benchmarků může podle nastavení aplikace buďto kdokoliv, kdo má přístup na stránky webového rozhraní, nebo pouze osoba přihlášená. Zatímco nutnost přihlašování snižuje pohodlí uživatele, zamezuje to, zvláště v případě vystavení stránky na Internetu, riziku zvýšeného zatížení serveru. Po přihlášení smí prohlížet data kterýkoliv uživatel s platným účtem.

Modifikace dat je v databázi benchmarků povolena pouze pro přihlášené uživatele s příslušným oprávněním.

Uživatelské účty lze modifikovat pouze s příslušným oprávněním. Z bezpečnostního hlediska bych chtěl vyzvednout dva aspekty návrhu: Uživatel nesmí modifikovat vlastní přístupová práva (to má význam zejména jako ochrana před naúmyslným zrušením práv uživatele) a uživatel nevidí hesla jiných uživatelů. Editace hesel je řešena jejich přepsáním, nikoliv editací.

Změna vlastního hesla může být u některých uživatelských účtů zakázáno. Význam to může mít například při hromadném sdělení jednoho uživatelského hesla více osobám (předpokládám přístup pro čtení), kde by zlomyslná nebo nechtěná změna hesla jedním z nich ohrozila přístup ostatních. Při změně vlastního hesla je nutno zadat původní heslo a pro snížení možnosti chyby zadat nové heslo dvakrát.

6.2.2 Vyřešení autorizace uživatele

Autorizace uživatele probíhá zadáním uživatelského jména a hesla. V průběhu práce je identifikace uživatele sledována pomocí tzv. relace za pomocí PHP kódu ve spolupráci s prohlížečem na straně klienta. Relace se ukládá pomocí cookie a reprezentuje ji jednoznačný, náhodně

generovaný identifikátor. Po zadání platného uživatelského jména a hesla je uživatelský účet zpárován s existující relací daného uživatele.

Nebezpečím je možnost odcizení nebo uhodnutí identifikátoru relace. Uhodnutí, předpokládáme-li rozumnou implementaci generátoru identifikátorů, nepřichází příliš v úvahu.

Odcizení identifikátoru je podstatně reálnější hrozbou. Existují tři místa, kde by bylo teoreticky možné cizí identifikátor získat: Počítač klienta, server a přenosový kanál mezi klientem a serverem. Získání identifikátoru na klientském počítači není nic, co bychom mohli ovlivnit, protože je plně v rukou správce klientského počítače, aktuálního uživatele počítače a klientského programu, tj. webového browseru. Odcizení na straně serveru je možné díky úspěšnému útoku na server a zabezpečení serveru jako takového na tomto místě řešit nebudu. Ochranou přenášených dat se zabývám dále v textu.

6.2.3 Ochrana před nahráním příliš velkého souboru

Nahrání příliš velkého souboru může mít za následek zahlcení serveru. Základní ochranou je omezení velikosti nahrávaných souborů přímo na

6.2.4 Ochrana přenášených dat

Přenášená data jsou ohrožena rizikem, že k nim získá přístup neoprávněná osoba. Ochrany dat přenášených po síti se dá docílit šifrovaným spojením (SSL, TLS). V návrhu databáze šifrování spojení neřeším, ale je možné zajistit šifrování spojení vhodnou konfigurací webového serveru.

6.2.5 Ochrana softwarového prostředí prostředky třetích stran

Každý program může obsahovat bezpečnostní chybu. U kompilovaných programů, zejména těch, které jsou psány v jazycích typu C, je riziko poměrně vysoké. Mezi nejznámější problémy patří přetečení bufferu, které, i přesto, že příčiny i řešení jsou vcelku obecně známé, se čas od čas často vyskytuje i ve známých a oblíbených programech.

Pro interpreter PHP existuje projekt zvaný *Hardened-PHP Project*, zvaný též Suhosin. Tento projekt obsahuje nízkoúrovňovou ochranu proti přetečení bufferu a zranitelnosti formátovacích řetězců a ochranu, která řeší některé potenciální problémy v návrhu samotné aplikace psané v PHP.

Pro ochranu hostitelského počítače před útokem pomocí webové (nebo jiné) aplikace lze též využít některou z možností izolace aplikace. Mezi oblíbené možnosti patří tzv. chroot prostředí - program běží v rámci samostatného podadresáře, který simuluje celý souborový systém a operační systém mu znemožní překročit hranice tohoto podadresáře. Další možností je využití některé z možností virtualizace - mezi oblíbené možnosti patří například VMWare, Xen, Usermode Linux.

Pilotní instalace databáze benchmarků využívá jak projektu Suhosin, tak i virtuálního serveru, který projekt sdílí s některými jinými projekty. Konkrétní rozhodnutí je ale v tomto případě vždy v rukou správce serveru či aplikace.

7 Popis implementace

7.1 Propojení webové aplikace v PHP a programů v jazyce Python

Při práci s benchmarky a jejich popisem nastává jsem použil dva programovací jazyky, což do problému vnáší heterogenitu. Taková situace v praxi nastává často - příčinou může být nutnost propojení dvou či více existujících systémů používajících různé technologie, vhodnost nějakého jazyka pro nasazení na dílčí úkol a podobně.

V našem případě nastávají dva problémy, které je třeba vyřešit:

- Nutnost komunikace mezi různými prostředími.
- Nutnost duplicitního nastavení.

7.1.1 Možnosti komunikace mezi heterogenními programy

Jelikož jde o samostatné programy, odpadá řešení kódové vazby a problém se redukuje na společnou komunikaci. Možnosti komunikace jsou například následující:

- Předání informací volanému programu pomocí příkazové řádky.
- Možnost propojení rourou.
- Propojení pomocí socketového propojení.
- Předání parametrů pomocí souboru.

Při komunikaci mezi databázovou aplikací a pomocnými programy v Pythonu používám předání základních informací pomocí příkazové řádky a vytvoření souboru pro další použití - výhodou je asynchronní komunikace. Konkrétnímu použití obou metod se budu věnovat na jiném místě.

7.1.2 Vyřešení problému duplicitního nastavení

Nutnost dvojí konfigurace přináší zásadní problém v tom, že oba programy přistupují ke stejným zdrojům a komunikačním kanálům. Pokud dojde k jakékoliv změně, je třeba změnu provést na dvou místech najednou, což přináší možnost chyb a opomenutí.

Existují tři možnosti, jak se k problému postavit:

1. Problém ignorovat a ponechat dvojí nastavení. Pokud počet voleb nastavení není mnoho, může tento přístup vyhovovat, neboť při omezeném počtu nastavených možností je vysoká pravděpodobnost, že se na problém okamžitě přijde. Při rozrůstání aplikace a zvyšování složitosti to však často přestává platit.
2. Konfiguraci uložit do společného konfiguračního souboru. Nevýhodou je nutnost udržovat nějaké neutrální umístění, které bude přístupné oběma aplikacím. Další nevýhodou je nutnost použít nějaký společný formát, například XML nebo Windows INI, což přináší jednak komplikovanější zpracování (potřeba použít příslušnou knihovnu nebo implementovat zpracování v obou jazycích) a jednak to znamená větší časovou náročnost, protože se tím zabraňuje interpreteru zkrátit čas načtení vlastními prostředky (cache interpreteru, přeložený modul *.pyc).

3. Vytvořit metakonfigurační soubor a vygenerovat z něj konfigurační moduly pro obě aplikace. V našem případě pythonovský modul (.py) a vkládací soubor PHP (.inc.php). Výhodou je větší rychlost zpracování a zjednodušení obou aplikací.

Zvolil jsem cestu metakonfiguračního souboru. Skript GenConfigFiles.py, který jsem vytvořil, dokáže ze souboru se syntaxí

```
název = hodnota
```

vygenerovat moduly v PHP a Pythonu, které se pak používají v obou částech aplikace. Zdrojový soubor může vypadat třeba následovně:

```
dbName                  = 'benchmark'
dbUser                  = 'jarda'
dbPort                  = '5432'
benchmarkFilesDirectory = '/home/jarda/projects/benchmarks/benchmarkFiles'
tmpDirectory            = '/home/jarda/projects/benchmarks/tmpFiles'
execDirectory           = '/home/jarda/projects/benchmarks/parseInfo'
existenceFileExtension  = 'existence'
archiveFileExtension    = 'zip'
```

Vygenerovaný konfigurační soubor v Pythonu:

```
# This file is generated by GenConfigFiles script

class BenchConfig(object):
    dbName                      = 'benchmark'
    dbUser                      = 'jarda'
    dbPort                      = '5432'
    benchmarkFilesDirectory     = '/home/jarda/projects/benchmarks/benchmarkFiles'
    tmpDirectory                = '/home/jarda/projects/benchmarks/tmpFiles'
    execDirectory               = '/home/jarda/projects/benchmarks/parseInfo'
    existenceFileExtension      = 'existence'
    archiveFileExtension        = 'zip'
```

Vygenerovaný konfigurační soubor v PHP:

```
<?php
# This file is generated by GenConfigFiles script

    class BenchConfig
    {
        static $dbName                  = 'benchmark';
        static $dbUser                  = 'jarda';
        static $dbPort                  = '5432';
        static $benchmarkFilesDirectory  = '/home/jarda/projects/benchmarks/benchmarkFile
        static $tmpDirectory            = '/home/jarda/projects/benchmarks/tmpFiles';
        static $execDirectory           = '/home/jarda/projects/benchmarks/parseInfo';
        static $existenceFileExtension   = 'existence';
        static $archiveFileExtension     = 'zip';
    }
?>
```

Nyní zbývá pouze importovat daný modul v cílovém jazyce a používat statické proměnné konfiguračních tříd. Příklad v Pythonu:

```
from BenchConfig import BenchConfig
print BenchConfig.dbName
```

Příklad v PHP:

```
require_once('config.inc.php')
echo BenchConfig::$dbName;
```

7.2 Řádkový analyzátor základních formátů benchmarků

7.2.1 Úvod

U zdrojových textů běžných programovacích jazyků je možné vytvářet syntaktické analyzátory (parsery) na základě známé gramatiky. Já jsem ve své práci stál před úkolem zpracovat soubory poměrně jednoduché struktury, ale nepříliš jasné gramatiky. V zásadě jsem vyšel z toho, že se soubory skládají z jednoduchých deklarací řádkového typu nebo mohou být na virtuální řádky převedeny tam, kde jde o deklaraci např. výrazu na více fyzických řádcích a deklarace je ukončena středníkem. Tento přístup výrazně usnadňuje následné zpracování.

7.2.2 Rozhraní programu

Program je spouštěn z příkazové řádky a výsledky vypisuje na standardní výstup. Syntaxe je následující:

./ParseInfo.py [-t *formát*] *seznam souborů*

Zadání formátu je nepovinné a aplikuje se na všechny soubory ze seznamu, pokud je zadáno. Jinak jsou formáty rozpoznávány z přípony. Explicitní uvedení formátu je nutné pro zpracování souborů s nestandardními příponami.

Příklady použití:

```
./ParseInfo.py s832.slif 5xp1.blif
./ParseInfo.py -t pla 5xp1.txt
```

Příklad výstupu:

```
** Benchmark file info parser **

Parsing file s832.slif

BUF count: 0
NOT count: 25
Outputs count: 19
OR count: 64
NAND count: 54
NOR count: 66
Gates count: 287
```

```
Inputs count: 19
DFF count: 0
AND count: 78
```

7.2.3 Volba lexikálního analyzátoru

Vzhledem k tomu, že jsem zvolil programovací jazyk Python, musel jsem vycházet z prostředků, které tento programovací jazyk má k dispozici. A různých prostředků existuje celá řada, počínaje jednoduchými lexikálními analyzátory až po komplexní systémy typu ANTLR. Nakonec jsem zvolil modul pro lexikální analýzu s názvem Plex. Výhodou Plexu je jeho snadno srozumitelná, syntaxe, kterou v následujícím textu ukáži na příkladu. Kód lexikálního analyzátoru vytvořeného pomocí nástroje Plex není vygenerovaný, ale každá změna v definici syntaxe se okamžitě projeví, což koresponduje s konceptem interpretovaných jazyků typu Python. Lexikální analyzátor vytvořený pomocí Plexu rovněž vykazoval poměrně slušný výkon v porovnání s některými jinými testovanými alternativami.

7.2.4 Analýza souboru

Analýza probíhá pomocí instance třídy BenchmarkFileParser, která využívá lexikální analyzátor vytvořený metodou createScanner(). Lexikální analyzátor je instancí třídy LinesProducingScanner(), která generuje symboly pro jednotlivé neprázdné řádky. Lexikální analyzátor si lze představit jako funkci A, jejíž parametry jsou vstupní soubor F a lexikon L a výstupem je posloupnost řádků symbolů S:

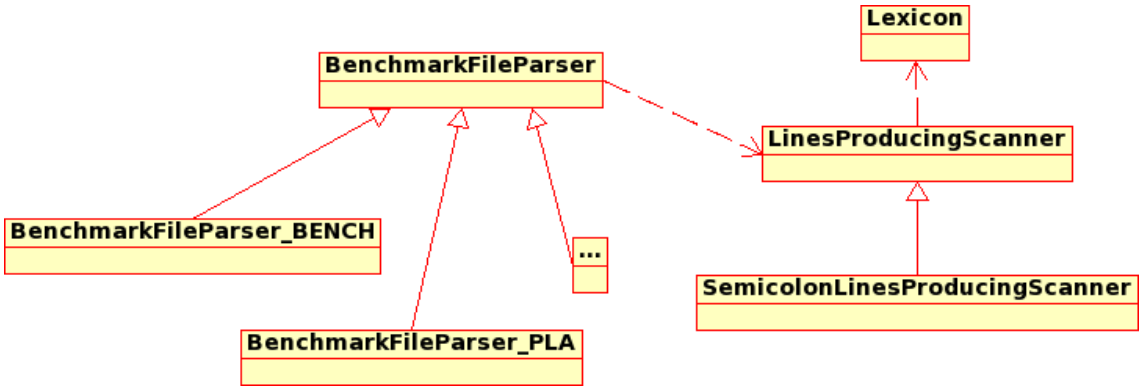
$S = A(F, L)$

7.2.5 Příklad definice lexikonu

```
self.lexicon = Lexicon([
    (Str('#') + Rep(AnyBut('\n')), ScannerConstants.T_COMMENT,),
    (Str('.inputs'), ScannerConstants.T_INPUTS,),
    (Str('.outputs'), ScannerConstants.T_OUTPUTS,),
    (Str('.names'), ScannerConstants.T_NAMES,),
    ScannerConstants.R_NEWLINE,
    ScannerConstants.R_IGNORE_SPACES_AND_TABS,
    ScannerConstants.R_TOKEN,
])
```

7.2.6 Hierarchie tříd

Pro bližší objasnění fungování parseru ukážu UML diagram základních použitých tříd:



Obrázek 7.1: Základní třídy analyzátoru

Pro každý formát je definována samostatná podtřída třídy BenchmarkFileParser. Tato podtřída analyzuje řádky získané z instance třídy LinesProducingScanner, která vytváří řádky symbolů na základě zadané instance třídy Lexicon().

