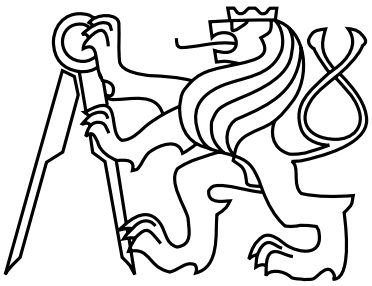
Zadání práce

Upravte ATPG (program pro generování testovacích vektorů) Atalanta pro použití pod více OS (Unix, Windows, ...), do čistého C++, tj. objektového modelu. Uvažujte možnosti rozšíření nástroje pro podporu rozsáhlejších vstupních obvodů, resp. pokuste se nalézt a eliminovat původní implementací daná úzká místa programu. Vytvořte rozhraní pro volání jednotlivých funkcí. Z celého nástroje vytvořte knihovnu (lib, dll). Funkčnost nástroje ověřte na zkušebních úlohách a srovnejte s původním kódem (z hlediska rychlosti). Původní zdrojový kód je v C.

České vysoké učení technické v Praze
Fakulta elektrotechnická



Bakalářská práce

Úprava ATPG nástroje Atalanta

Myslík Radovan

Vedoucí práce: Fišer Petr Ing., Ph.D.

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

červenec 2008

Poděkování

Chtěl bych poděkovat hlavně své manželce za její trpělivost, kterou se mnou měla při psaní této práce, ale i po celou dobu studia. Dále pak vedoucímu práce panu Fišerovi, který si na mne vždy udělal čas.

Prohlášení

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

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

V Příbrami dne 9. července 2008 .....

Abstract

This bachelor work deals with the transformation of source codes of the Atalanta-M application, from the C programming language to the C++ programming language. The resultant source code contains additional modifications which improve the functionality and future usability of the application. Key Improvements are the transformation from an application to a dynamically linked library and the possibility to compile this code on multiple operating systems. These source code modifications increase the usability of this library in the diagnostic process of real logic circuits.

Anotace

Tato práce se zabývá převodem zdrojových kódů aplikace Atalanta-M z jazyka C do jazyka C++. Výsledný programový kód bude navíc obsahovat úpravy pro vylepšení funkčnosti a budoucího použití této aplikace. Vylepšením, je převod z aplikace na dynamicky linkovanou knihovnu a kompilovatelnost na více operačních systémech. Tyto úpravy programového kódu zvýší použitelnost této knihovny v diagnostice reálných obvodů.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

V této kapitole se nejprve seznámíme s problematikou, kterou převáděný program řeší.

Jedním z hlavních směrů dnešní vědy je bezesporu oblast výpočetní techniky. V této oblasti dochází k největšímu pokroku od dob vynalezení tranzistoru. Lidé spatřují stále nové a nové možnosti využití zařízení výpočetní techniky. Vzhledem k tomu, že celý obor výpočetní techniky je velice rozsáhlý, začal se dělit na jednotlivá odvětví, na která je třeba se úzce specializovat, protože objem informací v jednotlivých oblastech je veliký a nelze tyto informace všechny vstřebat. Jedním z hlavních směrů je vývoj logických obvodů, které jsou velice úzce spjaty s obvody integrovanými, které jsou další fází ve výrobě. Dnešní integrace polovodičů je stále větší, tj. dnešní stejně veliké integrované obvody mohou mít (a mají) mnohem více funkcí, než měli jejich předchůdci z dob minulých. Se vzrůstající složitostí obvodů vzrůstá i chybovost, která může být způsobena špatným návrhem, ale také nedokonalou technologií při výrobě. Technologie výroby integrovaných obvodů nejsou vůbec dokonalé a obzvláště v malých měřítkách, jako jsou nanometrové technologie. Se vzrůstající chybovostí vzrůstají i požadavky na snížení až odstranění chybovosti v integrovaných obvodech. Jedna z metod jak chybovost IO (Integrovaný Obvod) snížit, je vylepšení technologie výroby, která s sebou nese spousty komplikací a nákladů, které nemusí být akceptovatelné. Druhou metodou jak zvýšit spolehlivost IO je jejich testování a špatné kusy vyřadit. Tímto se zvýší spolehlivost IO pro zákazníky, ale je to jen odstranění důsledků, nikoliv příčiny. Oblast testování logických nebo integrovaných obvodů neustále vzrůstá z důvodu toho, že náklady na testování a ztráty špatných obvodů s sebou nesou nižší náklady než vylepšení vlastní technologie výroby. Rozvíjejí se algoritmy jak rychle a efektivně testovat logické obvody s co nejnižšími náklady a největší rychlostí. Testy těchto obvodů bývají různé a neomezují se jen na prostý výsledek prošel/neprošel. V současné době je potřeba testovat i vnitřní struktury obvodu, aby se zjistilo, kde nastala chyba a případně jaká. Pro oblast testování logických obvodů se vžil termín diagnostika, který byl pro své velmi podobné metody a řešení převzat z oboru lékařství. Tato oblast má několik fází. Po vytvoření struktury logického obvodu, se provede vygenerování testů pro odzkoušení obvodu. Následuje simulace testů, což je kontrola testů zda jsou bezhazardové, úplné a rychlé. Po simulaci následuje fyzické otestování logického obvodu. V prvních dvou fázích není třeba žádný hardware, protože generování testů probíhá pouze na základě informace o struktuře obvodu nebo jeho funkce. To dělí generování testů na dvě oblasti, kde první je generování strukturních testů a druhá je generování funkčních testů. Druhá fáze je simulace, kde není potřeba vlastní testovaná jednotka, protože můžeme použít simulaci číslicovou, která poskytuje nejspolehlivější a nejrychlejší výsledky a umožňuje vytvářet podklady pro údržbu a zejména slovníky poruch. Velmi často jsou obě fáze spojeny a to tak, že po vygenerování testů je provedena jejich simulace.

Dnešní vývoj v této oblasti je hlavně v automatizaci testování obvodů, který dokáže spolupracovat se systémy CAD (Computer-Aided Design) a systémy CAM (Computer-Aided Manufacturing). Ze systémů CAD se berou výstupy, které obsahují strukturu obvodu, která je načtena do ATPG (Automatic Test Pattern Generator) nástroje, který generuje strukturní testovací vektory a provádí jejich simulaci. Výstupy tohoto nástroje jsou pak následně užity při výrobě logického obvodu, kde po jeho výrobě se na základě informací z ATPG ihned otestuje. Údaje z ATPG zpravidla obsahují seznam testovacích vektorů a seznam poruch. Tímto způsobem dochází k automatizaci procesu. Při fyzickém testování výrobku se vezme seznam vygenerovaných testovacích vektorů, který se postupně posílá na testovaný výrobek a porovnávají se výstupy (odezvy) tohoto výrobku s výstupy správnými.

Je nutné si předem ujasnit názvosloví, které je třeba striktně dodržovat. V problematice generování testů jsou zavedeny pojmy porucha a chyba. Porucha je jev, spočívající v ukončení schopnosti obvodu plnit požadovanou funkci. Chyba je rozdíl mezi správnou a skutečnou hodnotou nějake veličiny, zjištěný měřením nebo pozorováním. Chyba je vždy důsledkem nějaké

poruchy, ale porucha se nemusí vždy projevit chybou.

1.1 Generování testů

1.1.1 Funkční testy

Test se označuje jako funčkní, pokud byl sestaven na základě funkce testované jednotky, tj. bez znalosti její struktury. Funkční testy jsou generovány na základě několika metod mezi něž patří booleovská diference nebo tabulky úplných testů. Pro tyto testy se typicky vytvoří model, který je izomorfní se strukturou (má stejnou funkci) testovaného obvodu. Tento model se setatví na základě výše uvedených metod.

1.1.2 Strukturní testy

Strukturní test má za úkol co nejblíže specifikovat místo v obvodu, kde se porucha vyskytla. Při generování tohoto typu testů je nutné znát vnitřní strukturu testovaného obvodu. Testované poruchy jsou typu t0 nebo t1. Anglické označeni těchto poruch je stuck-at x, kde x je 0 nebo 1. Proto se často zaměňuje označení t0 za sa0 a t1 za sa1. Tyto typy poruch v reálu představují hlavně přerušení, kdy se na vstupu logického obvodu objeví logická 1 nebo zkrat, kde zkratem je míňeno zkrat na signálovou zem, kdy se na vstupu logického obvodu objeví logická 0. Tyto poruchy jsou trvalé, tj. s časem nemění svoji hodnotu. Strukturní testy jsou velmi důležité, protože pro technika, který provádí diagnostiku je velice důležité určit v jaké části obvodu se porucha(-y) nachází, aby mohl obvod opravit nebo vyměnit. V technické praxi se velmi praktikuje výměna celého technického celku. Z toho vyplývá, že již dále nezáleží zda se v technickém celku nachází jedna nebo více poruch, protože tento celek bude vyměněn stejně celý.

Při strukturních testech se využívá několik metod. První metodou je zcitlivění cesty. Tato metoda představuje základní kámen pro generování strukturních testů. Aby bylo možné testovat obvod na nějakou poruchu, je potřeba, aby se chyba způsobená touto poruchou projevila na výstupech obvodu. Toho je možné dosáhnout tak, že všechny logické obvody, přes které signál prochází směrem k výstupům obvodu, se musí nastavit tak, aby se změny tohoto signálu přenášely až na výstup(-y) testovaného obvodu. Zkráceně lze říci, pokud poruchový signál prochází logickým členem AND nebo NAND, pak je třeba ostatní vstupy tohoto logického členu nastavit do stavu log. 1 a naopak, pokud signál prochází logickým členem OR nebo NOR, pak je třeba ostatní vstupy tohoto logického členu nastavit do stavu log. 0. Metoda zcitlivění cesty není algoritmická, což znamená, že se nedá naprogramovat. To bylo důvodem vytvoření D-Algoritmu, který tuto metodu algoritmizoval a bylo tak možné provést zcitlivění cesty programově. Pro tento algoritmus byly nalezeny heuristiky, které byly příčinou nových algoritmů s názvy PODEM a FAN. Metoda zcitlivění cesty je použita při generování jednoho kroku testu.

Druhou metodou je generování jednoho kroku testu. Cílem této metody je vygenerování testovacího vektoru, který pokryje co nejvíc poruch. Tuto metodu opakujeme až do té doby než pokryjeme všechny možné poruchy. Nutno podotknout, že ne vždy je možné pokrýt všechny poruchy, protože některé poruchy jsou nedetekovatelné (například u redundatního obvodu). Úplným pokrytím je zde tedy myšleno vygenerování testovacích vektorů pro všechny detekovatelné poruchy.

1.2 Simulace

Simulace provádí kontroly odezev, úplnosti a bezhazardnosti. Každá z kontrol potřebuje jinou modifikaci algoritmu. My se budeme zabývat kontrolou úplnosti, která je z uvedených kontrol nejsložitější.

Simulace obecně rozdělujeme na tři formy a to: ruční, fyzická a číslicová. Z těchto tří forem nás zajímá ta poslední, která při kontrole testu dává nejrychlejší a nejspolehlivější výsledky.

