FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Komprese testu pro RAS architekturu založená na řešení SAT problému
Student:	Bc. Karel Pajskr
Vedoucí:	doc. Ing. Petr Fišer, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2020/21

Pokyny pro vypracování

Navrhněte a implementujte aplikaci pro generování komprimovaného testu (ATPG) pro číslicové obvody s „Random Access Scan“ (RAS) architekturou.
Využijte standardní metody softwarového inženýrství.
Systém nechť splňuje níže uvedené požadavky:
- Hlavní algoritmus řešící NP-těžký problém generování testu bude založený na implicitní reprezentaci testovacích vektorů, jako instance problému splnitelnosti booleovské formule (SAT).
- Při výpočtu použijte externí SAT řešič (MiniSAT), který bude integrován do aplikace jako knihovna.
- Pro optimalizaci délky testu modifikujte algoritmus pro použití pseudo-booleovské optimalizace (PBO) a řešič MinSAT+.
- Pro implementaci použijte jazyk C++.
- Použijte famework LogSynth pro manipulaci s logickými funkcemi.

Funkčnost aplikace otestujte simulací vytvořeného komprimovaného testu, proveďte experimentální vyhodnocení a porovnejte výsledky se stávajícími přístupy.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 18. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Komprese testu pro RAS architekturu založená na řešení SAT problému

Mgr. Karel Pajskr

Katedra softwarového inženýrství
Vedoucí práce: doc. Ing. Petr Fišer, Ph.D.

14. února 2020

Poděkování

Rád bych vyjádřil dík doc. Ing. Petru Fišerovi, Ph.D. za ochotu a trpělivost, se kterou tuto práci vedl, jakožto i všechny poskytnuté konzultace, rady a materiály.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 14. února 2020 . . . . . .

Odkaz na tuto práci

Pajskr, Karel. *Komprese testu pro RAS architekturu založená na řešení SAT problému.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Bakalářská práce se zabývá problematikou vytváření komprimovaného testu pro číslicové obvody s RAS architekturou. Práce se sestává z teoretického úvodu do problematiky, popisu a analýzy řešiče Minisat+, jeho integrace ve formě knihovny do frameworku LogSynth a tvorby programu pro generování testu, který je založen na novém originálním algoritmu.

Klíčová slova Random access scan, Minisat+, Pseudo-booleovská optimalizace

Abstract

This thesis deals with the problematics of test pattern generation for circuits with RAS architecture. The thesis consists of a theoretical introduction, description and analysis of Minisat+ solver, its integration into LogSynth framework as a library and a creation of a test pattern generator base on a novel algorithm.

Keywords Random access scan, Minisat+, Pseudo-boolean optimization

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Integrované obvody patří k technologiím, ve kterých za poslední desítky let došlo k neuvěřitelnému vývoji a dnes ovlivňují již téměř každou oblasti lidské činnosti. Tvoří jádra mobilů a počítačů, nalezneme je v průmyslu, dopravních prostředcích i ve spotřební elektronice. Dnešní čipy jako např. procesory či jádra grafických karet mívají miliardy tranzistorů. S nároky na technologii rostou i nároky na testování.

Ačkoliv se jedná řešení NP-úplného problému, představují SAT řešiče praktický, pohodlný a velice univerzální nástroj, který lze aplikovat i při návrhu integrovaných obvodů a jejich testování. V této práci se zabývám úpravou a použitím řešiče pro obvody s architekturou RAS, výsledkem je upravená knihovna a aplikace pro generování testů.

Kapitola 1

Cíl práce

Cílem bakalářské práce bylo navrhnout, implementovat a otestovat, příp. experimentálně vyhodnotit nový přístup k generování komprimovaného testu pro číslicové obvody s RAS architekturou založený na pseudo-booleovské optimalizaci vzdálenosti mezi testovacími vektory a odezvou. V rámci tohoto úkolu byl zapracován řešič Minisat+ jako knihovna do frameworku LogSynth.

Existující řešení

Komprese testu je kvůli svým výhodám obvyklý postup a existuje řada možných způsobů, jak kompresi provést, přičemž obvykle pracují se standardní architekturou scan-chain a předkládané řešení pro architekturu RAS je zcela nové a řadí ke kompresním metodám založených na podobnosti vektorů.

Co se použitých technologií týče, dostupnách SAT řešičů (včetně PBO) existuje několik[1], původní DPLL[2] algoritmus vznikl již v roce 1962, a většina novějších řešičů je na tomto algoritmu stále založena, přičemž modifikují heuristiku. Velký rozvoj nastal teprve po roce 2001 po publikaci článku [1] a řešiče Chaff na Princeton University.

Na téma aplikace pseudo-booleovské optimalizace na výpočet testovacích vektorů, existuje práce [2], která se zabývá optimalizací z hlediska maximalizace počtu neurčených bitů v testovacích vektorech pro RESPIN architekturu.

Analýza požadavků

Funkční požadavky

Funkční požadavky popisují nároky na funkce softwarového díla (co má program dělat). Ze zadání bakalářské práce a konzultaci s vedoucím jsem vyvodil následující požadavky:

[1] viz http://www.satlive.org/solvers/
[2] Davis–Putnam–Logemann–Loveland

- Implementace nového algoritmu ATPG pro RAS architekturu
- Program spustitelný z příkazové řádky s parametry a funkcemi obdobnými jako modul ATPG z frameworku

Nefunkční požadavky

Nefunkční požadavky popisují ostatní požadavky na systém (jak to má program dělat)

- Využití řešiče MiniSAT+ ve formě knihovny
- Implementace za použití a v rámci frameworku LogSynth, z čehož dále vyplývá
  - Použití jazyka C++
  - Možnost kompilace jak pod systémem Windows, tak i pod Linuxem

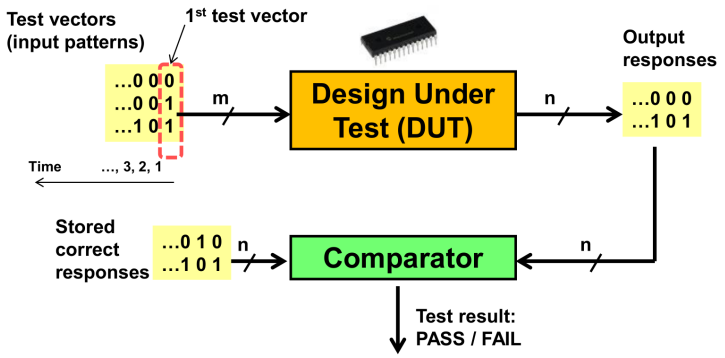
Kapitola 2

Testování integrovaných obvodů

2.1 Výroba integrovaných obvodů

Dle [3] a [4] se obvyklý postup vývoje integrovaných obvodů skládá z následujících činností:

1. Návrh
   - Funkce obvodu je popsána v jazyce typu VHDL či Verilog.
   - Proběhne verifikace, tj. testování správné funkce na logické úrovni a syntéza obvodu (převod do hradel)
2. Tvorba automatizovaného testu
   - Algoritmicky je připravena sada testovacích vstupů pro obvod (tzv. testovacích vektorů).
3. Výroba čipu
4. Testování
   - Čip je připojen a otestován pomocí ATE: na vstupy obvodu jsou posílány testovací vektory a odezva (výstup) obvodu je porovnávána s teoretickou (správnou) odezvou
5. Analýza
   - Výstup z testování je použit ke kontrole kvality (jde rozhodnout, zda daný čip je vadný či nikoliv) a též ke kontrole a optimalizaci předchozích kroků



Obrázek 2.1: Testování obvodu. Převzato z [3].

2.2 Motivace k testování, terminologie a stuck-at model

Na ose integrovaný obvod, deska (např. grafická karta, základní deska,...), zařízení (PC, ...) se celkem logicky ukazuje, že ekonomicky nejvýhodnější je provádět testování už na úrovni integrovaných obvodů, neboť testování o každou úrovneň výše je přibližně desetkrát dražší (tzv. pravidlo deseti).

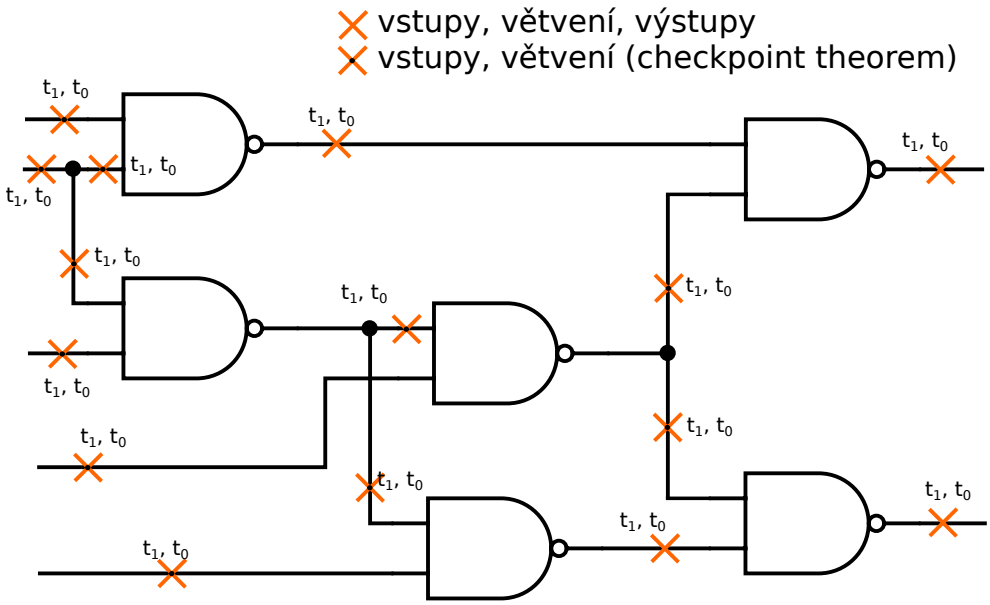
Při výrobě integrovaných obvodů dochází vinou různých vlivů jako např. nedokonalost výrobní technologie, nečistoty či fyzikální procesy k tvorbě defektů. Reprezentace defektu na úrovni hradel se nazývá porucha, přičemž častým typem defektů jsou přerušené či zkratované vedení signálu. K testování se používá tzv. stuck-at model, ve kterém jsou poruchami signály s trvalou hodnotou 1 (stuck-at-one) či 0 (stuck-at-0) na vstupech a výstupech hradel či na větveních. V daném modelu se nejčastěji též předpokládá současná přítomnost pouze jedné stuck-at poruchy v obvodu.

