Introduction to Electronic Design Automation

Final Project

2014 ICCAD CAD Contest Problem A

Simultaneous CNF Encoder Optimization with SAT Solver Setting Selection

B99901027 楊恭年
B99901086 吳浩寧

2014/6/25

CONTENTS

1 Introduction

The subject of this final project is problem A of 2014 ICCAD CAD Contest. For a detailed description of the problem, please visit the problem's webpage [1]. Section 1.1 gives a brief explanation of the problem, and in Section 1.2, we introduce the method we use to solve the problem.

1.1 Problem Description

The objective of the problem is to explore the best settings of the CNF encoder and the SAT solver, and then solve the SAT problem using the settings explored. Given 3 samples and 10 tests in each test case, the solving process is divided into 2 phases:

- **Exploration phase**. In this phase, the program (in exploration mode) should explore the best settings of CNF encoder and SAT solver based on 3 sample circuits (Verilog netlists). The time limit of this phase is 1800 seconds. After exploration, 2 setting files will be generated–the *encoder setting file* and the *solver setting file*.
- **Generation phase**. In this phase, the program (in generation mode) will first generate the CNF file of the test circuit (Verilog netlist) using the settings recorded in the encoder setting file. The CNF file, along with the solver setting file, is then presented to the SAT solver, which will then solve the SAT problem. The time limit of CNF generation is 25 seconds, and that of SAT solving is 100 seconds.

Figure 1 shows the flow charts of the two phases.

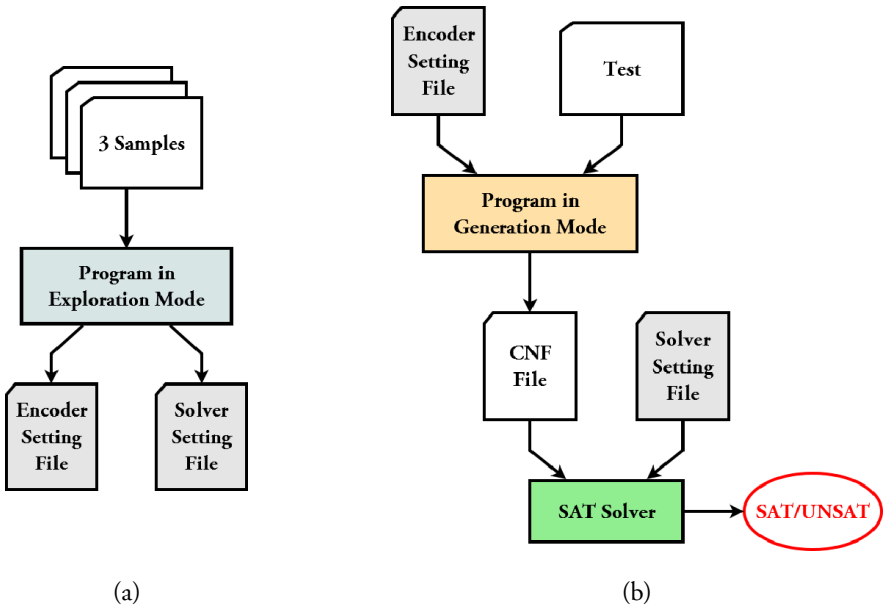

FIGURE 1 Program flow chart. (a) Exploration phase. (b) Generation phase.

1.2 Overview

The design of our program relies mainly on the ABC library. ABC is a system for sequential synthesis and verification developed by numerous people and maintained by Alan Mishchenko at UC Berkeley. The system also includes many other software packages, such as MiniSAT and BDD package [2].

ABC handles almost everything in our program, including file I/O, data storage and logic synthesis. The basic idea of CNF generation in our program is described as follows, where all of the steps are performed using ABC commands.

- Read a Verilog netlist and store the circuit as an AIG.
- Apply logic synthesis to simplify the circuit.
- Export the simplified circuit to a CNF file.

When using the method described above to encode the CNF file, the quality of the resulting CNF file depends mainly on the second step, that is, the synthesis process. As a consequence, the objective of the encoder setting exploration is to find such a process that minimizes the circuit. The algorithm is described in Section 2.2.

As for the solver setting exploration, we simply try out different parameter settings during exploration, and choose the one that makes the solving process progress the fastest. The SAT solver we use is Glucose 3.0 [3], and the algorithm is described in Section 2.3.

2 Algorithm and Implementation

The following sections describe the algorithm and some implementation issues of the program. For more details, please refer to the source code of the program as well as the README file.

2.1 Modifying ABC

The original ABC package is powerful, but it needs little modification to fit our needs.

