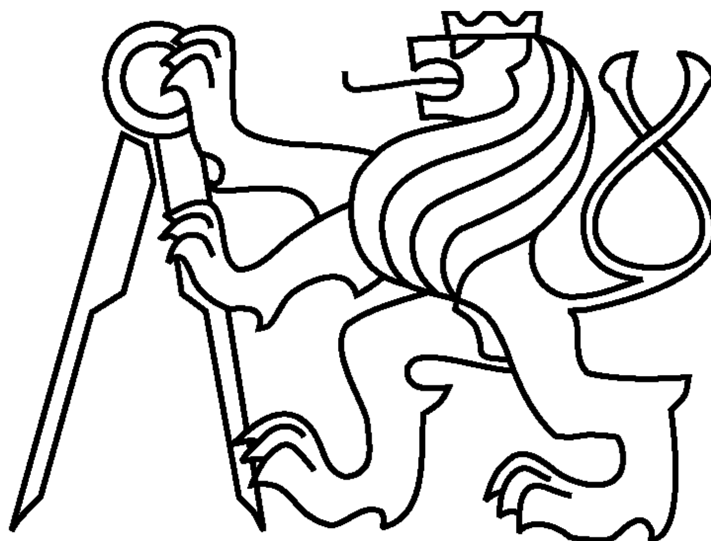
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Katedra počítačů



Diplomová práce :

Generátor komprimovaných testovacích vektorů založený na modifikaci ATPG

Bc. Jiří Balcárek

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika, strukturovaný, Navazující magisterský

Obor: Výpočetní technika

17. května 2009

Poděkování

Mé poděkování patří Ing. Petru Fišerovi za všechen čas, který mi věnoval, za jeho trpělivost a za připomínky a návrhy, které vedly k realizaci a zkvalitnění této práce. Dále bych chtěl poděkovat rodičům, za podporu, a trpělivost nejen při psaní této práce, ale také po dobu celého studia.

Čestné prohlášení

Prohlašuji, že jsem zadanou diplomovou práci zpracoval sám s přispěním vedoucího práce a používal jsem pouze literaturu uvedenou v přiloženém seznamu.
Dále prohlašuji, že nemám námitek proti užití tohoto školního díla ve smyslu §60 Zákona č.121/200 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 17.5.2009 ....

podpis

Abstrakt

Tato práce se zabývá implementací generátoru testovacích vektorů založeného na modifikaci ATPG (Automatic Test Pattern Generator). Tento nástroj používá ke generování komprimovaného testu nový algoritmus založený na řešení SAT problému. Nástroj provádí kompresi vhodným překrytím testovacích vektorů, které si sám generuje tak, aby bylo dosaženo maximální komprese. Hlavní výhodou tohoto nástroje je schopnost generovat vektory zaručující nejlepší překrytí a tím maximalizovat kompresi. Takto vytvořený komprimovaný test lze použít při testování kombinačních nebo full-scan obvodů. Nástroj je otestován na množině standardních úloh a výsledky jsou srovnány s nástrojem COMPAS (COMpressed Pattern Sequencer), který realizuje kompresi pomocí přerovnávání, kompakce a hledání nejlepšího překrytí předem vygenerovaného testu.

Abstract

This thesis deals with an implementation of a generator of compressed test vectors based on a modification of an ATPG (Automatic Test Pattern Generator). This software tool uses for generation of a compressed test a new algorithm based on SAT (satisfiability) solving. The tool performs the test compression by overlapping of test vectors, which are generated on-line in the process, to reach a maximum compression. The main advantage of this tool is the ability to generate vectors guaranteeing the best overlap and by this maximize the test compression. Test compressed by this way may be used for testing combinational or full-scan circuits. Our tool is tested on a set of benchmarks and the results are compared with the COMPAS (COMpressed Pattern Sequencer) compression tool, which implements a patterns compression based on reordering, compaction and searching for the best pattern overlapping the previously generated test.

Obsah

Poděkování . . . V
Čestné prohlášení . . . VII
Abstrakt . . . IX
Obsah . . . XI
Seznam obrázků . . . XIII
Seznam tabulek . . . XV

1. Úvod . . . 1
2. Motivace . . . 3
3. Základní pojmy . . . 4
  3.1. Úvod do názvosloví . . . 4
  3.2. Model poruch . . . 4
    3.2.1. Model poruchy Stuck-at . . . 5
  3.3. Seznam poruch . . . 5
  3.4. Druhy poruch . . . 5
  3.5. Typy testů . . . 6
  3.6. Ekvivalence a dominance poruchy . . . 6
  3.7. Checkpoint teorém . . . 7
  3.8. Konjunktivní normální forma CNF . . . 7
4. Testování číslicových obvodů . . . 9
  4.1. Metody testování . . . 9
    4.1.1. Sériové metody – SAS . . . 10
    4.1.2. Paralelní metody – RAS . . . 11
  4.2. Testovací architektury umožňující dekompresi testu . . . 13
    4.2.1. RESPIN architektura . . . 13
    4.2.2. Vestavěná diagnostika – BIST . . . 17
5. SAT-ATPG . . . 21
  5.1. Problém SAT . . . 21
  5.2. Davis & Putnam algoritmus . . . 21
  5.3. Metody generování testu pomocí ATPG . . . 22
    5.3.1. Strukturální metoda . . . 22
    5.3.2. Algebraická metoda . . . 24
  5.4. Transformace kombinační logiky do CNF . . . 24

5.4.1. Booleovská diference 25
5.4.2. Tsietsienovy transformace 25
5.5. SAT-solver 29
6. Atalanta-M 30
6.1. Simulace poruch 30
7. Návrh a realizace 31
7.1. Specifikace problému 31
7.2. Analýza problému 32
7.3. Popis základního algoritmu 34
7.4. Integrace součástí 37
7.5. Možná vylepšení základního algoritmu 38
7.5.1. Řazení poruch 38
7.5.2. Přidání DC stavů 40
7.5.3. Jednoduchá “Pre-simulace“ 41
7.5.4. BDD a možnosti využití 44
8. Nástroj COMPAS 46
8.1. Komprese 46
9. Testování a srovnání výsledků 48
9.1. Základní algoritmus 48
9.2. Základní algoritmus rozšířen o DC 50
9.3. Základní algoritmus rozšířen o DC a řazení poruch 51
9.3.1. Řazení poruch “max-min“ 53
9.3.2. Řazení poruch “min-max“ 54
9.4. Základní algoritmus rozšířen o řazení poruch a “pre-simulaci“ 56
9.5. Souhrnné výsledky 58
10. Závěr 61
11. Literatura 62
12. Seznam zkratek 63
13. Přílohy 65
13.1. Uživatelská příručka 65
13.2. Formát souboru .bench 66
13.3. Obsah CD 67

Seznam obrázků

Obrázek č.1 : *Základní schéma testu* **9**
Obrázek č.2: *Příklad skenovací buňky MD-FF [6]* **10**
Obrázek č.3: *Struktura "obecného" scan registru [6]* **11**
Obrázek č.4: *Paralelní (vlevo) vs. Sériová (vpravo) organizace skenovacích řetězců [7]* . . **11**
Obrázek č.5: *Struktura obvodu využívajícího RAS [6]* **12**
Obrázek č.6: ***A)*** *Standardní testovací architektura,* ***B)*** *Testovací architektura RESPIN [7]* **14**
Obrázek č.7: *ETC architektura pro více scan chainů [7]* **14**
Obrázek č.8: *Příklad testovací architektury ETC+CUT jeden [1] / více scan chainů [7]* . . . **15**
Obrázek č.9: *Tři deterministicky generované vzorky A, B a C v scan chainech CUT [7]* . . . **16**
Obrázek č.10: *Simulace průběhu testu v ETC a CUT [10]* **16**
Obrázek č.11: *Obecná bloková struktura BIST* **18**
Obrázek č.12: *Graf pokrytí pomocí pseudonáhodného TPG [4]* **19**
Obrázek č.13: *Obvod s poruchou Sa0 na vstupu hradla E [2]* **23**
Obrázek č.14: *Ukázka transformace základních hradel do CNF [10]* **26**
Obrázek č.15: *Kombinační obvod s hradly popsanými klauzulemi v CNF [10]* **27**
Obrázek č.16: *Kombinační obvod s poruchou Sa1 na D [10]* **28**
Obrázek č.17: *XOR obvodu s poruchou a bez poruchy [10]* **29**
Obrázek č.18: *Příklad komprese testu metodou překrytu vektorů* **32**
Obrázek č.19: *Příklad posunutí vektoru* **34**
Obrázek č.20: *Základní algoritmus našeho nástroje* **36**
Obrázek č.21: *Blokový diagram spolupráce modulů* **38**
Obrázek č.22: *Příklad bitů, které lze testovat na DC* **41**
Obrázek č.23: *Příklad práce s testovacími vektory při použití pre-simulace* **42**
Obrázek č.24: *BDD a jemu odpovídající pravdivostní tabulka [5]* **44**
Obrázek č.25: *Ukázka ROBDD vzniklého odstraněním redundancí [5]* **45**
Obrázek č.26: *Histogram pro základní algoritmus* **49**
Obrázek č.27: *Histogram pro základní algoritmus + přidávání DC* **51**
Obrázek č.28: *Histogram pro základní algoritmus + řazení Max-min + DC* **54**
Obrázek č.29: *Histogram pro základní algoritmus + řazení Min-max + DC* **56**
Obrázek č.30: *Histogram pro základní algoritmus + Max-min + pre-simulace* **58**
Obrázek č.31: *Adresářová struktura přiloženého CD* **67**

Seznam tabulek

Tabulka č.1 *: Pravdivostní tabulka hradla AND, poruchy a příklad třídy ekvivalence* **7**
Tabulka č.2 *: Pracovní módy ETC a CUT [7]* **15**
Tabulka č.3 *: Výsledky testování základního algoritmu* **49**
Tabulka č.4 *: Výsledky testování základního algoritmu + přidávání DC* **50**
Tabulka č.5 *: Výsledky testování základního algoritmu + řazení Max-min + DC* **53**
Tabulka č.6 *: Výsledky testování základního algoritmu + řazení Min-max+ DC* **55**
Tabulka č.7 *: Výsledky testování základního algoritmu + Max-min+ pre-simulace* **57**
Tabulka č.8 *: Výsledky souhrnného testování (počáteční stav daný poruchou č. 0)* **59**

1. Úvod

Při návrhu číslicových obvodů a SoC (System-on-a-Chip) se setkáváme s mnoha omezeními a překážkami, přičemž jednou z nich je také jejich testování. V dnešní době již není problematické navrhnout velmi složitý číslicový obvod, ale hlavním problémem je, jak otestovat jeho bezporuchovost. S rostoucí velikostí návrhu roste exponenciálně také velikost testu (úplného), což vede k potřebě větší paměti, zvyšuje se doba nutná k otestování návrhu a spotřeba energie. V praxi se proto využívají různé metody komprese testu a kombinace deterministického a pseudonáhodného testování poruch. Pseudonáhodným testem založeným na generování testovacího vektoru, např. rotací vhodného počátečního vektoru, pokryjeme "lehce detekovatelné" poruchy a zbylé pokryjeme vzorky vygenerovanými deterministickým generátorem testovacích vzorků. Výhodou tohoto je jednoduchá implementace pseudonáhodného generátoru testovacích vzorků přímo na čipu, což zjednoduší a zrychlí testování. Spotřeba paměti je minimální a zvyšuje se i rychlost (frekvence), nicméně vzhledem k pseudonáhodnému chování je nutné otestovat desítky tisíc vektorů, z nichž velká část nepokrývá žádnou poruchu, což způsobuje zvýšení energetické náročnosti testu. Doba testu a počet vektorů, které je nutno vygenerovat jsou velmi závislé na vhodné volbě prvního testovacího vektoru/ů. Mohlo by tedy být velmi užitečné vytvoření deterministického nástroje generujícího, komprimovaný test schopný otestovat návrh v co nejkratším čase, s co nejmenšími paměťovými a energetickými požadavky. Taktéž je třeba brát v úvahu složitost hardwaru na dekompresi testu v obvodu.

Tento nelehký úkol se dá více či méně efektivně řešit mnoha přístupy, přičemž jedním z velmi slibných je komprese testu kompakcí a hledáním nejlepšího překrytí vstupních vektorů, kterou představuje nástroj COMPAS(COMpressed PAttern Sequencer) [1]. Tento nástroj používá jako vstup seznam poruch společně s vektory, které je pokrývají, a pomocí sofistikovaných algoritmů se snaží najít nejlepší překryv těchto vektorů a vytváří tak komprimovaný řetězec bitů (bitstream). Kromě vysokého stupně komprese, je další výhodou tohoto přístupu jednoduchá dekomprese testovacích vzorků, což znamená, že je nutný pouze minimální zásah do testovaného hardwaru. Ve srovnání s ostatními nástroji řešícími tento problém, dosahuje velmi dobrých výsledků, při poměrně krátké době řešení, nicméně tyto jsou do značné míry závislé na vstupních vektorech, které by měli obsahovat co nejvíce tzv. DC (Don't Care) bitů. Na těchto bitech pro otestování dané poruchy nezáleží a mohou být tedy dále nastaveny podle potřeby na logickou hodnotu 0 resp. 1.

Cílem mojí práce je implementovat a otestovat nový přístup k řešení problému komprese testovacích vektorů založený na řešení splnitelnosti booleovské formule (SAT) a hledání nejlepšího překrytí testovacích vektorů. Na rozdíl od nástroje COMPAS, který vytváří sekvenci bitů testu tak, že hledá nejlepší překryv testovacích vektorů, které dostane jako vstup, se v této práci snažíme vytvářet testovací sekvenci vhodným generováním "navazujících" testovacích vektorů. Výstupem tohoto nástroje je tedy co nejoptimálnější řetězec bitů, s jehož pomocí jsme schopni otestovat všechny detekovatelné poruchy v daném obvodu.

Nástroj se skládá ze dvou částí, které spolupracují při hledání řešení. V první části se realizuje vygenerování instance SAT problému, jehož množinou řešení jsou vektory schopné detekovat konkrétní poruchu v obvodu. Tato část je tvořena automatickým generátorem testovacích vektorů (ATPG – Automatic Test Pattern Generator) založeném na splnitelnosti booleovské formule, který v rámci diplomové práce implementoval Jiří Červák [2]. Tento nástroj umožňuje mimo jiné vygenerovat pro konkrétní poruchu instanci SAT problému ve formátu DIMACS. Druhá část je tvořena simulátorem poruch, pomocí kterého testujeme, kolik poruch pokrývá vektor řešení nalezený v první části. Jako simulátor poruch použijeme upravený nástroj Atalanta, který v rámci bakalářské práce implementoval Radovan Myslík[3].

Dále je provedeno testování tohoto nástroje v jeho základní podobě a posléze s aplikací několika jednoduchých heuristik, pomocí kterých se snažíme docílit lepší komprese. Výsledky jsou srovnávány s referenčním nástrojem COMPAS [1].

Nežli začnu popisovat způsob implementace a jednotlivé heuristiky, kterými jsme se snažili zlepšit dosažený výsledek, rád bych nejprve provedl krátký teoretický úvod do problematiky týkající se testování a provedl seznámení s používanými nástroji a technikami.

2. Motivace

Při volbě tématu jsme se s vedoucím práce snažili nalézt prakticky zajímavý problém, který bychom se mohli pokusit řešit, a zároveň takový, kde by bylo možné využít dříve získaných znalostí v oblasti řešení SAT problému. Původní záměr počítal s návrhem SAT řešiče, který by byl schopen jako výsledek generovat vektory s co největším počtem DC bitů. Tyto řídké vektory jsou, jak již bylo dříve řečeno, velmi cenným vstupem pro nástroj COMPAS. Vzhledem k tomu bylo tedy nutné se s tímto kompresním nástrojem blíže seznámit. Po bližším obeznámení s principy, na kterých je COMPAS založen, jsme došli k závěru, že ačkoli je tato metoda poměrně efektivní a používá velmi sofistikované algoritmy, jejím hlavním nedostatkem je, že zde "pouze" rovnáme (překrýváme) sadu testovacích vektorů, které obdržíme na výstupu ATPG. Tyto vektory přitom mohou být naprosto nevhodné pro případné překrytí, popř. kompakci. Pokud si uvědomíme, že množinu všech testovacích vektorů poruchy lze reprezentovat pomocí instance SAT problému, musíme se logicky zeptat, jestli by nebylo z hlediska komprese výhodnější, vybírat navazující testovací vektor z této množiny, než se snažit najít nejlepší překrytí z určitého předdefinovaného testu ("náhodně" vybraná množina vektorů). Na základě teoretických předpokladů jsme tedy vymysleli nový jednoduchý algoritmus, který by měl být schopen dosahovat minimálně takové komprese jako COMPAS. Tento nový algoritmus na rozdíl od ostatních nekomprimuje předem vygenerovaný test, ale projde SAT instance jednotlivých poruch a z jejich množin řešení vybere vektor tak, aby bylo dosaženo maximálního překrytu a tudíž maximální komprese. Samozřejmě se zde dá očekávat vyšší časová složitost, která je dána nutností řešit velké množství SAT instancí, nicméně naším hlavním cílem je zde maximální komprese, které bychom tímto způsobem měli dosáhnout. Abychom si potvrdili tyto teoretické předpoklady, implementoval jsem náš základní algoritmus a několik jeho vylepšení, a provedl sadu měření, která by nám měla ukázat jakým způsobem se tento náš nový přístup promítne na kompresi testu. Výsledky jsou následně srovnány s nástrojem COMPAS. Jako prostředky k realizaci algoritmu mi posloužili modifikovaný simulátor poruch Atalanta-M a SAT-ATPG, impelmentovaný kolegou Červákem. Abych byl schopen algoritmus implementovat, musel jsem tyto nástroje rozšířit o množství funkcí zajišťujících jejich spolupráci a realizujících nové operace nutné k řešení problému. Bližší popis implementace společně s rozborem problému a následné zhodnocení výsledků je popsáno v následujících kapitolách.

3. Základní pojmy

3.1. Úvod do názvosloví

Pro náš další výklad si definujeme několik pojmů, se kterými budeme dále pracovat. Prvním z nich je **testovaná jednotka - CUT (Circuit Under Test)**, která pro nás představuje číslicový obvod nebo jeho část, jenž se snažíme testovat. Tento obvod obsahuje vstupy a výstupy, pomocí kterých komunikuje se svým okolím. Takovéto vstupy a výstupy budeme dále označovat jako **primární vstupy a výstupy**. Při testování se snažíme zjistit **technický stav** testované jednotky, tj. soubor hodnot charakterizujících podstatné vlastnosti funkční části testované jednotky v určitém časovém okamžiku, a na základě toho, jak je nebo není testovaná jednotka schopna plnit předepsanou funkci, definujeme dva základní stavy, a to **poruchový** a **bezporuchový**. Jako poruchový budeme označovat stav, kdy se v testované jednotce vyskytla alespoň jedna porucha. **Porucha** je dle normy definována jako jev spočívající v ukončení schopnosti objektu plnit požadovanou funkci podle technických podmínek. Projevem poruchy může být **chyba**, což je rozdíl mezi správnou a skutečnou hodnotou testovaného výstupu. Z definice vyplývá, že chyba je vždy důsledkem nějaké poruchy, nicméně každá porucha se nemusí projevit jako chyba (např. nepoužívá-li se při provádění funkce žádná ze součástek). Rozlišení poruchového a bezporuchového stavu **nazýváme detekce poruchy** a nalezení konkrétní poruchy označujeme jako **lokalizaci poruchy**.

Test číslicového systému je množina dvojic vzájemně přiřazených vstupních a výstupních vektorů. Jeden vstupní vektor a jemu odpovídající výstupní vektor (odezva) se nazývá **krokem testu**. Počet kroků udává **délku testu**. Důležitým parametrem testu je jeho **pokrytí**, které vyjadřuje počet poruch, detekovaných (tj. pokrytých) testem. V případě, že jsou detekovány všechny poruchy, jedná se o **úplný test,** a jestliže tento má nejmenší délku, nazýváme ho **minimální test**.

Při definování pojmů bylo čerpáno ze skript[4].

3.2. Model poruch

Poruchy v číslicových obvodech mohou mít různou fyzikální příčinu. Z hlediska ověření správné funkce obvodu nás nezajímá příčina, ale její projev – chyba tzn. zajímá nás jakým způsobem se určitá porucha při daném vstupu projeví na výstupních hodnotách. Bez znalosti

topologie obvodu by úplný test obvodu s *n* vstupy zahrnoval otestování 2^n možných ohodnocení vstupů. Je zřejmé, že provádět úplný test je pro větší počet vstupů nerealizovatelné a snažíme se tomuto naivnímu přístupu vyhnout použitím vhodného modelu poruch.

3.2.1. Model poruchy Stuck-at

Tento model poruchy je jedním z nejstarších a nejpoužívanějších. Rozlišujeme zde dvě poruchy Stuck-at 0 (Sa0) a Stuck-at 1 (Sa1). Poruchou Stuck-at zde rozumíme poruchu typu trvalá nula/jedna na vodiči. Porucha typu trvalá nula se projevuje jako konstantní hodnota “0“ na určitém vodiči. Obdobně trvalá jednička se projevuje jako konstantní hodnota “1“ na určitém vodiči. Prakticky může vzniknout např. jako zkrat vodiče na napájecí napětí, přerušení vstupního vodiče v TTL, přerušení napájecího/zemnícího vodiče v TTL atd.. Vodičem rozumíme spojení mezi dvěma hradly, nebo hradlem a větvením. Tento model předpokládá, že v obvodu se vyskytuje vždy **pouze jedna porucha** a to buď **Sa0** nebo **Sa1** [4].

3.3. Seznam poruch

Seznam poruch obsahuje všechny poruchy, které v daném modelu mohou nastat. Pro námi používaný model poruch Stuck-at lze celkový počet poruch vyčíslit jako [2]:

$$N_p = 2 * (N_i + N_h + N_v)$$

Kde : N_p – celkový počet možných Sa poruch
N_i – počet vstupů
N_h – počet hradel
N_v – počet větvení

3.4. Druhy poruch

Při generování testu pomocí ATPG může dojít k situaci, kdy pro určitou poruchu, popř. skupinu poruch, není ATPG schopen vytvořit testovací vektor. Z tohoto pohledu můžeme tedy poruchy klasifikovat do dvou skupin a to :

- ***Testovatelné*** – pro tyto je ATPG schopen sestavit testovací vektor
- ***Netestovatelné (Redundantní)*** – porucha, na kterou ***nelze*** sestavit testovací vektor

Redundantní porucha vzniká přidáním redundantního prvku do obvodu. Jako redundantní prvek označujeme součástku, kterou lze ze správně fungujícího obvodu vyjmout a nahradit zdrojem konstantního signálu 0 nebo 1, aniž by se tím změnila výstupní funkce obvodu. **Test redundantního obvodu nemůže být nikdy úplný**. Při generování testu je tedy důležité kromě testovacích vektorů vypsat netestovatelné poruchy.

3.5. Typy testů

V praxi lze pro generování testů použít velké množství algoritmů, které se ale většinou liší pouze dosaženými výsledky a dobou výpočtu. Z metodického hlediska lze testy a algoritmy používané k jejich sestavení rozdělit do dvou skupin, a to na **strukturní** testy a **funkční** testy.

- **Strukturní testy** – vychází ze znalosti “struktury“ (topologie) obvodu. Příprava strukturního testu vyžaduje podrobnou analýzu testované jednotky, vytvoření modelu všech poruch, které se v testované jednotce mohou vyskytovat a odvození kroku nebo skupiny kroků testu pro každou poruchu. Při generování těchto testů se používá téměř výhradně metoda zcitlivění cesty (např. D-algoritmus, PODEM – Path Oriented DEcision Making, FAN) [4].

- **Funkční testy -** vycházíme pouze z popisu funkce testované jednotky, takže podrobná znalost není nutná. Je třeba vytvořit model poruch chování na stejné úrovni, na jaké je popsána funkce, protože bez znalosti struktury nelze vycházet z modelu fyzikálních poruch [4].

V této práci se snažíme realizovat ATPG komprimovaných vektorů na základě struktury obvodu.

3.6. Ekvivalence a dominance poruch

Jak již bylo dříve řečeno, může vygenerovaný test obsahovat velké množství vektorů, což je z hlediska detekce poruchy poměrně nepříjemné. Jednou z možností, jak snížit počet kroků testu, je například pomocí ekvivalence a dominance poruch.