2.3 Jak testovat?

Mějme kombinační obvod skládající se z hradel a v něm stuck-at poruchu. Vstupy obvodu označme jako PI[3], výstupy jako PO[4]. Testování poruchy spočívá v nalezení takového ohodnocení PI, že dojde k

- excitaci poruchy = na místo se stuck-at-0 je přivedena 1 a naopak
- propagaci poruchy na PO = je ”zcitlivěna”cesta od místa poruchy k PO tak, že porucha projde až na výstup obvodu, tj. pro daný vstup se liší výstupy obvodu s poruchou a bez poruchy

[3]primary inputs
[4]primary outputs



Obrázek 2.2: Jaké poruchy je nutné testovat - příklad. Kompletní a redukovaná množina poruch pro úplný test daného obvodu. t_1 = stuck-at-1, t_0 = stuck-at-0. Adaptováno z [5].

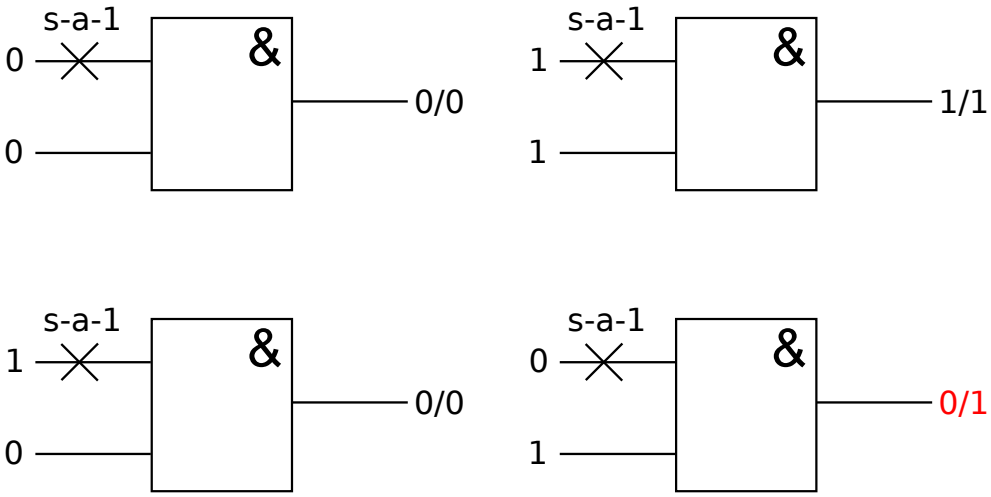
Smysl propagace je znázorňen na Obr. 2.3.

Daný problém tedy spočívá v nalezení testovacího vektoru pro danou poruchu a obecně v nalezení množiny testovacích vektorů, jejichž aplikace na obvod otestují co nejvíce možných poruch (je dosaženo požadované pokrytí). Hledat testovací vektor pro zadanou poruchu je možné pomocí několika různých algoritmů, přičemž nás dále bude zajímat především algoritmus založený na hledání splnitelnosti booleovské formule, tzv. SAT z anglického satisfiability problem.

2.4 Hledání testovacích vektorů pomocí SAT

SAT aneb Boolean satisfiability problem

Mějme zadaný booleovský výraz (formuli) tvořený proměnnými a logickými operacemi (např. or, and, negace). Problém splnitelnosti booleovské formule označuje hledání takového ohodnocení proměnných, že výsledný výraz je pravdivý. SAT je NP-úplný problém, nicméně existují algoritmy, které ho řeší poměrně efektivně, přičemž většina dnešních SAT solverů (řešičů) ho řeší v tzv. CNF-SAT podobě, kdy tyto algoritmy pracují s výrazy v konjunktivní normální formě. Přejděme k definicím z [6]:



Obrázek 2.3: Excitace a propagace poruchy v případě jednoduchého hradla. Značení výstupu je: bez-poruchy/s-poruchou. S-a-1 = stuck-at-1, tj. tento signál je v bodě poruchy trvale připojen na 1. Adaptováno z [5].

Problém SAT Logická formule je v **konjunktivní normální formě**, pokud se jedná o konjunkci disjunkcí literálů. Literálem se myslí x či negace $\neg x$, kde x je booleovská proměnná. Jednotlivé disjunkce se označují jako klauzule. Problém splnitelnosti booleovské formule či SAT spočívá v nalezení takového ohodnocení proměnných, že celkový výrok je pravdivý, což je ekvivalentní tomu, že při takovém ohodnocení se v každé klauzuli nachází alespoň jeden pravdivý literál.

Pseudo-booleovská optimalizace Pseudo-booleovská podmínka je nerovnost typu $C_0p_0 + C_1p_1 + \ldots + C_{n-1}p_{n-1} \geq C_n$, kde pro všechna i je p_i literál a C_i celočíselný koeficient, přičemž pravdivý literál je vyhodnocen jako 1 a nepravdivý jako 0. Výraz z levé strany rovnice, tj. lineární kombinace literálů má význam funkce. PB optimalizace spočívá v nalezení ohodnocení, pří kterém je zadaná množina PB podmínek pravdivá a pro které je hodnota zadané (lineární) minimalizační funkce minimální. *Poznámka:* Pro $C_i = 1$ pro všechna i je PB podmínka ekvivalentní klauzuli v CNF.

Konjunktivní normální forma

Každou booleovskou formuli lze převést do konjunktivní normální formy, což si zde předvedeme na výrazu A $\wedge$ (B $\Leftrightarrow$ C) Postup transformace výroku do CNF (či KNF):

1. Sestavím si tabulku pravdivostních hodnot.

A	B	C	A ∧ (B ⇔ C)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Tabulka 2.1: Pravdivostní tabulka pro formuli A ∧ (B ⇔ C).

2. Pro řádky, ve kterých je formule nepravdivá sestavím disjunkci negací původních elementárních proměnných.

3. Sestavím konjunkci těchto disjunkcí.

Z tabulky pro výraz A ∧ (B ⇔ C) (viz tab. 2.1) vidíme, že platí

$$\begin{aligned} \mathrm{A} \wedge (\mathrm{B} \Leftrightarrow \mathrm{C}) = {} & (\mathrm{A} \vee \mathrm{B} \vee \mathrm{C}) \wedge (\neg\mathrm{A} \vee \mathrm{B} \vee \neg\mathrm{C}) \wedge \\ & (\mathrm{A} \vee \neg\mathrm{B} \vee \mathrm{C}) \wedge (\mathrm{A} \vee \neg\mathrm{B} \vee \neg\mathrm{C}) \wedge \\ & (\neg\mathrm{A} \vee \mathrm{B} \vee \neg\mathrm{C}) \wedge (\neg\mathrm{A} \vee \neg\mathrm{B} \vee \mathrm{C}) \end{aligned}$$

Tseitinova transformace

Pro převod kombinačních obvodů do CNF se používá Tseitinova[5] transformace. Hradla v obvodu jsou popsána charakteristickými funkcemi v CNF a jejich vzájemným vynásobením je získána CNF pro celý obvod. Transformace je lineární s počtem hradel, avšak zvyšuje počet použitých proměnných. Příkladem charakteristické funkce v konjunktivní normální formě je funkce $\chi_F = (F' + x).(F' + y).(F + x' + y')$ pro hradlo $F = \mathrm{AND}(x, y)$. Charakteristická funkce popisuje správnou funkci hradla, tj. zde platí $\chi_F = 1$ pro ohodnocení $\{F, x, y\} = \{1, 1, 1\}$, $\{0, 1, 0\}$, $\{0, 0, 1\}$ a $\{0, 0, 0\}$.

Generování testovacích vektorů pomocí SAT

Hledání testovacího vektoru pro zadanou poruchu lze jednoduše zformulovat jako problém pro SAT následujícím postupem:

1. Vezmu bezporuchový obvod

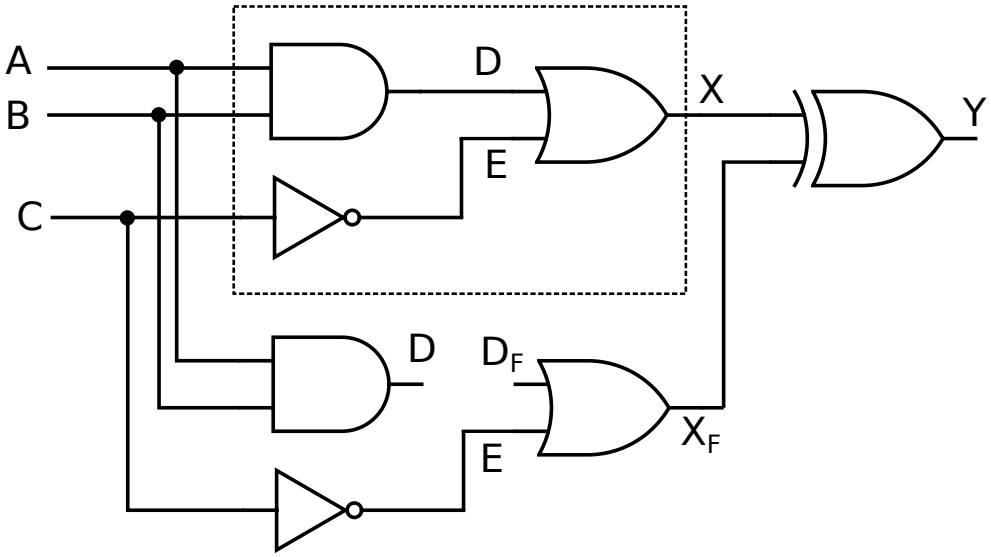
2. Vezmu obvod s poruchou

3. odpovídající PO obvodů spojím XOR hradlem

[5] V originále Цейтин Г. С. (autor)

4. výstupy XOR hradel spojím do OR hradla
    - výstup bude 1, liší li se alespoň jeden výstup poruchového a bezporuchového obvodu, tj. pokud se porucha propaguje na PO
5. Celý obvod popíšu pomocí charakteristických funkcí všech hradel v CNF
6. Tyto CNF spojím do jedné formule, kterou předám SAT solveru
7. Pokud je CNF splnitelná, řešením SAT problému je testovací vektor pro danou poruchu

Postup je pro jednoduchý obvod znázorněn na obr. 2.4, výraz 2.1 pak představuje výsledné CNF.

