České vysoké učení technické v Praze
Fakulta elektrotechnická

Diplomová práce

Automatický generátor testovacích vektorů (ATPG) založený na testu splnitelnosti booleovské formule

Jiří Červák

Vedoucí práce: Ing. Petr Fišer

Studijní program: Elektrotechnika a informatika strukturovaný magisterský

Obor: Informatika a výpočetní technika

leden 2008

Poděkování

Děkuji vedoucímu práce ing. Petru Fišerovi za cenné rady a připomínky při realizaci diplomové práce. Dále děkuji svým rodičům za podporu nejen při psaní této práce, ale i během celého studia.

Prohlášení

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

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

V Přerově nad Labem dne 16.1. 2008 ....

Abstract

In this thesis ATPG aplication based on solving SAT problem by open-source solver. SAT solver Minisat 2.0 is used, ways of generating don´t cares in test patterns are explored. Files in format ISCAS 89 are input of aplication, resulted test patterns are compared with ATPG tool Atalanta. Parameters of SAT instances solved during ATPG are analyzed. ATPG tool can by used as generator of SAT instances in DIMACS format.

Abstrakt

Tato práce se zabývá implementací ATPG nástroje využívající řešení SAT problému open-source řešičem. Jako řešič je použit Minisat 2.0, dále jsou zkoumány možnosti generování neurčitých hodnot v testovacích vektorech. Vstupem aplikace jsou soubory ve formátu ISCAS 89, výsledné sady vektorů jsou porovnány s ATPG nástrojem Atalanta. Dále jsou analyzovány parametry instancí SAT řešených při generování vektorů. Nástroj může být použit jako generátor instancí SAT problémů ve formátu DIMACS.

Obsah

Seznam obrázků xi

Seznam tabulek xiii

1 Úvod 1

1.1 Základní pojmy 1

1.1.1 Poruchy v číslicových obvodech 1

1.1.1.1 Model poruch 1

1.1.1.2 Seznam poruch 2

1.1.1.3 Sada testovacích vektorů 2

1.1.1.4 Druhy poruch 2

1.1.1.5 Ekvivalence a dominance poruch 2

1.1.1.6 Checkpoint teorém 3

1.2 Výroková logika 3

1.2.1 Výroková formule 3

1.2.2 Booleova logika 4

1.2.3 Konjunktivní normální forma 4

1.3 Problém SAT 5

2 Popis problému, specifikace cíle 6

2.1 Metody generování testovacích vektorů 6

2.2 Strukturální metody 6

2.3 Algebraické metody v ATPG 7

2.3.1 Obvod jako funkce vnitřních proměnných 7

2.3.2 Transformace kombinačního obvodu na formuli v CNF formě 7

2.4 SAT řešiče 10

2.4.1 Kompletní SAT algoritmy 10

2.4.2 Nekompletní SAT algoritmy 11

2.4.3 Preprocesory 11

2.4.4 open-source SAT řešiče 12

2.5 Atalanta-M 12

2.6 Cíl práce 12

3 Analýza a návrh řešení **13**

3.1 Výběr SAT-řešiče. 13

3.1.1 Open-source charakter 13

3.1.2 Výkonnost 14

3.1.3 Dokumentace 14

3.1.4 Portabilita 14

3.1.5 Kompletní algoritmus 14

3.1.6 Schopnost generovat neurčené hodnoty v řešení 14

3.1.7 Minisat 14

3.2 Převod obvodu do cnf formy 15

3.3 Neurčené stavy v řešení SAT 18

3.3.1 Modifikace SAT řešiče 18

3.3.2 Modifikace řešení SAT řešiče 18

3.3.2.1 Modifikace řešení pomocí formule 18

3.3.2.2 Modifikace řešení pomocí strukturáního algoritmu 21

3.4 Dominance 23

3.5 Kompakce 23

3.6 Redundantní proměnné 23

4 Realizace **25**

4.1 Volba programovacích prostředků 25

4.2 Návrh datových struktur 25

4.3 Podpora dalších SAT řešičů 25

5 Testování **27**

5.1 Celkové výsledky 27

5.2 Celkové výsledky 27

5.3 Srovnání časů a pokrytí 29

5.4 Aktivační proměnné 29

5.5 Počet DC hodnot 29

5.6 Klauzule a proměnné 30

6 Závěr **31**

A Seznam použitých zkratek **35**

B Uživatelská / instalační příručka **37**

B.1 Parametry souborů 37

B.2 Parametry TPG 37

B.3 Formáty souborů 37

C Obsah přiloženého CD **43**

Seznam obrázků

2.1 Obvod s poruchou 6
2.2 ukázkový obvod 7
2.3 hradlo and 8
2.4 hradlo nand 8
2.5 hradlo or 8
2.6 hradlo nor 9
2.7 hradlo xor 9
2.8 buffer 9
2.9 Invertor 10
2.10 Bezporuchový obvod 10
2.11 Obvod s poruchou 11
2.12 Booleovská diference 11

3.1 Třídy Solver a SimpSolver řešiče minisat 15
3.2 Stav obvodu s poruchou 16

4.1 Hierarchie třídy Gate a jejich potomků 26

Seznam tabulek

5.1 Přehled měření ATPG nástroje na ISCAS benchmarcích 28
5.2 Srovnání SAT ATPG a Atalanty 28
5.3 Srovnání časů při použití aktivačních proměnných 29
5.4 Porovnání počtu DC hodnot 30
5.5 Počty klauzulí a proměnných v generovaných booleovských formulích 30

Kapitola 1

Úvod

S výrobou stále složitějších číslicových obvodů je stále obtížnější nalézt rychlý a efektivní způsob testování vyráběných součástek. Doba testu a jeho schopnost odhalit nefunkční součástku významným způsobem ovlivňují náklady na výrobu. Přitom požadavky na maximální rychlost a efektivnost testu se navzájem vylučují. Při výrobě pak platí obecné pravidlo, že čím dříve je porucha nalezena, tim levnější je její odstranění. Úkolem automatického generátoru testovacích vektorů (ATPG) je v co nejkratším čase odhalit co nevíce poruch v číslicovém obvodu. Historicky nejdelší vývoj mají za sebou generátory založené na strukturálních algoritmech, později se ovšem objevily první efektivní generátory založené na testování splnitelnosti booleovské formule. Problém splnitelnosti booleovské formule (SAT) je tak příkladem obtížného matematického problému, jehož vyřešení přináší praktické využití. Programy, které řeší problém SAT, takzvané SAT řešiče, prodělávají neustálý vývoj a nové heuristiky a vylepšení přinášejí stále větší efektivnost řešení. Kromě komerčních SAT řešičů existuje i řada open-source programů, které dosahují zajímavých výsledků. Mým úkolem je vytvořit ATPG nástroj založený na řešení problému SAT, který bude používat některý z open-source řešičů. Při stejné efektivnosti je třeba získat co nejmenší počet testovacích vektorů. Snížení jejich počtu je dosaženo slučováním, kdy se nejlépe slučují vektory s velkým počtem neurčených hodnot (DC hodnot). Proto bude počet DC hodnot ve výsledných vektorech důležitým parametrem ATPG nástroje, který se pokusím ve své práci implementovat.

1.1 Základní pojmy

1.1.1 Poruchy v číslicových obvodech

1.1.1.1 Model poruch

Poruchy v číslicových obvodech mohou mít různou fyzikální příčinu. Pro ověření správné funkce obvodu není důležitá příčina poruchy, ale její projev. Zajímá nás, jak se porucha při daném vstupu projeví na výstupních hodnotách. Bez znalosti topologie obvodu by kompletní test funkčnosti obvodu s n vstupy zahrnoval otestování 2^n možných hodnot vstupů, což je pro větší počet vstupů nerealizovatelné. Tomuto naivnímu přístupu se lze vyhnout použitím vhodného modelu poruch.

Běžně používaným je model Single stuck-at poruch. Každá porucha v tomto modelu má tyto vlastnosti:

- Porucha je pouze na jediném vodiči v celém obvodě

- Porušený vodič má trvalou hodnotu 0 nebo 1
- Porucha může být na vstupu nebo výstupu hradla

Každý vodič v tomto modelu může mít tedy dva druhy poruch: Trvalá jednička, zkráceně **Sa1** a analogicky trvalá nula, **Sa0**. Vodičem rozumíme spojení mezi 2 hradly, nebo hradlem a větvením.

1.1.1.2 Seznam poruch

Seznam poruch obsahuje všechny poruchy, které v daném modelu mohou nastat. Pro Single stuck-at model lze celkový počet poruch vyčíslit:

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

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

1.1.1.3 Sada testovacích vektorů

Testovací vektor pro danou poruchu definuje takové ohodnocení vstupů číslicového obvodu, které zaručí, že přítomnost poruchy v testovaném obvodu se projeví změnou alespoň jednoho výstupu oproti správně fungujícímu obvodu. Seznam vektorů, které detekují poruchy v seznamu poruch, nazývame sadou testovacích vektorů.

1.1.1.4 Druhy poruch

ATPG nástroj obvykle nenalezne pro každou poruchu odpovídající testovací vektor, protože některé poruchy jsou netestovatelné. Z tohoto pohledu lze vektory klasifikovat:

- Testovatelná porucha: ATPG nástroj nalezl vektor testující tuto poruchu.
- Netestovatelná porucha: ATPG nástroj nenalezl vektor testující tuto. poruchu
- Redundantní porucha: Porucha, na kterou neexistuje test.

Redundantní poruchy jsou podmnožinou netestovatelných poruch. Snahou je docílit toho, aby ATPG nástroj nalezl testovací vektory pro všechny poruchy, které nejsou redundantní a dosáhl tak maximálního možného pokrytí. Pokrytí poruch je poměr počtu testovatelných poruch vůči celkovému počtu, udává se v procentech. Pokrytí dané sady vektorů lze ověřit pomocí simulátoru poruch.

1.1.1.5 Ekvivalence a dominance poruch

Seznam poruch lze redukovat pomocí ekvivalence a dominance poruch.

Definice 1.1.1 Poruchy f1 a f2 jsou ekvivalentní, právě když mají shodné sady vektorů, které je detekují.

Z této definice plyne, že pokud jsou nalezeny ekvivalentní chyby, je dostačující hledat testovací vektory jen pro jednu z nich. Lze zajít ještě dál a celý seznam poruch rozdělit na třídy ekvivalence a z každé třídy vybrat jednoho zástupce.

Definice 1.1.2 Porucha f1 dominuje poruše f2, jestliže všechny vektory detekující f2 detekují i f1.

Dominance nám umožňuje 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.