- **Ekvivalence** – poruchy f a f1 jsou ekvivalentní, právě když mají shodné sady vektorů, které je detekují. Pokud se tedy v testu vyskytují ekvivalentní poruchy, lze tento test rozdělit do tzv. **tříd ekvivalence** a pro každou třídu vygenerovat testovací vektor, který pokrývá její poruchy. Následující tabulka ukazuje

pravdivostní tabulku hradla AND se vstupy A,B a výstupem Y. Ve sloupci porucha jsou vypsány chybné odezvy pro všechny poruchy, které mohou pro toto hradlo nastat. Šedý podklad zvýrazňuje třídu ekvivalence poruch A/0, B/0, C/0 (poruchy Stuck-at 0). Z příkladu je také patrné, že poruchy v určité třídě ekvivalence jsou mezi sebou vzájemně **nerozlišitelné** [4].

			Porucha					
A	B	Y	A/0	B/0	C/0	A/1	B/1	Y/1
0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	1	0	1
1	0	0	0	0	0	0	1	1
1	1	1	0	0	0	1	1	1

. ***Tabulka č.1*** *: Pravdivostní tabulka hradla AND, poruchy a příklad třídy ekvivalence*

- **Dominance** – porucha f dominuje poruše f1, jestliže všechny vektory detekující f1 detekují také f. Z úvodní definice tedy vyplývá, že lze vyjmout dominující poruchu za předpokladu, že dominovaná porucha je testovatelná. V opačném případě se vypuštěním dominující poruchy můžeme dopustit chyby, protože dominující porucha může být testovatelná [4].

3.7. Checkpoint teorém

Primární vstupy a všechna následující větvení kombinačního obvodu označíme jako tzv. checkpointy (kontrolní body).

- **Checkpoint teorém** - sada vektorů, která detekuje všechny Stuck-at poruchy na všech checkpointech, detekuje také všechny Stuck-at poruchy v obvodě.

Teorém tedy říká, že pro kompletní pokrytí poruch v celém obvodu stačí do seznamu poruch přidat poruchy na všech primárních vstupech a na všech vstupech hradel bezprostředně za větvením. Po inicializaci seznamu poruch pomocí teorému lze dále tento seznam redukovat pomocí vztahů ekvivalence a dominance [2].

3.8. Konjunktivní normální forma – CNF

Výroková formule je v konjunktivní normální formě, resp. v klauzulární formě, je-li konjunkcí podformulí, z nichž každá je disjunkcí konečně mnoha literálů výrokových proměnných. Úplná konjunktivní normální forma formule ***A*** je její konjunktivní normální forma, v níž každý konjunkt obsahuje literály všech výrokových proměnných vyskytujících se

ve formuli ***A***, přičemž se v žádném konjunktu nevyskytuje současně pozitivní a negativní literál téže výrokové proměnné. Literál je logická proměnná nebo negace logické proměnné.

Tuto definici lze zapsat jako obecný výraz takto :

$$(\alpha_{11} \vee \alpha_{12} \vee \ \dots \vee \alpha_{1m}) \ \wedge (\alpha_{21} \vee \alpha_{22} \vee \dots \vee \alpha_{2m}) \ \wedge \dots \wedge (\alpha_{n1} \vee \alpha_{n2} \vee \ \dots \vee \alpha_{nm})$$

kde každé α_{nm} je logická proměnná nebo negace logické proměnné (literál).

$$\text{Př. } (a \vee b \vee \neg c) \ \wedge (\neg a \vee b \vee c) \ \wedge (\neg a \vee b \vee \neg c)$$

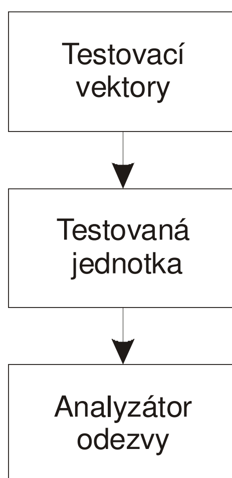
- **Tautologie** je formule pravdivá ve všech pravdivostních ohodnoceních (např. $\alpha \Leftrightarrow \alpha$).
- **Kontradikce** je formule nepravdivá ve všech pravdivostních ohodnoceních (např. $\alpha \wedge \neg \alpha$).
- **Splnitelná formule** je taková formule, která je pravdivá alespoň v jednom pravdivostním ohodnocení (např. $\alpha \wedge$ ß).

Převzato z [5].

4. Testování číslicových obvodů

4.1. Metody testování

Jak již bylo dříve řečeno, při testování vycházíme ze základního obecného schématu viz. obrázek č.1. Na vstupu testované jednotky máme sadu testovacích vektorů, které pokrývají detekovatelné poruchy testované jednotky a na výstupech testujeme odezvy na jednotlivé vektory. Tento základní koncept lze ovšem v praxi aplikovat mnoha způsoby, přičemž nyní si uvedeme nejjednodušší rozdělení a popis jednotlivých přístupů.



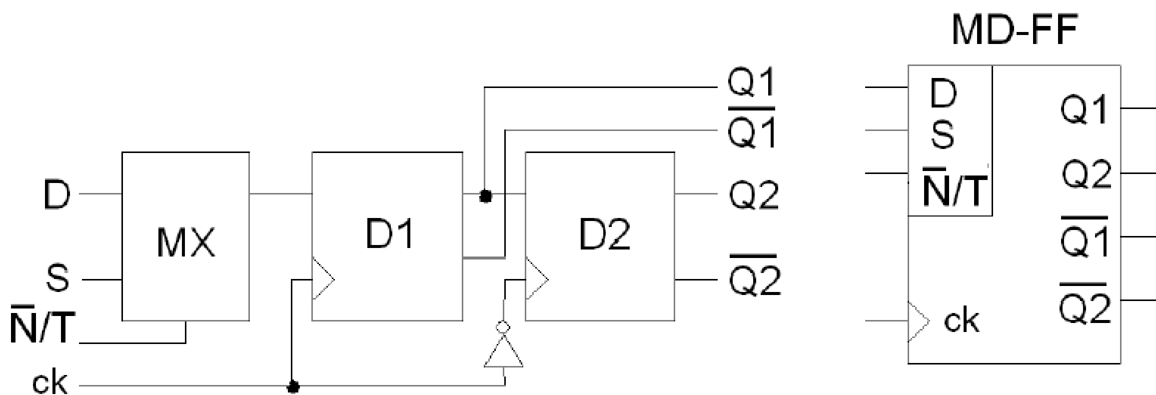
Obrázek č.1 : *Základní schéma testu*

Z hlediska testování je velmi oblíbeným prostředkem tzv. **scan technika**. Tato technika je založena na vložení paměťových buněk (tzv. scan buněk) do struktury obvodu, přičemž tyto buňky nemají v běžném provozu vliv na funkci obvodu, nicméně v diagnostickém režimu je možné pomocí těchto buněk přivádět testovací vzorky na konkrétní vstupy, resp. číst odezvy z konkrétních výstupů. Obvykle přitom rozlišujeme dvě varianty techniky scan, lišící se pouze ve způsobu přístupu k paměťovým prvkům vybraným pro test obvodu (nebo jeho části). První technika SAS (sériový scan) je založena na řetězení scan buňek do posuvných (tzv. scan) registrů, umožňujících sériové vkládání resp. sledování stavu scan buňek zařazených do tohoto registru. Druhou technikou je RAS, založená na přímé adresaci scan buňek. Obecně je možné a někdy také velmi výhodné, testovat obvod pomocí několika scan registrů, popř. jejich zřetězení.

4.1.1. Sériové metody - SAS

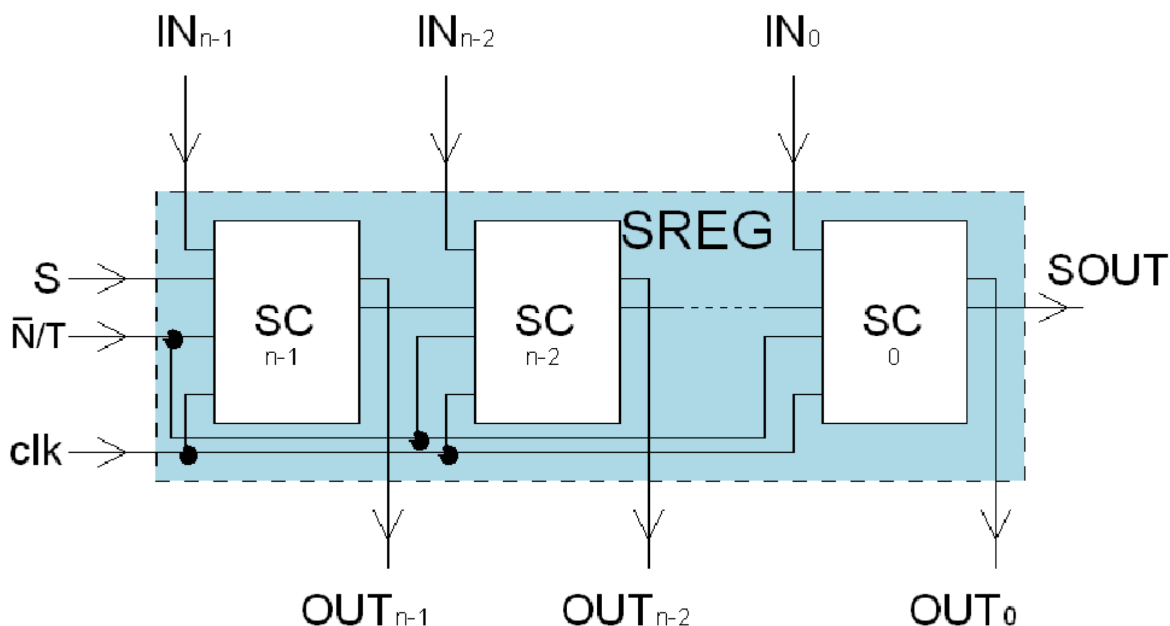
Tato metoda je založena na rozčlenění obvodu do kombinačních částí, oddělených paměťovými prvky vytvářejícími posuvný registr, který v režimu testu (diagnostický režim) umožňuje sériový přenos diagnostických dat z primárního vstupu obvodu, resp. na primární výstup obvodu. Vzhledem k tomu, že paměťové buňky tvoří posuvný registr, dojde při čtení odezvy testovaného obvodu k přepsání vektoru testu v paměťových buňkách. V dalším kroku testu je tedy nutné vysunout odezvy z paměťového řetězce, analyzovat je a zároveň nasunout nový testovací vektor.

Ačkoli možných realizací paměťové buňky je mnoho, ukážeme si zde pouze jeden příklad a tím je paměťová buňka MD-FF. Její schéma zapojení je vidět na obrázku č.2. Skládá se ze dvou klopných obvodů typu D, které v našem případě představují bloky D1 a D2 a multiplexeru, který představuje blok MX. Dále obsahuje vstup *S* zajišťující vstup dat v diagnostickém režimu a vstup *D* zajišťující vstup dat v normálním režimu. Vstupem *N/T* provádíme výběr mezi diagnostickým a normálním režimem. Výstup *Q1* představuje výstup v normálním režimu, zatímco *Q2* v diagnostickém. Vstup *ck* zde představuje hodiny.



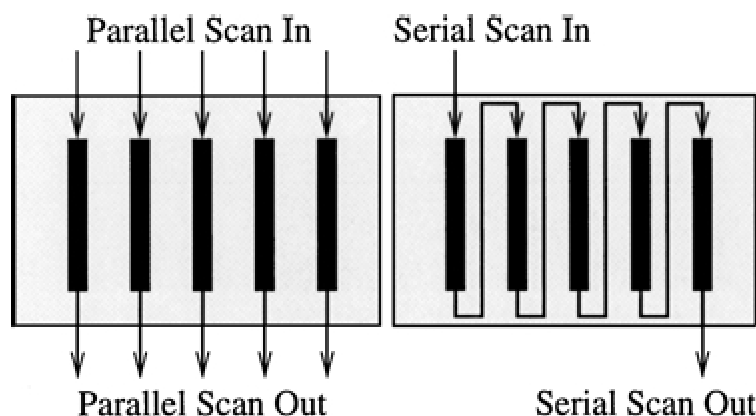
Obrázek č.2: *Příklad skenovací buňky MD-FF [6]*

Z takto, resp. podobně vytvořených paměťových buněk vytváříme poté vlastní skenovací registr. Jeho obecná struktura je naznačena na obrázku č. 3. Paměťové buňky skenovacího registru zde představují bloky SC. Dále zde máme vstupy registru reprezentované signály *IN*, výstupy reprezentovány signály *OUT* a sériový vstup testovacích dat *S,* resp. výstup *SOUT*. Vstupem *N/T* volíme mód, ve kterém registr pracuje tj. diagnostický/normální. Synchronizace buněk je zajištěna společným hodinovým signálem *clk*.



Obrázek č.3: *Struktura "obecného" scan registru [6]*

Obecně lze na testování čipu použít několik sériových skenovacích řetězců. Následující obrázek č.4 naznačuje vlevo strukturu čipu se skenovacími řetězci řazenými paralelně resp. vpravo se skenovacími řetězci zřetězenými do série.



Obrázek č.4: *Paralelní (vlevo) vs. Sériová (vpravo) organizace skenovacích řetězců [7]*

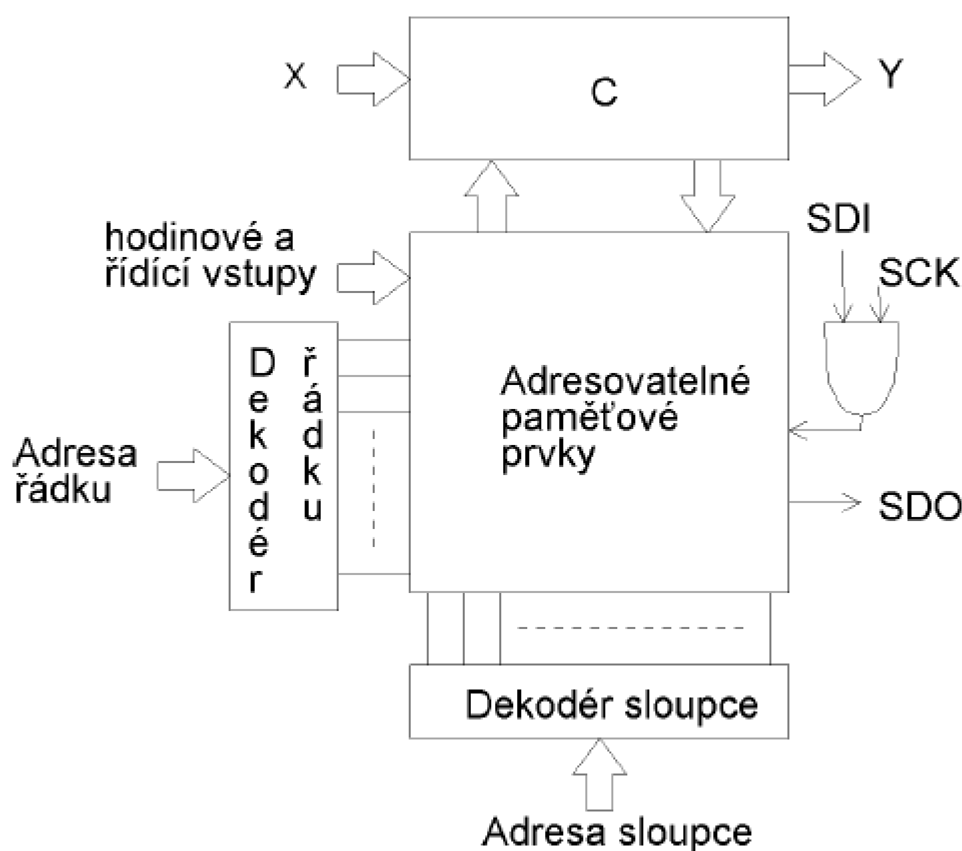
4.1.2. Paralelní metody - RAS

Další používanou metodou testování je RAS (Random Acces Scan). Tato metoda je podobná SAS, jelikož taktéž používá paměťové buňky k stimulování resp. snímání testovaných signálů, nicméně na rozdíl od SAS, zde nejsou paměťové buňky řazeny tak, aby utvářely posuvný registr, ale jsou realizovány jako adresovatelné buňky paměti s náhodným přístupem. Toto schéma je patrné z obrázku č.5. Stejně jako u SAS mohou paměťové buňky pracovat v režimu normálním a diagnostickém, ale na rozdíl od SAS je zde možné nastavit

obsah konkrétní paměťové buňky, aniž by docházelo k destrukci hodnot uložených v ostatních paměťových buňkách. Totéž platí také pro čtení obsahu paměťové buňky.

Jak bylo dříve řečeno, plní paměťové buňky v normálním režimu normální funkci jako v případě SAS. V diagnostickém režimu je adresován paměťový prvek, se kterým chceme pracovat, a pomocí datového vstupu SDI (Serial Data In) a hodinového vstupu SCK (Seriál ClocK) je zapsána nová hodnota popř. je čtena hodnota na výstupu SDO (Serial Data Out).

Na blokovém schématu na obrázku č.5 je názorně vidět, že k adresaci paměťových buněk je použito dvou dekodérů. Testovaný obvod je zde označován jako C (Circuit). Primární vstupy jsou označovány X a primární výstupy značíme Y.



Obrázek č.5: *Struktura obvodu využívajícího RAS [5]*

Z předchozího popisu vyplývá, že metoda RAS je pro testování obvodů z hlediska rychlosti a poměrně jednoduchého přístupu velmi zajímavá, nicméně v současnosti se od ní poměrně ustupuje vzhledem k složité hardwarové realizaci. V dnešních složitých obvodech se totiž jedním z hlavních problémů stává propojování členů a částí obvodů. Z tohoto důvodu je značně nepraktické, komplikovat si situaci realizací pole paměťových buněk s náhodným přístupem a návrhem sběrnic, nutných k výběru buňky. Mnohem jednodušší je v tomto případě vytvoření řetězce scan buněk jako posuvný registr a použití metody SAS popř.

provedení testování pomocí vestavěných diagnostických nástrojů, které budou stručně popsány dále.

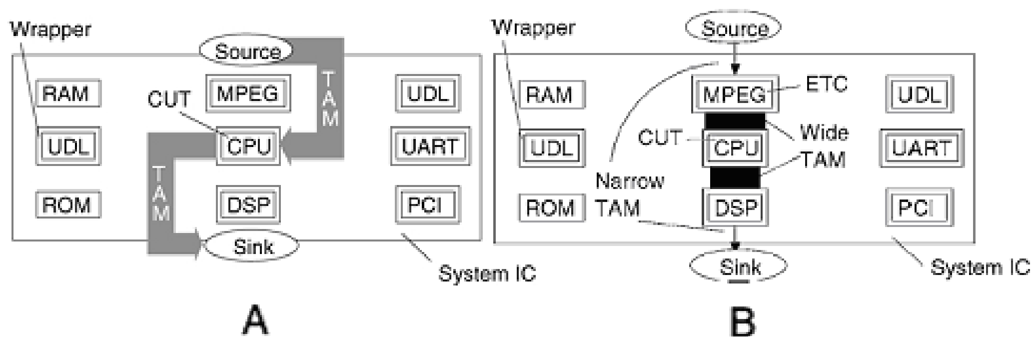
4.2. Testovací architektury umožňující dekompresi testu

V předchozí kapitole byly popsány dvě možné metody testování, resp. přístupu k paměťovým buňkám, pomocí kterých realizujeme test. Dále jsme dospěli k závěru, že perspektivnější z hlediska testování je v dnešní době metoda sériového přístupu SAS. Tato metoda je velmi vhodná i pro námi generovaný komprimovaný test. Komprimovaný bitstream lze totiž poměrně jednoduše dekomprimovat právě pomocí zřetězení paměťových buněk. V praxi se můžeme setkat s množstvím testovacích architektur, nicméně my se blíže seznámíme s testovací architekturou RESPIN [7], a její realizací jako BIST [8], pro níž je použití námi vytvořeného testu výhodné. Dále bude také objasněno, jakým způsobem probíhá dekomprese a v čem spočívá výhoda námi realizované komprese testu, což by mělo napomoci lepšímu pochopení problematiky testování.

4.2.1. RESPIN architektura

RESPIN (REusing Scan chains for test Pattern decompressIoN) [7] je testovací architektura, založená na sériovém řetězci paměťových buňek (scan chainu), popř. několika řetězcích řazených paralelně nebo zřetězených sériově viz. obrázek č.4. Pomocí této architektury lze přímo na čipu realizovat dekompresi testovacích vektorů do jednoho nebo několika paralelních scan chainů. Vstupem této testovací architektury je komprimovaná sekvence bitů, jejíž komprese je realizována např. pomocí kompakce a nejlepšího překrytu vektorů testu, což je náš případ a také případ COMPASu.

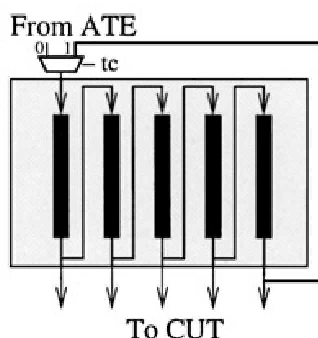
Na obrázku č.6 je zobrazeno blokové schéma standardní architektury a architektury RESPIN. Zdrojem testovacích vektorů je v tomto případě ATE (Automatic Test Equipment) – blok "source", který je umístěn mimo čip. Testovanou jednotku představuje CPU. U standardní architektury je vidět, že TAM (Test Access Mechanism), který slouží k "nahrání" testovacích vektorů, resp. snímání odezev, je poměrně široký (přenosy velkého množství dat), což vede k delší době testu a problematičtější realizaci v hardwaru.



Obrázek č.6: *A) Standardní testovací architektura,* ***B)*** *Testovací architektura RESPIN [7]*

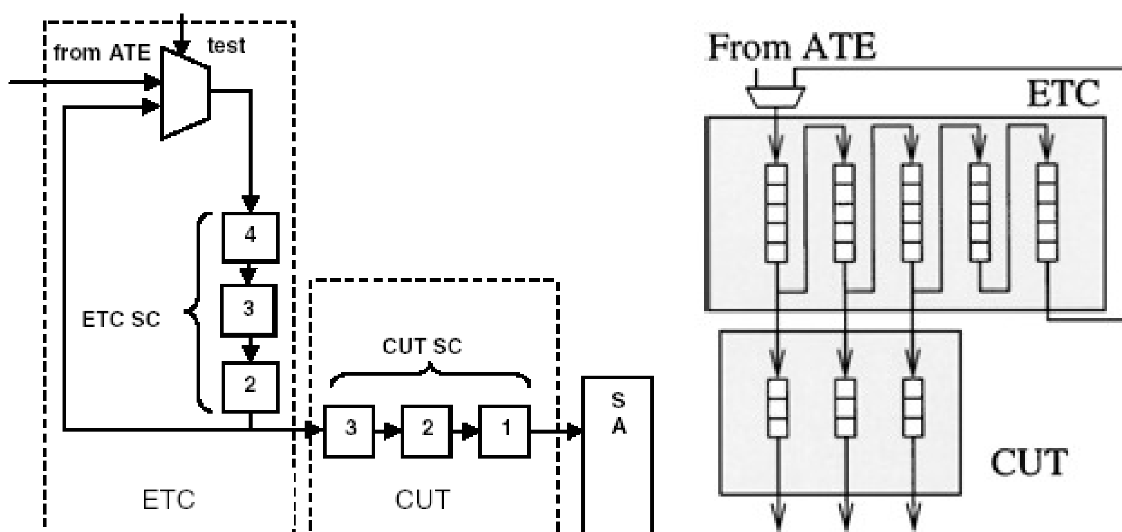
Testovací architektura RESPIN řeší problém "širokého" TAM přidáním tzv. ETC (Embedded Tester Core), které v našem případě představuje blok MPEG a "kompaktoru" odezev obvodu, který může být realizován například jako LFSR - v našem případě ho představuje blok DSP. Z obrázku je na první pohled patrné, že tímto způsobem odstraníme "široký" TAM mezi ATE a čipem, a to tak, že přes "úzký" TAM (málo dat, nízká frekvence) nahráváme komprimované testovací vektory do ETC (formou bitstreamu), kde se provede jejich dekomprese. Poté jsou přes vnitřní "široký" TAM dekomprimované vektory dopraveny do příslušných scan chainů, provede se odečtení odezvy na tyto vektory a přes "široký" TAM jsou odezvy přeneseny do "kompaktoru" odezev, kde dojde k jejich kompresi a přes "úzký" TAM jsou posílány mimo čip k vyhodnocení.

Jedním ze základních kamenů této architektury je ETC. Toto jádro, které provádí dekompresi bitstreamu přicházejícího z ATE, může být realizováno zřetězením scan chainů, viz. obrázek č.4 (vpravo), přidáním zpětné vazby a multiplexeru, viz. obrázek č.7. Takto může vypadat dekodér testovacích vzorků (ETC) pro několik scan chainů. Je patrné, že zásah do hardware je minimální.



