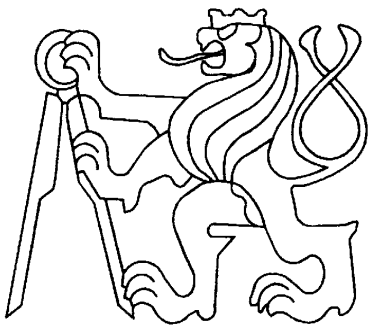
ČVUT PRAHA

Fakulta elektrotechnická



Diplomová práce

2005 Kamil Staufčík

== Zadávací formulář ==

Téma: Automatické generování testovacích vektorů pro kombinační číslicové obvody

Combinational ATPG - Automatic Test Pattern Generator

ATPG - Automatické generování testovacích vektorů pro kombinační číslicové obvody.

Úkolem je prověřit možnosti algoritmů pro ATPG, navrhnout a vytvořit software pro generování testovacích vektorů s důrazem na nestandardní postupy a možnosti speciálního nastavení průchodu.

Autor: Kamil Staufčík

Vedoucí: Ing. Petr Fišer

Anotace

ATPG - Automatické generování testovacích vzorů pro kombinační číslicové obvody. Požadavky pro testování vycházející z návrhu pro snadnou testovatelnost a průmyslovou výrobu číslicových obvodů; získání kompaktní sady testovacích vektorů pro daný obvod, která bude co nejmenší, s možností omezení pomocí vnějších podmínek pro vektory v postupně vytvářených sadách, bez ztráty jejich efektivity. Kromě těchto vlastností by mělo generování probíhat rychle a spolehlivě, s ohledem na znalosti topologie a funkčních jednotek v obvodu použitých. Generování používá heuristiky pro speciální redukci seznamu poruch. Hlavní algoritmus využívá postupného slučování podsad vstupních vektorů, s dalšími omezujícími podmínkami. V práci byly také použity netypické techniky pro průchod obvodem a vyzkoušeny nové postupy řešení problému.

Abstract

ATPG - Automatic Test Pattern Generation for Combinational Circuits. Design for testability and industrial production of circuits require compact vector test sets that are small in size, with adjustable properties, and efficient in testing. These are the main goals to be achieved in test pattern generation. Generation should also be done quickly, using the knowledge of the circuit topology and function of the units used. The proposed test generation method uses a heuristic for special reduction of the fault list. The basic algorithm uses partial sub-groups consolidation with limiting conditions. Also another non-typical approaches to circuit traversal were used in this work.

Prohlášení

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

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

V Praze dne ……. …….
podpis

Obsah

1. Úvod 5
1.1. Důvody testování obvodů 5
1.2. Základní pojmy 6
1.2.1. Typy poruch 6
1.2.2. Seznam poruch 6
1.2.3. Redukce seznamu poruch, dominance, ekvivalence 7
1.2.4. Generování testovacích vektorů 10
1.2.5. Další možnosti testovacích vektorů 11
1.2.6. Referenční ATPG Software 13
2. Rozbor a zpracování problému 14
2.1. Úvodní práce 14
2.2. Portabilita 14
2.3. Kompatibilita 15
2.4. Základní funkce 16
2.5. Změna struktury 16
2.6. Simulace 17
2.6.1. Injekce 17
2.6.2. Paralelně-sériová injekce 18
2.6.3. Sériově-paralelní injekce 18
2.6.4. Implementace simulace 18
2.7. Vytvoření seznamu poruch 19
2.8. Redukce v praxi 20
2.8.1. Redukce u ostatního SW 20
2.8.2. Implementace redukce 21
2.8.3. Srovnání pokrytí 22
2.8.4. Rozbor závislostí mezi poruchami 24
2.9. Generování vektorů 28
2.9.1. Počáteční algoritmus – jednoduchý průchod 29
2.9.2. Průchod s návraty – varianta 1 30
2.9.3. Průchod s návraty – varianta 2 31
2.9.4. Algoritmus s množinami vektorů 34
2.9.5. Implementace ATPG 36
3. Struktury 39
4. Závěr 42
5. Seznam použité literatury 43
6. Přílohy 44
6.1. Příklad souboru s poruchami 44
6.2. Příklad souboru s testovacími vektory 44
6.3. Příklad souboru se specifikací kombinačního obvodu 45
6.4. Skutečný formát souboru s vektory 46
6.5. Úplný test na c17 47
6.6. Grafické porovnání různých seznamů poruch 51
6.7. Seznamy poruch z dalších benchmarků 53
6.8. Parametry aplikace *muse* 53

1. Úvod

1.1. Důvody testování obvodů

Automatické generování testovacích vektorů je jeden z velmi potřebných oborů v testování číslicových obvodů. Vzhledem k rozsahům dnešních číslicových obvodů není snadné (a prakticky ani možné) vytvořit rozhraní, pomocí kterého by se daly otestovat všechny funkce obvodu. Cílem výrobců je vyrábět s co největším ziskem, tedy co nejrychleji a nejlevněji. To, jak bude číslicový obvod levný také závisí na tom, kolik jich vyrobíme správně funkčních. Odhalit poruchu součástky co nejdříve je vždy nejlevnější. Nejvíc stojí oprava již kompletní sestavy zařízení z těchto součástek sestavených. Pokud máme požadavek vyrábět co nejrychleji, bez vadných kusů se obejít nemůžeme. Přílišná pečlivost při výrobě by totiž zase cenu součástky zvýšila. Také nejen z ekonomického, ale i z technického hlediska nelze mít všechny součástky dokonalé. Nemůžeme odhlédnout od fluktuací parametrů výroby, které způsobují statistické odlišnosti součástek a s nenulovou pravděpodobností také jejich nefunkčnost. Nejdražší na výrobě obvodu je zapouzdření s vývody mimo čip. Proto je potřeba mít co nejméně výstupů, a to jde proti požadavku na testovatelnost obvodu. Tento nedostatek se v současnosti řeší tak, že se testují samostatně části obvodu, které lze logicky oddělit ještě před zapouzdřením, a poté pomocí vestavěných mechanismů pro testování obvodu – tzv. vestavěnou diagnostiku obvodu - BIST (Built-in Self Test) .

V obou případech potřebujeme, abychom funkce obvodu, nebo jeho části, otestovali co nejrychleji a s co nejmenšími nároky na další přidané prvky do obvodu. Oba tyto požadavky plynou z finančních důvodů při výrobě číslicových obvodů. Např. pokud budeme chtít vyrábět sto tisíc obvodů denně a máme pouze jedno testovací zařízení, je potřeba testovat jeden obvod za méně než 1 sekundu! Jen obvod se sto vstupy by pro kompletní otestování potřeboval zpracovat 2^{100} (zhruba 10^{30}) různých vstupních vektorů.

Při jednom testu za 1ns bychom testovali 40 bilionů let. Pokud bychom chtěli tento test zabudovat do obvodu, aby se mohl otestovat sám, museli bychom k němu navíc přidat paměť o velikosti $100*2^{100}$ bitů, tj. 15 miliard ExaByte. Nezbývá než konstatovat, že takovýto obvod pro všechny kombinace nikdy neotestujeme. To ovšem neznamená, že takovýto obvod nelze vyrobit a nebude správně fungovat. Právě naopak. Pokud vedle sebe položíme 100 vodičů, vytvořili jsme takovýto obvod, a zda funguje, otestujeme dvěma vektory[1]; nejprve současně na každý vodič přivedeme napětí s úrovní pro 0 a ve druhém kroku s úrovní pro 1.

Vidíme, že za určitých okolností je možné obvod otestovat velice rychle, zvláště pokud obsahuje nezávislé větve. Této i jiných topologických vlastností obvodů se dá s výhodou použít k rychlému generování vektorů, které pokryjí všechny detekovatelné poruchy v testovaném obvodu.

[1] Pozn.: Pro názornost v příkladu s vodiči neuvažujeme, že by došlo k tzv. přemostění, v tom případě by bylo testování složitější.

1.2.Základní pojmy

1.2.1. Typy poruch

V příkladu s vodiči jsme otestovali dvě základní poruchy, které nás u číslicových obvodů zajímají, trvalou jedničku a trvalou nulu. Porucha trvalá jednička – **Sa1** (Stuck-at 1) modeluje stav, ve kterém má vodič stále hodnotu, logickou 1, u trvalé nuly – **Sa0** (Stuck-at 0) naopak 0. V této práci budeme uvažovat pouze tyto poruchy, protože ostatní poruchy, které se v obvodu hledají, jako například pomalý náběh/sestup (Slow-to-Rise/Fall), zkraty (přemostění, Shorts), přerušení (Opens), ale i poruchy citlivé na vzorek (Pattern-Sensitive) nebo parametrické poruchy jsou ve svých projevech, při vhodném použití, shodné s poruchami Sa0 a Sa1. Vždy to samozřejmě neplatí. Některé poruchy lze otestovat pouze speciálními mechanismy, nebo za jiných podmínek. Trvalé poruchy nejsou tedy univerzálním nahrazením libovolné poruchy číslicového obvodu, ale jedná se o model, který je pro nás velmi snadno představitelný, se kterým se relativně dobře pracuje a který nám pro naše účely vyhovuje.

1.2.2. Seznam poruch

Vygenerování kompletní sady testovacích vektorů má několik fází. První z nich je vygenerování seznamu poruch, tedy všech možných poruch, které se v obvodu mohou objevit. Prakticky je seznam poruch identický s počtem vodičů, které se v obvodu vyskytují, vynásobeným dvěma. Pokud se vodič větví, je třeba také započítat dvakrát všechny jeho větve. Dvakrát více poruch je právě proto, že nás zajímají na každém vodiči již zmíněné dvě trvalé poruchy, Sa0 a Sa1. Jiné poruchy nás nezajímají, protože jejich projevy lze zjistit i jen pomocí těchto dvou poruch vodičů (např. i poruchu hradla, kdy je poškozeno tzv. přerušením jeho vnitřních spojů tranzistorů a není schopno správně měnit svou hodnotu, lze také zjistit testem na trvalou 0 nebo 1 na vodiči, který je na jeho výstupu). Seznam poruch, který získáme, slouží jako základní seznam toho, co máme zkontrolovat, a který si můžeme průběžně upravovat podle potřeb. Z něj vybíráme postupně jednotlivé poruchy ke kterým generujeme testovací vektory, poruchy které jsme již našli, nebo které jsou pro nás aktuálně něčím zajímavé, nebo naopak rušíme již nepotřebné poruchy, které jsme již odhalili jiným způsobem.

1.2.3. Redukce seznamu poruch, dominance, ekvivalence

V další fázi ATPG se provádí takzvaná redukce seznamu poruch. Redukce je vyjmutí poruch, které jsou v seznamu redundantní. Jejich redundance může být způsobena několika fakty; buď je porucha pokryta jinou poruchou, je ekvivalentní s dalšími poruchami, nebo je zřejmé, že se jedná o nedetekovatelnou poruchu a nemá tedy smysl se jí zabývat jinak, než jako hodnotou pro statistiku poruch obvodu. To, že porucha je pokryta jinou poruchou, označujeme termínem **dominance**. Přesnější definice dominance může znít takto:

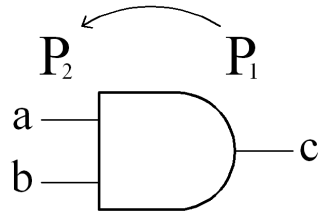
Porucha P1 dominuje P2 právě tehdy, když

- Každá sada vektorů které detekují P1, obsahuje sadu detekující P2
- P2 je detekovatelná (sada detekující P2 je neprázdná)

Poruše P2 říkáme že je dominovaná poruchou P1.

Dominanci rozlišujeme dvojího druhu: „u hradla“ a „u větvení“.

Dominovaná porucha u hradla (*Obr. 1*)je vlastně porucha na **citlivé cestě** (viz kap. 1.2.4), která by změnou hodnoty vodiče, na kterém se tato porucha nachází, vyvolala změnu na dalším vodiči, který je v této cestě bezprostředně za ní následující, tedy na výstupu z hradla. Pokud tedy budeme testovat dominovanou poruchu na vstupním vodiči, otestujeme současně i příslušnou poruchu na vodiči výstupním, tedy dominujícím To vyplývá přímo z definice, neboť sada vektorů, kterou otestujeme poruchu na vstupním vodiči, je vždy obsažena v sadě detekující poruchu na výstupním vodiči hradla.



Obr. 1

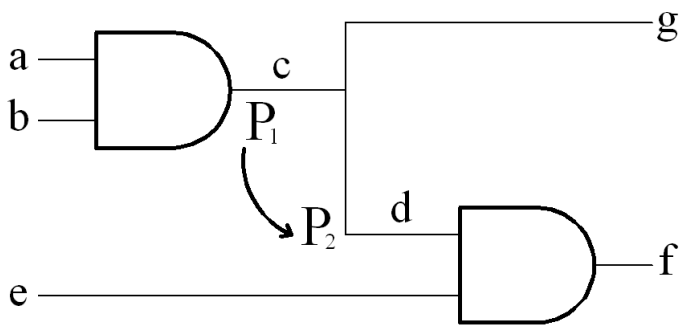
Obrázek ilustruje dominanci poruch u hradla AND. Porucha P1 dominuje poruchu P2. Obě poruchy jsou v tomto případě Sa1. Testovací vektory pro P1 jsou (0,1) a (1,0), pro P2 je to (0,1) – v pořadí (a,b). Pro poruchu Sa0 se jedná o ekvivalenci, viz níže.

Typ poruchy na výstupu (Sa0 nebo Sa1), bude záviset na tom, na jakou hodnotu by se změnila hodnota vodiče v případě uplatnění dominované poruchy. Zjednodušeně řečeno, dominovanou porucha nelze otestovat aniž by se otestovala dominující.

U větvení platí, že porucha před větvením dominuje poruchu stejné hodnoty za větvením (viz *Obr. 2*). Typ dominované poruchy je vždy stejný jako typ poruchy dominující. Větvení je velmi názorné pro pochopení dominance. Je velmi snadné si uvědomit, že pokud na některou z větví, které vedou z větvícího se vodiče

potřebujeme přivést logickou 0 nebo 1, musíme nutně tuto hodnotu nastavit i před větvením. Pokud nám tato hodnota ověří příslušnou poruchu Sa1 nebo Sa0, na sledované větvi, je zřejmé, že pokud by byla porucha před větvením, zjistili bychom ji také.

Vztahy dominovaná-dominující obvykle uvažujeme pouze jedním směrem. U větvení je to směrem od primárních výstupů k primárním vstupům, u hradel naopak.



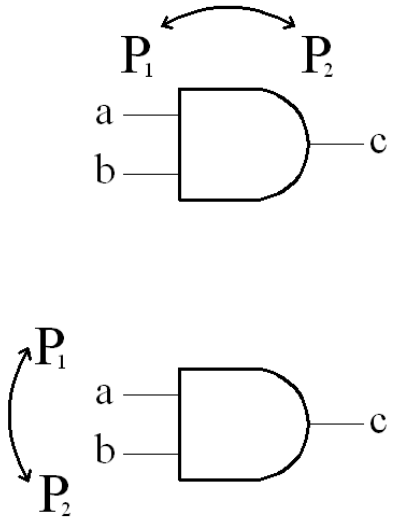
Obr. 2

Obrázek ilustruje dominanci u větvení vodiče ***c****. P1 dominuje P2 v tomto případě pro Sa0 i Sa1. Poruchu P1 Sa0 detekují vektory (a,b,e): (1,1,X) a (1,1,1). Poruchu P2 Sa0 vektor (1,1,1). Poruchu P1 Sa0 detekují vektory (a,b,e) : (0,X,X), (X,0,X), (0,X,1) a (X,0,1). Poruchu P2 Sa0 vektory (X,0,1) a (0,X,1). Pokud citlivá cesta prochází na vodič* ***g****, na vstupu* ***e*** *nezáleží, to ovšem neznamená, že druhá varianta, kdy musí mít hodnotu 1, je méněcenná, nebo ji můžeme zanedbat.*

U hradel s jedním vstupem i výstupem (většinou pouze hradla NOT, logická negace, i když počítáme i s jednovstupovými AND a OR, které odpovídají identitě) by se dalo říci, že platí relace oboustranně. V takovém případě ale mluvíme o ekvivalenci poruch. Ekvivalenci definujeme takto:

Porucha P1 je ekvivalentní s P2 právě tehdy, když mají shodné sady vektorů, které je detekují.

Podobně jako u dominance si můžeme ekvivalenci zjednodušeně představit také tak, že hradlo, kterým prochází citlivá cesta, má na všech vstupech stejnou hodnotu. U hradel, které uvažujeme, jsou všechny vstupy ekvivalentní a pokud je hradlo zcitlivěno, detekujeme všechny poruchy se stejnou hodnotou, opačnou vůči citlivé hodnotě na vodičích. Kromě toho detekujeme současně i příslušnou poruchu na výstupu.

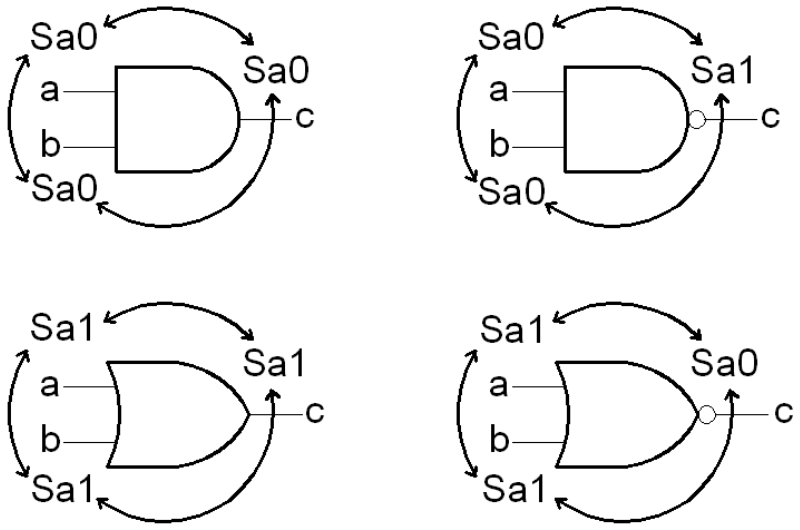


Obr. 3

Obrázek ilustruje dvě ekvivalentní dvojice u hradla AND. P1 a P2 jsou Sa0. Sady vektorů detekujících Sa0 na všech vodičích tvoří jeden vektor: (1,1).