1.1.1.6 Checkpoint teorém

Primární vstupy a všechna větvení kombinačního obvodu jsou checkpointy (kontrolní body). Checkpoint teorém pak říká:

Definice 1.1.3 Sada vektorů, která detekuje všechny Sa poruchy na všech checkpointech, detekuje také všechny Sa poruchy v daném obvodě.

Teorém říká, že pro kompletní pokrytí poruch v celém obvodě 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.

1.2 Výroková logika

1.2.1 Výroková formule

Atomická formule výrokové logiky se nazývá proměnná. Výroková logika označuje obecně formální odvozovací systém, ve kterém atomické formule tvoří výrokové proměnné. [7]. Výrokové formule se definují takto:

Definice 1.2.1 Výroková formule

1. Každá atomická výroková formule je též výroková formule.
2. Jestliže A je výroková formule, je i $(\neg A)$ výroková formule.
3. Jsou-li A, B výrokové formule, je i $(A \rightarrow B)$ výroková formule.

Definice 1.2.2 Pravdivostní ohodnocení atomických proměnných je zobrazení $\nu : P \rightarrow 0, 1$, kde P je množina atomických proměnných.

Hodnoty 1 a 0 se často označují hodnotami Pravda / Nepravda nebo anglickými ekvivalenty True / False. Rozšíření pravdivostního ohodnocení ϖ na výrokové formule definujeme takto:

Definice 1.2.3 Pravdivostní ohodnocení výrokových formulí ϖ

1. $\varpi(A) = \nu(A)$ je-li A atomická formule
2. $\varpi(\neg A) = 1$ je-li $\varpi(A) = 0$
3. $\varpi(\neg A) = 0$je-li $\varpi(A) = 1$
4. $\varpi(A \rightarrow B) = 0$ jestliže $\varpi(A) = 1$ a $\varpi(B) = 0$
5. $\varpi(A \rightarrow B) = 1$ jestliže $\varpi(A) = 0$ nebo $\varpi(B) = 1$

Symboly $\neg$ a $\rightarrow$ definující logické operace (operátory) negace a implikace nazýváme logickými spojkami. Pro $\neg A$ se používá zápis $\overline{A}$. Pomocí logických operací $\neg$ a $\rightarrow$ lze definovat další známé a používané binární operace konjunkce ($\wedge, and$), disjunkce $\vee, or$, ekvivalence ($\equiv, \leftrightarrow$). Dále se zavádí exkluzive or ($\oplus, xor$). Jestliže pro konkrétní ohodnocení proměnných je ohodnocení formule rovno 1, říkáme, že je pravdivá. V opačném případě je nepravdivá. Jestliže je formule pravdivá pro všechna možná ohodnocení, nazýváme jí tautologií. Formule, která pro žádné ohodnocení není ohodnocena 1, je kontradikce, říkáme o ní, že je nesplnitelná. Splnitelná formule je formule, která není kontradikcí. Dvě formule jsou ekvivalentní, jestliže mají stejné ohodnocení pro všechna možná ohodnocení proměnných.

1.2.2 Booleova logika

Booleova logika je výroková logika nad množinou X, která používá operátorů $\wedge, \vee, \neg$ [14]. Booleovská formule pak užívá pouze operátorů booleovské logiky.

Definice 1.2.4 Booleovská formule je výroková formule, která obsahuje pouze operátory $\wedge, \vee, \neg$.

Kromě booleovské formule se zavádí pojem booleovská funkce $f : B^k \rightarrow B$, kde $B = 0, 1$. Boleovská formule je jedním z možných vyjádření booleovské funkce. Booleovská diference dvou booleovských funkcí f a g je definována jako:

Definice 1.2.5 Booleovská diference funkcí: $B_d(f, g) = f \oplus g$

1.2.3 Konjunktivní normální forma

Protože ke každé výrokové formuli existuje nekonečně mnoho ekvivalentních formulí, na které lze formuli přepsat, jsou zavedeny takzvané normální formy. Jsou to disjunktivní normální forma (DNF) a konjunktivní normální forma (CNF).

Definice 1.2.6

- Literál je atomická formule nebo její negace
- Klauzule je konjunkce literálů
- Formule, která je disjunkcí literálů, je v konjuktivní normální formě

Definice 1.2.7

- Literál je atomická formule nebo její negace
- Klauzule je disjunkce literálů
- Formule, která je konjunkcí literálů, je v konjuktivní normální formě

Významná je pro nás CNF forma, proto dále v textu bude pojem klauzule vyhrazen disjunkci literálů. Budu-li zmiňovat booleovskou funkci, bude se jednat o její CNF formu, nebude-li uvedeno jinak.

1.3 Problém SAT

Problém splnitelnosti booleovské formule se nazývá problém SAT. Je to rozhodovací problém, kdy se hledá odpověď na otázku, zda je daná booleovská formule splnitelná. Součástí tvrzení "Ano" musí být i důkaz ověřitelný v polynomiálním čase, tedy příslušné ohodnocení logických proměnných. SAT je NP-úplný problém. Třída NP problémů obsahuje takové problémy, u kterých důkaz ověření kladného řešení musí být ověřitelný v polynomiálním čase deterministickým Turingovým strojem. [7]. NP-úplné problémy jsou takové NP problémy, kdy každý NP problém je na ně převoditelný. V současnosti neexistuje algoritmus, který by řešil NP-úplné problémy v polynomiálním čase.

Kapitola 2

Popis problému, specifikace cíle

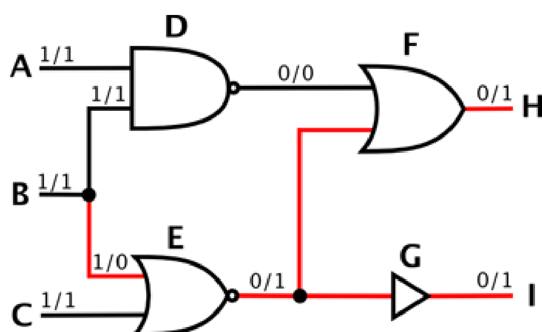
2.1 Metody generování testovacích vektorů

Pro generování testovacích vektorů se používají dva odlišné přístupy. Jedny vycházejí ze struktury hradel a reprezentaci číselného obvodu jako grafu, druhé převádějí vnitřní strukturu do algebraické formy.

2.2 Strukturální metody

Strukturální metody využívají k hledání testovacích vektorů různé algoritmy založené na prohledávání grafů. S obvodem se zachází jako s grafem, kde vrcholy jsou tvořeny hradly a hrany vodiči. Jednoduchý algoritmus bude pracovat ve třech krocích:

- Nastavení výstupu hradla před poruchou na hodnotu opačnou k druhu chyby.
- Propagace, neboli hledání citlivé cesty od poruchy na některý z primárních výstupů.
- Hledání takového nastavení primárních vstupů, které je konzistestntní s nastavením vnitřních hradel.



Obrázek 2.1: Obvod s poruchou

Na obrázku 2.1 je vidět jednoduchý kombinační obvod s poruchou SaO na vstupu hradla E. Na vstupy je přiveden vektor (1,1,1). Hodnoty hradel jsou vyznačeny dvojicemi hodnot poruchový/neporuchový. Vektor detekuje poruchu dokonce na dvou výstupech. Pro neporuchový obvod bude na výstupu vektor (0,0) a pro obvod s touto konkrétní poruchou vektor (1,1).

Červeně jsou vyznačeny citlivé cesty. Citlivou cestou rozumíme takové nastavení vstupů hradel na cestě od poruchy k výstupu, aby se hodnoty výstupů těchto hradel v bezporuchovém a poruchovém obvodu lišily. Například pro poruchu Sa0 (trvalá nula) na některém ze vstupů hradla AND je třebe nastavit ostatní vstupy hradla na hodnotu 1. Výstupem hradla pak bude hodnota 1 ve správně fungujícím obvodu a hodnota 0 v obvodu s uvedenou chybou. Citlivé cesty se mohou dělit, jak bylo ukázáno na příkladu. Jejich opětovné spojení se nazývá rekonvergencí a bývá zdrojem problémů při hledání testovacích vektorů. V další fázi je třeba naopak nastavit primární vstupy tak, aby se hradla v obvodu nastavila na hodnoty zjištěné ve fázi propagace a všechny hodnoty hradel byly konzistentní. Zdánlivě jednoduchý postup komplikuje větvení obvodu. Algoritmus se tak musí vracet a zkoušet jiné cesty respektive jiná nastavení hradel. Vracení se, neboli backtracking, je pro strukturální algoritmy typický. V této práci se strukturálními metodami nebudu blíže zabývat, ale je dobré si uvědomit jejich princip a určité analogie s algebraickými metodami. Zmíním jen efektivní algoritmy D-algoritmus a PODEM, které tvoří základ většině moderních algoritmů. Blíže se jimi zabýval například [13].

2.3 Algebraické metody v ATPG

V předešlé části byl aplikován strukturální pohled na kombinační obvod. Algebraické metody se snaží problém nalezení testovacího vektoru pro danou poruchu převést na problém řešení algebraického výrazu, ten vyřešit a z něj odvodit řešení původního problému. Nadneseně řečeno formulovat problém tak, aby ho byl schopen vyřešit výborný matematik s nulovým povědomím o hradlech či kombinačních obvodech.

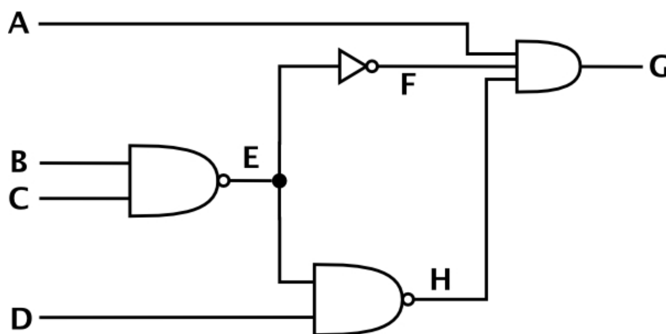
2.3.1 Obvod jako funkce vnitřních proměnných

Kombinační obvod přiřazuje konkrétnímu nastavení vstupů hodnoty výstupů, lze na něj tedy pohlížet jako na logickou funkci: $(y_1, y_2, ..., y_m) = f(x_1, x_2, ..., x_n)$ kde, Y_m je m-tice primárních výstupů a X_n je n-tice primárních vstupů.