Obrázek č.7: *ETC architektura pro více scan chainů [7]*

Jednoduchý příklad testovací architektury je ukázán na obrázku č.8. Zde je vidět konkrétní realizace propojení ETC a CUT. Vlevo pro jeden scan chain a vpravo pro několik paralelních scan chainů.



Obrázek č.8: *Příklad testovací architektury ETC+CUT jeden [1] / více scan chainů [7]*

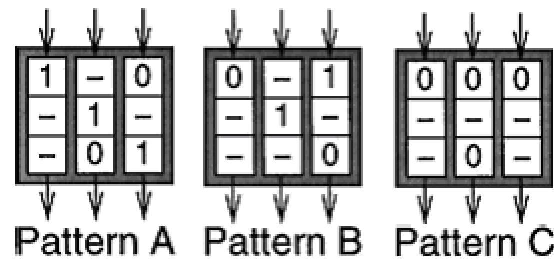
Dále si na obrázku č.5 všimněme signálu t_c, který řídí multiplexer na vstupu ETC a určuje tedy, použijí-li se pro vstup do ETC data z ATE nebo uplatní-li se zpětná vazba v ETC. Jeli vstup t_c=0, pracuje ETC v tzv. sériovém módu (seriál mode), což znamená, že vstupní data jsou dodávána do ETC z ATE. V případě, že t_c=1 se uplatní na vstupu ETC zpětná vazba a ETC pracuje v tzv. módu kruhového posunu scan chainů (circular mode). V tomto případě je výstup scan chainů ETC přiváděn zpětnou vazbou na vstup ETC. Signál t_c kromě ovládání vstupu ETC také řídí scan chainy v CUT, resp. určuje nachází-li se paměťové buňky chainů ve skenovacím módu (scan mode), kdy stimulují obvod testovacími vzorky nebo v módu, kdy snímají odezvy na testovací vzorky z výstupů CUT (capture mode). Z popisu tedy vyplývá, že řídící signál t_c pro CUT představuje řídící signál "scan enable". Tabulka č.2, ukazující nastavení signálu t_c a pracovní módy ETC a CUT, demonstruje spolupráci ETC a CUT. Jeli CUT v režimu zachycení odezev obvodu, provádí ETC načtení zakódovaného bitu z ATE, zatímco pro scan mode v CUT provádí kruhový posun. Aby mohly být tyto činnosti realizovány, musí být frekvence ETC a CUT stejné.

t_c = Scan enable	0	1
CUT	Capture mode	Scan mode
ETC	Serial mode	Circular mode

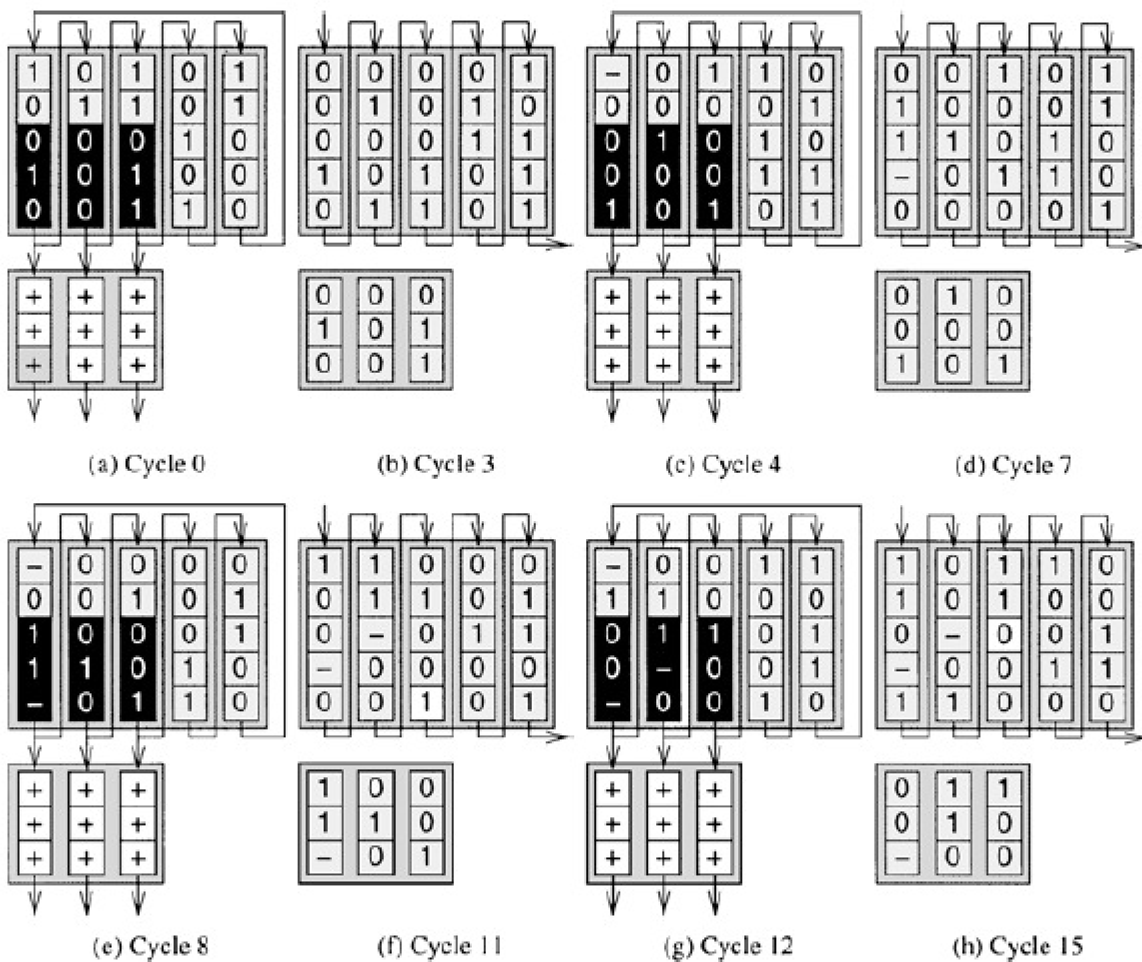
Tabulka č.2 *: Pracovní módy ETC a CUT [7]*

Průběh dekomprese a testu si nyní přiblížíme na následujícím příkladu. Budeme zde vycházet z architektury na obrázku č. 8 (vpravo), kde je CUT testován pomocí tří paralelních

scan chainů délky 3 a ETC obsahuje zřetězení pěti scan chainů délky 5. Dále máme k dispozici tři vzorky deterministicky generované pomocí ATPG. Tyto vzorky jsou zobrazeny v příslušných scan chainech na obrázku č.9.



Obrázek č.9: *Tři deterministicky generované vzorky A,B a C v scan chainech CUT [7]*



Obrázek č.10: *Simulace průběhu testu v ETC a CUT [7]*

Na obrázku č.10 je poté ukázána simulace průběhu testu pro tři testovací vzorky A, B a C z obrázku č.9. Obrázek č.10 (a) ukazuje systém v počátečním stavu. CUT pracuje v módu capture (značeno +) . ETC pracuje v circular módu, což je naznačeno šipkou zpětné vazby a je nastaven na počáteční posloupnost bitů (reset/scan in). Černým pozadím jsou zvýrazněny bity, které budou v následujících třech taktech přes "široký" TAM nasunuty do scan chainů v CUT.

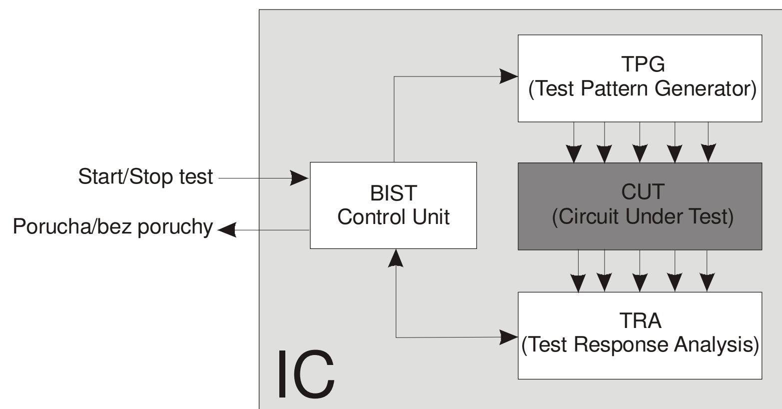
Všimněme si, že tento vzorek odpovídá vzorku C z obrázku č.9. Stav po provedení operace nasunutí vzorku C do scan chainu ukazuje obrázek č.10 (b). V dalším cyklu pracuje ETC v sériovém módu a dojde tedy k vysunutí bitu z ETC a nasunutí nového z ATE. CUT pracuje mezitím v capture módu, tzn. do scan chainů jsou uloženy odezvy na testovací vzorek C . Tuto situaci ukazuje obrázek č.10 (c). Dále je zde také vidět na černém pozadí nové vektory, které budou v následujícím kroku nasunuty do CUT. ETC pracuje opět v circular módu. V dalších třech cyklech dojde k nasunutí nových vzorků do CUT a zároveň k vysunutí odezev na předchozí vzorky, které jsou posléze analyzovány. Po těchto třech cyklech bude stav ETC a CUT odpovídat obrázku č.10 (d). Jednoduchým porovnáním zjistíme, že obsah scan chainů v CUT nyní neodpovídá žádnému testovacímu vzorku. Provedeme opět jeden cyklus, ve kterém čteme odezvy v CUT a nasunujeme nový bit do ETC, což ukazuje obrázek č.10 (e) a v dalších třech taktech nasuneme do CUT nové testovací vektory. Tyto odpovídají testovacímu vzorku A z obrázku č.9. Výsledek po operaci nasunutí vzorků do CUT je ukázán na obrázku č.10 (f). Následující obrázek č.10 (g) ukazuje opět čtení odezev v CUT. Dále zde dochází k opravě čtvrtého bitu v druhém scanovacím řetězci z DC na 1. Tento bit je původně nastaven jako tzv. don't care, nicméně ve vzorku B, kterému toto nastavení odpovídá, je nastaven na 1. Výsledná hodnota bitu, pro kterou bude tedy provedeno testování bude 1 (tyto operace jsou běžně realizovány při generování testu). Obrázek č.10 (h) už ukazuje pouze nasunutí posledního testovacího vzorku B z obrázku č.9.

Z předchozího popisu této metody je jasné, že je vhodná jak pro testování pomocí jednoho, tak pro testování pomocí více scan chainů. Odstraňuje problémy s "širokým" TAM na rozhraní čipu a ATE, čímž může test značně urychlit a díky minimálním změnám v hardwaru, nutným na dekompresi testovacích vzorků, a jejich doručení příslušnému scan chainu, je tato metoda pro praktické použití velmi přínosná. Pro nás představuje jednu z možných testovacích architektur, kterou můžeme použít pro realizaci testu komprimovaného naším nástrojem. V další kapitole bude popsána vestavěná diagnostika, která je v dnešní době velmi rozšířená a lze v ní využít postupy popsané v této kapitole.

4.2.2. Vestavěná diagnostika - BIST

Jendou z testovacích technik, ve které lze provádět testování s použitím sériového řetězce paměťových buněk je vestavěná diagnostika. Pro toto testování je charakteristické, že testovací jednotka je umístěna přímo na čipu a obecně tyto techniky nazýváme BIST (Build In Self Test). Jak může takový systém obecně vypadat je ukázáno na obrázku č.11. Skládá se

obvykle ze tří částí. První je TPG (Test Pattern Generator), což je generátor testovacích vzorků, který může být založen na některé z metod generování pseudonáhodných vzorků, nicméně v našem případě bude tento blok realizovat dekompresi komprimovaného testovacího bitstreamu. Druhou částí je tzv. TRA (Test Response Analyser), což je analyzátor odezev na vstupní vzorky, který vyhodnocuje, je-li odezva správná/špatná, resp. obsahuje-li obvod poruchu či nikoli (výstup porucha/bez poruchy). Někdy tento blok bývá označován jako SA (Signature Analyzator). Celou činnost pak řídí a synchronizuje řadič BIST [8].



Obrázek č.11: *Obecná bloková struktura BIST*

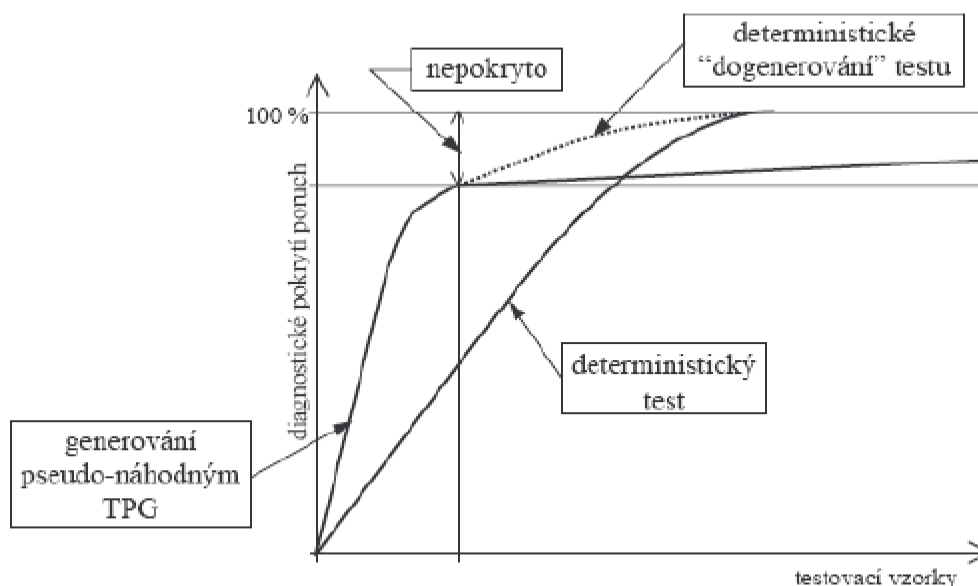
Ačkoli je na obrázku č.11 znázorněn pouze jeden "segment" BIST, typicky mohou být návrhy rozděleny do mnoha segmentů, jejichž jednotlivé části spolu mohou, ale také nemusejí spolupracovat, popř. některé části mohou být sdílené.

BIST techniky lze dále dělit do následujících skupin [6] :

- ***On-line BIST*** – při použití této techniky probíhá testování za podmínek pro běžnou funkci obvodu a není tedy třeba vybavovat obvod diagnostickým režimem činnosti. Dále lze dělit na :
    - ***Souběžný*** – kdy je umožněno testování zároveň s původní činností obvodu čehož je obvykle dosaženo pomocí vhodných kódovacích technik (bezpečnostní, detekční a korekční kódy), duplikace a porovnávání.
    - ***Nesouběžný*** - se využívá tehdy, není-li nutné souběžné testování a postačuje-li testovat obvod v tzv. klidovém režimu, což je stav, kdy obvod dokončil svou normální činnost a čeká na podnět k zahájení další činnosti. Test bývá většinou spouštěn softwarově. V průběhu je možné jej přerušit a pokračovat v další činnosti obvodu.

- ***Off-line BIST*** – provádí testování pouze v situaci, kdy je obvod přepnut do diagnostického režimu tzn. neprovádí činnost v normálním režimu. S tím souvisí také případné pozdější rozpoznání poruchy, protože testování neprobíhá tak často jako u on-line testování, ale jeho realizace bývá mnohem snadnější, z čehož plyne nižší cena tohoto řešení. Dále lze off-line BIST rozdělit na:
    - ***Funkční*** – vychází z funkčního popisu testovaného obvodu (viz.výše funkční test).
    - ***Strukturální*** – vychází z analýzy obvodové struktury (viz. výše strukturální test).

Jednou z velmi důležitých částí tohoto testovacího mechanizmu je generátor testovacích vzorků TPG. Tento bývá obvykle realizován jako lineární zpětnovazební posuvný registr LSFR (Linear FeedBack Shift Registr), nějaká jeho modifikace nebo celulární automat. Tyto zajišťují pseudonáhodné generování testovacích vektorů, pomocí kterých lze pokrýt většinu lehce-detekovatelných poruch viz. graf na obrázku č.12, ale ostatní jim činí velké problémy. Dalším nedostatkem pseudonáhodného generování testu je nutnost otestovat velké množství testovacích vektorů, které nepokrývají žádnou poruchu, zatímco v "deterministicky" generovaném testu každý vektor pokrývá alespoň jednu poruchu.



Obrázek č.12: *Graf pokrytí pomocí pseudonáhodného TPG [4]*

V našem případě BIST reprezentuje konkrétní realizaci RESPIN architektury. Výhody této realizace jsou tedy minimální nároky na "vnější" rozhraní, tzn. na komunikační rozhraní mezi čipem a vnějším testerem (TAM – Test Access Mechanism). Veškeré testování probíhá

pomocí vestavěného hardwaru a vnější tester řídí většinou pouze spouštění/zastavení testu a ověření signatury výsledku porucha/bez poruchy. Tímto se značně sníží nároky na přesun dat mezi vnějším testerem a čipem a testování se tímto způsobem poměrně zrychlí. Další výhodou je jednoduchá realizace v hardwaru, což usnadní vlastní návrh obvodu.

5. Sat-Atpg

Tento nástroj, resp. jeho modifikovaná verze, tvoří jednu z částí implementovaného generátoru komprimovaných vzorků. Jedná se o generátor testovacích vzorků, založený na testování splnitelnosti booleovské formule (SAT-ATPG – SATisfiability-Automatic Test Pattern Generator). My jsme se rozhodli pro použití řešiče vytvořeného v rámci diplomové práce kolegou Červákem [2]. Tento řešič je pro nás velmi vhodný vzhledem k tomu, že je k dispozici odpovídající dokumentace, což nám zjednodušilo jeho další modifikace.

5.1. Problém SAT

Problém splnitelnosti booleovské formule, zkráceně problém SAT, je rozhodovací problém, který hledá odpověď na otázku, zda je daná booleovská formule splnitelná. Výstupem je tedy rozhodnutí splnitelná/nesplnitelná (sat/unsat) a v případě splnitelnosti také ohodnocení jednotlivých proměnných.

Problém SAT patří do třídy NP-úplných problémů, což znamená, že hledání řešení má exponenciální složitost, nicméně certifikát lze ověřit v polynomiálním čase.

5.2. Davis & Putnam algoritmus

Tento algoritmus, resp. některá jeho modifikace, je jedním z nejpoužívanějších při řešení SAT problému (v CNF). Obecný algoritmus Davis & Putnam (DP) [9], je založen na rezoluci, a definuje několik pravidel, pomocí kterých lze SAT instanci v CNF upravit, aniž by tato úprava vedla na nesplnitelnost. V této sekci se nebudeme zabývat konkrétním popisem algoritmu, ale uvedeme si pravidla, která používáme při nastavování proměnných v CNF.

- **Pravidlo “jednotkového literálu“** – toto pravidlo říká, že obsahuje-li formule v CNF klauzuli, ve které se nachází pouze jeden literál, lze jej odstranit z CNF bez ovlivnění její splnitelnosti. Odstranění provedeme nastavením literálu v CNF na hodnotu danou jednotkovým literálem (nastavíme tak, aby byl splněn). Máme-li tedy například formuli $(a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (\neg a)$, která obsahuje jednotkovou klauzuli $(\neg a)$, provedeme nastavení proměnné ***a*** na hodnotu 0. Z toho vyplývá, že druhá a třetí klauzule je splněná. Lze je tedy z CNF odstranit, a z první klauzule lze odstranit literál ***a***, protože nebude nikdy 1, tzn. výsledek nemůže ovlivnit. Redukovaná formule bude tedy obsahovat pouze klauzuli $(b \vee \neg c)$.

- **Pravidlo "unipolárního literálu"** – toto pravidlo říká, že vyskytuje-li se v CNF literál, který má ve všech výskytech stejnou polaritu, lze všechny klauzule, ve kterých se nachází z CNF odstranit, aniž by byla ovlivněna splnitelnost formule. Máme-li tedy například formuli $(e \vee \neg c) \wedge (a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c)$, lze odstranit všechny klauzule obsahující literál ***b***, aniž by to ovlivnilo splnitelnost. Po provedení této úpravy bude formule redukována na $(e \vee \neg c)$.

V našem případě se nesnažíme redukovat CNF eliminací proměnných, ale potřebujeme provést nastavení podmnožiny proměnných na konkrétní hodnoty. To lze realizovat například přidáním jednotkové klauzule obsahující literál požadované polarity a aplikací prvního pravidla. Při nastavování je ovšem nutné brát zřetel na případné konflikty způsobené dalšími jednotkovými klauzulemi, ve kterých se tento literál může vyskytovat.. Pokud by se ve formuli vyskytoval nastavovaný literál v jednotkové klauzuli v opačné polarizaci, než jakou chceme nastavit, lze přerušit algoritmus a prohlásit formuli pro dané ohodnocení za nesplnitelnou, což nám ušetří práci s dalším řešením.

5.3. Metody generování testu pomocí ATPG

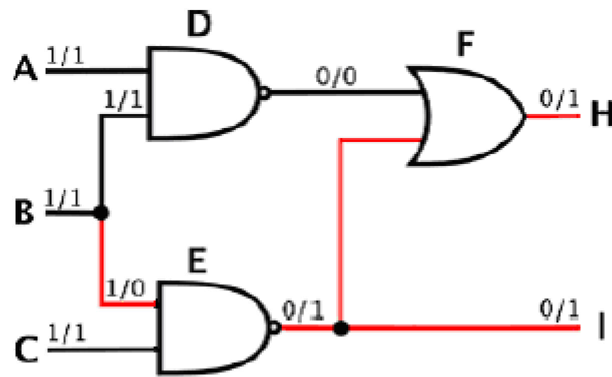
Jak bylo již dříve uvedeno, námi zvolený nástroj pracuje na bázi splnitelnosti booleovské formule, která je v našem případě reprezentována ve formě CNF. Obecně lze ovšem použít dvě metody generování testu pomocí ATPG, a to strukturální, vycházející ze struktury hradel a reprezentace číslicového obvodu jako grafu, a algebraické, převádějící strukturu obvodu do algebraické formy. Námi zvolený Sat-ATPG nástroj používá ke generování testu algebraické metody, pomocí které převede poruchu na instanci problému SAT reprezentovanou jako CNF. Tato normální forma byla již dříve popsána a nyní se jí tedy již nebudeme zabývat. Zaměříme se spíše na algebraické metody transformace obvodu do této normální formy, přičemž kvůli zachování konzistence a zprůhlednění jisté analogie mezi strukturální a algebraickou metodou bude v další kapitole stručně popsána také strukturální metoda.

5.3.1. Strukturální metoda

Tato metoda využívá k nalezení testovacích vektorů algoritmy založené na prohledávání grafů. K obvodu přistupuje jako ke grafu, ve kterém hradla tvoří vrcholy a vodiče tvoří hrany grafu. Základní algoritmus, který poté na graf aplikujeme se obecně skládá ze tří kroků [2]:

1. Nastavení výstupu hradla, popř. primárního vstupu/výstupu na opačnou logickou hodnotu vzhledem k testované poruše (tzn. Sa0 => nastavím log 1, Sa1 => nastavím log 0).
2. Provedeme "propagaci", resp. hledáme citlivou cestu z místa poruchy na primární výstupy.
3. Projdeme graf od primárních výstupů k primárním vstupům a nalezneme ohodnocení primárních vstupů konzistentní s nastavením vnitřních hradel.

Tento algoritmus bude dále demonstrován na příkladu:



Obrázek č.13: *Obvod s poruchou Sa0 na vstupu hradla E [2]*

Na obrázku č.13 je nakresleno zapojení jednoduchého kombinačního obvodu s poruchou typu Sa0 na vstupu hradla E. Dále je každý vodič (vstup/výstup hradla) označen dvojicí čísel, z nichž první je logická hodnota v obvodu bez poruchy a druhé je logická hodnota s poruchou. Je zde vidět, že podle prvního bodu algoritmu je hodnota na vstupu hradla E nastavena na logickou 1, jelikož se snažíme detekovat poruchu Sa0. Dále je červenou barvou znázorněna propagace citlivé cesty z místa poruchy na primární výstupy obvodu. V našem případě detekujeme výskyt poruchy na obou primárních výstupech H i I. V případě, že porucha nenastane, bude na primárních výstupech vektor (0,0), zatímco pro poruchu Sa0 na vstupu hradla E bude na výstupech vektor (1,1). Dále je vidět, že primární vstupy jsou nastaveny tak, aby neovlivnily šíření citlivé cesty na primární výstupy, tzn. vstupy hradla D jsou nastaveny na log. 1, což zaručuje log. 0 na vstupu hradla F, přes které vede citlivá cesta na jeden primární výstup, a dále nastavení druhého vstupu hradla E na log. 1, což umožňuje propagaci citlivé cesty hradlem E. Vstupní vektor, který umožňuje otestovat poruchu Sa0 na vstupu hradla E je tedy (1,1,1).