- The Verilog file reader in ABC is not general enough to parse all Verilog files presented in the test cases. To be more specific, it cannot parse wires declared between the instantiations of logic gates. To enable ABC to parse such wires, 2 lines of code were added:

abc/src/base/ver/verCore.c

```
494 else if ( !strcmp( pWord, "wire" ) )
495     RetValue = Ver_ParseSignal( pMan, pNtk, VER_SIG_WIRE );
```

- In order to access the information about the circuit stored in ABC, 2 functions were added:

abc/src/base/abci/abc.c

```
64 int Abc_NtkNodeNumNonStatic(Abc_Ntk_t *pNtk) { return pNtk->nObjCounts[ABC_OBJ_NODE]; }
65 int Abc_NtkPiNumNonStatic  (Abc_Ntk_t *pNtk) { return Vec_PtrSize(pNtk->vPis); }
```

The first function returns the gate count of the circuit, and the second one returns the number of PIs. These data are used in both exploration and generation.

- To compile the ABC library on systems without the ncurses, readline and pthread libraries, the following lines of code were commented:

abc/src/misc/util/abc_global.h	lines 279-280
abc/src/aig/gia/giaKf.c	line 1023

These modifications should have no effect on the functionality of the program.

2.2 Encoder Setting Exploration

As described in Section 1.2, the exploration of encoder settings is to find a synthesis process that can minimize the circuit. A synthesis process is a series of synthesis commands provided by ABC. The commands used in the program are "resyn2", "compress2" and "drwsat2". These commands are called "standard (synthesis) scripts" in ABC and are composed of several elementary commands such as "rewrite" and "refactor". For the details of the synthesis scripts, please refer to the file **abc/abc.rc**.

In the program, there are 2 arrays, or command lists, *cmdList*[0] and *cmdList*[1], each of which stores a sequence of commands. The contents of these 2 arrays are listed in Table 1.

TABLE 1 Contents of the command lists.

Command List	Commands				
cmdList[0]	resyn2	resyn2	resyn2	resyn2	drwsat2
cmdList[1]	compress2	compress2	compress2	compress2	(none)

A synthesis process is defined by the 5-element Boolean vector *comb* (abbreviation of "combination"), a vector that contains a sequence of TRUE or FALSE values. These values are used to select commands stored in the command lists. If the element is FALSE, then the command is taken from *cmdList*[0]; if it is TRUE, then the command is taken from *cmdList*[1]. For example, if *comb* = {TRUE, FALSE, FALSE, TRUE, FALSE}, then the synthesis process will be

compress2; resyn2; resyn2; compress2; drwsat2

Different truth values of *comb* will lead to different synthesis processes. Since *comb* has 5 elements, there is a total of 32 combinations. The exploration process simply tries out all these combinations and chooses the one that produces the circuit with the smallest gate count. The combination that produces the smallest circuit is called *minComb*. After *minComb* is obtained, the synthesis process defined by *minComb* is repeated several times in order to further simplify the circuit. The number of iterations, with the upper bound 8, that produces the smallest circuit is recorded.

The above process is applied to all of the 3 samples. The (simplified) pseudocode in Code 1 summarizes the algorithm.

CODE 1 Simplified pseudocode of encoder setting exploration.

```
1   for each sample
2      for each combination comb
3         read the sample circuit
4         synthesize the circuit according to comb
5         minComb = the comb that minimizes the circuit
6      end
7      read the sample circuit
8      for iter = 1 to 8
9         synthesize the circuit according to minComb
10        numIter = the iter that minimizes the circuit
11     end
12     record minComb and numIter in the encoder setting file
13  end
```

The format of encoder setting file is illustrated in Figure 2. The first line is file header, where the first number represents the number of samples and the second one represents the number of commands in a synthesis process. The following 3 lines are the exploration results of the 3 samples. In each line, the first 5 numbers represent the *minComb* vector, and the last number represents *numIter*.

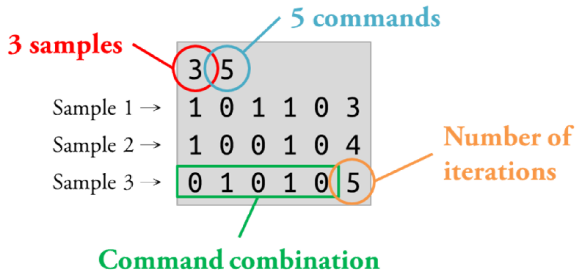

FIGURE 2 Encoder setting file format.

2.3 Solver Setting Exploration

The exploration of solver settings follows that of encoder settings. As mentioned in Section 1.2, the method we use is simply trying out different settings and select the one with the best performance. The settings we try during exploration are

```
glucose (default settings)
glucose -K=0.75
glucose -K=0.85
glucose -R=1.5
glucose -R=1.5 -K=0.75
glucose -R=1.5 -K=0.85
```

Before solver setting exploration, the program will first generate 3 CNF files using the 3 sample circuits and the settings just explored. These files will be removed by the program after exploration since we are not allowed to generate files that are not specified in the problem but we are allowed to generate such files and remove them after execution. Each setting is tried 3 times with the 3 sample CNF files. The setting with the best average performance is recorded in the setting file.