7.2.7 Využití dynamických vlastností jazyka Python pro flexibilní podporu formátů

Program využívá možnosti dynamického sestavování názvů tříd a dynamického importu pomocí následujícího kódu:

```
# get the format ID
benchmarkFormat = Constants.EXT_TO_BENCHMARK_FORMAT[lowerCaseName]

# get the internal format name
internalBenchmarkFormatName = Constants.INTERNAL_BENCHMARK_FORMAT_NAME[benchmarkFormat]

# construct module name from the prefix and internal format name
parserModuleName = Constants.BENCHMARK_FILE_PARSER_MODULE_PREFIX
    + internalBenchmarkFormatName

# import the module
parserModule = __import__(parserModuleName)

# construct the class name
parserClassName = Constants.BENCHMARK_FILE_PARSER_CLASS_PREFIX
    + internalBenchmarkFormatName

# get the parser class
parserClass = getattr(parserModule, parserClassName)
```

Dynamismus získání potřebné třídy je zajištěn funkcemi __import__ a getattr. Tyto metody umožňují dokonce přidávat moduly a atributy za běhu aplikace, což v tomto případě není třeba. Výhodou zmíněného dynamismu je vynucení jednotného pojmenování modulů a tříd pro parsery, což zajistí lepší čitelnost a udržovatelnost kódu. Pro přidání dalšího formátu je třeba:

- Vytvořit novou konstantu pro daný formát.
- Přidat interní název formátu.
- Vytvořit modul s názvem odpovídajícím stanovenému schématu a internímu názvu formátu, který obsahuje třídu s názvem odpovídající stanovenému schématu a internímu názvu formátu.

Funkce, která se stará o vybrání správného parseru, se nemění.

7.2.8 Vyřešení závislosti analyzátoru na interpreteru jazyka Python

Interpretované jazyky přinášejí nutnost vyřešení problematiky distribuce na počítače, kde interpreter jazyka není nainstalován. V případě jazyka Python existují dvě řešení problému:

- Nainstalovat plnohodnotný interpreter.
- Vytvořit distribuci programu se zabaleným interpreterem, pokud možno redukovaným tak, aby zajišťoval pouze funkce našeho programu a nenesl s sebou redundantní data.

Zatímco první cestu lze, pokud je to z nějakého důvodu výhodné (např. distribuce Linuxu často již obsahují interpreter Pythonu a dokonce jej používají pro základní činnosti nutné k běhu či údržbě), pro distribuci ke koncovému uživateli, zejména na platformě MS Windows je výhodnější vytvořit distribuci se zabaleným interpreterem. V následujícím textu demonstruji konkrétní řešení.

7.2.8.1 Vytvoření distribuce programu pomocí rozšíření py2exe

py2exe je rozšíření standardního mechanismu pro tvoření distribučních balíčků a automatizovanou instalaci s názvem Distutils. Po nainstalování py2exe je třeba vytvořit skript setup.py. Jeho základní podoba pro náš program má následující podobu:

```
from distutils.core import setup
import py2exe

setup(console=['ParseInfo.py'])
```

Tento skript vytvoří distribuci na základě hlavního skriptu s názvem ParseInfo.py. Distribuce se skládá ze spustitelného programu (ParseInfo.exe), interpreteru obsaženého v dynamické knihovně (Python24.dll), modulů potřebných pro práci programu (Library.zip) a několika pomocných souborů.

Rozšíření py2exe využívá analýzu příkazů import. Dynamická stavba programu s využitím dynamických importů tuto analýzu znesnadňuje a py2exe proto dynamické moduly vynechá. Řešením je jejich explicitní uvedení ve skriptu setup.py:

```
from distutils.core import setup
import py2exe

setup(console=['ParseInfo.py'] options={'py2exe : { 'includes' : [
    'BenchmarkFileParser',
    'BenchmarkFileParser_BENCH',
```

```
    'BenchmarkFileParser_BLIF',
    'BenchmarkFileParser_KISS2',
    'BenchmarkFileParser_PLA',
    'BenchmarkFileParser_SLIF',
    'Constants',
    'LinesProducingScanner',
    'ScannerConstants',
]}})
```

Po spuštění příkazu

```
python setup.py
```

se vytvoří adresář dist obsahující všechny potřebné soubory. Tento způsob distribuce má dva nedostatky:

1. Nutnost hlídání přidávaných souborů, což snižuje výhody dynamické stavby programu.
2. Nekompaktnost takovéto distribuce - nutnost zabalit vzniklé soubory například do archivu typu zip.

Automatické přidávání modulů do distribuce lze vyřešit několika způsoby. Řešení, které zde předvedu, využívá toho, že ve skriptu setup.py lze používat běžné konstrukce jazyka Python a pro funkci setup() připravit parametry dynamicky. Následující příklad řeší řeší jednoduše dynamické přidávání modulů pro jednotlivé formáty. Jednotná stavba názvů modulů situaci výrazně usnadňuje:

```
from distutils.core import setup
import py2exe
import glob

includes = [
    'BenchmarkFileParser',
    'Constants',
    'LinesProducingScanner',
    'ScannerConstants',
]

includes += [filename[:-3] for filename in glob.glob('BenchmarkFileParser_*.py')]

setup(console=['ParseInfo.py'] options={'py2exe : { 'includes' : includes}})
```

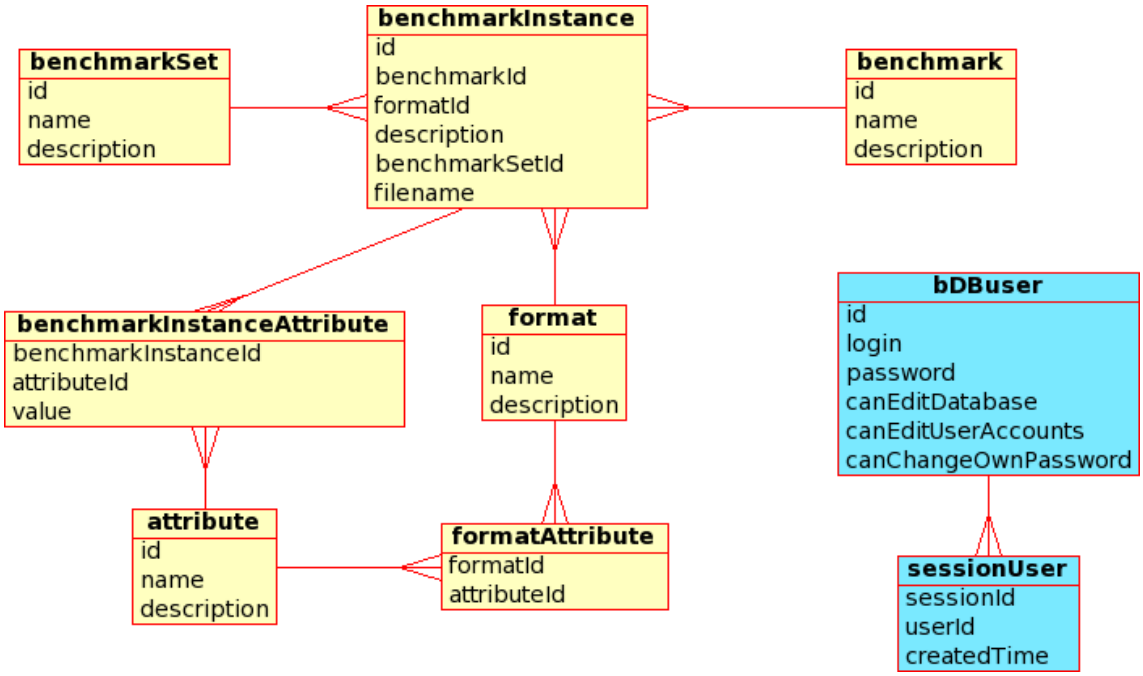
V příkladu jsme s výhodou použili modul glob pro práci se seznamem souborů vytvořeným pomocí vzoru a konstrukci 'list comprehension' a ořezání přípon souborů.

Problém s kompaktností distribuce lze vyřešit použitím některého systému pro snadné vytváření instačních programů pro platformu MS Windows, například systému Inno Setup. Touto problematikou se zde zabývat nebudu, použití je triviální.

7.3 Databáze benchmarků - struktura

7.3.1 Úvod

V databázi benchmarků se nachází celkem 9 tabulek. 7 z nich je použito pro uložení vlastních dat a 2 slouží pro zabezpečení přístupu.



Obrázek 7.2: Struktura databáze

7.3.2 Tabulky pro zabezpečení přístupu

Pro uložení permanentních záznamů o jednotlivých uživatelích a jejich právech slouží tabulka bDBUser (zkratka z benchmark DB User), která obsahuje přístupové jméno uživatele, jeho heslo, ID a určuje, zda má uživatel právo k editaci dat a uživatelů.

Tabulka sessionUser slouží k přiřazení uživatele k jeho relaci.

7.3.3 Tabulky pro uložení dat o benchmarcích

Tyto tabulky lze rozdělit do tří typů: tabulky "číselníkové", které umožňují kategorizaci benchmarků, tabulku instance benchmarku, která přímo koresponduje s nějakým benchmarkovým souborem a tabulky spojovací, které konkretizují vlastnosti a možnosti vložených dat.

Tabulka benchmarkSet obsahuje jednotlivé sady benchmarků.

Tabulka format obsahuje podporované formáty.

Tabulka benchmark slouží jako úložiště "tříd" benchmarků. Typicky v tabulce benchmark jeden záznam představuje určitý obvod a k němu se vztahuje jedna nebo více instancí. Kromě názvu obsahuje i obecný popis.

Tabulka attribute definuje atributy, které popisují benchmark - atributem může být například počet hradel určitého typu nebo počet termů.

Tabulka benchmarkInstance obsahuje informace o konkrétním benchmarkovém souboru. Obsahuje odkaz na benchmarkovou sadu, formát, "třídu" benchmarku, jméno uloženého souboru a popis instance benchmarku (např. informaci o tom, že jde o optimalizovanou implementaci).

Tabulka formatAttribute říká, které atributy mohou být definovány ve kterém formátu. To slouží jako vodítko pro ruční zadávání údajů.

Tabulka benchmarkInstanceAttribute obsahuje hodnotu atributu v dané instanci.

7.3.4 Používané datové typy

V databázi jsem použil pouze 2 datové typy:

- INTEGER pro identifikátory jednotlivých záznamů (ID),
- CHARACTER pro datové sloupce.

Datové sloupce jsou opět dvojího typu:

- Běžné, které jsem omezil velikostí 100 znaků,
- sloupce dlouhých popisů, omezené 2000 znaky.

Tyto délky jsou zvoleny tak, aby poskytly dostatečnou rezervu pro velikost užitečných dat, ale zároveň omezily rozpínání databáze při případných nesprávných vstupech.

7.3.5 Použitá omezení

Omezení (constraints) slouží k hlídání integrity dat. V databázi jsem použil následující omezení:

- UNIQUE - zaručuje jedinečnost hodnoty daného sloupce v každém řádku (například název formátu),
- FOREIGN KEY - svazuje hodnotu sloupce s hodnotou sloupce v jiné tabulce; například sloupec formatId v tabulce benchmarkInstance musí obsahovat ID některého řádku z tabulky format, jinak daný řádek v tabulce benchmarkInstance postrádá význam.

7.4 Implementace databázové aplikace

7.4.1 Úvod

Webové rozhraní databáze benchmarků řeší několik záladních požadavků. Snažil jsem se zajistit, aby aplikace poskytovala alespoň minimální potřebnou bezpečnost, aby byla schopná pracovat nad různými databázovými stroji bez nutnosti rozsáhlých změn a aby její kód byl přehledný a snadno udržovatelný. Následující text bude tedy popisovat nejenom základní funkčnost aplikace, ale i aspekty bezpečnostní, techniky přístupu k databázi a základní strukturu, na níž je aplikace postavena.

7.4.2 Zachování důvěrnosti informací

Existuje několik způsobů, jak ve webové aplikaci zajistit tyento požadavek. Mezi způsoby, které naprosto nevyhovují z hlediska bezpečnosti a flexibility, patří například omezení přístupu na pevně danou IP adresu a sledování odkazující stránky. Zvažoval jsem dva způsoby kontroly oprávnění:

Klasickým řešením je použít tzv. HTTP autentizaci, která využívá schopnosti HTTP serveru povolit přístup k vybraným zdrojům pouze některým uživatelům (soubor .htaccess). Tento způsob má mnohé nevýhody:

- Nutnost přímé editace souborů na serveru.
- Systém oprávnění je spjat se souborovou strukturou na serveru.
- Změny v aplikaci vyžadují často změny v souboru .htaccess - snadná možnost zanesení chyby.

Řešení, které jsem zvolil já, využívá uživatelských účtů uložených v databázi, které mohou mít tři úrovně oprávnění. Dále je možné v konfiguraci přepnout možnost prohlížení a stahování benchmarků pouze pro určité osoby nebo pro všechny návštěvníky bez nutnosti přihlášení.

7.4.3 Přístup aplikace k databázi

Pro přístup k databázi byla zvolena knihovna AdoDB. Důvodu pro její zvolení byly:

- Podpora mnoha volně dostupných i komerčních databázových systémů.
- Vysoká rychlost.
- Vysoká úroveň abstrakce a zároveň podpora potřebných vlastností databáze.
- Možnost nezávislé slovníkové definice databázové struktury.
- Existence této knihovny i pro jazyk Python, což zajišťuje jednotný styl přístupu k datům ve webové aplikaci i pomocném řádkovém programu.