Metod simulace je několik. První a nejjednodušší je metoda programové injekce poruch do simulovaného obvodu. Zkoumaný obvod při simulaci nějaké poruchy se změní na jiný obvod, který je třeba simulovat v samotném průchodu. Další metoda je paralelní simulace, kde se využívá možnosti zpracování několika bitových operací najednou. Vzhledem k nezávisloti těchto operací a uložení jsou jednotlivé bity do obvodu injektovány najednou, kde každý bit je rezervován pro simulaci jedné poruchy. Další metoda je deduktivní simulace, která je založena na simulaci bezporuchového obvodu a odvozováním detekovaných poruch na jednotlivých vodičích. Další metoda je souběžná simulace, která funguje tak, že každý logický člen obvodu je vybaven seznamem poruch a jejich možných důsledků. Více o těchto metodách v [3].

2 Cíle bakalářské práce

Mým úkolem v této bakalářské práci, je převod programu ATPG (Program na generování testovacích vektorů) do programového jazyka C++ a zároveň z daného programu udělat dynamicky linkovanou knihovnu. Programový kód musí být kompilovatelný alespoň pod operačními systémy Linux a Windows. Tento úkol se dělí na několik dílčích úkolů, které jsem řešil každý zvlášť. Zkrácený popis těchto úkolů následuje a více je popsán v kapitole 4

- Převod do jazyka C++ a objektový model: Původní program Atalanta-M je naprogramován v jazyce C. Já jsem dostal program vytvořený v C++ studentem Zlatuškou, který jej vypracoval jako semestrální projekt. Problém tohoto kódu v C++ bylo, že nefungoval správně, respektive výsledky, které generoval, byly zcela odlišné od původního programu Atalanta-M a navíc v několika případech došlo i k jeho zacyklení nebo spadnutí. Z tohoto upraveného programu jsem převzal objektový model, který jsem celý zachoval a nijak zásadně neupravil. Funkce původního programu byly skoro všechny implementovány, tj. já jsem doprogramoval ty chybějící.
- Rozšíření pro rozsáhlejší obvody: Dalším dílčím úkolem bylo najití "úzkých" míst programu, které omezovala užití tohoto programu pro rozsáhlejší obvody. Bylo nutné nalézt tyto místa a pokusit se je eliminovat. V programu od studenta Zlatušky již byla tato místa identifikována a upravena na užití standartních tříd knihovny STL užívané v jazyce C++.
- Vytvoření rozhraní pro volání jednotlivých funkcí: Vzhledem k tomu, že program byl převeden na dynamicky linkovanou knihovnu, tak bylo nutné pozměnit rozhraní volání jednotlivých funkcí.
- Vytvoření dynamicky linkované knihovny: Původní programový kód bylo nutné upravit tak, aby byl kompilovatelný i jako dynamicky linkovaná knihovna a bylo jej možné využívat i v jiných programech bez nutnosti volání tohoto programu pomocí příkazové řádky s parametry. Toto přináší mnohem lepší funkčnost, protože sdílená data je možné předat přímo bez nutnosti ukládání na disk.
- Ověření funkčnosti: Funčknost výsledného programu bylo nutné ověřit. Ověření funkčnosti jsem prováděl pomocí porovnání výsledků obou programů. Pro testování byla použita schémata reálných obvodů.
- Srovnání rychlosti: Nakonec jsem provedl srovnání rychlosti obou programů. Pro výsledky jsem použil data generovaná přímo v průběhu zpracování testovacích vzorků. Tato data jsem následně vynesl do tabulky.

Cíle bakalářské práce jsou následující:

- Převod do C++
  - Stanovení objektové struktury výsledné programové knihovny
  - Přiřazení funkcí ke třídám C++
  - Úprava C funkcí pro C++
- Najít a eliminovat úzká místa programu
- Vytvoření dynamické knihovny
  - Vytvoření rozhraní pro volání jednotlivých funkcí

  - Úprava kódu na dynamickou knihovnu
- Úprava zdrojových kódů tak, aby byly kompilovatelné pod OS Windows a Linux
- Srovnání z hlediska rychlosti obou programů

2.1 Motivace bakalářské práce

Motivací této bakalářské práce je převod programu původně naprogramovaného v jazyce C do jazyka C++. Pro nový program stanovit objektový model, který kompletně pojme původní funkčnost programu. Nový program však půjde ještě dále a bude z něj vytvořena dynamicky linkovaná knihovna, kterou lze použít i v jiných programech, které předcházejí nebo navazují v procesu vývoje logických obvodů. V praxi se tato knihovna použije v aplikacích CAD nebo CAM (viz. 1). Existuje zde reálná šance implementovat tuto knihovnu jako přídavný prvek do aplikace typu CAD, kde by výstupem z této aplikace pak nebyl pouze soubor, který by obsahoval strukturu, ale také rovnou soubory, které se využijí pro testování daného výrobku. Pak je možné zajít ještě dále a to, kdy se přes tuto knihovnu propojí systémy CAM a CAD, kdy po vygenerování testovacích vektorů dojde k jejich předání aplikaci CAM, která ze struktury obvodu a popis jeho testování může začít vyrábět otestované obvody. To je asi ještě budoucností, ale taková je idea výroby logických obvodů.

Další motivací této bakálářské práce je možnost volby operačního systému. Zpravidla nemusí aplikace typu CAD a CAM běžet na stejném operačním systému, proto tato volba. Z této vlastnosti vyplývá další podmínka a to, že program musí být kompilovatelný pod více operačními systémy. Tímto krokem se výrazně rozšířilo množství využitelného softwaru.

Programem Atalanta-M je možné zkoušet obvody s určitým omezením. V současné době jsou produkovány větší obvody, která mohou překročit velikost omezení Atalanty-M, a program se tak stává nepoužitelným, alespoň z hlediska generování testovacích vektorů. Tyto limity je třeba najít a odstranit.

2.2 Stávající stav programového kódu

Převod z jazyka C do jazyka C++ a stanovení objektového modelu bylo již provedeno studentem Zlatuškou. Implementace kódu v C++ však obsahuje množství chyb a program není schopen vygenerovat ani jeden výstup stejný, jako původní Atalanta-M, napsaná v jazyce C. Nejsou implementovány všechny funkce, ale chybí pouze nějaké pomocné funkce, které však jsou potřebné pro správnou funkci programu. Objektový model bude převzat beze změny struktury.

Převod na dynamicky linkovanou knihovnu včetně rozhraní této knihovny je nutné kompletně implementovat. Je nutné vymyslet jakým způsobem se změní rozhraní, a toto nové rozhraní popsat, aby tato knihovna byla použitelná.

Aby byl program kompilovatelný pod více OS je nutné udělat nějaké změny programového kódu, které tuto možnost zajistí. Tyto úpravy chybí a je potřeba je dodělat.

Eliminace úzkých míst již byla z velké části dokončena studentem Zlatuškou, který implementoval dynamická pole místo statických. Tato změna programového kódu také není bezproblémová a je nutné ji řádně odladit, protože způsobuje zacyklení nebo spadnutí programu.

Program převzatý od studenta Zlatušky je tedy z velké části převeden do objektového modelu. Chybí implementace některých funkcí, které je třeba doplnit. Program sám o sobě pravděpodobně obsahuje spoustu chyb, protože provedené testy programu od studenta Zlatušky vygenerovaly úplně jiný výsledek než program Atalanta-M. Při implementaci programového kódu v C++ došlo zároveň k obměně statických polí za dynamická pole. Na několika místech je tato implementace provedena chybně, a proto i tyto úseky je třeba opravit, aby fungovaly správně.

2.3 Nové funkce

K programu bude přidána i jedna nová funkce a tou je možnost simulace obvodu na základě náhodně vygenerovaného LFSR vektoru.

3 Analýza převodu

3.1 Analýza aplikace Atalanta-M

Nejprve je potřeba vytvořit analýzu původního rozhraní a funkčnosti aplikace Atalanta-M a následně implementovat úpravy vyplývající z této analýzy.

3.2 Stručný popis funkcí

Aplikace Atalanta-M je ATPG nástroj a simulátor poruch. Dokáže generovat testovací vektory pro všechny s-a poruchy, nebo pro jednotlivě specifikované. Umožňuje třístavovou simulaci. Vstupní formát souboru BENCH.

3.3 Objektový model

Na začátek upřesnění pojmů třída, objekt, objektový model, funkce a metoda. Dle definice jazyka C++, je za třídu považován programový kód, který je definicí objektu, tj. jaké bude mít objekt proměnné, jaké funkce bude mít a co budou dělat. Za objekt je považována až instance třídy, tj. kdy dle definice je vytvořen daný objekt v paměti, který již obsahuje reálná data. Objektový model se zachoval z minulosti, kdy předchozí programovací jazyky mluvily o objektu, ať už se jednalo o definici objektu (dnes třída) nebo instaci objektu (dnes objekt). Funkce je samostatný programový kód, který zpracovává vstupní data. Pokud funkci definujeme uvnitř třídy tak se této funkci říká metoda.

Objektový model byl již vytvořen studentem Zlatuškou. Tento objektový model jsem převzal a začal jej zkoumat. Pro lepší orientaci jsem si vytvořil UML diagram, který jsem následně aktualizoval dle mých úprav programového kódu. Výsledný UML diagram je na obrázku 8.1. Hlavním vodítkem pro stanovení jmen tříd bylo rozdělení funkcí do souborů. Funkce jsou rozděleny do souborů podle toho jak k sobě logicky patří, tj. v souboru jsou obsaženy funkce, které řeší jeden specifický algoritmus. Příkladem je soubor fan.c, který obsahuje všechny funkce potřebné pro provádění algoritmu FAN, nebo soubor ppsfp.c, který obsahuje všechny funkce potřebné pro paralelní simulaci poruchových vektorů.

Existují funkce, které jsou definovány v jednom souboru, ale používané jsou i v jiném(-ých) souboru(-ech). Tyto společné funkce nám stanovily dědičnost mezi třídami. Funkce byly definovány s přístupem protected, což zaručuje, že třídy, které od této třídy dědí, mohou tuto funkci bez problémů používat.

Stejně tak jako jsou funkce definovány v jednotlivých třídách, tak je nutné definovat i datové proměnné, se kterými funkce pracují. Při definici proměnné, je třeba uvést její přístupnost, tj. oblast, kde je proměnná platná. Přístupové módy jsou public - třída/metoda je veřejně dostupná, protected - třída/metoda je dostupná v základní třídě a třídách, které dědí z této třídy a private - třída/metoda je dostupná pouze v třídě, ve které je definována.

Všechny třídy jsou umístěny do jmenného prostoru atalantadll.