Jako předpoklad byla stejná hodnota na všech vstupních vodičích, to ale také znamená, že tuto poruchu nelze detekovat jinak než tímto způsobem. Detekujeme tedy současně jednu hodnotu na každém vodiči vstupujícím a vystupujícím z hradla. Všechny tyto poruchy jsou tedy v tomto kontextu naprosto ekvivalentní, a stačí uvažovat pro detekci pouze jednu libovolnou z nich, protože nastavení vstupů hradla, při kterém to bude možné, je pouze jedno a vždy detekuje všechny ostatní zmíněné poruchy. To je opět v souladu s definicí, neboť pokud všechny poruchy detekujeme při jediném nastavení vstupů hradla, jsou sady testovacích vektorů pro všechny tyto poruchy shodné. Souboru poruch, které jsou navzájem ekvivalentní také říkáme třída ekvivalence.

Tuto třídu netvoří vždy pouze poruchy na jednom hradle. Porucha na výstupu hradla může totiž být současně vstupní poruchou na vstupu dalšího hradla, ve kterém je opět ekvivalentní.Při podrobnějším zpracování jsem zjistil, že velice výhodnou je také kombinace dominance-ekvivalence. Pokud je v obvodu ekvivalentní třída, ve které jedna z poruch dominuje poruše, která v této třídě není, dominuje této poruše libovolná porucha z této třídy. Současně platí, že libovolný vektor ze sady, která detekuje dominovanou poruchu, detekuje všechny poruchy v dominující třídě. Naopak pokud jedné poruše ze třídy dominuje nějaká porucha mimo ni, dominuje tato porucha libovolné z ekvivalentních poruch.Více ale v kapitole 2.8. Pomocí dominance a ekvivalence odstraníme tedy ze seznamu ty poruchy, které v něm jsou z našeho hlediska navíc.



Obr. 4

Obrázek ilustruje třídy ekvivalencí čtyř základních typů hradel. Pro všechny tři zobrazené ekvivalentní poruchy hradel AND a NAND je testovací vektor (a,b) roven (1,1), pro OR a NOR je to (0,0).

Jednotlivé poruchy které nám zbyly, je potřeba prověřit již detailněji.

1.2.4. Generování testovacích vektorů

K tomu slouží vlastní fáze generování vektorů. Každá porucha, kterou testujeme, musí mít možnost se projevit na výstupních vodičích. Pro tento fakt se používají termíny zcitlivění poruchy a cesty. Pokud chceme odhalit, že na vodiči je trvale hodnota 0, musíme na něj přivést hodnotu opačnou. To ale nestačí - pokud by tato hodnota byla po cestě na výstup blokována, nezáleželo by na tom, jestli má ve skutečnosti hodnotu 0 nebo 1. Cesta, na které se promítne změna hodnoty vodiče od něj až na nějaký primární výstup se nazývá **citlivá cesta**. Postup, při kterém zajišťujeme, aby hodnoty vodičů, které vedou do hradel v citlivé cestě byly takové, že nebudou bránit změně na cestě a ta se bude správně projevovat až po příslušné výstupy, se nazývá **zcitlivění cesty**, nebo **propagace**. Nastavení primárních vstupů obvodu tak, aby měl testovaný vodič požadovanou hodnotu se nazývá též **excitace poruchy**.

Při vytváření citlivé cesty může také nastat situace, kdy tato vede přes větvení a její rozvětvené vodiče se opět s pomocí nějakého hradla spojují. To může nastat i po delší „době," tedy po odděleném průchodu cest několika dalšími hradly. Tyto různé cesty, které se rozdělí a opět spojí, mohou být vzhledem k topologii obvodu obě citlivé současně i zvlášť. Pokud dojde k současnému zcitlivění, hovoříme o **rekonvergenci**, nebo rekonvergenci citlivé cesty. Ta bývá hlavním

zdrojem potíží u testování kombinačních obvodů. Tím, že jsou obě cesty citlivé, se nemusí změna před větvením projevit. To záleží na tom jakého je typu hradlo, které je spojuje. U hradel AND,NAND,OR a NOR nesmí být vstupující vodiče zcitlivěny na opačnou hodnotu. Pokud by se vyskytla porucha před větvením, tyto hodnoty by se pouze „přehodily," a hodnota na výstupu hradla by se nezměnila. Výjimku tvoří hradlo XOR, to nesmí mít citlivé vstupy ani se stejnou hodnotou. Existuje tedy i výjimka z pravidla, které říká, že hradlo XOR je vždy citlivé. To platí pouze na jeho jednotlivých vstupech, ale nikoliv na rekonvergentní citlivé cestě.

Jakmile si vytýčíme cestu po které chceme poruchu šířit, je potřeba zjistit, jaké musí být pro její zcitlivění primární vstupy obvodu, tedy vytvořit citlivou cestu. Po nalezení všech nezbytných vstupních hodnot obvodu jsme již zjistili jeden testovací vektor. Skládá se právě z těchto hodnot a z hodnot, na kterých nezáleží, které se běžně označují X nebo DC z anglického Don't Care. Nalezený vektor je také potřeba vyzkoušet, jednak jestli je funkční, tedy jestli je schopen detekovat požadovanou poruchu, ale také je dobré vědět, které další případné poruchy jím můžeme zjistit. K tomu slouží tzv. **simulace poruch,** o které se více dozvíte v kapitole 2.6.

Jestliže vyžadujeme nalezení všech testovacích vektorů pro danou poruchu, musíme projít všechny možnosti, kdy je možno se při propagaci cesty nebo při excitaci poruchy rozhodovat. Velmi mnoho možností musíme projít také i v případě, kdy se nějaké z těchto rozhodnutí ukáže jako nesprávné a nezbývá než se vracet a vyzkoušet jiné. Lze dokázat, že problém nalezení testovacího vektoru je NP-úplný [Gar79].

Číslicový obvod jako schéma můžeme považovat v určitém zjednodušení jako orientovaný graf. Kombinační obvod je navíc grafem acyklickým. Problém nalezení testovacího vektoru lze také převést na problém Splnitelnosti booleovského výrazu – SAT (Satisfiability of Boolean Formula).Tuto možnost využívají ATPG založené na splnitelnosti, tzv. SAT-based ATPGs, které se také stávají velice perspektivní; v této práci se jimi však zabývat nebudeme. [v9]

1.2.5. Další možnosti testovacích vektorů

Je tedy zřejmé, že podobně jako nelze otestovat všechny kombinace hodnot vstupů obvodu, není prakticky možné pro větší obvody vygenerovat pro danou poruchu všechny testovací vektory. To vnáší do celé problematiky otázky: Které vektory generovat? Které cesty používat pro zcitlivění? Tyto otázky lze pojmout také z jiného hlediska, a to tak, že bychom vlastně rádi uměli generovat vektory s jistými pro nás vhodnými parametry, a které by se tudíž mohly šířit po námi preferovaných cestách.

Existuje mnoho oborů, pro které je vytvoření testovacích vektorů „na míru“ velmi žádoucí. Jedná se především o návrhy s vestavěnou diagnostikou obvodu - BIST, kde se využívá různých vlastností vektorů pro jejich kompresi neboli Test Pattern Compression. Jedním ze zajímavých projektů je COMPAS [Com03], který využívá překryvů ve vektorech. Ty se totiž na vstup obvodu přivádí často posuvnými registry a pokud je konec jedné sekvence shodný se začátkem nové, sníží počet posuvů, které je potřeba provést pro změnu testovacího vektoru na vstupu testovacího obvodu. Současně se také ušetří paměť obsahující testovací vektory. Jako další využití vektorů s požadovanými parametry lze uvést LFSR - Linear Feedback Shift Registers (LZPR - Lineární Zpětnovazebné Posuvné Registry), pomocí kterého se v BIST generují sekvence testovacích vektorů.

Jeho výhodou je opět zmenšení nároků na paměť pro vektory. LFSR představuje automat, který mění cyklicky zadaný vstupní vektor. V několika málo krocích lze dostat vektory, které otestují mnoho poruch obvodu. Návrh LFSR také může usnadnit generování vektorů, které mají na určitých vstupech námi požadované hodnoty.

Také stojí za zmínku jeden typ testování, který se zdárně rozvíjí, tzv. Low-Power testing.Jedná se o šetrnější přístup k hardware při testování obvodů. Postupně aplikované testovací vzorky jsou totiž v mnoha případech schopny měnit současně takové množství hodnot na hradlech, že překračují několikanásobně běžné podmínky a dochází k nadměrné teplotní zátěži na čipech. V těchto případech je nutné paradoxně testovat kvůli přehřívání pomaleji, což je zřejmě nevýhodné. Kromě toho nemůžeme testovat poruchy zpoždění, které vyžadují naopak rychlé testování. Řešení tohoto problému je opět vytvářet takové sady vektorů, které nemění současně příliš mnoho hodnot, tedy například můžeme jisté procento vstupů po nějakou dobu zafixovat.

Ve standardním přístupu k ATPG se dále ze všech úspěšně vygenerovaných vektorů provede jakési zahuštění, nazývané **kompakce**. Kompakce sloučí vektory, které mají stejné hodnoty nebo na místě, kde se liší, v některém z vektorů na dané hodnotě nezáleží. Vznikne nám vektor, který otestuje současně obě slučované poruchy. Protože nám jde o minimální sadu testů, která pokryje všechny poruchy obvodu, je potřeba vzít pro každou poruchu pouze jeden vektor, který nejvíce vyhovuje. Nejvíce bude vyhovovat ten, který by se dal sloučit současně s co nejvíce vektory jiných poruch, optimálně ale tak, aby výsledná sada byla minimální. Speciální strategii vyžadují právě zmíněné návrhy pro vestavěnou diagnostiku (BIST) a nástroje pro automatické testování (ATE).

K tomu, abychom mohli sami určovat, které vektory se nám lépe hodí, je implicitní kompakce spíše nevýhodou. Je lepší mít větší počet vektorů s co nejvíc neurčenými vstupy. Další nevýhodnou strategií, která se v současném ATPG používá k velkému zrychlení generování, je fáze náhodného generování vektorů. Její výhoda je, že náhodné vektory jsou schopny rychle pokrýt velkou část seznamu poruch, takže v dalších fázích se hledají pouze zbylé nepokryté poruchy. Tento postup je ovšem za cenu toho, že vznikají vektory zcela bez kontroly a většinou bez

DC hodnot, a tak i se špatnou možností vlastní kompakce. V této práci se jí nebudu věnovat. Mohla by se ale dále využít např. pro kontrolované generování vektorů.

1.2.6. Referenční ATPG Software

V současné době existuje pouze několik kvalitních aplikací, které umí generovat vhodné testovací vektory ze zadaného obvodu. Zmínit je vhodné profesionální nástroje FlexTest od Mentor Graphics nebo TetraMAX od firmy Synopsys. Tyto a jim podobné nástroje jsou ovšem chráněny autorskými právy a není možné se blíže seznámit s principy, na kterých pracují, ani jinak ovlivnit výstup, který produkují. Již v úvodním semináři k diplomové práci jsem se seznámil se systémy Atalanta [LH93] a Milef [Gla92], které byly velmi vhodné právě k tomuto účelu. Systém Atalanta pochází od Prof. Dong S. Ha a H. K. Lee z Technické univerzity ve Virginii. Je vyvíjen od roku 1990, poslední větší úpravy byly provedeny v roce 1997. Systém Milef (neboli Mixed Level Automatic Test Pattern Generation) byl navržen v roce 1992 v Německu U. Gläserem a H.T. Vierhausem. Tyto SW jsou důležité hned z dvojího pohledu: jsou považovány za výkonné nástroje, které velice rychle a efektivně řeší problémy generování testovacích vzorů a tudíž je velice užitečné seznámit se s algoritmy, které používají, a navíc je možné je považovat také za jakýsi etalon, který dává (pokud se správně použije) "správné", přehledné a jednoduché výstupy.

Oba zmíněné softwary používají mnoho velmi sofistikovaných metod pro generování vektorů i jejich testování a kompakci. Schází jim však důmyslnější možnosti pro ovládání a nastavení, která nás více zajímají. Oba počítají také celou řadu různých statistik a dávají mnoho jiných informací o obvodech, které nejsou pro samotné generování testů zcela nezbytné. Dále byly při jejich testování zjištěny nedostatky zvláště týkající se zpracování některých DC vstupů a výstupů. Podrobnější rozbor a srovnání je v kapitole 2.9.5 a příloze 6.5.

2. Rozbor a zpracování problému

2.1. Úvodní práce

Seznámil jsem se se zdrojovým kódem software pro převádění formátů pro specifikaci číslicových obvodů od Vlastimila Kozáka (DP) [Koz05].Tento program umí načítat běžně používané typy souborů (např. bench). Mým úkolem bylo použít z něj právě tuto funkční část a zpracovat základní návrhy a myšlenky pro tvorbu vlastního software pro ATPG, vycházejícího z klasických i novějších přístupů a algoritmů které se používají pro generování testovacích vektorů (vzorů). Poskytnutý SW mi posloužil jako základ, na kterém jsem mohl vytvořit vlastní struktury pro generátor. Tím nebylo nutné řešit problematiku jako návrh lexikálního analyzátoru pro různé typy vstupních souborů, vytváření obecných struktur, studování možností daných formátů a podobně. Zejména užitečné bylo načítání souborů do velice důmyslně provázané struktury poskytující nejdůležitější obecné informace o obvodu.

Nastudoval jsem z různých poskytnutých zdrojů možnosti, které se týkají dané problematiky, generování testovacích vzorů pro číslicové obvody. Důležitými zdroji mi byly přednáškové materiály od prof. Chien Mo James Li [Ch04], popis a materiály pana Hyung K. Lee a Prof. Dong S. Ha [LH93] a prof. Robert K. Braytona . Dalším velmi podnětným zdrojem informací mi byl předmět Diagnostika a spolehlivost který vyučoval prof. Ing. Ondřej Novák, CSc. [Nov04], jakož i jeho elektronické materiály dostupné v tomto předmětu a také skripta Diagnostika a spolehlivost od prof. Ing. Jana Hlavičky, DrSc [Hla98]. Seznámil jsem se se zdrojovými kódy daných aplikací a pokusil jsem se navrhnout vlastní jednoduchý přístup řešení daného problému.

Pro ukázku a srovnání výstupů, rychlosti a efektivity ATPG jsem používal již zmíněný software Atlanta [LH93] a Milef [Gla92]. Pro testování jsem používal benchmarky ISCAS'85 [Brg85], na kterých jsem srovnával funkce těchto systémů.

2.2. Portabilita

Základní projekt, ze kterého jsem využil načítání struktury bench souborů byl již zmíněný Translator [Koz05]. Tento SW byl vytvořen pro specifické prostředí, konkrétně pro GUI, využívající základní WinAPI, tedy pro prostředí Windows. S vedoucím diplomové práce jsme se dohodli, že by tento projekt měl být více obecný, ve smyslu otevřenosti a přenositelnosti zdrojových kódů. Jako základní jazyk jsme určili C++. Co je to přesně C++ a jak různí tvůrci překladačů chápou standardy a nadstavby jsem se seznámil při pokusech zkompilovat Translator na různých X-Nixových a Linuxových systémech. Celkově byl projekt (kromě závislosti na konkrétním GUI) vyhovující standardu ANSI/ISO až na tyto nedostatky:

- nestandardní funkce stricmp pro porovnání řetězců bez ohledu na velikost písmen
- redefinice základních logických operací: not, and, or, xor
- ve třídě Objekt se používaly v rozporu se standardem vlastnosti Wire a Port před jejich definicí.

Všechny tyto nedostatky šlo vcelku lehce odstranit, obtížnější bylo před tím zjistit, jaké jsou v tomto případě standardy a proč byly porušeny. Další potíže vznikly při hledání různých operačních systémů, na kterých by šlo vyvíjené ATPG otestovat. Na naší fakultě jsem měl v době zpracování diplomové práce k dispozici systémy SunOS 5.9, Sillicon Graphics / IRIX 6.5.4, Windows 2000 a XP. Na těchto systémech je potřeba mít vývojové prostředí (nebo alespoň funkční kompilátor C++). Používal jsem prostředí založené na kompilátorech g++. Pod Unixu podobnými systémy je možné nainstalovat právě tento kompilátor a standardní C++ knihovny. Pod Windows existuje 32-bitové vývojové prostředí DEV-C++ a CygWin, používající také g++. Vzhledem ke změnám na systémech, které provozuje fakulta, nebylo možné testovat po celou dobu vývoje, a v současné době pod SunOs ani IRIX nejsou zprovozněny požadované vývojové nástroje. Aplikaci jsem dále proto testoval pouze na systémech RedHat 7.3 a 8, a pod Windows 2000 a Windows XP, ve zmíněných vývojových prostředích.