Tenkou vrstvu rozhraní ke knihovně AdoDB představuje soubor database.inc.php, který zajišťuje připojení k databázi na základě konfiguračního souboru.

7.4.4 Souborová struktura aplikace

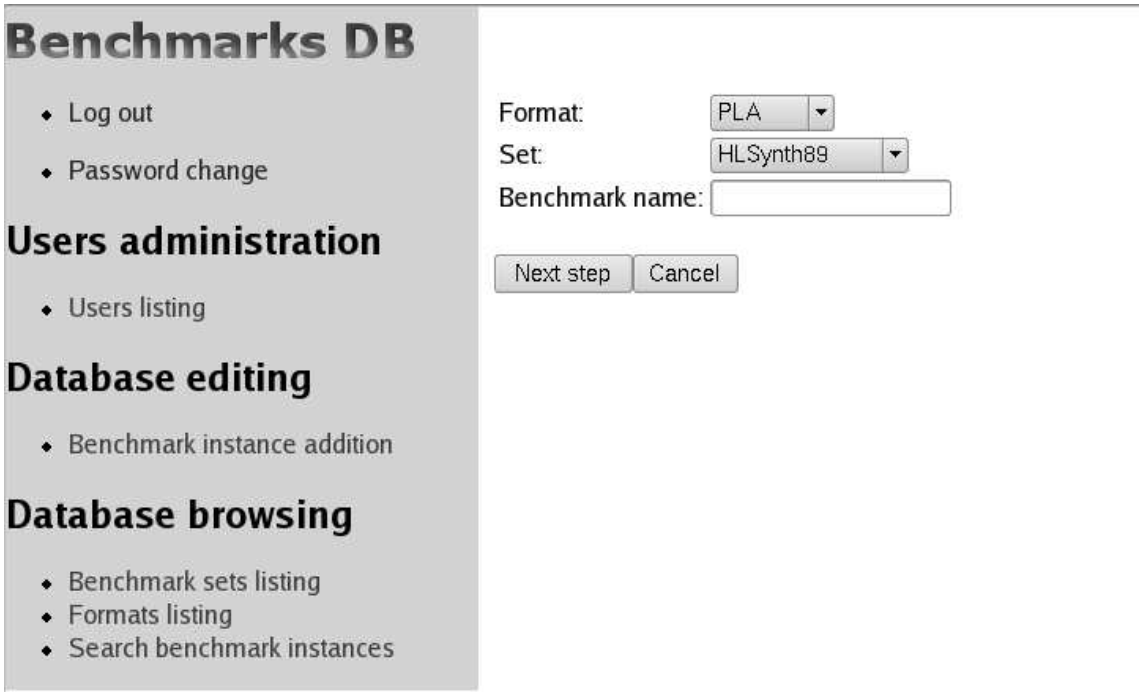
Kód aplikace je tvořen soubory v jazyce PHP. Zobrazované soubory mají název*zaklad.php*, pomocné soubory vkládané do jiných souborů mají název *zaklad.inc.php*. Indexový soubor se nazývá *index.php*, ostatní zobrazované soubory mají, až na výjimky, názvy tvořené podle šablony *typdat_TYPAKCE.php* - například benchmark_ADD.php. Použité názvy akcí jsou:

- EDIT - stránka s formulářem pro editaci dat
- CHANGE - stránka, která provede změnu v databázi podle údajlů z formuláře EDIT akce
- NEW - stránka s formulářem pro vytvoření nových dat
- ADD - stránka, která vloží data zadaná formulářem ve stránce NEW
- SHOW - zobrazení dat

- SEARCH - vyhledání dat
- DOWNLOAD - stažení souboru
- LISTING - zobrazení výpisu
- DELETE - vymazání dat

7.4.5 Základní uživatelské rozhraní

Uživatelské rozhraní aplikace je dáno dvousloupcovým plovoucím designem založeným na kaskádových stylech (CSS). V levé části stránky se nachází menu, v pravé části samotná zobrazovaná data nebo zadávané hodnoty.



Obrázek 7.3: Rozvržení stránky

7.4.6 Objektová struktura aplikace

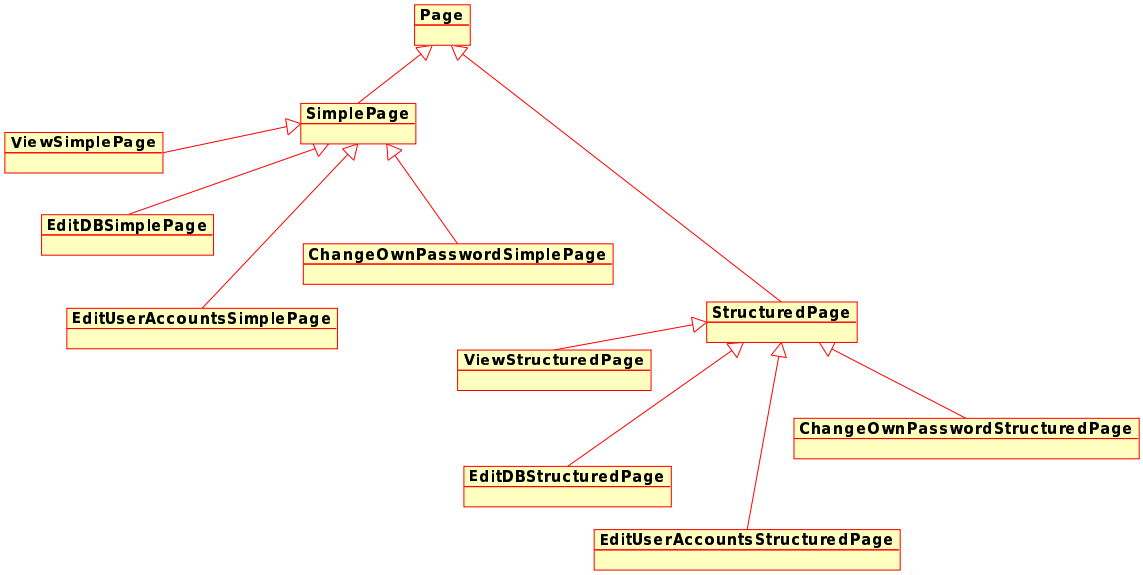
Základním kamenem objektové hierarchie je třída Page. Instance některé z podtříd této třídy je obsažena na každé zobrazené stránce.

7.4.6.1 Třída Page

Třída Page definuje metody, které jsou používány konkrétními potomky této třídy. Jedná se o metody, které umožňují zjistit, zda má aktuální uživatel právo provést akci určitého typu, dokáže vytvořit standardní HTML hlavičku včetně kódování, kaskádového stylu a podobně. Ve třídě Page je také definována statická proměnná $allow_view_for_anyone, která určuje, zda právo prohlížení a stahování benchmarků mají i nepřihlášení uživatelé.

7.4.6.2 Metoda run() a funkce launchPage()

Metoda run() spustí standardní proces tvorby stránky s využitím metod definovaných v podtřídách třídy Page(). Kód metody je následující:



Obrázek 7.4: Hierarchie tříd stránek

```
function run()
{
    $this->initSession();
    $this->ensureUserHasNecessaryPermission();
    $this->customInitialization();
    if (!$this->checkEnvironment())
        $this->handleInvalidEnvironment();
    else
        $this->createPage();
}
```

Význam kódu je následující: Nejprve se inicializuje relace, která překlenuje nestavovost HTTP protokolu a zajišťuje kontinuitu identifikace přihlášeného uživatele. Po identifikaci se metoda ujistí, že uživatel má potřebná práva - má potřebnou úroveň oprávnění podle metody getRequiredAccessLevel(); v případě selhání se vypíše chybová hláška a zpracování končí. Poté dostanou podtřídy možnost provést vlastní inicializaci, například předpřipravit nějaká pomocná pole. Poté podtřída může otestovat, zda je prostředí v pořádku - např. jestli jsou připravena potřebná data v superglobálních proměnných $_GET či $_POST. Pokud se zjistí nedostatek, proběhne jeho ošetření, jinak dostane podtřída za úkol vytvořit samotnou stránku.

Funkce launchPage() zajistí vytvoření instance třídy Page() a spuštění její metody run():

```
function launchPage()
{
    $pageObject = new PageClass();
    $pageObject->run();
}
```

7.4.6.3 Struktura typické stránky

Typická stránka se skládá ze tří částí - importu souboru definujícího předka třídy stránky, vytvoření třídy stránky a zavolání funkce launchPage():

```
<?php
    require_once('simplePage.inc.php');

    class PageClass extends EditDBSimplePage
    {
        # Zde se nachází implementace konkrétních metod
    }

    launchPage();
?>
```

Na rozdíl od ranné verze webového rozhranní tento jednotný styl zajišťuje velmi dobrou přehlednost a srozumitelnost, což napomůže udržovatelnosti a rozšiřitelnosti aplikace. Z kódu je možno vyčíst následující:

- Z volání funkce require_once() je vidět, do které větve stránka patří (simple / structured).
- Z deklarace předka třídy PageClass je navíc na první pohled zřejmé, jaká je potřeba úroveň oprávnění uživatele.
- Volání funkce launchPage() zajistí jednotný proces zpracování stránky.
- V kombinaci se znalostí funkce launchPage() můžeme říci, že třída stránky se musí vždy nazývat PageClass.

7.4.6.4 Třída SimplePage a třída StructuredPage

Jak je vidět z hierarchie tříd uvedené výše, jsou podtřídy třídy Page rozděleny do dvou větví:

- Třída SimplePage je určena k vytváření stránek, které jsou určeny zejména ke zpracování dat a obvykle po zpracování provede přesměrování na některou jinou stránku podle kontextu dané akce.
- Třída StructuredPage zobrazuje klasický design aplikace - hlavní menu a specifický obsah.

Podtřídy třídy SimplePage implementují metodu createPage(). Implementace obyčejně obsahuje práci s databází a následné přesměrování na jinou stránku. Může však i zobrazit nějaký obsah; k vytvoření základní hlavičky lze s výhodou použít metodu getHeader().

Třída StructuredPage implementuje metodu createPage() tak, že zobrazí základní rozložení stránky a kontextové menu a zavolá metodu printContents(), která vytvoří obsahovou část stránky.

7.4.7 Řízení přístupu

Třída Page() ve své metodě run() hlídá, zda připojený uživatel disponuje potřebnou úrovní oprávnění. Jak je vidět z hierarchie tříd, názvy listových tříd hierarchického stromu mají předponu, která udává, jakou úroveň oprávnění musí uživatel mít, aby dostal přístup na stránku vytvořenou instancí dané třídy.

7.4.7.1 Proces přihlášení uživatele

Uživatel je při pokusu o přihlášení vyzván k zadání uživatelského jména a hesla. Po jejich zadání je přihlášení provedeno následujícím kódem:

```
$db = connect();

$query = "SELECT * from bDBUser " .
         "WHERE login = '$_POST[login]' " .
         "AND password = '$_POST[password]'";

$recordSet = $db->Execute($query);

$row = $recordSet->FetchRow();

if ($row) {
    session_name('benchmark');
    session_start();
    session_regenerate_id();

    # remove old records from session user table {{{
    $date = date("Y-m-d H:i:s", strtotime('-2 days'));

    $db->Execute("DELETE FROM sessionUser WHERE createdTime < '$date'");
    # }}}

    # add a new a new record to user session table {{{
    $date = date("Y-m-d H:i:s");

    $sessionId = session_id();

    $query = "INSERT INTO sessionUser (sessionId, userId, createdTime) " .
             "VALUES ('$sessionId', $row[id], '$date')";

    $db->Execute($query);
    # }}}

}
```

Napřed je provedeno vyhledání záznamu o uživateli v příslušné tabulce. Pokud je uživatel s daným uživatelským jménem a heslem nalezen, vytvoří se nová relace, vymažou se staré páry ID relace : ID uživatele (proces slouží i pro čištění starých záznamů z tabulky) a do tabulky se vloží nový pár. Od této chvíle je relace spojena s přihlášeným uživatelem.

7.4.7.2 Identifikace uživatele a zjištění jeho vlastností

Jak jsem uvedl výše, relace zajišťuje kontinuální identifikaci uživatele. Pomocí relace lze získat informace o uživateli včetně přístupových oprávnění. Tyto informace jsou získány pomocí metody getUserInfo():

```
function getUserInfo()
{
```

```
    $db = connect();

    $sessionId = session_id();

    $query = "SELECT bDbUser.id AS userId, "
             "bDBUser.canEditDatabase AS canEditDatabase, " .
             "bDBUser.canChangeOwnPassword AS canChangeOwnPassword, " .
             "bDBUser.canEditUserAccounts AS canEditUserAccounts, " .
             "bDBUser.password AS password " .
             "FROM bDBUser, sessionUser " .
             "WHERE sessionUser.sessionId = '$sessionId' AND " .
             "bdbUser.id = sessionUser.userId";

    $recordSet = $db->Execute($query);

    return $recordSet->FetchRow();
}
```

7.4.7.3 Proces odhlášení uživatele

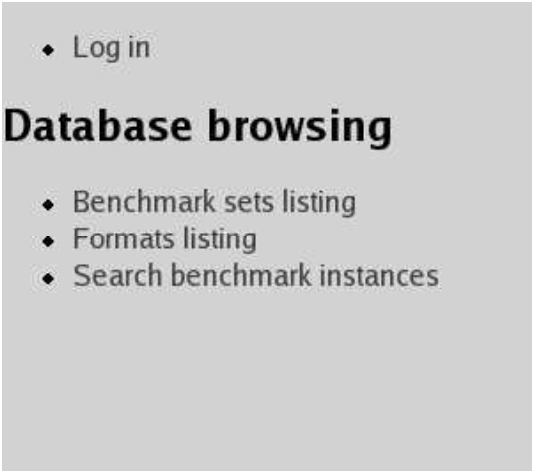
Odhlášení uživatele spočívá ve zrušení vazby jeho relace k uživatelskému účtu. Následující třída pochází ze souboru logout.php a demonstruje zároveň jednoduché použití podtřídy SimplePage:

```
class PageClass extends ViewSimplePage
{
    function createPage()
    {
        $sessionId = session_id();

        $db = connect();

        $db->Execute("DELETE FROM sessionUser WHERE sessionId = '$sessionId'");

        session_regenerate_id();

        header("Location: index.php"); /* Redirect browser */
    }
}
```

Kód je jednoduchý: Z databáze je vymazáno párování ID relace a ID uživatele a poté je ještě ID relace změněno. Po odhlášení je prohlížeč uživatele přesměrován na hlavní stránku.