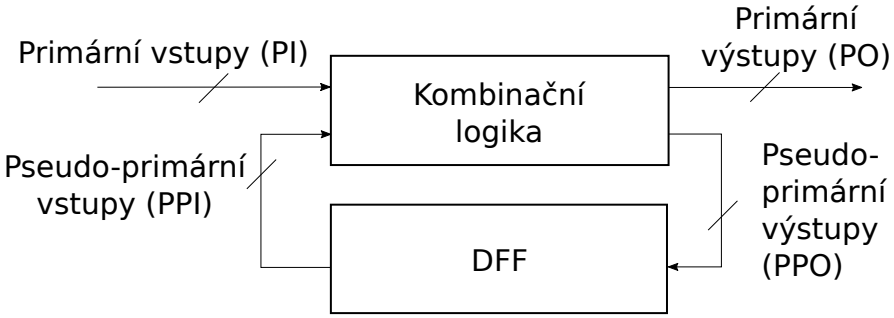


Obrázek 2.4: Obvod pro výpočet SAT (Pro více výstupů obvodu by byly výstupy z XOR hradel spojeny do OR hradla). Bezporuchový obvod (v rámečku) je spojen přes XOR hradlo s obvodem s poruchou. Platí Y = 1 pokud je porucha detekována. Adaptováno z [7].

$$
\begin{aligned}
&(\mathrm{D'} + \mathrm{A}) \cdot (\mathrm{D'} + \mathrm{B}) \cdot (\mathrm{D} + \mathrm{A'} + \mathrm{B'}) \cdot \\
&(\mathrm{C} + \mathrm{E}) \cdot (\mathrm{C'} + \mathrm{E'}) \cdot \\
&(\mathrm{X} + \mathrm{D'}) \cdot (\mathrm{X} + \mathrm{E'}) \cdot (\mathrm{X'} + \mathrm{D} + \mathrm{E}) \cdot \\
&(\mathrm{D_F}) \cdot \\
&(\mathrm{X_F} + \mathrm{D_F'}) \cdot (\mathrm{X_F} + \mathrm{E'}) \cdot (\mathrm{X'} + \mathrm{D_F} + \mathrm{E}) \cdot \\
&(\mathrm{C} + \mathrm{E}) \cdot (\mathrm{C'} + \mathrm{E'}) \cdot \\
&(\mathrm{X'} + \mathrm{X_F} + \mathrm{Y}) \cdot (\mathrm{X} + \mathrm{X_F'} + \mathrm{Y}) \cdot (\mathrm{X} + \mathrm{X_F} + \mathrm{Y'}) \cdot (\mathrm{X'} + \mathrm{X_F'} + \mathrm{Y'}) \cdot \mathrm{Y}
\end{aligned}
\tag{2.1}
$$

2.5 Testování sekvenčních obvodů

Testování kombinačních obvodů sestává z aplikace testovacích vektorů, které excitují a propagují danou poruchu. V případě sekvenčních obvodů je situace složitější, neboť obecně stav a chování obvodu závisí na jeho vnitřním stavu, což problém hledání testovacího vektoru značně zesložiťuje ([8]). Možným a používaným řešením je převedení daného sekvenčního obvodu na kombinační, resp. využití jeho rozdělení na kombinační a DFF část a následné testování této kombinační části, přičemž množina vstupů (výstupů) obvodu je nyní tvořena jak vstupy (výstupy) původního obvodu, označovanými jako primární vstupy (výstupy), tak i vstupy (výstupy) z klopných obvodů, tzv. pseudoprimárních vstupů (výstupů), jak je znázorněno na obr. 2.5. Tímto postupem se problém



Obrázek 2.5: Model sekvenčního obvodu. Převzato z [9].

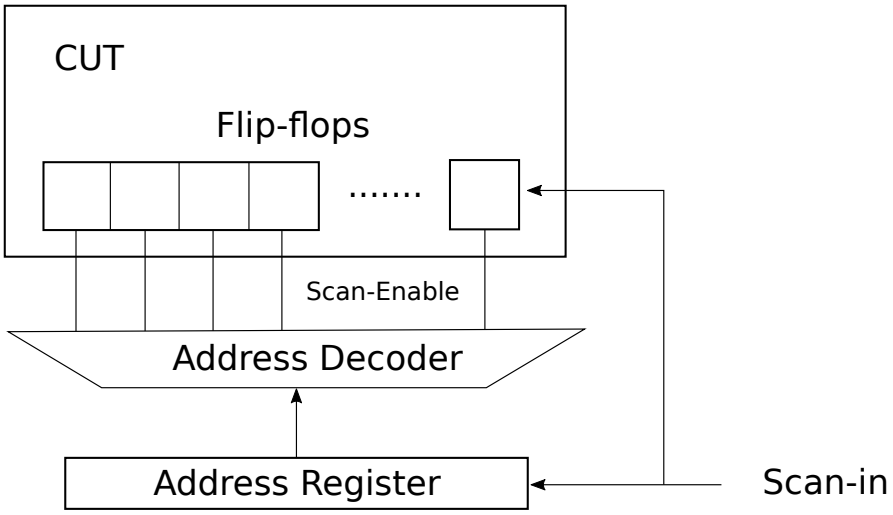
převedl na testování kombinačního obvodu, které je jednodušší ale zároveň i na problém, jak dostat do DFF data pro PPI.

2.6 Architektura RAS a komprese testu

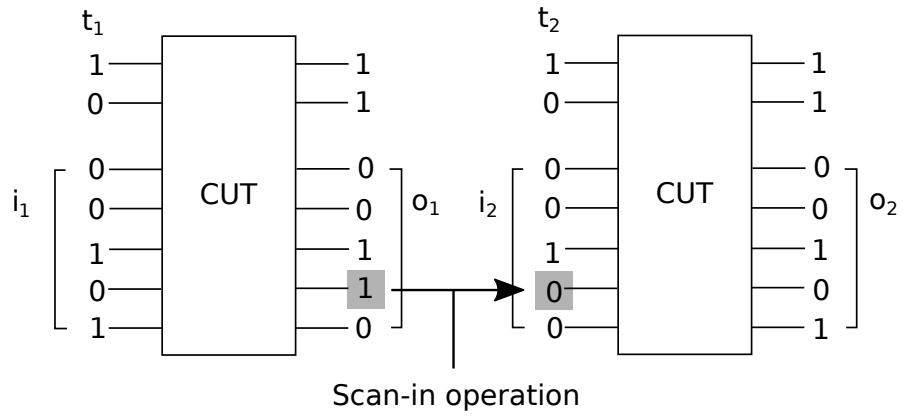
Architektura Random Access Scan (RAS) řeší tento problém jednoduše tím, že jednotlivé buňky paměti (klopné obvody) jsou přímo adresovatelné. Pokud má obvod n klopných obvodů, pak adresa má velikost $\log_2 n$ bitů a na aplikaci jednoho testovacího vektoru obecně připadá $n.\log_2 n$ bitů. RAS architektura avšak umožňuje měnit pouze potřebné bity. Po aplikaci funkčních hodin se totiž v klopných obvodech nachází odezva na předchozí testovací vektor [10] a minimalizací Hammingovy vzdálenosti mezi odezvou a následujícím testovacím vektorem tudíž dosahnu komprese testu, tj. zmenšení objemu zasílaných dat (či zkrácení doby testování). Obvykle používaný algoritmus komprese pro RAS obvody se skládá z

1. Vygenerování testovacích vektorů
2. Změny pořadí testovacích vektorů tak, aby se minimalizovala Hammingova vzdálenost pro celý test

Tento problém je ekvivalentní s problémem obchodního cestujícího v jeho asymetrické variantě. Architektura RAS je znázorněna na obr. 2.6 a princip komprese pro RAS obvod, tj. nutnost nasunutí pouze změněných bitů v PPI, na obr. 2.7.



Obrázek 2.6: Architektura RAS. Tato architektura umožňuje přímý zápis do klopných obvodů pomocí $\log_2 n$ adresních bitů, kde n je počet klopných obvodů. Převzato z [11].



Obrázek 2.7: Architektura RAS. Znázornění možnosti měnit jen některé bity. Převzato z [11].

Kapitola 3

Navrhovaný algoritmus

Komprese testu pro RAS pomocí pseudo-booleovské optimalizace

V této práci jsme se pokusili o jiný přístup, a to o hledání testovacích vektorů minimalizujících Hammingovu vzdálenost již při řešení SAT. Algoritmus spočívá v tom, že se při hledání testovacích vektorů pomocí SAT přidá PBO podmínka s cílem minimalizovat bitovou (Hammingovu) vzdálenost od předchozí odezvy pro PPI a od předchozího testovacího vektoru u PI. Minimum se hledá přes všechny zbývající poruchy. Níže je algoritmus popsán v pseudokódu:

```
Vygeneruj seznam poruch;
Vyber poruchu, vygeneruj testovací vektor a poruchy nalezené tímto
  vektorem odeber ze seznamu;
while seznam poruch je neprázdný do
    foreach porucha v seznamu do
        vygeneruj testovací vektor s nejmenší vzdáleností pro RAS
          pomocí SAT s PBO
    end
    vyber testovací vektor s nejmenší vzdáleností pro přes všechny
      zbývající poruchy;
    poruchy nalezené tímto vektorem odeber ze seznamu;
end
```

Minimalizace vzdálenosti pomocí PBO

PBO lze využít k minimalizaci vzdálenosti vektorů hledaných pomocí SAT. Přepis CNF klauzulí do PB-podmínek je přímočarý; Podmínku minimalizující vzdálenost hledaného vektoru lze zadat zvolením koeficientů v minimalizační funkci: pokud chceme, aby hledaný literál p_i nabýval hodnoty 1, volíme koe-

ficient $C_i = -1$ v objektivní funkci a podobně pokud chceme, aby p_i nabýval hodnoty 0, za koeficient volíme $C_i = -1$. Nalezený literál, jehož hodnota je odlišná od cílové hodnoty zvyšuje hodnotu objektivní funkce, přičemž SAT solver hledá minimum, tj. v důsledku co nejméně vzdálený vektor. Příklad je zobrazen v tab. 3.1. Zde hledáme vektor $x_0x_1x_2x_3x_4$ tak, aby byl co nejméně vzdálený vektoru 10110.

literál	x_0	x_1	x_2	x_3	x_4
cílová hodnota	1	0	1	1	0
funkce pro minimalizaci solverem	$-x_0 + x_1 - x_2 - x_3 + x_4$				

Tabulka 3.1: Příklad „goal“ funkce minimalizující vzdálenost pro vektor hledaný SAT. Hodnota objektivní funkce je zde rovna mínus třem plus počtu změněných bitů.