2.3. Kompatibilita

Vyvíjená aplikace byla srovnávána převážně se systémem Atalanta, který používá jako vstupní soubory ve formátu ISCAS'85 [Brg85], nebo '89 [Brg89]. V této práci si vystačíme s jednoduchým formátem souboru ISCAS'85 (přesněji formát .bench, popř. full-scan verze sekvenčních obvodů s příponou.scan), protože nás zajímají pouze kombinační obvody ISCAS'85. Proto jsem také naimplementoval podporu tohoto formátu, a to s využitím aplikace Translator [Koz05]. Protože tento software podporuje také jiné formáty, bylo by možné v případě potřeby z něj využít i dalších formátů, které umí načítat. Atalanta umožňuje dále načítání seznamu poruch ze souboru ve svém vlastním formátu, který využívá také simulátor HOPE, který je součástí systému Atalanta. Simulátor HOPE umí načítat zase soubory, které generuje Atalanta jako výstup při vytváření testovacích vektorů. Pro porovnání těchto systémů s aplikací, kterou jsem vyvíjel bylo dobré opět umět načíst tato data.

2.4.Základní funkce

Jako první úkol jsme s vedoucím diplomové práce vytýčili zvládnutí základních rysů generátoru, což jsou: **vytvořit seznam poruch**, který bude do budoucna vhodné zredukovat, tedy **vytvořit redukovaný seznam poruch**. Pro testování je výhodné také implementovat načítání seznamu poruch ze souboru. Ze seznamu se dále obvykle pokračuje **vygenerováním testovacího vektoru** pro každou poruchu. Abychom mohli zkontrolovat, které všechny poruchy testovací vektor pokrývá, neobejdeme se bez **simulátoru** poruch. Pro jednoduchost jsem zvolil právě simulaci pomocí injekce poruch (viz 2.6.1), tedy nastavení obvodu testovacím vektorem a porovnávání správných výstupů hradel s výstupy, kdy postupně vnucujeme poruchu příslušným vodičům odpovídajícím právě kontrolované poruše. Dalším rozšířením bylo také načítání testovacích vektorů ze souboru. Generování vektorů nebo jejich prosté načtení ze souboru jsou dva přístupy, které mohou být velmi různé. Implementoval jsem dvě možnosti jak lze při generování postupovat.

Při zpracování těchto základních rysů jsem přicházel na různé typické i méně běžné až specifické problémy tohoto úkolu. Dále se budu snažil je stručně vysvětlit a objasnit, proč jsem se rozhodl řešit je daným způsobem.

2.5.Změna struktury

Po prostudování SW Translatoru pro převod různých formátů specifikací logických obvodů [Koz05] jsem si uvědomil nutnost zmenšit strukturu, do které se specifikace načítají. A to jak kvůli nárokům na paměť, tak i pro urychlení celého výpočtu. Strukturu jsem nenavrhl pevně s tím, že by již měla být trvale v dané podobě, ale ponechal jsem si prostor pro několik variant. A to z důvodu potřeb různých algoritmů, kterými lze tento problém řešit. Principy jsou mnohdy tak odlišné, že vyžadují naprosto jiný pohled na strukturu obvodu. Při návrhu jsem se v současné době vydal směrem pro paralelní zpracování více poruch při simulaci (viz kap.2.6) s použitím vícehodnotové logiky. Podobná situace je nejen u simulace, ale také i u generování testovacích vektorů, zcitlivění cest, propagace a popř. dále u injekce, obnovení hodnot atd. Jak již bylo zmíněno (viz kap. 2.1), obsahuje Translator velice důmyslné struktury, které jsou pro potřeby ATPG příliš komplexní. Struktury, které jsem navrhl, obsahují pouze nejnutnější informace o vzájemném propojení a větvení. Také neobsahují velice obecně řešený typ hradla, ke kterému bylo použito porovnání MStr řetězců, což je z hlediska rychlosti výpočtu výstupních hodnot hradel velmi nepraktické.

2.6. Simulace

Pro jednoduchost, ale také kvůli otestování správného načítání a změn struktury jsem začal svou práci vytvořením simulátoru poruch. Simulace funkce číslicového obvodu se v ATPG používá k tomu, abychom zjistili, zda je vygenerovaný testovací vektor schopen otestovat požadovanou poruchu, a také které jiné poruchy odhalí. Tento celý postup se označuje jako **simulace poruch**. Simulaci poruch lze provádět několika způsoby, nás bude zajímat především tzv. **injekce poruch**. Nejprve jsem vytvořil jednoduchou dvouhodnotovou logiku, kterou jsem poté rozšířil na tříhodnotovou (a pro jednu metodu generování vektorů pak i čtyřhodnotovou) logiku. Držel jsem se klasické injekce, ale ponechal jsem i možnost pro budoucí rozšíření na některý z dalších typů simulace, které by mohly v budoucnu celý proces generování testovacích vektorů urychlit.

2.6.1. Injekce

Při injekci se nejprve na primární vstupy přivede testovací vektor. Provede se kompletní nastavení obvodu odpovídající těmto vstupům až na primární výstupy. Poté se na vodič, na kterém je požadovaná porucha, nastaví hodnota logická 0 nebo 1, podle typu poruchy, tedy 0 pro Sa1 nebo 1 pro Sa0. Měla by být odlišná od hodnoty původní, jinak by bylo zřejmé, že je testovací vektor pro tuto poruchu nevhodný. Změna hodnoty na sledovaném vodiči by se měla propagovat alespoň na jeden primární výstup. Celá cesta od poruchového vodiče až na tento výstup (výstupy) je citlivá na zkoumanou poruchu, ale také na poruchy, které by vyvolaly stejnou změnu kdekoliv na citlivé cestě. Testovací vektor tedy odhalí nejen první poruchu, ale i tyto všechny ostatní a není třeba pro ně mít zvláštní testovací vektory. Toho se využívá tak, že se tyto poruchy rovnou odeberou ze seznamu poruch jako již detekované. Pro simulaci lze využít některých zjednodušení a zrychlení. Jednou z nich je paralelní zpracování struktury obvodu. Struktura může mít v tomto případě podobný formát jako při jednoduchém zpracování. Seznam poruch obsahuje všechny poruchy s příslušnými odkazy přímo na hradla a vstupy, a je proto možné provést přímou injekci.

Injekce se musí provádět pro všechny testovací vektory pro každou poruchu. Složitost je tedy O(PxHxV), kde P je počet poruch v seznamu, H počet hradel obvodu a V počet testovacích vektorů, které chceme aplikovat. Počet poruch bývá přímo úměrný počtu hradel, závislost je tedy pro jednu poruchu zhruba kvadratická vzhledem k počtu prvků obvodu. Pokud bychom měli pro každou poruchu jeden testovací vektor, celková složitost je odhadem kubická. Oproti složitosti vlastního generování vektorů je tato zanedbatelná, ale přesto je u simulace poruch pomocí injekce dobré použít metody pro zrychlení, neboť obě fáze jsou na sobě závislé.

2.6.2. Paralelně-sériová injekce

Tato metoda využívá zpracování více poruch současně při jednom testovacím vektoru. Většinou se využívá toho, že každý vodič má hodnotu, kterou lze zapsat jedním nebo dvěma bity. Současně lze tedy paralelně zpracovávat tolik poruch, kolik bitů má datový typ použitý pro výpočet. Pokud je délka slova *d*, je možné současně zpracovávat *d-1* poruch. Jedna zůstává rezervována pro hodnotu, kdy je obvod bez poruchy. Průchod je třeba provést celý, nelze jej ukončit dříve než najdeme všechny poruchy. U jednobitové reprezentace hodnot bez DC je mechanismus velmi snadný. Pokud uvažujeme také DC, musíme zavést aritmetiku vhodnou pro snadné výpočty hodnot. U některých operací nelze zavést jednoduché logické operace, což celý výpočet zpomalí.

2.6.3. Sériově-paralelní injekce

Metoda zpracovává postupně jednotlivé poruchy, ale umožňuje v obvodu současně aplikovat více testovacích vektorů najednou. Takto lze rychleji nalézt vektor, který danou poruchu detekuje. Pokud tuto informaci již ale máme, není tato metoda zvlášť výhodná. Výhodou je, že ihned po nalezení vektoru můžeme hledání přerušit. Opět je využito vícebitové reprezentace použitého datového typu a také je potřeba pečlivě postavit operace pro vícehodnotovou logiku. Deduktivní simulace poruch je podle [LH91] rychlejší než obě výše zmíněné metody. Bohužel toto platí pouze pro konkrétní hodnoty na vodičích, nikoliv pro neurčené hodnoty.

2.6.4. Implementace simulace

Při implementaci jsem použil pouze zmíněnou jednoduchou injekci, v průběhu prací nebylo potřeba urychlovat tuto část tvorby ATPG ,neboť jsem se zaměřil nejprve na vygenerování kompletních sad, nikoliv na postupné upravování seznamu vektorů v průběhu generování. Tuto část nebylo potřeba rozvíjet, pro naše účely bylo důležitější prozkoumat kompletní sady vektorů, optimalizované, velmi kompaktní sady umí efektivně generovat systém Atalanta. Abychom mohli porovnávat testovací sady vytvořené Atalantou a našimi algoritmy, bylo nutné vytvořit takový výstup, který by mohl načíst také systém Atalanta. V příloze 6.2 uvádím formát vektorů, jaký je zmíněn v dokumentaci k tomuto systému. Po vytvoření funkcí, které uměly tyto seznamy načíst jsem zjistil, že formát není Atalantou dodržen. Základní rozdíly uvádím opět v příloze (6.4). Jedná se především o řádky mimo komentář, a dále popis poruchy a texty před vlastním testovacím vektorem. To zkomplikovalo situaci při načítání vektorů. Implementaci všech tří možných nedostatků jsem se rozhodl neprovádět, a ponechat původní čistý formát. Ten jsem také použil dále pro načítání seznamu vektorů, které budou použity jako maska vstupních hodnot obvodu při generování testů.

2.7. Vytvoření seznamu poruch

V kap. 1.2.2 jsem uvedl, že na seznam poruch lze snadno pohlížet jako na seznam vodičů v obvodu, přičemž na každém vodiči sledujeme dvě poruchy, Sa0 a Sa1. Proto jsem také upravil strukturu obvodu tak, že v ní jsou zastoupeny všechny vodiče jako samostatné objekty. Seznam poruch se nestane ničím jiným, než právě zmíněným seznamem vodičů, který si vhodně zdvojíme. Pro reprezentaci seznamu jsem zvolil dynamický seznam s ukazateli dalšího prvku. Ve srovnání se strukturou virtuálních vodičů, které používá Atalanta, se jedná o triviální záležitost. Ve skutečnosti ale Atalanta používá mechanismy, které seznam poruch nahrazují, takže není potřeba aby klasický seznam obsahovala. Pro naše účely je ale seznam velmi důležitý, je ho potřeba pro načítání zadaných poruch ze souboru a porovnání s ostatními ATPG softwary.

V další kapitole se zaměříme na redukci, která je na seznamu poruch založena. To, že se staví k seznamu poruch autoři jiných SW tak odlišně, způsobuje velmi překvapivá zjištění při testování. Seznam poruch je také základem k tomu, abychom určili důležitý ukazatel – pokrytí poruch (fault coverage). Ten udává, kolik procent poruch z celkového počtu všech možných poruch v obvodu naše testy odhalí.

Při generování vektorů je dobré, když lze specifikovat, které z poruch chceme pokrýt. K tomu jsem naimplementoval načítání testovacích vektorů ze souboru. Popis souboru viz příloha 6.2. Vzhledem k jednoduchosti struktury seznamu poruch je načítání ze souboru pouze problém nalezení poruchy v seznamu podle jména, které načteme ze souboru, a její označení, že bude dále zpracována. Označení poruchy je velmi jednoduché, a proto nebyl problém dodržet formát, který používají aplikace Atalanta a HOPE. Jediný rozdíl ve výstupních formátech které generuje Atalanta a ve výstupu z mé aplikace je v identifikaci poruch na vodiči, který se nevětví. Atalanta takovýto vodič označuje pouze jedním jménem s udáním hodnoty poruchy, já jsem se rozhodl ponechat pro více informací i jméno vodiče na výstupu hradla, do kterého vodič s poruchou vstupuje.

2.8. Redukce v praxi

2.8.1. Redukce u ostatního SW

Přes očekávání jsme zjistili, že redukci poruch neprovádějí zmíněné programy Milef a Atalanta tak, jak by odpovídalo teoretickým pojednáním na toto téma. Obzvláště nás zarazily přebývající poruchy, které by bylo možné odstranit pomocí dominance. Po zevrubném prozkoumání obvodu, kde docházelo k těmto jevům jsme došli k názoru, že toto chování má příčinu v nedeterminismu, kterého se musí heuristika dopustit kvůli výběru ekvivalentních poruch.

Naskýtá se otázka, zda není možné jinou heuristikou dojít k vylepšení (a tedy i zmenšení) seznamu poruch pomocí vztahu ekvivalence a dominance (třeba jen ve zřejmých situacích). Bližším prozkoumáním závislostí jsem nezjistil, zda by skutečně v konkrétním případě došlo záměnou ekvivalentní poruchy k tomu, že by bylo třeba procházet znova celou strukturu. To ostatně odpovídá definici ekvivalence. Mým úkolem bylo dále prověřit experimentálně pokrytí poruch obvodů, ve kterých se provede rozsáhlejší redukce bez detailního rozboru závislostí, což by mohlo vést k chybám při generování, a tedy k menšímu výslednému pokrytí.

Základní obvod, ze kterého jsem vycházel, byl testovací c17 benchmark [Brg85], který obsahuje 17 vodičů, 6 hradel a trojí větvení (2 rekonvergentní). Benchmarky slouží k ověření a porovnání ATPG, ale i jiných nástrojů pracujících s číslicovými obvody. Tyto soubory by měly být prověřeny a mělo by být známo, čím je daný obvod možno otestovat, zredukovat poruchy a jak vypadají sady kompaktních testovacích vektorů a podobně. Vzhledem k tomu, že tyto soubory ale nikde v dokumentaci softwaru ani na webových stránkách nebyly podrobněji rozebrány, a že výstupy z programů nejsou, a ani nemohou být vždy stejné, bylo potřeba udělat detailnější rozbory manuálně. K tomu byl nejvhodnější právě zmíněný obvod c17. Vzhledem k 17 vodičům bylo potřeba projít 34 poruch a zjistit závislosti. Protože c17 je dostatečně malý obvod, který neobsahuje nedetekovatelné poruchy, mohl jsem srovnat vypočtené poruchy s tím, co produkovaly nástroje Atalanta a Milef. Překvapivé zjištění bylo, že ve zredukovaném seznamu poruch zůstaly i poruchy, které byly zřejmě dominovány. Ne sice všechny, ale přesto bylo zřejmé, že Atalanta neprodukuje minimální seznam poruch.

Z toho vycházela další práce, ve které jsem se blíže seznamoval se závislostmi mezi poruchami, ekvivalentními třídami a dominancemi. Zjistil jsem, že jednoduchým použitím upravených pravidel pro ekvivalenci a dominanci můžeme seznam poruch velmi zjednodušit. Toto zjednodušení je ale možné pouze za přijetí podmínek, které nelze zjistit jinak, než pomocí vlastního generování vektorů. To plyne přímo z definice dominance. Dominující a dominovaná porucha mají mít totiž neprázdný průnik sad testovacích vektorů. Pokud je potencionálně dominovaná porucha detekovatelná, je průnik zřejmě neprázdný, testovací vektor musí nutně být i v sadě dominující poruchy. Pokud ji detekovat nelze, není tato porucha dominovaná. Důvodem, proč ani jeden ze softwaru dominanci neuvažuje je ten, že

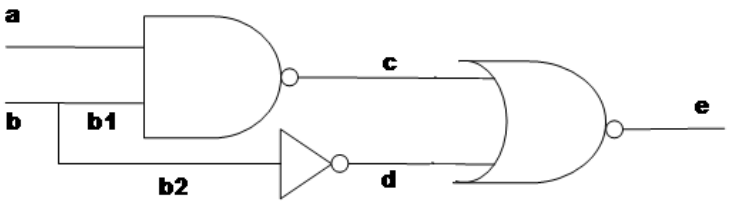
redukci seznamu poruch provádí ještě bez znalosti toho, zda poruchu detekovat lze nebo ne. Nedetekovatelná porucha u kombinačních obvodů má ale také svá pravidla výskytu. Je to právě rekonvergence citlivé cesty, která brání její detekci. Pokud se vyskytnou velké redukce v okolí větvení, je potřeba prozkoumat pečlivě možné závislosti. Když budeme striktně oddělovat redukci seznamu poruch od generování testovacích vektorů, není jiná možnost, než se smířit pouze s odstraněním ekvivalentních poruch a nedbat i na zcela zřejmé závislosti plynoucí z dominance. Pro vyzkoušení redukce v praxi bylo nutné již přistoupit k implementaci.

2.8.2. Implementace redukce

Při implementaci jsem využil toho, že struktura pro poruchu je velmi jednoduchá (viz kapitola 3), a rozšířil ji o ukazatele na ekvivalentní, dominující a dominované poruchy. S využitím znalosti závislostí popsaných v kapitole 2.8.4 jsem vytvořil jednoduchý algoritmus, který najde všechny poruchy, které s jistotou nelze z obvodu odstranit. Tento problém lze řešit také např. metodou pokrytí, která již ale má větší znalosti o vztahu mezi poruchami a je celkově více komplexní. Shrnutí jednoduchých závislostí lze vyjádřit takto:

Porucha je vždy buď ekvivalentní, dominovaná, nebo dominující. Ze seznamu poruch můžeme odebrat všechny dominující poruchy. Pokud je některá porucha ekvivalentní s libovolnou dominující poruchou, můžeme ji také odebrat. Zbylé poruchy jsou pouze dominované, a jejich sady testovacích vektorů jsou podmnožinou všech poruch, které jsme odstranili.

Při tomto radikálním odstranění se dopouštíme chyby, protože dříve než proběhne generování vektorů nevíme, zda lze pro všechny dominované poruchy které zbyly vygenerovat testovací vektor. Jinými slovy, zda je každá zbylá porucha detekovatelná. Jak již bylo zmíněno, pokusíme se zjistit, jaký má toto zanedbání vliv, a srovnat pokrytí sad vektorů vygenerovaných pro plný seznam poruch, redukovaný seznam poruch a pokrytí bez načítání externího seznamu poruch.