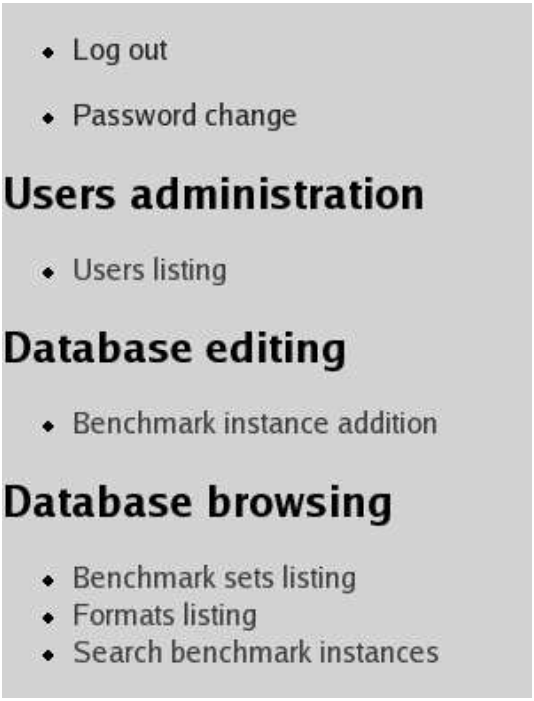
7.4.7.4 Kontextově závislá nabídka operací

Hlavní nabídka neboli menu je závislá na úrovni oprávnění uživatele. Pro demonstraci ukážu menu uživatele, jenž má právo pouze prohlížet obsah databáze:



Obrázek 7.5: Základní nabídka operací

a nabídka uživatele s maximálním oprávněním:



Obrázek 7.6: Maximální nabídka operací

7.4.8 Implementované funkce aplikace

Webové rozhraní je poměrně rozsáhlé a nemá smysl podrobně popisovat všechny funkce. Zaměřím se pouze na nejzajímavější použité techniky a krátce popíši základní implemento-

vané operace.

7.4.8.1 Vytvoření tabulek

Vstupem na stránku createTables.php je vytvořena databázová struktura - všechny potřebné tabulky. Tato stránka není chráněna heslem, neboť její opakované spuštění nezpůsobí žádné škody - tabulky jsou již vytvořeny a proto je akce ignorována.

Tabulky se vytvářejí pomocí knihovny AdoDB s využitím tzv. slovníkové definice. Ta umožňuje co největší přenositelnost definice při zachování co nejširších vyjadřovacích prostředků. Dokonce dokáže emulovat některé vlastnosti například pomocí triggerů. Příkladem je vytvoření autoinkrementačních sloupců.

Slovníkové definice lze sestavit dvěma způsoby; já jsem použil a demonstrovat zde budu, novější, deklarativní styl. Na následujícím příkladu předvedu definici tabulky v AdoDb s použitím slovníkové definice:

```
$tableName = 'benchmarkInstance';

$fields = "
    id I AUTO KEY,
    benchmarkId I CONSTRAINTS 'FOREIGN KEY REFERENCES benchmark',
    benchmarkSetId I CONSTRAINTS 'FOREIGN KEY REFERENCES benchmarkSet',
    formatId I CONSTRAINTS 'FOREIGN KEY REFERENCES format',
    description C(2000),
    fileName C(100)
";

createTable($tableName, $fields);
```

Nadeklaroval jsem název tabulky, seznam sloupců tabulky a poté jsem zavolal funkci createTable(), která tabulku vytvoří. Jak vidíme, deklarace sloupce se skládá z názvu, definice typu s případným zadáním rozměrů sloupce a definice doplňkových vlastností. Například sloupec id jsem nadefinoval jako primární klíč, jehož hodnota je automaticky zvyšována o jedničku (autoincrement). Tři následující sloupce mají definován cizí klíč pro zachování integrity dat. Použité datové typy jsou INTEGER a VARCHAR.

Funkce pro vytvoření tabulky:

```
function createTable($tableName, $fields, $tabOptArray=false)
{
    $db = connect();
    $dict = NewDataDictionary($db);
    $sqlArray = $dict->CreateTableSQL($tableName, $fields, $tabOptArray);
    $result = $dict->ExecuteSQLArray($sqlArray);
}
```

Pár poznámek k funkci - po připojení k databázi se ze slovníku vytvoří pole SQL příkazů. Toto pole příkazů se provede metodou ExecuteSQLArray(). Existují však i další možnosti využití: vygenerovat textový soubor s SQL příkazy k dalšímu zpracování nebo automaticky převést na kód využívající jinou knihovnu pro práci s databází. Tímto převodem se však ztrácí možnost nasazení na široké škále databázových systémů bez dalších úprav.

7.4.8.2 Inicializace tabulek

Pokud není vytvořen žádný uživatelský účet, je vytvořen první účet s administrátorskými právy. Poté jsou tabulky naplněny základními daty. Mezi tato základní data patří zejména podporované formáty, atributy těchto formátů a základní benchmarkové sady.

7.4.8.3 Přihlášení, odhlášení, změna hesla

Nepřihlášený uživatel se může přihlásit uživatelským jménem a heslem, přihlášený uživatel se může odhlásit. Pokud má uživatel oprávnění, může změnit vlastní heslo.

7.4.8.4 Administrace uživatelů

Uživatel s oprávněním k editaci uživatelských účtů může přidávat a rušit uživatelské účty, může měnit oprávnění jednotlivých uživatelů

7.4.8.5 Práce s formáty

Formáty lze zobrazovat a editovat jejich popis a podporované atributy.

7.4.8.6 Práce s benchmarkovými sadami

Je možné měnit popis sady, zobrazovat informace o sadě a vyhledávat instance.

7.4.8.7 Práce se sadami benchmarků

Uživatel může vytvářet a vyhledávat instance benchmarků. Všechny nalezené benchmarky pro daná kritéria je možno stáhnou ve vytvořeném souboru formátu ZIP nebo si stáhnout pouze zvolený benchmark.

7.5 Plnění databáze

Jednotlivé nebo dávkové plnění databáze skriptem je založeno na třech základních operacích, což jsou:

- Analýza souboru pomocí vypracovaného parseru.
- Vložení analyzovaných dat do databáze.
- Vložení souboru do adresářové struktury.

Použitý parser je stejný jako u řádkového analyzátoru, vložení analyzovaných dat do databáze se provádí pomocí knihovny AdoDB a vložení souboru do adresářové struktury je založeno na znalosti názvu sady benchmarků a na základě jména původního souboru.

7.6 Agregované stahování

Agregované stahování slouží ke stažení všech benchmarků splňujících vyhledávací kritéria. Všechny vyhovující benchmarky jsou zabaleny v archivu formátu ZIP a nabídnuty uživateli ke stažení.

Archiv je vytvořen externím skriptem, který dostane jako parametr dotaz v jazyce SQL, pomocí něhož provede selekci souborů.

Předání informace proběhne pomocí předání základu názvu souboru, otevře se soubor s příponou .query, vyhledají se vyhovující soubory a vloží se do archivu (.zip) a po dokončení archivu se vytvoří soubor s příponou .existence, který pro webovou aplikaci signalizuje, že archiv je kompletní a připravený ke stažení.

8 Poznámky k vývoji a udržovatelnosti

Strukturování kódu, jeho dokumentace, správa zdrojových kódů, to jsou aspekty, které výrazně ovlivňují udržovatelnost kódu projektu; proto se v této kapitolce o těchto tématech krátce zmíním.

8.1 Strukturování a pojmenování kódu

Z praxe vím, že pro programátora, který má prohlížet a upravovat cizí kód, ale i pro samotného autora, zvláště po delší době od vytvoření, je důležité, aby ze samotné struktury zdrojových kódů a názvů jednotlivých částí kódu získal co nejrychleji přehled o funkci jednotlivých částí a jejich vzájemných vztazích. Důležité je, aby kód působil jednotně, takže může programátor z modulů, tříd a funkcí, které již zná, usuzovat na chování modulů, tříd a funkcí, které jsou názvem či funkčností příbuzné.
Ve své práci jsem se snažil členit kód do samostatných modulů, nejlépe podle jednotlivých tříd. Více tříd v jednom modulu jsem použil pouze v případě, že třídy byly malé a logicky patřily k sobě. Názvy modulů, tříd a proměnných jsem se snažil udržet v jednotné podobě. Příkladem mohou být PHP stránky, které, až na pár důvodných výjimek, pojmenovávám jednotným stylem (*typdat_druhakce.php*), jednotné pojmenování třídy vytvářející stránku (PageClass) nebo v řádkovém analyzátoru pojmenování modulů BenchmarkFileParser_formát.py.

8.2 Dokumentace kódu

Zatímco mnohé je možné zjistit ze samotných názvů tříd, modulů, funkcí, proměnných apod., jsou-li dobře navrženy, je více než vhodné v komentáři popsat funkci modulu, třídy, funkce nebo menší části kódu. Tuto zásadu jsem se snažil dodržet ve všech zdrojových kódech aplikační části.

8.3 Správa zdrojových kódů

Existují různé strategie ukládání, zálohování a sdílení souborů. Soubory se dají ukládat na jednom místě na serveru, rozesílat mezi vývojáři apod. Různá naivní a ad hoc řešení se ukazují jako příliš slabá a existuje mnoho softwarových balíků, které umožňují efektivně sdílet kód mezi vývojáři projektu, spravovat několik různých větví, získávat přehled o změnách a podobně. Ze zkušeností z praxe mohu říci, že pro práci v malých týmech lze s úspěchem použít systém CVS, který vychází z RCS. Na projektu jsem pracoval sám, ale i pro samostatného vývojáře poskytuje software pro správu zdrojových kódů (anglicky SCM - Source Code Management) několik výhod, které se pokusím popsat zde na mnou zvoleném systému CVS.

Časté ukládání verzí do SCM systému (anglicky se tato operace nazývá commit) umožňují vývojářům sledovat historii změn. Ke každému uložení se přidává popis změn. Je tedy možné sledovat, kdo, kdy a jaké změny provedl a v ideálním případě i rozumný popis těchto změn.

Před posláním verze do systému se vývojář může podívat, co skutečně změnil a naposled se zamyslet, zda něco neopomenul, nedopustil se omylem nějakého překlepu apod.

Do každého zdrojového souboru je možné vložit makro, které při ukládání aktualizuje informaci o změně - vloží číslo verze, jméno autora změny a čas změny. V systému CVS se používá makro $Id:. Příklad vložené informace:

 # $Id: benchmarkSets_LISTING.php,v 1.4 2006/12/13 00:12:55 jarda Exp $

Přestože nemohu předat s programem celý repozitář CVS, považuji za vhodné, aby případný další správce kódu zařadil projekt do vlastního CVS nebo jiného SCM systému.

9 Testování aplikace

V této kapitole se chci zabývat testováním aplikační části diplomové práce, tedy databáze benchmarků, řádkového analyzátoru a pomocných programů.

Testování je důležitou součástí každého softwarového projektu. Softwarové systémy jsou obyčejně natolik komplikované, že není možné předpokládat, že funguje naprosto bezchybně ihned po napsání. Proto po vytvoření aplikace, ale obyčejně i v jejím průběhu, dochází k testování aplikace. V případě, že vývojový cyklus je složitější (program je časem rozšiřován, upravován apod.), testovací procedury se opakují.

Z hlediska znalosti rozdělujeme testování na dvě skupiny:

- Black box testing - tester nebere v potaz nebo dokonce nezná vnitřní fungování testovaného systému. Tomuto druhu testování se také říká testování zaměřené na vstup-výstup (input-output driven) nebo zaměřené na data (data-driven).
- White box testing - tester testuje systém se zřetelem k vnitřnímu fungování systému. Testovací data jsou volena s ohledem ke znalosti logické struktury programu.

Z hlediska rozsahu rozeznáváme čtyři druhy testů:

- Jednotkové testy (unit tests) testují funkčnost samostatné jednotky. V objektovém programování se často jedná o jednu třídu.
- Integrační testování - testuje se skupina jednotek spojená ve větší logický celek.
- Systémové testování - testuje se celý systém, zda vyhovuje specifikovaným požadavkům. Systémové testování by mělo probíhat stylem black box testingu.
- Akceptační testy - testování celého systému zákazníkem, zda systém vyhovuje specifikovaným požadavkům. Akceptační testování patří rovněž do kategorie black box testingu.

Zaměřím se na systémové testování a budu předpokládat, že akceptační testy proběhnou podle stejných scénářů. Na závěr se dotknu jednotkového white box testingu programů v jazyce Python a ukážu praktické nasazení na analyzátoru benchmarkových souborů.

9.1 Systémové a akceptační testy

9.1.1 Skupiny scénářů

Systémové a akceptační scénáře jsem rozdělil do několika skupin:

- Testování instalace a inicializace databáze - budeme předpokládat existenci prázdné databáze, která má obsahovat tabulky benchmarků. Nainstalujeme aplikaci, provedeme vytvoření tabulek a naplnění základními daty.
- Testování řádkového analyzátoru - otestujeme, zda je řádkový analyzátor schopen analyzovat soubory všech podporovaných formátů a zda jsou údaje ve výpisu v pořádku.
- Testování plnění databáze skriptem - otestujeme, zda je skript schopen dávkově naplnit data všech skriptů z daného adresáře do databáze.
- Testování webového rozhraní - otestujeme, zda webové rozhraní funguje podle specifikace ve čtyřech skupinách.

  - Prohlížení databáze
  - Editace databáze
  - Správa uživatelských účtů
  - Přístupová práva

- Testování přístupu - otestujeme, zda výhradně uživatelé s potřebnými právy mohou provádět zadané akce.