V předchozím příkladu byl použit pojem citlivá cesta, kterým rozumíme takové nastavení vstupů hradel na cestě od poruchy k výstupu, aby se hodnoty výstupů těchto hradel v bezporuchovém a poruchovém stavu lišily, čímž je detekována porucha. Například pro detekci poruchy Sa0 na vstupu hradla AND nastavíme jeho vstupy na log. 1. Pro detekci Sa1 na vstupu hradla OR nastavíme jeho vstupy na log. 0 atd.. Obecně tedy nastavíme vstup hradla, které testujeme, na opačnou logickou hodnotu a ostatní vstupy hradla nastavíme tak, aby na nich výstup nezávisel. Jestliže poté získáme na výstupu hradla logickou hodnotu, která neodpovídá nastavení jeho vstupů, vyskytuje se na testovaném vstupu porucha.

Z obrázku č.13 je dále patrné, že citlivá cesta se může větvit. Problémy nastanou v případě, že po větvení citlivé cesty dochází k jejímu spojení tzv. rekonvergenci. V tomto případě je nutné se vrátit, tzn. provést backtracking, a hledat jinou cestu, resp. jiné nastavení vstupů hradel.

5.3.2. Algebraická metoda

Při hledání testovacího vektoru pro danou poruchu pomocí algebraické metody se snažíme poruchu převést na problém řešení algebraického výrazu. Tento výraz vyřešíme a z výsledku poté odvodíme řešení původního problému. Jinými slovy transformujeme problém hledání vektoru na algebraický výraz, který poté zvládne dobrý matematik vyřešit bez znalosti hradel či kombinačních obvodů [2].

Kombinační obvod lze z algebraického hlediska považovat za logickou funkci, která přiřazuje konkrétnímu nastavení vstupů odpovídající hodnoty výstupů. Toto jde zapsat jako $(y_1,y_2,\ldots,y_m) = f(x_1,x_2,\ldots,x_n)$, kde Y_m je vektor primárních výstupů a X_n je vektor primárních vstupů. Jestliže funkce $f : X_n \rightarrow Y_m$ představuje správně fungující obvod, pak obvod s poruchou bude reprezentován jako $f^c : X_n \rightarrow Y_m$. Pro otestování poruchy tedy hledáme taková X_n, pro které platí $f(x_1,x_2,\ldots,x_n) \oplus f^c(x_1,x_2,\ldots,x_n) = 1$, což lze interpretovat jako řešení problému splnitelnosti booleovské formule. V našem případě je problém SAT reprezentován logickou funkcí zapsanou v CNF. V následující kapitole bude popsána transformace strukturního popisu obvodu na algebraický.

5.4. Transformace kombinační logiky do CNF

Cílem této transformace je vytvořit pro konkrétní poruchu instanci SAT problému v CNF. Každá tato instance SAT problému potom představuje kompletní množinu testovacích

vektorů pro danou poruchu. Jestliže je porucha nedetekovatelná, bude výsledná instance SAT nesplnitelná.

K transformaci obvodu na problém splnitelnosti booleovské formule lze použít množství algoritmů, nicméně my si zde popíšeme pouze metodu booleovských diferencí a Tsietsienovy transformace, které z této metody vychází [2][10].

5.4.1. Booleovská diference

Jednou z nejznámějších algebraických metod je booleovská diference. Booleovská diference jakékoli vstupní proměnné x_i z funkce f lze zapsat jako:

$$f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \oplus f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$

Toto lze zapsat jako df/dx_i, přičemž tímto zápisem reprezentujeme množinu všech vektorů, pro které existuje citlivá cesta ze vstupu x_i na výstup funkce f. Množinu testovacích vektorů pro x_i a poruchu Sa0 získáme jako $x_i * df/dx_i$ a množinu testovacích vektorů pro Sa1 získáme jako $\neg x_i * df/x_i$. Takto získanou funkci jde dále pomocí operací booleovské algebry upravit do CNF. Tento přístup ovšem většinou nebývá použit vzhledem k složitější implementaci.

Příklad booleovské diference pro funkci F = A¬B + AC + CD a generování kroků testu pro poruchu Sa0 na vstupu A. Dle definice řešíme rovnici A(¬B¬C + C¬D) = 1, kterou dále upravíme na A¬B¬C + AC¬D = 1, z čehož vyplývá, že poruchu Sa0 na vstupu A lze detekovat vektory ABCD = 100x a 1x10 (tzn. ABCD=1000,1001,1010,1110) [4].

5.4.2. Tsietsienovy transformace

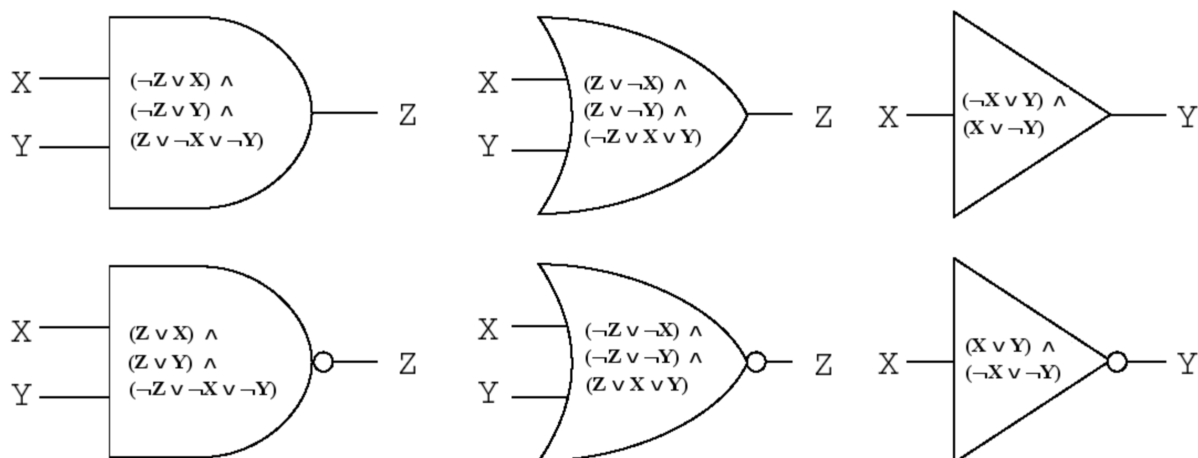
Z předchozí kapitoly je jasné, že metoda booleovských diferencí a následná úprava výrazu, není pro převod poruchy na instanci SAT v CNF nejvhodnější a v praxi se také častěji setkáme s využitím tzv. Tsietsienových transformací [2][10], pomocí kterých lze poruchu převést do CNF v lineárním čase podle počtu hradel.

Zjednodušeně řečeno, lze každé hradlo popsat skupinou klauzulí, která představuje jeho popis v CNF. Při generování CNF obvodu provedeme konjunkci klauzulí hradel, které jsou na citlivé cestě. Množiny klauzulí pro jednotlivá hradla musejí být splnitelné, z čehož plyne, že jestliže je porucha detekovatelná, musí být jejich konjunkce také splnitelná.

Nyní si ukážeme, jakým způsobem lze odvodit tuto transformaci pro hradlo AND. Uvažujme dvou-vstupové hradlo AND se vstupy X, Y a výstupem Z. Toto hradlo lze popsat

formulí $Z = X \wedge Y$. Dále platí, že $P = Q$ je logicky ekvivalentní $(P \rightarrow Q) \wedge (Q \rightarrow P)$, podle čehož lze původní formuli hradla AND upravit na $(Z \rightarrow (X \wedge Y)) \wedge ((X \wedge Y) \rightarrow Z)$. V posledním kroku provedeme převod implikace na disjunkci s použitím pravidla $P \rightarrow Q$ je logicky ekvivalentní $\neg P \vee Q$ a získáme formuli $(\neg Z \vee X) \wedge (\neg Z \vee Y) \wedge (\neg X \vee \neg Y \vee Z)$. Na první pohled je zde vidět, že tato formule je v CNF. Dále si všimněme, že tato formule je splněná pouze tehdy, jeli ohodnocení proměnných konsistentní s pravdivostní tabulkou hradla AND. Transformaci lze samozřejmě zobecnit i na N-vstupová hradla, přičemž transformujeme-li například N-vstupové hradlo AND, tzn. $Z = AND(X_n)$, dostaneme jako výstup N binárních termů $(\neg Z \vee x_i)$, na které lze pohlížet jako na implikace $Z \rightarrow x_i$ nebo $\neg x_i \rightarrow Z$. Dále se ternární term 2-vstupového hradla AND rozšíří na N-nární term $(\neg x_1 \vee \neg x_2 \vee \ldots \vee \neg x_n \vee Z)$ vyjadřující vztah $(\neg x_1 \wedge \neg x_2 \wedge \ldots \wedge \neg x_n) \rightarrow Z$. Podobě lze transformaci odvodit pro všechny základní hradla [10].

Transformace základních hradel do CNF je ukázána na obrázku č.14. Pomocí Tsietsienových transformací lze pro jakýkoli obvod sestavit jeho CNF, přičemž pro generování testu konkrétní poruchy se využívá booleovské diference funkčního obvodu a obvodu s poruchou.

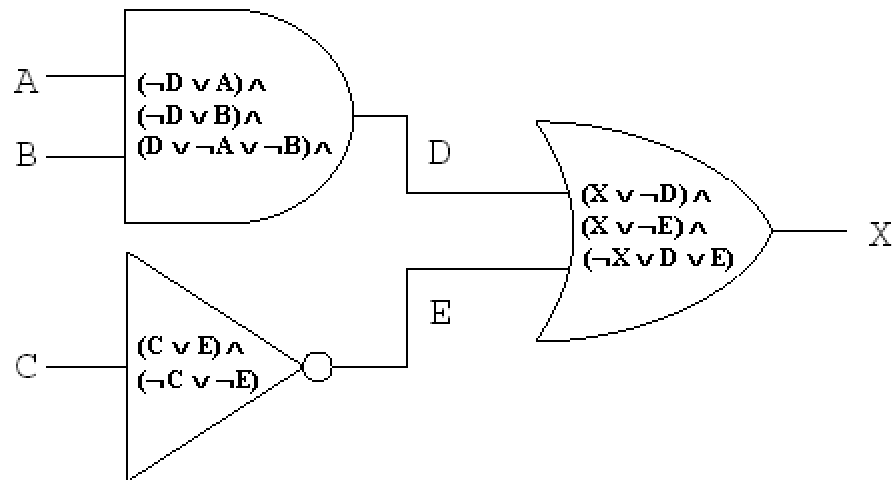


Obrázek č.14: *Ukázka transformace základních hradel do CNF[10]*

Protože každé hradlo, i větvení, je ohodnoceno formulí, která musí být splněná, můžeme pro každý výstup obvodu (ne jen primární) sestavit charakteristickou formuli v CNF průchodem grafu od konkrétního výstupu, přičemž při průchodu hradlem nebo větvením rozšíříme charakteristickou formuli o její konjunkci s klauzulemi popisujícími člen, kterým procházíme. A jak již bylo dříve řečeno, jestliže klauzule popisující jednotlivé prvky obvodu musí být nezávisle na sobě splnitelné, musí být jejich konjunkce také splnitelná. Obrázek č.15 ukazuje kombinační obvod jehož hradla jsou popsány svými charakteristickými formulemi.

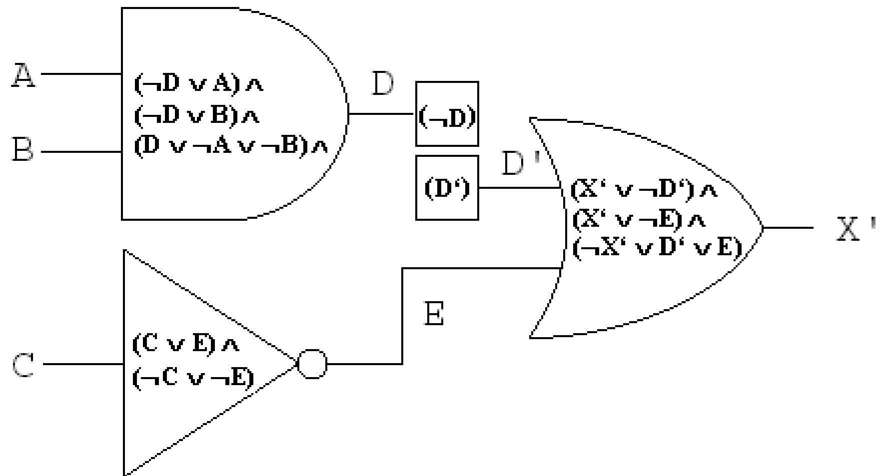
Vybereme-li si například výstup X, bude charakteristická formule obvodu vypadat následovně:

$$(X \vee \neg D) \wedge (X \vee \neg E) \wedge (\neg X \vee D \vee E) \wedge$$
$$(\neg D \vee A) \wedge (\neg D \vee B) \wedge (D \vee \neg A \vee \neg B) \wedge$$
$$(C \vee E) \wedge (\neg C \vee \neg E)$$



Obrázek č.15: *Kombinační obvod s hradly popsanými klauzulemi v CNF [10]*

Předchozí příklad demonstroval vytvoření charakteristické formule v CNF pro obvod bez poruchy. My ovšem potřebujeme pro realizaci booleovské diference také charakteristickou formuli obvodu s poruchou. Tuto formuli získáme prostým zkopírováním obvodu, resp. jeho CNF, přičemž ale do místa, kde chceme testovat poruchu, vložíme novou proměnnou, která bude reprezentovat předpokládanou poruchu spoje v porouchaném obvodu. Novou proměnnou, která "popisuje" chování poruchy v charakteristické formuli porouchaného obvodu dále nastavíme přidáním "jednotkového" termu určujícího typ testované poruchy, tzn. pro Sa1 nastavíme proměnnou na logickou 1, zatímco pro Sa0 bude nastavena na logickou 0. Vzhledem k tomu, že porouchaný a bezporuchový obvod budou mít identickou strukturu, kromě místa poruchy, není nutné přejmenovávat všechny proměnné, ale pouze proměnné ležící na cestě mezi poruchou a výstupy. Praktická ukázka tohoto postupu je naznačena na následujícím příkladu. Kombinační obvodu na obrázku č.16 ukazuje obvod z předchozího příkladu, ovšem nyní s poruchou Sa1 na výstupu D.



Obrázek č.16: *Kombinační obvod s poruchou Sa1 na D [10]*

Na obrázku č.16 je názorně vidět přidání nové proměnné D‘ do obvodu. Touto proměnnou popisujeme poruchové chování obvodu. Testujeme-li v našem případě poruchu Sa1, bude podle předchozí definice do CNF popisující tento obvod přidán jednotkový term D‘ (tím dojde k nastavení D‘ na trvalou logickou 1). Termy obsahující proměnnou D lze z charakteristické formule porouchaného obvodu vypustit, protože nemohou ovlivnit výstupní stav obvodu. Dále je nutné provést přejmenování proměnných na cestě od poruchy k výstupu, což je v našem případě pouze X na X‘. Výsledná charakteristická formule obvodu s poruchou tedy bude vypadat následovně :

$$\mathbf{(X' \lor \neg D') \land (X' \lor \neg E) \land (\neg X' \lor D' \lor E) \land (D') \land (C \lor E) \land (\neg C \lor \neg E)}$$

Pro otestování vybrané poruchy poté stačí najít nastavení primárních vstupů obvodu, pro které se bude výstup obvodu s poruchou a bez poruchy lišit. Výslednou formuli v CNF, jejímž řešením jsou všechny vektory detekující testovanou poruchu tedy získáme tak, že provedeme konjunkci formulí porouchaného obvodu a obvodu bez poruchy, a přidáme klauzule reprezentující XOR (diferenci) výstupů porouchaného a bezporuchového obvodu. V našem případě bude mít tento XOR vstupy X, X’ a výstup BD. Výsledkem je následující formule v CNF, jejímž řešením jsou všechny vektory detekující poruchu Sa1 na D.

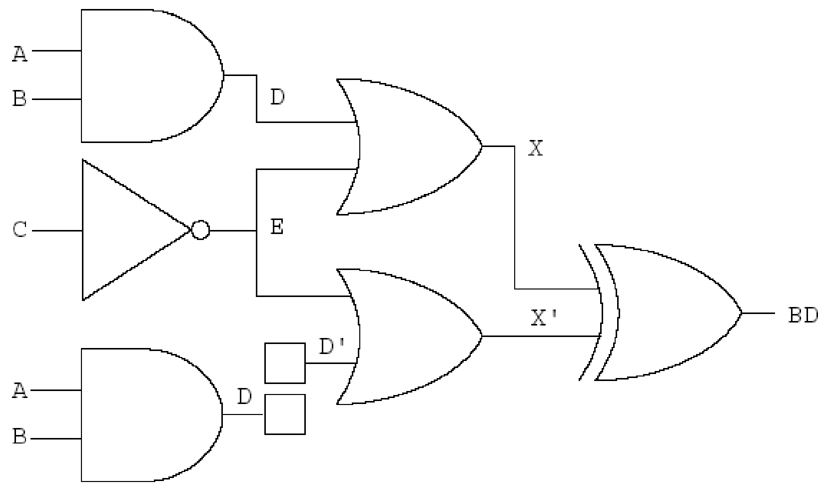
$$\mathbf{(X' \lor \neg D') \land (X' \lor \neg E) \land (\neg X' \lor D' \lor E) \land (D') \land (C \lor E) \land (\neg C \lor \neg E) \land}$$

$$\mathbf{(X \lor \neg D) \land (X \lor \neg E) \land (\neg X \lor D \lor E) \land (\neg D \lor A) \land (\neg D \lor B) \land (D \lor \neg A \lor \neg B) \land}$$

$$\mathbf{(\neg X \lor X' \lor BD) \land (X \lor \neg X' \lor BD) \land (X \lor X' \lor \neg BD) \land (\neg X \lor \neg X' \lor \neg BD)}$$

V této výsledné formuli první řádek popisuje chování obvodu s poruchou, druhý řádek popisuje chování obvodu bez poruchy a poslední řádek realizuje XOR výstupů obvodu

s poruchou a bez poruchy. Formule lze tedy zakreslit jako kombinační obvod na obrázku č.17. Jestliže vstupní vektor detekuje poruchu, bude na výstupu BD logická 1.



Obrázek č.17: *XOR obvodu s poruchou a bez poruchy [10]*

5.5. SAT-solver

Z předchozího popisu vyplývá, že důležitou součástí ATPG založeného na řešení problému SAT je nástroj, pomocí kterého jsme tento problém schopni vyřešit neboli SAT-solver. Rychlost s jakou je SAT-solver schopen řešit tyto komplexní problémy je přitom kritická pro celkovou rychlost ATPG. Kolega Červák, z jehož práce vycházíme, vytvořil rozhraní, které umožňuje pro řešení SAT instancí využít "jakýkoli" SAT-solver, přičemž je ovšem velmi vhodné omezit se na řešiče používající tzv. "kompletní" SAT algoritmy, tzn. takové, které jsou schopny najít řešení v konečném počtu kroků.

Vzhledem k výsledkům, které dosahoval původní ATPG, jsme se rozhodli, že budeme nadále využívat SAT-solver MinSat 2.0 [11]. Tento je schopen vyřešit velmi rychle i složité SAT instance. Používá sofistikované předzpracování SAT problému pomocí nástroje SatELite(eliminace proměnných/klauzulí) a kompletní algoritmy založené na DLL (Davis,Logemann and Loveland), které používají back-trackingu ke snížení paměťové náročnosti, učení zajišťují případné odhalení konfliktních klauzulí a ořezání stavového prostoru. Bohužel tento nástroj v základní verzi není schopen generovat vektory s don't care stavy, které jsou pro nás z hlediska další komprese velmi vhodné (to ovšem neumí žádný z dostupných open-source řešičů). Generování DC stavů lze dosáhnout pomocí modifikace tohoto nástroje, nicméně ta je spojena se značným zvýšením doby potřebné k nalezení řešení.

6. Atalanta-M

Je velmi sofistikovaný ATPG s integrovaným simulátorem poruch HOPE, který vznikl úpravou programu Atalanta, vyvinutého na Virginia Polytechnic & State University. V naší práci jsme se rozhodli použít modifikaci nástroje Atalanta-M [12], kterou implementoval v rámci bakalářské práce kolega Myslík [3]. Ve své práci přepsal původní Atalantu-M z jazyka C do jazyka C++, čímž vytvořil objektovou verzi tohoto nástroje, a poté převedl aplikaci na dynamicky linkovanou knihovnu, což zjednodušilo integraci Atalanty do dalších systémů a usnadnilo kompilovatelnost pod více operačními systémy. Dále odstranil omezení, kterými trpěla Atalanta-M. Jednalo se hlavně o odstranění tzv. úzkých míst programu, která znemožňovala použití aplikace pro rozsáhlejší obvody. Zde se jednalo většinou o úpravy datových struktur. Díky těmto úpravám je integrace tohoto nástroje poměrně jednoduchou záležitostí, a díky vylepšení v oblasti datových struktur, jsme s ním schopni zpracovávat i obvody velké složitosti, což je právě naším cílem.

V naší práci realizujeme generátor komprimovaných testovacích vzorků, a Atalantu tedy budeme využívat pouze kvůli jejímu sofistikovanému simulátoru poruch. Nebudeme tedy v tomto dokumentu popisovat algoritmy, které jsou v tomto nástroji implementovány. Jejich popis je např. v [3][12][13].

6.1. Simulace poruch

Simulace provádí kontroly odezev, úplnosti a bezhazardovosti. Metod simulace poruch je hned několik. První a nejjednodušší je metoda programové injekce poruch do simulovaného obvodu. Zkoumaný obvod se při simulaci poruchy změní na jiný obvod, který je třeba simulovat v samotném průchodu. Další metodou je paralelní simulace, která využívá možnosti zpracovat několik bitových operaci najednou, což je dáno nezávislostí těchto operací. Tímto způsobem jsou jednotlivé bity do obvodu injektovány najednou a každý bit je rezervován pro simulaci jedné poruchy. Další zde zmíněnou metodou je deduktivní simulace, která je založena na simulaci bezporuchového obvodu a odvození detekovaných poruch na jednotlivých vodičích. Poslední zde zmíněnou metodou je souběžná simulace, která funguje tak, že každý logický člen obvodu je vybaven seznamem poruch a jejich možných důsledků. Podrobnější popis těchto metod lze nalézt v [3][4].

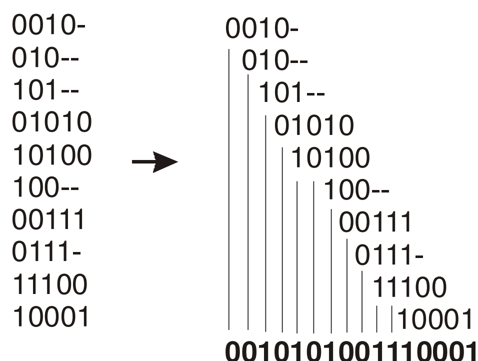
7. Návrh a realizace

Cílem naší práce je vytvořit nástroj schopný generovat komprimovaný bitstream testovacích vzorků na základě popisu obvodu ve formátu .bench (viz. kapitola 13.2). Základem našeho algoritmu je princip použitý v COMASu, tj. hledání nejlepšího překrytu testovacích vektorů. Nástroj COMPAS ovšem pracuje tak, že pomocí vhodného generátoru testu si vygeneruje testovací vektory pro jednotlivé poruchy a tyto vektory se posléze snaží pomocí sofistikovaných heuristických algoritmů maximálně překrýt. Co se týče výsledné komprese i rychlosti, dosahuje COMPAS velmi dobrých výsledků a pokud se tedy dá hovořit o nějakých slabinách tohoto řešení, jedná se bezesporu o přímé vygenerování testovacích vektorů, které jsou posléze zpracovávány. Z předchozího vyplývá, že výsledek může tedy do značné míry záviset na vektorech testu, které COMPAS získá jako vstup. Pro COMPAS je velmi důležité, aby jednotlivé vektory obsahovaly co nejvíce tzv. don't care stavů a v případě, že je neobsahují, dojde k citelnému zhoršení výsledků.

S vedoucím práce jsme se tedy zabývali způsoby odstranění slabiny COMPASu a sestavili jsme algoritmus, založený na SAT-ATPG, který bude následně popsán. Stejně jako COMPAS používáme myšlenku hledání nejlepšího překrytí.V našem přístupu ovšem nevycházíme z předem vygenerovaného testu, ale snažíme se přímo generovat testovací vektory tak, aby došlo k co největšímu překrytu, což nám jak doufáme, pomůže vytvořit velmi dobře komprimovaný testovací bitstream i za cenu případné delší doby řešení.