Obr. 5

Nedetekovatelné poruchy Sa1 na b1 a b2, které jsou dominovány b.

Na obrázku vidíme obvod s větvením na vstupu **b**. Větve **b1** a **b2** jsou dominovány pro poruchu Sa0 (i Sa1). Každá z nich by detekovala poruchu Sa1 na b. To je ale to, co ani jedna z nich nedokáže, protože jsou obě nedetekovatelné. Samotná Sa1 na **b** však detekovat lze, protože při změně hodnoty na **b** se změní hodnota v obou větvích, a změna projde i přes hradlo NOR, které by změnu pouze jedné větve blokovalo.

2.8.3. Srovnání pokrytí

Po implementaci navrženého algoritmu pro redukci (bez ohledu na detekovatelnost poruch) jsem srovnával pomocí simulátoru HOPE [LH96] (který je součástí systému Atlanta) můj redukovaný seznam poruch se seznamem z aplikace Atlanta. Redukovaný seznam pokrýval pro všechny kombinační benchmarky stejné procento detekovaných poruch v obvodu jako pokrýval seznam poruch, který generuje Atalanta, byl však zhruba 2x až 3x menší. Po podrobnějším prozkoumání jsem zjistil, že pokrytí, které mělo být s použitím redukovaného seznamu poruch menší než plný seznam, bylo způsobeno testovacími vektory, které vytváří Atalanta.

obvod	Plný FL			Redukovaný FL			Atalanta - D		
	čas [sec]	pokrytí [%]	poč. testů	čas [sec]	pokrytí [%]	poč. testů	čas [sec]	pokrytí [%]	poč. testů
c1355	4.6	97.122	2632	1.717	96.169	954	2.9	96.823	1524
c17	0	100	34	0	100	12	0	100	22
c1908	9.933	99.528	3798	3.167	99.026	1220	4.833	99.202	1864
c1and	0	100	6	0	100	3	0	100	4
c3540	51.867	96.299	6818	16.367	94.267	2154	25.417	95.916	3288
c432	0.267	98.148	848	0.083	98.142	317	0.15	98.855	518
c499	0.817	91.383	912	0.417	95	570	0.55	95.515	724
c5315	124.683	99.351	10561	43.767	98.305	3770	62.983	98.785	5285
c6288	431.15	80.121	10076	167.133	76.572	3677	266	80.837	6260
c7552	424.55	96.306	14546	154.05	94.239	4973	219.8	95.497	7210
c880	0.633	100	1760	0.217	100	609	0.35	100	942
c10-or	0	100	22	0	100	11	0	100	12
c3-and	0	100	6	0	100	3	0	100	4
c3-nand	0	100	6	0	100	3	0	100	4
c3-nor	0	100	6	0	100	3	0	100	4
c3-or	0	100	6	0	100	3	0	100	4
c3-xor	0	100	6	0	100	6	0	100	6
c4-and	0	100	8	0	100	4	0	100	5
c4-nand	0	100	8	0	100	4	0	100	5
c4-nor	0	100	8	0	100	4	0	100	5
c4-or	0	100	8	0	100	4	0	100	5
c4-xor	0	83.333	10	0	83.333	10	0	83.333	10
c5-xor	0	100	16	0	100	14	0	100	16

Tab. 6

Tabulka porovnání různých seznamů poruch systémem Atalanta vzhledem k vygenerovaným testovacím vektorům z těchto poruch.

Ta totiž i bez kompakce vytváří vektory, které jsou příliš moc definované – vstupy mají definovány hodnoty přestože by mohly mít hodnotu DC. Tím je způsobeno, že detekují i poruchy, které by méně definované vektory neodhalily. Vektory jsou tedy pro nás „příliš dobré“ – pro nás jsou výhodnější ty, které můžeme vlastními metodami dále dodefinovat.

Generování jsem tedy provedl jinak. Neověřoval jsem již dále pokrytí zvlášť pomocí simulátoru, ale ponechal jsem spočítat pokrytí pouze aplikaci Atalanta. Tabulka ukazuje pokrytí jednotlivých benchmarků třemi způsoby: Pokrytí plným seznamem poruch, Pokrytí redukovaným seznamem poruch a Pokrytí systémem Atalanta, který generoval pro každou poruchu 1 vektor, bez kompakce, s neurčenými hodnotami.

V tabulce *Tab. 6* jsou uvedeny dva typy testů, klasické ISCAS’85 a menší testovací obvody, které jsem vytvořil pouze pro základní ověření funkce ATPG. Pro ty byl čas generování zanedbatelný a pokrytí až na 1 výjimku vždy 100%. U těchto testů stojí za povšimnutí počty testovacích vektorů. Těch bylo kromě dvou případů vždy méně, než kolik jich potřebovala pro 100% pokrytí vygenerovat Atalanta. To platí bez výjimky pro všechny větší obvody. U těchto větších obvodů je kromě toho již nezanedbatelně menší čas pro vygenerování kompletní sady testů. U c6288 trvalo generování pro redukovaný seznam poruch 63% času než seznamu z Atalanty. Téměř 39% času který bylo potřeba pro úplný seznam poruch. Pokud si povšimneme pokrytí u větších obvodů, zjistíme, že redukovaný seznam poruch je většinou o méně než 1% horší, než seznam z Atalanty (kromě výjimečného c6288, který obsahuje velké procento nedetekovatelných poruch). To již odpovídá počtu nedetekovatelných poruch v obvodu, které byly dominovány některou z odstraněných poruch. V některých případech si můžeme povšimnout, že Atalanta nevygenerovala optimální testovací vektory, přestože byla spuštěna v režimu, ve kterém měla pokrýt všechny detekovatelné poruchy (např. c7552 a c1355). Plný FL pokrývá větší procento poruch. Naopak u některých případů (např. c432 a c499) je tomu naopak. Vypadá to, jako by úplný seznam poruch k detekci nestačil. Ve skutečnosti se nejspíš jedná opět o neschopnost nalézt testovací vektor pro každou poruchu.

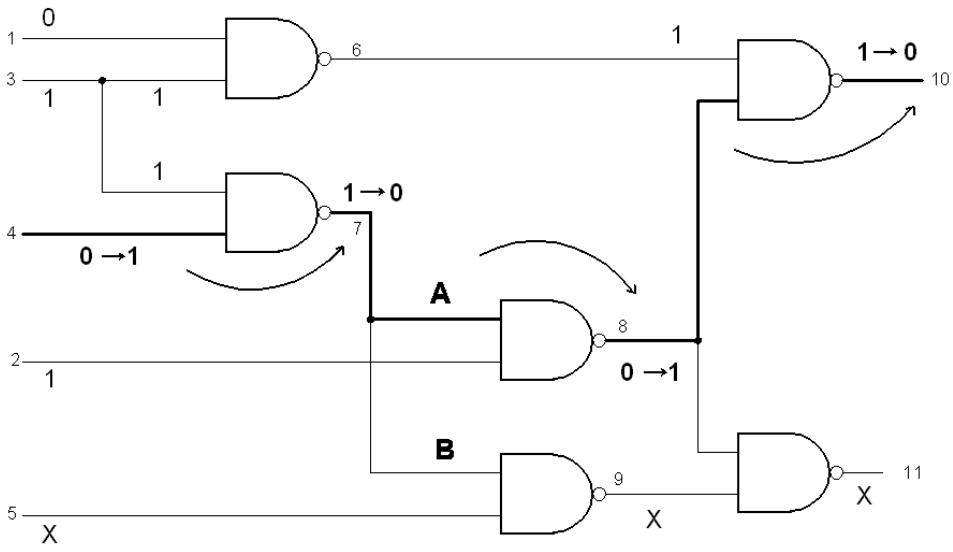
Pro vizuální srovnání tří uvedených způsobů uvádím v příloze grafy, viz kap. 6.6, které porovnávají časy, pokrytí a počty vektorů ještě s výstupem, který Atalanta vyprodukuje pokud se jí ponechá vlastní automatické řízení procesu generování testovacích vektorů. Jak již bylo zmíněno, v tomto případě dostaneme velmi malou a dobrou sadu pro otestování celého obvodu, bohužel i přes explicitní zákaz kompakce dostaneme sady již hodně kompaktní a s určenými hodnotami, což je pro naše účely nevýhodné.

Další testování jsem provedl na benchmarcích ISCAS’89, které byly upraveny ze sekvenčních obvodů na kombinační. Zvláště kvůli tomu, že je jejich škála je trochu pestřejší než u klasických kombinačních ISCAS, což se při testování také potvrdilo. Některé tyto benchmarky není schopen pojmout ani systém Atalanta, přestože jinak se choval velice šetrně s pamětí, začal se u obvodů s více jak 10 tisíci vodiči chovat nevhodně, a v některých případech byl ukončen nebo selhal. Tyto

benchmarky nejsou kvůli jejich úpravám standardní, výsledky tedy uvádím pouze jako přílohu (viz kap. 6.7). Přesto jsou srovnání počtu vektorů i časů někdy zajímavá a svědčí o tom, že problém generování testovacích je skutečně netriviální záležitost a má opravdu svá úskalí, která se mohou projevit i po několika letech vývoje softwaru.

2.8.4. Rozbor závislostí mezi poruchami

Po diskuzi zjištěné možnosti redukce bylo třeba ještě uvážit, zda opravdu touto redukcí neztrácíme důležité informace, které nesou dominované poruchy a jim ekvivalentní poruchy. Samotné ekvivalentní poruchy je možno v každém případě nahradit kteroukoliv z nich. Jak z literatury, tak jednoduchým náhledem je zřejmé, že každá ekvivalentní porucha (viz kap. 1.2.4) vždy vyžaduje zcela stejné nastavení vstupních vodičů hradla, přičemž je výstup citlivý na změnu libovolného z nich, takže dané nastavení umožní otestovat všechny ekvivalentní poruchy, včetně odpovídající poruchy na výstupu. Vzhledem k tomu, že pro testování obecně platí zásada pouze jedné současné poruchy v obvodu, můžeme testovat jedním vektorem všechny tyto poruchy; vícenásobné testování je touto zásadou obecně podmíněno, nebylo by bez ní možné ani smysluplně testovat, ani slučovat testovací vektory. U ekvivalentních poruch nejen že můžeme otestovat všechny současně, jiná možnost ani neexistuje. Otestujeme je, i kdybychom nechtěli.



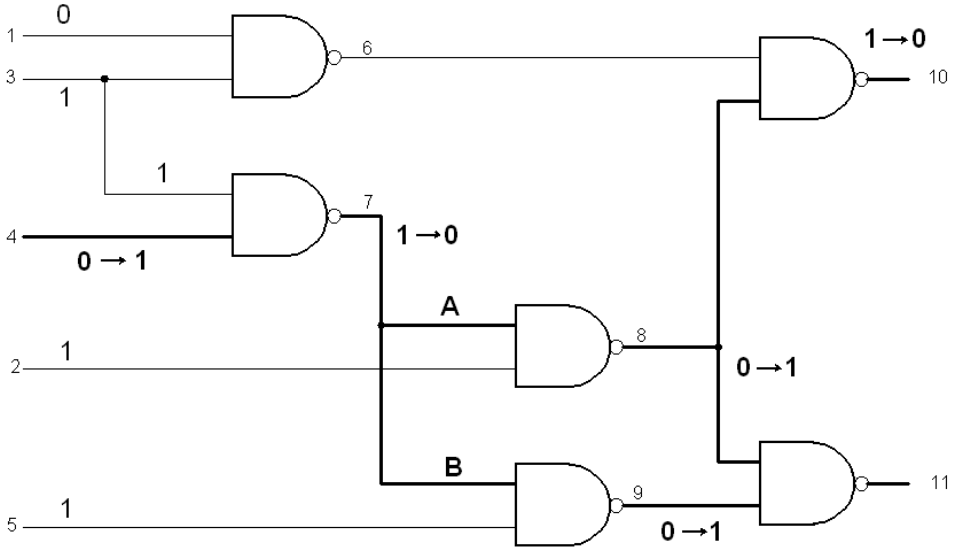
Obr. 7

Benchmark c17.bench s ukázkou citlivé cesty. Citlivá cesta pro poruchu Sa1 na vodiči4. Vstupní vektor (0,1,1,0,X) který je schopen detekovat tuto poruchu, detekuje současně příslušné poruchy na zvýrazněných vodičích. Kromě toho ale i další vedle cesty, ty které jsou nastavené tak, aby byla cesta citlivá.

Trochu jiná je situace u dominance. Jak již bylo zmíněno, můžeme si ji představit také jako závislosti poruch na citlivé cestě. Pokud je zkoumaná porucha na vstupu hradla, musí skrz toto hradlo vést citlivá cesta. Jednoduchým náhledem můžeme ověřit, že pokud by nastala změna před hradlem, bude odpovídat výsledek tomu jako by proběhla příslušná změna za hradlem. To je vlastnost, kterou zde máme pro šíření poruchy, je to nutné pro její odhalení. Pokud tedy budeme mít vektor, který otestuje první poruchu, musí otestovat i následující, jako i všechny poruchy s příslušnou hodnotou na citlivé cestě. Viz obrázek.

Zatím však nemůžeme říct, kudy tato cesta povede. Nyní nám postačí, že není nutné vytvářet testovací vektor pro dominující poruchu. Podívejme se ještě na druhý typ dominance, u větvení. Zde je situace zcela opačná než předchozí. Větvení nám citlivou hodnotu může rozvést do různých větví. Právě tady je kámen úrazu, protože nevíme, kudy je lepší vést citlivou cestu. Pokud povede cesta přes větev A, ovlivní změna před větvením celou tuto cestu, pokud přes B, ovlivní změna celou cestu přes B.

Může se také stát, že budou zcitlivěné obě cesty. Bylo by to velice výhodné, ovšem pokud by nedošlo díky rekonvergenci k blokování (viz v kap.1.2.4). Vektor, který by testoval počátek větve, by otestoval současně obě větve, a tedy ještě více poruch. Zamysleme se nad tím, co se děje se závislostí těchto poruch. Pokud máme poruchu na vodiči A, který jde z větvení, a vektor, který ji testuje, musí citlivá cesta procházet počátkem větve a tedy testuje také stejnou poruchu na počátečním vodiči. Říkejme opět, že porucha na vodiči za větvením dominuje poruchu na vodiči před ním. Současně toto platí také pro vodič B nebo pro další vodiče vycházející z větvení.

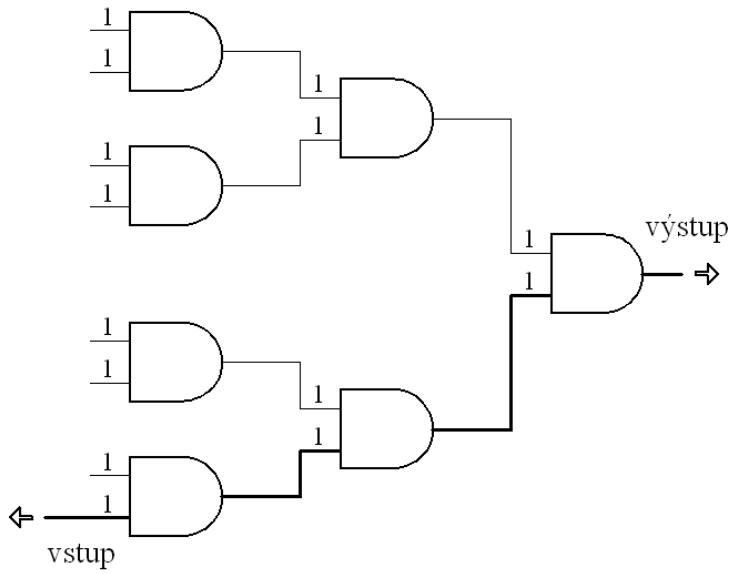


Obr. 8

Benchmark c17.bench a ukázka větvení citlivé cesty

Dominanci můžeme shrnout jednoduše takto: Testovací vektor, který odhalí dominovanou poruchu, odhalí vždy i dominující poruchu. Opačná implikace ale neplatí. Uvědomme si, že toto by platilo pouze tehdy, pokud by se v obvodu citlivá cesta nikde nevětvila a to ani ve směru šíření této cesty, ani opačným směrem. Takovýto obvod by se dal ale rozdělit na nezávislá vlákna sestávající se pouze z 1 vodiče a jednovstupových hradel - invertorů nebo identit, a ta by šla kompletně otestovat vždy jen dvěma vektory 0 a 1, pro Sa1 a Sa0. Pokud by se vyskytlo větvení "zpět", tedy vícevstupové hradlo, nemůžeme rozhodnout obecně o poruchách ve zpětném směru, nevíme odkud se vlastně šíří citlivá cesta. Pokud by se objevilo větvení na vodiči, nevíme kam povede. Naopak to ale můžeme říct jednoznačně.

Vytýčili jsme si dva okruhy závislostí poruch, které můžeme zjistit ze základní znalosti topologie obvodu. V prostudovaných materiálech ani ve zkušebním software však nebyly tyto okruhy dány blíže do souvislosti, která je podle mne ze zkoumaných závislostí velice jednoduchá a dokonce i snadno viditelná. Velmi prostou úvahou lze udělat počáteční krok. Vezměme detekovatelnou poruchu, kterou dominuje jedna z ekvivalentních poruch. Bude dominována i ostatními poruchami? Nebo vezměme jinou detekovatelnou poruchu např. A, která má několik ekvivalentních poruch, z nichž jedné dominuje nějaká porucha B. Dominuje tedy A také porucha B? Odpovědí je v obou případech "ano". Jak již bylo řečeno, ekvivalentní poruchy jsou prakticky libovolně zaměnitelné, nejde je nijak preferovat, ani vyloučit. Chovají se v podstatě jako jeden celek.