Jestliže funkce $f : X_n \rightarrow Y_m$ představuje správně fungující obvod, pak obvod s chybou reprezentuje funkce $f^c : X_n \rightarrow Y_m$. Pro otestování chyby hledáme X_n takové, že $f(x_1, x_2, ..., x_n) \oplus f^c(x_1, x_2, ...x_n) = 1$. Požadovek na rovnost rovnice lze interpretovat jako problém splnitelnosti booelovské formule.

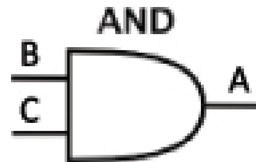
2.3.2 Transformace kombinačního obvodu na formuli v CNF formě

Cílem transformace je získat formuli v CNF. Na příkladu ukáži, jak můžeme postupovat.



Obrázek 2.2: ukázkový obvod

Obvod z obrázku 2.2 lze popsat: $G \equiv A \wedge F \wedge H \equiv A \wedge (\neg E) \wedge (\neg(E \wedge D) \equiv \ldots$ další postup by byl pomocí transformací vzplývajících ze zákonů booleovy logiky. Přestože to není na první pohled patrné, převodem logické formule do CNF formy může počet logických výrazů exponencielně narůst. Prakticky se proto používají Tsietsienovy transformace, které nám dovolí převést obvod přímo do CNF formy v lineárním čase podle počtu hradel. Negativním důsledkem je přidávání dalších proměnných reprezentujících jednotlivá hradla obvodu.



Obrázek 2.3: hradlo and

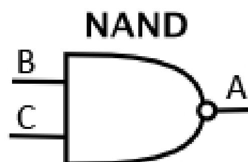
Postup převodu lze ukázat na hradlu AND (obr. 2.3) Formule odpovídající hradlu je $A \equiv B \wedge C$. Dále postupujeme pomocí transformací:

$$A \equiv B \wedge C$$

$$(\neg A \vee (B \wedge C)) \wedge (\neg(B \wedge C) \vee A)$$

$$(\neg A \vee B) \wedge (\neg A \vee C) \wedge (A \vee \neg B \vee \neg C)$$

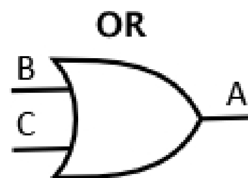
Postup lze zobecnit na N-vstupové hradlo AND $A = AND(X_n)$. Transformací dostáváme N binárních termů $(\neg A \vee x_i)$, na které můžeme pohlížet jako na implikace $A \rightarrow x_i$ nebo $\neg x_i \rightarrow A$. Dále se ternární term 2-vstupového hradla AND rozšířil na n-nární term $(A \vee \neg x_1 \vee \neg x_2 \vee ... \vee \neg x_n)$ vyjadřující vztah $(x_1 \wedge x_2 \wedge ... \wedge x_n) \rightarrow A)$. Pro ostatní základní hradla je postup transformace podobný.



Obrázek 2.4: hradlo nand

2-NAND (obr. 2.4) $(A \vee B) \wedge (A \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$

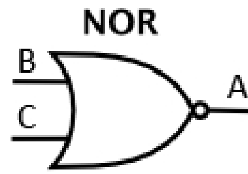
n-NAND $(A \vee x_1) \wedge (A \vee x_2) \wedge ... \wedge (A \vee x_n) \wedge (\neg A \vee \neg x_1 \vee \neg x_2 \vee ... \vee \neg x_n)$



Obrázek 2.5: hradlo or

2-OR (obr. 2.5)$(A \vee \neg B) \wedge (A \vee \neg C) \wedge (\neg A \vee B \vee C)$

n-OR $(A \vee \neg x_1) \wedge (A \vee \neg x_2) \wedge ... \wedge (A \vee \neg x_n) \wedge (\neg A \vee x_1 \vee x_2 \vee ... \vee x_n)$



Obrázek 2.6: hradlo nor

2-NOR (obr. 2.6)$(\neg A \vee \neg B) \wedge (\neg A \vee \neg C) \wedge (A \vee B \vee C)$

n-NOR $(\neg A \vee \neg x_1) \wedge (\neg A \vee \neg x_2) \wedge ... \wedge (\neg A \vee \neg x_n) \wedge (A \vee x_1 \vee x_2 \vee ... \vee x_n)$

Obrázek 2.7: hradlo xor

XOR (obr. 2.7)$(\neg A \vee B \vee C) \wedge (A \vee \neg B \vee C) \wedge (A \vee B \vee \neg C) \wedge (\neg A \vee \neg B \vee \neg C)$

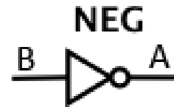
Obrázek 2.8: buffer

BUFFER (obr. 2.8)$(\neg A \vee B) \wedge (A \vee \neg B)$

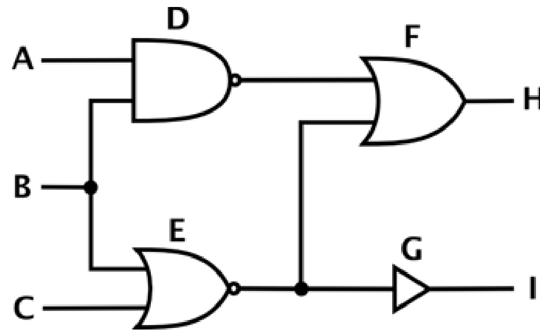
Invertor (obr. 2.9)$(A \vee B) \wedge (\neg A \vee \neg B)$

Pomocí Tsientsienových transforamcí lze pro jakýkoliv kombinační obvod získat jeho logickou formuli v CNF. Pro ATPG se využívá booleovské diference funkčního obvodu a obvodu s poruchou.

Příkladem je obvod z obrázku 2.10 s chybou Sa0 na vstupu hradla E. Poruchový obvod je znázorněn na dalším obrázku 2.11. Obvod reprezentující booleovskou diferenci vznikne spojením výstupu obvodů pomocí hradel XOR. Spojením jejich výstupů hradlem OR získáváme obvod 2.12, který pro každý vstupní vektor testující dotyčnou poruchu bude mít na výstupu logickou 1. (Pozn. A=A' , B=B', C=C') Transformací tohoto obvodu získáme logickou formuli v CNF reprezentující všechny vektory schopné testovat poruchu. Klasické algebraické metody nyní používají algebraické manipulace pro nalezení správného ohodnocení. Další možností, která se objevila na počátku 90. let minulého století [1], je řešení splnitelnosti této formule pomoci SAT řešiče.



Obrázek 2.9: Invertor



Obrázek 2.10: Bezporuchový obvod

2.4 SAT řešiče

SAT řešiče jsou nástroje pro řešení SAT problému. Implementují 2 odlišné skupiny algoritmů, kompletní a nekompletní [14].

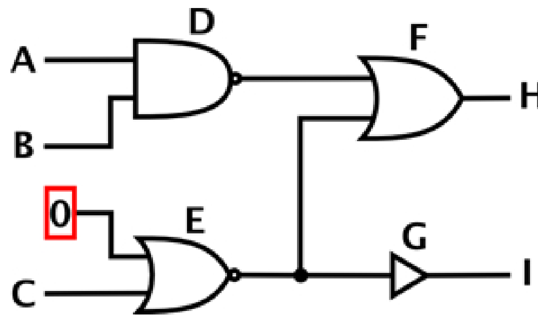
2.4.1 Kompletní SAT algoritmy

Kompletní SAT algoritmy jsou ty, které pro každou instanci SAT problému dávají odpověď ano/ne v konečném počtu kroků. Patří sem zejména:

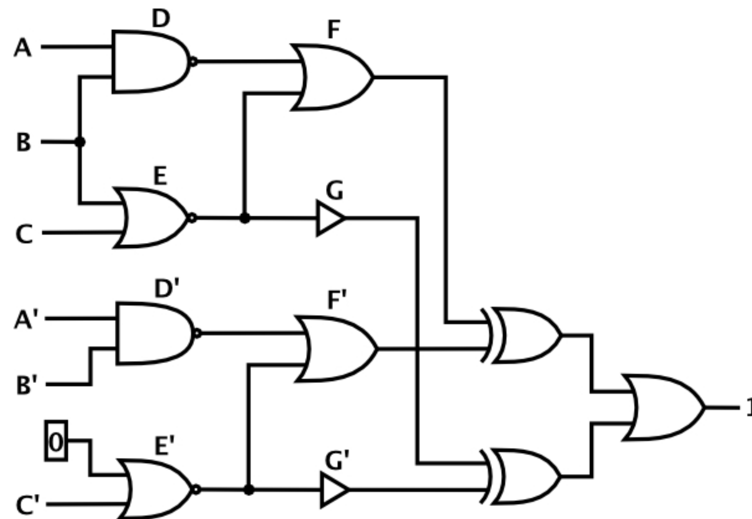
- Rezoluce (Resolution)
- Zpětné prohledávání (Backtrack search)
- Rekurzivní učení (Rekurzive learning)

Autory prvního algoritmu řešící SAT založeného na rezoluci jsou Davis a Putham. Jeho nedostatkem je epxponencielní paměťová náročnost. Paměťová náročnost rezolučního algoritmu byla odstraněna pomocí back-trackingu (Davis, Logemann and Loveland) a tato verze je dnes známá pod jménem DLL algoritmus [15]. Jedná se o algoritmus, který využívá většina moderních SAT řešičů. Pro Formuli F v CNF vypadá algoritmus zhruba takto [7]

```
function DPLL(F)
   if F is a consistent set of literals
       then return true;
   if F contains an empty clause
       then return false;
   for every unit clause l in F
      F=unit-propagate(l, F);
   for every literal l that occurs pure in F
      F=pure-literal-assign(l, F);
   l := choose-literal(F);
   return DPLL(F and l) OR DPLL(F and not(l));
```



Obrázek 2.11: Obvod s poruchou



Obrázek 2.12: Booleovská diference

Unit clause je klauzule obsahující jen jeden nepřiřazený literál, zbytku je přiřazena hodnota false. Aby byla tato klauzule splněna, je mu přiřazena hodnota v závislosti na polaritě a následně je provedena propagace. Pure literal označuje takový literál, který se ve všech klauzulích vyskytuje se stejnou polaritou. Proto ho můžeme nahradit hodnotou 1. Nejdůležitější funkcí celého algoritmu je operace Choose-literal. Výběr správné proměnné pro větvení algoritmu je klíčový pro celkovou efektivitu algoritmu. Dobrá heuristika výběru může značně zvýšit šance na rychlé nalezení řešení. V moderních řešičích je přidáváno učení založené na konfliktech. To umožňuje vyhnout se větvím stavového prostoru, o kterých už víme, že jsou neperspektivní.