Kapitola 4

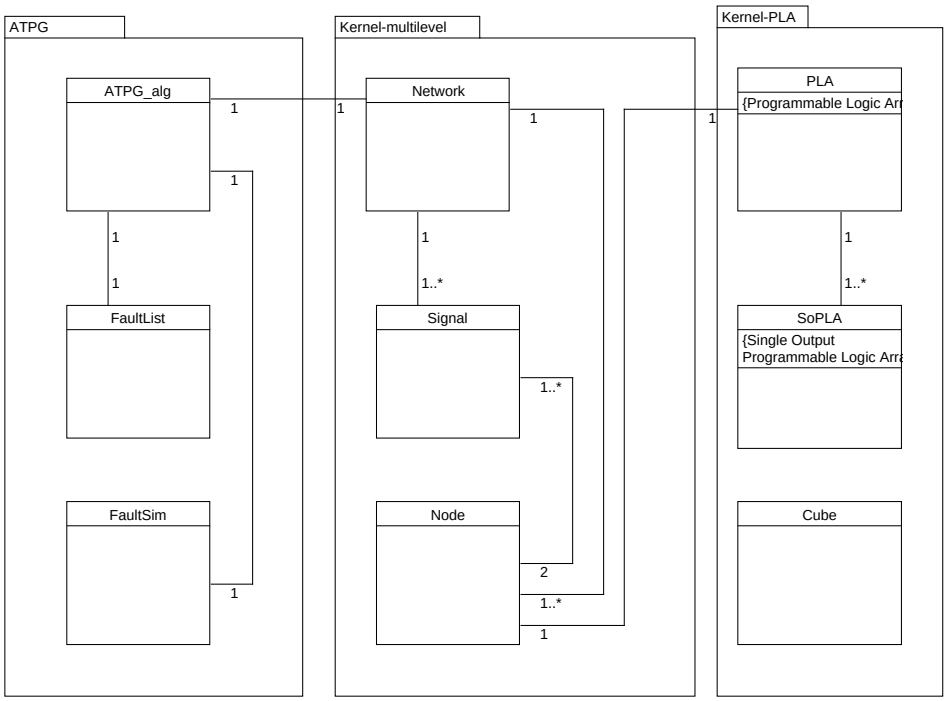
Analýza

4.1 LogSynth

LogSynth je framework pro manipulaci s logickými funkcemi vyvíjený na Katedře číslicového návrhu FIT ČVUT. Jádrem frameworku jsou moduly Kernel-PLA a Kernel-multilevel, které definují datové typy a třídy umožňující načtení a práci s obvody tvořenými sítí vzájemně propojených hradel. Další z mnoha modulů, modul ATPG, je určen pro hledání testovacích vektorů pomocí SAT a obsahuje funkcionalitu nutnou pro generování množiny poruch pro daný obvod, CNF pro danou poruchu v obvodu a pro simulaci testovacího vektoru a následné určené poruch detekovaných tímto vektorem. Základní rozhraní pro práci s logickými obvody je v použitém frameworku LogSynth tvořeno třídami `Network` a `ATPG_alg`. Třída `Network` reprezentuje celý logický obvod tvořený hradly (popsané objekty třídy `Node`), které jsou vzájemně propojené „dráty“ (třída `Signal`). Hradlo (`Node`) je popsáno objektem třídy `PLA`, což je tabulka popisující výstupy hradla pro dané vstupy. Třída `Cube` pak reprezentuje bitový vektor.

Zde je popis použitých metod z Kernel-multilevel:

- `void Network::LoadBlif(string filename)`,
  `void Network::LoadBench(string filename)` – načte vstupní soubor (logický obvod) ve formátu .blif či .bench do objektu
- `int Network::getDFFs()`,
  `int Network::inputs()`,
  `int Network::outputs()` – vrátí počet DFF, PI, PO v načteném obvodu
- `SoPLA* Network::ToCNF(void) const` – Zkonvertuje daný obvod do CNF (pro předání do řešiče)
- `vector<Literal> Network::Simulate(const Cube &t, Fault *flt, bool all)` – simulace testovacího vektoru v daném obvodu



Obrázek 4.1: Zjednodušený diagram tříd modulů ATPG, Kernel-multilevel a Kernel-PLA frameworku LogSynth.

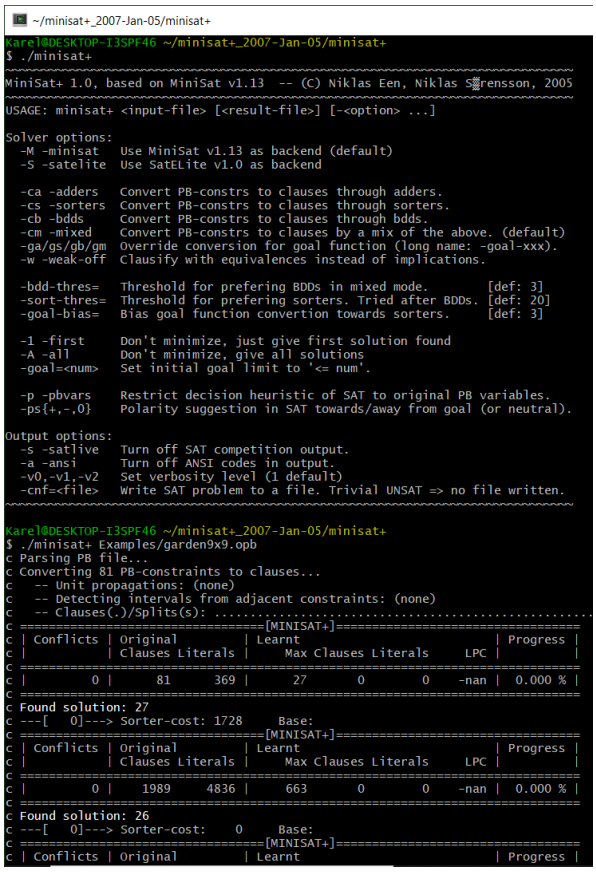
Modul ATPG frameworku obsahuje funkcionalitu pro generátor ATPG, koncentrující se převážně ve třídě `ATPG_alg`. Použité metody:

- `ATPG_alg::ATPG_alg (Network& network)` – vygeneruje instanci ATPG pro daný obvod
- `FaultList& ATPG_alg::FaultListRef()` reprezentuje seznam poruch pro daný obvod
- `Network *ATPG_alg::GenerateMiter(SA_Fault *flt)` vygeneruje obvod pro testování pomocí SAT (viz. 2.4)
- `void ATPG_alg::Drop(list<TestPattern>::iterator p_it)` detekuje poruchy nalezené daným testovacím vektorem a označí je v seznamu poruch

Framework je v práci využit se souhlasem autora/autorů.

4.2 Minisat+

Minisat+ je SAT-PBO řešič od autorů Niklase Eéna a Niklase Sörenssona. Jedná se o pozměněný řešič MiniSat od stejných autorů, respektive o nástavbu, převádějící PB podmínky na problém SAT a využívající Minisat jako backend. Program se zdrojovými kódy je dostupný volně ke stažení na webových stránkách autorů[6]. Webová stránka ani kód samotný (stažený z těchto stránek) neobsahuje licenční ujednání, nicméně jeden z autorů kód umístil i na GitHub[7], kde jako licenční ujednání uvedl licenci MIT.



Obrázek 4.2: Minisat+ spuštěný v prostředí Cygwin pod Windows. Spuštění a) bez parametru, b) se souborem garden9x9.opb z adresáře Examples.

Zdrojové kódy jsou určené pro kompilátor gcc/g++ pod linuxem a součástí archivu s programem je i spustitelný binární soubor pro linux a několik ukázkových souborů. Na obr. 4.2 je běh programu v prostředí Cygwin a na 4.3 je ukázka zdrojového formátu, tj. zadání PBO problému.

[6] http://minisat.se/MiniSat+.html
[7] https://github.com/niklasso/minisatp

```
* #variable= 81 #constraint= 81
* converted from file: submitted/sorensson/garden/g9x9.opb
min: +1*x1 +1*x2 +1*x3 +1*x4 +1*x5 +1*x6 +1*x7 +1*x8 +1*x9 +1*x10 +1*x11 +
1*x12 +1*x13 +1*x14 +1*x15 +1*x16 +1*x17 +1*x18 +1*x19 +1*x20 +1*x21 +1*x22
+1*x23 +1*x24 +1*x25 +1*x26 +1*x27 +1*x28 +1*x29 +1*x30 +1*x31 +1*x32 +1*
x33 +1*x34 +1*x35 +1*x36 +1*x37 +1*x38 +1*x39 +1*x40 +1*x41 +1*x42 +1*x43 +
1*x44 +1*x45 +1*x46 +1*x47 +1*x48 +1*x49 +1*x50 +1*x51 +1*x52 +1*x53 +1*x54
+1*x55 +1*x56 +1*x57 +1*x58 +1*x59 +1*x60 +1*x61 +1*x62 +1*x63 +1*x64 +1*
x65 +1*x66 +1*x67 +1*x68 +1*x69 +1*x70 +1*x71 +1*x72 +1*x73 +1*x74 +1*x75 +
1*x76 +1*x77 +1*x78 +1*x79 +1*x80 +1*x81 ;
+1*x1 +1*x2 +1*x10 >= +1;
+1*x10 +1*x11 +1*x1 +1*x19 >= +1;
+1*x19 +1*x20 +1*x10 +1*x28 >= +1;
+1*x28 +1*x29 +1*x19 +1*x37 >= +1;
+1*x37 +1*x38 +1*x28 +1*x46 >= +1;
+1*x46 +1*x47 +1*x37 +1*x55 >= +1;
+1*x55 +1*x56 +1*x46 +1*x64 >= +1;
+1*x64 +1*x65 +1*x55 +1*x73 >= +1;
+1*x73 +1*x74 +1*x64 >= +1;
+1*x2 +1*x1 +1*x3 +1*x11 >= +1;
+1*x11 +1*x10 +1*x12 +1*x2 +1*x20 >= +1;
+1*x20 +1*x19 +1*x21 +1*x11 +1*x29 >= +1;
+1*x29 +1*x28 +1*x30 +1*x20 +1*x38 >= +1;
+1*x38 +1*x37 +1*x39 +1*x29 +1*x47 >= +1;
+1*x47 +1*x46 +1*x48 +1*x38 +1*x56 >= +1;
```

Obrázek 4.3: Formát zdrojového souboru pro Minisat+ na příkladu garden9x9.opb z adresáře Examples. Je zde vidět funkce pro minimalizaci (objektivní funkce) a CNF klauzule. (Soubor na obrázku není celý.)