Obr. 9

Strom ANDů. Všechny vstupy na obrázku jsou nastaveny na logickou 1. Takto lze detekovat libovolnou poruchu Sa0. Všechny tyto poruchy jsou ekvivalentní

Pro představu si můžeme vzít strom hradel bez větvení, který je tvořen pouze hradly AND, viz Obr. 9. Pokud budeme chtít otestovat libovolnou poruchu Sa0, musíme v celém obvodu nastavit samé 1. Kromě toho otestujeme také každý jiný vodič na Sa0. Pokud bychom výstup z tohoto celku měli jako vstup do nějakého hradla, dominovala by libovolná porucha z celku odpovídající poruše na výstupu z tohoto hradla. Naopak pokud bychom jeden ze vstupů do stromu ANDů měli veden z větvení nějakého vodiče, dominuje každá Sa0 mezi ANDy Sa0 na tomto vodiči. Opačné závislosti jsou viditelné také velmi snadno. Tedy že porucha Sa0 na větvení z výstupu stromu by dominovala celému stromu a příslušná vstupní porucha nějakého hradla na vstupu stromu by dominovala opět celému stromu - velice odvážná, ale zřejmá myšlenka!

Účel, ke kterému chci využít tyto úvahy, je redukce poruch. Strom ANDů lze otestovat na všechny Sa0 pomocí jediného vektoru. Tento vektor bude testovat také všechny poruchy dominované libovolnou poruchou tohoto stromu. Obecně tedy pokud najdu propojení tříd ekvivalence pomocí dominancí, mohu dominovanou třídu vyjmout ze seznamu poruch. Navíc toto platí také tranzitivně pro další třídy, kterým dominuje libovolná z vyjmutých poruch této třídy. Úspora seznamu poruch je značná. Přestože výsledný seznam poruch není řádově menší, pro generování testovacích vektorů je zmenšení příjemné a dle mého názoru by šlo tuto myšlenku využít i pro usnadnění tvorby jednotlivých vektorů.

Pozn.: Poruchy Sa1 v daném stromu ANDů nejsou ekvivalentní, ale jsou dominovány. Sice jen vždy po jedné citlivé cestě, ale bez větvení lze dojít od primárních vstupů s dominancí až na výstup. Tedy stačí otestovat jedinou Sa0 a všechny Sa1 pouze na primárních vstupech. Pokud by byl strom binární, s 2^{n-1} hradly, tedy $2*2^{(n+1)}-1$ poruchami, kde n je výška stromu, stačí otestovat $1+2^n$ poruch. Tedy zhruba 3x méně. To ostatně odpovídalo i při testování, úspora byla max. třetinová, v průměru vychází úspora na polovinu.

2.9. Generování vektorů

K prověření pokrytí redukovaného seznamu poruch bylo nejprve potřeba doprogramovat samotné generování testovacích vektorů. Vycházel jsem nejprve z algoritmu D, který jsem předpokládal, že rozšířím o podmínky použité v heuristikách PODEM a FAN [Ch04]. Obecně tento algoritmus používá primitivní a přenosové D-krychle, které se skládají do tabulky singulárního pokrytí a do tabulky D-krychlí, což jsou většinou řídké matice.

Z druhé tabulky se při prvním kroku tvorby vektoru vybírají řádky, které jsou vhodné pro vytvoření citlivé cesty. Po úspěšném zcitlivění se z první tabulky vybírají řádky vhodné pro tzv. konzistenci, to je nastavení potřebných vstupů. Vzhledem k velkému počtu rozhodování a tedy i návratů je tento algoritmus velmi pomalý.Také kvůli paměťově náročným tabulkám jsem se rozhodl postupovat jinak. Zavedl jsem informace o hradlech jen do struktury a podle typu hradla se provedou příslušná nastavení až při zpracování. Operaci konzistence jsem se rozhodl pozměnit, aby byla jednodušší a neprocházela zbytečně větve, které nejsou potřeba nebo většinou nevedou správným směrem. K tomu lze využít jednoduchou znalost topologie, kterou používá FAN. Jedná se o znalost toho, který vstup (resp. výstup) je blíže primárnímu vstupu (resp. výstupu). Dále se při rozhodování jak nastavit vstupy používá dvou strategií: **všechny vstupy - nejtěžší** a **libovolný vstup - nejlehčí**. Tedy pokud je potřeba nastavit na hradle všechny vstupy, testuje se nejprve ten, který jde nastavit nejhůře. Pokud stačí nastavit jeden vstup, zkusím naopak nejprve ty nejsnáze nastavitelné.

2.9.1. Počáteční algoritmus – jednoduchý průchod

Jednoduchý průchod po citlivé cestě

Algoritmus, který jsem nejprve vytvořil pro průchod strukturou:

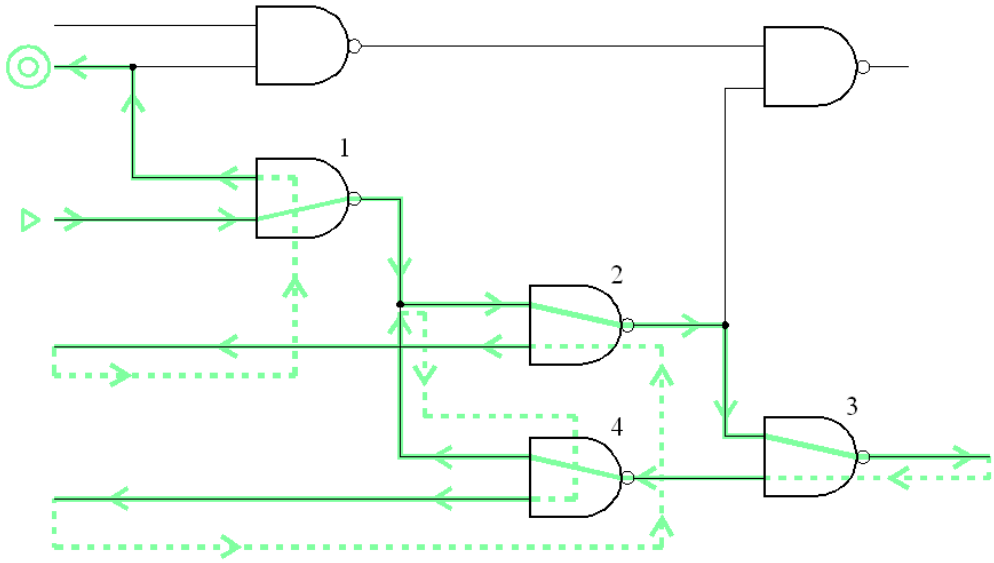
Pro danou poruchu volám funkci sensW, která zcitliví vodič s danou poruchou. Jedná se o rekurzivní proceduru, která se snaží zcitlivět další úroveň směrem k výstupům, pomocí volání funkcí sensG a sensW, podle toho co je v cestě. Když narazí citlivá cesta úspěšně na primární výstup, zavolá funkci setbackW, která se vrací s pomocí funkce setbackG v obvodu zpět a ověřuje všechny vstupy, které je potřeba nastavit. Původně nebylo potřeba procházet znova celý obvod, protože jsem volal už při zcitlivění setbackW, která se pokusila nastavit potřebnou hodnotu již při zcitlivění. To ale může vést ke kolizi s dalšími nastaveními, která budeme provádět v budoucnu a pokud již dál nemáme na výběr, znamenala by tato skutečnost vlastně zamítnutí právě zvolené citlivé cesty. Funkce můžeme zjednodušeně zapsat takto:

```
sensW:
            If W.toGate W.toGate.sensG
            Else If W.toWires for X in W.toWires X.SensW
            Else if W.setbackW Write(vstupy a vystupy obvodu)

sensG:
            If G.toWire G.toWire.sensW

setbackW:
            If W.fromWire W.setbackW
            Else If W.fromGate W.setbackG

setbackG:
            If G.fromWires for X in G.fromWires X.setbackW

Použití ze seznamu poruch FL:
FL.getWire.sensW
```

Pokud se ovšem při volání funkce setbackW musíme rozhodnout pro nějakou variantu zcitlivění vstupů a poté se z funkce vrátíme, již ztrácíme informaci o dalších možnostech zcitlivění. Toto je důležitý fakt, který je velmi omezující při tvorbě testovacích vektorů. Znamená, že při průchodu obvodem je potřeba všechna rozhodnutí ukládat, protože se k nim při průchodu musíme vracet. Atalanta řeší tento problém vlastním mechanismem ukládání na zásobník. Je to jedna z možností, kterou jsem také zvažoval. Rozhodl jsem se v tomto případě použít spíše zásobníku, který je vlastní programovacímu prostředí s voláním procedur, tedy pomocí rekurzí. Heuristika, kterou jsem nejprve začal vytvářet, měla několik nedostatků, zvlášť přílišné ořezání stavového prostoru, a tím pádem v některých případech neschopnost nalézt řešení - testovací vektor. To může být nepříjemné také z toho důvodu, že v některých případech je potřeba znát pokud možno co nejvíce testovacích vektorů, ne pouze jeden. Proto tedy bylo potřeba změnit algoritmus průchodu.

2.9.2. Průchod s návraty – varianta 1

Průchod se dá realizovat několika způsoby. Jedna možnost je procházet rovnou od vodiče s poruchou směrem šíření signálu až na primární výstup. Nejprve se tedy tímto průchodem vytvoří citlivá cesta. Poté, co dojdeme k primárnímu výstupu, pokusíme se nastavit vstupy, na které se při zcitlivění nehledělo. Protože je nutné kvůli návratům pamatovat si všechna rozhodnutí která při zpětném nastavení provedeme, je potřeba uložit na zásobník, kde pokračovat při nalezení primárního vstupu. To hlavně z důvodu nastavení více vstupů na hradle, kdy musím počítat se vzájemnou provázaností vstupů. Na obrázku Obr. 10 je jeden takový průchod naznačen. Pro jednoduchou orientaci jsem v něm použil značky z orientačního běhu. Trojúhelníček označuje start a souosé kružnice cíl cesty. Čísla u hradel označují v jakém pořadí jsme se k těmto hradlům po cestě dostali, ale současně také převrácené pořadí, v jakém se k nim budeme v průchodu vracet.



Obr. 10

Průchod obvodem c17.bench. Průchod označuje plná čára. Přerušované čáry naznačují skoky při průchodu provedené na základě známých hodnot vodičů a předaných odkazů.

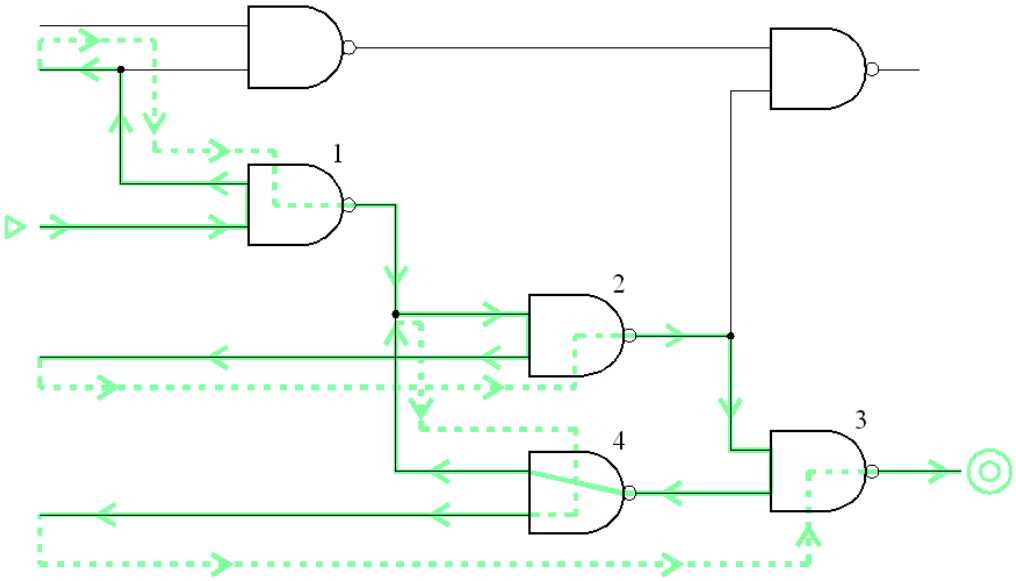
Od startu pokračujeme postupně přes hradla označená 1 2 a 3 až na primární výstup. Průchod označuje plná čára. Na výstupu musíme provést operaci konzistence – nastavit zpětně všechny potřebné hodnoty pro to aby cesta byla citlivá. Přerušovaná čára označuje skok z jednoho místa obvodu na jiné. Budeme tedy pokračovat necitlivým vstupem u hradla 3, směrem k hradlu (na jehož výstupu bude zřejmě potřeba nastavit hodnotu 1). Zde zjistíme potřebné hodnoty na vstupech a pokračujeme v cestě. Na obrázku je naznačen neúspěšný pokus o nastavení hodnoty vodiče, který vychází z větvení, a je na něm tedy citlivá cesta. Na druhém vodiči byl pokus již úspěšný, a pokračujeme tedy dalším nevyřešeným hradlem – hradlem 2. Takto se postupně dostaneme (pokud všechna nastavení byla

úspěšná) až ke vstupu hradla 1, kde již není známo žádné další hradlo, které je potřeba nastavit. V tuto chvíli je na vstupních vodičích požadovaný testovací vektor.

Hradlo, ke kterému se v cestě bezprostředně skokem vrátíme, se předává jako parametr příslušné rekurzivní funkce. Při volání vyřešení hradla 4 již víme, že se jedná o poslední vstup hradla 3, a jako parametr se předá nevyřešené hradlo předcházející hradlu 3. V tomto případě se jedná o hradlo 2. Kolize zde může vzniknout vždy jen pokud při průchodu zpět narazíme na větvení s hodnotou, která se vylučuje s hodnotou, kterou požadujeme. To můžeme způsobit např. již volbou citlivé cesty.

2.9.3. Průchod s návraty – varianta 2

Druhá možnost je procházet vstupy již při vytváření citlivé cesty – viz Obr. 11. Tím si budeme vždy jisti, že nevznikla kolize již při jejím vytvoření a neprocházeli jsme od kolidujícího místa dál již zbytečně. V této variantě průchodu ale může vzniknout kolize jinak. Pokud si budeme muset při průchodu vybrat z více možností, jak vytvořit potřebnou hodnotu vodiče, možná vytvoříme budoucí kolizi s některou z takto předem vytvářených operací konzistence a bude nutné se vrátit v citlivé cestě zpět a znova ji vytvářet.



Obr. 11
Průchod obvodem, druhá možnost, s předem nastavenými vstupy na citlivé cestě. Nejprve postupně nastavíme všechny potřebné vstupy hradel 1, 2 a 3 (to vyžaduje nastavení 4). Je vidět, že po nastavení hradla 4 již víme, že se vracíme přímo na č. 3. To už má na výstupu primární výstup obvodu a všechny potřebné hodnoty jsou na primárních vstupech.

Tyto průchody jsou v podstatě rovnocenné, záleží na topologii obvodu, který je vhodnější. Návraty jsou realizovány podobným mechanismem jako v prvním průchodu, skoky se jen provádějí „blíž“ v cestě kterou procházíme. Rozdíl je v tom, že konec cesty je na primárním výstupu, opět ale platí, že na primárních vstupech obvodu bude po dosažení cíle požadovaný testovací vektor.

V obrázcích Obr. 10 a Obr. 11 je záměrně porucha na primárním vstupu, začíná se tedy bez potřeby nastavit hodnoty primárních vstupů ještě pro správnou hodnotu na vodiči s poruchou. U obou variant je totiž ještě možné vybrat si, zda při průchodu provést nejdříve nastavení hodnoty poruchového vodiče nebo s nastavením počkat až po vytvoření citlivé cesty. Pokud bychom nastavovali hodnotu před vytvářením cesty, odpovídá průchod navíc druhému typu průchodu, končíme na výstupu, který je na vodiči s poruchou. Pokud bychom chtěli řešit tento problém až po nastavení celého obvodu, odpovídá situace prvnímu průchodu, kdy se skokem dostaneme na vstup poruchového vodiče a nastavíme jeho hodnoty. Je proto výhodnější použít pro první variantu nastavení po průchodu celým obvodem a u druhé ještě před průchodem. Není pak potřeba implementovat oba styly průchodu, je to pak také dobré kvůli jednotnosti celého průchodu.

Po úspěšném nastavení všech potřebných vodičů je testovací vektor zaznamenán na primárních vstupech obvodu a stejně tak i výstup by měl odpovídat bezporuchovému stavu. Poruchový stav se dá jednoduše odvodit díky nastaveným citlivým hodnotám. Ostatní výstupy obvodu lze provést velmi rychle částečnou simulací funkce obvodu - z jednotlivých výstupů až po nalezení hodnot jednoznačně určující zkoumaný výstup.

Problém u jednoduchého průchodu nastává, pokud jsou při nastavení cesty maskovány některé dominující poruchy. To je způsobeno tehdy, když citlivá cesta rekonverguje (viz také kap. 1.2.4), přičemž nevybraná větev blokuje zvolenou citlivou cestu. Je potřeba zavést podobně jako v D algoritmu více hodnot. Jednu z nich požadovanou pevnou a druhou pro případ rekonvergence citlivé cesty, citlivou. D algoritmus využíval krychlí, ve kterých byly obsaženy současně pevné hodnoty a zcitlivěné hodnoty. Pro případ rekonvergence je potřeba také uvažovat krychli, ve které jsou zcitlivěny všechny vstupy - kromě XORu lze mít u všech hradel. (Ve skriptech od prof. Hlavičky [Hla98] není tato drobnost zmíněna, pro uvedené příklady ale nebylo této úpravy potřeba.) Složitější obvody ovšem tímto způsobem zcitlivět nelze, je potřeba zcitlivět obě cesty současně. Nelze počítat s tím, že některá půjde jako samostatně citlivá propagovat. U standardního postupu u operace konzistence bychom v těchto případech vždy narazili.