9.1.2 Testování instalace a inicializace

9.1.2.1 Instalace aplikace

Předpoklady: Aplikace není nainstalována

Akce: Nainstalujeme aplikaci podle instalační příručky

Úspěch: Aplikaci se povedlo nainstalovat

Neúspěch: Aplikaci se nepovedlo nainstalovat

9.1.2.2 Vygenerování konfiguračních souborů

Předpoklady: Žádné

Akce: Vyplníme metakonfigurační soubor podle instalační příručky a konfigurace hostitelského systému, vygenerujeme konfigurační soubory pro PHP a Python, zkopírujeme je do příslušných adresářů

Úspěch: Soubory se povedlo vygenerovat a zkopírovat

Neúspěch: Soubory se nepovedlo vygenerovat nebo zkopírovat

9.1.2.3 Vytvoření tabulek

Předpoklady: Aplikace je správně nainstalována a zkonfigurována, hostitelská databáze neobsahuje žádnou z tabulek databáze benchmarků

Akce: Vytvoříme tabulky podle instalační příručky

Úspěch: Tabulky se vytvořily podle zadaného souboru

Neúspěch: Tabulky se nevytvořily podle zadaného souboru

9.1.2.4 Vložení základních dat

Předpoklady: Aplikace je správně nainstalována a zkonfigurována, tabulky jsou vytvořeny a jsou prázdné.

Akce: Naplníme databázi základními daty podle instalační příručky

Úspěch: Tabulky obsahují očekávaná data

Neúspěch: Tabulky neobsahují očekávaná data

9.1.3 Testování řádkového analyzátoru

9.1.3.1 Testování správné analýzy všech formátů

Předpoklady: Aplikace je řádně nainstalována

Akce: Spustíme řádkový analyzátor, jeho parametry jsou soubory všech podporovaných formátů

Úspěch: Řádkový analyzátor správně zanalyzuje všechny soubory

Neúspěch: Výpočet skončí běhovou chybou nebo jsou vypsány nesprávné výsledky

9.1.3.2 Testování explicitního zadání formátu

Předpoklady: Aplikace je řádně nainstalována

Akce: Spustíme řádkový analyzátor, jeho parametry jsou jméno formátu a soubor přejmenovaný tak, aby z přípony nebylo možné formát určit.

Úspěch: Řádkový analyzátor správně zanalyzuje zadaný soubor

Neúspěch: Výpočet skončí běhovou chybou nebo jsou vypsány nesprávné výsledky

9.1.4 Testování plnění databáze dávkovým skriptem

9.1.4.1 Testování plnění známou sadou

Předpoklady: Aplikace je řádně nainstalována

Akce: Spustíme skript pro dávkové plnění, parametrem bude adresář obsahující známou sadou benchmarků (např. LGSynth91).

Úspěch: Všechny benchmarky sady jsou správně naplněny do databáze.

Neúspěch: Benchmarky nejsou správně naplněny do databáze.

9.1.5 Testování funkcí webového rozhraní - prohlížení dat

9.1.5.1 Vypsání seznamu benchmarkových sad a jejich vlastností

Předpoklady: Aplikace je řádně nainstalována a nakonfigurována, uživatel má dostatečná přístupová práva.

Akce: Vypíšeme si seznam benchmarkových sad, vypíšeme si obsah sady, vyhledáme instance benchmarků sady.

Úspěch: Vypíší se správná data

Neúspěch: Dojde k běhové chybě nebo jsou vypsána nesprávná data.

9.1.5.2 Vypsání seznamu formátů a jejich vlastností

Předpoklady: Aplikace je řádně nainstalována a nakonfigurována, uživatel má dostatečná přístupová práva.

Akce: Vypíšeme si seznam formátů, vypíšeme si informace o formátu

Úspěch: Vypíší se správná data

Neúspěch: Dojde k běhové chybě nebo jsou vypsána nesprávná data.

9.1.5.3 Vyhledání instancí benchmarků podle zadaných kritérií, zobrazení dat instance

Předpoklady: Aplikace je řádně nainstalována a nakonfigurována, uživatel má dostatečná přístupová práva.

Akce: Spustíme vyhledávání, zobrazíme data některé z vyhledaných instancí

Úspěch: Vyhledají se správné instance, vypíší se správná data

Neúspěch: Dojde k běhové chybě nebo jsou vyhledány nesprávné instance nebo vypsána nesprávná data.

9.1.5.4 Stažení jednotlivého souboru

Předpoklady: Provedli jsme vyhledávání, jehož výsledkem je benchmark s nám známým obsahem.

Akce: Zadáme stažení souboru.

Úspěch: Stáhne se soubor se správným obsahem.

Neúspěch: Dojde k běhové chybě nebo se stáhne nesprávný soubor.

9.1.5.5 Agregované stažení vyhledaných benchmarků

Předpoklady: Provedli jsme vyhledávání, jehož výsledkem je neprázdná množina vyhledaných benchmarků.

Akce: Zadáme agregované stažení souborů.

Úspěch: Stáhne se ZIP archiv obsahující vyhledané benchmarky

Neúspěch: Dojde k běhové chybě, archiv je poškozený, neobsahuje odpovídající soubory.

9.1.6 Testování funkcí webového rozhraní - vkládání a změna dat

9.1.6.1 Vložení instance benchmarku se zadáním hodnot atributů

Předpoklady: Máme nainstalovanou a nakonfigurovanou databázi benchmarků obsahující minimálně jeden formát a jednu benchmarkovou sadu, uživatel smí editovat databázi benchmarků

Akce: Vložíme novou instanci benchmarků a hodnoty atributů zadáme ručně.

Úspěch: Vloží se nová instance s vloženými hodnotami.

Neúspěch: Instance se nevloží nebo vložené hodnoty neodpovídají vstupu.

9.1.6.2 Vložení instance benchmarku s volbou automatického vyplnění hodnot atributů

Předpoklady: Máme nainstalovanou databázi a nakonfigurovanou benchmarků obsahující minimálně jeden formát a jednu benchmarkovou sadu, uživatel smí editovat databázi benchmarků

Akce: Vložíme novou instanci benchmarků, zadáme benchmarkový soubor a zaškrtneme automatické vyplnění atributů.

Úspěch: Vloží se nová instance se správně vyplněnými hodnotami hodnotami.

Neúspěch: Instance se nevloží nebo vložené hodnoty neodpovídají obsahu souboru a zadaným údajům.

9.1.6.3 Editace instance

Předpoklady: Máme nainstalovanou databázi a nakonfigurovanou benchmarků obsahující minimálně jednu vloženou instanci benchmarku.

Akce: Zeditujeme hodnoty instance.

Úspěch: Hodnoty se změní podle specifikace.

Neúspěch: Hodnoty se nezmění správným způsobem.

Poznámka: Podobný test provedeme i se sadou benchmarků a formátem.

9.1.7 Testování webového rozhraní - správa uživatelských účtů

9.1.7.1 Vypsání uživatelů

Předpoklady: Máme nainstalovanou a nakonfigurovanou databázi benchmarků a více než jednoho uživatele, přihlášený uživatel smí pracovat s uživatelskými účty.

Akce: Zobrazíme si výpis uživatelů.

Úspěch: Zobrazí se správný výpis uživatelů, je možné editovat nebo mazat cizí účty, nikoli vlastní.

Neúspěch: Výpis se nezobrazí korektně nebo není možné editovat cizí účet nebo je možné editovat vlastní účet.

9.1.7.2 Přidání uživatele

Předpoklady: Máme nainstalovanou a nakonfigurovanou databázi benchmarků, přihlášený uživatel smí editovat uživatelské účty.

Akce: Přidáme nový uživatelský účet

Úspěch: Uživatelský účet je správně vytvořen, lze se přihlásit s daným heslem a přístupová práva jsou správně nastavena.

Neúspěch: Účet se nevytvoří, nelze se přihlásit nebo přístupová práva nejsou v pořádku.

9.1.7.3 Smazání uživatelského účtu

Předpoklady: Máme nainstalovanou a nakonfigurovanou databázi benchmarků a minimálně jednoho dalšího uživatele, přihlášený uživatel smí editovat uživatelské účty.

Akce: Smažeme uživatelský účet.

Úspěch: Uživatel zmizí z databáze, nemůže se přihlásit a již přihlášený uživatel nemůže dále pracovat s vyššími právy, než má nepřihlášený uživatel.

Neúspěch: Uživatelský účet nadále existuje, uživatel může dále pracovat s vyššími právy.

Poznámka: Podobně postupujeme u testu editace uživatelského účtu.

9.1.8 Testování webového rozhraní - přístupová práva

9.1.8.1 Testování volného přístupu při povolení volného přístupu

Předpoklady: Máme nainstalovanou a nakonfigurovanou databázi benchmarků. Je povoleno nepřihlášeným uživatelům prohlížet databázi.

Akce: Vstoupíme do webového rozhraní jako nepřihlášený uživatel

Úspěch: Lze prohlížet databázi benchmarků

Neúspěch: Nelze prohlížet databázi benchmarků

Poznámka: Provedeme opačný test - nepřihlášený uživatel nesmí mít přístup, pokud je volný přístup zakázán.

9.1.8.2 Testování změny vlastního hesla

Předpoklady: Uživatel je přihlášen a smí měnit vlastní heslo.

Akce: Změníme vlastní heslo uživatele

Úspěch: Heslo se změnilo

Neúspěch: Heslo se nezměnilo

Poznámka: Provedeme opačný test - uživatel se zákazem změny hesla nesmí mít možnost měnit heslo

9.1.8.3 Testování přístupových práv pro prohlížení

Předpoklady: Uživatel je přihlášen a volný přístup do databáze je zakázán.

Akce: Uživatel zkusí základní akce - například si zobrazí seznam sad.

Úspěch: Prohlížení je možné

Neúspěch: Prohlížení není možné

9.1.8.4 Testování změny uživatelských účtů bez příslušného oprávnění

Předpoklady: Uživatel je přihlášen a nesmí editovat uživatelské účty.

Akce: Pokusíme se vypsat uživatelské účty.

Úspěch: Výpis není k dispozici.

Neúspěch: Výpis je k dispozici

Poznámka: Provedeme tento test i pro nepřihlášeného uživatele při volně dostupném prohlížení databáze. Oba testy provedeme analogicky i pro akci, která edituje databázi benchmarků - například editaci sady benchmarků.

9.2 Jednotkové testování

Jednotkové (modulární) testování je testování jednotlivých funkcí, metod, v objektovém programování nejčastěji tříd použitých v programu. Účelem je otestování, zda chování testované jednotky odpovídá zadané specifikaci.

9.2.1 Jednotkové testování v jazyce Python

V jazyce Python existuje několik frameworků používaných pro jednotkové testování. Mezi ně patří:

- Modul *unittest*, dříve zvaný PyUnit, je založen na principech frameworku JUnit používaného pro jazyk Java.
- Modul *testoob*, který rozšiřuje možnosti modulu *unittest* o podporu barevného výstupu, výběr testů pomocí regulárních výrazů, výstup do XML a HTML.
- Modul doctest, založený na příkladech použití uložených v dokumentačních řetězcích funkcí, metod, tříd a modulů.

9.2.1.1 Testování s pomocí modulu doctest

Modul doctest umožňuje vkládat testovací příklady do dokumentačních řetězců. Tyto příklady jsou v lidsky čitelné formě a simulují práci v interaktivním interpreteru Pythonu. Následující dokumentační řetězec dokumentuje správnou funkci standardních funkcí min() a max():

```
"""Example of correct output of function min():
>>>min(3, 8, 1, 5)
1

Example of correct output of function max():
>>>max(3, 8, 1, 5)
8
"""
```

K provedení testu ovšem musíme zajistit import modulu doctest a spustit v něm definovanou funkci testmod(). To zajistíme například přidáním následujícího kódu na konec testovaného modulu:

```
import doctest
doctest.testmod()
```

Pak už stačí jenom na příkazové řádce zadat příkaz (za předpokladu, že je testovaný modul v souboru mymodule.py:

```
python mymodule.py
```

9.2.2 Příklad nasazení v řádkovém analyzátoru benchmarků

Příkladem nasazení jednotkového testování v řádkovém analyzátorů je kontrola správnosti detekce typu hradla v PLA tabulce formátu BLIF. Třída BenchmarkFileParser_BLIF definuje metodu getGateType(), která na základě seznamu symbolů korespondujících s PLA tabulkou detekuje typ hradla. Tato detekce je poměrně složitá a náchylná k chybám. Vhodnou aplikací testování pomocí modulu doctest() je možno otestovat správnou funkčnost této detekční metody. Nejprve se vytvoří testovací kód v dokumentačním řetězci metody:

```
>>> parser = BenchmarkFileParser_BLIF()
>>> parser.getGateType([[(0, '1',), (0, '1',),],]) ==
... BenchmarkFileParser_BLIF.GT_BUF
True
>>> parser.getGateType([[(0, '1',), (0, '0',),],]) ==
... BenchmarkFileParser_BLIF.GT_NOT
True
>>> parser.getGateType([
... [(0, '00',), (0, '0',),],
... [(0, '01',), (0, '0',),],
... [(0, '10',), (0, '0',),],
... [(0, '11',), (0, '1',),],
... ]) == BenchmarkFileParser_BLIF.GT_AND
True
>>> parser.getGateType([
... [(0, '00',), (0, '0',),],
... [(0, '01',), (0, '1',),],
... [(0, '10',), (0, '1',),],
... [(0, '11',), (0, '1',),],
... ]) == BenchmarkFileParser_BLIF.GT_OR
True
>>> parser.getGateType([
... [(0, '00',), (0, '1',),],
... [(0, '01',), (0, '0',),],
... [(0, '10',), (0, '0',),],
... [(0, '11',), (0, '0',),],
... ]) == BenchmarkFileParser_BLIF.GT_NOR
True
>>> parser.getGateType([
... [(0, '00',), (0, '1',),],
... [(0, '01',), (0, '1',),],
... [(0, '10',), (0, '1',),],
... [(0, '11',), (0, '0',),],
... ]) == BenchmarkFileParser_BLIF.GT_NAND
True
```

Tento kód napřed vytvoří instanci třídy BenchmarkFileParser_BLIF(), na níž opakovaným voláním testujeme správnost výstupů funkce pro určité PLA tabulky.