2.4.2 Nekompletní SAT algoritmy

Nekompletní SAT řešiče využívají nesystematické prohledávání stavového prostoru, nevedou tedy zaručeně k cíli v konečném počtu kroků. Do této kategorie patří genetické algoritmy a algoritmy využívající lokální hledání. Příkladem je SAT řešič WalkSat, jehož autoři jsou Selman, Kautz a Cohen.

2.4.3 Preprocesory

Důležitou součástí SAT řešiče je preprocesor. Ten může výrazně zrychlit vlastní řešení tím, že sníží počet proměnných a klauzulí ve vstupní formuli. Použití preprocesoru může být v některých případech nevhodné, protože může dojít k deformaci stavového prostoru úlohy.

Příkladem preprocesoru je SatELite [11]

2.4.4 open-source SAT řešiče

SAT řešiče se velmi rychle vyvíjejí. Každý rok se pořádá soutěž, střídavě The international SAT Competitions [2] a Sat race, poslední ročník zde [4]. Setkávají se tam řešiče z komerční i akademické sféry. Výsledky této soutěže představují do jisté míry objektivní měřítko výkonosti Sat řešičů. Přesto je nutné si uvědomit, že instance problému řešené v této soutěži mohou mít jiný charakter, než mají instance generované z kombinačního obvodu. Dalším zdrojem informací o problematice SAT je internetový portál SatLive [3] shromažďující aktuální informace z problematiky řešení SAT problémů.

2.5 Atalanta-M

Atalanta-M [17] je nástroj spojující ATPG a simulátor poruch. Jedná se o modifikaci programu Atalanta vyvinutého na Virginia Polytechnic & State University. Jeho funkce dobře popsal K. Staufčík ve své diplomové práci [?]. Pomocí Atalanty jsem si udělal konkrétní představu a podobě vstupů a výstupů aplikace a v průběhu vývoje aplikace Atalanta sloužila jako kontrolní nástroj pro dílčí výsledky. Po dohodě s vedoucím diplomové práce jsem se rozhodl přizpůsobit ovládání aplikace Atalantě.

2.6 Cíl práce

Cílem práce je vytvořit Automatický generátor testovacích vektorů za využití open-source SAT řešiče a jeho porovnání s referenčním nástrojem, Atalantou.

Výsledný nástroj by v souladu se zadáním měl mít tyto parametry:

- Vstupní formát ISCAS 89 [18]
- Formáty výstupních souborů viz Atalanta
- Možnost načtení seznamu poruch
- Možnost ovlivnit počet generovaných DC stavů
- Vypínatelná kompakce
- Mozšíření o další SAT řešiče.

Způsob řešení dílčích problémů je popsán v další kapitole.

Kapitola 3

Analýza a návrh řešení

V této kapitole jsou navržena některá konkrétní řešení dílčích problémů. Zmíním zde výběr open-source SAT-řešiče, který je použit pro ATPG nástroj, konkrétní způsob převodu obvodu do CNF formy pomocí Tsientsienových transformací a návrh postupů získání DC hodnot a více testovacích vektorů pro jednu poruchu.

3.1 Výběr SAT-řešiče.

Při výběru SAT řešiče jsem si vytýčil několik kritérií:

- Open-source charakter
- Výkonnost
- Dokumentace
- Portabilita
- Kompletní algoritmus
- Schopnost generovat neurčené hodnoty v řešení.

3.1.1 Open-source charakter

Standarty open-source programů definuje nezisková organizace Open source initiative. Uveřejňuje také seznam open-source licencí. Pro úplnost zmíním hlavní myšlenky open-source programů, které jsou důležité pro mou práci:

- Volně šiřitelné
- Dostupné kompletní zdrojové kódy
- Možnost modifikovat a šířit dále
- Žádná omezení co se týče platformy nebo použití s jiným programem

Řešiče, na které jsem narazil, většinou neměly dostupné zdrojové kódy, nebo neobsahovaly žádnou licenci, nebo obsahovaly vlastní licenci v open-source duchu. Některé, jako je například chaff [8] je dostupný pro nekomerční použití. RSAT je licencován jen pro nekomerční, výzkumné a výukové účely. Naopak Minisat je šířen pod open-source licencí MIT uznávanou organizací Open-source initiative.

3.1.2 Výkonnost

Kritérium výkonnosti splňují všechny řešiče, které se v soutěžích [2],[4] umístily na předních příčkách. Podrobněji jsem si prostudoval dobře zdokumentované výsledky ročníku 2006. Na předních příčkách se v posledních 2 letech umístily řešiče RSAt, PikoSAT, March KS, Minisat 2.0, Eureka 2006, satZilla.

3.1.3 Dokumentace

Co se týče dokumentace, nejlepším se jeví Minisat [6]. Většina ostatních řešičů je zdokumentována pouze obecním popsáním principů, ale vlastní implementace není příliš popsána. Naopak Minisat je poměrně dobře popsán v [10]. Heuristiky a další postupy jsou zde vysvětleny s odkazem na konkrétní metody programu.

3.1.4 Portabilita

Většina řešičů je napsána v jazyce C++, pro který existuje překladač g++ [16] dostupný pro většinu platforem. Protože programy neobsahují grafické uživatelské rozhraní, neměly by jejich případné použití komplikovat na platformně závislé knihovny. Pod portabilitu bych zahrnul i schopnost sat řešiče poskytovat aplikační rozhraní externím programům. To, jak snadno lze připojit SAT řešič k jiné aplikaci, se pozná až při vlastní tvorbě aplikace. Přesto lze vypozorovat jistá vodítka, jako je třeba zapouzdření vlastního řešiče do jedné třídy.

3.1.5 Kompletní algoritmus

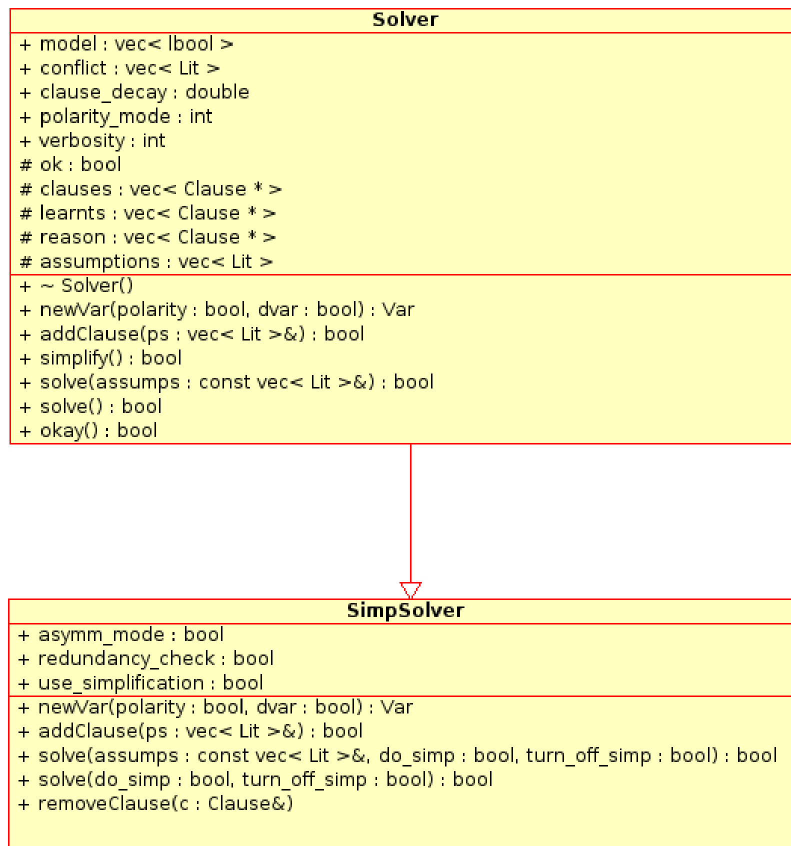
Pro generování vektorů je stěžejní nalézt pro každou poruchu testovací vektor, nebo zjistit, že porucha je redundantní a testovací vektor neexistuje. Proto jsem hledání omezil na kompletní algoritmy. SAT soutěží se účastní pouze řešiče s kompletními algoritmy a SAT řešiče založené na DLL algoritmu převažují i celkově.

3.1.6 Schopnost generovat neurčené hodnoty v řešení

Nepodařilo se mi nalézt žádný řešič, který by umožňoval hledání řešení obsahující DC stavy. Později jsem se dozvěděl, že ke stejném závěru došel i Lomitzki ve své bakalářské práci[12] Generování DC stavů jsem proto podrobil bližšímu zkoumání v této práci.

3.1.7 Minisat

Po zvážení výše zmíněných kritérií jsem se rozhodl použít Minisat 2.0, poslední verzi SAT řešiče. Požadavky splňuje výborně, je to výkonný, poměrně dobře zdokumentovaný open-source SAT řešič. Je založen na DLL proceduře, používá obvyklé heuristiky a postupy popsané v ??. Využívá například i učení, kdy jsou do databáze klauzulí předávány nové klauzule reprezentující konflikty. Heuristika je použita i na výběr přidaných klauzulí, které jsou zahazovány z důvodu nadměrného růstu databáze klauzulí. Jeho velkou výhodou je poměrně obsáhlá dokumentace na vlastním internetovém portálu. V [10] jsou popsány principy a také metody, které tyto principy implementují. Vlastní řešič je implementován ve třídě Solver. Voláním metod instance této třídy je řešeno aplikační rozhraní s obslužnými programy. Ve verzi Minisat 2.0 je k dispozici i vestavěný preprocesor, kde třída SimpSolver rozšíruje třídu Solver(obr 3.1 . Pro moji aplikaci to znamená značné zjednodušení, protože formule generované ze struktury



Obrázek 3.1: Třídy Solver a SimpSolver řešiče minisat

obvodu se vyznačují velkým počtem nadbytečných klauzulí a proměnných, které je třeba redukovat použitím preprocesoru.