7.1. Specifikace problému

Jak již bylo řečeno v úvodu této kapitoly, snažíme se vytvořit kompresi vektorů testu pomocí jejich vhodného překrytu a vytvořit tak komprimovaný bitstream. Princip této komprese je graficky naznačen na obrázku č.18. Vlevo na obrázku můžeme vidět nekomprimovaný test vygenerovaný pomocí běžného ATPG. Šipka naznačuje jakým způsobem v našem případě realizujeme kompresi a tučně je vyznačen komprimovaný bitstream. Je zde také vidět, že i pro takto malý test lze dosáhnout velmi pěkné komprese. Původní test obsahoval deset vektorů délky pět, což znamená, že celkový počet bitů testu byl 50, zatímco komprimovaný test má délku pouze 16 bitů. Už z této malé úlohy je tedy zřejmé, že pro větší instance můžeme tímto způsobem dosáhnout velmi významné úspory paměti.



Nekomprimovaný test : 0010-010--101--0101010100100--001110111-1110010001
Komprimovaný test : 0010101001110001

Obrázek č.18: *Příklad komprese testu metodou překrytu vektorů*

Dále, jak jsme již uvedli, nevycházíme při kompresi z vygenerovaného testu, ale snažíme se přímo generovat vhodné vektory. Z tohoto důvodu používáme generátor testovacích vektorů, založený na řešení problému splnitelnosti booleovské formule. Tento nástroj pracuje s jednotlivými poruchami jako s instancemi SAT problému. Každá tato instance obsahuje kompletní množinu vektorů detekujících danou poruchu a v tom spočívá právě výhoda tohoto přístupu. Zatímco COMPAS je odkázán na výsledky z ATPG, náš nástroj projde jednotlivé instance SAT reprezentující příslušné poruchy a z množin jejich řešení vybere to nejlepší tak, aby došlo k maximální kompresi.

7.2. Analýza problému

Z předchozího popisu je jasné, že při řešení problému jako je tento máme mnoho stupňů volnosti, pomocí kterých můžeme řešení směrovat k optimálním výsledkům. My se snažíme dosáhnout především maximální komprese testu i za cenu delší doby řešení. Rozebereme si nyní několik základních myšlenek, ze kterých můžeme dále vycházet. Z hlediska maximální komprese se můžeme pokusit ubírat dvěma směry. První vychází z poznatků, ke kterým došli tvůrci COMPASu, tzn. že komprese je nejlepší v případě velkého množství DC stavů. Je jasné, že čím více mají vektory DC, tím jednodušší je nalézt vhodné překrytí. Na druhou stranu je nutné si uvědomit, že čím mají vektory více DC, tím méně pokrývají poruch a je tedy zapotřebí provést překrytí mnohem většího počtu vektorů (COMPAS pracuje s jedním vektorem na jednu poruchu => max. počet DC). Z časového hlediska je situace obdobná, neboť nalezení dalšího vektoru pro překrytí je poměrně jednoduché (nejsme tak limitováni nastavením proměnných) a není třeba k jeho nalezení řešit tolik SAT instancí, ale vzhledem k tomu, že musíme nalézt více takovýchto vektorů, nemusí se přidání DC projevit na zrychlení aplikace. Druhým přístupem může být odstranění DC z testovacích vektorů.

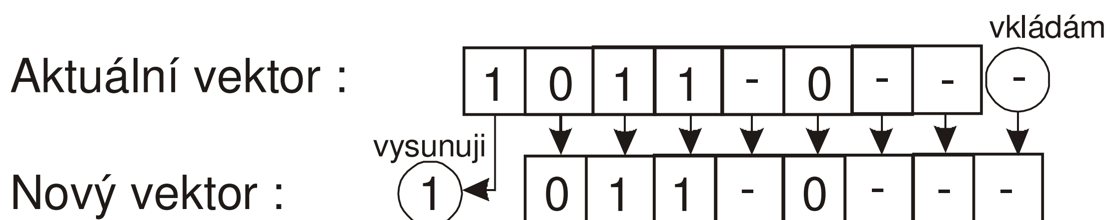
Vycházíme zde přitom ze základní vlastnosti našeho řešiče, tzn. generuje si přímo vhodný následný vektor a nezáleží nám na tom, jak daný vektor vypadá. Pro nástroj COMPAS jsou životně důležité DC stavy, vzhledem k tomu, že jen "rovná" testovací vektory, které dostane. My se naopak můžeme pokusit nahradit DC vhodnými hodnotami, abychom maximalizovali počet poruch, které daným vektorem pokryjeme, čímž snížíme počet vektorů testu. Bohužel nastavením DC na "špatnou" hodnotu můžeme zapříčinit poměrně velké zhoršení komprese. Z časového hlediska lze rozumně předpokládat, že budeme-li generovat plné vektory, budeme pravděpodobně muset k nalezení dalšího vhodného vektoru muset vyřešit větší množství SAT instancí, což se samozřejmě projeví na době řešení. Dále je třeba si uvědomit, že i při řešení více SAT instancí provádíme vždy nastavení proměnných podle vektoru testu, který zde představuje nastavení proměnných primárních vstupů. Pokud je tento vektor plně nastaven, znamená to, že budou nastaveny všechny proměnné primárních vstupů v SAT instanci, čímž v závislosti na jejich zastoupení, může dojít k značnému zjednodušení SAT problému, popř. se přímo při nastavování projeví nesplnitelnost. Tímto způsobem tedy může naopak dojít k rychlejšímu zpracování SAT instancí a zrychlení aplikace. Už tento základní rozbor naznačuje, že se jedná o poměrně komplikovaný problém k jehož vyřešení bude zapotřebí provést velké množství měření, ze kterých snad bude možné nalézt nejlepší přístup.

V předchozím textu jsme se zabývali především analýzou závislosti mezi časem a kompresí, nicméně je třeba brát v úvahu také závislost mezi časem a pamětí, kterou můžeme k řešení využít. My jsme zde opět upřednostnili snížení paměťových nároků na úkor doby řešení. Nabízí se zde několik možností. První z nich umožňující zrychlení aplikace by mohla spočívat v ukládání všech řešení pro každou SAT instanci. Takto bychom aplikaci mohli značně urychlit, neboť bychom jednou vyřešili sadu SAT instancí a poté jen pracovali s polem jejich řešení. Na druhou stranu je ovšem jasné, že paměťové nároky tohoto řešení by byly neúnosné. Řešením, byť jen jedné SAT instance, mohou být milióny vektorů značné délky. V našem případě navíc pracujeme s velkým množstvím SAT instancí. Tato možnost tedy nepřipadá v úvahu. Další, mnohem reálnější možností, je zde uchovávání CNF pro každou poruchu. Tento přístup klade mnohem menší nároky na spotřebu paměti vzhledem k tomu, že celá množina řešení je zde reprezentována pomocí CNF. Na druhou stranu jsme nuceni pro získání vektoru řešit SAT problém, což se podepíše na době řešení. Dále je také třeba si uvědomit, že ačkoli jedna SAT instance v CNF nemusí klást příliš velké nároky na paměť, dá se předpokládat, že v případě větších obvodů, kde bychom uchovávali množství velkých SAT instancí, bychom mohli narazit na paměťová omezení. Jistým řešením by mohlo být

uchovávání pouze některých "vhodných" CNF, popř. jejich řešení (např. vektor s nejvíce DC).

7.3. Popis základního algoritmu

V této kapitole bude popsán základní algoritmus, který je použit pro generování komprimovaného testu. Jeho vývojový diagram je naznačen na obrázku č.20. V první fázi je provedeno načtení obvodu ze vstupního souboru. Tento obvod je ve formátu .bench (viz. příloha) a obsahuje ISCAS netlist. Dále je inicializován seznam poruch (Fault List), které mohou pro daný obvod nastat, a v případě, že není prázdný, začneme poruchy postupně pokrývat. Vybereme tedy ze seznamu první poruchu a vygenerujeme pro ni instanci SAT problému v CNF. Jestliže je instance SAT pro první vybranou poruchu nesplnitelná, tzn. testovaná porucha je nedetekovatelná, algoritmus v základní modifikaci skončí. V další verzi bylo toto ošetřeno výběrem dalších poruch ze seznamu, dokud nenalezneme splnitelnou. Po vyřešení první SAT instance máme k dispozici "základní" testovací vektor, na kterém dále rozvíjíme řešení. Po získání tohoto prvního vektoru provedeme jeho simulaci a odstraníme ze seznamu poruch všechny poruchy, které lze tímto vektorem detekovat. Nyní provedeme posun testovacího vektoru o jeden bit vlevo, popř. lze i vpravo, čímž získáme první bit (MSB resp. LSB testovacího vektoru) komprimovaného bitstreamu. Tato operace je naznačena na obrázku č. 19. Na místo, které se při posunutí uvolní vložíme don't care bit. Jedná se tedy o jakýsi posuvný registr (okénko), kde z jedné strany vysunujeme bitstream a z druhé nasunujeme DC bity. Na první pohled je zde také vidět souvislost mezi kompresí a následnou dekompresí bitstreamu.



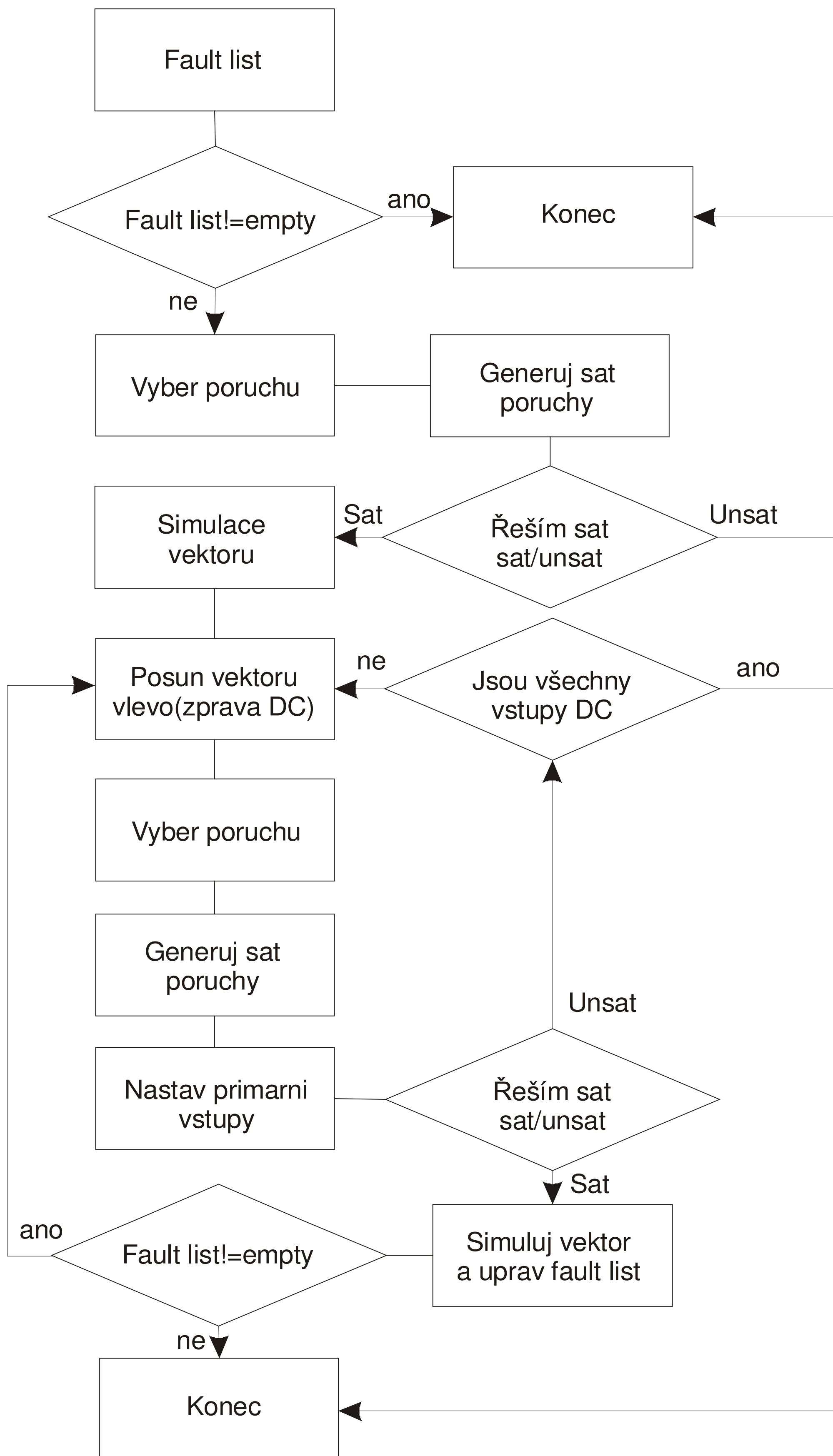
Obrázek č.19: *Příklad posunutí vektoru*

V dalším kroku procházíme seznam poruch a pro každou postupně generujeme instanci SAT problému, kterou se snažíme vyřešit. Před řešením ovšem provedeme nastavení všech proměnných reprezentujících primární vstupy v instanci, podle posunutého vektoru. Tímto nastavením dojde k částečné redukci SAT problému a případně se už nyní může projevit nesplnitelnost instance, což nám ušetří čas nutný na její další řešení. V následujícím kroku tedy provedeme řešení SAT instance redukované nastavením proměnných primárních vstupů,

a jestliže narazíme při průchodu seznamem poruch na takovou, jejíž SAT instance je pro toto nastavení splnitelná, přerušíme další procházení seznamu poruch a provedeme simulaci vektoru, který jsme získali jako řešení. Při simulaci odstraníme ze seznamu poruch všechny poruchy, které lze pomocí získaného vektoru detekovat. Dále provedeme opět posun nového testovacího vektoru a základní cyklus se opakuje. Základní cyklus opakujeme dokud není seznam poruch prázdný. Takto postupuje algoritmus v ideálním případě, nicméně jen zřídka se stává, že by zřetězení testovacích vektorů bylo tak ideální, že v každém kroku najdeme testovací vektor. Pokud projdeme seznam poruch a pro dané nastavení proměnných primárních vstupů není splnitelná ani jedna z odpovídajících SAT instancí, provedeme posunutí aktuálního vektoru o jeden bit a pokračujeme v hledání splnitelných instancí pro nové nastavení proměnných primárních vstupů. Další problém zde představují nedetekovatelné poruchy. Tyto jsou v základní verzi algoritmu po celou dobu součástí seznamu poruch a vzhledem k tomu, že neexistuje vektor, kterým bychom je mohli otestovat a následně vyřadit ze seznamu poruch, nedošlo by k vyprázdnění seznamu a vytvoří se nekonečná smyčka. Řešení tohoto problému je mnoho, nicméně pro základní algoritmus byl do programu přidán "vnitřní čítač", který hlídá, kolik bitů bylo z aktuálního vektoru vysunuto a v případě, že dojde k vysunutí celého vektoru, tzn. při řešení SAT instancí nenastavujeme žádné proměnné a přesto žádná z nich není splnitelná, jsou zbylé poruchy prohlášeny za redundantní, tzn. nedetekovatelné a program je ukončen.

Z předchozího popisu základního algoritmu je zřejmé, že k nalezení komprimovaného testu je nutné vyřešit velké množství instancí SAT problému. Dále je zde v této základní verzi velká časová režie na generování SAT instancí a "kopírování" instance z reprezentace v SAT-ATPG do MinSatu. V dalších verzích se předpokládá sjednocení reprezentace a částečné uchovávání SAT instancí, což by mělo do jisté míry snížit časovou režii, nicméně stále bude nutné řešit velké množství SAT instancí. Tento problém, lze samozřejmě dál řešit a určitě se do budoucna budeme snažit o maximální urychlení, nicméně pro nás je z hlediska testování stále nejdůležitějším parametrem stupeň komprese, kterého jsme schopni dosáhnout. To je naším primárním cílem, na který bude kladen důraz při následných optimalizacích.

K implementaci tohoto základního algoritmu je zapotřebí zajistit spolupráci generátoru SAT instance pro danou poruchu (v našem případě SAT-ATPG), simulátoru poruch, kterým je v našem případě modifikace programu Atalanta-M a SAT-solveru MinSat, kterým řešíme jednotlivé instance. Způsob jejich integrace bude popsán v další kapitole.

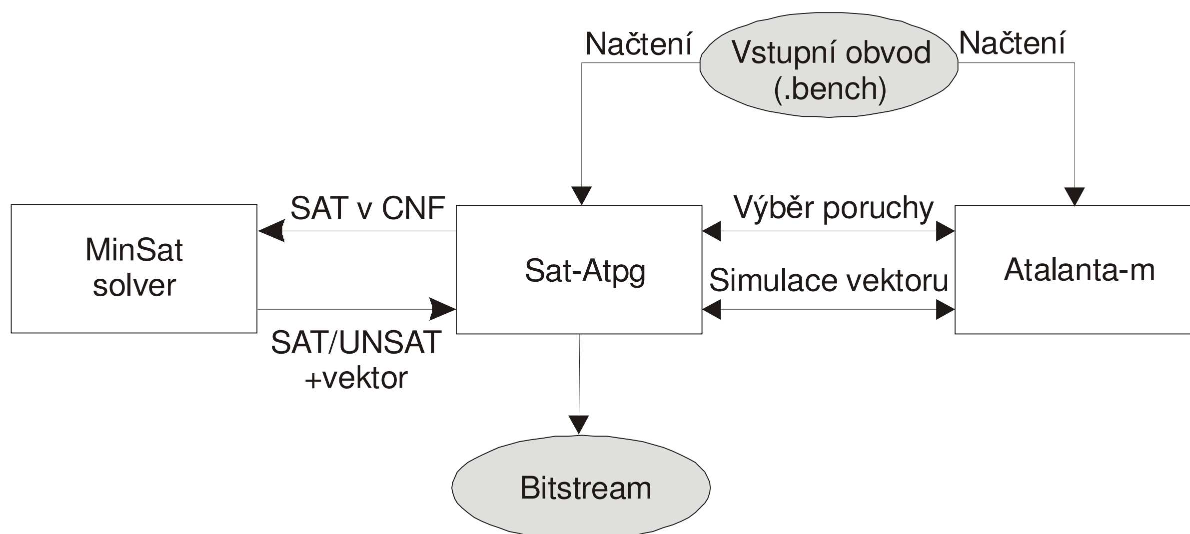


Obrázek č.20: *Základní algoritmus našeho nástroje*

7.4. Integrace součástí

Z předchozího popisu algoritmu vyplývá, že pro realizaci našeho nástroje potřebujeme realizovat tři základní operace, kterými jsou generování SAT problému (v CNF), řešení problému SAT a simulaci získaného řešení. Při jejich realizaci budeme vycházet z jichž implementovaných nástrojů Atalanta-M a SAT-ATPG, jehož součástí je i zmíněný MinSat. Hlavními problémy při vytvoření nástroje, který by umožňoval spolupráci těchto "modulů" jsou rozdíly v datové reprezentaci a použitých algoritmech.

Používáme nástroj Atalanta-M (simulátor poruch) a nástroj SAT-ATPG (generování a řešení SAT instancí), přičemž oba si po načtení vytváří seznam poruch, nicméně rozdílná reprezentace a značení poruch, resp. prvků obvodu nám nedovoluje provést sjednocení této pro nás velmi důležité struktury. Dále je také poměrně neefektivní pracovat se dvěma seznamy poruch. Vzhledem k tomu jsme se rozhodli, že nadále budeme využívat seznam poruch Atalanty-M, který je přímo při simulaci vektoru testu aktualizován, tzn. jsou z něj odstraněny všechny ekvivalentní poruchy. Modul SAT-ATPG dále používáme pouze pro vygenerování SAT instance určité poruchy a jeho součást MinSat k jejímu vyřešení. Toto řešení ovšem předpokládá, že jsme schopni vygenerovat konkrétní poruchu v SAT-ATPG podle popisu poruchy dodaného ze seznamu poruch v Atalantě. Jak již bylo ovšem dříve psáno, vzhledem k odlišným algoritmům a datové reprezentaci, bylo nemožné přenést potřebné parametry poruchy z Atalanty do SAT-ATPG a bylo tedy třeba provést na začátku algoritmu sjednocení datové reprezentace obvodu pomocí mapování indexů. V této fázi bylo nutné nastavit v SAT-ATPG stejné indexy prvků obvodu, které využívá Atalanta. Po vyřešení tohoto problému bylo již pouze nutné vyřešit předávání parametrů poruchy z Atalanty do SAT-ATPG. Každá porucha se dá v SAT-ATPG popsat trojicí odkud, kam a typ. Do nástroje Atalanta byla tedy doimplementována datová struktura sdružující tyto parametry poruchy a sada funkcí pro práci s ní. Jsou zde implementovány funkce umožňující nastavení i předání parametrů, přičemž je zde možnost vybrat vždy poruchu na počátku listu, nebo libovolnou poruchu (volbou jejího indexu). Dále byl SAT-ATPG rozšířen o funkce na bázi DP (Davis & Putnam), které provádí nastavení proměnných primárních vstupů podle vektoru, který je jim zadán jako parametr. V případě, že je pro nastavení proměnných SAT instance nesplnitelná indikují to pomocí návratové hodnoty. V neposlední řadě byly do SAT-ATPG přidány funkce realizující "okénkový" posuv vektoru popsaný v předchozí kapitole a naznačený na obrázku č.19. Blokový diagram spolupráce jednotlivých modulů je ukázán na obrázku č.21.



Obrázek č.21: *Blokový diagram spolupráce modulů*

Z blokového diagramu na obrázku č.21 je vidět, že načtení vstupního obvodu provedou oba nástroje Atalanta-M i SAT-ATPG. Každý si vytvoří svou reprezentaci obvodu a nástroj Atalanta-M také vygeneruje seznam poruch. SAT-ATPG provede mapování indexů prvků obvodu podle reprezentace obvodu v Atalantě a začíná základní cyklus. Výsledkem je poté soubor obsahující komprimovaný test vstupního obvodu, pokrývající všechny detekovatelné poruchy ve formě bitstreamu.

7.5. Možná vylepšení základního algoritmu

Výše popsaný základní algoritmus je schopen dosáhnout poměrně pěkných výsledků, jak bude dále ukázáno v kapitole testování. Zamyslíme-li se nad vlastním algoritmem, a uvědomíme-li si jakým způsobem pracuje, dojdeme jistě k přesvědčení, že má bezesporu potenciál k dosažení mnohem lepších výsledků. Bohužel, nebylo v této práci z časových důvodů možné implementovat sofistikovanější verzi algoritmu schopnou dosáhnout lepších výsledků v kratším čase, ale pokusili jsme se tento algoritmus rozšířit o několik jednoduchých modifikací, vycházejících z teoretických předpokladů, které by měli vést k zlepšení výsledků. Tyto budou dále popsány a jejich případný přínos bude zhodnocen v kapitole testování.

7.5.1. Řazení poruch

Základní verze algoritmu vychází ze seznamu poruch, který je na počátku vygenerován Atalantou. Řazení tohoto seznamu poruch je tedy dáno algoritmem, který Atalanta používá k jeho sestavení. Vzhledem k tomu, že při hledání testovacího vektoru začínáme seznam poruch prohledávat od začátku, přičemž pro každou vygenerujeme SAT instanci, jež se

snažíme vyřešit, nabízí se myšlenka uspořádat seznam poruch podle nějakého pravidla. Zamyslíme-li se nad touto problematikou, představuje pro nás seznam poruch z hlediska SAT-ATPG seznam instancí SAT problémů. Z tohoto pohledu také můžeme vycházet při sestavování ohodnocení jednotlivých poruch. Mohli bychom zde vyjít například z velikosti SAT instance, přičemž budeme předpokládat např. že nejprve zpracujeme malé instance, což půjde rychle a přitom může dojít k eliminaci velkých, což může urychlit řešení. Musíme si zde ovšem uvědomit, že řešením malých instancí budou vektory obsahující velké množství DC a tyto by se nám z hlediska komprese hodily spíše v pozdější fázi, kdy hledáme vhodný navazující vektor, vzhledem k tomu, že nejsou úplně definovány. Dále je třeba si uvědomit, že pokrytí poruch takto řídkých vektorů bude menší, než pokrytí plnějších vektorů. Z hlediska generování testu by to následně znamenalo, že budeme muset nalézt překrytí více vektorů testu, což se samozřejmě může negativně projevit na výsledné kompresi. Nabízí se zde tedy opačný přístup a to řazení SAT instancí od největších po nejmenší. Zde je třeba ovšem brát v úvahu, že budeme pracovat převážně s plnými vektory a tudíž bude složitější najít vhodné překrytí. Dále je třeba si uvědomit, že vzhledem k tomu, že nejsložitější problémy budou na začátku seznamu poruch, budou zpracovávány vždy jako první, což může mít negativní vliv na dobu řešení. Výhoda naopak může spočívat v tom, že takto plně určené vektory budou pokrývat mnohem větší množství poruch než řídké vektory a k pokrytí detekovatelných poruch bude zapotřebí provést překrytí mnohem menšího počtu vektorů. Dále mohou být poruchy reprezentovány složitými SAT instancemi na počátku listu pokryty mnohem dříve a budeme poté muset pracovat pouze s malými SAT instancemi, které produkují řídké vektory, pro které je mnohem jednodušší nalézt vhodné překrytí.