3.4 Rozšíření pro rozsáhlejší obvody

Původní program Atalanta-M má některá omezení, která jej nedovolují použít na větší obvody. Tento stav je způsoben hlavně datovými strukturami. V programu jsou definovány preprocesorové konstanty, které nastavují limity datových struktur. Po změně těchto konstant je nutné program překompilovat. Navíc některé datové struktury užívají datové typy, které nemohou pojmout celý rozsah, pokud je nastaven příliš velký, tzn. naráží zde na omezení datových typů. Pro odstranění těchto limitujích faktorů, jsou použity dynamické datové struktury z knihovny STL

([1]), kde jsou již implementovány základní a nejpoužívanější dynamické datové struktury jako dynamické pole, dynamický seznam atd. Pro eliminaci tohoto problému jsou s výhodou použity tyto datové struktury, které umožňují dynamicky se měnící velikost, tj. velikost která se mění za běhu programu. Protože se velikost mění dynamicky, nejsou zde limity, které by omezovaly velikost tohoto pole kromě dostupné operační paměti a následně pak možná velikost adresace. Vzhledem k tomu, že je možné adresovat až 2GB na 32-bitových OS tak se domnívám, že bych se dostal do potíží s adresací. Tato adresace se pomocí jistého nastavení kompilátoru dá ještě zvýšit, ale předpokládám, že to nebude třeba.

Tato úzká místa již byla identifikována a nahrazena zmíněnými dynamickými datovými strukturami studentem Zlatuškou. Vzhledem k náhradě těchto datových struktur, došlo i ke změně kódu, který však není všude správně implementován, a proto je nutno jej opravit. Tím je míněno, že změněný kód má na některých místech jinou funkci než původní program a to vede k pádům aplikace. Tento programový kód je nutno opravit.

3.5 Vytvoření rozhraní dynamické knihovny

Vzhledem k tomu, že program bude převeden do dynamické knihovny je potřeba vytvořit rozhraní pro volání jednotlivých funkcí. Protože pracujeme s objekty, tak je nutné definovat pouze dostupnost funkcí vně jmenného prostoru tj. správné nastavení vlastností public, metod jednotlivých tříd.

Rozhraní musí umožňovat nastavení stejné funkčnosti jako předchozí aplikace, tj. nastavení příslušných parametrů algoritmů a zároveň předání vstupních a výstupních dat aplikaci, která knihnovnu bude využívat.

Prvním rozdílem rozhraní je předávání vstupních parametrů. Vstupní parametry programu Atalanta-M jsou předávány pomocí příkazové řádky. Příkazová řádka se rozloží na jednotlivé "přepínače" a jejich případné parametry. Popis jednotlivých přepínačů je v [2]. Příkazová řádka slouží pouze k nastavení parametrů užitých algoritmů, ale vlastní zpracovávaná data jsou předávána pomocí souborů. Odkazy na soubory, ve kterých jsou data uložena, jsou ve formě jména souboru včetně jeho plné nebo relativní cesty.

Z výše uvedeného vyplývá, že pro vygenerování testovacích vzorků bylo nutné všechna potřebná data uložit na disk do souboru(-ů) a tyto soubory pak zpětně načítat pomocí programu Atalanta-M. Diskové operace jsou nejpomaleší operace v počítači. Tyto operace lze eliminovat a to tak, že potřebná data předáme pouze v operační paměti bez zápisu na disk, respektive si je můžeme uložit i na disk jako případnou zálohu nebo uložení výsledku operací, ale dynamické knihovně je předáme pouze v operační paměti. Abych toho docílil, bylo nutné upravit stávavající rozhraní programu. Možné způsoby předání jsou:

1. Předání pomocí řetězce, který bude mít stejný formát jako na příkazové řádce. Řetězec se rozloží na jednotlivé přepínače a jejich parametry. Takto připravený řetězec pak postupně nastaví parametry v pořadí v jakém byly zadány v řetězci.

2. Pro jednotlivé parametry algoritmů budou vytvořeny nové metody, které budou dané parametry nastavovat a číst. Pro každý parametr bude dodána jedna metoda, která daný parametr nastaví. Metody budou dodány do tříd, kde je možné nastavit příslušnou proměnnou programu. Každá metoda bude mít vlastní ověření platnosti zadané hodnoty, aby bylo možné nastavit pouze validní hodnotu.

3. Pro jednodušší práci s parametry, zavést novou třídu Params, která bude sloužit pro hromadné nastavení parametrů pomocí jedné metody. Třída bude implementovat operátor přiřazení, který bude využit při testování správnosti metod nastavujících parametry.

Výhoda třídy spočívá v tom, že je možné implementovat serializaci této třídy, která nám umožní uložení a zpětné načtení konfigurace ze souboru nebo jiného proudu a tímto způsobem je možné uchovat několik nastavení - konfigurací.

4. Použití nějaké statické třídy, která bude obsahovat statické proměnné, které se nastaví dle vstupních parametrů.

První způsob bych zde zachoval z důvodu zpětné kompatibility. Není potřeba měnit žádný kód, protože aplikaci, která bude užívat tento způsob, stačí pouze vytvořit pole typu char*, kam uloží jednotlivé přepínače a jejich hodnoty v pořadí, jako na příkazové řádce a toto pole, spolu s počtem prvků v tomto poli, předat metodě pro spustění programu. Druhý možný způsob je nechat předat pole přímo z příkazové řádky, tj. aplikace, která využívá tuto knihovnu, vezme pole, které jí bylo předáno při spuštění a to pouze předá metodě pro spuštění programu. Zde však nastává riziko, že spolu s těmito parametry budou předány i jiné parametry, které nejsou určeny pro program Atalanta. Pokud k tomu dojde, tak Atalanta vypíše nápovědu k jednotlivým parametrům a skončí. Existuje možnost toto chování změnit a prostě neplatný přepínač ignorovat, ale zde hrozí riziko, že by mohlo dojít k chybné interpretaci následujících parametrů a to by bylo nežádoucí. Nehledě na to, že aplikace, která by užívala knihovnu, by nemohla používat přepínače definované v Atalantě.

Druhý způsob bych implementoval. Důvodem pro implementaci je, že jednotlivé metody jsou umístěny v třídách podle toho kde je proměnná deklarována. V těchto metodách se udělají základní validace nastavovaných proměnných, aby byly zajištěny korektní hodnoty v programových proměnných.

Třetí způsob bych implementoval. Důvodem pro implementaci je možnost nastavení všech parametrů najednou. V metodě, která bude nastavovat parametry lze následně provést validaci parametrů. Validace je možná i vlastním porovnáním parametrů, které by se mohli ovlivňovat. Jednou z dalších výhod definice této třídy je možnost vytvoření několika instancí této třídy, které si samostaně nastavíme a tyto instance budeme používat podle potřeby. Nejlépe to vysvětlím na příkladě.

Uvažujme příklad, kdy budeme chtít pro každý obvod provést vygenerování testovacích vektorů a jejich následné odsimulování. V této situaci si vytvoříme dvě instance třídy Params, kde v první instanci této třídy si nastavíme parametry Atalanty tak, jak požadujeme vygenerovat testovací vektory a v druhé instanci si nastavíme parametry Atalanty tak, abychom tyto vektory odsimulovaly. Zavoláme metodu setParams, kde parametr bude první instance třídy Params, která obsahuje nastavení Atalanty na generování testovacích vektorů. Spustíme Atalantu. Po doběhnutí zavoláme znovu metodu SetParams, ale tentokrát bude parametrem druhá instance třídy Params, která obsahuje nastavení parametrů na simulaci testovacích vektorů. Znovu spustíme Atalantu. Po doběhnutí si můžeme prohlédnout výsledky našich testů.

Čtvrtý způsob bych neimplementoval. Důvodem je to, že v jeden okamžik může být aktivní pouze jedna konfigurace. Je to dáno tím, že změnou statického parametru se tento parametr nezmění pouze pro jednu danou instanci Atalanty, ale pro všechny instance, i ty, co již běží. Toto chování programu je nežádoucí.

Po analýze vstupních parametrů z manuálu [2] byly stanoveny metody, které jsou vypsány v tabulce 8.1.

Vstupní a výstupní soubory jsou nahrazeny proudy (streams). Třídy, kam bude definována příslušnost těchto proudů, budou deklarovat typy proudů, které budou odpovídat typu souboru jako u aplikace Atalanta. Je tím míněno, že pokud je soubor Bench vstupní, tak jím i nadále zůstane. To bude zajištěno tak, že v dané třídě bude deklarován proud jako vstupní (istream) a pro výstupní soubor bude deklarován proud výstupní (ostream). Jednotlivé proudy mají za úkol předávání dat mezi knihovnou a aplikací, která knihovnu využívá. Daty jsou v tomto případě myšleny informace, které knihovna čte či generuje. Výstupní data, například testovací vektory,

mohou být dále využity pouhým přečtením proudu. Aby bylo možné tento proud zpětně přečíst, je nutné soubor v aplikaci, využívající tuto knihovnu, definovat jako proud vstupně/výstupní (iostream). Tento proud se předá knihovně jako výstupní proud, například proud pro uložení testovacíh vektorů. Proudy mají ještě navíc i tu výhodu, že je možné předat jako vstupní nebo výstupní soubor i fstream. Fstream je proud, který umožňuje uložení či načtení dat ze souboru. Výhoda spočívá v tom, že zůstává možnost realizovat načítání a ukládání do souborů tak, jak tomu bylo v původní aplikaci Atalanta. Jednotlivé způsoby se dají i kombinovat.

Příklad kombinace iostream a fstream. Aplikace, která generuje bench soubor se upraví tak, že bude používat tuto knihovnu. Textovou informaci, kterou by ukládala do souboru, uloží do vstupně/výstupního proudu (iostream) a tento stream se následně předá knihovně jako vstupní stream. Jako výstupní stream se nastaví soubor na disku (fstream). Poté se nastaví parametry jednotlivých algoritmů pro generování správných testovacích vektorů a spustí se generování. Výsledkem tohoto příkladu bude vygenerování testovacích vektorů, které budou uloženy na disku v souboru, který byl definován, ale data původně ukládaná do bench souboru se do tohoto souboru vůbec neuloží, ty pouze "projdou" pamětí počítače, ale na disku se vůbec neuloží. To by mělo mít za následek zrychlení zpracování vstupních dat, protože diskové operace spojené s načítáním souboru budou vyměněny za operace s pamětí, které jsou o mnoho rychlejší.

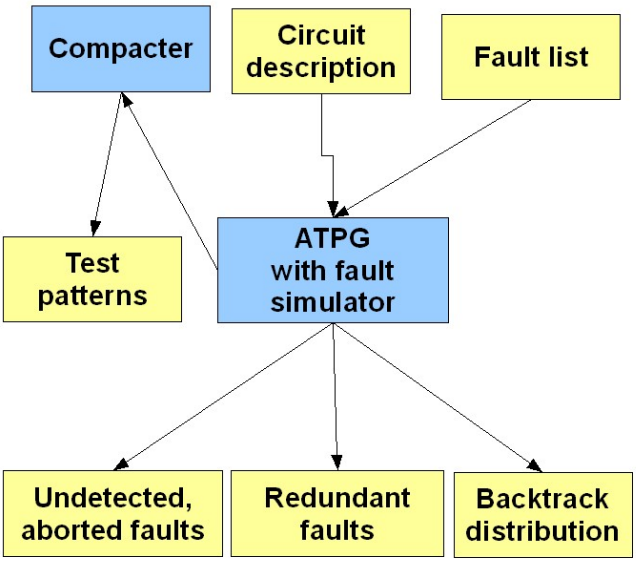
Implementace příkladu (uvedeného zde 3.5) může být provedena právě pouze v paměti číslicového počítače a to tak, že si nadefinuji vstupně/výstupní proud (iostream), který předám jako výstupní proud pro generování testovacích vektorů a po vygenerování těchto vektorů, spustím atalantu znovu s nastavením parametrů pro simulaci a nadefinovaný vstupně/výstupní proud, který byl naplněn v předchozím kroku, předám jako vstupní parametr pro simulaci. Výsledek tohoto běhu si uložím na disk. Zde je vidět, že mezivýsledky v podobě testovacích vektorů a seznamu poruch není třeba ukládat na disk, ale pouze předat.