3.2 Převod obvodu do cnf formy

V předchozí kapitole byl ukázán princip převodu obvodu do CNF pomocí Tsientsienových transformací. Ten lze aplikovat více způsoby. První možností je fyzické zkopírování poruchové části obvodu, spojení vstupů pomocí hradel XOR a OR, jak je ukázáno na obrázku 2.12. Transformace by pak vypadala zhruba následovně:

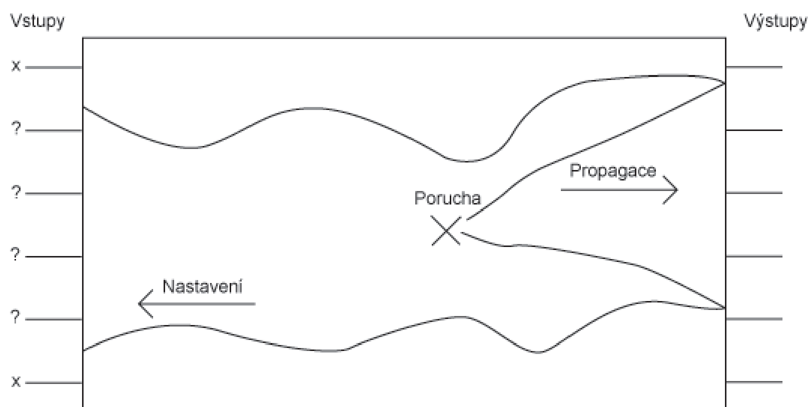
```
 procedure transformate(Circuit,CNF,Fault)
 begin
   CNF := empty CNF;
   Circuit_faulty := Circuit;
   Circuit_faulty.make_fault(Fault);
   Circuit_C := OR(Circuit XOR Circuit_faulty)
   For each gate from Circuit_C do
     CNF.add_clauses(g);

end
```

Na tomto způsobu je zřejmá nevýhoda kopírování obvodu, protože obě kopie jsou až na místo poruchy identické. Proto nám pro transformaci postačí jedna kopie. Musíme ovšem označit

místo poruchy pro vygenerování rozdílných klauzulí pro místo poruchy.

```
 procedure transformate_2(Circuit,CNF,Fault)
 begin
   CNF := empty CNF;
   Circuit.mark_faulty_gates();
   For each gate from Circuit
     begin
       CNF.add_clauses(g);
       CNF.add_faulty_clauses(g);
     end
end
```

Tento způsob není stále optimální, protože se zbytečně generuje velké množství identických klauzulí pro místa obvodu, která nejsou ovlivněny poruchou.



Obrázek 3.2: Stav obvodu s poruchou

Na obrázku 3.2 je znázorněn stav číselného obvodu s poruchou. Hradla se mohou nacházet ve čtyřech základních stavech:

1. **Poruchový**: Na vstupu hradla se nachází porucha. Generovány budou klauzule pro neporuchový obvod a pro poruchový.

2. **Dopředný**: Hradlo je ovlivněno poruchou a bude generovat klauzule pro poruchový obvod a neporuchový obvod.

3. **Zpětný**: Hradlo není ovlivněno poruchou a vede z něj cesta do některého primárního výstupu obvodu ovlivněného poruchou. Hradlo bude generovat klauzule pro neporuchový obvod.

4. Neaktivní: Hradlo není ani v jednom z předchozích stavů a nebude generovat žádné klauzule.

Termíny Dopředný a Zpětný jsem odvodil od způsobu, jak lze hradla v těchto stavech nalézt. Při představě kombinačního obvodu jako acyklického orientovaného grafu lze Dopředná hradla nalézt prohledáváním od Poruchových hradel ve směru orientace hran. Zpětná hradla lze naopak nalézt prohledáváním od Dopředných výstupů proti směru hran. Prohledávání lze realizovat pomocí zásobníku. Stavy se rozšíří kvůli zaznamenání už navštívených hradel.

```
procedure transformace_3 (Circuit,CNF,F)
var
  Stack_forward;
  Stack_backward;
begin
  CNF := empty CNF;
  for each g in Circuit do
    g.state := NEAKTIVNI;
  for each g on F do
    begin
      g.state := PORUCHOVY;
      Stack_forward.push(g);
    end
  while (Stack_forward not empty) do
    begin
      g := Stack_forward.pop();
      for each h in g.outputs() do
        if (h.state=NEAKTIVNI) then
          begin
            h.state := DOPREDNY;
            Stack_forward.push(h);
          end
      if (g is output) then
        begin
          g.state := DOPREDNY_ZPET;
          Stack_Backward.push(g);
        end
    end
  while (Stack_Backward not empty) do
    begin
      g := Stack_Backward.pop();
      if (g.state <> ZPETNY) then
        CNF.add_clauses_faulty(g);
      CNF.add_clauses(g);
      for all h in g.inputs do
        case h.state of
          DOPREDNY: begin
                      h.state := DOPREDNY_ZPET;
                      Stack_backward.push(h);
                    end
          NEAKTIVNI: begin
                       h.state := ZPETNY;
                       Stack_backward.push(h);
                     end
          PORUCHOVY: begin
                       h.state := PORUCHOVY_ZPET;
                       Stack_backward.push(h);
                     end
        end
```

```
    end
end
```

Tuto transformaci jsem se rozhodl naimplementovat v generátoru. Časová složitost transformace je $T(N, V) = O(N + V)$, kde N je počet hradel a V počet vodičů. Za předpokladu N V(typické pro kombinační obvody) je složitost $T(N) = O(N)$. V typickém kombinačním obvodu je složitost transformace lineární.

3.3 Neurčené stavy v řešení SAT

Žádné dostupné open-source SAT řešiče nedokáží najít řešení obsahující neurčené hodnoty (DC). Ve své práci jsem se snažil nalézt způsob, jak tyto neurčené hodnoty nalézt. V principu existují dvě možnosti:

- modifikace SAT řešiče
- modifikace řešení SAT řešiče

3.3.1 Modifikace SAT řešiče

Modifikací SAT řešiče se ve své bakalářské práci zabývá Jan Lomitzki [12]. Pozměnil DLL proceduru řešiče Minisat tak, že literálům jsou přiřazovány kromě hodnot 1 a 0 i hodnoty DC. Rozšíření na DC hodnoty ale přináší větší časovou náročnost programu, což se ukázalo i při měření, které provedl. Rozhodl jsem se jeho verzi přidat k výsledné aplikaci a otestovat na benchmarcích.

3.3.2 Modifikace řešení SAT řešiče

3.3.2.1 Modifikace řešení pomocí formule

Modifikace řešení je možná dvěma přístupy. Prvním je manipulace s proměnnými ve vztahu k formuli:

1. Získání ohodnocení promměnných, které řeší problém aplikací SAT řešiče
2. Nastavení ohodnocení proměnných reprezentujících primární vstupy na DC
3. Odstranění všech klauzulí, které jsou splnitelné pomocí literálů
4. Ohodnocování DC proměnných původní hodnotou, dokud není každá klauzule splněna

Označíme formuli F, klauzule k, literál l, množinu proměnných P,proměnné p a množinu vstupních proměnných V. $V \subseteq P$. Algoritmus v pseudokódu:

```
procedure cnf_dc(P,F,V)
begin
ohodnoceni(P):= sat_reseni(F)
for each p in V do //pro všechny proměnné reprezentující primární vstupy
  begin
```

```
    store(p); //uložení stavu proměnné p
    p:=DC;   //nastavení na neurčenou hodnotu (DC)
  end
do
  begin
    pocet := F.clause_count(); //pocet klauzulí v F
    for each k in F do // Cyklus A pro všechny klauzule v F
      begin
        for each l in k do// Cyklus B pro každý literál v klauzuli
          begin
            if (value(l)=polarity(l)) then  //literál má hodnotu true
              begin
                F.remove(k);   //odstranění klauzule
                continue(A);   //pokračování cyklu A na další klauzuli
              end
            else if (value(l)=-polarita(l)) then //literál má hodnotu false
                   begin
                     k.remove(l); //odstraněni literálu z klauzule
                   end
          end //Cyklus B
        if (k.literal_count = 0) then // Je-li klauzule prázdná
          begin
            F.remove(k); //odstranění klauzule
            continue(A); //pokračování cyklu A na další klauzuli
          end
        if (k.literal_count = 1) then // Obsahuje pouze jeden literál
          begin
            load(l); //načte hodnotu proměnné, kterou reprezentuje literál
            F.remove(k); //odstranění klauzule
            continue(A); //pokračování cyklu A na další klauzuli
          end
      end //cyklus A
    if (pocet = F.pocet()) then //počet klazulí se nezměnil
      begin
        choose_variable(F,V); //vybereme vhodnou proměnnou
        load(v); //načte hodnotu proměnné v
      end
  end //do Cyklus
while (not F.empty())
end //procedure
```

Algoritmus **??** na začátku uloží hodnoty proměnných reprezentujících primární vstupy (dále jen vstupní proměnné) a nastaví je na neurčené hodnoty. Důležitý je první průchod hlavním cyklem do-while smyčky. Při něm se odstraní všechny klauzule, kde nejsou obsaženy literály vstupních proměnných. Dále se ze zbylých klauzulí odstraní všechny literály nevstupních proměnných a rovněž klauzule, kde tyto literály zaručovaly pravdivé ohodnocení. Při druhém a dalším průchodu zbývají jen klauzule obsahující literály vstupních proměnných. Klauzule s jedním literálem jsou odstraněny a příslušné vstupní proměnné je načtena původní hodnota. Jestliže při průchodu nedojde k odebrání žádné klauzule, vybere se vhodná vstupní proměnná a přiřadí se jí původní hodnota. Výběr vhodné vstupní proměnné je klíčovým místem ovlivňující úspěšnost al-

goritmu. Jako nejvýhodnější se mi jeví taková strategie, která ohodnocením proměnné odstraní co nejvíce klauzulí, jako je to patrné na příkladě:

Example 3.3.1 Zbývající proměnné: $a(not), b(not), c(and)$ Zbývající klauzule: $(\neg a \vee \neg b), (\neg b \vee c)$

Je zřejmé, že nejvýhodnější je ohodnotit proměnnou b, což způsobí odebrání obou klauzulí. Proto jednoduchá heuristika vypadá takto:

```
index function choose_variable(F)
histogram[];
begin
for each klauzule k in F do
  begin
    for each literal l in k do
       begin
         if (value(l)=polarity(l)) then
           begin
             histogram[p(l)]++;
           end
       end
  end
return index of maximum from histogram[];
end //function
```

V průběhu testování jsem zjistil, že tuto heuristiku lze ještě vylepšit. V klauzulích se totiž mohou vyskytovat pro proměnou jen literály s opačnou polaritou. To znamená, že můžeme znegovat původní hodnotu a vylepšit tak řešení. Heuristická funkce pak nehledá jen index nejvýhodnější proměnné, ale také je schopna změnit původní řešení znegováním proměnné. pro ukázku pozměněný příklad:

Example 3.3.2 Zbývající proměnné: $a(not), b(and), c(and)$ Zbývající klauzule: $(\neg a \vee \neg b), (\neg b \vee c)$

Původní heuristika by vybrala ve 2 krocích proměnné a a c. Pozměněná zneguje b a vybere ji:

```
index function choose_variable_2(F,V)
histogram_good[]; //literál reprezentující proměnou splní klauzuli
histogram_bad[];  //literál reprezentující proměnou nesplní klauzuli
begin
for each klauzule k in F do
  begin
    for each literal l in k do
       begin
         if (value(l)=polarity(l)) then
           begin
             histogram_good[p(l)]++;
           end
         else
```

```
           begin
             histogram_bad[p(l)]++;
           end
       end
 end
for each v in V do
  begin
    if (histogram_good[v]=0) then
      begin
        histogram_good[v] := histogram_bad[v];
        load[v];
        v := \neg v;
        store v;
      end
  end
return index of maximum from histogram_good[];
end //function
```

Tento algoritmus by měl dávat výsledky podobné modifikovanému DC řešiči. Na jednoduchém příkladu si ukážeme, že je limitován nutnou podmínkou ohodnotit všechny klauzule hodnotou 1. Pro každá 2 hradla spojená vodičem platí, že jejich odpovídající proměnné nemohou mít najednou hodnotu DC, protože jejich vztah je reprezentován binární klauzulí (U hradla XOR jsou to ternární klauzule, ale jejich kombinace znemožňuje současné DC).Jakýkoliv algoritmus operující pouze s formulí reprezentující poruchu bez znalostí dalších informací nemůže nalézt maximální možný počet neučených hodnot. I navržený algoritmus jistou informaci navíc používá, je to znalost toho, které literály reprezentují primární vstupy obvodu.