Spuštění testů zajistí následující kód na konci modulu:

```
def _test():
    # DOC {{{
    """Runs doctest on the module.
    """
    # }}}

    # CODE {{{
    import doctest
    doctest.testmod()
    # }}}


# when the script is the main script, run the test {{{
if __name__ == "__main__":
    _test()
# }}}
```

Při úspěšném složení testu nevypíše příkaz *./python BenchmarkFileParser_BLIF.py* nic, pokud však modifikujeme např. detekční kód tak, že vynecháme detekci hradla NAND a metoda tedy vrací kód pro neznámý typ hradla, vypíše se následující chybová zpráva:

```
**********************************************************************
File "BenchmarkFileParser_BLIF.py", line 232,
in __main__.BenchmarkFileParser_BLIF.getGateType
Failed example:
    parser.getGateType([
    [(0, '00',), (0, '1',),],
    [(0, '01',), (0, '1',),],
    [(0, '10',), (0, '1',),],
    [(0, '11',), (0, '0',),],
    ]) == BenchmarkFileParser_BLIF.GT_NAND
Expected:
    True
Got:
    False
**********************************************************************
1 items had failures:
   1 of   7 in __main__.BenchmarkFileParser_BLIF.getGateType
***Test Failed*** 1 failures.
```

9.3 Kontrola korektnosti kódu

Korektnost nebo nekorektnost napsaného kódu se nejlépe prokáže při běhu aplikace. Ale jelikož není obvykle možné zkoumat korektnost funkce aplikace za všech možných podmínek, je vhodné zkontrolovat korektnost kódu před spuštěním. Programátor se může dopustit syntaktických

chyb, chyb sémantiky a logických chyb. Syntaktické a sémantické chyby jsou obyčejně odhaleny kompilátorem nebo vstupem modulu do interpreteru v případě interpretovaných jazyků. Logické chyby se odhalují hůře, ale v jednotlivých programovacích jazycích existují konstrukce, které lze s největší pravděpodobností považovat za chybné - ať už jde o překlep nebo chybu návrhu. Ke kontrole kódu v Pythonu jsem využil program *Pylint*, který je jakousi obdobou programu *lint* známého z unixového prostředí, kde se dlouhá léta používal pro kontrolu korektnosti zdrojových kódů v jazyce C.

9.4 Závěrečné poznámky

Funkčnost aplikace jsem testoval podle vytvořených scénářů, objevené problémy jsem opravil. Testoval jsem programy i z hlediska rychlosti a shledal jsem tuto rychlost dostatečnou a odpovídající. V případě webového rozhraní databáze šlo zejména o (z hlediska uživatele) v podstatě okamžitou odezvu, řádkový analyzátor zpracovává soubory úměrně jejich složitosti.

10 Závěr

Tato práce si kladla za cíl zjistit, jaké zkušební obvody pro logickou syntézu jsou k dispozici, jaké formáty souborů se v praxi používají a jaké mají vyjadřovací možnosti. Zjistil jsem, že od 80. let existuje snaha připravit pro akademickou veřejnost i komerční organizace ucelené sady zkušebních obvodů a s postupem času se tyto sady staly obsáhlejší a přispívaly do nich výzkumné i komerční instituce. Zatímco v počátcích se popisy obvodů často omezovaly na několikařádkový slovní popis funkce obvodu, později sílily snahy tyto popisy lépe formalizovat a přinášet je v podobě jak lidsky čitelné, tak i strojově zpracovatelné pro účely simulace nebo logické syntézy.

Úroveň popisů obvodů se může značně lišit. K dispozici jsou zdrojové kódy v jazycích typu VHDL a Verilog na úrovni behaviorálního popisu obvodu, kdy se obvod tváří jako černá skříňka s algoritmicky popsaným chováním, na druhé straně některé popisy jsou přímo na úrovni zapojení obvodu ve formě tzv. netlistu.

Ve své práci jsem se zaměřil zejména na pět často používaných formátů: PLA, BENCH, KISS2, BLIF a SLIF.

Formát PLA je dvouúrovňovým zápisem logické funkce a lze jej použít pro popis funkce kombinačního obvodu.

Formát BENCH je typickým reprezentantem formátů popisujících zapojení obvodu na úrovni elementárních logických prvků - hradel a klopných obvodů.

Formát KISS2 popisuje sekvenční obvod formou konečného automatu.

Formát SLIF popisuje kombinační nebo sekvenční obvod pomocí logických rovnic.

Formát BLIF je velmi univerzální, může obsahovat jak zapojení obvodů tvořených knihovními hradly a klopnými obvody, tak i popisy jednotlivých hradel pomocí PLA tabulky a popisy jednotlivých klopných obvodů ve formátu KISS.

Probral jsem několik sad zkušebních obvodů pro logickou syntézu z hlediska obvodů v nich obsažených a používaných formátů. U některých sad se jsou k dispozici verze označené rokem vzniku; na těchto sadách lze pozorovat, jak se rozšiřovaly tyto sady jak z hlediska počtu obsažených zkušebních obvodů, tak i počtu a kvality jejich popisů vzhledem k potřebám a zkušenostem přispívajících osob a organizací.

Dalším úkolem bylo vytvořit analyzátor (parser) některých souborových formátů zkušebních obvodů. Parser, který jsem naprogramoval, umožňuje analyzovat atributy obvodů ve formátu PLA, BENCH, KISS2, SLIF a BLIF a vypisuje zjištěné výsledky na příkazový řádek. Tento parser je možné poměrně snadno doplnit o podporu dalších formátů.

Znalost formátů souborů a atributů, které lze z těchto formátů vyčíst, umožnila vytvořit databázovou aplikaci, která uchovává informace o dostupných obvodech. Zároveň jsou ukládány i původní soubory samotné.

Databáze je postavena na SQL databázovém stroji PostgreSQL a přistupovat k ní lze pomocí webového rozhraní napsaného pomocí dynamického jazyka PHP. Rozhraní umožňuje vkládat

instance zkušebních obvodů, vyhledávat je podle různých kritérií jako je název, formát, sada, do níž přísluší a hodnoty atributů. Vyhledané instance zkušebních obvodů je možné editovat, zobrazovat jejich podrobné informace a stahovat jejich soubory jednotlivě nebo agregovaně pro všechny výsledky hledání najednou.

Součástí řešení webového rozhraní je i systém uživatelských práv. Přístup k prohlížení databáze lze omezit pouze na registrované uživatele nebo povolit veřejný přístup všem. Pouze uživatelé s dostatečným oprávněním mohou editovat databázi nebo spravovat uživatelské účty.

Doplňkové funkce, které jsem v rámci diplomové práce připravil, jsou dávkové plnění databáze skriptem a možnost parsování souboru s automatickým plněním hodnot atributů při vkládání nové instance přes webové rozhraní.

Ve své práci se mimo jiné zabývám i otázkami bezpečnosti aplikace a vhodným stylem vývoje a psaní kódu. Pojednal jsem krátce o použitých technologiích a jednu kapitolu jsem věnoval scénářům testování aplikace.

Myslím, že práce splnila kladené cíle. Možnosti dalšího vývoje vidím v přidání možnosti analýzy dalších formátů, další vývoj v oblasti dostupných zkušebních obvodů a nasazení databáze a pomocných programů v praxi zcela jistě přinese další podněty k rozšíření a vylepšení aplikace.

11 Seznam literatury

[1] Berkeley Logic Interchange Format (BLIF). http://embedded.eecs.berkeley.edu/research/vis/blif.ps.

[2] D. Bryan. Description of Iscas85 circuits, 1988.

[3] D. Cenderholm. *Webdesign s webovými standardy.* Zoner Press, 2004.

[4] J. Douša. *Jazyk VHDL.* České vysoké učení technické v Praze, 2003.

[5] F. Brglez et al. Combinational Profiles of Sequential Benchmark Circuits, 1991.

[6] F. Corno et al. RT-Level ITC 99 Benchmarks and First ATPG Results, 1999.

[7] G. J. Myers. *The Art od Software Testing.* John Willey & Sons, Inc., 2004.

[8] Input file format for espresso (pla-file). http://www.engineering.uiowa.edu/~switchin/espresso.5.html.

[9] C. Shiflett. The Truth about Sessions. http://shiflett.org/articles/the-truth-about-sessions.

[10] C. Shiflett. *Essential PHP Security.* O'Reilly, 2005.

[11] SLIF - Structure Logic Intermediate Format.

[12] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0, 1991.

[13] Apache HTTP Server Version 2.2 Documentation. http://httpd.apache.org/docs/2.2/.

[14] PHP Manual. http://www.php.net/docs.php.

[15] PostgreSQL Documentation. http://www.postgresql.org/docs.

[16] Python 2.5 Documentation. http://docs.python.org.

[17] ADOdb Library for PHP. http://phplens.com/lens/adodb/docs-adodb.htm.

A Používané formáty benchmarků

V této kapitole popíši nejčastěji používané formáty benchmarkových obvodů, které se vyskytují v sadách benchmarků. Logické obvody je možné popsat různými způsoby - třeba slovním popisem, pomocí logické funkce, sadami termů, konečným automatem nebo komplexním programovacím jazykem (HDL - hardware definition language).

Ve vyšších jazycích používaných pro logickou se používají tři druhy popisu obvodu:

- Behaviorální popis udává způsob chování obvodu navenek, ale neříká nic o vnitřní struktuře obvodu - obvod je pro pozorovatele černou skříňkou.
- Register Transfer Level (RTL) popis rozděluje obvod na jednotlivé sekvenční části - registry a na kombinační obvody, které spojují tyto bloky. RTL popis udává, jakým způsobem jsou data přenášena mezi těmito registry.
- Strukturální popis přesně ukazuje, jakým způsobem je daný obvod zapojen - jaké elementární prvky (např. hradla a klopné obvody) jsou v zapojení použity a jak jsou mezi sebou propojeny. Strukturálnímu popisu se také říká *netlist*.

Formáty PLA, BENCH, KISS2, SLIF a BLIF se tato práce zabývá podrobněji a součástí práce je i parser těchto formátů. Zmíním se i o formátech NETBLIF a ESPRESSO-MV, které jsou použity v jedné starší sadě. Formáty VHDL, Verilog a EDIF zmiňuji pro úplnost.

A.1 Formát PLA / Espresso

Tento formát je dvouúrovňový popis logické funkce. Nelze použít k popisu sekvenčních obvodů.

A.1.1 Používaná syntaxe a popis deklaračních řádků

Formát je řádkově orientovaný, každý významný řádek představuje řádek PLA matice nebo deklaraci uvozenou klíčovým slovem. Soubor může obsahovat komentáře uvozené znakem # na začátku řádku (objevil se ale i soubor s komentářem za samotným obsahem řádku) - vše za tímto znakem je při zpracování ignorováno. Přebytečné mezery a jiné bílé znaky mezi elementy jsou ignorovány, stejně jako řádky obsahující pouze bílé znaky nebo komentář.

Identifikátory (vstupů a výstupů) mohou obsahovat znaky alfanumerické znaky (a-z, A-Z, 0-9), dále znaky _~-[]<>+.

Následuje popis nejdůležitějších deklaračních řádků skládajících se klíčových slov a jejich parametrů:

- .i *číslo* - počet vstupů
- .o *číslo* - počet výstupů
- .ilb *identifikátor identifikátor...* - jména vstupů
- .ob - jména výstupů
- .p *číslo* - počet součinových termů
- .e - konec dat

V sadě benchmarků LGSynth89 se u obvodu alu4.pla vyskytují následující klíčová slova:

- .I - pojmenování jednotlivého vstupu
- .O - pojmenování jednotlivého výstupu
- .N - název popisovaného obvodu

Domnívám se však, že autor souboru nesprávně pochopil specifikaci formátu a že se tedy jedná o anomálii.

Další možné deklarace obsahuje manuálová stránka popisující formát Espresso. Tyto deklarace slouží zejména pro použití v programech pro logickou syntézu.

A.1.2 Popis logické funkce v PLA matici

Logická funkce je vyjádřená pomocí řádků, které určují, jaký výsledkem kombinace hodnot na vstupech bude hodnota na jednotlivých výstupech. Logická funkce pro každý výstup je vyjádřena třemi sadami termů:

- ON-set je sada termů, pro které má výstup hodnotu 1.
- OFF-set je sada termů, pro které má výstup hodnotu 0.
- DC-set je sada termů, u kterých na výstupní hodnotě nezáleží (don't care).

PLA matice je posloupností řádků, které přiřazují jednotlivé termy do některé sady (ON, OFF, DC) pro každý výstup. Sloupce matice jsou rozděleny do dvou částí - v levé části se nacházejí vstupní proměnné logické funkce a v pravé části se nacházejí výstupní hodnoty. Obě části jsou obyčejně odděleny mezerou, narazil jsem ale i na jejich oddělení svislou čarou (|).

PLA matice nemusí obsahovat definici všech sad. Používají se následující druhy PLA matic:

- Typ f definuje pouze ON sadu. OFF sada je doplňkem ON sady a DC sada je prázdná.
- Typ fd definuje ON sadu a DC sadu. OFF sada je doplňkem sjednocení obou definovaných sad.
- Typ fr definuje ON a OFF sadu. DC sada je doplňkem sjednocení obou definovaných sad.
- Typ fdr definuje ON, OFF i DC sadu.

Hodnoty vstupní části matice:

- '1' - hodnota vstupní proměnné v termu je přímá
- '0' - hodnota vstupní proměnné v termu je invertovaná
- '-' - vstupní proměnná se v termu nevyskytuje

Ve výstupní části matice se vyskytují symboly '1', '0', '-', '~'. Zatímco u prvních tří se (jak ukážu níže) význam liší pro jednotlivé typy PLA matic, význam symbolu '~' je vždycky stejný - daný term nemá pro hodnotu konkrétní výstupní funkce, na rozdíl od jiných výstupních funkcí, žádný význam. Chceme-li zkoumat chování výstupu obvodu, jehož hodnota koresponduje s danou výstupní funkcí, je třeba se obrátit k jiným termům.

Pro ilustraci je dobré si představit Karnaughovy mapy pro všechny výstupní funkce a uvědomit si, že pokud nejsou všechny výstupní funkce stejné, budou výsledkem pokaždé jiné sady termů - ne každý term má význam pro každou výstupní funkci.