Program používá dva vstupní soubory a pět výstupních. Soubory pro program Atalanta jsou na 3.1, kde jsou označeny žlutou barvou. Jednotlivé formáty jsou popsány na [2]:

- Vstupní sobory
  - Circuit description - soubor, který obsahuje definici a strukturu testovaneho obvodu
  - Fault list - soubor, který obsahuje seznam poruch k otestování
- Výstupní soubory
  - Test patterns - soubor, kam se ukládájí výsledné testovací vektory
  - Processed faults - soubor, kam se ukládají zpracované poruchy
  - AbortedFaults - soubor, kam se ukládají přerušené nebo nedetekované poruchy
  - FaultMask - soubor, kam se ukládají masky poruch
  - Report - soubor, kam se ukládá výsledná zpráva programu

Druhým zásadním rozdílem je změna z aplikace na knihovnu. Aplikace je samostatně spustitelný kód, zatímco knihovna je samostatně existující kód, který není přímo spustitelný, ale umožnujě tento kód znovu použít v jiných programech, které využívají funkčnosti této knihovny a komunikují s ní pomocí předem stanoveného rozhraní, které je v knihovně implementováno. Toto rozhraní by mělo poskytovat možnost plně nastavit všechny potřebné parametry nutné pro správnou funkčnost knihovních funkcí a k co největšímu uspokojení požadavků třetích stran, které budou tuto knihovnu používat. Programový kód knihovny není dostupný. Dostupný je pouze seznam funkcí/metod rozhraní této knihovny a jejich popis.

Třetím zásadním rozdílem bude i chybový systém. Chybový systém je v tomto případě myšleno zpětné hlášení chyb aplikaci, která využívá knihovnu. V původním programu byl tento systém zajištěn pomocí funkce error, která prováděla vypsání poruchové hlášky na standardní



Obrázek 3.1: Schéma vstupů a výstupů programu Atalanta-M

výstup a následně ukončila program. Tento způsob není možné použít u knihovny, protože zde standardní výstup nemusí vůbec být používán a ukončení celého programu je nežádoucí. V tomto případě využijeme rozšíření jazyka C++ a sice vyjímky (exceptions). Veškerá chybová hlášení musí být převedena na vyjímky. Text pro chybové hlášení bude uložen do stringu, který bude obsahovat popis chyby. Volání programového kódu atalanty by pak mělo být uzavřeno v sekci try a catch, aby bylo možné vyjímku zpracovat.

Vytvoření konzolové aplikace stejně jako je předchozí Atalanta-M je vytvoření a spustění tohoto programového kódu:

```
int main(int argc,char** argv)
{
  Atalanta *simulation = NULL;
  int res = 0;
  try {
    simulation=new atalantadll::Atalanta();
    res=simulation->run(argc, argv);
    delete simulation;
  }
  catch(string s) {
    cerr << s;
  }
return res;
};
```

Tento programový kód je vytvořen i na dodaném CD k této bakalářské práce, protože byl použit pro testování funkčnosti programového kódu.

3.6 Dynamická knihovna

Pro dynamickou knihovnu je ještě nutné dodělat jisté úpravy pro operační systém windows, který umožňuje definovat, které třídy a metody budou dostupné z dynamicky linkované knihovny a které ne. Pro operační systém Linux se nic takového nedefinuje, proto je zde nutné zavést preprocesorové makro. Toto makro má tvar:

```
#ifdef _WIN32
  // Export symbols to .dll while using MS C++ compiler
  # define SHARE_EXPORT __declspec(dllexport)
#else
  // There is no need to declare export on Linux
  # define SHARE_EXPORT
#endif
```

Makro se umisťuje před třídy, které chceme vyexportovat z knihovny. Jednotlivé metody vyexportované třídy si stále zachovávají svoji přístupnost, která je definována pomocí klíčových slov public, protected nebo private.

3.7 Kompilovatelnost na více OS

Z názvu již vyplývá, že pro každý operační systém je nutno knihovnu zkompilovat, protože programový kód je překládán do nativního kódu procesoru dle daného OS a neběží v žádném frameworku, který by tento kód interpretoval. Programový kód musí být uzpůsoben tomu, aby ho bylo možné kompilovat na jednotlivých operačních systémech beze změny, proto jsou zaváděna preprocesorová makra, pomocí kterých se řeší tyto rozdíly. Jeden rozdíl je popsán v kapitole 3.6. K tomu, aby tyto preprocesorová makra fungovala správně, je nutné zkontrolovat, zda při kompilaci na jednotlivých OS jsou během kompilace definovány určité preprocesorové konstanty, které určují, jaká část kódu se zkompiluje. Pro operační systém Windows je třeba definovat konstantu _WIN32 a pro operační systém Linux je to _LINUX. Tyto preprocesorové konstanty bývají běžně definovány pokud kompilujeme programový kód z nějakého vývojového prostředí.

Pokud je programový kód kompilovaný pouze pomocí programu make, potom je třeba tuto konstantu nadefinovat. Příkladem souboru, který make používá, je na přiloženém CD v adresáři linux a jménem "'makefile"'.

3.8 Ověření funkčnosti

Aby bylo možné konstatovat, že je program funkční, tak je třeba ověřit jeho funkčnost. V mé situaci mám tu výhodu, že mám již existující funkční a odladěný program, který mi poskytne výsledky pro porovnání. Ověření proto bude nejlepší jako porovnání výsledků jednoho a druhého programu. Výsledky se v tomto případě myslí výstupní soubory z obou programů. U každého testu jsou vypsány výstupní soubory, které se budou porovnávat a výsledek porovnání. Pro tento účel by bylo vhodné vypracovat nějaký testovací skript, který otestuje funkce programu na různých testovacích vzorcích. Testované vzorky budou reálné obvody, které mají uloženou strukturu v bench souborech. Výsledky obou programů se pak následně porovnají. Pokud budou výsledky stejné, pak můžeme s určitostí říci, že je nově naprogramovaný program funkční.

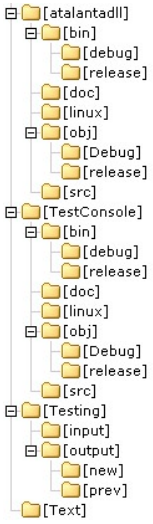
3.9 Srovnání rychlosti

Nakonec je ještě potřeba provést srovnání rychlosti, které nám ukáže, jak se zlepšila nebo zhoršila výkonnost programu při převodu do C++. K porovnání využijeme měření procesorového času v jednotlivých programech. V původním programu je měření procesorového času již implementováno. Toto měření je třeba implementovat také v nově vytvořeném programu. Výsledky měření se zaznamenají do grafu, kde budou všechny křivky vedle sebe a snáze tak vyhodnotíme, zda došlo ke zrychlení nebo zpomalení. Jako zajímavost zde uvádím i výsledky programů, které jsou zkompilovány bez optimalizace.

4 Popis implementace

V této kapitole je popsáno jakým způsobem probíhala implementace a jakým problémům bylo nutné čelit během této fáze. Implementační část je v procesu převodu programu nejnáročnější. Implementace probíhala v Microsoft Visual Studiu 2003, protože to jediné bylo schopné zkompilovat původní kód v jazyce C, kdežto novější Microsoft Visual Studio 2005 spadlo ihned po otevření, tzn. že zde byl zde nějaký problém s analýzou obsahu jednoho souboru.

Vlastní implementace probíhala tak, že jsem nejprve vytvořil projekt a do něj jsem zkopíroval zdrojové kódy, které jsem získal od svého vedoucího práce a které vypracoval student Zlatuška. Projekt jsem nastavil tak, aby kompilátor generoval normální konzolovou aplikaci. To bylo z toho důvodu, že veškeré ladění kódu a jednotlivých funkcí se tak podstatně zrychlilo. Konzolová aplikace měla ještě jednu výhodu, a tou bylo, že přebraný kód od studenta Zlatušky jsem nemusel vůbec změnit, abych byl schopen jej překompilovat, protože používal Microsoft Studio 6.0, které jsem používal i já. Vytvořil jsem nový projekt pro kód a stanovil adresářovou strukturu tak, jak je na 4.1. Tato struktura byla zvolena kvůli lepší orientaci v programovém kódu a s ostatními soubory, jako například projektové soubory aj.



Obrázek 4.1: Struktura adresářů

Obsah adresářů je následující:
atalantadll - obsahuje programový kód knihovny
bin - obsahuje binární spustitelné soubory
doc - obsahuje dokumenty pro dokumentaci programového kódu
linux - obsahuje makefile pro kompilaci pod operačním systémem Linux
obj - obsahuje soubory, které vznikají při kompilaci kódu
src - obsahuje soubory zdrojového kódu, tj. soubory *.h a soubory *.cpp
TestConsole - obsahuje programový kód aplikace užité pro testování
Testing - skripty použité pro testování
Text - zdrojové soubory pro bakalářskou práci

4.1 Objektový model

V objektovém modelu nebylo potřeba udělat žádné zásadní změny a víceméně zůstal beze změny. Byla provedena analýza jednotlivých funkcí a zkontrolováno, zda jsou všechny funkce imple-

mentovány. Na základě toho vznikl dokument, přiloženo na CD, který obsahuje komentáře a vyhodnocení tohoto porování. Byly nalezeny funkce, které nebyly implementovány a to z toho důvodu, že v původním programu sice jsou, ale nejsou využívány, proto nebylo přistoupeno k jejich převodu do programu nového. V průběhu této analýzy jsem zjistil, že chybí několik funkcí a bylo nutné je doprogramovat. Během této analýzy byl tvořen i UML diagram, který následně umožňoval lepší orientaci mezi třídami a jejich metodami. Tento UML diagram navíc do jisté míry ujasnil, k čemu jednotlivé třídy slouží. Objektový model jsem kompletně převzal od studenta Zlatušky, který naprosto vyhovuje představám jak by měl být postaven. Objektový model přibližně odpovídá struktuře souborů původního programu, což je vidět v dříve zmiňovaném souboru, kde vedle jednotlivých metod je uveden i soubor, ze kterého pochází původní programová funkce. V této části nevznikaly žádné vážnější problémy.

4.2 Rozšíření pro rozsáhlejší obvody

Tato náhrada byla již implementována studentem Zlatuškou, který zde ovšem zanechal všechny limity jako původní program, tj. nechal zde omezení na počet vstupů, výstupů a hradel i přesto, že již byly implementovány potřebné datové struktury místo původních, které právě obsahovaly omezení. Z tohoto důvodu jsem musel projít program a tato omezení vyjmout a upravit kód tak, aby fungoval správně a bez omezení.