3.3.2.2 Modifikace řešení pomocí strukturáního algoritmu

Druhý přístup k řešení vede k návrhu algoritmu založenému na znalosti struktury obvodu, který by měl být schopen ohodnotit sousední proměnné DC hodnotami současně. Nejprve je třeba si všimnout, za jakých podmínek může hradlo nabývat hodnoty DC. Je to tehdy, když jeho výstup neurčuje hodnotu žádného následného hradla. Například hradlo AND potřebuje na hodnotu 0 alespoň jeden vstup nastavený na 0. Aby algoritmus mohl skutečně efektivně měnit hodnoty vstupního vektoru na DC hodnoty, musí být schopen vyhodnotit co nejvíce vnitřních hradel jako DC nositele. K tomu je potřeba poznat aktivní hradla, tedy ta hradla, po které vede citlivá cesta propagující poruchu na výstup. Jestliže jsou známy hodnoty proměnných v obvodu bez poruchy a obvodu s poruchou, lze jednoduše nalézt pomocí prohledávání do hloubky všechny citlivé cesty. Potřebujeme jen seznam výstupních hradel *Out*.

```
boolean function gate.differs()
begin
  return (variable in non-faulty circuit = \neg variable in faulty circuit)
end

procedure find_active_gates(Out)
begin
Stack s;
for each g in Out do
  begin
```

```
    if g.differs() then
      begin
        g.visited := true;
        s.push(g);
      end
  end
while (not s.empty())do
   begin
     g = s.pop()
     active := true;
     for each h in g.inputs do
       begin
         if ((not h.visited) and  (h.differs())) then
           begin
             h.visited := true;
             s.push(h);
           end
       end
   end
end //procedure
```

Po označení citlivých cest zbývá formulovat vlastní způsob hledání DC hodnot. U každého hradla platí, že je třeba nejdříve určit možnosti DC hodnot jeho následovníků. Vzhledem k tomu, že kombinační obvod je acyklický graf, lze využít jeho topologického uspořádání:

```
bool function gate.agree(bool input_var)
begin
  if (input_var) then
    return (free_ones > 0)
  else
    return (free_zeros > 0)
end

procedure find_dc(Circuit)
begin
  for each gate g in Circuit in reverse topological order do
    begin
      if !((g is output) and (g is active)) then
        begin
          s := first_succesor(g)
          while s.agree(g.var)
            begin
              s := next_succesor(g);
            end
         if (all succesors agree) then
           begin
             g.set(DC);
             all succesors.decrease(var);
           end
```

```
        end
      g.compute free_values();
    end
end
```

Algoritmus je zřejmý, každé hradlo zkoumá, jestli je jeho výstup pro všechny jeho následovníky nezbytný. Pokud jeho výstup ani jeden následník nepotřebuje, nastaví se na hodnotu DC. V závslosti na své hodnotě pak nastaví, kolik volných vstupů nastavených na hodnotu jedna a na hodnotu nula může postrádat. Příkladem budiž trojvstupové hradlo and s hodnotou 0, která se v poruchovém obvodu neliší. Hradlo prozkoumá vstupy a zjistí, že hodnoty vstupů jsou pro poruchový/neporuchový obvod 0/1,0/0,0/0. Hradlo tedy nastaví počet volných nul na jedna a počet volných jedniček na nula. Algorimus nepovažuji za příliš dobře navržený a efektivní, ale přesto mne zajímalo jeho srovnání s algoritmem uvedeným v předchozí sekci.

3.4 Dominance

Ekvivalence je využíváná při generování seznamu poruch. Dominance poruch umožnuje odstranit dominující poruchy, pokud je nalezen testovací vektor k poruše dominované. Poruchy lze odstraňovat na citlivé cestě, pokud nedochází k rekonvergenci. Poruchy lze odstraňovat na citlivé cestě tak dlouho, dokud nedojde nedojde k větvení vodičů. Dominanci lze tedy jednoduše implementovat a v mé práci bude využita.

3.5 Kompakce

Kompakce testovacích vektorů není pro mou práci příliš důležitá, proto jsem se rozhodl implementovat jen jednoduchou kompakci. Každý nový vektor je při přidávání do sady vektorů porovnáván se všemi vektory a v případě, že jsou nalezeny kompatibilní vektory, sloučí se nový vektor tak aby vznikl vektor s co nejvíce DC.

3.6 Redundantní proměnné

Pro zlepšení výkonu SAT řešiče se přidávají redundantní proměnné, které sice zvýší počet proměnných a klauzulí ve formuli, ale pomáhají nasměrovat SAT řešič k rychlejšímu nalezení řešení [1]. Jednou z popsaných metod je přidávání aktivačních proměnných. Ty vyjadřují, zda hradlo leží na citlivé cestě či nikoliv. U všech hradel s potenciálem citlivé cesty (Dopředná hradla) se přidává aktivační proměnná A. X je proměnná popisující normální stav hradla a X_f je proměnná popisující poruchový stav hradla. A_i jsou aktivační proměnné hradel na výstupu. Za těchto předpokladů lze formulovat vztahy:

- $A \rightarrow (X \equiv X_f)$
- $A \rightarrow (A_1 \vee A_2 \vee ... \vee A_n)$

Úpravou těchto vztahů se získají nové klauzule:

- $(\neg A \vee X \vee X_f), (\neg A \vee \neg X \vee \neg X_f)$
- $(\neg A \vee A_1 \vee A_2 \vee ... \vee A_n)$

Zbývá jen pro n poruchových hradel přidat klauzuli $(A_1 \vee A_2 \vee ... \vee A_n)$, kde A_i je aktivační proměnná i-tého poruchového hradla. Aktivační proměnné jsem se rozhodl použít v aplikaci.

Kapitola 4

Realizace

4.1 Volba programovacích prostředků

Pro implementaci programu jsem zvolil programovací jazyk C++. Jeho výhodou je robusnost a zejména dostupnost překladačů pro více platforem. Dalším argumentem ve prospěch volby tohoto jazyka je fakt, že Minisat2 je v tomto jazyce implementován. Cílem bylo vytvořit přenositelnou aplikaci, program jsem během vývoje překládal pomocí GNU kompilátoru g++ - pod operačními systémy Linux a Windows. V operačním systému Linux jsem používal pro vývoj prostředí KDevelope, které podporuje g++ kompilátor, debugger a Valgrind Memory Checker. V operačním systému Windows je k dispozici volně šiřitelné vývojové prostředí Bloodshed Dev-C++. Toto vývojové prostředí jsem používal pro překlad aplikace pro operační systém Windows, protože podporuje překladač g++.

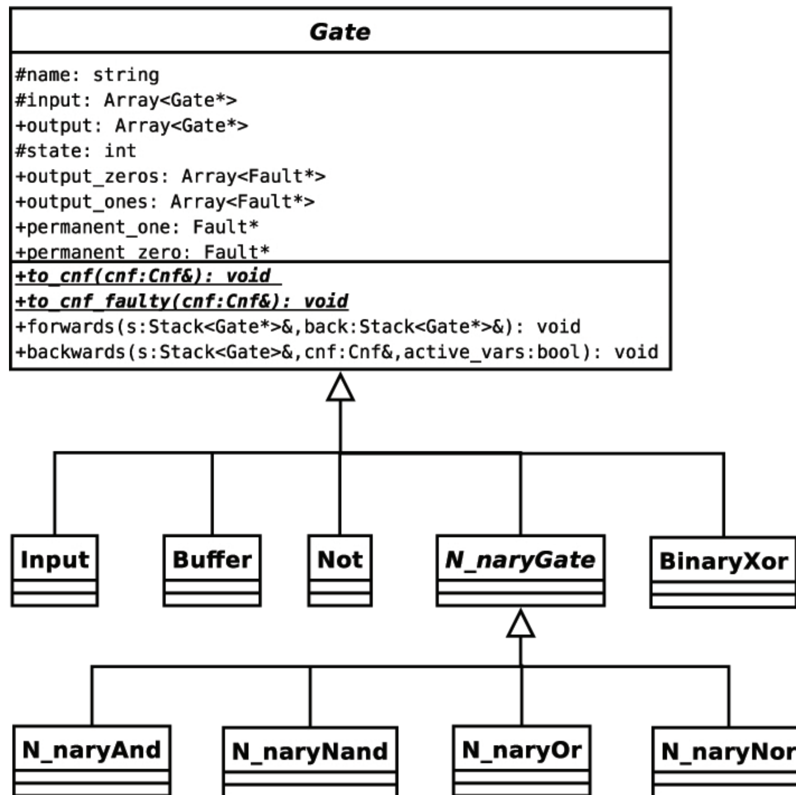
4.2 Návrh datových struktur

Rozhodl jsem se využít objektových rysů jazyka C++ a datové struktury jsem se snažil zapouzdřit do vhodně navržené objektové hierarchie. Stěžejní je návrh datového úložiště pro hradla. Rozhodl jsem se využít dědičnosti a polymorfismu pro snažší implementaci algoritmů procházejících grafovou strukturu obvodu. Toto řešení zároveň umožňuje v budoucnu snadno přidat další typy hradel jednoduchým poděděním základní třídy Gate.