Představme si PLA matici typu f pro 2 vstupní proměnné a 2 výstupní funkce AND a OR. Víme a pomocí pravdivostní tabulky a Karnaughovy mapy si ověříme, že tyto funkce lze zapsat jako:

```
y = ab
z = a + b
```

V těchto dvou funkcích jsou 3 termy: a, b, ab (v ON sadě). Odpovídající PLA matice tedy bude:

```
-1 ~1
1- ~1
11 11
```

Z matice vyplývá, že pokud víme, že na vstupu a nebo b je logická 1, je na prvním výstupu rovněž logická 1, ale bez znalosti hodnoty druhého vstupu nemůžeme určit hodnotu druhého výstupu.

Hodnoty výstupní části matice pro typ f:

- '1' - term patří do ON sady
- '0' - term nemá pro hodnotu dané funkce význam
- '-' - term nemá pro hodnotu dané funkce význam

Hodnoty výstupní části matice pro typ fd:

- '1' - term patří do ON sady
- '0' - term nemá pro hodnotu dané funkce význam
- '-' - term patří do DC sady

Hodnoty výstupní části matice pro typ fr:

- '1' - term patří do ON sady
- '0' - term patří do OFF sady
- '-' - term nemá pro hodnotu dané funkce význam

Hodnoty výstupní části matice pro typ fdr:

- '1' - term patří do ON sady
- '0' - term patří do OFF sady
- '-' - term patří do DC sady

Pro výstupní hodnoty existují následující povolená synonyma:

- '2' místo '-'
- '3' místo '~'
- '4' místo '1'

A.1.3 Jednoduché příklady

Následující příklad ukazuje popis funkce se sedmi vstupními proměnnými pojmenovanými (f, b, c, d, a, h, g) a dvěma výstupními proměnnými (f0 a f1). Funkce je vyjádřená 9 řádky PLA matice typu fr.

```
.i 7
.o 2
.ilb f b c d a h g
.ob f0 f1
.p 9
-1--1-- 10
1-11--- 10
-001--- 10
01---1- 10
-0--0-- 01
1---0-- 01
0-----0 01
01--1-- 01
10-0--- 01
.e
```

Další příklad ukazuje jiné vyjádření téže funkce PLA maticí s 18 řádky před provedením minimalizace.

```
.i 7
.o 2
.p 18
.ilb f b c d a h g
.ob f0 f1
.type fr
-001--- 1~
1-11--- 1~
01---1- 1~
10-0--- ~1
01--1-- ~1
-1--1-- 1~
0-----0 ~1
-0--0-- ~1
1---0-- ~1
001---- 0~
-0-0--- 0~
01--00- 0~
110-0-- 0~
1--00-- 0~
00--1-1 ~0
01--0-1 ~0
11--1-- ~0
1--11-- ~0
.e
```

Další informace lze nalézt v [8] nebo v [12].

A.2 Formát BENCH

Jedná se o víceúrovňový popis kombinačního nebo sekvenčního obvodu - jde o přímé propojení hradel a klopných obvodů (tzv. netlist).

A.2.1 Syntaxe

INPUT(**identifikátor**) - definuje vstupní proměnnou

OUTPUT(**název_proměnné**) - definuje vstupní proměnnou

identifikátor = *logický prvek*(identifikátor[, identifikátor ...]) - kombinační prvek (hradlo) nebo klopný obvod s jedním nebo několika vstupy a jedním výstupem

Identifikátor může obsahovat jakékoliv znaky kromě znaků ()=#, klíčové slovo (INPUT, OUTPUT nebo podporovaný název logického prvku) nesmí být použito jako identifikátor proměnné.

A.2.2 Podporované logické prvky

- BUF nebo BUFF - opakovač
- NOT - invertor
- AND - logický součin
- OR - logický součet
- XOR - obvod exclusive OR
- XNOR - negace exkluzivního OR
- NAND - negace funkce AND
- NOR - negace funkce OR
- DFF - klopný obvod typu D

A.2.3 Příklad

Následující příklad je převzatý z benchmarku s27.bench:

```
# 4 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)

INPUT(G0)
INPUT(G1)
INPUT(G2)
INPUT(G3)

OUTPUT(G17)
```

```
G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)

G14 = NOT(G0)
G17 = NOT(G11)

G8 = AND(G14, G6)

G15 = OR(G12, G8)
G16 = OR(G3, G8)

G9 = NAND(G16, G15)

G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
```

Formát je popsán v [5].

A.3 Formát KISS2 (Berkeley KISS2 format)

Formát KISS2 obsahuje definici sekvenčního obvodu ve formě konečného automatu. Skládá se z hlavičky a seznamu přechodů. Formát je řádkově orientovaný - každý řádek hlavičky obsahuje maximálně jeden údaj a každý řádek seznamu přechodů obsahuje jeden přechod.

A.3.1 Popis syntaxe a údajů v hlavičce

Hlavička může obsahovat následující informace:

- .i *číslo* - počet vstupů
- .o *číslo* - počet výstupů
- .p *číslo* - počet přechodů
- .s *číslo* - počet stavů
- .r *identifikátor* - stav po resetu

Pouze počet vstupů a výstupů jsou povinné údaje.

A.3.2 Seznam přechodů

Po hlavičce přichází řádky seznamu přechodů - každý přechod automatu je na samostatném řádku.

Jedna řádka seznamu přechodů vypadá následovně:

vstupní symbol výchozí stav cílový stav výstupní symbol

Stavy jsou označeny identifikátorem, pokud je na místě výchozího stavu hvězdička (*), znamená to, že pro daný vstupní symbol přechází automat do zadaného stavu a produkuje zadaný

výstupní symbol bez ohledu na aktuální stav. Hvězdička na místě cílového stavu znamená, že taková kombinace výchozího stavu a vstupního symbolu by neměla vůbec nastat.

Vstupy a výstupy nabývají hodnot 0, 1, - (don't care). Řetězec těchto hodnot představuje vstupní, resp. výstupní symbol.

A.3.3 Jednoduchý příklad

Následující příklad je převzatý z benchmarku s27.kiss2:

```
.i 4
.o 1
.p 34
.s 6
.r 000
010- 000 001 1
011- 000 000 1
110- 000 101 1
111- 000 100 1
10-0 000 100 1
00-0 000 000 1
-0-1 000 010 0
0-0- 001 001 1
0-1- 001 000 1
1-0- 001 101 1
1-1- 001 100 1
0-0- 101 001 1
0-1- 101 000 1
1-0- 101 101 1
1-1- 101 100 1
010- 100 001 1
011- 100 000 1
00-- 100 000 1
111- 100 100 1
110- 100 101 1
10-- 100 100 1
0-1- 010 010 0
000- 010 010 0
010- 010 011 0
1101 010 101 1
1111 010 100 1
10-1 010 010 0
1100 010 101 1
1110 010 100 1
10-0 010 100 1
0-0- 011 011 0
0-1- 011 010 0
1-1- 011 100 1
1-0- 011 101 1
```

Další informace lze nalézt v [12].

A.4 Konečný automat ve formátu ESPRESSO-MV

V sadě LGSYNTH91 se objevuje popis konečného automatu ve formátu ESPRESSO-MV.

A.4.1 Syntaxe souboru

Soubor je tvořen hlavičkou, po které následuje tabulka konečného automatu ve formátu KISS. Soubor je zakončen direktivou *.end*.

A.4.2 Jednoduchý příklad

Následující příklad je převzatý z benchmarku bbtas.esp:

```
.type fr
.mv 5 2 -6 -6 2
.kiss
.p 24
00 st0 st0 00
01 st0 st1 00
10 st0 st1 00
11 st0 st1 00
00 st1 st0 00
01 st1 st2 00
10 st1 st2 00
11 st1 st2 00
00 st2 st1 00
01 st2 st3 00
10 st2 st3 00
11 st2 st3 00
00 st3 st4 00
01 st3 st3 01
10 st3 st3 10
11 st3 st3 11
00 st4 st5 00
01 st4 st4 00
10 st4 st4 00
11 st4 st4 00
00 st5 st0 00
01 st5 st5 00
10 st5 st5 00
11 st5 st5 00
.end
```

A.5 Formát SLIF

SLIF je formát s poměrně volnými pravidly, příkazy mohou začínat kdekoliv na řádku a mohou obsahovat libovolný počet mezer, tabelátorů a mohou být rozděleny do několika řádků. Příkaz formátu SLIF je ukončen středníkem.

Identifikátory mohou obsahovat písmena, číslice a znaky :^%[]_./-

Proměnné, názvy modelů i instancí jsou identifikátory. "0" i "1" jsou proměnné se speciálním významem a reprezentují logické hodnoty True a False.

Logické výrazy používají operátory + (OR), * (AND, může být vynechán), ' (negace) a části výrazů mouhou být uzavřeny do kulatých závorek. Příklad deklarace obsahující logický výraz:

out = (aa + (bb cc)')';

Formát SLIF obsahuje dvě předdefinované funkce. **D**(a, c) a **T**(a, b). Funkce D je klopný obvod typu D se vstupním signálem a a hodinovým signálem c. **T** je třístavový budič, jehož výstupem je a jestliže b je má hodnotu True, jinak je výstup ve stavu vysoké impedance.

A.5.1 Příkazy

- .attribute *název_typu název_proměnné parametry;* Specifikuje parametry jedné proměnné.
- .call *název_instance název_modelu (vstupy; vstup_výstupy; výstupy;)* Vytvoří instanci název_instancemodelu název_modelu popsaného ve stejném souboru nebo v souboru specifikovaném elementem .search.
- .date *časové_razítko*; - specifikuje čas poslední modifikace (nepovinné).
- .endmodel *name;* - ukončení definice modelu.
- .global_attribute *název_typu parametry;* - parametry atributu platného pro celý model.
- .include *název_souboru*; - vložení obsahu jiného souboru.
- .inouts *proměnná1 proměnná2 ... proměnnán*; - deklaruje vstupně-výstupní proměnné.
- .inputs *proměnná1 proměnná2 ... proměnnán*; - deklaruje vstupní proměnné.
- .library; - identifikuje model jako knihovní modul.
- .model *název*; - indikuje začátek dalšího modelu a přiřadí mu jméno.
- .net *proměnná1 proměnná2 ... proměnnán*; - seznam vzájemně propojených proměnných.
- .outputs *proměnná1 proměnná2 ... proměnnán*; - seznam výstupních proměnných.
- .search *název_souboru*; - indikuje, že modely je možné hledat v zadaném externím souboru, je-li třeba.
- .type *název_typu spec1 spec2 ... specn*; - deklaruje typ jako sekvenci specifikací.

A.5.2 Příklad

Následující příklad je převzatý z benchmarku s27.slif:

```
.model s27 ;
# 4 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)
.inputs G0 G1 G2 G3  CK ;
```

```
.outputs G17  ;
G5 = @D ( G10, CK ) ;
G6 = @D ( G11, CK ) ;
G7 = @D ( G13, CK ) ;
G14 = G0' ;
G17 = G11' ;
G8 = G14 G6 ;
G15 = G12 +G8 ;
G16 = G3 +G8 ;
G9 = ( G16 G15 )' ;
G10 = ( G14 +G11 )' ;
G11 = ( G5 +G9 )' ;
G12 = ( G1 +G7 )' ;
G13 = ( G2 +G12 )' ;
.endmodel s27 ;
```

Další informace lze vyčíst z [11] nebo z [12].

A.6 Formát BLIF

Formát BLIF se používá k víceúrovňovému popisu logického obvodu v textové podobě. Popisovaný obvod může obsahovat odkazy na externí definice hradel a KO (.mlatch, .gate)
Může obsahovat popis konečného automatu vložením .kiss bloku
Podpora hierarchické struktury pomocí konstrukce .subckt - vložení jednoho modulu do druhého

A.6.1 Základní používané elementy

- .model *identifikátor* - název modelu
- .inputs *identifikátor identifikátor...* - seznam vstupů
- .outputs *identifikátor identifikátor...* - seznam výstupů
- .clock *identifikátor identifikátor...* - seznam hodinových signálů
- .latch - použitý KO
- .names - tabulka (formát PLA), za .names následují názvy vstupů a výstupu, pak řádky tabulky
- .end - konec modelu

A.6.2 Dvouúrovňový popis hradla pomocí PLA tabulky

Hradla se ve formátu BLIF definují pomocí PLA tabulky s jedním výstupem - před tabulkou je předřazen řádek se jmény vstupů a výstupu. Příklad:

```
.names a b n
11 1
00 1
```

A.6.3 Externí definice hradel a klopných obvodů

Formát BLIF umožňuje využít externě definovaná hradla z různých knihoven apod. Syntaxe je následující:

- .gate *název formální_název=skutečný_název...*
- .mlatch *název formální_název=skutečný_název... řízení* [*počáteční_stav*]

  Název hradla nebo klopného obvodu odpovídá názvu v knihovně, formální název je název parametru v knihovně, skutečný název odpovídá proměnné v našem souboru. Řízení může být buďto některý hodinový signál (definovaný pomocí .clock), některý výstup nebo NIL.

A.6.4 Popis sekvenčního obvodu konečným automatem

Formát BLIF může obsahovat popis sekvenčního obvodu pomocí konečného automatu ve formátu KISS. Definice je uzavřena mezi direktivami *.start_kiss* a *.end_kiss*:

```
.model 101  # outputs 1 whenever last 3 inputs were 1, 0, 1
.start_kiss
.i 1
.o 1
0 st0 st0 0
1 st0 st1 0
0 st1 st2 0
1 st1 st1 0
0 st2 st0 0
1 st2 st3 1
0 st3 st2 0
1 st3 st1 0
.end_kiss
.end
```

A.6.5 Další možnosti

A.6.6 Příklad typického souboru ze sady benchmarků

Následující příklad pochází z benchmarku s27.blif:

```
.model s27.bench
.inputs G0 G1 G2 G3
.outputs G17
.wire_load_slope 0.00
.latch     G10 G5  0
.latch     G11 G6  0
.latch     G13 G7  0
.names G11 G17
0 1
.names G14 G11 G10
00 1
.names G5 G9 G11
00 1
.names G2 G12 G13
00 1
```

```
.names G0 G14
0 1
.names G14 G6 G8
11 1
.names G1 G7 G12
00 1
.names G12 G8 G15
1- 1
-1 1
.names G3 G8 G16
1- 1
-1 1
.names G16 G15 G9
0- 1
-0 1
.end
```

Další informace lze najít v [1] nebo [12].