Rozšíření odpovídá náhradě statických polí polemi dynamickými. V našem případě bylo použito instancí třídy list. Tato třída umožňuje vkládání prvků a jejich následné projití pomocí iterátorů. Je to vlastně kruhový spojový obousměrný seznam, který uchovává ukazatele na vkládané prvky. Třída list je definována jako generická, kde se při vytváření instance definuje typ, který se bude do list-u ukládat. Výhodou je, že tato třída při procházení vrací ukazatel na tento typ prvku a není proto třeba provádět přetypování. Tato třída je plně dostačující, protože v našem případě není potřeba k prvkům přistupovat pomocí indexu, protože nám plně postačují funkce, které tato třída nabízí a to jsou vrácení posledního prvku, vrácení prvního prvku a projití všech prvků v seznamu. Tato implementace byla provedena studentem Zlatuškou. Já jsem ji pouze upravil na místech kde bylo kde byly metody této třídy špatně užity a často kvůli tomu docházelo k zacyklení programu, kdy program běžel jako v nekonečné smyčce.

Původní program měl limity v podobě nastavení statických konstant, které jsou použity na různých místech programu. Všechna tato místa jsem přeprogramoval tak, že jsem tyto limity odstranil a využil tak možností, které nabízí náhrada statických polí za pole dynamická.

4.3 Vytvoření rozhraní knihovny

Rozhraní knihovny bylo vytvořeno dle analýzy. Byly implementovány jednotlivé metody, které umožňují nastavení příslušného parametru nebo proudu.

4.4 Vytvoření dynamicky linkované knihovny

Tento úkol se pro mne zpočátku zdál neřešitelný, protože jsem měl na začátku velké problémy se správným nastavením, aby byla knihovna generována správně. První problémy nastaly při nastavení projektu. Tyto problémy se vyskytly hlavně v operačním systému windows, kde bylo nutné nastavit linker, aby při linkování dynamické knihovny zároveň vytvořil i lib soubor, který je potřebný při linkování dynamické knihovny do jiného programu.

Pří implementaci bylo použito makra, které bylo definováno v hlavičkovém souboru Defines.h. Pro operační systém LINUX se nemusí uvádět nic, proto je druhá část makra prázdná. Toto je příklad jakým způsobem funguje preprocesor, který nám následně umožňuje kompilovat kód na různých operačních systémech.

Abych byl schopen vyzkoušet funkčnost knihovny, bylo nutné vytvořit nový projekt, který generoval program jako konzolovou aplikaci. V tomto programu byl program pro otestování základních funkcí dynamické knihovny. Tato aplikace je příznačně nazvána TestConsole. V tomto consolovém programu jsem nastavil linkeru referenci na lib soubor a následně stačilo definovat deklaraci using a příslušný namespace. Nutno podotknout, že linker dá do souboru lib pouze ty funkce, které jsou exportovány z DLL. Tyto funkce jsou pak volně přístupné externím programům, které tuto knihovnu budou využívat. Knihovnu lze volat a inicializovat dynamicky, ale to jsem do testovací konzole nezahrnul.

Po správném nastavení projektu a následné kompilaci, která se zdařila bez chyby, jsem program vyzkoušel. Jenže jsem narazil na to, že program generoval exception ihned na začátku programu při spouštění metody run, která provádí veškeré funkce programu dle toho jak jsou nastavené. Tento exception obsahoval, že se snažím přistupovat do oblasti na kterou nemám práva. Po delším zkoumání a různém nastavování kompilátoru jsem konečně přišel na to v čem spočívala chyba. Problém byl v tom, že jsem neměl nastaven export celé třídy, ale pouze metod této třídy. To znamená, že kompilátor negeneroval žádnou chybu při kompilaci testovacího programu, ale nastala vyjímka během jeho spuštění s hláškou popsanou výše. Po nastavení exportu celé třídy vše začlo fungovat naprosto bez problémů.

4.5 Nová funkcionalita programu

Do programu byla naprogramována nová funkce. Tato funkcionalita rozšiřuje program o možnost odsimulování seznamu poruch nebo všech poruch pomocí náhodného vektoru pro LFSR (Linear Feedback Shift Register). Programová metoda generuje náhodný polynom a seed. Tento algoritmus má dva parametry. Prvním parametrem je počet unikátních testovacích vektorů. Tento parametr udává periodu, kdy nesmí dojít k zopakování prvního LSFR vektoru. Pokud k tomu během generování dojde, pak se znovu vygeneruje polynom a seed a spustí se generování znovu. Druhým parametrem je výstupní proud, kam se uloží náhodně vygenerovaný polynom a seed, které vyhověly při generování, tj. vygenerovaly dostatečně veliký seznam testovacích LSFR vektorů.

Formát přepínače na příkazové řádce je: -g n ./output/file.lfsr, kde n je délka periody.

5 Výsledky testování

Pro testování byla zvolena metoda porovnání výsledků (výstupních souborů) předchozího a nového programu. Pro testování byly vybrány reálné vzorky obvodů a pro tyto vzorky byl sestaven testovací skript. Pro každý vzorek byla provedena série testů. Při vytváření testů je třeba dbát na to, aby byl stejně nastaven generátor náhodných čísel, aby byly výsledky porovnatelné. Některé testy jsou totiž ovlivněné právě tímto generátorem náhodných čísel a proto by generovaly jiné testovací vektory nebo poruchy.

Testy byly prováděny na strukturách obvodu, které jsou v tabulce 5.1.

V parametrech testu se vyskytuje řetězec %1, který zde zastupuje jméno souboru, které zpracováváme. Je to z toho důvodu, že jsem vytvořil dávkový testovací soubor, který spouští testy s různými bench soubory, aby bylo možné ověřit funkčnost programu co nejvíce. Pro každý testovaný vzorek byly spušteny následující testy:

1. Pro tento test byly použity tyto parametry:
   -s 23 -t ./output/new/%1t1.pat -W 1 %1.bench
   Parametry nastaví generátor náhodných čísel (-s) a výstupní soubor (-t), kam se uloží výsledné testovací vektory a výsledné testovací vektory budou ve formátu 1, což je pouze nastavení vstupů. Pro porovnání budou použity soubory ./output/new/%1t1.pat. Tyto soubory se musí zcela rovnat.

2. Pro tento test byly použity tyto parametry:
   -s 294 -t ./output/new/%1t2.pat -W 1 -F ./output/new/%1t2.flt %1.bench
   Parametry nastaví generátor náhodných čísel (-s) a výstupní soubor (-t), kam se uloží výsledné testovací vektory, výsledné testovací vektory budou ve formátu (-W) 1, což je pouze nastavení vstupů a konečně výstupní soubor (-F), kam se uloží seznam zpracovaných poruch. Pro porovnání budou použity soubory ./output/new/%1t2.pat a ./output/new/%1t2.flt, které se musí zcela rovnat.

3. Pro tento test byly použity tyto parametry:
   -s 23 -t ./output/new/%1t3.pat -W 2 %1.bench
   Parametry nastaví generátor náhodných čísel (-s) a výstupní soubor (-t), kam se uloží výsledné testovací vektory a výsledné testovací vektory budou ve formátu (-W) 2, což je nastavení vstupů a výsledné nastavení výstupů při dané poruše. Tento test je obdoba Testu 1, ale zde se testuje jiný výstupní formát souboru. Pro porovnání budou použity soubory ./output/new/%1t3.pat. Tyto soubory se musí zcela rovnat.

4. Pro tento test byly použity tyto parametry:
   -D 1 -t ./output/new/%1t4.pat -W 2 %1.bench
   Parametry nastaví počet generovaných vektorů na každou poruchu (-D) což je jeden v našem případě, výstupní soubor (-t) kam se uloží výsledné testovací vektory a výsledné testovací vektory budou ve formátu (-W) 1, což je nastavení vstupů. Pro porovnání budou použity soubory ./output/new/%1t4.pat. Tyto soubory se musí zcela rovnat.

5. Pro tento test byly použity tyto parametry:
   -t ./output/new/%1t5.pat -f ./input/%1.flt -s 23 -W 1 %1.bench
   Parametry nastaví generátor náhodných čísel (-s), výstupní soubor (-t), kam se uloží výsledné testovací vektory, výsledné testovací vektory budou ve formátu (-W) 1, což je nastavení vstupů. Atalanta zde používá i vstupní soubor, který je definován parametrem -f a jménem souboru, který obsahuje seznam poruch, které budou testovány. Pro tento test byly vzaty soubory, které byly vygenerovány testem č. 2. Pro porovnání budou použity soubory ./output/new/%1t5.pat. Tyto soubory se musí zcela rovnat.

bench soubor	poly	seed
c17.bench	26	30
c1355.bench	31533157253127	34153607033667
c1908.bench	52615617223	53522557663
c2670.bench	234651236547745612336 450221471230445602776 312354675621446325000 522463700443472	334665566277021644465 633374220214654676333 246520777232446210642 021465467633324
c3540.bench	03654772134652462	00100674372655462
c432.bench	100400000001	765177704700
c499.bench	34561232244723	26577520334465
c5315.bench	13322442203214563 77732242030003413 24322133144321034 443247724	0442323344432224332121 3134477566654555421321 6543220021230520
c6288.bench	02432334562	12345731451
c7552.bench c7552.bench c7552.bench c7552.bench	003143335422477765545 752106315472130546321 646732163445610216473 105463	001432444777556232430 216473132024304646024 762215363616163244061 640770
c880.bench	13245563770242321072	74423064424434506124

Tabulka 5.1: Seznam souborů použitých pro testování

6. Pro tento test byly použity tyto parametry:
   -s 56 -t ./output/new/%1t6.pat -P ./output/new/%1t6.rep -m ./output/new/%1t6.mask -W 1 %1.bench
   Parametry nastaví generátor náhodných čísel (-s), výstupní soubor (-t), kam se uloží výsledné testovací vektory, výsledné testovací vektory jsou uloženy ve formátu (-W) 1 což je nastavení vstupů, výstupní soubor se zprávou (-P) a výstupní soubor kam se uloží maska poruch. Pro porovnání budou použity soubory ./output/new/%1t6.pat, ./output/new/%1t6.rep a ./output/new/%1t6.mask. Z těchto souborů se soubory s extension .pat a .mask musí zcela rovnat. V souborech s extension .rep mohou být rozdíly, ale pouze na poslední řádce, která obsahuje čas, jak dlouho trvalo vygenerování výstupních souborů. Ostatní údaje v těchto souborech se musí rovnat.

7. Pro tento test byly použity tyto parametry:
   -S -t ./input/%1.pat -P ./output/prev/%1t7.rep %1.bench
   Parametry nastaví simulační mód, vstupní soubor (-t), který obsahuje testovací vektory a výstupní soubor (-P), kam se uloží výsledná zpráva simulace. Pro porovnání budou použity soubory ./output/prev/%1t7.rep, které se mohou lišit, ale pouze na poslední řádce, která obsahuje informaci o čase, jak dlouho trvalo odsimulování vstupních souborů. Ostatní údaje v těchto souborech se musí rovnat.