Minisat+ se skládá z části převádějící PB podmínky do CNF a z SAT řešiče Minisat. Pro převod PB podmínek do CNF se využívá lineární převoditelnost logických obvodů do CNF, přičemž PB podmínku lze do logického obvod u převést pomocí ([6]):

- BDD (Binary decision diagramů)
- sčítaček
- třídících sítí

Minimalizace objektivní funkce je realizována iterativním voláním řešiče - v první iteraci je vyhodnocena její hodnota $f(p_i) = k$ a v následném volání je přidána PB podmínka $f(p_i) < k$. Pokud řešič vrátí nesplnitelnost, je k minimum, v opačném případě je řešič volán znova s podmínkou se zmenšenou hodnotou k. Jako řešič SAT je použit Minisat, jehož algoritmus je popsán v článku [12].

Zdrojový kód Minisat+ bohužel nemá k disposici dokumentaci a kód obsahuje poměrně málo komentářů. Z kódu programu ve funkci main však bylo

možné vytvořit si představu o fungování tříd PbSolver, Solver a MiniSat. V tab. 4.1 je sestaven popis nejdůležitějších programových modulů.

Z analýzy dále vyplynulo, že samotný řešič, reprezentovaný objektem třídy PbSolver je ovládán metodami a proměnnými:

- `void PbSolver::allocConstrs(int n_vars, int n_constrs)` – alokuje datové struktury pro řešení problému o daném počtu proměnných a PB podmínek.
- `int PbSolver::getVar(const char *name)` - alokuje v solveru booleovskou proměnnou s daným indexem
- `bool PbSolver::addConstr (const vec<Lit>& ps, const vec<Int>& Cs, Int rhs, int ineq)` - přidá do solveru danou PB podmínku
- `void PbSolver::addGoal(const vec<Lit>& ps, const vec<Int>& Cs)` - přidá do solveru danou objektivní funkci pro minimalizaci
- `int PbSolver::solve(solve_Command cmd = sc_Minimize)` - řeší zadaný problém
- `Int PbSolver::best_goalvalue` - Int_MAX v případě UNSAT či hodnota objektivní funkce pro nalezené (minimální) řešení
- `vec<bool> PbSolver::best_model` - nalezené nejlepší řešení problému

Heuristika solveru (např. výběr metody převodu zadání do CNF) je ovládána pomocí statických proměnných a solver též využívá několik statických datových proměnných pro urychlení výpočtů v programovém modulu Hardware_clausify.

Programový modul	Popis
VecAlloc, StackAlloc	alokátory paměti
Map, Heap	definice datových typů
VecMaps	optimalizovaný vektor bool hodnot; VecMap: vektor; DeckMap: vektor s ± indexy
Int, SolverTypes, Global	definice datových typů:číslo, booleovská proměnná, literál,lbool (true, false, undef, error)
FEnv	reprezentace logických výrazů a operací
Solver	interface pro backend SAT solver
MiniSat, SatELite	třída backend SAT solveru
PbSolver	definice třídy Linear (uložení PB podmínky); třída PbSolver (PBO solver)
Hardware a Hardware_X PbSolver_convert a PbSolver_convertY	funkce pro převod PB podmínek do CNF; X=adders, clausify,sorters; Y=Add, Bdd, Sort

Tabulka 4.1: Popis programových modulů ze zdrojového kódu Minisat+.

Kapitola 5

Realizace

V této kapitole popisuji implementaci algoritmu z kapitoly 3 využívající framework LogSynth a knihovnu Minisat+. Minisat+ (původně aplikace) bylo nutné převést do formy knihovny a vytvořit rozhraní pro zadávání a řešení PB problémů. Toto rozhraní je posléze využito z C++ aplikace pro příkazový řádek a napojeno na jádro frameworku Logsynth, které poskytuje funkcionalitu pro práci s logickými obvody. Ovládání programu je řešeno parametry („přepínači“), zadávanými z příkazové řádky, jejichž struktura je převzata z modulu ATPG frameworku.

5.1 Volba technologie

V návaznosti na fakt, že Minisat+ i LogSynth jsou napsané v C++ bylo ve vývoji pokračováno též v jazyce C++. Využito bylo vývojové prostředí Visual Studio.

5.2 Převod Minisat+ na knihovnu

Původní kód Minisat+ (program) bylo poměrně jednoduché importovat do Visual Studia a upravit na statickou knihovnu, neboť nekompatibilita s kompilátorem ve Windows byla způsobena převážně POSIX voláními ve funkci `main` programu. Po odstranění funkce `main` zbyly převážně třídy pracující s daty, avšak výsledné funkce přišly o možnost přerušovat (dlouhé) výpočty pomocí Ctrl-C s vypsáním dosud nejlepšího nalezeného řešení.

Změny v kódu a interface

Analýzou zdrojového kódu Minisat+ bylo zjištěno, že hlavní částí řešiče je třída `PbSolver`. Objekt této třídy slouží k definici zadání daného PB problému a hledání řešení. Dále však bylo zjištěno, že chod PbSolveru závisí na řadě globálních proměnných. Tyto bylo možné rozdělit na dvě kategorie. První

Globální proměnná	Popis	1	2	3
opt_satlive	změna tvaru výpisů pro SAT competition	ne	ne	true
opt_ansi	ANSI kódy ve výpisech	ne	ne	true
opt_cnf	soubor pro výpis v CNF	ano	ne	NULL
opt_verbosity	nastavení množství výpisů	ano	ne	1
opt_try	nastavení parsování vstupů	ne	ne	false
opt_solver	volba backend solveru	ano	ne	st_MiniSat
opt_convert	způsob převodu PB podmínek do CNF	ano	ano	ct_Mixed
opt_convert_goal	způsob převodu goal funkce do CNF	ano	ne	ct_Undef
opt_convert_weak	ekvivalence místo implikací při převodu klauzulí	ano	ne	true
opt_bdd_thres	práh preference BDD v mixed módu	ano	ne	3.0
opt_sort_thres	práh preference třídících sítí v mixed módu	ano	ano	20.0
opt_goal_bias	násobící konstanta pro opt_sort_thres	ano	ne	3.0
opt_goal	počáteční hodnota (limit) pro goal funkci	ano	ne	Int_MAX
opt_command	nastavení módu řešení	ne	ne	cmd_Minimize
opt_branch_pbvars	omezení heuristiky na původní proměnné	ano	ne	false
opt_polarity_sug	nastavení polarity SAT vůči goal funkci	ano	ne	1
opt_input	jméno vstupního souboru	ne	ne	NULL
opt_result	jméno výstupního souboru	ne	ne	NULL

Tabulka 5.1: Globální proměnné opt_XXXXX v Minisat+. 1 = PbSolver závisí na proměnné, 2 = PbSolver může měnit, 3 = Výchozí nastavení.

část jsou proměnné s předponou `opt_`, které v původním programu sloužily pro předávání speciálních parametrů do řešiče. Tyto proměnné jsou zmíněny v tab. 5.1. Druhou část posléze byly globální proměnné sloužící jako sdílené datové struktury mezi částmi řešiče. Využití globálních proměnných zřejmě bylo zvoleno kvůli optimalizaci rychlosti. Zde se jedná statické proměnné třídy `Clausifier`, tj. o

```
CMap<int> Clausifier::occ
CMap<Var> Clausifier::vmap
CMap<Lit,true> Clausifier::vmapp
```

a též o globální proměnné z namespace FEnv:

```
vec<NodeData> FEnv::nodes
Map<NodeData, int> FEnv::uniqueness_table
vec<int> FEnv::stack
```

Jelikož cílem bylo použít Minisat+ jako knihovnu, bylo nutné při návrhu rozhraní zajistit, aby se pro opakování volání řešiče datové globální proměnné

vymazaly. Dalším specifikem bylo, že řešič v původní formě programu nalezená řešení pouze vypisoval na obrazovku a tedy nové rozhraní muselo umožnit předávání nalezených řešení a nikoliv jenom výpis, přičemž řešič PbSolver pracuje ve třech možných módech: minimalizace zadané funkce, hledání prvního řešení (bez minimalizace), hledání všech řešení (bez minimalizace).

Návrh jsem tedy volil takový, aby

- rozhraní odstínilo komplikované vzájemné závislosti programových modulů Minisat+
- celé rozhraní řešiče bylo objektové, tj. bez nutnosti práce s globálními parametry
- se zjednodušila definice PB podmínek, zejména definice literálů
- použití řešiče nemění globální parametry a automaticky resetuje globální datové struktury
- řešič podporoval všechny módy, tj. byl schopný vracet 0, 1 i více řešení

V následující části popisuji navržené a využité třídy.

PBLit

Třída reprezentující literál vynásobený celočíselnou konstantou.

V zadání PB podmínek a minimalizační funkce se vyskytuj výrazy typu $k \cdot \mathrm{p}\,(')$, kde k je celočíselná konstanta a p booleovská proměnná (s případnou negací). Původní složitou registraci jména literálu u objektu třídy `PbSolver` jsem nahradil třídou `PBLit`, jejíž objekty reprezentují celý tento výraz. Popis atributů/metod níže.