Jedna z možností je tuto operaci pozměnit tak, že bude uvažovat více možností současně, tedy hodnot, kterých může vstup nabývat. Například u hradla AND je potřeba na nezcitlivěných vstupech nastavit "samé 1", aby byla cesta citlivá. Pokud je ale nějaká z těchto jedniček citlivá, změnila by se na nulu a cesta by se při poruše mohla zablokovat. To by nastalo v tom případě, kdy by byla na citlivém vstupu 0. Pokud by zde byla 1, cesta by zůstala otevřená přes to, že by došlo ke dvěma změnám. Je tedy potřeba, aby vodiče mimo citlivou cestu na hradle AND (NAND) nebyly citlivé (byly 1) nebo měly stejnou hodnotu jako je citlivý

vstup hradla - tedy citlivou 0 pokud je citlivá 0 a naopak. U hradla OR (NOR) je tomu stejně tak s tím rozdílem, že pokud hodnota ostatních vodičů není citlivá, musí být 0. Komplikovanější je jako již tradičně hradlo XOR, které je citlivé vždy, ale pouze pokud je citlivá jen jedna hodnota. To opět opomínají některé zdroje zmínit. Tedy obvod složený pouze ze samých XORů není vždy všude zcitlivěn. To nastane tehdy, pokud se v obvodu vyskytuje rekonvergence. Při obou změnách zůstane výstup nezměněn.

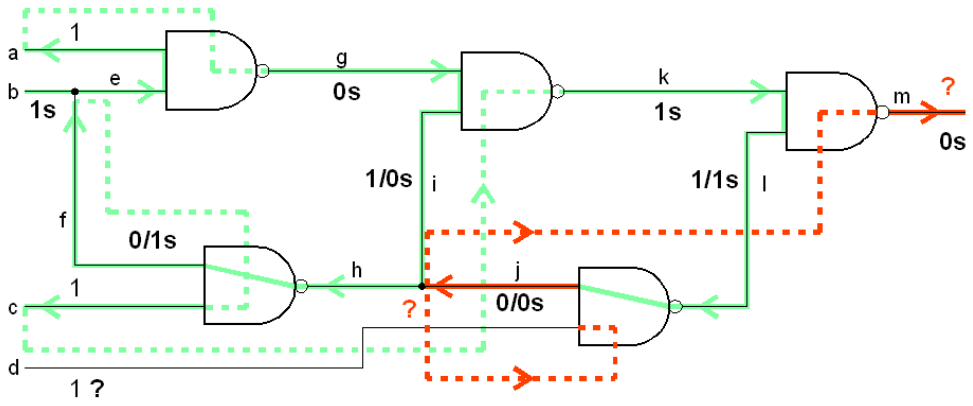
A,B	A',B'	XOR (A,B) = XOR(A',B')
0,0	1,1	0
0,1	1,0	1
1,0	0,1	1
1,1	0,0	0

Tab. 12

Necitlivost hradla XOR na změnu obou vstupů.

Jestliže připustíme, abychom měli na hradlech možnost dvojí hodnoty kvůli rekonvergenci na citlivé cestě, musíme se postarat o to, aby po průchodu byla výsledná hodnota známa. To z toho důvodu, že pokud bude po cestě větvení, můžeme k němu přijít z druhé větve a v té může požadavek kolidovat s jednou z možností. V tom případě je nutné vědět, ke které možnosti jsme dospěli. Zda jsme tedy narazili na citlivou cestu nebo nikoliv. Jak tuto skutečnost zjistíme závisí také na tom, jaký používáme typ průchodu.

Na obrázku Obr. 13 jde vidět pokus o vytvoření citlivé cesty z vodiče a na výstup m. Průchod obvodem je zde proveden druhou variantou, při konzistenci se využívá dvojího ohodnocení vodiče. Na vodiči i mohou být teoreticky dvě hodnoty – logická 1 nebo citlivá 0 (značeno 0s) a cesta bude v obou případech citlivá. Protože poté co zjistíme, jestli jedné z těchto hodnot lze opravdu dosáhnout provedeme skok zpět na citlivou cestu – vodič k, je potřeba zaznamenat, jestli je i citlivý nebo ne. Pokud bychom to neudělali, nemáme tuto informaci pro konzistenci kterou provádíme dále na vodič l, u kterého opět můžeme mít dvě hodnoty. Až bychom se dostali ke větvení, chyběla by informace, jestli máme vstup nastaven správně, či nikoliv, a tedy kterou cestou se má dále obvod procházet.



Obr. 13

Průchod obvodem s kolizí citlivé cesty při použití dvojího ohodnocení vodičů.

2.9.4. Algoritmus s množinami vektorů

Třetí možnost vytváření cesty je poněkud odlišná, týká se přímo nastavování vstupů při průchodu obvodem. Zatímco nejprve jsme procházeli obvod a zkoušeli provést nastavení, zde přímo nastavujeme a porovnáváme. Opět si můžeme vybrat pořadí nastavení vodiče s poruchou. Celkově tento způsob odpovídá spíše druhému typu průchodu. Pro vodič na citlivé cestě se nejprve zavolá funkce getInSet, která vrátí sadu vstupních vektorů, kterými lze vodič nastavit na požadovanou hodnotu. Tuto sadu předáme hradlu na citlivé cestě a srovnáme ji se sadou, kterou dostaneme voláním getInSet na ostatní vstupní vodiče hradla, s příslušným parametrem jakou požadujeme od nich hodnotu, popřípadě více hodnot. Opět musíme počítat s citlivou cestou, na kterou můžeme narazit. Jednalo by se pouze o speciální hodnotu, podobně jako D a D` u D-algoritmu.

Porovnáním sad vektorů, nebo lépe řečeno jejich průnikem a sjednocením, vzniká nová sada, která je nutná pro nastavení hradla v citlivé cestě. Pokud bychom narazili na primární výstup, dospěli jsme ke konci a máme sadu vektorů pro zvolenou cestu. Pokud je při vytváření cesty možnost volby, vrátíme se a projdeme takto celý obvod. Tím získáme všechny testovací vektory pro zvolenou poruchu. V tomto schématu odpadá nutnost procházet možnosti volby ve směru "zpět", tedy k primárním výstupům. Všechny kombinace dostaneme již jako sadu vektorů, kterými lze nastavit požadovanou hodnotu. Není tedy nutné vracet se podobně jako v první části a ztrácet čas backtrackingem. Zřejmě ale operace, které budou vytvářet sady vektorů a jejich průniky a porovnání budou náročnější než prostý průchod, který odpovídá předávání ukazatelů a porovnání jednoduchých celých čísel.

Reprezentace vektorů může zásadně ovlivnit rychlost zpracování hodnot a celého generování testu. V poruchové simulaci se s výhodou používají pro urychlení logické operace nad celočíselnou reprezentací vektorů, jehož hodnoty odpovídají bitům celého čísla. Speciální hodnoty jako DC se řeší zdvojením vektorů pro masku těchto hodnot. Podobně i v našem případě můžeme rozvinout dvě možnosti použití tohoto modelu. Buď můžeme použít celé číslo jako vstupní vektor, kde jednotlivé bity mohou odpovídat vstupům obvodu (přesnějším definováním průniku by pak bylo velmi rychlé vytvoření nového vektoru ze sady a každý vektor by byl jedno celé číslo), anebo lze vytvořit logiku, ve které bude každá hodnota vstupu reprezentována celým číslem. Jeho bity budou odpovídat hodnotám jednotlivých vektorů ze sady. Pro toto vyjádření sady vektorů je výhodou nezávislost na velikosti celého čísla, co se počtu vstupů obvodu týče. Naopak nevýhodou je správa jednotlivých bitů a po překročení velikosti sady vektorů nad počet bitů nastane menší problém i tady. Pro vyjádření hodnot metodou „jeden vektor - jedno číslo“ je jednodušší použít operaci slučování. Nevýhodou je omezení celého čísla kvůli počtu vstupů. To ovšem není neřešitelný problém. Dá se zvládnout celkem snadno vlastní, trochu rozšířenou, aritmetikou.

Pravidla pro slučování vektorů jsou celkem zřejmá. Jedná se v podstatě o speciální průnik nebo sjednocení přes obě množiny. Pokud je nutné nastavit na hradle všechny vstupy, jedná se o průnik. Pokud stačí nastavit jeden vstup, je to sjednocení obou množin. Hodnota proměnné, odpovídající jednomu primárnímu vstupu, může nabývat čtyř hodnot: 1, 0, X (DC) a prázdný průnik - označme si jej Z. Sjednocení se provede prostým sloučením dvou množin, ve kterých se vyjmou duplikované prvky. Jedná se o komutativní a asociativní operace. Tabulka Tab. 14 ukazuje výsledky operace průnik pro všechny kombinace ze čtyř možných hodnot proměnné vektoru.

A\B	0	1	X	Z
0	0	Z	0	Z
1	Z	1	1	Z
X	0	1	X	Z
Z	Z	Z	Z	Z

Tab. 14
Průnik proměnných A a B dvou vektorů

Abychom tuto operaci udělali co nejsnazší, musíme vytvořit reprezentaci vektoru takovou, pro kterou můžeme použít nějakou elementární bitovou logickou operaci. Vektor bude tvořen dvěmi sadami bitů, pro každou hodnotu proměnné potřebujeme dva bity. Prof. James Li [Ch04] uvádí pro logické operace v simulaci poruch pouze tříhodnotovou logiku: 0, 1 a X, kde hodnoty 0 a 1 ve vektorech jsou shodné, pro X je v prvním vektoru 0 a ve druhém 1.

Při podrobnějším zkoumání závislostí operací vidíme, že vektor pro Z v naší aritmetice odpovídá nulovému prvku, protože každá operace s ním dá výsledek Z. Naopak vektor X odpovídá neutrálnímu prvku. Z každé relace s ním vyjde druhý prvek nezměněn. Dále je potřeba aby 0 a 1 byly neutrální k sobě samým a inverzní navzájem. Nabízí se dvě základní možnosti, jak aritmetiku vytvořit. Pro operaci průniku, která bude bitový "and", by prvek X byl (11), Z (00), 0 (01) a 1 (10). Pokud použijeme "or", budou prvky X a Z přehozeny. Vzhledem k tomu, že pro 0 a 1 platí inverzně všechna pravidla, na zvoleném pořadí bitů u nich nezáleží.

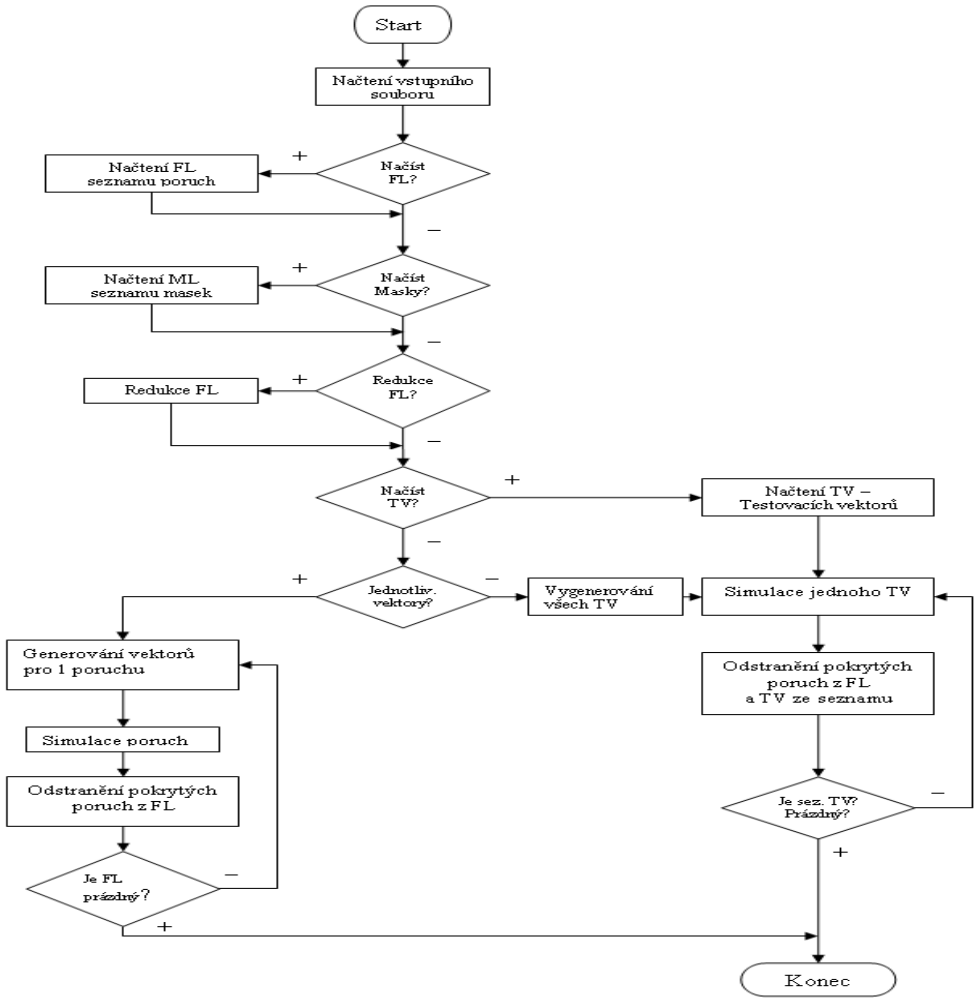
Nyní máme operace vytýčeny, zbývá jen operace porovnání na rovnost a odhalení, že ve vektoru došlo ke kolizi, tedy že v něm vznikla hodnota Z. Vezměme bez ohledu na obecnost variantu logiky s operací "or". To lze jednoduše provést vymaskováním pomocí operace and. Pokud dvě složky vektoru logicky vynásobíme, dostaneme na místě Z logickou 1. Pro variantu s operací "and" by bylo potřeba použít logický součet a prázdný průnik by identifikovala 0. Při implementaci jsem zvolil pro operaci průniku bitový "or". Více o operacích a struktuře viz kap. 3- Struktury.

2.9.5. Implementace ATPG

Po vytvoření struktur pro seznam poruch, jeho redukci, vytvoření poruchového simulátoru a načítání seznamu poruch ze souboru, jsem dále pokračoval implementací vlastního ATPG. Pro implementaci jsem si nejprve vybral zmíněný velmi jednoduchý průchod popsaný v kap.2.9.1. Jeho výsledky nebyly uspokojivé, kvůli zmíněnému ořezávání dobrých variant testovacích vektorů. Proto jsem pokračoval v rozšíření tohoto průchodu a jeho úpravám. Rozhodl jsem se naimplementovat druhou variantu průchodu, při které se nastavují nejdříve vstupy hradel na citlivé cestě, viz kap. 2.9.3.

Průchod bylo potřeba otestovat pomocí jednodušších benchmarků, které jsem si za tímto účelem vytvořil. Použil jsem pro ně formát ISCAS'85. Jednalo se o několik základních hradel a jejich jednoduché kombinace. Účelem bylo ověřit zejména změny na citlivé cestě, způsobené rekonvergencí, viz kap. 1.2.4.

Tvorbu vektorů jsem nejprve vytvářel samostatně, poté jsem ji dal do kontextu s ostatními prvky, které tvoří ATPG jako celek. Vzhledem k tomu, že program měl provádět více rozdílných operací, bylo potřeba oddělit generování jednotlivých vektorů a generování kompletní sady ze seznamu poruch. Do programu jsem tedy přidal ovládání pomocí parametrů, kterými lze určit, zda se má načíst seznam poruch, testovací vektory a vektory masky. Dále jestli se má seznam poruch redukovat či ne, jestli se má vypisovat, a který. Ve výchozím nastavení aplikace provádí generování kompletní sady vektorů, ale i toto nastavení lze potlačit. Chování programu je znázorněno na obrázku Obr. 15**.** Parametry jsou popsány v příloze, kap. 6.8.



Obr. 15

Diagram znázorňující zjednodušeně chování programu.

Při načtení vstupního souboru se vytvoří plný seznam poruch. Pokud se zadá načtení externího seznamu poruch, pouze se pozmění plný seznam. Podle vstupních parametrů se také rozhodne, zda načíst ze souboru testovací masky a vektory. V případě, že se načítají vektory se pokračuje simulací kompletního seznamu. Bez souboru s vektory program vygeneruje testovací vektory sám. Pokud je zadán parametr pro postupné testování, generují se vždy vektory pouze pro 1 poruchu. Teprve po otestování a odebrání detekovaných poruch se generují další.

Při implementaci jsem narážel na problémy týkající se hlavně složitějších postupů v případě nalezení citlivé cesty, kdy je potřeba předat tuto informaci do struktury a dál s ní umět správně naložit. Vzhledem ke stylu průchodu bylo snadné zabudovat do algoritmu nastavování masek pro určení požadovaných hodnot na vstupech. Algoritmus průchodu je dost časově náročný, protože nevyužívá

zjednodušení, které si mohou dovolit jiné programy, když generují sady bez dalších omezení. V našem případě je zjednodušujícím faktorem právě maska, která zmenší prostor, který je potřeba projít abychom nalezli požadovaný vektor.

V příloze s detailními výpisy testovacích vektorů pro obvod c17 (kap. 6.5) je patrné, že Atalanta se dopouští při generování vektorů nepřesností, a že je prakticky nevhodná pro generování úplné sady testovacích vektorů. Přestože bylo explicitně zadáno, že požadujeme všechny vektory, nejsou sady kompletní. V některých případech jsou určeny hodnoty, které vůbec nebylo potřeba určit. V tomto případě se jistě nejedná o chybu v aplikaci, ale o použití nevhodného nástroje pro generování úplné sady, kterým Atalanta jak je vidět není.

Proto jsme také přistoupili k vlastní implementaci generování testovacích vektorů.