8. Pro tento test byly použity tyto parametry:
   -S -t ./input/%1.pat -P ./output/new/%1t8.rep -U ./output/new/%1t8.ud -v %1.bench
   Parametry nastaví simulační mód, vstupní soubor (-t), který obsahuje testovací vektory, výstupní soubor (-U), kde budou vypsány nezpracované nebo nedetekované poruchy a ve výstupních souborech bude seznam všech nedetekovaných poruch. Pro porovnání budou

použity soubory ./output/new/%1t8.rep a ./output/new/%1t8.ud. Soubory s extension .ud se musejí zcela rovnat. Soubory s extension .rep se mohou lišit, ale pouze na poslední řádce, která obsahuje informaci o čase, jak dlouho trvalo odsimulování vstupních souborů. Ostatní údaje v těchto souborech se musí rovnat.

9. Pro tento test byly použity tyto parametry:
   -S -t ./input/%1.pat -P ./output/new/%1t9.rep -m ./output/new/%1t9.msk %1.bench
   Parametry nastaví simulační mód, vstupní soubor (-t), který obsahuje testovací vektory, výstupní soubor (-m), který bude obsahovat masku poruch. Pro porovnání jsou použity soubory ./output/new/%1t9.rep a ./output/new/%1t9.msk, kde soubory s extension .msk se musí zcela rovnat. Soubory s extnsion .rep se mohou lišit, ale pouze na poslední řádce, která obsahuje informaci o čase, jak dlouho trvalo osdimulování vstupních soborů. Ostatní údaje v těchto souborech se musí rovnat.

10. Pro tento test byly použity tyto parametry:
    -l poly seed 1000 -U ./output/new/%1t10.flt -v -P ./output/new/%1t10.rep %1.bench
    Parametr nastaví simulační mód, testování pomocí LSFR. Tento test obsahuje parametry navíc, které jsou specifické pro každý testovaný vzorek. Parametry poly a seed jsou čísla v oktalové soustavě, která slouží ke generování LSFR vektorů, které jsou pak následně otestovány. Tato oktalové čísla musí obsahovat stejný počet bitů jako je počet vstupů testovaného obvodu. Pokud tomu tak není tak nová atalanta vyhodí vyjímku a předchozí atalanta může i provést neplatnou operaci, kterou je pokus o zápis do nepřidělené paměti, které nastane, pokud oktalové číslo obsahuje méně bitů než je počet vstupů testovaného obvodu. Pro jednotlivé vzorky byla vygenerována oktalová čísla, která jsou uvedena v tabulce 5.1, kde údaj poly na příkazové řádce je nahrazen údajem poly z tabulky 5.1 a parametr seed je nahrazen hodnotou ze sloupce seed z tabulky 5.1.

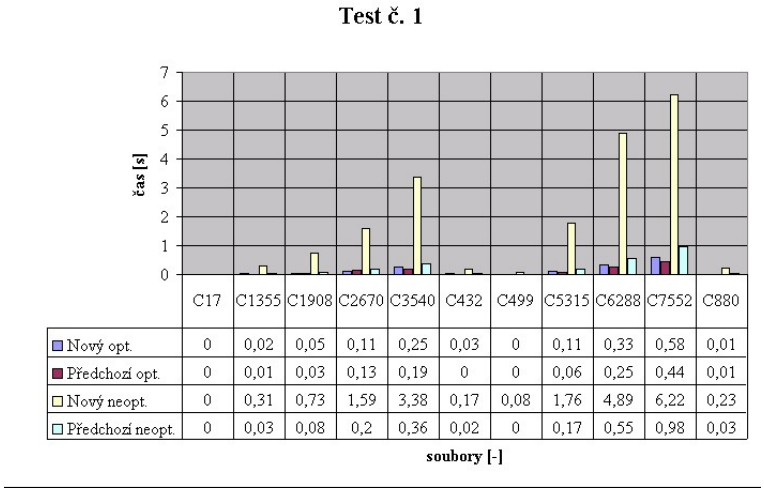
11. Pro tento test byly použity tyto parametry:
    -S -g 10000 ./output/new/%1t11.gr -t ./input/%1.pat -U
    Tento test slouží pro otestování nové funkcionality, kde do souboru s extension .gr jsou uloženy výsledné nastavení pro LSFR simulaci.

5.1 Výsledky testování

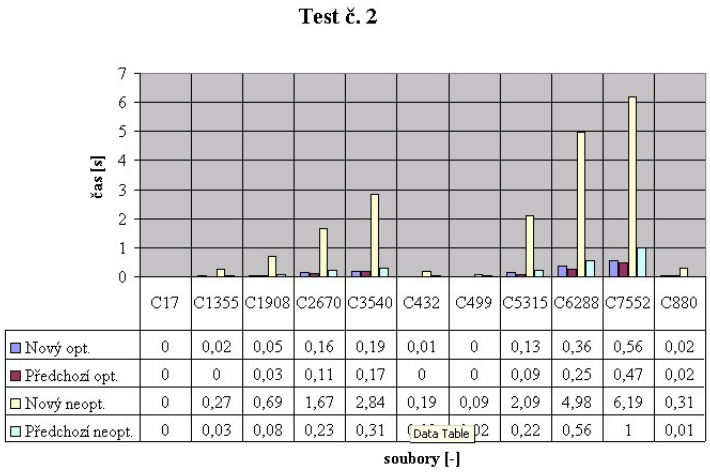
Byl spuštěn testovací skript, který provedl všech 11 testů na každém bench souboru z tabulky 5.1. Jednotlivé soubory byly porovnány pomocí nástroje v TotalCommanderu 7.03, který je schopen porovnávat soubory dle obsahu. Celkem bylo vygenerováno 385 souborů, z toho je 209 vygenerováno převedeným programem Atalanta a 176 původním programem Atalanta-M. Rozdíl je způsoben testem číslo 11, který lze spustit pouze s novým programem Atalanta, protože starý program tuto funkcionalitu neobsahuje. Pokud tedy odečteme 3*11=33 souborů, protože test č. 11 generuje 3 výstupní soubory, tak dostáváme číslo 176, které je stejné jako počet vygenerovaných souborů původním programem Atalanta-M. Po porovnání těchto zbývajících souborů, jsem dostal 45 různých souborů. Všechny soubory s různým obsahem mají příponu .rep, to znamená, že se liší pouze soubory, u kterých jsem tuto možnost připustil. Pro kontrolu byl ješte zkontrolován celkový počet souborů s extension .rep. Bylo jich 55 což souhlasí s počtem testů, které tento soubor generují tj. 5 a to vynásobeno počtem testovacích souborů nám dává 55 což je správně. U souborů které se liší, jsem musel ručně zkontrolovat, zda se liší pouze v posledním řádku, který obsahuje časový údaj. To jsem provedl a všechny soubory se liší pouze v posledním řádku. Tj. všechny soubory jsou stejné.

5.2 Porovnání rychlosti

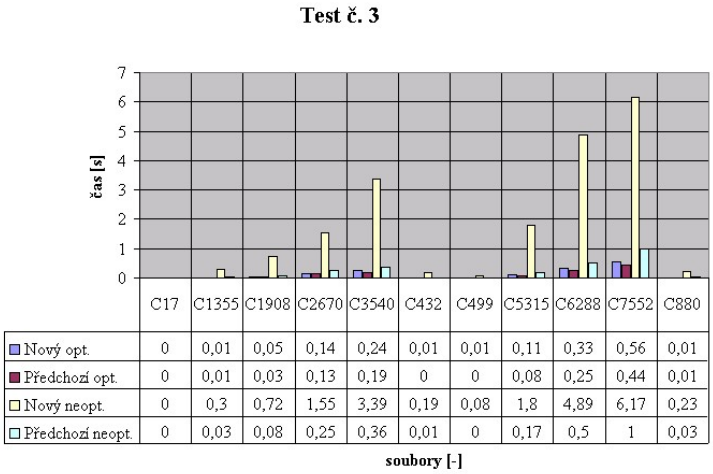
Pro každý test se zapisoval časový údaj o tom, jak dlouho test trval. Tyto údaje jsou vyneseny v následujících 10 grafech, kde každý graf je pro jeden test. Z celkových údajů vyplývá, že nově vytvořený program je téměř stejně rychlý jako původní program Atalanta-M. Pro porovnání rychlosti jsem vzal největší hodnoty naměřených časů pro jeden test, což vychází na test č.4 pro obvod c6288, kde původní program Atalanta-M dosáhl času 7,44 [s] a nový program dosáhl hodnoty 7,53 [s]. Údaj se liší o 1,21% původní hodnoty, to znamená, že došlo k nepatrnému zpomalení. Pro zajímavost jsem uvedl i měření pro zkompilované programy bez optimalizace kódu, kde je vidět, že programovací jazyk C nepotřebuje příliš optimalizovat, protože údaje naměřené s optimalizovaným a neoptimalizovaným programem jsou pouze v nepatrných rozdílech. Naproti tomu když se podíváme na výsledky u nového programu, tak vidíme, že zpomalení u neoptimalizovaného kódu je velice markantní.



Obrázek 5.1: Výsledky testu č. 1

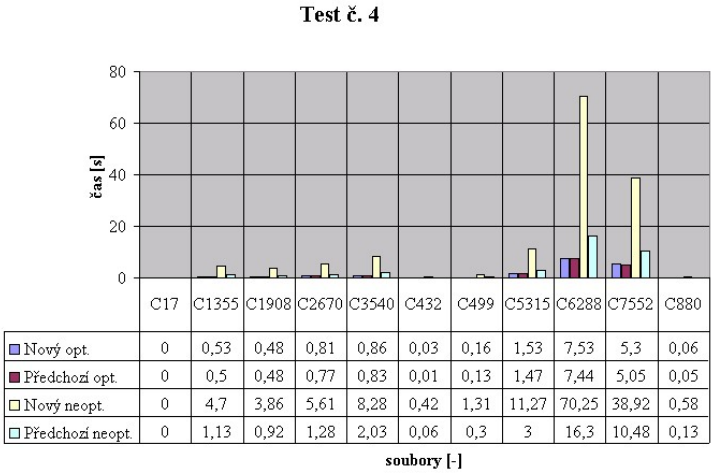


Obrázek 5.2: Výsledky testu č. 2



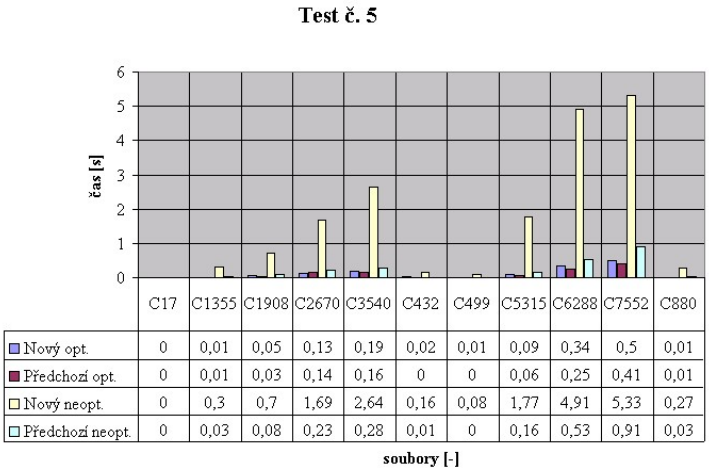
	C17	C1355	C1908	C2670	C3540	C432	C499	C5315	C6288	C7552	C880
Nový opt.	0	0,01	0,05	0,14	0,24	0,01	0,01	0,11	0,33	0,56	0,01
Předchozí opt.	0	0,01	0,03	0,13	0,19	0	0	0,08	0,25	0,44	0,01
Nový neopt.	0	0,3	0,72	1,55	3,39	0,19	0,08	1,8	4,89	6,17	0,23
Předchozí neopt.	0	0,03	0,08	0,25	0,36	0,01	0	0,17	0,5	1	0,03