Gate (diagram 4.1) je abstraktní třída obsahující základní data, která mají všechny typy hradel společné. Mimo jiné je to seznam poruch na výstupu z hradla. Pro generování klauzulí do logické formule se využívá polymorfismu. Virtuální metody to_cnf (Cnf &cnf) a to_cnf_faulty (Cnf &cnf)zajišťují přidávání klauzulí bezporuchového a poruchového obvodu. Jinou možností, jak se vypořádat s různými typy hradel, by bylo definování nové třídní proměnné třídy Gate, která by identifikovala typ hradla. Generování klauzulí by se pak řešilo kontrukcí Case podle typu hradla. Pak by ale bylo obtížné přidávat do aplikace nové typy hradel.

4.3 Podpora dalších SAT řešičů

V implementaci jsem se zaměřil na to, aby bylo snadné přidávat další SAT řešiče do aplikace. Vytvořil jsem proto abstraktní třídu Sat_Solver, která definuje rozhraní pro komunikaci s řešiči. Nový řešič lze přidat vytvořením potomka této abstraktní třídy a definováním abstraktních metod. MiniSat byl takto přidán vytvořením třídy MiniSat_Solver. Prostudováním zdrojového kódu v souboru satSolver.h lze jednoduše nahlédnout, jakým způsobem ATPG komunikuje se



Obrázek 4.1: Hierarchie třídy Gate a jejich potomků

sat řešičem. Konkrétní způsob předávání hodnot mezi řešičem a třídou Cnf reprezentující logickou formuli lze najít v souboru minisat2Solver, kde je implementováno rozhraní s MiniSatem. Při přidávání DC Satu [12] jsem musel řešit konflikt jmen mezi MiniSatem a DC Satem, který je modifikací starší verze MiniSatu. Vytvořil jsem proto pro oba řešiče oddělené jmenné prostory (name space). Je žádoucí, aby se nově přidávané řešiče uzavíraly do jmenných prostorů kvůli vzájemné kompatibilitě.

Kapitola 5

Testování

Testy byly prováděny na počítači s konfigurací Intel Pentium 4 3.00 Ghz, 512 MB RAM, pod operačním systémem Windows.

5.1 Celkové výsledky

Pokrytí bylo ověřováno pomocí simulátoru HOPE, který je obsažen v nástroji Atalanta. K měření pokrytí testovacích vektorů souboru example.bench slouží příkazy:

```
sat_atpg -t example.pat -W 1 <DALSI PARAMETRY> -n example.bench
atalanta-M -S -t example.pat -P example.rep -n example.bench
```

Protože dle [13] Atalanta v normálním módu nepokrývá všechny poruchy, byly pro měření času a pokrytí atalanty použity příkazy:

```
atalanta-M -D 1 -t example.pat -W 1 -n example.bench
atalanta-M -S -t example.pat -P example.rep -n example.bench
```

5.2 Celkové výsledky

V první tabulce 5.1 jsou přehledně uvedeny hlavní paramatry programu, tedy počet pokrytých chyb, počet vektorů a celkový čas.V prvním sloupci je uveden celkový počet poruch ve vygenerovaném seznamu. Ve druhém sloupci je počet poruch, které byly odstraněny pomocí vztahu dominance. Netestovatelné poruchy jsou takové, jejichž ekvivalentní booleovskou formuli prohlásil SAT řešič za nesplnitelnou. Efektivita kompakce je poměrně malá, protože při generování vektorů nebyl použit žádný z algoritmů zvyšující počet DC hodnot ve vektorech. Už na první pohled jsou celkové časy generování vektorů velmi dlouhé. To je potvrzeno i v následující tabulce, kde je ATPG nástroj porovnán s Atalantou.

bench	Poruchy			Vektory		DC rating	čas
	Celkem	Odstraněno	Netestovatelné	celkem	po kompakci		
c1355	1574	364	8	1202	1202	0.00	33.57
c17	22	6	0	16	9	0.02	0.00
c1908	1879	313	10	1556	1553	0.66	56.32
c2670	2747	425	118	2204	1701	46.90	59.28
c3540	3428	618	137	2673	2600	2.18	180.58
c432	524	106	4	414	408	0.31	4.78
c499	758	233	8	517	484	0.00	9.18
c5315	5350	858	59	4433	3471	57.46	171.40
c6288	7744	1920	34	5790	5719	0.16	871.31
c7552	7550	1418	131	6001	5669	41.32	261.35
c880	942	197	0	745	551	27.13	6.64

Tabulka 5.1: Přehled měření ATPG nástroje na ISCAS benchmarcích

Benchmark	sat atpg		Atalanta	
	čas [s]	pokrytí [%]	čas [s]	pokrytí [%]
c1355	33.57	99.49	0.05	99.49
c17	0.00	100.00	0.00	100.00
c1908	56.32	99.52	0.09	99.52
c2670	59.28	95.74	0.44	95.74
c3540	180.58	96.00	0.45	96.00
c432	4.78	99.24	0.02	99.05
c499	9.18	98.94	0.02	96.70
c5315	171.40	98.90	1.34	98.88
c6288	871.31	99.56	1.00	99.56
c7552	261.35	98.26	3.59	97.81
c880	6.64	100.00	0.06	100.00

Tabulka 5.2: Srovnání SAT ATPG a Atalanty

Obvod	Aktivační proměnné	
	ANO	NE
	čas [s]	čas [s]
c1355	33.57	38.16
c17	0.00	0.00
c1908	56.32	63.09
c2670	59.28	65.38
c3540	180.58	210.47
c432	4.78	4.98
c499	9.18	8.42
c5315	171.40	173.60
c6288	871.31	-
c7552	261.35	273.83
c880	6.64	6.20

Tabulka 5.3: Srovnání časů při použití aktivačních proměnných

5.3 Srovnání časů a pokrytí

Už na první pohled jsou celkové časy generování vektorů velmi dlouhé. To je potvrzeno i v následující tabulce 5.2, kde je ATPG nástroj porovnán s Atalantou. Zatímco procentelní pokrytí poruch je srovnatelné, časy jsou hlavním problémem SAT ATPG.

5.4 Aktivační proměnné

V tabulce 5.3 je zajímavé srovnání, kdy v prvním sloupci jsou uvedeny časy, při kterých byly generovány aktivační proměnné a v druhém sloupci časy, kdy výše zmíněné nebyly generovány. Aktivační proměnné zvyšují počet proměnných a klauzulí, ale potvrzují pozorovaný fakt, že redundantní klauzule mohou zkrátit výpočetní dobu, jestliže nesou užitečnou informaci [1]. Pro benchmark c6288 dokonce doba testu bez aktivačních proměnných překročila časový limit. V dalším vývoji aplikace by proto bylo vhodné prozkoumat další způsoby přidávání klauzulí s užitečnou informační hodnotou.

5.5 Počet DC hodnot

V dalším testu (tab 5.4) jsou porovnány počty DC hodnot v generovaných sadách. Aby bylo měření měření objektivní, pro srovnání jednotně použit parametr -D 1. To znamená, že všechny uvedené metody generování vektorů generovaly pro stejný seznam poruch stejný počet vektorů bez kompakce. Jak lze vypozorovat, Atalanta je nejlepší z hlediska počtu neurčených hodnot. Potvrdil se předpoklad, že strukturální algoritmy vygenerují více neurčených hodnot, než algoritmy založené na úpravě logické formule. Nevýhodou Atalanty je fakt, že generování neurčených hodnot neumí nijak řídit. Proto se mi perspektivním jeví algoritmus cnf, kde můžeme výběr proměnné s neurčenou hodnotou řídit úpravou heuristické funkce choose_variable() v algoritmu cnf_dc 3.3.2.1. DC_SAT dává zajímavé výsledky na benchmarku c17, těžší instance problému však není schopen v rozumném čase vyřešit.

	DC počet [%]				
Obvod	normální	cnf	struktrální	Atalanta	DC Sat
c1355	0.00	0.22	9.13	13.33	-
c17	0.11	27.27	37.27	43.64	34.55
c1908	0.66	1.73	29.27	44.69	-
c2670	46.90	78.91	88.18	90.47	-
c3540	2.18	25.28	56.13	74.61	-
c432	0.31	4.70	44.57	56.23	-
c499	0.00	0.68	13.09	0.94	-
c5315	57.46	70.39	90.09	92.51	-
c6288	0.16	-	18.72	19.30	-
c7552	41.32	62.21	80.89	84.84	-
c880	27.13	54.96	75.23	82.22	-

Tabulka 5.4: Porovnání počtu DC hodnot

Obvod	Klausule			Proměnné		Poměr klauzule / proměnné		
	maximum	průměr	binární	maximum	průměr	minimum	průměr	maximum
c1355	3493.00	2229.55	1348.33	1232.00	797.90	2.66	2.78	2.84
c17	57.00	31.73	15.82	24.00	15.09	1.67	2.03	2.38
c1908	4889.00	2576.37	1891.95	1744.00	958.02	2.30	2.63	2.81
c2670	6213.00	2138.35	1616.34	2318.00	862.73	1.38	2.47	2.68
c3540	11787.00	5018.55	3690.71	4184.00	1847.99	1.67	2.80	2.82
c432	1278.00	837.33	488.26	435.00	293.75	1.54	2.84	2.95
c499	1621.00	906.97	274.77	536.00	305.99	2.65	2.92	3.06
c5315	10367.00	2007.63	1504.78	3672.00	773.19	1.50	2.50	2.94
c6288	20227.00	9163.29	5573.01	6801.00	3101.99	1.50	2.96	2.98
c7552	27578.00	2835.54	2097.33	9877.00	1142.02	1.43	2.47	2.79
c880	2414.00	626.12	398.44	870.00	245.86	1.42	2.37	2.78

Tabulka 5.5: Počty klauzulí a proměnných v generovaných booleovských formulích

5.6 Klauzule a proměnné

Tabulka 5.5 zachycuje základní parametry generovaných booleovských formulí. Velký počet binárních klauzulí umožňuje vyslovit hypotézu, že dlouhé časy způsobuje nevhodná aplikace MiniSatu. Ten je sice považován za univerzální sat řešič, ale formule generované transformací obvodu mají specifický charakter. V citeLarrabee je použit pro řešení instancí SAT problému 2-SAT řešič. Předstupněm Minisatu by měl být modul umožňující efektivně předzpracovat klauzule například pomocí převodu na implikační graf a následné odstranění silných komponent.