Abychom prozkoumali chování algoritmu pro předchozí úvahy, provedli jsme rozšíření datové struktury reprezentující poruchu v seznamu poruch o proměnnou cena, která obsahuje ohodnocení poruchy a bylo implementováno několik funkcí, pomocí kterých provádíme nastavení ceny poruch a následné seřazení seznamu poruch. V našem případě ovšem nevycházíme při hodnocení poruch z velikosti SAT instancí, protože tento přístup by mohl být poměrně zavádějící, ale hodnotíme jednotlivé poruchy podle zastoupení primárních vstupů v jejich SAT instancích, tzn. jestliže jedna SAT instance poruchy má 2500 proměnných, 10000 termů a obsahuje 10 primárních vstupů, bude její ohodnocení pouze 10, protože vektor, který jejím řešením získáme bude obsahovat 10 nastavených hodnot a zbylé budou DC. Je tedy možné, že i menší SAT instance budou řazeny před většími, což nám ovšem vůbec nevadí. Nás zajímá pouze nastavení primárních vstupů a podle jejich zastoupení

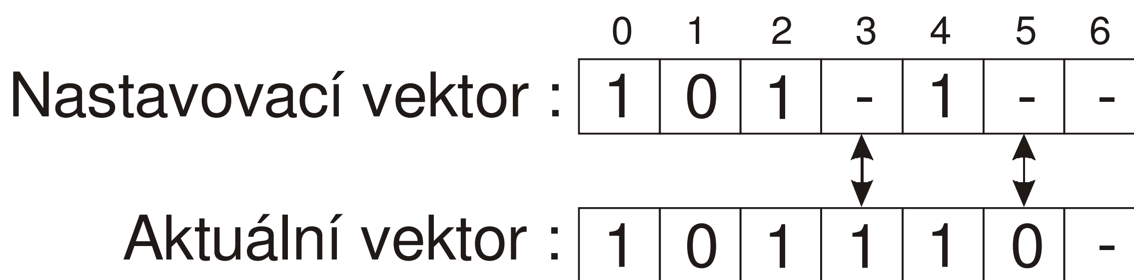
tedy hodnotíme tzv. velikost SAT instance (je ovšem pravděpodobné, že větší SAT instance budou obsahovat větší počet proměnných primárních vstupů).

7.5.2. Přidání DC stavů

Další vylepšení, které by mohlo přispět ke zvýšení komprese je zvýšení počtu don't care (DC) stavů ve vektorech, které se snažíme překrýt. Vzhledem k tomu, že neexistuje dostatečně rychlý SAT solver, pomocí kterého bychom byly schopni vygenerovat vektor s co největším počtem DC stavů, budeme nuceni přidávat DC pomocí několika dalších operací, což se samozřejmě negativně projeví na době řešení. Pokud bychom ovšem dosáhli většího zastoupení DC stavů, zjednodušili bychom hledání dalšího vektoru, který vyhovuje podmínkám nastavení daných předchozím vektorem, což by mohlo vést k lepšímu překrytu vektorů a vyšší kompresi. Je ale třeba zvážit také negativní dopady, které mohou vést naopak ke zhoršení vlastností nástroje. Jedná se především o možné značné zvýšení doby řešení a paradoxně může dojít také ke zhoršení komprese. Nesmíme totiž zapomínat na skutečnost popsanou již dříve, tj. méně určené vektory pokrývají menší množství poruch. Nastavením některých bitů na hodnoty DC tedy můžeme zvýšit počet vektorů, které je třeba k pokrytí všech poruch a tím pádem může v některých případech docházet ke zhoršení komprese.

Základní myšlenka přidání DC do vektoru řešení SAT instance spočívá v tzv. flipování bitů. Postupně provedeme pro každý určený bit testovacího vektoru změnu jeho "polarity", tzn. $0 \rightarrow 1$ resp. $1 \rightarrow 0$, a pro toto nové nastavení provedeme ověření certifikátu pro odpovídající instanci SAT problému. Jestliže je pro upravené nastavení instance SAT splnitelná, nastavíme flipovaný bit na hodnotu DC. V opačném případě vrátíme bit na původní hodnotu. Stejným způsobem otestujeme všechny bity vektoru. Takto lze v obecném případě dosáhnout poměrně značného nárůstu zastoupení DC ve vektoru. V našem případě je situace bohužel poněkud komplikována tím, že nepracujeme s vektorem reprezentujícím všechny proměnné v SAT instanci, ale pouze s podmnožinou reprezentující primární vstupy. Bohužel tedy nelze aplikovat přístup dříve popsaný a musíme použít poněkud komplikovanější postup. Přidávání DC stavů do vektoru provádíme v základním algoritmu vždy před jeho simulací a používáme k němu upravený algoritmus DP (Davis & Putnam [9]). Je jasné, že v novém vektoru nelze nastavovat na DC všechny proměnné, ale pouze ty, které byly při nastavování před řešením SAT instance nastaveny jako DC. Ostatní musí zůstat nastaveny na hodnoty definované předchozími vektory, aby zůstalo zachováno správné překrytí. Příklad tohoto je naznačen na obrázku č.22. Nastavovací vektor zde představuje

vektor po vysunutí jednoho bitu do výsledného bitstreamu a nasunutí DC. Pomocí tohoto vektoru nastavíme primární proměnné v SAT instanci a jejím řešením získáme "aktuální vektor". Srovnáním aktuálního a nastavovacího vektoru zjistíme, že se můžeme pokusit nastavit na DC pouze bity 3 a 5. Vygenerujeme si tedy znovu odpovídající SAT instanci a z ní odstraníme všechny literály odpovídající proměnné reprezentované bitem č.3. Dále provádíme také nastavení proměnných na hodnoty dané vektorem. Jestliže při těchto úpravách detekujeme, že vedou na nesplnitelnost, ukončíme jejich provádění a indikujeme, že nelze nastavit proměnnou na DC. V případě, že se při úpravách neprojeví nesplnitelnost, je nutné vyřešit redukovanou SAT instanci MinSatem, a v případě, že je splnitelná nastavit testovaný bit na DC. V opačném případě nastavíme bit na původní hodnotu. Takto postupujeme pro všechny bity. Určujeme tak tedy, které ze skupiny testovaných bitů lze pro dané nastavení ze SAT instance odebrat, aniž by se to projevilo na její splnitelnosti.

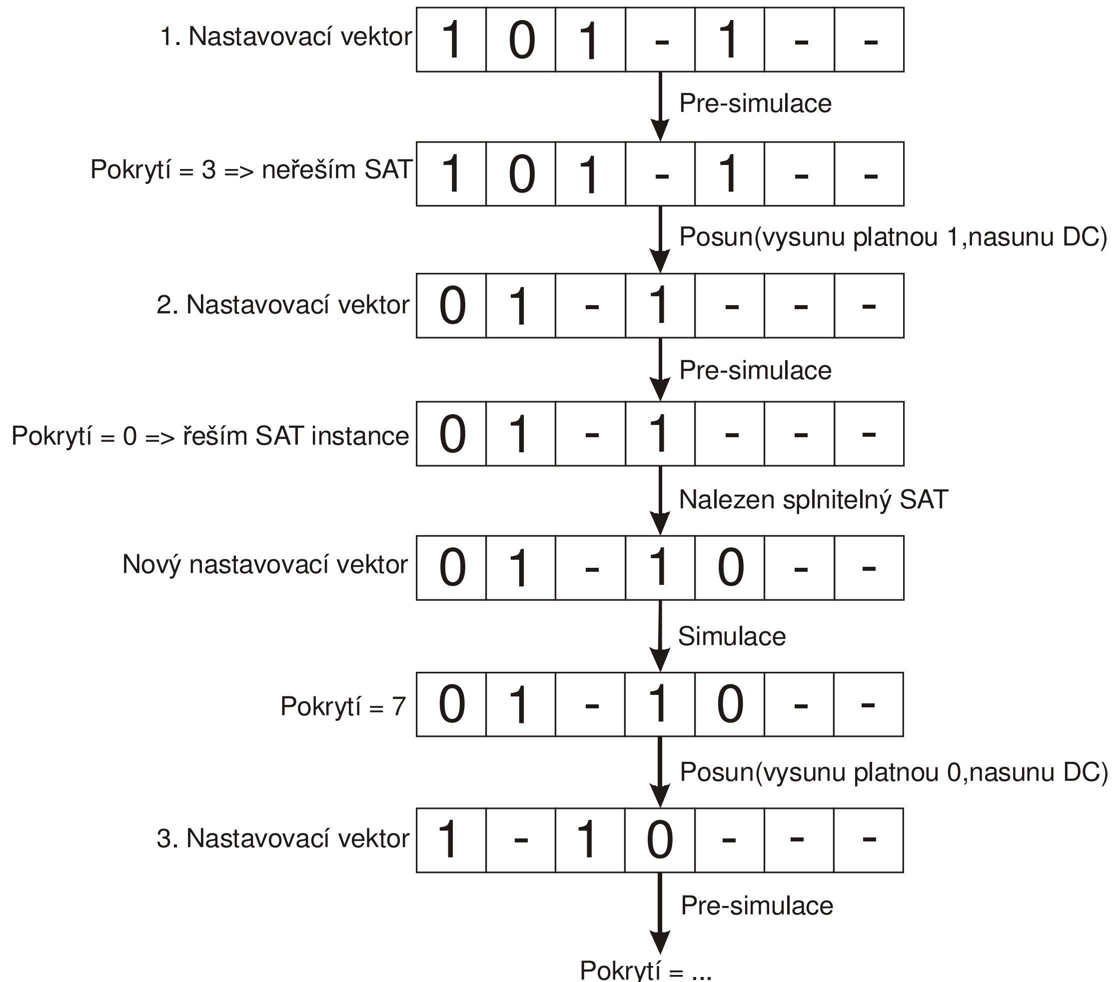


Obrázek č.22: *Příklad bitů, které lze testovat na DC*

7.5.3. Jednoduchá "pre-simulace"

Ačkoli jsme už pomocí základního algoritmu schopni dosáhnout poměrně slušných výsledků, stále jsme bohužel odkázáni na cyklické řešení SAT problému, což může pro složité SAT instance vést k poměrně dlouhé době řešení. Hlavní problém pro nás ovšem představují situace, ve kterých neexistuje navazující vektor a dochází k posunutí o větší množství bitů. V takovémto případě musíme pro každý vysunutý bit projít celý seznam poruch a pro každou poruchu vyřešit jí odpovídající SAT instanci, přičemž pouze potvrdíme, že žádná z nich není splnitelná a testované překrytí tedy neexistuje. Vyvstává tedy otázka, jeli nutné pokaždé procházet všechny SAT instance a není-li možné hned po získání nového vektoru, tzn. po vysunutí prvního bitu rozhodnout, že pro toto nastavení neexistuje splnitelná SAT instance. Tímto by se nám situace velmi zjednodušila. Bohužel vzhledem k časovým možnostem nebylo možno implementovat algoritmus schopný řešit tento problém. Zamysleme se ovšem nyní nad situací opačnou. Získáme nový vektor nastavení primárních proměnných v SAT instancích. Tento nový vektor obsahuje největší možný počet DC stavů, jaký jsme schopni

získat. Při následném řešení SAT instance dojde již pouze k nastavení některých DC na konkrétní hodnoty, přičemž nalezení instance splnitelné pro dané ohodnocení může trvat poměrně dlouho. Tuto prodlevu řešíme pomocí dalšího rozšíření základního algoritmu.



Obrázek č.23: *Příklad práce s testovacími vektory při použití pre-simulace*

Nové rozšíření základního algoritmu spočívá v přidání tzv. "pre-simulace". Příklad práce s vektory ukazuje obrázek č. 23. Algoritmus upravíme tak, že do fáze po vysunutí platného bitu a nasunutí DC, kdy máme k dispozici nový "nastavovací" vektor provedeme jeho simulaci. Jestliže zjistíme, že vektor pokrývá nějakou poruchu/y, jsou tyto vyřazeny ze seznamu poruch a celá fáze prohledávání SAT instancí je přeskočena. V dalším kroku se provede opět vysunutí platného bitu a nasunutí DC a opět je provedena simulace. V případě, že nový "nastavovací" vektor nepokrývá žádnou poruchu, musíme se vrátit k základnímu cyklu tzn. procházet seznam poruch a hledat SAT instanci splnitelnou pro dané ohodnocení.

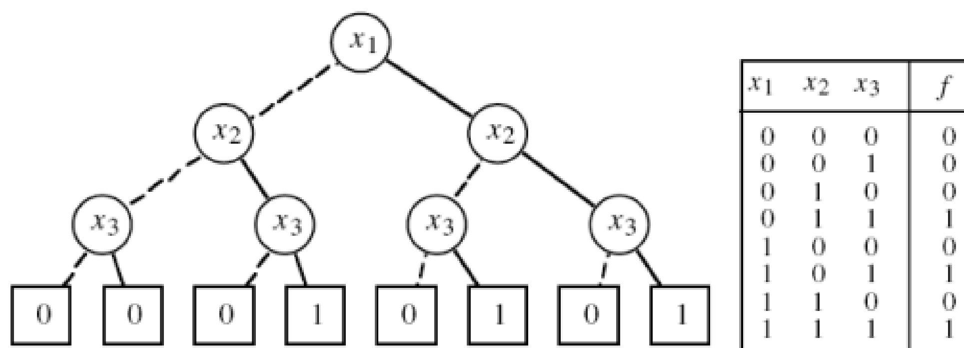
Na první pohled by se mohlo zdát, že pro případ, kdy v "pre-simulaci" zjistíme, že nastavovací vektor nepokrývá žádnou poruchu, znamená to, že pro toto nastavení neexistuje splnitelná SAT instance a nemá cenu tedy procházet seznam poruch a pokoušet se řešit jim

odpovídající SAT instance. Naopak v případě, kdy vektor s daným nastavením pokrývá nějaké poruchy, měli bychom se pokusit řešit SAT instance a nalézt tak nastavení zbylých DC a tímto způsobem se pokusit maximalizovat počet poruch, které daný vektor pokrývá. Bohužel takto nelze k problému přistupovat vzhledem k tomu, že nastavovací vektor, jak již bylo výše řečeno, obsahuje maximální počet DC stavů a tudíž, i když tento vektor nedetekuje žádnou poruchu, je zde velká pravděpodobnost, že při nastavení některého z DC bitů se tato situace změní. Pokud bychom tedy postupovali podle varianty popsané v tomto odstavci, došlo by brzy k vysunutí celého vektoru a zbylé poruchy by byly prohlášeny za nedetekovatelné. Tato úvaha zde je tedy uvedena pouze jako naznačení možné práce do budoucna. Pokud bychom totiž byli schopni při simulaci nějakým způsobem zpracovat DC bity a dokázali bychom takto detekovat alespoň, že pro určité nastavení existuje/neexistuje porucha, kterou může pokrývat (s tím že nastavíme některé DC), mohli bychom tímto způsobem vyřešit problém vysunování bitů v případě, že všechny SAT instance jsou nesplnitelné a zároveň bychom mohli pracovat na případném zlepšení komprese zvýšením počtu poruch, které jednotlivé vektory detekují, resp. snížením počtu testovacích vektorů nutných k pokrytí detekovatelných poruch. Naivním přístupem bychom mohli tento problém vyřešit například tak, že provedeme pre-simulaci pro všechna možná nastavení DC v nastavovacím vektoru. Pro většinu testovacích vektorů by tato varianta šla bez problémů realizovat vzhledem k tomu, že většinou pracujeme s poměrně zaplněnými vektory, ale pro větší instance je toto řešení nepřijatelné. Představme si například obvod s tisícem primárních vstupů, z nichž na počátku algoritmu můžeme získat velmi řídký vektor, pro který bychom měli simulovat všechny nastavení DC. Složitost takovéto pre-simulace by pravděpodobně způsobila neřešitelnost problému v "reálném" čase. V případě použití podobného přístupu by ovšem mohlo být zajímavé vypustit cyklus řešení SAT instancí, vygenerovat první vektor a poté pracovat již jen se simulátorem dle předchozího popisu (nesnažili bychom se tedy maximalizovat počet pokrytých poruch na vektor).

7.5.4. BDD a možnosti využití

Binární rozhodovací diagramy - BDD (Binary Decision Diagrams) [5] jsou jednou z mnoha reprezentací logické funkce. Mezi další patří např. pravdivostní tabulka nebo konjunktivní, popř. disjunktivní normální forma. Tyto další formy reprezentace jsou ovšem poměrně nevhodné pro reprezentaci větších logických funkcí.

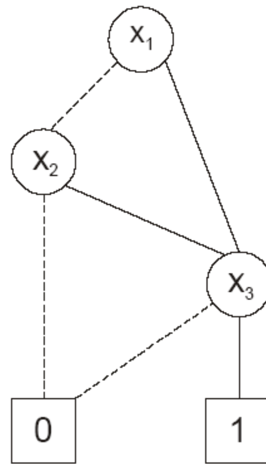
Binárním rozhodovacím diagramem pro Booleovskou funkci $f(x_0,x_1, \ldots ,x_{n-1})$ je acyklický orientovaný graf (strom) s jedním počátečním uzlem, a vnitřními uzly, které představují jednotlivé proměnné a jedním, popř. dvěma koncovými uzly (terminály), které jsou většinou ohodnoceny logickými hodnotami 0,1. Každý nekoncový uzel má dva následníky nazývané 0-následník a 1-následník, které jsou rozlišené ohodnocením příslušných hran. Obrázek č.24 ukazuje BDD a jemu odpovídající pravdivostní tabulku [5].



x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Obrázek č.24: *BDD a jemu odpovídající pravdivostní tabulka [5]*

Z obrázku č.24 je vidět, že BDD v základním tvaru stále nejsou pro reprezentaci nejefektivnější. Obsahují totiž spoustu redundantních uzlů. V praxi je tedy výhodnější použít jejich modifikaci tzv. redukované uspořádané binární rozhodovací diagramy (Reduced Ordered Binary Decision Diagrams - ROBDD). Tyto ROBDD vzniknou z BDD odstraněním redundantních prvků a seřazením proměnných do vhodného pořadí. Ukázka ROBDD vzniklého z BDD na obrázku č.24 je na obrázku č.25. Je zde vidět, že oproti původnímu BDD, který obsahoval 15 uzlů, obsahuje ROBDD pouze 5 uzlů. Dále si všimněme, že i při této nové reprezentaci realizuje ROBDD všechny vektory uvedené v tabulce na obrázku č.24. Toto lze ověřit jednoduchým průchodem ROBDD od kořene k terminálnímu uzlu přičemž jdeme-li přes plnou hranu, je daná proměnná nastavena na logickou 1. Procházíme-li přes přerušovanou hranu, je proměnná nastavena na logickou 0. Nevyskytuje-li se proměnná na cestě do terminálu, tak na ní nezáleží, tzn. proměnná je nastavena na dont't care.



Obrázek č.25: *Ukázka ROBDD vzniklého odstraněním redundancí [5]*

Z předchozího popisu je zřejmé, že tímto způsobem lze jednoduše realizovat SAT řešič, schopný generovat vektory s co největším počtem DC. Stačí pouze na základě reprezentace obvodu sestavit jeho ROBDD a poté pomocí grafových algoritmů najít nejkratší cestu z kořene do jedničkového terminálu. Pokud bychom byli schopni vytvořit takovýto SAT řešič, mohl by nám značně zjednodušit situaci, a dá se předpokládat, že bychom taktéž dosáhli mnohem větší komprese než při použití MinSatu.

Bohužel v dnešní době nemáme k dispozici SAT řešič, který by byl použitelný v naší aplikaci. Překážkou totiž zůstává tvorba ROBDD z popisu obvodu. Ačkoli z předchozích obrázků se může zdát, že je tento problém poměrně triviální, nepodařilo se dosud implementovat algoritmus, který by nám umožňoval práci s většími obvody. Při konstrukci ROBDD totiž počet uzlů výrazně závisí na pořadí proměnných. S rostoucím počtem uzlů roste také složitost operací, které s nimi provádíme a paměťová náročnost. V praxi se mi tak podařilo s použitím několika heuristik, které dosti zásadně snižovaly počet uzlů v ROBDD vyřešit pouze malé a pro nás nezajímavé instance SAT problému [5]. V budoucí práci by ovšem určitě stálo za úvahu pokusit se takovýto řešič implementovat, resp. rozšířit stávající řešení o nové heuristiky. Pokud bychom totiž byli schopni realizovat optimální ROBDD, stačilo by po konstrukci ROBDD pouze jednoduchým průchodem ověřit splnitelnost/nesplnitelnost, popř. přímo při průchodu nalézt nejlepší vektor pro překryv. Při použití MinSatu jsme bohužel odkázáni na reprezentaci v CNF, přičemž pokaždé musíme provést nastavení proměnných pomocí modifikace DP algoritmu a poté upravenou instanci vyřešit, což celkem prodlužuje dobu řešení a výsledky bohužel neobsahují takové množství DC jaké bychom si představovali. Toto prozatím kompenzuje poměrně velká rychlost MinSatu.

8. Nástroj COMPAS

Tento nástroj stál u zrodu myšlenky, která vedla k realizaci našeho generátoru komprimovaných vzorků a proto je logické, že náš nový přístup k problému komprese porovnáváme právě s přístupem, který implementuje COMPAS. Následující popis tohoto nástroje byl převzat z [14].

COMPAS – Compressed Test Pattern Sequencer je software pro kompresi testovacích vektorů pro obvody se sériovým diagnostickým přístupem. Program slouží pro kompresi testovacích dat pro kombinační obvody nebo pro full-scan obvody. Je zaměřen na zpracování testovacích dat pro model trvalých poruch, přičemž využívá popis obvodu na úrovni hradel a simulátor poruch. Vstupní data mají formu dvojic, které jsou tvořeny nekomprimovaným testovacím vektorem a příslušnou poruchou. Je vhodné, aby testovací vektor obsahoval co největší množství nespecifikovaných bitů (tzv. DC – don't care bitů), protože ty lze nejsnáze překrýt.

Algoritmus obecně umožňuje kompresi libovolných testovacích vektorů. Vektory mohou obsahovat jen specifikované bity, pak je ale obtížné je překrýt a komprese nedosahuje dobrých výsledků. Vektory mohou být vytvořeny pro testování jiných než trvalých poruch. V tom případě ale není možné využít úzké spolupráce se simulátorem poruch, a je tak dosaženo horší komprese. Simulátor není možné použít ani v případě, že není k dispozici popis obvodu.

Kompresor COMPAS je platformě nezávislý, testován byl na platformách x86, x86-64 a sparc, pod operačními systémy Windows XP, Linux a SunOS ve verzích pro 32 i 64 bitů. Zdrojový kód je v jazyce C.

8.1. Komprese

Jako vstupní data pro program slouží nekomprimované testovací vektory obsahující nespecifikované vstupní bity (don't care bity). Vzorky mají formu dvojice porucha – testovací vektor, tedy každé poruše přísluší právě jeden testovací vektor s co nejvíce nespecifikovanými bity. Výstupem programu je sekvence bitů (nul a jedniček), neobsahující žádné nespecifikované bity, která tvoří komprimovanou testovací posloupnost.

Metoda komprese spočívá v postupném hledání testovací posloupnosti po jednom bitu z předem připravené sady testovacích vektorů – úplného testu. Pro kombinační obvody tato posloupnost při vlastním testu vstupuje do skenovací smyčky, jejíž velikost je stejná jako

počet vstupů obvodu, pro sekvenční obvody je vhodné použít například výše popsanou testovací architekturu RESPIN [7].

Na začátku algoritmu máme seznam zatím nedetekovaných poruch obvodu, seznam všech testovacích vektorů tvořících úplný test a ve skenovací smyčce jsou samé nuly. Krok algoritmu pak probíhá tak, že je nalezen další jeden bit posloupnosti, dále je provedena simulace a ze seznamu poruch jsou odstraněny ty, které jsou simulací detekovány. Algoritmus končí tehdy, když je seznam nedetekovaných poruch prázdný.