Obrázek 5.3: Výsledky testu č. 3



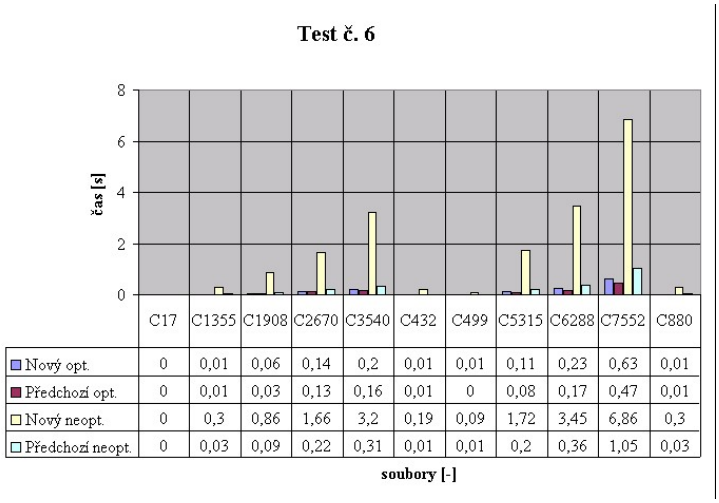
	C17	C1355	C1908	C2670	C3540	C432	C499	C5315	C6288	C7552	C880
Nový opt.	0	0,53	0,48	0,81	0,86	0,03	0,16	1,53	7,53	5,3	0,06
Předchozí opt.	0	0,5	0,48	0,77	0,83	0,01	0,13	1,47	7,44	5,05	0,05
Nový neopt.	0	4,7	3,86	5,61	8,28	0,42	1,31	11,27	70,25	38,92	0,58
Předchozí neopt.	0	1,13	0,92	1,28	2,03	0,06	0,3	3	16,3	10,48	0,13

Obrázek 5.4: Výsledky testu č. 4



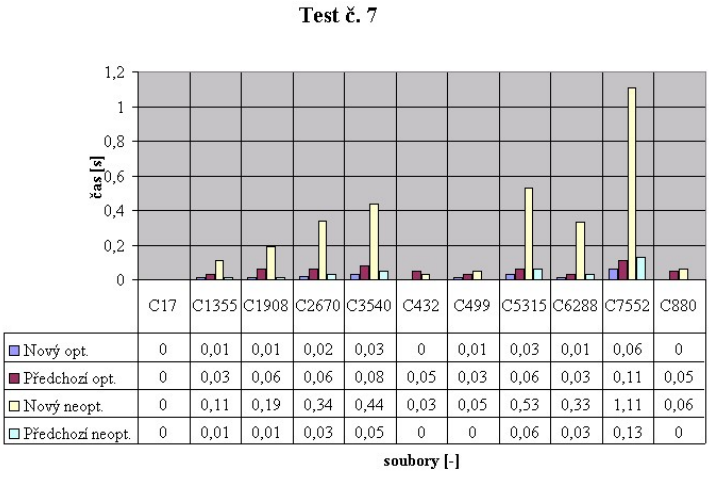
	C17	C1355	C1908	C2670	C3540	C432	C499	C5315	C6288	C7552	C880
Nový opt.	0	0,01	0,05	0,13	0,19	0,02	0,01	0,09	0,34	0,5	0,01
Předchozí opt.	0	0,01	0,03	0,14	0,16	0	0	0,06	0,25	0,41	0,01
Nový neopt.	0	0,3	0,7	1,69	2,64	0,16	0,08	1,77	4,91	5,33	0,27
Předchozí neopt.	0	0,03	0,08	0,23	0,28	0,01	0	0,16	0,53	0,91	0,03

Obrázek 5.5: Výsledky testu č. 5



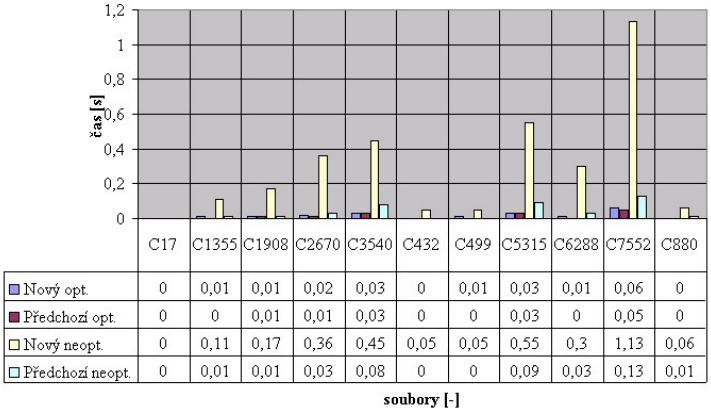
	C17	C1355	C1908	C2670	C3540	C432	C499	C5315	C6288	C7552	C880
Nový opt.	0	0,01	0,06	0,14	0,2	0,01	0,01	0,11	0,23	0,63	0,01
Předchozí opt.	0	0,01	0,03	0,13	0,16	0,01	0	0,08	0,17	0,47	0,01
Nový neopt.	0	0,3	0,86	1,66	3,2	0,19	0,09	1,72	3,45	6,86	0,3
Předchozí neopt.	0	0,03	0,09	0,22	0,31	0,01	0,01	0,2	0,36	1,05	0,03

Obrázek 5.6: Výsledky testu č. 6



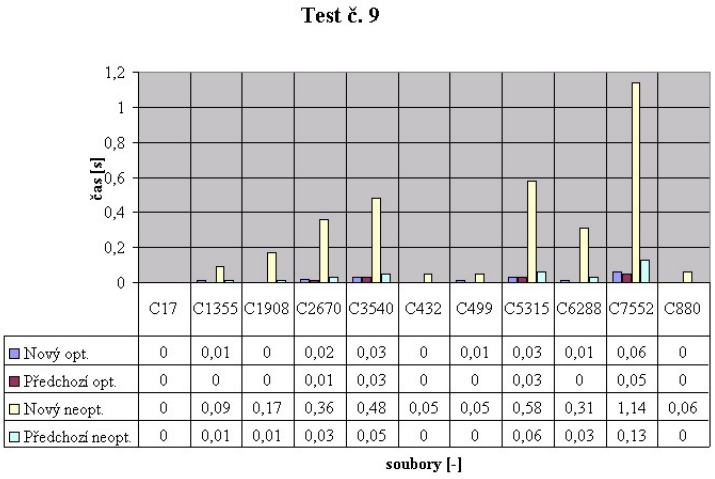
	C17	C1355	C1908	C2670	C3540	C432	C499	C5315	C6288	C7552	C880
Nový opt.	0	0,01	0,01	0,02	0,03	0	0,01	0,03	0,01	0,06	0
Předchozí opt.	0	0,03	0,06	0,06	0,08	0,05	0,03	0,06	0,03	0,11	0,05
Nový neopt.	0	0,11	0,19	0,34	0,44	0,03	0,05	0,53	0,33	1,11	0,06
Předchozí neopt.	0	0,01	0,01	0,03	0,05	0	0	0,06	0,03	0,13	0

Obrázek 5.7: Výsledky testu č. 7



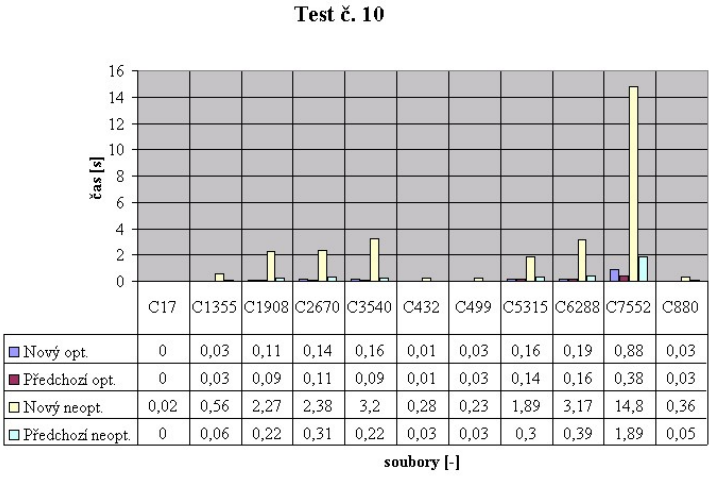
	C17	C1355	C1908	C2670	C3540	C432	C499	C5315	C6288	C7552	C880
Nový opt.	0	0,01	0,01	0,02	0,03	0	0,01	0,03	0,01	0,06	0
Předchozí opt.	0	0	0,01	0,01	0,03	0	0	0,03	0	0,05	0
Nový neopt.	0	0,11	0,17	0,36	0,45	0,05	0,05	0,55	0,3	1,13	0,06
Předchozí neopt.	0	0,01	0,01	0,03	0,08	0	0	0,09	0,03	0,13	0,01

Obrázek 5.8: Výsledky testu č. 8



	C17	C1355	C1908	C2670	C3540	C432	C499	C5315	C6288	C7552	C880
Nový opt.	0	0,01	0	0,02	0,03	0	0,01	0,03	0,01	0,06	0
Předchozí opt.	0	0	0	0,01	0,03	0	0	0,03	0	0,05	0
Nový neopt.	0	0,09	0,17	0,36	0,48	0,05	0,05	0,58	0,31	1,14	0,06
Předchozí neopt.	0	0,01	0,01	0,03	0,05	0	0	0,06	0,03	0,13	0

Obrázek 5.9: Výsledky testu č. 9



	C17	C1355	C1908	C2670	C3540	C432	C499	C5315	C6288	C7552	C880
Nový opt.	0	0,03	0,11	0,14	0,16	0,01	0,03	0,16	0,19	0,88	0,03
Předchozí opt.	0	0,03	0,09	0,11	0,09	0,01	0,03	0,14	0,16	0,38	0,03
Nový neopt.	0,02	0,56	2,27	2,38	3,2	0,28	0,23	1,89	3,17	14,8	0,36
Předchozí neopt.	0	0,06	0,22	0,31	0,22	0,03	0,03	0,3	0,39	1,89	0,05

Obrázek 5.10: Výsledky testu č. 10

6 Závěr

Převod ATPG nástroje, programu Atalanta-M je hotov. Projdu jednotlivé cíle bakalářské práce a zhodnotím co z nich bylo splněno a co ne.