A.7 Formát NETBLIF

Formát NETBLIF je strukturální popisem obvodu. Skládá se z direktiv uvozených klíčovým slovem začínajícím tečkou. Zajímavé jsou zejména následující direktivy:

- .inputs *seznam vstupů*
- .outputs *seznam vstupů*
- .gate *název hradla seznam parametrů*

  Seznam parametrů je ve tvaru *formální parametr=skutečný parametr*. Formálními parametry jsou vývody hradla, skutečnými parametry jsou signály.

A.7.1 Příklad

Následující příklad pochází z benchmarku b1.netblif:

```
.model b1
.inputs a b c
.outputs d e f g
.gate nor1 a=c y=g
.gate nor1 a=c O=inv__c
.gate nor1 a=inv__c O=d
.gate nor1 a=a y=[72]
.gate nor1 a=b y=[73]
.gate nor2 a=a b=b y=[46]
.gate nor2 a=[72] b=[73] y=[47]
.gate nor2 a=[46] b=[47] y=e
.gate nor2 a=a b=b y=[40]
.gate nor2 a=g b=[40] y=[42]
.gate nor2 a=[72] b=[73] y=[41]
.gate nor2 a=c b=[41] y=[43]
.gate nor2 a=b b=[72] y=[44]
```

```
.gate nor2 a=a b=[73] y=[45]
.gate nor4 a=[42] b=[43] c=[44] d=[45] y=f
.end
```

A.8 Formát VHDL

Zkratka VHDL znamená VHSIC (Very-High-Speed Integrated Circuit) Hardware Definition Language. Jak zkratka napovídá, tento jazyk byl navržen pro návrh složitých integrovaných obvodů a používá se pro návrh obvodů realizovaných hradlovými poli, aplikačně specifických integrovaných obvodů a pro automatický návrh digitálních obvodů. Jazyk vychází z programovacího jazyka Ada a umožňuje behaviorální a algoritmický návrh logického obvodu na různých úrovních abstrakce. Parsování souborů ve formátu VHDL je netriviální problém a překračuje rozsah této práce.

A.8.1 Příklad

Následuje příklad benchmarku (gcd.vhdl):

```
use work.vvectors.all;

entity gcd is
 port (xi, yi : in bit16;
       rst    : in bit;
       ou     : out bit16);
end gcd;

architecture behavior of gcd is
begin

  process
   variable x,y: integer;

  begin
    wait until (rst = '0');

    x := xi;
    y := yi;

    while (x /= y) loop
      if (x < y)
        then y:=y-x;
        else x:=x-y;
      end if;
    end loop;

    ou <= x;
  end process;

end behavior;
```

Bližší informace o jazyce VHDL obsahuje [4].

A.9 Formát Verilog

Verilog je dalším komplexním jazykem pro popis hardware a lze jej použít pro návrh, verifikaci a implementaci analogových, digitálních i smíšených obvodů. Návrháři jazyka se snažili víceméně syntakticky přiblížit jazyku C, namísto složených závorek ale např. používají dvojici klíčových slov Begin a End. Tímto formátem se rovněž nebudu podrobněji zabývat.

A.10 Formát EDIF

Formát EDIF používá závorkovou syntaxi známou z jazyků typu LISP, resp. PostScript, tato syntaxe je velmi vhodná pro strojové zpracování z důvodu jednoduchosti a přímočarosti. V této práci jsem se však automatickou analýzou formátu EDIF nezabýval. Z mně dostupných sad obsahuje benchmarky ve formátu EDIF pouze LGSynth93 a tato sada obsahuje přímo program edf2fmt, který dokáže převést soubory ve formátu EDIF do formátu BLIF nebo SLIF, které lze snadněji číst a zpracovávat dostupnými parsery.

A.10.1 Jednoduchý příklad

Následující příklad je převzatý z benchmarku modulo12.edif:

```
(edif netlist
  (edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap (keywordLevel 0))
  (status
    (written
      (timeStamp 1993 2 0 16 45 4)
      (author "David Rickel")
      (program "autologic")
    ))
  (external PRIMLIB
    (edifLevel 0)
    (technology
      (numberDefinition
      )
      (simulationInfo
        (logicValue H (booleanMap (true)))
        (logicValue L (booleanMap (false)))
      ))
    (cell FALSE
      (cellType GENERIC)
      (view INTERFACE
        (viewType NETLIST)
        (interface
          (port out (direction output))
        )))
    (cell DC
      (cellType GENERIC)
      (view INTERFACE
        (viewType NETLIST)
        (interface
```

```
        (port out (direction output))
      )))
  (cell MUX2
    (cellType GENERIC)
    (view INTERFACE
      (viewType NETLIST)
      (interface
        (port out (direction output))
        (port in0 (direction input))
        (port in1 (direction input))
        (port sel (direction input))
      ))))
(library USER_LIB
  (edifLevel 0)
  (technology
    (numberDefinition
    )
    (simulationInfo
      (logicValue H (booleanMap (true)))
      (logicValue L (booleanMap (false)))
    ))
  (cell TOP
    (cellType GENERIC)
    (view NETLIST
      (viewType NETLIST)
      (interface
        (port modulo12_out (direction output))
        (port modulo12_in (direction input))
        (port clock (direction input)))
      (contents
        (instance G_G0
          (viewRef INTERFACE (cellRef MUX2 (libraryRef PRIMLIB))))
        (instance G_G1
          (viewRef INTERFACE (cellRef DC (libraryRef PRIMLIB))))
        (instance G_Gd17
          (viewRef INTERFACE (cellRef FALSE (libraryRef PRIMLIB))))
        (instance G_G117
          (viewRef INTERFACE (cellRef FALSE (libraryRef PRIMLIB))))
        (net N_N0
          (joined
            (portRef in1 (instanceRef G_G0))
            (portRef out (instanceRef G_G1))
          ))
        (net N__clk
          (joined
            (portRef clock)
          ))
        (net N_N2
          (joined
            (portRef out (instanceRef G_G0))
```

```
            (portRef modulo12_out)
          ))
         (net N_N3
          (joined
            (portRef sel (instanceRef G_G0))
            (portRef out (instanceRef G_Gd17))
          ))
         (net N_N4
          (joined
            (portRef in0 (instanceRef G_G0))
            (portRef out (instanceRef G_G117))
          ))
         (net N_N5
          (joined
            (portRef modulo12_in)
          )))))))
```

Jak vidíme, číst tento benchmark je pro člověka poměrně nesnadné, přestože se jedná snad o nejjednodušší benchmark ve formátu EDIF ze sady LGSynth93. Po převedení do formátu BLIF je to daleko snadnější:

```
#
# Written by e2fmt Fri Dec 29 16:33:23 2006
##############################################
.model top
.inputs modulo12_in clock
.outputs modulo12_out
# connect ports to nets with different names
.names n_n2 modulo12_out
1  1
.names modulo12_in n_n5
1  1
.names clock n__clk
1  1
# instance g_g117
.names  n_n4
  0
# instance g_g1
.names  n_n0
  -
# instance g_g0
.names n_n4 n_n0 n_n3  n_n2
1-0  1
-11  1
# instance g_gd17
.names  n_n3
  0
.end
```

B Sady benchmarků

K dispozici jsem měl několik zip archivů obsahujících benchmarkové sady. Popíšu nejdůležitější z nich, ostatní jsou buďto jejich variací nebo obsahují jenom velmi málo benchmarků.

B.1 HLSynth89

Tato sada benchmarků byla připravena pro 4. mezinárodní workshop o vysokoúrovňové syntéze, která se konala v říjnu 1989 v Kennebunkportu v USA. Obsahuje benchmarky ve formátech VHDL, Verilog a Hardware C, některé benchmarky jsou popsány mikroprogramem (ISP popis) nebo slovním popisem.

Obsahuje tyto benchmarky (viz dokumentaci přiloženou v sadě):

- Procesor mc6502 (ISP popis)
- Procesor mc68000 (ISP popis)
- Obvod i8251 (ISP popis, Hardware C)
- Amiga BLIT chip (Hardware C)
- Obvod pro FFT (slovní popis)
- Obvod pro extrakci výšky tónu (slovní popis)
- Kalmanův filtr (slovní popis)
- Obvod pro řízení semaforů (Verilog a VHDL)
- Obvod pro nalezení největšího společného dělitele (Verilog a VHDL)
- Čítač (Verilog a VHDL)
- Prefetch obvod (Verilog a VHDL)

B.2 HLSynth91

Sada obsahuje obvody ze sady HLSynth89 doplněné o obvody dodané G. De Michelim ze Stanfordské univerzity. Přidané obvody jsou ve formě unixových shell skriptů, které vytvoří benchmarky v jazyce Hardware C. Přidané obvody jsou (viz dokumentaci přiloženou v sadě):

- DAIO (Digital Audio Input-Output) fázový dekodér
- DAIO přijímač
- Obvodu pro výpočty s pevnou řádovou čárkou
- DAIO fázový dekodér
- Obvod pro korekci chyb na sériové lince
- Eliptický filtr 5. řádu
- Jednoduchý mikroprocesor typu RISC

- Výpočet největšího společného dělitele
- Řízení semaforů (Mead/Conway)
- Příklad prof. Alice Parkerové
- Příklad od firmy Tseng Labs

B.3 HLSynth92

Tato sada rozšiřuje popisy obvodů, obsahuje rozsáhlé slovní popisy, signálové grafy, testovací vektory. Snaží se o větší systematičnost a přidává některé další obvody, které jsou popsány pomocí VHDL, slovních popisů apod.:

- Obvod pro řešení diferenciálních rovnic
- Obvod pro čtení čárových kódů
- Řadič sedmisegmentových zobrazovacích jednotek pro zobrazování minut a sekund
- Kalkulátor adresy s falešnými cestami (FALSEPATH)
- FANCY čip vytvořený pro ilustraci procesu RTL optimalizace
- Kompletnější popis obvodu pro výpočet největšího společného dělitele
- Obvod pro řízení dopravy na dálniční křižovatce
- QRS čip pro zpracování ECG dat v medicínských přístrojích
- Armstrongův čítač
- Sekvencer mikroprogramové adresy AMD2910
- Čtyřbitový mikroprocesorový řez AMD2901
- Procesor 8251

B.4 ISCAS85

Sada ISCAS85 je tvořena kombinačními obvody, původně ve zvláštním netlistovém formátu ISCAS85, které jsou k dispozici ve formátech BENCH a BLIF.

Seznam obvodů dle [2]:

- C432 - dekodér priority
- C499, C1355 - obvody pro korekci jedné chyby
- C1908 - obvod pro korekci jedné chyby a detekci dvou chyb
- C880, C2670, C3540, C5315, C7552 - aritmeticko logická jednotka a řadič
- C6288 - 16 bitová násobička

B.5 ISCAS89

Sada ISCAS89 obsahuje sekvenční obvody ve formátech BENCH a BLIF. Seznam základních obvodů dle [5]:

- s349 - 4bitová násobička
- s298, s400, s444, s526 - obvody pro řízení semaforů
- s9234, s13207, s15850, s38417, s38584 - modifikované obvody
- s386, s510, s953, s1494 - řadiče syntetizované z vysokoúrovňových popisů
- s1238 - kombinační obvod s náhodně vloženými klopnými obvody
- s208, s420, s838 - digitální obvody pro násobení zlomků, hierarchicky syntetizované z vysokoúrovňového popisu
- s298, s208, s713, s641 a s832 jsou postavené na programovatelných zařízeních

B.6 LGSynth89

Obsahuje příklady z Mezinárodního workshopu pro logickou syntézu z roku 1989. Benchmarky z této sady patří do 3 základní kategorií:

1. Benchmarky v dvouúrovňové logice ve formátu PLA (ESPRESSO).
2. Tabulky konečných automatů ve formátu KISS nebo ESPRESSO-MV.
3. Benchmarky ve víceúrovňové logice ve formátu BLIF nebo Netlist-BLIF.

B.7 LGSynth91

Obsahuje příklady benchmarků použité v souvislosti s Mezinárodním workshopem pro logickou syntézu v roce 1991. Základní čtyři kategorie jsou:

1. Tabulky konečných automatů ve formátu KISS2.
2. Sekvenční víceúrovňová logika ve formátu BLIF nebo SLIF.
3. Kombinační víceúrovňová logika ve formátu BLIF nebo SLIF.
4. Dvouúrovňová logika ve formátu PLA (ESPRESSO) nebo SLIF.

B.8 ITC99

Sada ITC99 obsahuje benchmarky pocházející z různých zdrojů. Protože testovací obvody pocházející od komerčních subjektů, nejsou volně k dispozici a významnou část sady ITC99 představují obvody pocházející z Polytechnické univerzity v Turíně, popíšu obvody pocházející z této univerzity. Popisy těchto testovacích obvodů jsou k dispozici ve formátech BENCH, BLIF, EDIF a VHDL. Většina těchto obvodů je k dispozici v několika různých implementacích.

Seznam obvodů z Polytechnické univerzity v Turíně (dle [6]):

- b01 - konečný automat, který porovnává sériové toky dat

- b02 - konečný automat, který rozpoznává BCD čísla
- b03 - obsluha požadavků
- b04 - obvod pro výpočet minima a maxima
- b05 - obvod pro vyhodnocení obsahu paměti
- b06 - řadič přerušení
- b07 - počítá body na úsečce
- b08 - hledá společné podmnožiny v sekvencích čísel
- b09 - konvertor z jednoho sériového kódu na jiný
- b10 - volební systém
- b11 - obvod pro šifrování řetězce proměnnou šifrou
- b12 - hra pro jednoho hráče (hádání sekvence)
- b13 - rozhranní pro meteorologická čidla
- b14 - podmnožina procesoru Viper
- b15 - podmnožina procesoru 80386
- b16 - parametrizovaný, težce inicializovatelný obvod
- b17 - tři kopie obvodu b15
- b18, b19 - dvě kopie b14 a dvě kopie b17
- b20 - kopie b14 a modifikovaná verze b14
- b21 - dvě kopie b14
- b22 - kopie b14 a dvě modifikované verze b14

C Obsah přiloženého CD

Obsah přiloženého CD je rozčleněn do následujících adresářů:

documentation - obsahuje instalační a uživatelskou příručku

materials - materiály k souborovým formátům testovacích obvodů a některým sadám

programs - zdrojové kódy ke všem programům zpracovaným v rámci diplomové práce

text - text diplomové práce ve formátu PDF a zdrojové kódy v systému LaTeX