An auxiliary program, called `glucose_exp`, is used as the SAT solver during exploration. This program is basically the same as the original Glucose program except some modifications in the file **`glucose_exp/core/Main.cc`**.

`glucose_exp/core/Main.cc`

```
 88  static void signalHandler(int a) { raise(SIGINT); }
...
 92  signal(SIGALRM, signalHandler);
 93  signal(SIGINT, SIGINT_interrupt);
 94  alarm(60);
```

Since the process of SAT solving may consume a lot of time, we need a way to terminate the SAT solver after some time interval so that we can try out every settings within the time limit. The above modifications enable Glucose to terminate itself after a fixed time interval (60 seconds). In addition, Glucose provides a handler that can catch the `SIGINT_interrupt` signal and print the statistics of the solving process at the end of the execution. These statistics are used to analyze the performance of the solver.

Glucose provides several parameters to optimize its solving strategy. The biggest difference between Glucose and MiniSAT is that Glucose allows users to set parameters controlling the restart strategy. These parameters include *szLBDQueue, szTrailQueue, R* and *K*. *szLBDQueue* and *R* are used to accelerate the restart process. When

$K \times$ (the average of the last *szLBDQueue* LBD scores) > the average of all LBD scores

where *szLBDQueue* = 50, restart will be instantly triggered. On the contrary, *szTrailQueue* and K are used to postpone the restart process. When a conflict occurs and

trail size > $R \times$ (the average trail size of the last *szTrailQueue* conflicts)

where *szTrailQueue* = 5000, *LBDQueue* will be emptied, and thus a possible restart is postponed. The default values of K and R in Glucose are 0.8 and 1.4, which have been proven to be the best for general cases [4].

In the program, we focus on tuning the parameters K and R. We simply assign values slightly different from the default values to K and R, and try out some combinations of these settings. If the solving process ends before timeout, we use CPU time to evaluate the progress. Otherwise we use "nb learnts DL2", part of the statistics, as an estimation of the progress since most of the other statistics seem to be less relevant to the solving speed. We assume that the greater the value of "nb learnts DL2", the faster the solving process is. Figure 3 shows the relation between "nb learnts DL2" and the solving speed.

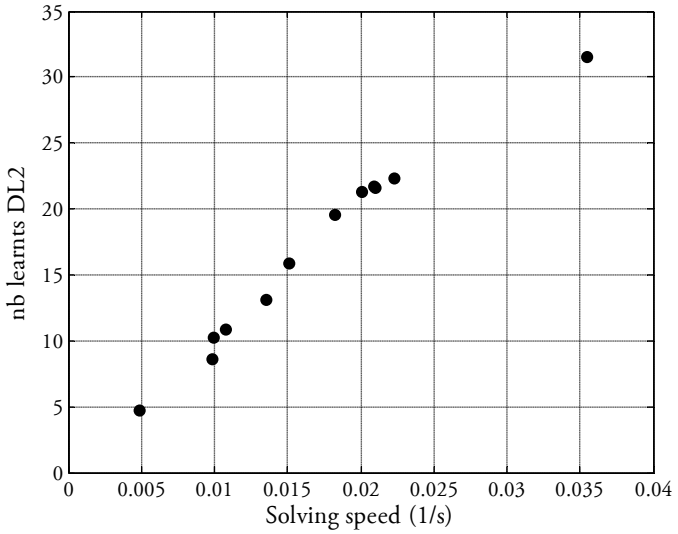

FIGURE 3 Relation between "nb learnts DL2" and solving speed. These data are obtained using the CNF file of the circuit **`test9.v`** of test case ut2.

2.4 CNF Generation

Given the encoder setting file, the program should generate the CNF file of the test circuit based on the settings explored. In this section, we illustrate the generation process with an example. Suppose that the setting file contains the following data:

```
3 5
1 0 1 1 0 3
1 0 0 1 0 4
0 1 0 1 0 5
```

The 3 *minComb* vectors are {1, 0, 1, 1, 0}, {1, 0, 0, 1, 0} and {0, 1, 0, 1, 0}. The vector that has the minimum *mean squared error* with respect to the 3 *minComb*s is the one that defines the synthesis process for the test circuit. The error here is the Hamming distance between the two vectors. In this case, the resulting combination is {1, 0, 0, 1, 0}, and the corresponding synthesis process is

Synthesis process = compress2; resyn2; resyn2; compress2; drwsat2

The number of iterations is obtained simply by taking the average of the 3 numbers in the last column, that is, 3, 4 and 5.