Kompresní algoritmus COMPASu začíná ze známého stavu, kterým je většinou vynulovaný skenovací řetězec. Dá se totiž předpokládat, že test obvodu začíná zapnutím napájení a vynulováním. Komprese ale může v případě nutnosti začít i s jiným stavem skenovacího řetězce.

Použitý algoritmus nepoužívá zpětné prohledávání, nevrací se k bitům vygenerovaným v předchozích krocích. Jakmile je jednou komprimovaný bit vygenerován, je zařazen do výsledné posloupnosti a už není nikdy změněn. Obecně by kompresní algoritmus mohl vyzkoušet všechna myslitelná pořadí seřazení vektorů, tím by se kompletně prohledal stavový prostor a bylo by nalezeno optimální řešení. Hledání by ale trvalo příliš dlouho, protože by bylo nutné vyzkoušet všechny permutace pořadí vektorů.

Tím, že nedochází k návratu k předchozím bitům, dochází k výraznému zrychlení algoritmu. Algoritmus pracuje tak, že v každém kroku ohodnotí kritériem všechny možnosti a vybere aktuální lokální minimum s tím, že existuje šance, že trvalým vybíráním lokálního minima bude nalezeno minimum globální. Jedná se tedy o hladový algoritmus (greedy search). Nalezené řešení není nejmenší možné, je ale dostatečně kvalitní.

9. Testování a srovnání výsledků

V této kapitole budou prezentovány výsledky testování základního algoritmu a několika jeho modifikací. Pro testování je třeba si uvědomit, že náš algoritmus vygeneruje počáteční vektor (vyřešením SAT instance první poruchy) a na tento základ navazuje další vektory tak, aby dosáhl nejlepšího překrytí. Počáteční vektor je tedy jedním z důležitých parametrů, který může nasměrovat algoritmus správným nebo naopak špatným směrem. Je zřejmé, že pro různé počáteční vektory dosáhneme různého překrytí a tudíž rozdílné komprese. Je zde tedy obtížné provést srovnání několika přístupů pouze na základě testování pro jednu výchozí poruchu, resp. vektor získaný vyřešením jí odpovídající SAT instance. Dále je třeba si taktéž uvědomit, že stejná situace nastává při nastavení DC na určitou hodnotu, popř. přidáním DC stavu. Tímto způsobem změníme "cestu" mezi prvním a posledním přidávaným vektorem a ovlivníme tak překrytí a celou kompresi. Z tohoto důvodu jsme prováděli testování vždy pro "všechny" možné počáteční vektory, resp. pro všechny možné "výběry" první poruchy. Neuvažujeme tedy naprosto všechny testovací vektory, které první poruchu pokrývají, ale pouze jeden testovací vektor z množiny řešení její SAT instance (máme tedy tolik "počátečních" vektorů, kolik je poruch obvodu). Pro každý testovaný obvod jsme provedli tolik měření, kolik poruch obsahuje jeho seznam poruch a získali jsme takto množinu výsledků, kterou je nutné dále zpracovat. Vyhodnocení provádíme srovnáním maximálních, minimálních a průměrných hodnot.

Máme-li např. benchmark s1196, který obsahuje 1242 poruch, provedeme testování pro výběr první poruchy 0-1241. Jedno vyřešení instance trvá přibližně 270s tzn. celková doba testování této instance pro jednu metodu je 1242*270 =335340s, což jsou téměř 4 dny. Z tohoto důvodu byla pro kompletní otestování vybrána skupina pěti "vhodných" úloh, které by měli demonstrovat možnosti, ale také nedostatky, na kterých by mohlo být patrné případné zlepšení po modifikaci základního algoritmu.

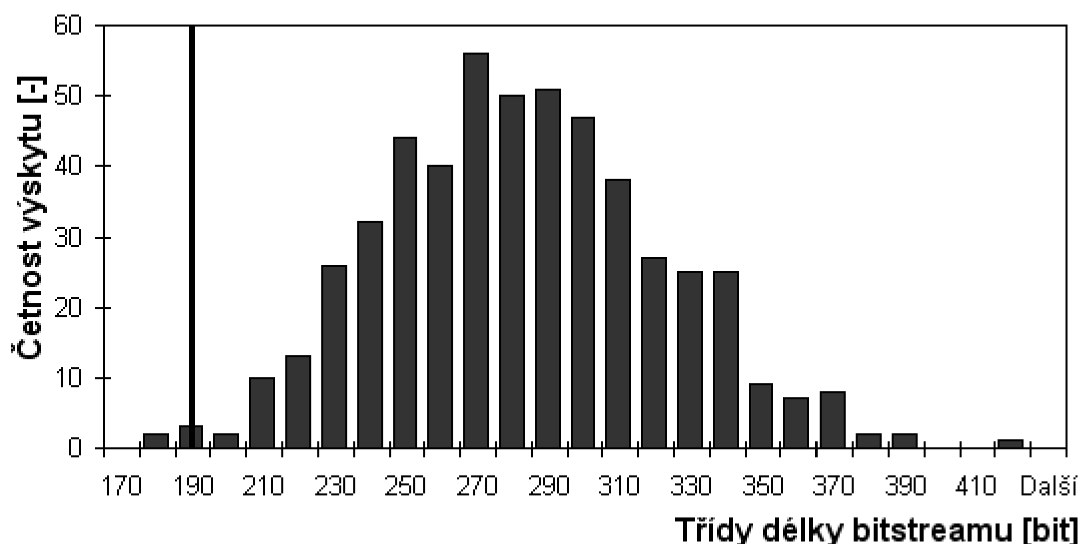
9.1. Základní algoritmus

Tato kapitola prezentuje výsledky dosažené základním algoritmem popsaným v kapitole 7.3. Výsledky měření ukazuje tabulka č.3.

Benchmark			c432	c499	c880	s1196	c1355
COMPAS		[bit]	195	224	412	717	1040
Náš nástroj - Základní algoritmus	Max	[bit]	413	239	1101	1856	388
	Min	[bit]	171	165	539	1376	280
	Průměr	[bit]	278,26	201,69	803,35	1654,50	334,73
	Počet vektorů	[-]	79	68	117	226	103
	Velikost testu(A-m)	[bit]	1872	2214	3180	4544	3444
	Nekomprimováno	[bit]	2853	2786	7048	7221	4239
	Komprese(nas test)	[%]	90	93	89	77	92
	Komprese(vs. A-m)	[%]	85	91	75	64	90
	Čas	[s]	12,21	7,06	40,39	273,36	60,66

Tabulka č.3 *: Výsledky testování základního algoritmu*

Horní řádek tabulky udává 5 benchmarků ze sady ISCAS, pro které bylo kompletní testování provedeno. Další řádek pak obsahuje délku bitstreamu, který generoval nástroj COMPAS, s nímž provádíme srovnání. Dále je zde prezentována maximální, minimální a průměrná hodnota bitstreamu generovaného naším nástrojem pomocí základního algoritmu a počet vektorů, které jsme museli překrýt (počet kroků testu). Další položkou je "velikost testu (A-m)", která udává velikost nejmenšího testu, který bychom vygenerovali pomocí nástroje Atalanta-M a dále položka "nekomprimováno", která reprezentuje velikost našeho nekomprimovaného testu. Nejdůležitější je pro nás ovšem komprese, které jsme dosáhli. Ta je vyhodnocována vzhledem k našemu testu a také vzhledem k testu, který jsme vygenerovali Atalantou. Položka čas zde reprezentuje průměrný čas nutný k vygenerování komprimovaného testu.



Obrázek č.26: *Histogram pro základní algoritmus*

Všimněme si, že náš algoritmus i v této základní verzi dosahuje poměrně slušné komprese a např. pro benchmark c1355 dosahuje mnohem lepších výsledků než nástroj COMPAS. Dále si ovšem také všimněme, že např. pro benchmark s1196 dosahuje naopak výsledků mnohem horších a to i v porovnání s minimální délkou bitstreamu. Za povšimnutí stojí také benchmark c432, jehož průměrná délka bitstreamu je sice mnohem delší než u nástroje COMPAS, ale minimální bitstream naznačuje, že je zde určitě možnost zlepšení. Obrázek č.26 ukazuje možnou reprezentaci množiny řešení pro benchmark c432 formou histogramu. Je zde vidět rozložení četností délek bitstreamů jednotlivých řešení a tučnou čárou je naznačena délka bitstreamu, které dosáhl COMPAS.

9.2. Základní algoritmus rozšířen o DC

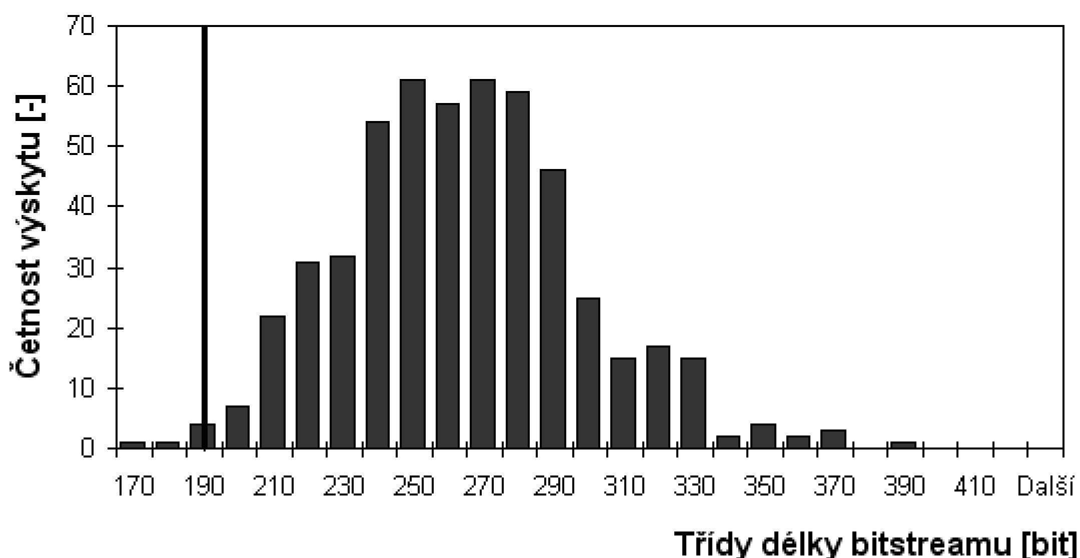
V této kapitole si předvedeme výsledky měření pro základní algoritmus rozšířený o "přidávání" DC stavů do testovacích vektorů. Tato modifikace algoritmu je blíže popsána v kapitole 7.5.2. Naměřené hodnoty prezentuje tabulka č.4.

Benchmark			c432	c499	c880	s1196	c1355
COMPAS		[bit]	195	224	412	717	1040
Náš nástroj - Základní algoritmus	Max	[bit]	413	239	1101	1856	388
	Min	[bit]	171	165	539	1376	280
	Průměr	[bit]	278,26	201,69	803,35	1654,50	334,73
	Počet vektorů	[-]	79	68	117	226	103
	Velikost testu(A-m)	[bit]	1872	2214	3180	4544	3444
	Nekomprimováno	[bit]	2853	2786	7048	7221	4239
	Komprese(nas test)	[%]	90	93	89	77	92
	Komprese(vs. A-m)	[%]	85	91	75	64	90
	Čas	[s]	12,21	7,06	40,39	273,36	60,66
Náš nástroj - Základní algoritmus+DC	Max	[bit]	388	230	1188	1554	394
	Min	[bit]	170	162	474	1221	269
	Průměr	[bit]	259,81	195,56	820,70	1391,56	323,14
	Počet vektorů	[-]	78	68	123	239	103
	Velikost testu(A-m)	[bit]	1872	2214	3180	4544	3444
	Nekomprimováno	[bit]	2806,06	2784,88	7357,90	7653,75	4236,46
	DC testováno	[bit]	290,28	209,47	894,45	1757,95	345,10
	DC nastaveno	[bit]	39,04	12,98	108,14	407,14	21,02
	DC test/nastav	[%]	15,54	6,61	13,75	30,14	6,49
	Komprese(nas test)	[%]	91	93	89	82	92
	Komprese(vs. A-m)	[%]	86	91	74	69	91
	Čas	[s]	11,71	7,42	52,87	260,32	57,66

Tabulka č.4 *: Výsledky testování základního algoritmu + přidávání DC*

Po prozkoumání tabulky č.4 můžeme dojít k závěru, že ve většině testovaných případů se přidání DC stavů do testovacích vektorů pozitivně projevilo na výsledné kompresi a dokonce nevedlo ani k prodloužení doby řešení. Při vyhodnocování věnujeme pozornost také novým

položkám “DC testováno“ (kolik bitů jsme se pokoušeli nastavit), “DC nastaveno“ (kolik z testovaných bitů bylo nakonec nastaveno na DC) a “DC test/nastav“ (kolik procent z testovaných bylo nastaveno na DC). Z analýzy těchto hodnot vyplývá, že pouze velmi malé procento testovaných bitů bylo nakonec možné jako DC nastavit. Vzhledem k tomu pravděpodobně také nedošlo k většímu zlepšení výsledků. Tento stav bude nejspíše zapříčiněn poměrně značným ořezáním CNF, nastavením definovaných hodnot primárních vstupů. Díky tomuto ořezání se zmenší množina řešení a eliminace každé další proměnné vede na nesplnitelnost SAT instance. Díky předchozímu tvrzení je také pravděpodobně nesplnitelnost detekována již při pokusu o nastavení proměnných a není posléze nutné řešit SAT, díky čemuž nedojde ke zhoršení doby řešení.



Obrázek č.27: *Histogram pro základní algoritmus + přidávání DC*

Srovnáme-li histogram pro základní algoritmus rozšířený o DC, který je na obrázku č.27 s histogramem pro základní algoritmus z obrázku č.26, je na první pohled vidět, že přidáním DC došlo k posunu rozložení četností doleva a tím pádem ke zlepšení výsledné komprese. Totéž naznačuje také průměrná délka bitstreamu v tabulce č.4.

9.3. Základní algoritmus rozšířen o DC a řazení poruch

Tato kapitola nám představí výsledky měření pro základní algoritmus rozšířený o přidávání DC do vektorů a řazení seznamu poruch. Algoritmus, který používáme je popsán v kapitole 7.5.1. Kromě přidávání DC do testovacích vektorů se zde ověřuje vliv řazení seznamu poruch na výslednou kompresi a dobu potřebnou k řešení problému. Seznam poruch

je řazen podle zastoupení primárních vstupů v jim odpovídajících SAT instancích a to buď vzestupně nebo sestupně. Vycházíme zde z předpokladu, že SAT instance s větším počtem primárních vstupů budou produkovat plnější vektory, zatímco v opačném případě budou vektory obsahovat větší množství DC. Testujeme zde tedy, je-li výhodnější překrývat nejprve plnější testovací vektory a poté řídké testovací vektory nebo překrývat nejprve řídké testovací vektory a poté plnější testovací vektory.

9.3.1. Řazení poruch "max-min"

Rozšířením základního algoritmu o řazení seznamu poruch by podle rozboru a popisu takto vylepšeného algoritmu v kapitole 7.5.1 mohlo vést k dalšímu zlepšení komprese. Předpokládáme zde totiž, že budeme-li přednostně zpracovávat SAT instance s větším zastoupením primárních vstupů je pravděpodobné, že dojde k pokrytí jim odpovídajících poruch a v konečné fázi, kdy je hledání vhodného testovacího vektoru pro překrytí nejtěžší, nám zůstanou relativně řídké vektory, pro které by mělo být mnohem snazší nalézt optimální překryv. Je ovšem také možné, že plnější vektory, odpovídající poruchám řazeným na začátek seznamu poruch, detekují také poruchy, odpovídající řídkým vektorům a ke zlepšení nedojde.

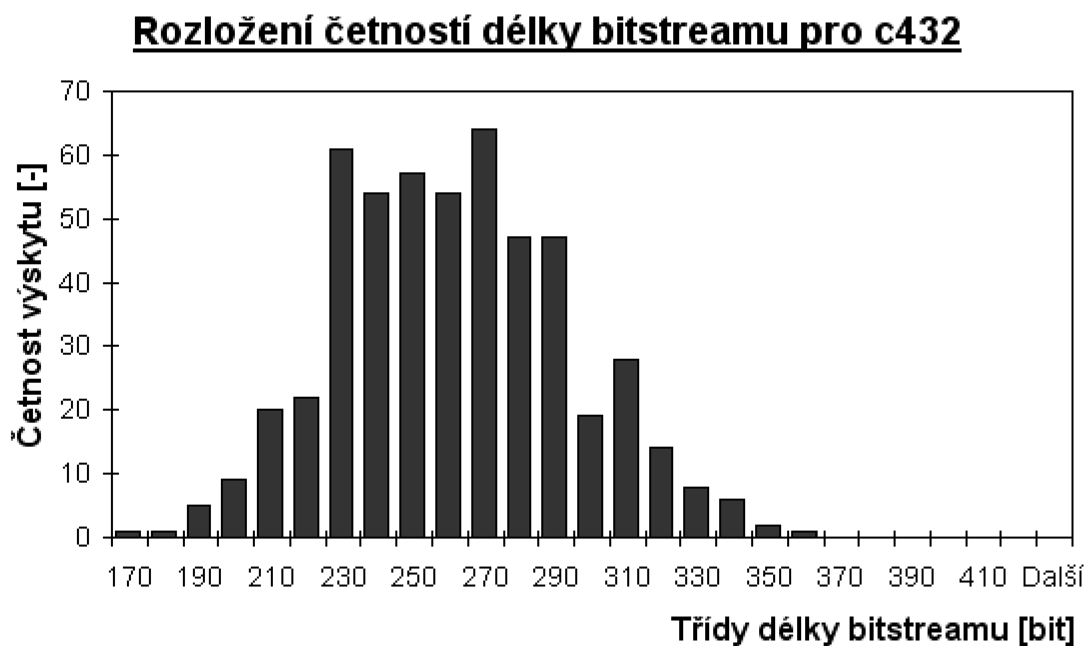
Benchmark			c432	c499	c880	s1196	c1355
COMPAS		[bit]	195	224	412	717	1040
Náš nástroj - Základní algoritmus	Max	[bit]	413	239	1101	1856	388
	Min	[bit]	171	165	539	1376	280
	Průměr	[bit]	278,26	201,69	803,35	1654,50	334,73
	Počet vektorů	[-]	79	68	117	226	103
	Velikost testu(A-m)	[bit]	1872	2214	3180	4544	3444
	Nekomprimováno	[bit]	2853	2786	7048	7221	4239
	Komprese(nas test)	[%]	90	93	89	77	92
	Komprese(vs. A-m)	[%]	85	91	75	64	90
	Čas	[s]	12,21	7,06	40,39	273,36	60,66
Náš nástroj - Základní algoritmus+DC	Max	[bit]	388	230	1188	1554	394
	Min	[bit]	170	162	474	1221	269
	Průměr	[bit]	259,81	195,56	820,70	1391,56	323,14
	Počet vektorů	[-]	78	68	123	239	103
	Nekomprimováno	[bit]	2806	2785	7358	7654	4236
	DC testováno	[bit]	290	209	894	1758	345
	DC nastaveno	[bit]	39	13	108	407	21
	DC test/nastav	[%]	16	7	14	30	6
	Komprese(nas test)	[%]	91	93	89	82	92
	Komprese(vs. A-m)	[%]	86	91	74	69	91
	Čas	[s]	11,71	7,42	52,87	260,32	57,66
Náš nástroj - Základní algoritmus+DC+MaxMin	Max	[bit]	356	237	1034	1530	384
	Min	[bit]	166	163	422	1168	274
	Průměr	[bit]	257,19	195,95	748,81	1357,15	324,45
	Počet vektorů	[-]	78	68	99	230	104
	Nekomprimováno	[bit]	2812	2787	5934	7359	4266
	DC testováno	[bit]	285	210	808	1704	346
	DC nastaveno	[bit]	29	13	92	387	21
	DC test/nastav	[%]	12	7	13	29	6
	Komprese(nas test)	[%]	91	93	87	82	92
	Komprese(vs. A-m)	[%]	86	91	76	70	91
	Čas	[s]	15,47	11,13	51,49	257,33	76,80

Tabulka č.5 *: Výsledky testování základního algoritmu + řazení Max-min+DC*

Z tabulky č. 5 vyplývá, že přidáním řazení seznamu poruch sice k jistému zlepšení výsledků došlo, ale opět se nejedná o zásadnímu změny. Nejvíce se toto řazení projevilo u

benchmarku c880, ve kterém díky němu klesl počet vektorů testu a to se následně projevilo na zvýšení komprese vzhledem k testu vygenerovanému Atalantou. Dále si ovšem všimněme, že komprese vzhledem k velikosti testu generovaného naším nástrojem naopak poklesla. Z předchozího popisu lze tedy konstatovat, že provádíme překrytí menšího počtu testovacích vektorů, což vede na kratší bitstream, ale vzhledem k tomu, že kratší test musí obsahovat plnější vektory, nejsme schopni realizovat tak kvalitní překrytí jako v předchozím případě. Do budoucna bude tedy zřejmě nutné zvolit vhodný kompromis mezi délkou testu a kvalitou překrytí.

Na obrázku č. 28 je zobrazen opět histogram ukazující rozložení četností délky bitstreamu. Porovnáme-li ho s předchozími, je zde vidět "směrování" výsledné délky bitstreamu k jeho průměrné délce. Četnosti výsledků na levé (minima) i na pravé (maxima) straně se snížily a naopak došlo k nárůstu četností ve střední části grafu.



Obrázek č.28: *Histogram pro základní algoritmus + řazení Max-min + DC*

9.3.2. Řazení poruch "min-max"

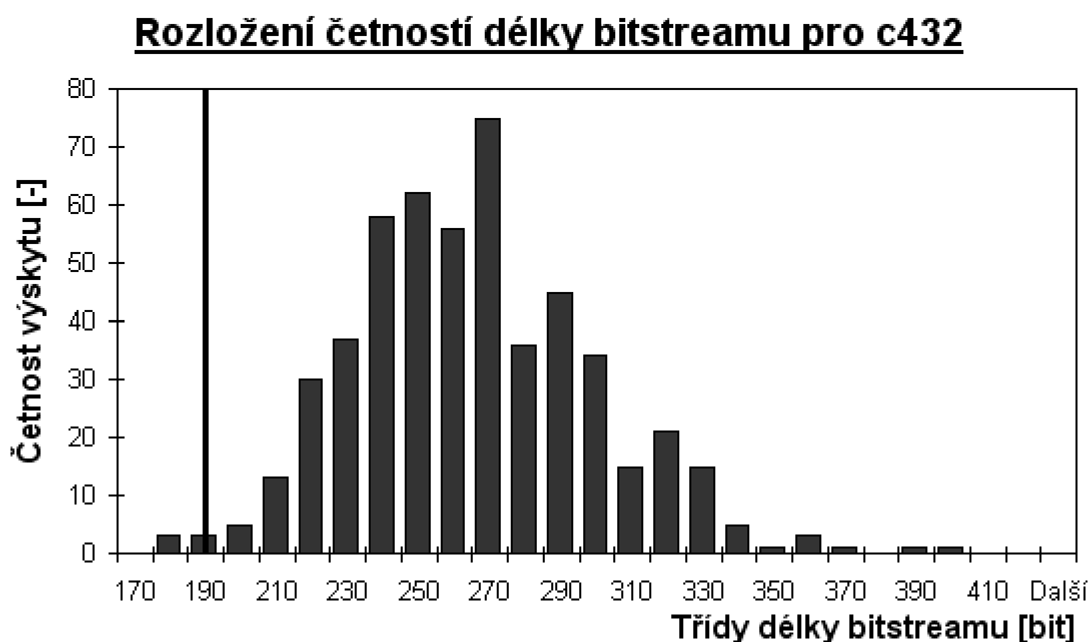
Podobně jako v předchozí kapitole se zde snažíme o řazení seznamu poruch, ale tentokrát v obráceném pořadí a zajímá nás, jestli se tato změna nějakým způsobem projeví na výsledné kompresi. Opět zde vycházíme z rozboru situace a modifikace algoritmu popsané v kapitole 7.5.1. Naměřené výsledky jsou zobrazeny v tabulce č.6. Z této tabulky je patrné, že změna řazení "max-min" na "min-max" se ve většině případů nijak významně neprojevila. Dále si všimněme, že zásadnější změny nastaly pouze u benchmarků, kde jsme již dříve dosahovali

horších výsledků tzn. c880 a s1196. U těchto benchmarků došlo k prodloužení délky testu a taktéž k většímu zhoršení průměrné délky bitstreamu. Z měření by se tedy dalo usuzovat, že benchmarky, pro které jsme již dříve dosahovali dobrých výsledků, obsahují SAT instance s podobným zastoupením proměnných primárních vstupů a tudíž se zde řazení nijak zásadně neprojeví. Na druhou stranu je zde vidět, že přidáním jakékoli formy řazení dojde k určitému zlepšení výsledků vzhledem k základnímu algoritmu. U benchmarků c880 a s1196 se dá naopak předpokládat, že pracujeme s mnohem různorodější skupinou vektorů (vzhledem k zastoupení DC) a tomu také odpovídají výsledky, které by potvrzovaly naši původní úvahu a to, že je lepší nejprve zpracovat plnější vektory, čímž vznikne úplně určený vektor, na který dále navazujeme řídké vektory, díky čemuž se zvětší pravděpodobnost, že bude nalezeno vhodné překrytí.