Kapitola 6

Závěr

Naprogramoval jsem ATPG nástroj založený na řešení problému splnitelnosti booleovské formule. Výsledky jsem porovnal s nástrojem Atalanta. Měření ukázalo, že aplikace zaostává v rychlosti generování vektorů. Příčina může být jednak v neefektivních datových strukturách nebo v nevhodném použití SAT řešiče MiniSat. Pokrytí poruch na testovaných benchmarcích je srovnatelné s Atalantou. Prozkoumal jsem možnosti generování neurčitých hodnot v testovacích vektorech. DC Sat se ukázal nepoužitelný pro složitější obvody. Mnou navržené algoritmy na získání neurčených hodnot z konkrétního řešení se ukázaly poměrně efektivní. Algoritmus 3.3.2.1založený na úpravě vstupních proměnných postupným splňováním klauzulí umožňuje sice získat menší počet neurčených hodnot, ale jeho potenciál je ukryt v možnosti výběru proměnných, které se budou ohodnocovat. DC_SAT se osvědčil jen pro jednoduchý obvod a složitější kombinační obvody nebyl schopen vyřešit. Další postup ve vývoji aplikace by měl směřovat k vytvoření nového modulu, který by efektivně předzpracovával získané formule, například převodem do grafu implikací a odstraněním silných komponent. To je žádoucí vzhledem k velkému množství binárních klauzulí generovaných z kombinačního obvodu. Rychlost výpočtu by pozitivně ovlivnily i další pokročilé techniky transformace obvodové struktury na logickou formuli, jako je použití redundantních aktivačních proměnných, jejichž přínost potvrdilo měření.

Literatura

[1] 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.

[2] The international SAT Competitions web page `http://www.satcompetition.org/`.

[3] Up-to-date links for the SATisfiability problems `http://www.satlive.org/`.

[4] SAT-Race 2006 `http://fmv.jku.at/sat-race-2006/`.

[5] Open source `http://www.opensource.org/`.

[6] The minisat page `http://minisat.se/`.

[7] Wikipedia `http://en.wikipedia.org`.

[8] SAT Research at Princeton `http://www.princeton.edu/~chaff/`.

[9] RSAT `http://reasoning.cs.ucla.edu/rsat/`.

[10] Niklas Een, Niklas Sörensson An Extensible SAT-solver,SAT 2003.

[11] Niklas Een, Armin Biere Effective Preprocessing in SAT through Variable and Clause Elimination, SAT 2005.

[12] J. Lomitzki Řešení problému splnitelnosti booleovské formule (SAT) Bakalářská práce, FEL ČVUT v Praze, 2007.

[13] K. Staufčík Automatické generování testovacích vektorů pro kombinační číslicové obvody. Diplomová práce, FEL ČVUT v Praze, 2005.

[14] M. Kutil Splnitelnost Booleovských formulí. Research Report, Praha: ČVUT FEL, Katedra řídicí techniky, 2005.

[15] Davis, Martin; Logemann, George, and Loveland, Donald A Machine Program for Theorem Proving. *Communications of the ACM*, 5 (7): 394-397.

[16] GCC, the GNU Compiler Collection `http://gcc.gnu.org/`.

[17] Atalanta-M 1.1 `http://service.felk.cvut.cz/vlsi/prj/Atalanta-M/`.

[18] Iscas benchmarks `http://www.fm.vslib.cz/~kes/asic/iscas/`.

Dodatek A

Seznam použitých zkratek

ATPG Automatic Test Pattern Generator (Automatický generátor testovacích vektorů)

DC Dont Care (Neurčená hodnota)

SAT Satisfiability

TPG Test PAttern Generation (Generování testovacích vektorů)

⋮

Dodatek B

Uživatelská / instalační příručka

Použití programu sat_atpg:

```
sat_atpg [options] [-n] jmeno_souboru_bench
```

B.1 Parametry souborů

```
-n <FILE>       jméno vstupního souboru ve formátu bench.
-t <FILE>       jméno výstupního souboru pro sadu testovacích vektorů.
-f <FILE>       vstupní soubor se seznamem poruch
-F <FILE>       výstupní soubor seznamu poruch
-P <FILE>       výstupní soubor se zprávou
-U <FILE>       výstupní soubor pro netestovatelné chyby (narozdíl od Atalanty
```

-d ¡FILE/DIR¿ export formulí do DIMACS formátu. Pokud je v seznamu jedna porucha, bere se parametr jako soubor, pokud jich je více, bere se parametr jako adresář, do kterého jsou uloženy jednotlivé soubory reprezentující poruchy. Při aktivní volbě -d se neprovádí TPG.

B.2 Parametry TPG

-A pro každou poruchu v seznamu najde všechny vektory -D ¡n¿ pro každou poruchu v seznamu najde n vektorů -e ¡0,1,2¿ expanze DC hodnot v řešeni: 0 - žádné, 1 - cnf algoritmus, 2 - strukturaální algoritmus -N neprovádí se kompakce (pro volby D, A impicitní) -r ¡0,1,2¿ Redukce poruch:0 - žádná (jen ekvivalence při generování seznamu poruch), 1 - dominance při TPG (default), 2-riskantní (odebrány jsou i nepokryté poruchy vlivem konvergence citlivé cesty, spíše experimentální) -a připojí jen jednu řádku hodnot do souboru daného parametrem P. Má význam jen v průněhu více testů po sobě v jednom měření, je třeba přímo ve zdrojovém kódu funkce main() nastavit, které údaje budou vypisovány.

-ch seznam poruch vytváří dle Checkpoint theoremu. -W ¡0,1,2,3¿ Formát vektorů: 0 - žádný výstup, 1 - vstupy, 2 - vstupy a odezvy 3 - číslovaný seznam při volbě -D či -A.

-s přepne na dc_solver -l vytiskne popis obvodu -V nepoužívá aktivační proměnné

B.3 Formáty souborů

Převzato z [17]. BENCH Format

Vstupním formátem apliakce je ISCAS89 netlist formát. Řádky začínající # jsou komentáře. Pořadí hradel v souboru není pevně dáno. Jména hradel mohou být alfanumerické znaky (0-9, A-Z, a-z, _, [, or]).

```
Podporovaná hradla
-------------------------------------------------------
   syntaxe                  typ hradla
-------------------------------------------------------
   INPUT                    primární vstup
   OUTPUT                   primární výstup
   AND                      hradlo and
   NAND                     hradlo nand
   OR                       hradlo gate
   NOR                      hradlo nor
   XOR                      2 vstupové hradlo exclusive-or
   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)
Test Pattern Files
```

Formát PAT pro ukládaní sad testovacích vektorů (-W 1).

Obsahuje pouze hodnoty vstupů. i-tý bit test vektoru je hodnota i-tého bitu vstupu.

Příklad: Sada testovacích vekotrů pro C17 (získáno pomocí sat_atpg -t pattern.pat -W 1 c17.bench)

```
00101
10100
```

```
01110
01010
10011
01100
x0010
01111
11011
```

PAT Format s výstupy: -W 2. Více vektorů pro každou chybu: -W 3

```
Formát souborů s pouchami
------ EXAMPLE: A FAULT LIST FILE ----------------------
gate_A->gate_B /1
gate_A->gate_B /0
gate_A /1
gate_B /1
-------------------------------------------------------
```

Vtomto příkladu jsou gate_A a gate_B jména hradel. První řádek, "gate_A-¿gate_B /1" popisuje poruchu stuck-at 1 na vstupu hradla gate_B, kterým je spojeno s hradlem gate_A. Podobně, druhá řádka "gate_A-¿gate_B /0" popisuje stuck-at 0 poruchu na vstupu hradla gate_B input, kterým je spojeno s hradlem gate_A. Třetí a čtrvtý řádek popisují poruchy stuck-at 1 na výstupu hradel gate_A a gate_B.

Formát souboru se zprávou (definovaný pomocí parametru -P)

```
Inputs: 5
//počet primárních vstupů
Outputs: 2
//počet primárních výstupů
Gates: 6
//počet hradel
Faults: 22
//počet vygenerovaných nebo načtených poruch
Faults removed (dominance): 0
//počet poruch odstraněných pomocí dominance
Faults not covered: 0
//netestovatelné chyby (pravděpodobně redundantní)
Total vectors: 22
//Celkový počet vygenerovaných vektorů
Total vectors after compaction 22
//počet vektoů zbylých po kompakci.
Average dc ratio: 0.345455
// průměrný počet dc hodnot ve vektorech

Total time: 0.01 s
//celkový čas
Average count of clauses in CNF: 31.7273
//průměrný počet klauzulí v CNF
Maximum of clauses in CNF: 57
//Maximální počet klauzulí v CNF
Average count of variables in CNF: 15.0909
```

```
//Průměrnýc počet ptoměnných v CNF
Maximum of vars in CNF: 24
//Macimální počet proměnných v CNF
Average count of binary clauses in CNF: 15.8182
//průměrný počet klauzulí, které obsahují právě 2 literály.
Average clauses / vars ratio: 2.03425
//průměrný poměr počtu klauzulí a proměnných
Maximum of clauses / vars ratio: 2.375
//maximální poměr počtu klauzulí a proměnných
Minimum of clauses / vars ratio: 1.66667
//minimální poměr počtu klauzulí a proměnných
```

DIMACS formát Řádky na začátku souboru začínající znakem c jsou komentáře. Ty využívám pro informace o zakódování proměnných obvodu. A_10 je například aktivační proměnná hradla 10 a f_22 je proměnná reprezentující hodnotu hradla 22 v poruchovém obvodu. Řádek p cnf 13 27 znamená konjuktivní normální formu s 13 proměnnými a 27 klauzulemi.

```
c v 13
c 1 A\_10
c 2 1
c 3 A\_22
c 4 10
c 5 16
c 6 22
c 7 f\_10
c 8 f\_22
c 9 2
c 10 11
c 11 3
c 12 6
c 13 f\_1
p cnf 13 27
1 0
2 0
3 0
-4 -5 -6 0
4 6 0
5 6 0
-7 -5 -8 0
7 8 0
5 8 0
-3 6 8 0
-3 -6 -8 0
-9 -10 -5 0
9 5 0
10 5 0
-11 -12 -10 0
11 10 0
12 10 0
-2 -11 -4 0
2 4 0
```

```
11 4 0
-13 0
-1 3 0
-1 4 7 0
-1 -4 -7 0
-13 -11 -7 0
13 7 0
11 7 0
```

Dodatek C

Obsah přiloženého CD

```
CD struktura
|-- {bin}
|-- {exe}
|-- src
|   `-- sat_atpg
|       |-- lib
|       |   |-- minisat
|       |   `-- minisat_dc
|       `-- src
`-- text
    `-- latex-src
```