3. Struktury

Při implementaci jsem navrhoval struktury tak, aby byly pokud možno minimální, a zároveň aby zachovaly co největší informovanost o obvodu i o blízkém okolí jednotlivých prvků obvodu. Ve snaze co nejvěrněji přenést reálné závislosti do struktury jsem upustil od zjednodušení, ve kterém jsou některé prvky pouze virtuální (jako je tomu např. u Atalanty, kde jsou vodiče nahrazeny pouze odkazy na další hradla - to sice uspoří několik hodnot, které vodič nese, ale ztrácíme informaci o větvení). V této struktuře se dají větvení na vstupu hradla rozpoznat jedině tak, že předchozí hradlo, které je na vstupu zkoumaného hradla, má na výstupu více než jedno hradlo. Pokud chci přidat do seznamu poruch poruchu pro výstupní vodič hradla, nemohu přidat pouze jednu, ale musím spočítat všechna větvení a přidat ještě jednu poruchu pro vodič před větvením. Vzhledem k tomu, že je tento vodič již zcela imaginární, nepracuje se s tímto modelem moc dobře. Pokud bychom počítali poruchy pro všechny vývody z hradla, opět by nám přebývala jedna porucha u každého nevětvícího se vodiče.

Proto jsem vytvořil návrh, ve kterém byly hradla i vodiče. Zkoumal jsem také možnost mít pouze spoje hradlo-vodič a naopak. To ale mělo opět za následek problémy s větveními. Proto jsem do struktury vložil dvě možnosti – spoje hradlo – vodič (a naopak) a spoje vodič-vodič, pro větvení. Tato struktura umožňuje velmi snadnou správu seznamu poruch. Ten je tvořen všemi vodiči ve struktuře, každý je zahrnut dvakrát; zvlášť pro trvalou 0 a trvalou 1.

Pro vodič jsem navrhl strukturu DDWire, která vypadá zjednodušeně takto:

```
class DDWire {
          int outputs;
          int un_checked;
          int level,olevel;
          int value;
          char *name;
          DDFault *FL;
          DDGate *fromgate, *togate;
          DDWire *fromwire, **towires;
}
```

DDFault je struktura seznamu poruch a DDGate struktura pro hradlo. Jak jsem již zmiňoval v kap. 2.7, díky jednoduchému vztahu mezi seznamem vodičů a seznamem poruch je velmi snadné vytvoření a práce se seznamem poruch. Jeho zjednodušená struktura vypadá takto:

```
class DDFault {
          DDWire *fw;
          int stuckAt;
          int active;
}
```

Pro snadnou redukci seznamu poruch obsahuje každá porucha ještě ukazatele na dominované, dominující a ekvivalentní poruchy. Struktura má také další ukazatele jako *next, čímž je možné z ní vytvořit seznam.

DDGate je struktura hradla; vypadá takto:

```
class DDGate {
     int gTyp;
     int inputs;
     int un_checked;
     int level,olevel;
     int value;
     char *name;
     DDWire **inwires, *outwire;
}
```

Toto jsou základní prvky, v průběhu jsem strukturu měnil, podle toho, jaký algoritmus jsem se snažil implementovat a prozkoumat jeho možnosti. Ve výsledku jsem dospěl ke dvěma algoritmům. Jeden jsem ponechal ve fázi, kdy generuje pouze jeden testovací vektor na poruchu, druhý využíval navíc strukturu logicSet pro množiny se čtyřhodnotovou logikou.

Struktura logicSet obsahovala dynamický seznam vektorů, tvořených třídou logic4. Ta vypadá takto:

```
class logic4 {
     int *val1,*val2;
     int size,bits;
public:
     logic4();
     logic4(int bit);
     logic4& operator|(logic4& lr);
     logic4& operator=(logic4& lr);
     logic4& setPIN(int bit, int val);
     int isEmpty();
     int operator==(logic4& lr);
     ~logic4();
}
```

Základní hodnoty vektoru jsou zakódovány v proměnných val1 a val2 pomocí aritmetiky s bitovým or, viz kap. 2.9.4, počet bitů je v proměnné bits a délka pole ve velikosti int v proměnné size. Nejdůležitější pro aritmetiku je operátor | reprezentující průnik vektorů pomocí zmíněného bitového or. Pro předání s vytvořením nového vektoru jsem vytvořil operátor =. Pro nastavení konkrétního primárního vstupu na požadovanou hodnotu slouží metoda setPIN, kontrolu zda je vektor neplatný zjišťuje metoda isEmpty.

Třída obsahuje i metodu pro výpis pomocí ostreamu, operátor <<.

Tuto třídu používá třída „záznam logické proměnné“ logicRec. Ten je důležitý jenom jako prvek nesoucí vlastní hodnotu reprezentovanou proměnnou typu struktura logic4 a ukazatel na další prvek. Pro zpřehlednění je nadstavbou této třídy ještě logicSet, která představuje množinu vstupních vektorů, a ve které jsou již implementovány operátory * a + pro slučování těchto množin pomocí průniku a sjednocení. Tyto operace jsou vlastně operacemi nad logicRec, průnik provede průniky každého vektoru z jedné množiny s každým vektorem z druhé množiny, sjednocení je pouze spojí. Tyto operace jsou základní kameny pro funkce, které prochází obvod. Dále se již využívají hlavně ukazovatele na hradla DDGate a vodiče DDWire, jejich hodnoty a u hradel typ.

Pro další pomocné hodnoty v obvodu je používána proměnná un_checked a nebo proměnná state, která má podobnou funkci - udává v jakém stavu je který prvek při průchodu algoritmem. Na začátku je obvyklé, že jsou všechny v nedefinovaném stavu. Pokud je rozpracován, používá se stav Open, pokud je zpracování ukončeno, stav Close. Protože struktura může být rozsáhlá, lze očekávat, že mnoho času se zbytečně stráví mazáním stavů prvků. Proměnná un_checked dynamicky mění stavy při průchodu, čímž situaci zjednodušuje. Při vytváření množin je totiž potřeba pamatovat např. citlivou cestu kvůli návratu při rekonvergenci, což lze pomocí této pomocné proměnné taky jednoduše zajistit.

Celá načtená struktura je typu DDObj, který obsahuje základní informace o obvodu - počty vstupů a výstupů a samotnou strukturu obvodu, kterou tvoří pole ukazatelů na primární vstupy a výstupy. Dále pak už jen seznam poruch. Strukturu obvodu využívají funkce Simulate a Inject, které provádí simulaci obvodu a poruchovou simulaci injekcí. Struktury jsou v souborech structure.h a structure.cpp.

Kvůli rozsáhlejšímu kódu jsou funkce pro transformaci struktury a průchod obvodem uvedeny v souborech utility.h a utility.cpp. Tyto soubory obsahují také další pomocné funkce, jako jsou výpis seznamu poruch, jeho redukce, a podobně. Z hlavního programu se volají přímo funkce makeTest, které prochází seznam poruch a volají průchody obvodem, tedy vlastní generování testů. Pro použití aplikace pro další vývoj ATPG jsou důležité hlavně výše zmíněné soubory a struktury. Další obsahují podpůrné funkce pro načítání a ukládání souborů a ladění.

4. Závěr

Při zpracování diplomové práce se vyskytlo mnoho zajímavých témat, ale i problémů týkajících se různých oborů číslicové techniky, a dále pak návrhu algoritmů nebo heuristik potřebných k průchodu obvodem, různé přístupy ke tvorbě programů a práci se strukturami v jazyce C/C++.

V průběhu práce se mi podařilo prozkoumat velkou část základních metod generování testovacích vektorů, jednak obecně, z hlediska potřeby obvyklých návrhů obvodů, ale i vzhledem ke speciálním požadavkům, které jsem měl v rámci diplomové práce řešit. Byly to hlavně požadavky k získání testovacích vektorů na základě zadaných omezujících podmínek. Tedy vytvoření masky, která bude při generování aplikována na vstupní vodiče obvodu a podle ní bude sada testovacích vektorů omezena. K tomu bylo potřeba zjistit možnosti současných ATPG nástrojů, které jsem měl k dispozici, ověřit jejich funkce, výhody a slabiny.

Návrhy pro vestavěnou diagnostiku, low-power testing a další okruhy týkající se testování, jako např. komprese testovacích vektorů, vyžadují od ATPG nástrojů možnost diktovat si, jak bude vypadat sada testovacích vektorů; tedy ne pouze aby byla sada co nejmenší, ale také aby byla vhodná k tomu, aby se jednoduše vložila do testovacích nástrojů obvodu, a aby mohlo být efektivně a bez problémů této sady použito k testování.

Výsledkem práce je nástroj, který umí na základě požadavků generovat sady vektorů s externě zadanými omezeními. Kromě toho tento nástroj využívá objevených vazeb mezi poruchami jednotlivých vodičů k tomu, aby silně zredukoval seznam poruch obvodu. Na testovacích specifikacích různých kombinačních číslicových obvodů jsem prakticky ověřil, že tyto závislosti skutečně fungují, ke kterým jevům při redukci dochází, a jak je možné zjištěné nevýhody obejít. Přestože samotné generování nedosáhlo úrovně se srovnávaným softwarem, zvláště kvůli velmi náročnému procházení struktury, dosahuje vyvinutá aplikace sad vektorů, které byly v jiných případech zbytečně zanedbány krátkým průchodem, nebo ztraceny kompakcí, či jiným způsobem, kdy byly nastaveny v testovacích vektorech vstupy, které nebylo potřeba nastavovat.

Nástroj je možné dále využít k vývoji aplikace, která by využila výhod vlastního nastavení testovacích vektorů, ať už k jakémukoliv z výše zmíněných účelů. Algoritmy pro tvorbu testovacích vektorů lze rozšířit, např. o generování náhodných vektorů se zadanou maskou, upravit pomocí dalších znalostí topologie obvodu průchod, vytvořit pro rychlejší generování předzpracování obvodu nebo použít některé pokročilejší metody prohledávání, třeba pomocí učení.

5. Seznam použité literatury

[Hla98] prof. Ing. Jan Hlavička, DrSc: Diagnostika a spolehlivost, ČVUT Praha, 1998

[Nov04] Materiály prof. Ing. Ondřej Novák, CSc., Technická Univerzita Liberec http://www.fme.vslib.cz/~kes/staff/kes_on.html

[Com03] Zahradka, J., Holubec, M., Novak, O.: COMPAS - System for Finding of a Compress Test Pattern Sequence. Proc. of ECMS'03, Liberec, June 2003

[Koz05] Ing. Vlastimil Kozák: Převaděč formátů pro specifikaci logických obvodů (DP ČVUT, FEL, 2005)

[LH93] Lee, H.K. - Ha, D.S.: Atalanta: an Efficient ATPG for Combinational Circuits. Technical Report, 93-12, Dep't of Electrical Eng., Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1993

[LH91] Lee, H.K. - Ha, D.S.: An Efficient Forward Fault Simulation Algorithm Based on the Parallel Pattern Single Fault Propagation. Proc. of the 1991 International Test Conference, pp. 946-955, Oct. 1991.

[LH96] H. K. Lee and D. S. Ha, "HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 9, pp. 1048- 1058, September 1996.

[Ch04] Chien Mo James Li, National University of Taiwan, Lecture notes, 2004. http://cc.ee.ntu.edu.tw/~cmli/Course/VLSITesting/04S/

[Gla1] U.Gläser, "Mehrebenen-Testgenerierung für synchrone Schaltwerke", Dissertation an der Gerhard Mercator Universität Gesamthochschule Duisburg.

[Gla92] U. Gläser, U. Hübner, H.T. Vierhaus, "Mixed Level Hierarchical Test Generation for Transition Faults and Overcurrent Related Defects", in Proceedings of the International Test Conference 1992, pp. 23-29

[Brg85] Brglez, F. - Fujiwara, H.: A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan. Proc. of International Symposium on Circuits and Systems, pp. 663-698, 1985

[Brg89] F. Brglez, D. Bryan and K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits, Proc. of International Symposium of Circuits and Systems, pp. 1929-1934, 1989

[Gar79] Gary, M. – Johnson, D.: Computers and Interactibility: A guide to the theory of NP Completenes, Freeman, 1979

[Lar92] T. Larrabee. Test pattern generation using Boolean satisfiability. IEEE Trans. on CAD, Vol. 11, 4–15, 1992.

6. Přílohy

6.1. Příklad souboru s poruchami

Toto je ukázka formátu souboru se seznamem poruch, který umí načíst software Atalanta. Načítání se provede pomocí volby –f soubor.flt, kde soubor.flt může vypadat takto:

```
gate_A->gate_B /1
gate_A->gate_B /0
gate_A /1
gate_B /1
```

Kde gate_A a gate_B jsou názvy vodičů ve vstupním souboru.
Řádek "gate_A->gate_B /1" znamená, že se do seznamu poruch má zahrnout porucha Sa1 na vodiči the gate_A, který jde do hradla s výstupem gate_B. Řádek "gate_A->gate_B /0" říká obdobnou informaci o poruše Sa0. Z dalších řádek je zřejmé, že gate_A se rozlišuje jako výstup hradla gate_A a vstup do gate_B. Další řádky označují poruchy Sa1 na gate_A a gateB..

Pozn.: Při načítání seznamu poruch do software atalanta se neprovádí kompakce testovacích vektorů, takže je očekávána již kompaktní sada poruch.

6.2. Příklad souboru s testovacími vektory

Ukázka souboru s testovacími vektory pro software atalanta. Pokud začíná řádek jedná se o komentář, a je při načítání ignorován. Ostatní řádky se zpracovávají tak, že se načte testovací vektor, který je za znakem ":" tak, že pokud je jako další parametr programu soubor se specifikací obvodu s n vstupními vodiči, čte se ze souboru s testovacími vektory pouze n bitů, další se přeskočí. Tedy například pro obvod c17.bench, který obsahuje vstupy: input1, input2, input3, input6 and input7 se načte pro testování pouze prvních pět bitů, tak jak jdou ve specifikaci za sebou. Soubor pro c17 může vypadat takto:

```
* Name of circuit:  c17

  1: 01010
  2: 11110
  3: 10101
  4: 00111
  5: 10010
  6: 00101
```

6.3. Příklad souboru se specifikací kombinačního obvodu

Soubor se specifikací obvodu podle ISCAS89. První řádek musí být komentář začínající znakem # za kterým se uvede jméno obvodu. Další řádky začínající # jsou při načítání ignorovány. Na pořadí v jakém se vkládají jména hradel do souboru v této verzi nezáleží. Jména hradel mohou začínat znaky 0-9, A-Z, a-z, _, [, nebo]. Ve formátu ISCAS 85 nemohou jména začínat číslem.

Příklad souboru c17.bench specifikovaného ISCAS89:

```
# c17
# 5 inputs
# 2 outputs
# 0 inverters
# 6 gates ( 6 NANDs )

INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)

OUTPUT(22)
OUTPUT(23)

10 = NAND(1, 3)
11 = NAND(3, 6)
16 = NAND(2, 11)
19 = NAND(11, 7)
22 = NAND(10, 16)
23 = NAND(16, 19)
```

Typy hradel povolených v ISCAS89:

syntax	gate type
INPUT	primary input
OUTPUT	primary output
AND	and gate
NAND	nand gate
OR	or gate
NOR	nor gate
XOR	2 input exclusive-or gate
BUFF or BUF	buffer

```
  NOT                inverter
-----------------------------------------------------
```

U hradle nezáleží na tom, jestli jsou psána malými nebo velkými písmeny.