Benchmark			c432	c499	c880	s1196	c1355
COMPAS		[bit]	195	224	412	717	1040
Náš nástroj - Základní algoritmus	Max	[bit]	413	239	1101	1856	388
	Min	[bit]	171	165	539	1376	280
	Průměr	[bit]	278,26	201,69	803,35	1654,50	334,73
	Počet vektorů	[-]	79	68	117	226	103
	Velikost testu(A-m)	[bit]	1872	2214	3180	4544	3444
	Nekomprimováno	[bit]	2853	2786	7048	7221	4239
	Komprese(nas test)	[%]	90	93	89	77	92
	Komprese(vs. A-m)	[%]	85	91	75	64	90
	Čas	[s]	12,21	7,06	40,39	273,36	60,66
Náš nástroj - Základní algoritmus+DC+MaxMin	Max	[bit]	356	237	1034	1530	384
	Min	[bit]	166	163	422	1168	274
	Průměr	[bit]	257,19	195,95	748,81	1357,15	324,45
	Počet vektorů	[-]	78	68	99	230	104
	Nekomprimováno	[bit]	2812	2787	5934	7359	4266
	DC testováno	[bit]	285	210	808	1704	346
	DC nastaveno	[bit]	29	13	92	387	21
	DC test/nastav	[%]	12	7	13	29	6
	Komprese(nas test)	[%]	91	93	87	82	92
	Komprese(vs. A-m)	[%]	86	91	76	70	91
	Čas	[s]	15,47	11,13	51,49	257,33	76,80
Náš nástroj - Základní algoritmus+DC+MinMax	Max	[bit]	395	232	1163	1561	395
	Min	[bit]	171	152	561	1211	270
	Průměr	[bit]	261,11	195,64	866,86	1389,30	324,06
	Počet vektorů	[-]	78	68	137	255	103
	Nekomprimováno	[bit]	2804	2774	8228	8148	4227
	DC testováno	[bit]	253	167	929	1724	304
	DC nastaveno	[bit]	37	13	126	397	21
	DC test/nastav	[%]	17	8	16	30	8
	Komprese(nas test)	[%]	91	93	89	83	92
	Komprese(vs. A-m)	[%]	86	91	73	69	91
	Čas	[s]	11,26	6,50	52,90	220,33	57,30

Tabulka č.6 *: Výsledky testování základního algoritmu + řazení Min-max+DC*

Obrázek č. 29 nám opět ukazuje histogram pro benchmark c432. Porovnáme-li ho s histogramem na obrázku č.28, tzn. s výsledky pro opačné řazení, dojdeme k závěru, že ačkoli se nejedná o žádné dramatické změny, je zde patrný posun četností doprava a s tím spojené zhoršení výsledků, které je patrné také ze zhoršení průměrné délky bitstreamu. Dále si všimněme, že pokud jsme u řazení "max-min" hovořili o jakémsi "směrování" četností délky bitstreamu ke střední hodnotě, lze v tomto případě hovořit o "rozptylu" a nárůstu četností v pravé části grafu, což indikuje jisté zhoršení.



Obrázek č.29: *Histogram pro základní algoritmus + řazení Min-max + DC*

9.4. Základní algoritmus rozšířen o řazení poruch a "Pre-simulaci"

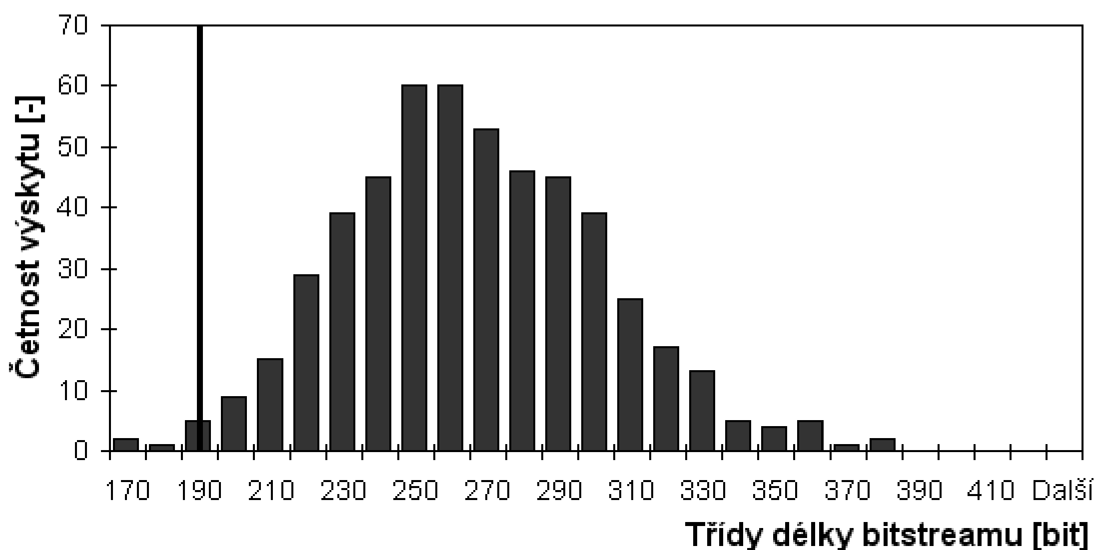
V předchozích kapitolách jsme se snažili kompresi zlepšit přidáním DC, popř. řazením seznamu poruch. Nyní si ukážeme jaký má na kompresi vliv tzv. "pre-simulace" popsaná v kapitole 7.5.3. Pomocí této metody bychom měli dosáhnout zvýšení počtu DC v testovacích vektorech, které se snažíme překrýt a zároveň by mělo dojít k urychlení aplikace. Nesmíme zde ovšem zapomínat na skutečnost, že zvýšením zastoupení DC stavů pravděpodobně dojde také k nárůstu počtu vektorů testu, který může vést k tomu, že se zlepšení neprojeví. Při měření jsme kromě "pre-simulace" použili také řazení seznamu poruch "Max-min", které by mělo zajistit, že budeme v první fázi pracovat s plnějšími vektory, které bude možné pomocí pre-simulace "vysunovat" a tak pokrýt další poruchy, aniž by bylo nutné řešit větší množství SAT instancí. Výsledky měření ukazuje tabulka č.7. Všimněme si zde, že evidentně došlo ke zvýšení zastoupení DC ve vektorech, jak jsme předpokládali, a to i vzhledem k předchozí

metodě přidávání DC. Tuto skutečnost indikuje poměrně velký nárůst testovacích vektorů. Dále je ovšem také zřejmé, že ačkoli došlo ke zvýšení zastoupení DC, nedošlo téměř k žádnému zlepšení ve výsledné kompresi. Ve většině případů zde došlo pouze ke zrychlení aplikace, což bylo zřejmě způsobeno tím, že díky pre-simulaci nebylo nutné řešit takové množství SAT instancí.

Benchmark			c432	c499	c880	s1196	c1355
COMPAS		[bit]	195	224	412	717	1040
Náš nástroj - Základní algoritmus	Max	[bit]	413	239	1101	1856	388
	Min	[bit]	171	165	539	1376	280
	Průměr	[bit]	278,26	201,69	803,35	1654,50	334,73
	Počet vektorů	[-]	79	68	117	226	103
	Velikost testu(A-m)	[bit]	1872	2214	3180	4544	3444
	Nekomprimováno	[bit]	2853	2786	7048	7221	4239
	Komprese(nas test)	[%]	90	93	89	77	92
	Komprese(vs. A-m)	[%]	85	91	75	64	90
	Čas	[s]	12,21	7,06	40,39	273,36	60,66
Náš nástroj - Základní algoritmus+DC	Max	[bit]	388	230	1188	1554	394
	Min	[bit]	170	162	474	1221	269
	Průměr	[bit]	259,81	195,56	820,70	1391,56	323,14
	Počet vektorů	[-]	78	68	123	239	103
	Nekomprimováno	[bit]	2806	2785	7358	7654	4236
	DC testováno	[bit]	290	209	894	1758	345
	DC nastaveno	[bit]	39	13	108	407	21
	DC test/nastav	[%]	16	7	14	30	6
	Komprese(nas test)	[%]	91	93	89	82	92
	Komprese(vs. A-m)	[%]	86	91	74	69	91
	Čas	[s]	11,71	7,42	52,87	260,32	57,66
Náš nástroj - Základní algoritmus + presimulace	Max	[bit]	380	255	1013	1806	393
	Min	[bit]	165	178	501	1424	291
	Průměr	[bit]	261,85	213,45	772,00	1626,38	342,24
	Počet vektorů	[-]	85	78	146	252	110
	Nekomprimováno	[bit]	3077	3191	8775	8076	4527
	Komprese(nas test)	[%]	91	93	91	80	92
	Komprese(vs. A-m)	[%]	86	90	76	64	90
	Čas	[s]	8,49	8,11	37,68	241,82	74,77

***Tabulka č.7** : Výsledky testování základního algoritmu + Max-min+ presimulace*

Tak jako v předchozích kapitolách, je i zde na obrázku č. 30 ukázán histogram pro benchmark c432. Překvapivě zde podobně jako v předchozím případě došlo k “rozptylu“ četností délek bitstreamu po celé ose. Došlo tak ke zvýšení četností minimálních i maximálních hodnot a snížení hodnot blížícím se průměru. Tento rozptyl měl za následek také výsledné zvýšení průměrné délky bitstreamu.



Obrázek č.30: *Histogram pro základní algoritmus + Max-min + presimulace*

9.5. Souhrnné výsledky

V předchozích kapitolách jsme se snažili demonstrovat vliv několika modifikací algoritmu na výslednou kompresi popř. dobu řešení. Abychom mohli objektivně posoudit případný přínos určité modifikace, prováděli jsme měření pro všechny možné výběry první poruchy. Tímto způsobem jsme částečně mohli eliminovat vliv "počátečního" stavu na výsledky a omezit tak případné zkreslení vzniklé vhodným nebo naopak špatným výběrem počátečního stavu. Jak již ale bylo také řečeno, bylo toto testování poměrně časově náročné a vzhledem k tomu nebylo možné takto otestovat větší množství úloh. Z naměřených hodnot je ovšem zřejmé, že výběr počátečního stavu má na dosažené výsledky velký vliv (viz. předchozí histogramy). Naneštěstí nebylo zatím s časových důvodů možné implementovat heuristiku provádějící výběr vhodného počátečního stavu resp. poruchy, od které začneme. K navržení takovéto heuristiky bude zapotřebí provést velké množství měření a na základě jejich analýzy se snad podaří implementovat heuristiku schopnou vybrat co možná nejlepší počáteční stav.

Následující tabulka č.8 prezentuje výsledky pro jednotlivé modifikace základního algoritmu a jejich srovnání s nástrojem COMPAS (z hlediska komprese). Pro všechna měření byla jako výchozí stav použita porucha na začátku seznamu poruch, resp. vektor odpovídající řešení její SAT instance. Komprese je vyhodnocována vůči délce testu generovanému v nástroji Atalanta-M.

Benchmark		c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c6288	c7552	s1196	s1238	s5378	s9234
COMPAS	[bit]	195	224	412	1040	979	6091	726	1217	82	7115	717	783	1989	10598
Komprese (COMPAS)	[%]	89,58	89,88	87,04	69,80	74,43	75,79	90,12	94,00	92,23	84,01	84,44	83,35	96,36	88,86
Základní algoritmus (ZA)	[bit]	364	205	643	332	547	2102	3211	906	93	6100	1756	1663	2505	11292
Komprese (ZA)	[%]	80,56	90,74	79,78	90,36	85,71	91,65	56,31	95,54	91,19	86,29	61,89	64,65	95,41	88,13
ZA + DC	[bit]	269	196	763	344	547	2064	2043	1048	186	4779	1406	1423	2505	11212
Komprese (ZA+DC)	[%]	85,63	91,15	76,01	90,01	85,71	91,80	72,20	94,84	82,39	89,26	69,49	69,75	95,41	88,21
ZA + DC + Max-min	[bit]	202	187	646	317	603	2217	1947	853	113	5014	1373	1436	2397	11317
Komprese (ZA+DC+Max-min)	[%]	89,21	91,55	79,69	90,80	84,25	91,19	73,51	95,80	89,30	88,73	70,20	69,47	95,61	88,10
ZA + DC + Min-max	[bit]	222	210	1045	345	563	2250	2150	964	147	4921	1414	1456	2640	11569
Komprese (ZA+DC+Min-max)	[%]	88,14	90,51	67,14	89,98	85,29	91,06	70,75	95,25	86,08	88,94	69,31	69,05	95,16	87,83
ZA + Max-min + Presim	[bit]	215	201	685	360	609	2405	2690	840	133	5576	1486	1845	2559	10858
Komprese (ZA+Max-min+Presim)	[%]	88,51	90,92	78,46	89,55	84,09	90,44	63,40	95,86	87,41	87,47	67,75	60,78	95,31	88,58

Tabulka č.8 *: Výsledky souhrnného testování (počáteční stav daný poruchou č. 0)*

Z tabulky č.8 vyplývá, že pro většinu testovaných úloh byl námi implementovaný nástroj schopen dosáhnout podobné komprese jako nástroj COMPAS. Případy, ve kterých jsme dosáhli lepší komprese jsou odlišeny tmavším pozadím. Dále si ovšem také můžeme všimnout, že rozdíly v kompresi mezi jednotlivými modifikacemi nejsou nijak zásadní a bude tedy zřejmě nutné provést množství dalších měření, abychom nalezli heuristiky schopné dosáhnout významnějšího zlepšení. Nesmíme zde ovšem také zapomínat, že výsledky mohou být pozitivně i negativně ovlivněny výběrem počátečního stavu daného poruchou na vrcholu seznamu (hlavně provádíme-li řazení seznamu poruch).

10. Závěr

Naprogramoval jsem nástroj realizující kompresi pomocí nového algoritmu, který jsme vymysleli. Dále jsem provedl sérii testů, která nám měla naznačit, kudy se ubírat při hledání dalších vylepšení a ověřit funkčnost algoritmu v praxi. Ověření správné funkce programu bylo provedeno pomocí nástroje Atalanta-M [3][12], kterým jsem provedl simulaci vygenerovaného test. Zjistil jsme, že pomocí tohoto nového algoritmu lze dosáhnout poměrně slušné komprese testovacích vzorků, která se ve většině případů vyrovná referenčnímu nástroji COMPAS [1]. Měřením jsem prokázal, že výběr vhodného počátečního stavu může do značné míry ovlivnit výslednou kompresi a bude tedy do budoucna nutné implementovat heuristiku schopnou vhodně zvolit počáteční poruchu. Dále z měření vyplývá, že řazení seznamu poruch (pořadí jejich zpracování) může přispět ke zlepšení komprese a v neposlední řadě může kompresi ovlivnit také zastoupení DC v testovacích vektorech (ne ovšem tak významně jako u nástroje COMPAS).

Tento náš nový algoritmus má bezesporu velký potenciál a umožňuje množství modifikací, které je možné realizovat. V první řadě by určitě bylo výhodné rozšířit nástroj o BDD SAT řešič popsaný v kapitole 7.5.4, pomocí kterého by bylo mnohem snazší vybírat vhodné vektory pro překrytí. Ačkoli v základní verzi je tento nástroj příliš pomalý k efektivní práci, dá se předpokládat, že s použitím vhodného preprocesingu by šlo řešené instance značně redukovat a zpracování by posléze bylo možné (alespoň pro část SAT instancí). Dále by bylo z časových důvodů velmi vhodné vytvořit jednu aplikaci s jednotnou reprezentací sdružující funkce potřebné k řešení a dále ukládat některé mezivýsledky. Je zde třeba zvolit vhodný kompromis mezi spotřebou paměti a dobou řešení problému. Další vylepšení budou vycházet z analýzy SAT instancí, které zpracováváme, přičemž by mohlo být zajímavé zaměřit se na "problematické" instance a jejich případné odhalení.

11. Literatura

[1] Novák, O. – Zahrádka, J.: COMPAS – Compressed Test Pattern Sequencer for Scan Based Circuits, Proc. Of EDCC 2005.

[2] Červák, J.: Automatický generátor testovacích vektorů (ATPG) založený na testu splnitelnosti booleovské formule. Diplomová práce, FEL ČVUT v Praze, 2008.

[3] Myslík, R.: Úprava ATPG nástroje Atalanta. Diplomová práce, FEL ČVUT v Praze, 2008.

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

[5] Balcárek, J.: Řešení problému splnitelnosti booleovské formule (SAT) pomocí binárních rozhodovacích diagramů (BDD). Bakalářská práce, FEL ČVUT v Praze, 2007.

[6] Strnadel, J.: Analýza a zlepšení testovatelnosti číslicového obvodu na úrovni meziregistrových přenosů. Disertační práce, FIT VUT v Brně, 2004.

[7] Dorsch, R. and Wunderlich, H-J.:Reusing Scan Chains for Test Pattern Decompression. Proc. of the European Test Workshop, 2001, pp. 24-32

[8] Densmore, D.: Built-In-Self Test (BIST) Implementations An overview of design tradeoffs. University of Michigan, EECS 579 – Digital Systems Testing, Professor John P. Hayes 12/7/01.

[9] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. J. ACM, 7(3):201 215, 1960.

[10] T. Larrabee. Test Pattern Generation Using Boolean Satisfiability. Test Pattern Generation Using Boolean Satisfiability. IEEE Transactions on Computer-Aided Design, pages 4-15, Jan, 1992.

[11] The minisat page http://minisat.se/.

[12] Fišer, P.: Atalanta-m 2.0 Manual, 2005. http://service.felk.cvut.cz/vlsi/prj/Atalanta-M/man.html.

[13] Lee, H. K.; Ha, D. S.: HOPE: an efficient parallel fault simulator for synchronous sequential circuits. In DAC ’92: Proceedings of the 29th ACM/IEEE conference on Design automation, Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, ISBN 0-89791-516-X, s. 336–340.

[14] Jeníček, J.: Komprese testovacích dat založená na překrývání vzorků. Autoreferát disertační práce, Fakulta mechatroniky, informatiky a mezioborových studií Technická universita v Liberci, 2009

12. Seznam zkratek

ATE Automatic Test Equipment
Externí testovací vybavení

ATPG Automatic Test Pattern Generator
Automatický generátor testovacích vektorů

BDD Binary Decision Diagram
Binární rozhodovací diagram

BIST Built-in Self Test
Vestavěné diagnostické prostředky

CNF Conjunctive Normal Form
Konjunktivní normální forma

COMPAS COMpressed Pattern Sequencer

CUT Circuit Under Test
Testovaný obvod

DC Don‘t Care
Nespecifikovaná hodnota

DLL Davis Logemann Loveland
Vylepšený algoritmus Davis Putnam

DP Davis Putnam
Davis Putnam algoritmus

ETC Embedded Test Core
Vestavěné testovací jádro

LSB Least Significant Bit
Nejméně významný bit – nejmenší váha

LSFR Linear Feedback Shift Register
Lineární zpětnovazební registr

MSB Most Significant Bit
Nejvýznamnější bit – s nejvyšší váhou

RAS Random Access Scan
Metoda umožňující "paralelní" přístup k paměťovým buňkám

RESPIN REusing Scan chains for test Pattern decompressIoN

ROBDD Reduced Ordered Binary Decision Diagrams
Redukovaný uspořádaný binární rozhodovací diagram

SA Signature Analyser
Příznakový analyzátor

SAS Seriál Access Scan
Skenovací řetězec se sériovým přístupem

SAT SATisfiability
Splnitelnost

SC Scan Chain
Skenovací/testovací řetězec

SoC System-On-a-Chip
Systém na čipu

TAM Test Access Mechanism
Testovací mechanizmus

TPG Test Pattern Generator
Generátor testovacích vektorů

TRA Test Response Analyser
Analyzátor odezev na testovací vektory

TTL Transistor-Transistor Logic
Tranzistorově-tranzistorová logika

13. Přílohy

13.1. Uživatelská příručka

Práce s námi vytvořeným nástrojem je velice jednoduchá. Stačí zadat jméno spustitelného souboru naší aplikace (newAtpg.exe) a příslušné parametry. Ke správné funkci programu je zapotřebí mít v pracovním adresáři knihovnu pokusat.dll. Vstupem aplikace je .bench soubor obsahující popis obvodu v ISCAS89 netlist formátu. Výstupem aplikace je soubor obsahující testovací bitstream. Program lze spouštět s následujícími parametry:

-n <FILE>	za parametrem následuje vstupní soubor .bench
-t <FILE>	za parametrem následuje výstupní soubor (bitstream)
-W <X>	za parametrem následuje číslo poruchy, která se bude zpracovávat jako první.
-DC	povoluje přidávání DC do vektorů (implicitně vypnuto)
-PS	povoluje provádění pre-simulace (implicitně vypnuto)
-S <X>	nastavuje řazení seznamu poruch : 0 – neřadím (implicitně) 1 – řazení “max-min” 2 – řazení “min-max”

Př. spouštění aplikace provádějící pouze základní algoritmus, který započne generování testu od poruchy číslo 0:

newAtpg –n c432.bench –t c432.out –W 0

V případě, že chceme základní algoritmus rozšířit o další vylepšení, přidáme další parametry dle předchozího popisu.

13.2. Formát souboru .bench

Převzato z dokumentu [12]. BENCH format.

Vstupním formátem aplikace je ISCAS89 netlist formát. Řádky začínající znakem # jsou komentáře. Pořadí hradel v obvodu není pevně stanoveno. Jména hradel mohou být tvořena alfanumerickými znaky (0-9, A-Z, a-z, _, [,nebo]).

Definovaná hradla

syntaxe	typ hradla
INPUT	primární vstupy
OUTPUT	primární výstupy
AND	hradlo AND
NAND	hradlo NAND
OR	hradlo OR
NOR	hradlo NOR
XOR	2-vstupové hradlo XOR
BUFF or BUF	buffer
NOT	invertor

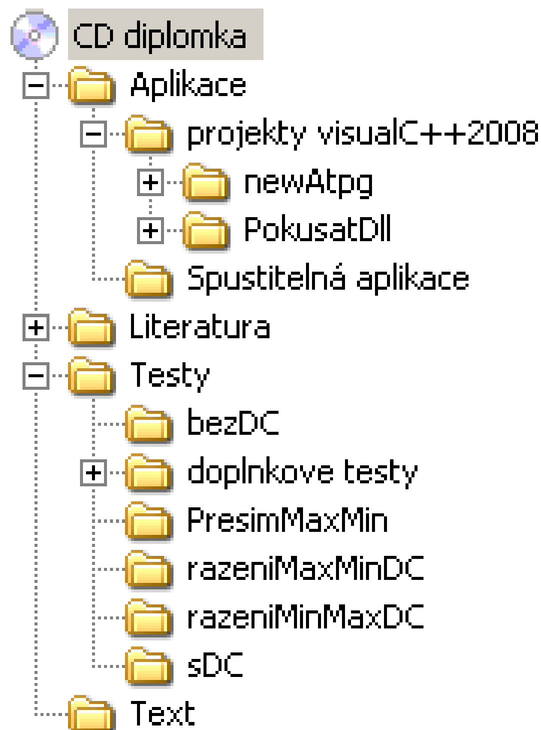
* hradla mohou být psána též malými písmeny

PŘÍKLAD: ISCAS89 NETLIST FORMAT (c17.bench)

```
# 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)
```

13.3. Obsah CD

Obrázek č.31 znázorňuje adresářovou strukturu přiloženého CD. V kořenovém adresáři se nachází složka aplikace obsahující spustitelnou aplikaci a projekty “newAtpg“ (naše aplikace) a “pokusatDLL“ (upravená Atalanta-M – výstupem je knihovna, kterou používáme). Dále je zde složka literatura obsahující většinu literatury, ze které bylo při práci čerpáno. Složka testy obsahuje výsledky všech prováděných měření a dokumenty aplikace Excel, kde jsou tyto výsledky zpracovány do tabulek (použitých v diplomové práci). Poslední složkou je text, která obsahuje diplomovou práci ve formátu .doc a .pdf.



Obrázek č.31: *Adresářová struktura přiloženého CD*