Název atributu	Datový typ	Popis
`c`	int	konstanta násobící literál v PB podmínce
`var`	size_t	číslo PB proměnné
`neg`	bool	true pokud je proměnná v negaci

Params

Touto třídou jsem zakryl původní přístup k globálním parametrům Minisat+. Nepotřebné parametry (tj. „ne“ v obou sloupcích v tab. 5.1) jsem mohl zcela vynechat.

Popis atributů/metod níže.

Název metody	Návratový typ	Popis
`setDefaults`	`void`	statická metoda
`setOptVerbosity`, `getOptVerbosity`, ...	`void`	statické metody

Helper

Třída reprezentující řešení PB problému. Třída využívá návrhový vzor Observer - objekt této třídy se zaregistruje u objektu třídy Solver a pokud je v rámci metody `solve` nalezeno řešení, je zavolána metoda solution, přičemž v atributu `v` se nachází toto řešení. Dle parametrů volání Solveru je možné, že je solution volána 0, 1 i vícekrát. Sekvenční diagram viz obr 5.1.

Název atributu	Datový typ	Popis
`values`	`vector<bool>`	vektor s řešením
Název metody	Návratový typ	Popis
`solution`	`void`	metoda volána po nalezení řešení

Solver

Třída reprezentující řešič/zadání PB problému a tvořící adaptér nad originální třídou PbSolver z Minisat+. Třída využívá návrhový vzor Singleton.

Voláním statické metody `getInstance` je vrácena reference na singleton objekt. Voláním metody `init` je specifikován počet proměnných a počet PB podmínek. Následně je možné pracovat s objekty typu `PBLit` s atributem `var` v rozsahu 1 až n, kde n je počet proměnných předaných konstruktoru Solveru. Levé strany PB podmínek a funkce pro minimalizaci jsou pak určeny jako `vector<PBLit>`. Jako první je nutné přidat minimalizační funkci pomocí metody `addGoal` a posléze stejný počet PB podmínek, jako byl určen v konstruktoru. Dalšími parametry při přidání PB podmínky je typ znaménka s možnostmi `INEQ_LT` ($<$), `INEQ_LE`($\leq$), `INEQ_EQ`($=$), `INEQ_GE`($\geq$), `INEQ_GT`($>$) a celočíselná konstanta na pravé straně podmínky. Metoda `solve` provede hledání řešení. Jejím parametrem je typ řešení, které se má hledat s možnostmi `FIND_ALL` (hledá všechna řešení bez ohledu na objektivní funkci), FIND_FIRST (hledá první řešení bez ohledu na minimalizační funkci), `FIND_MINIMIZE` (hledá řešení minimalizující goal funkci). Ukázkový kód se nachází ve výpisu 5.1.

Název metody	Návratový typ	Popis
`getInstance`	`Solver&`	vrací referenci na singleton objekt (statická metoda)
`init`	`void`	nastavuje solver na daný počet proměnných a podmínek a provádí reset parametry: `int n_vars, n_constrs`
`addGoal`	`void`	nastaví funkce pro minimalizaci parametry: `vector<PBLit> &v`

`addConstr`	`void`	přidá PB podmínku
		parametry: `vector<PBLit> &v`
		`Solver::Inequality eq`
		`int val`

`solve`	`void`	hledání řešení parametry: `Solver::Command cmd`

Listing 5.1: Solver - ukázka kódu.

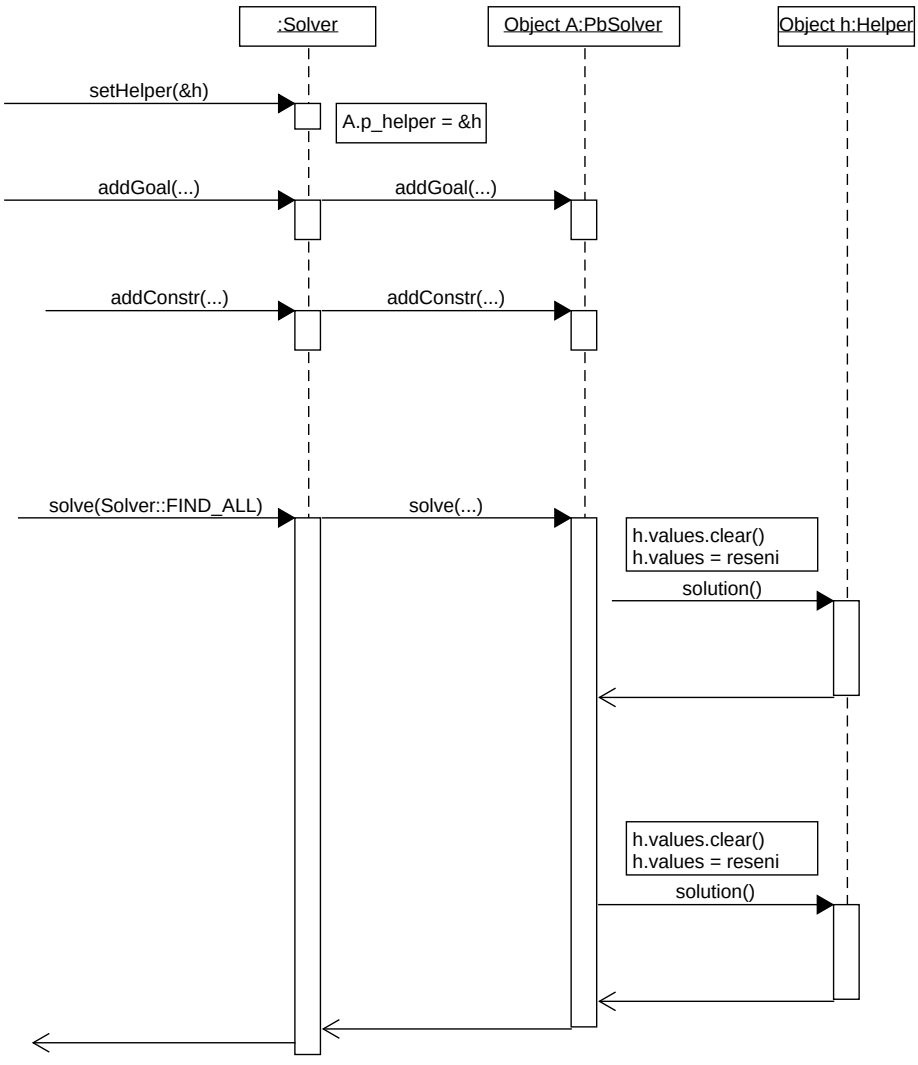
```
#include "minisatp/include/PBLit.h"
#include "minisatp/include/Params.h"
#include "minisatp/include/Helper.h"
#include "minisatp/include/Solver.h"

class WriteSolution : public Helper {
        void solution () {
                cout << "Nalezene␣reseni:" << endl;
                for (size_t i = 0; i < v.size(); ++i) {
                        cout << v[i] << "␣"
                }
                cout << endl;
        }
}

int main(void) {
        Params::setDefaults();
        Solver& s = Solver::getInstance();
        s.init(3,1);
        WriteSolution h;
        s.setHelper(&h);

        vector<PBLit> v1{PBLit(1, 1, false),
        PBLit(1, 2, false),
        PBLit(1, 3, false)}
//vektor reprezentujici levou stranu LS = 1*x1 + 1*x2 + 1*x3
        s.addConstr(v1, Solver::INEQ_GE, 1);
//PB podminka ve tvaru LS >= 1
```

```
        s.solve(Solver::FIND_ALL);
//postupne vypise vsechna reseni {x1, x2, x3} =
//{1, 0, 0}, {0, 1, 0}, {0, 0, 1},
//{0, 1, 1}, {1, 0, 1}, {1, 1, 0} a
//{1, 1, 1}
}
```



Obrázek 5.1: Sekvenční diagram spolupráce tříd Solver a Helper.

5.3 Hlavní program

V následujcím texu podávám popis použitých tříd v hlavním programu nazvaném ATPG_RAS. Načtení obvodu je provedeno přímo frameworkem LogSynth. Další výpočty pak probíhají v rámci instance třídy RAS a pomocná data jsou předávána instancemi třídy Crate.

Crate

Pomocná třída reprezentující návratovou hodnotu z výpočtů, tj. počty změněných bitů a nalezených poruch.

Název atributu	Datový typ	Popis
`changedBitsPI`	`unsigned int`	počet změněných bitů v PI
`changedBitsFF`	`unsigned int`	počet změněných bitů v FF
`droppedFaults`	`unsigned int`	počet nalezených poruch

RAS

Hlavní třída programu.

Popis atributů viz níže.

Název atributu	Datový typ	Popis
`network`	Network&	Síť obvodu
ATPG	ATPG_alg&	ATPG obvodu
remainingFaults	unsigned int	počet zbývajích poruch
numPI	unsigned int	počet PI
numPO	unsigned int	počet PO
numFF	unsigned int	počet FF
testVector	Cube*	poslední vygenerovaný testovací vektor
response	vector<Literal>*	odezva obvodu na poslední testovací vektor
patterns	list<Cube*>	načtené externí testovací vektory

Popis metod viz níže.