Number of iterations = (3 + 4 + 5) / 3 = 4

The test circuit is then synthesized using the settings obtained above, and the CNF file is generated using the corresponding ABC command.

To speed up SAT solving, in some cases the test circuit is *collapsed* (using the ABC command "collapse") before exported to a CNF file. If the original test circuit satisfies condition C (explained later), the CNF file is generated directly through collapsing the circuit. If the circuit after synthesis satisfies condition C, then it is also collapsed before writing the CNF file. The condition C in the program is

(gate count < 2000 AND number of PI < 30) OR (gate count < 1100)

which means that collapsing is only applied to small circuits. For large circuits, collapsing becomes inefficient.

3 Testing Results

This section shows some results of our tests. We ran our tests on Cygwin on a computer with Intel Core i5-2410M, 2.30GHz CPU and 4G RAM. Red numbers in the Tables indicate that the execution times exceed the time limit.

- Test case ut1

TABLE 2 ut1. Exploration time: 648.858 (sec)

Test number	CNF generation (sec)	SAT solving (sec)	SAT solving with default settings (sec)	SAT solving result
1	0.483	0	0	UNSAT
2	0.639	0.031	0	UNSAT
3	10.062	0	0	UNSAT
4	21.371	0	0	UNSAT
5	17.347	0.031	0.015	UNSAT
6	13.837	0	0.015	UNSAT
7	18.361	0	0	UNSAT
8	20.389	0	0	UNSAT
9	18.767	0	0	UNSAT
10	31.34	0	0.015	UNSAT

- Test case ut2

TABLE 3 ut2. Exploration time: 878.325 (sec)

Test number	CNF generation (sec)	SAT solving (sec)	SAT solving with default settings (sec)	SAT solving result
1	15.912	1.31	1.887	UNSAT
2	19.952	4.992	3.744	UNSAT
3	21.559	28.797	31.122	UNSAT
4	23.962	timeout	timeout	--
5	26.176	timeout	timeout	--
6	20.015	6.38	4.524	UNSAT
7	19.937	14.289	22.276	UNSAT
8	20.108	12.074	35.786	UNSAT
9	22.214	44.881	83.242	UNSAT
10	22.401	0	0.015	SAT

- Test case ut3

TABLE 4 ut3. Exploration time: 1595.5 (sec)

Test number	CNF generation (sec)	SAT solving (sec)	SAT solving with default settings (sec)	SAT solving result
1	95.114	timeout	timeout	--
2	70.434	timeout	timeout	--
3	81.292	timeout	timeout	--
4	87.095	timeout	timeout	--
5	88.717	timeout	timeout	--
6	81.604	timeout	timeout	--
7	98.749	timeout	timeout	--
8	57.696	timeout	timeout	--
9	74.163	timeout	timeout	--
10	79.232	timeout	timeout	--

- Test case ut5

TABLE 5 ut5. Exploration time: 893.769 (sec)

Test number	CNF generation (sec)	SAT solving (sec)	SAT solving with default settings (sec)	SAT solving result
1	4.087	timeout	timeout	--
2	4.386	timeout	timeout	--
3	5.147	timeout	timeout	--
4	5.725	timeout	timeout	--
5	6.037	timeout	timeout	--
6	5.724	timeout	timeout	--
7	6.504	timeout	timeout	--
8	6.068	72.68	104.473	UNSAT
9	7.099	44.117	25.506	UNSAT
10	7.331	20.311	13.884	UNSAT

4 Conclusions and Future Work

In SAT solving phase, most of the cases' solving times either expire time limit, 100 seconds, or are too small to justify the performance of our program. Nevertheless, we can still notice the significant improvement in case ut2. In the future, we will work on finding new ways to estimate SAT solver's progress more precisely, testing more combinations of different parameters. We are also thinking about using machine learning methods to analyze the sample files directly.

5 References

[1] 2014 ICCAD CAD Contest Problem A.
http://cad_contest.ee.ncu.edu.tw/CAD-contest-at-ICCAD2014/problem_a/default.html

[2] Main page of ABC.
http://www.eecs.berkeley.edu/~alanmi/abc/

[3] Home page of the Glucose SAT solver.
http://www.labri.fr/perso/lsimon/glucose/

[4] Audemard, G., Simon, L. *Refining restarts strategies for SAT and UNSAT*

6 Appendix

6.1 Job Division

TABLE X Job Division

Task	Person
Studying ABC	楊恭年
Studying Glucose	吳浩寧
Formal testing	吳浩寧
Report	楊恭年
Source code	
main.cpp	楊恭年
explorer.h	楊恭年
explorer.cpp	楊恭年
exploreGlucose.cpp	吳浩寧
generator.h	楊恭年
generator.cpp	楊恭年