6.4. Skutečný formát souboru s vektory

Skutečný výstup ze software Atalanta pro obvod c17.bench (ISCAS89).
Soubor s testovacími vektory:

```
* Name of circuit:  netlist.txt
* Primary inputs :
  1 2 3 6 7

* Primary outputs:
  22 23

* Test patterns and fault free responses:

   1: 11101 11
   2: 01111 00
   3: 10000 00
   4: 10111 10
   5: 11010 11
   6: 00011 01
```

Zcela stejný vstupní soubor, pouze převeden na ISCAS85. Atalanta s parametrem -D 1, pro nalezení jednoho vektoru pro jednu poruchu, vytvoří tento výstup jako soubor s testovacími vektory:

```
* Name of circuit:  netlist.txt
* Primary inputs :
  g1 g2 g3 g6 g7

* Primary outputs:
  g22 g23

* Test patterns and fault free responses:

g7->g19 /0
     1: x00x1 01
g7->g19 /1
     1: x00x0 00
g6->g11 /1
     1: 0110x 1x
```

```
g3->g11 /1
    1: x101x 1x
g11->g19 /1
    1: xx111 x0
g11->g16 /1
    1: x111x x0
g3->g10 /0
    1: 101xx 1x
g3->g10 /1
    1: 100xx 0x
g2->g16 /1
    1: 000xx 0x
g16->g23 /1
    1: x10x0 11
g16->g22 /1
    1: 010xx 11
g1->g10 /1
    1: 001xx 0x
```

6.5. Úplný test na c17

Rozdílný pohled na to, jak má vypadat úplný seznam poruch. V prvním sloupci je výstup z mé aplikace, pouze s odstraněním duplicitních vektorů. Druhý sloupec je výstup z Atalanty, která dostala pomocí parametru –f seznam všech poruch ve vektoru, a pomocí parametru –A měla vygenerovat všechny testovací vzory. Ve třetím sloupci je výstup pouze s parametrem –A. Aby vygenerovala Atalanta kompletní seznam všech poruch a vektorů implicitně zadat nejde.

```
Object name: netlist
Number of gates: 6
Input ports: 5
Output ports: 2
```

```
* Name of circuit:  netlist.txt
* Primary inputs :
  g1 g2 g3 g6 g7

* Primary outputs:
  g22 g23
```

```
* Name of circuit:  netlist.txt
* Primary inputs :
  1 2 3 6 7

* Primary outputs:
  22 23
```

```
g7->g19  /0
    1:X00X1 X1
    2:X0X01 X1
g7->g19  /1
    1:X00X0 X0
    2:X0X00 X0
g6->g11  /0
    1:0111X 0X
```

```
g7->g19 /0
    1: x00x1 01
    2: x0101 x1
g7->g19 /1
    1: x00x0 00
    2: x0100 x0
g6->g11 /0
    1: 0111x 0x
```

```
7 /1
    1: x00x0 00
```

2:X1110 X0	2: 11110 10	
3:X1110 X0	3: 11111 10	
4:X0111 X0	4: x0111 x0	
g6->g11 /1	g6->g11 /1	6 /1
1:0110X 1X	1: 0110x 1x	1: 0110x 1x
2:X1100 X1	2: 11100 11	
3:X1100 X1	3: 11101 11	
4:X0101 X1	4: x0101 x1	
g3->g11 /0	g3->g11 /0	
1:0111X 0X	1: 0111x 0x	
2:X101X 0X	2: 11110 10	
3:X1X10 X0	3: 11111 10	
4:X0X11 X0	4: x0111 x0	
g3->g11 /1	g3->g11 /1	3->11 /1
1:0101X 1X	1: x101x 1x	1: x101x 1x
2:X101X 1X	2: x0011 01	2: x0011 01
3:X1X10 X1		
4:X0X11 X1		
g11->g19 /0	g11->g19 /0	
1:X00X1 X1	1: x00x1 01	
2:X0X01 X1	2: x0101 x1	
g11->g19 /1	g11->g19 /1	11->19 /1
1:XX111 X0	1: xx111 x0	1: xx111 x0
g19->g23 /0	g19->g23 /0	
1:X011X X0	1: x0xx0 x0	
2:XX11X X0	2: x0111 x0	
3:X0XX0 X0	3: x111x x0	
4:XX110 X0		
g19->g23 /1	g19->g23 /1	19 /1
1:X00X1 X1	1: x00x1 01	1: x00x1 01
2:X0X01 X1	2: x0101 x1	2: x0101 x1
g11 /0	g11 /0	11 /0
1:010XX 1X	1: x10xx 1x	1: x10xx 1x
2:X10XX 1X	2: 0110x 1x	2: 0110x 1x
3:X10X0 X1	3: 11100 11	3: 11100 11
4:X00X1 X1	4: 11101 11	4: 11101 11
5:01X0X 1X	5: x00x1 01	5: x00x1 01
6:X100X 1X	6: x0101 x1	6: x0101 x1
7:X1X00 X1		
g11 /1	g11 /1	11 /1
1:0111X 0X	1: 0111x 0x	1: 0111x 0x
2:X1110 X0	2: 11110 10	2: 11110 10
3:X1110 X0	3: 11111 10	3: 11111 10
4:X0111 X0	4: x0111 x0	4: x0111 x0
g11->g16 /0	g11->g16 /0	
1:010XX 1X	1: 010xx 1x	
2:X10XX 1X	2: 0110x 1x	

3:X10X0 X1
4:01X0X 1X
5:X100X 1X
6:X1X00 X1

g11->g16 /1
1:0111X 0X
2:X111X X0

g3 /0
1:101XX 1X
2:0111X 0X
3:X1110 X0
4:X0111 X0

g3 /1
1:100XX 0X
2:0101X 1X
3:X1010 X1
4:X0011 X1

g3->g10 /0
1:101XX 1X
2:1X11X 1X

g3->g10 /1
1:100XX 0X
2:1X11X 0X

g2->g16 /0
1:010XX 1X
2:X10XX 1X
3:X10X0 X1
4:01X0X 1X
5:X100X 1X
6:X1X00 X1

g2->g16 /1
1:000XX 0X
2:X00XX 0X
3:X00X0 X0
4:00X0X 0X
5:X000X 0X
6:X0X00 X0

g16->g23 /0
1:X011X X0
2:X0XX0 X0
3:XX11X X0
4:XX110 X0

g16->g23 /1
1:X10X0 X1
2:X1X00 X1

g23 /0
1:X10XX X1
2:X1X0X X1

3: 110xx 1x
4: 11100 11

g11->g16 /1
1: x111x x0

g3 /0
1: 101xx 1x

g3 /1
1: 100xx 0x

g3->g10 /0
1: 101xx 1x
2: 1111x 10

g3->g10 /1
1: 100xx 0x

g2->g16 /0
1: 010xx 1x
2: 0110x 1x
3: 110xx 1x
4: 11100 11

g2->g16 /1
1: 000xx 0x
2: 0010x 0x
3: 100xx 0x
4: 10100 10

g16->g23 /0
1: x0xx0 x0
2: x0111 x0
3: x111x x0

g16->g23 /1
1: x10x0 11
2: x1100 11

g23 /0
1: x10xx 11
2: x110x 11

11->16 /1
1: x111x x0

3 /1
1: 100xx 0x

3 /0
1: 101xx 1x

3->10 /1
1: 100xx 0x

2 /1
1: 000xx 0x

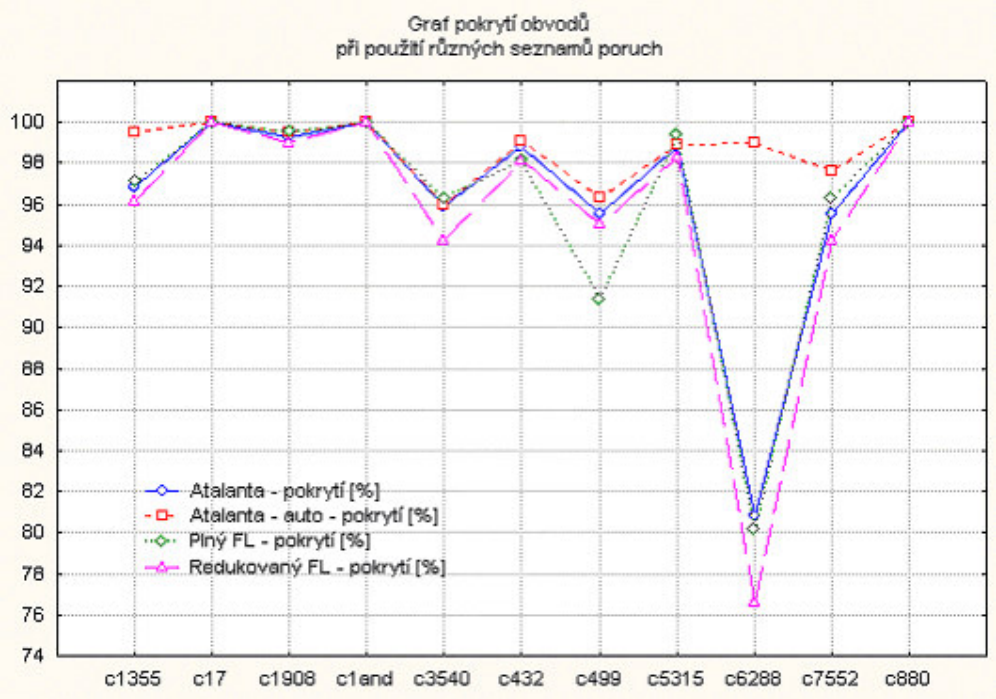
16->23 /1
1: x10x0 11
2: x1100 11

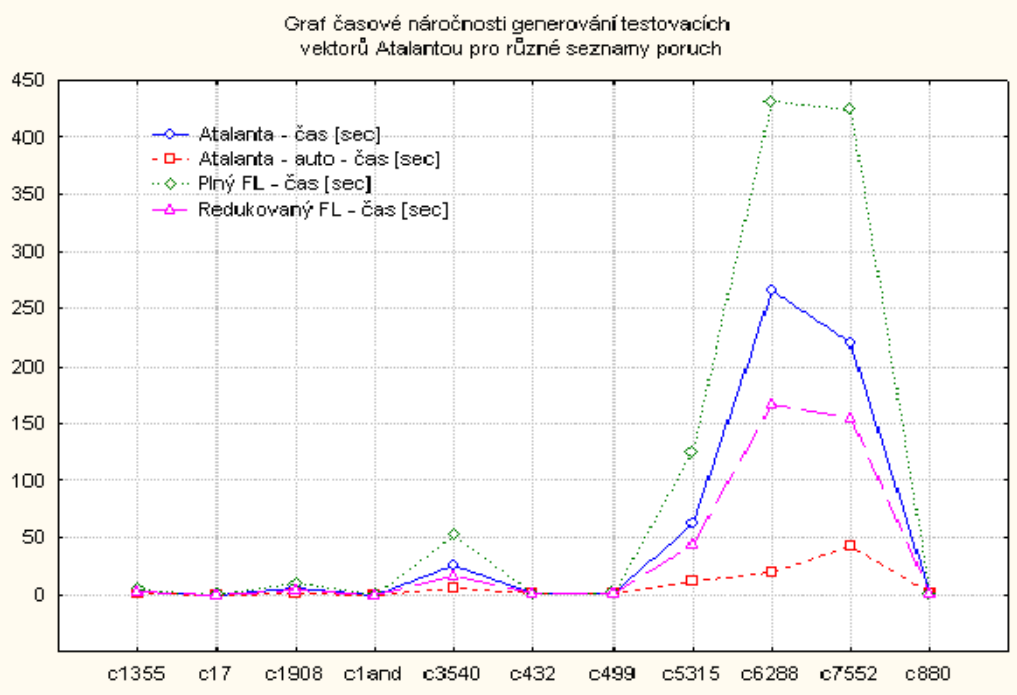
23 /0
1: x10xx 11
2: x110x 11

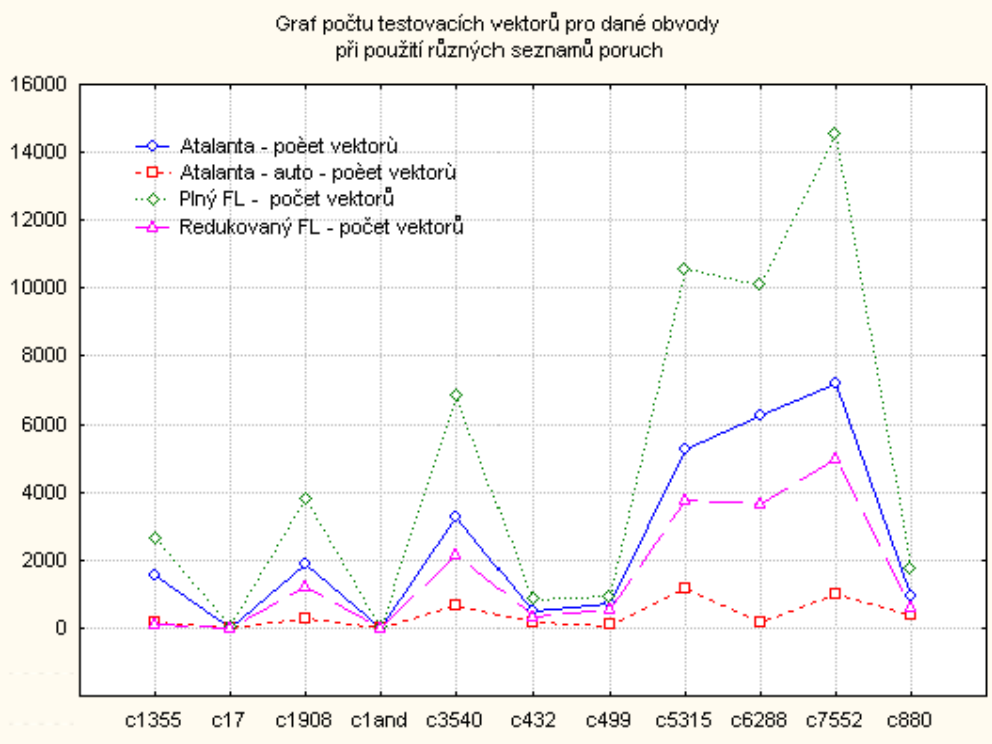
3:XX0X1 X1	3: x00x1 01	3: x00x1 01
4:XXX01 X1	4: x0101 x1	4: x0101 x1
g23 /1	g23 /1	23 /1
1:X011X X0	1: x0xx0 x0	1: x0xx0 x0
2:X0XX0 X0	2: x0111 x0	2: x0111 x0
3:XX11X X0	3: x111x x0	3: x111x x0
4:XX110 X0		
g16 /0	g16 /0	16 /0
1:00XXX 0X	1: 00xxx 0x	1: 00xxx 0x
2:X00XX 0X	2: 0111x 00	2: 0111x 00
3:X011X X0	3: 100xx 0x	3: 100xx 0x
4:X0XX0 X0	4: 101x0 10	4: 101x0 10
5:0X11X 0X	5: 10111 10	5: 10111 10
6:XX11X X0	6: 1111x 10	6: 1111x 10
7:XX110 X0		
g16 /1	g16 /1	16 /1
1:010XX 1X	1: 010xx 1x	1: 010xx 1x
2:X10XX 1X	2: 0110x 1x	2: 0110x 1x
3:X10X0 X1	3: 110xx 1x	3: 110xx 1x
4:01X0X 1X	4: 11100 11	4: 11100 11
5:X100X 1X		
6:X1X00 X1		
g16->g22 /0	g16->g22 /0	
1:00XXX 0X	1: 00xxx 0x	
2:X00XX 0X	2: 0111x 00	
3:0X11X 0X	3: 100xx 0x	
g16->g22 /1	g16->g22 /1	16->22 /1
1:010XX 1X	1: 010xx 11	1: 010xx 11
2:X10XX 1X	2: 0110x 11	2: 0110x 11
3:01X0X 1X	3: 110xx 11	3: 110xx 11
4:X100X 1X		
g1->g10 /0	g1->g10 /0	
1:101XX 1X	1: 101xx 1x	
2:1X11X 1X	2: 1111x 10	
g1->g10 /1	g1->g10 /1	1 /1
1:001XX 0X	1: 001xx 0x	1: 001xx 0x
2:0X11X 0X	2: 0111x 00	
g10->g22 /0	g10->g22 /0	
1:00XXX 0X	1: 00xxx 0x	
2:0X11X 0X	2: 0111x 00	
3:X00XX 0X	3: 100xx 0x	
g10->g22 /1	g10->g22 /1	10 /1
1:101XX 1X	1: 101xx 1x	1: 101xx 1x
2:1X11X 1X	2: 1111x 10	2: 1111x 10
g22 /0	g22 /0	22 /0

1:1X1XX 1X	1: 1x1xx 1x	1: 1x1xx 1x
2:X10XX 1X	2: 110xx 11	2: 110xx 11
3:X1X0X 1X	3: 010xx 11	3: 010xx 11
	4: 0110x 11	4: 0110x 11
g22 /1	g22 /1	22 /1
1:00XXX 0X	1: 00xxx 0x	1: 00xxx 0x
2:0X11X 0X	2: 0111x 00	2: 0111x 00
3:X00XX 0X	3: 100xx 0x	

6.6. Grafické porovnání různých seznamů poruch




Graf časové náročnosti generování testovacích vektorů Atalantou pro různé seznamy poruch
Atalanta - čas [sec]
Atalanta - auto - čas [sec]
Plný FL - čas [sec]
Redukovaný FL - čas [sec]
450
400
350
300
250
200
150
100
50
0
c1355
c17
c1908
c1and
c3540
c432
c499
c5315
c6288
c7552
c880


Graf počtu testovacích vektorů pro dané obvody při použití různých seznamů poruch
Atalanta - počet vektorů
Atalanta - auto - počet vektorů
Plný FL - počet vektorů
Redukovaný FL - počet vektorů
16000
14000
12000
10000
8000
6000
4000
2000
0
c1355
c17
c1908
c1and
c3540
c432
c499
c5315
c6288
c7552
c880

6.7. Seznamy poruch z dalších benchmarků

	Plný FL			Redukovaný FL			Atalanta - D		
obvod	čas [sec]	pokrytí [%]	poč. testů	čas [sec]	pokrytí [%]	poč. testů	čas [sec]	pokrytí [%]	poč. testů
s1196	0.733	100	1242	0.467	100	814	1.317	100	2392
s1238	0.75	94.908	1286	0.483	92.474	811	1.367	96.769	2396
s13207.1				0	nan	0	0	nan	0
s1423	1.817	98.482	1492	1.167	97.78	925	3.55	98.278	2797
s1488	1.717	100	1486	1.283	100	1025	3.717	100	2976
s1494	1.8	99.203	1494	1.25	98.844	1026	3.6	99.465	2972
s15850.1				0	nan	0	0.033	100	1
s208.1	0.017	100	217						
s208	0.017	100	215	0.017	100	128	0.05	100	416
s27	0	100	32	0.017	100	22	0	100	54
s298	0.033	100	308	0.017	100	222	0.067	100	596
s344	0.067	100	342	0.033	100	221	0.117	100	688
s349	0.05	99.429	348	0.017	99.115	224	0.1	99.427	694
s382	0.05	100	399	0.05	100	270	0.117	100	764

6.8. Parametry aplikace *muse*

Základní použití aplikace muse:

muse [options] circuit_file

options:
[-bc] [-f] [-y] [-1] [-g level] [-V maxvectors] [-k FL_file] [-m masks_file] [-v vector_file]

-b, -c	Výpis načteného obvodu od vstupů nebo výstupů
-f	Výpis seznamu poruch
-r	Redukovat seznam poruch
-V max	Číslo max určuje maximální počet vygenerovaných vektorů
-k soubor	Načte ze souboru seznam poruch
-m soubor	Načte ze souboru masky pro generování vektorů
-v soubor	Načte ze souboru testovací vektory
-1	Generování se provádí pro jednotlivé poruchy s postupným odstraňováním ze seznamu
-y	ATPG se neprovádí
-g level	Ladící úroveň se nastaví na úroveň level

Pozn.: Formát souboru pro masky je shodný s formátem pro testovací vektory. Společně s formátem pro soubor se seznamem poruch jsou uvedeny v příloze, v kapitolách 6.1 a 6.2