6.1 Převod do jazyka C++ a objektový model

Převod do jazyka C++ a vytvoření objektového modelu bylo již z velké části vytvořeno studentem Zlatuškou. Já jsem implementoval chybějící metody, které byly potřeba pro správnou funkci programu. Navíc jsem během ladění programu přišel na spoustu chyb, které jsem musel opravit, aby program fungoval správně. To mi zabralo spoustu času stráveného s otevřeným debugerem a zkoumáním kódu co vlastně dělá a co by dělat měl. Musím říci, že ladění algoritmu FAN bylo velmi náročné, protože je zde jeden veliký switch (programový kód pro větvení programu), který je navíc rozdělen do několika metod. Nicméně tento převod je hotový a finální verze je vidět na diagramu UML. Tento cíl byl tedy splněn a nemám k němu žádné další poznámky.

6.2 Rozšíření pro rozsáhlejší obvody

Rozšíření pro rozsáhlejší obvody spočívalo ve vyhledání "'úzkých"' míst, které omezovala spuštění programu Atalanta-M. Tyto místa byla identifikována a implementován příslušných programový kód, který by tyto omezení eliminoval. Bohužel při testování jsem narazil na problémy. Problémy spočívaly v tom, že v programu byla vyhozena vyjímka, která značila nepřípustný zápis do paměti. Najití příčiny problému nebylo možné najít v rozumném časovém horizontu, protože větší obvody jsou zpracovávány neoptimalizovaným programem až **40 minut** než dojde k dané chybě. Spuštění neoptimalizovaného programového kódu je nutné kvůli debugeru, aby bylo možné ladit programový kód. Pokud se program přeloží s optimalizacemi a spustí se debuger, tak je možné krokovat kód, ale pouze v jazyku assembler. Aby byla zachována funkčnost programu tak, byly vráceny konstanty omezující funkčnost, ale dynamická pole zůstala. Z toho vyplývá, že program je připraven na odstranění těchto limit, ale tyto limity odstraněny nebyly. Tento cíl nebyl úplně splněn a bude třeba strávit další čas na odladění tohoto problému.

6.3 Vytvoření rozhraní pro volání jednotlivých funkcí

Bylo vytvořeno rozhraní pro volání jednotlivých funkcí. Metody, které by měly být přístupné jsou definovány jako public. Ostatní metody jsou definovány buď jako protected nebo private a vně knihovny nejsou proto použitelné.

6.4 Vytvoření dynamicky linkované knihovny

Programový kód byl upraven tak, aby bylo možné ze zdrojových kódů zkompilovat dynamickou knihovnu, která bude využitelná i v jiných aplikacích. Abych umožnil plné využití programových funkcí nástroje ATPG a simulátoru, bylo nutné upravit rozhraní mezi knihovnou a ostatními aplikacemi. Byly vytvořeny potřebné metody pro nastavení parametrů. Tyto metody ve výsledné knihovně nejsou definovány jako public, protože byla implementována třída Params, která obsahuje všechny nastavitelné parametry. Byla definována metoda setParams, která parametry nastaví pro vlastní algoritmy. Třída Params obsahuje i ověření platnosti nastavovaných parametrů. Pro výměnu většího objemu dat byly definovány datové proudy, které přesun většího počtu dat umožňují a jejich abstrakce umožňuje na vstup nebo výstup připojit jak diskový soubor, tak i nějaké jiné zařízení nebo programovou aplikaci, bez nutnosti změny zdrojového

programu. Myslím si, že tento cíl byl splněn. V adresáři doc je navíc html nápověda, která obsahuje popis rozhraní knihovny.

6.5 Ověření funkčnosti

Ověření proběhlo pomocí testů na reálných obvodech, kde byly porovnávány vygenerované soubory. Blíže jsou výsledky popsány v kapitole **??**. Vygenerovné soubory byly shodné. Tím došlo k ověření funkčnosti nově vygenerovaného programu.

6.6 Srovnání rychlosti

Nakonec bylo provedeno srovnání rychlosti. Nově vytvořený program je o něco málo pomalejší, ale pouze za předpokladu, že jsou při kompilaci nastaveny optimalizace generovaného kódu.

6.7 Zhodnocení

Všechny cíle kromě jednoho byly splněny. Program bude třeba ještě upravit, aby se odstranily limity pro malé obvody. Variabilita programu však velice stoupla, protože byl převeden na dynamickou knihovnu. Po odladění knihovny pro velké soubory je možné pokračovat v integraci této knihovny do ostatních aplikací typu CAD nebo CAM, aby se docílilo automatizace a tím i zrychlení výroby.

7 Literatura

[1] C++ reference. `http://www.cplusplus.com/reference`.

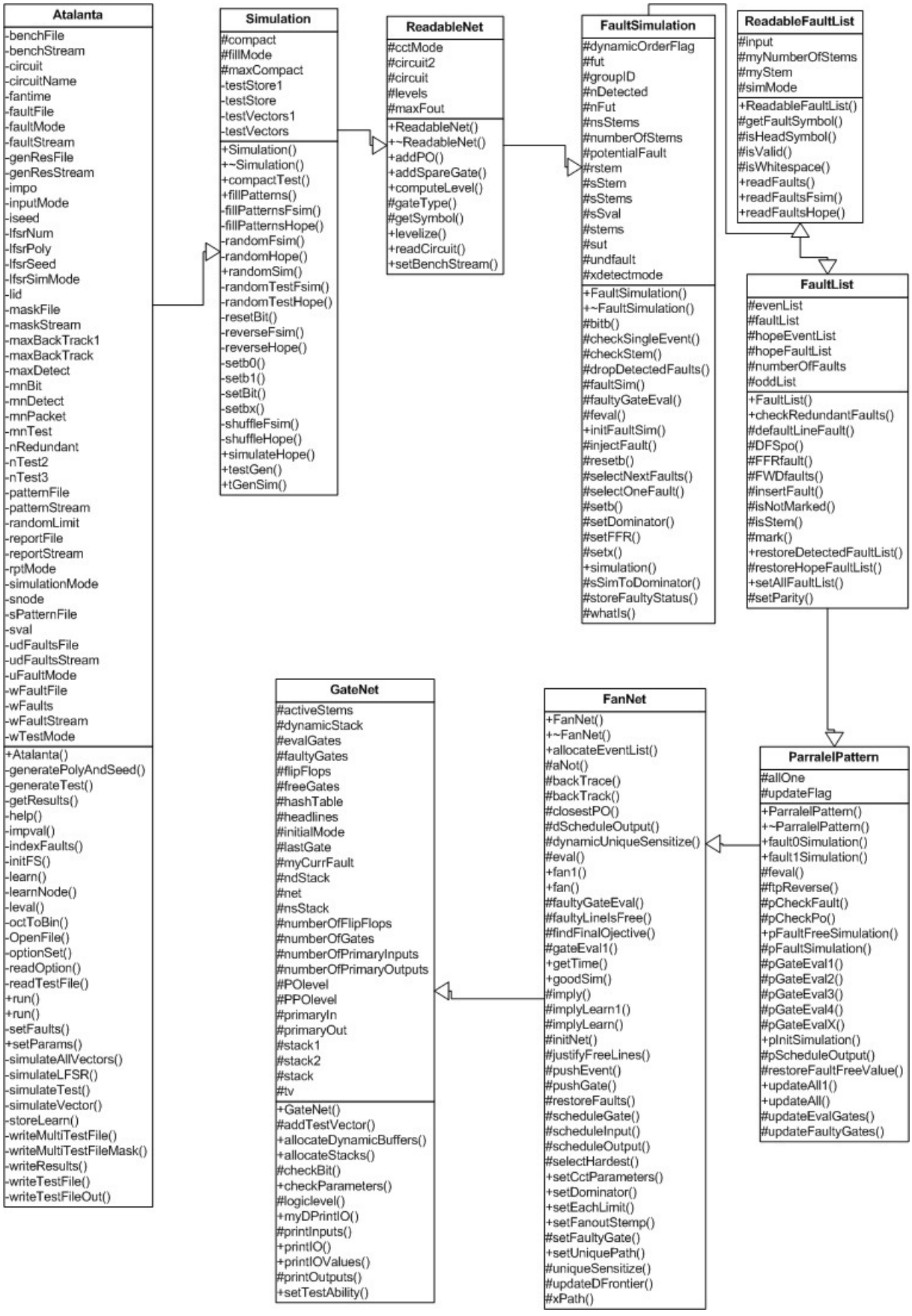
[2] P. Fišer. Atalanta-m 1.1b manual, 2005.
`http://service.felk.cvut.cz/vlsi/prj/Atalanta-M/man.html`.

[3] H. Jan. *Diagnostika a spolehlivost*. Vydavatelství ČVUT, Zikova 4, Praha 6, CZ, 2nd edition, 1998.

8 Přílohy

Metoda	Popis metody
void SetBenchStream(std::streambuf *buf)	Nastavení vstupního streamu
void SetFaultStream(std::streambuf *buf)	Nastaví vstupní stream fault listu
void SetProcFaultStream(std::streambuf *buf)	Nastaví stream pro zpracovaný seznam poruch
void SetTestPatternStream(std::streambuf *buf)	Nastaví stream pro testovací vzorky
void SetAbortedStream(std::streambuf *buf)	Nastaví stream pro přerušené nebo nedetekované poruchy
void PrintRedundantFaults(bool printRedundant)	Nastaví přepínač tisku všech identifikovaných redundatních poruch do souboru nastaveného pomocí SetAbortedStream.
void SetFaultMask(std::streambuf *buf)	Nastaví stream pro poruchové masky
void SetReportFile(std::streambuf *buf)	Nastaví stream pro report ATPG/FS
void SetOutputFileFormat(int fileFormat)	Nastaví formát výstupního souboru.
void SetDeriveAllPatterns(bool deriveAllPat)	Nastaví přepínač generování všech vzorků pro každou poruchu.
void SetNumPatterns(int numPatterns)	Nastaví počet generovaných testovacích vzorků pro každou poruchu.
void SetNumFanBacktrack(int numBacktrack)	Nastaví maximální počet návratů pro FAN algoritmus fáze 1
void SetNumFanBacktrack2(int numBacktreack)	Nastaví maximální počet návratů pro FAN algoritmus fáze 2
void SetHopeSimulation(bool hopeSim)	Nastaví užití simulace pomocí HOPE a třístavové logiky
void SetStaticLearn(bool staticLearn)	Nastaví přepínač statického učení
void SetLimitCompactShuf(int numShuffles)	Nastaví počet ???
void SetNoTestCompaction(bool noCompaction)	Nastaví přepínač užití ???
void SetRPT(int rpt)	Nastaví Random Pattern Testing
void SetRPTSeed(int seed)	Nastaví sekvenci náhodného generování
void SetTestPatternFEFault(bool testFEFault)	Nastaví přepínač generování testovacího vzorku pro každou poruchu
void SetStatusUnspecInputs(char status)	Nastaví stav nespecifikovaných vstupů
void SetSimulationMode(bool simulation)	Nastaví užití simulačního módu
void SetLFSR(char *poly, char* seed, int num)	Nastaví vektor LFSR
void SetParameters(Parameters param)	Nastavení parametrů všech parametrů

Tabulka 8.1: Nové metody rozhraní knihovny



Obrázek 8.1: UML Diagram