Název metody	Návratový typ	Popis
SAT	bool	nalezne testovací vektor pro první dostupnou poruchu z Faultlistu
SAT_PBO	bool	nalezne testovací vektor s minimální vzdáleností dle RAS
fillDCs	void	nahradí DC hodnoty v
loadPatterns	void	načte testovací vektory z externího souboru parametry: string pr_filename
calculate	Crate	provede fault dropping a simulaci pro aktuální testovací vektor. Vrátí počet změněných bitů a počet nalezených poruch

Parametry spuštění

Program (spustitelný soubor) je na příkazové řádce spouštěn příkazem ve tvaru

```
ATPG_RAS parametry zdrojový-soubor[.blif|.bench]
```

kde jako parametry akceptuje následující přepínače:

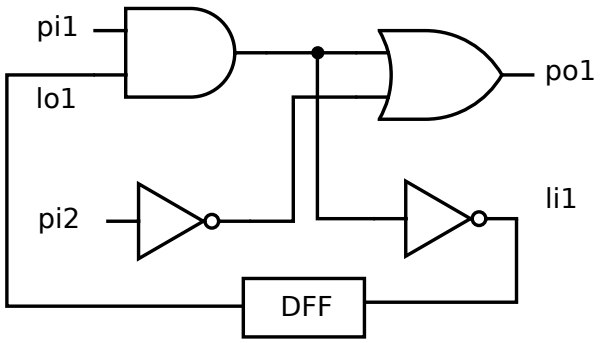
- `-v` Vypíše verzi programu.
- `-pw jmeno-souboru` Nastaví výpisování nalezených testovacích vektorů do daného souboru.
- `-pr jmeno-souboru` Z daného souboru jsou načteny a aplikovány testovací vektory, posléze jsou vektory hledány algoritmem pro RAS.
- -flw soubor Do daného souboru vypíše faultlist
- -ud soubor Do daného souboru jsou vypsány nalezené nedetekovatelné poruchy
- -rep soubor Nastaví vypisování informací o řešení do daného souboru. Do souboru jsou zapisovány informace o počtu změněných bitů a o počtu nalezených poruch. **Povinný parametr**.
- -nomin Vypne hledání minimální vzdálenosti přes zbývající poruchy.
- -rnd n Aplikace vygeneruje n náhodných testovacích vektorů do souboru zadaného pomocí -pw a skončí.
- -dc DC[8] hodnoty v nalezených testovacích vektorech jsou doplňovány náhodně 0/1.

[8] don't care

- -verbose n Nastaví množství informací výpisovaných na standardní výstup z vlastní aplikace (n = 0 až 2, 0 = nejméně výpisů)
- -mv0, -mv1, -mv2 Nastaví množství informací výpisovaných na standardní výstup z výpočtu Minisat+ (-mv0 nejméně informací)

5.4 Testování

Výslednou aplikaci jsem podrobil manuálnímu testování. Vytvořil jsem několik jednoduchých obvodů, viz např. obr. 5.2 a následně jsem tyto použil jako vstup programu.



Obrázek 5.2: Jednoduchý testovací obvod.

Při testování jsem zkoumal

- zda funguje simulace obvodu - výstupy obvodu odpovídají vstupům
- funkce řešiče z hlediska SAT - zda testovací vektor vypočtený pomocí SAT pro jistou poruchu tuto poruchu i skutečne nalezne
- funkce řešiče z hlediska PBO - zda u po sobě jdoucích testovacích vektorů skutečně nastává nějaká minimalizace bitové vzdálenosti

Pro obr. 5.2 je v tabulce 5.2 přehled výstupů z testu.

5.5 Instalace

Program je distribuován ve spustitelné binární formě i v podobě zdrojového kódu. Pro kompilaci je v prostředí OS Windows nutné otevřít řešení LogSynth ve Visual Studiu a zkompilovat projekt ATPG_RAS, přičemž je automaticky vyřešena závislost na projektech Minisat+ a Kernel, které jsou kompilované jako statické knihovny, a na projektu ATPG, se kterým ATPG_RAS sdílí kód. Pro platformu linux je k disposici makefile a program je zkompilován pomocí `make ATPG_RAS`. Závislosti na Minisat+, ATPG a Kernel jsou vyřešeny v rámci makefile.

vstup (pi1, pi2, lo1)	výstup (po1, li1)	změnene bity (pi, FF)	SAT porucha	nalezené poruchy
11 1	1 0	2 1	pi1 / 0	pi1 / 0 lo1 / 0 d → li1 / 0 d → po1 /0
11 0	0 1	0 0	pi2 / 0	pi2 / 0 lo1 / 1 d → li1 / 1 d → po1 / 1
01 1	0 1	1 0	pi1 / 1	pi1 / 1
00 1	1 1	1 0	pi2 / 1	pi2 / 1

Tabulka 5.2: Výstupy z programu pro vstup z obr. 5.2. (d = výstup z AND hradla před větvením)

Kapitola 6

Experimentální výsledky

Vytvořený program jsem se pokusil aplikoval na obvody z balíku ISCAS[9]. Program jsem spouštěl ve versi pro OS Windows na počítači s hardwarovými parametry CPU Intel i5-8250U, 8 GB RAM. Bohužel se ukázalo, že pro složitější obvody algoritmus běží poměrně dlouho (např. pro obvod ISCAS s13207 by byl potřeba přibližně týden) a v rozumném čase byl výpočet schopný doběhnout pouze na jednodušších obvodech. Do experimentálního vyhodnocení tedy zařazuji pouze výsledky z „menších“ obvodů.

V tab. 6 uvádím přehled informací o dvou testovaných obvodech a získané výsledky. Výpočet jsem prováděl nejprve algoritmem pro RAS a porovnávám ho s výpočtem s vypnutou minimalizací. Do délky bitstreamu započítávám počet měněných bitů z PI a počet měněných bitů z FF znásobený délkou adresy ($\lceil \log_2 n_{\mathrm{FF}} \rceil$)

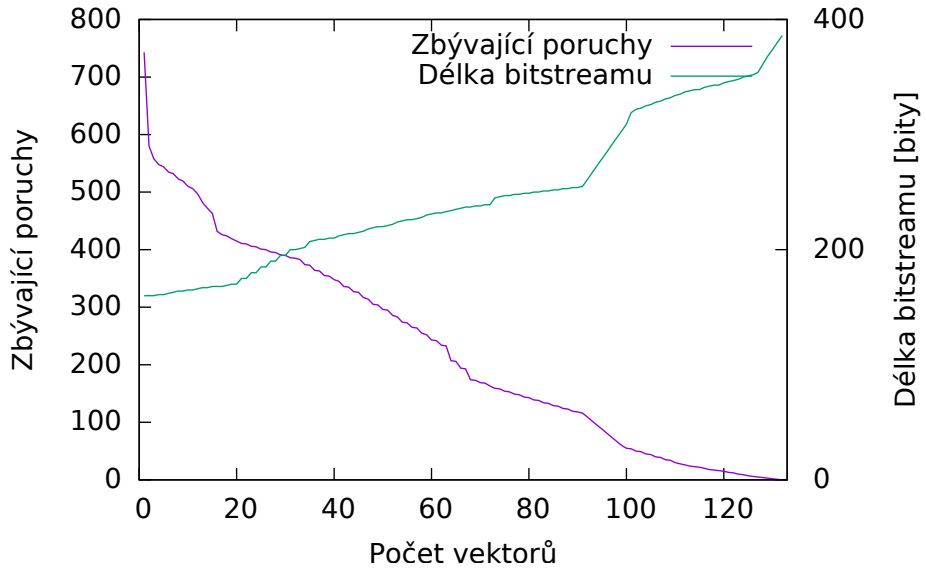
Obvod	PI	PO	FF	Poruch	Vektorů	Délka bitstreamu [bitů]	Délka výpočtů
ISCAS s991	65	17	19	908	132	386	≈ 10 min
ISCAS s991 bez PBO	65	17	19	908	108	6713	≈ 3 min
ISCAS s1423	17	5	74	1495	207	2030	≈ 20 min
ISCAS s1423 bez PBO	17	5	74	1495	88	20316	≈ 5 min
ISCAS s13207	30	116	669	9682	1385	36766	≈ 120 hod

Tabulka 6.1: Přehled pro obvody s991 a s1423

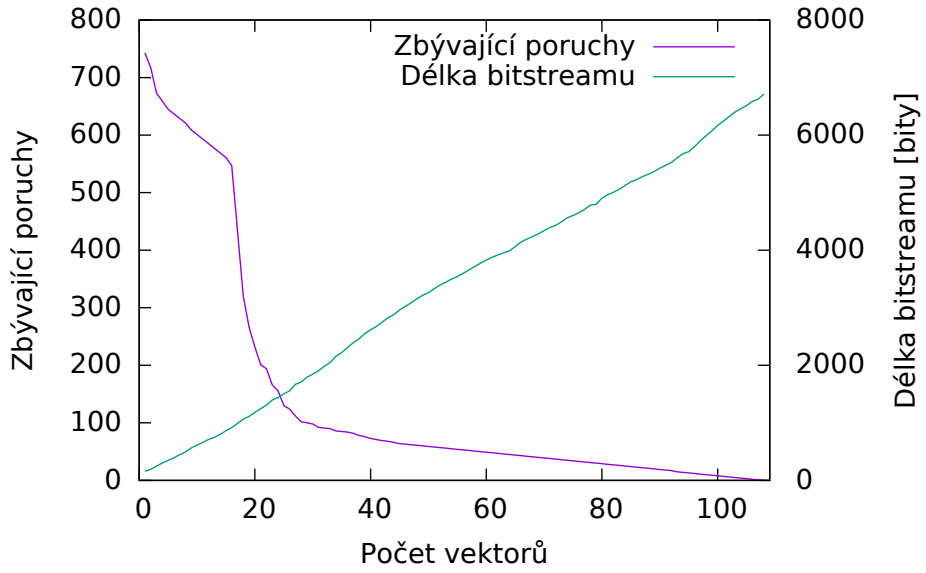
Pro změřené výsledky platí, že algoritmus sice zvýší počet nalezených testovacích vektorů, ale řádově sníží délku bitstreamu (V porovnání s výpočtem, kdy je vypnutá minimalizace). Co se týče výpočtu pro ISCAS s13207, tak se

[9] International Symposium on Circuits and Systems

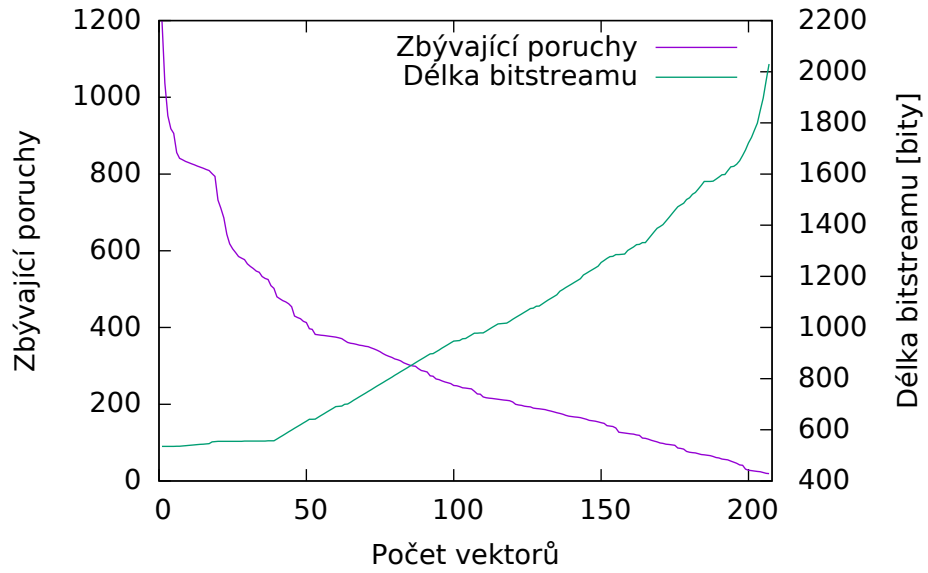
zdá, že získaná data jsou horší než v [11]. Grafické průběhy z výpočtů jsou na obr. 6.1 až 6.5.



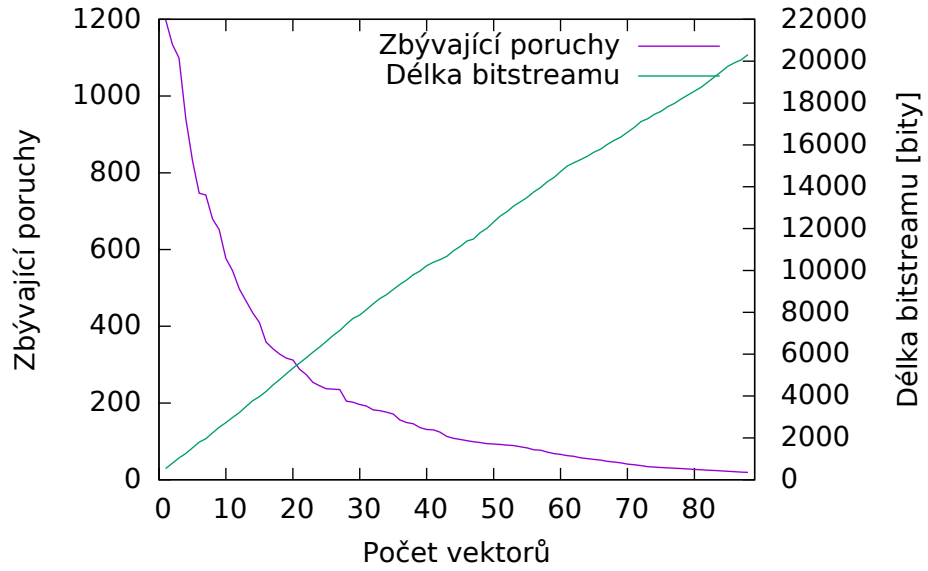
Obrázek 6.1: ISCAS s991.



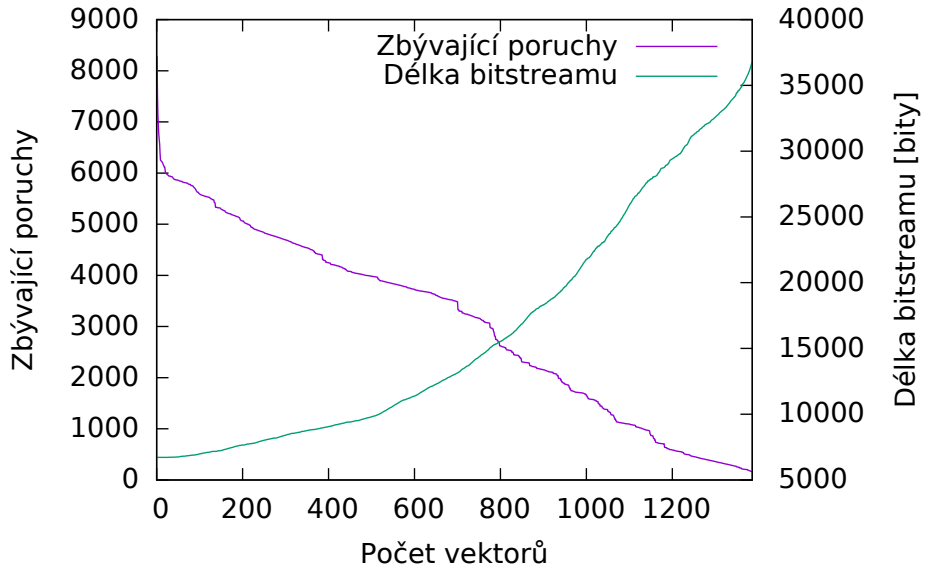
Obrázek 6.2: ISCAS s991 bez minimalizace (pouze SAT).



Obrázek 6.3: ISCAS s1429.



Obrázek 6.4: ISCAS s1429 bez minimalizace (pouze SAT).



Obrázek 6.5: ISCAS s13207.

Kapitola 7

Další vývoj

Kvůli nedostatku času se mi bohužel nepodařilo zrealizovat vše, co jsem původně zamýšlel. Jako další nápady či návrhy pro rozvoj programu bych zmínil zejména:

Analýza vlivu parametrů PbSolveru

Při vývoji programu bohužel nevyzbyl čas na zkoumání vlivu parametrů solveru, tj. vliv nastavení původních globálních proměnných `opt_X` (přistupovaných pomocí statických metod třídy `Params`). Takto jde např. omezit metoda převodu PB podmínek do SAT a též je možno nastavovat několik číselných parametrů. Pro toto by zřejmě bylo nutné důkladně nastudovat článek autorů solveru, neboť ze zdrojového kódu programu je význam proměnných zřejmý jen rámcově. Dále by bylo nutné se zamyslet nad způsobem vstupu těchto parametrů do programu, protože větší počet přepínačů ve vstupu by učinil program ještě více nepřehledným.

Paralelizace

Dalším vhodným kandidátem na další rozvoj programu by byl převod na paralelní výpočet. Výhody takového přístupu jsou jasné - v dnešní době je u většiny PC k disposici 4 či více jader procesoru a došlo by tak k (přibližně) stejnému urychlení běhu programu. Jádro algoritmu tvořené hledáním minima přes množinu poruch by bylo jednoduché upravit jako úlohu producent-konzument, globální datové struktury, které jsou v PbSolveru sdíleny, by však bylo nutné odstranit.

Závěr

Cílem této práce bylo navrhnout a implementovat aplikaci pro generování komprimovaného testu pro číslicové obvody s RAS architekturou. Aplikace měla být založena na frameworku LogSynth a řešiči Minisat+. I přes nedostatek času mám za to, že se tento cíl z větší části naplnit podařilo, přestože se při vývoji aplikace objevila řada jevů, které by bylo dobré dále hlouběji zkoumat a věnovat jim pozornost, jako např. vliv doplňování DC hodnot do testovacích vektorů. Jako pozitivní bych hodnotil to, že se v rámci vývoje podařilo objevit a opravit i několik chyb v použitém frameworku a v Minisat+.

Další vývoj a práce s aplikací by mohly zahrnovat zkoumání na více rozsáhlejších obvodech, paralelizaci či optimalizaci simulace obvodů v LogSynthu, čímž by se výpočty mohly urychlit.

Literatura

[1] Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; aj.: Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, ACM, 2001, s. 530–535.

[2] Balcarek, J.; Fiser, P.; Schmidt, J.: PBO-based test compression. *Proceedings - 2014 17th Euromicro Conference on Digital System Design, DSD 2014*, 10 2014: s. 679–682, doi:10.1109/DSD.2014.86.

[3] Ababei, C.: COEN-4730 Computer Architecture Lecture 12 Testing and Design for Testability (focus: processors). Dostupné z: `http://www.dejazzer.com/coen4730/doc/lecture13_testing.pdf`

[4] Fišer, P.: Testování a spolehlivost ZS 2017/2018, 1. přednáška - Úvod Terminologie, typy defektů, poruch. Dostupné z: `https://moodle-vyuka.cvut.cz`

[5] Fišer, P.: Testování a spolehlivost ZS 2017/2018, 2. přednáška - Testování kombinačních obvodů Intuitivní zcitlivění cesty, D-algoritmus. Dostupné z: `https://moodle-vyuka.cvut.cz`

[6] Een, N.; Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. *JSAT*, ročník 2, 03 2006: s. 1–26, doi:10.3233/SAT190014.

[7] Fišer, P.: Testování a spolehlivost ZS 2017/2018, 3. přednáška - Pokročilé algoritmy generování testu. Dostupné z: `https://moodle-vyuka.cvut.cz`

[8] Fišer, P.: Testování a spolehlivost ZS 2017/2018, 4. přednáška - Testování sekvenčních obvodů. Simulace poruch. Kompakce testu. Dostupné z: `https://moodle-vyuka.cvut.cz`

[9] Fišer, P.: Testování a spolehlivost ZS 2017/2018, 8. přednáška - Testování sekvenčních obvodů. Scan návrh. Dostupné z: `https://moodle-vyuka.cvut.cz`

[10] Fišer, P.: Testování a spolehlivost ZS 2017/2018, 11. přednáška - Komprese testu. Dostupné z: `https://moodle-vyuka.cvut.cz`

[11] Baik, D.; Saluja, K.; Kajihara, S.: Random access scan: A solution to test power, test data volume and test time. 02 2004, ISBN 0-7695-2072-3, s. 883– 888, doi:10.1109/ICVD.2004.1261042.

[12] Eén, N.; Sörensson, N.: An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, editace E. Giunchiglia; A. Tacchella, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ISBN 978-3-540-24605-3, s. 502–518.

Příloha **A**

Seznam použitých zkratek

ATPG Automatický generátor testovacích vektorů pro číslicové obvody (*Automatic Test Pattern Generator*)

CNF, KNF Konjunktivní normální forma

CUT, DUT Testovaný obvod (*chip under test, device under test, design under test*)

DFF Klopný obvod typu D, 1-bitová paměť

PBO Pseudo-booleovská optimalizace

PI Primární vstup či vstupy

PPI Pseudo-primární vstup či vstupy

PO Primární výstup či výstupy

PPO Pseudo-primární výstup či výstupy

RAS Random access scan

SAT Problém splitelnosti booleovské formule

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `bin` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF