SOC Verification HW5Report

Arvind Singh Rathore
R10943157

I. Implementation

i) V3NetId SATMgr::buildInitState() const

In order to accomplish this, the flip-flops must be reset to the initial state 0. The function can be written as follows: $\prod(S_i \equiv 0) = \prod(S_i0)(\sim S_i1) = \prod(\sim S_i)$. Therefore, I inversed all states and "and" them.

ii) void SATMgr::itpUbmc(const V3NetId& monitor, SatProofRes& pRes)

This is the main part of the Algorithm. I followed the TODO as decribed in satMgr.cpp

PART I: Build Initial State:

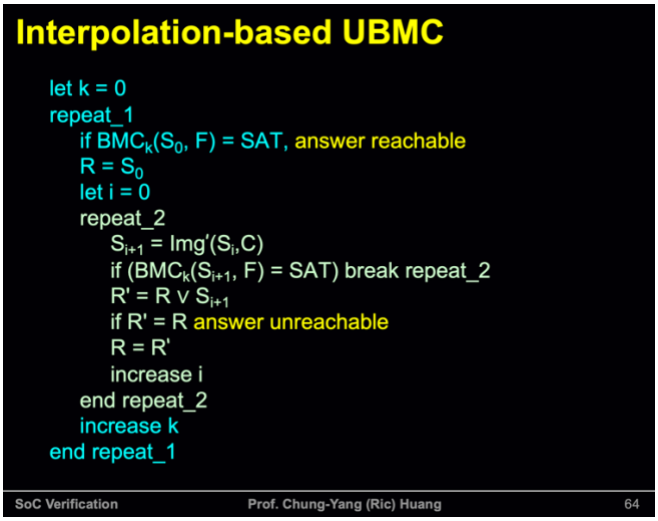
First I called the above function to get initial function. Also, I set some preliminaries such as S_0 by addBoundedVerifyData(I, 0), P by addBoundedVerifyData(monitor, 0). Still SAT based solver find the answer by solving $\sim P$, therefore, I call assumeProperty(monitor, false, 0). Clauses are added by the same approach in the below part.

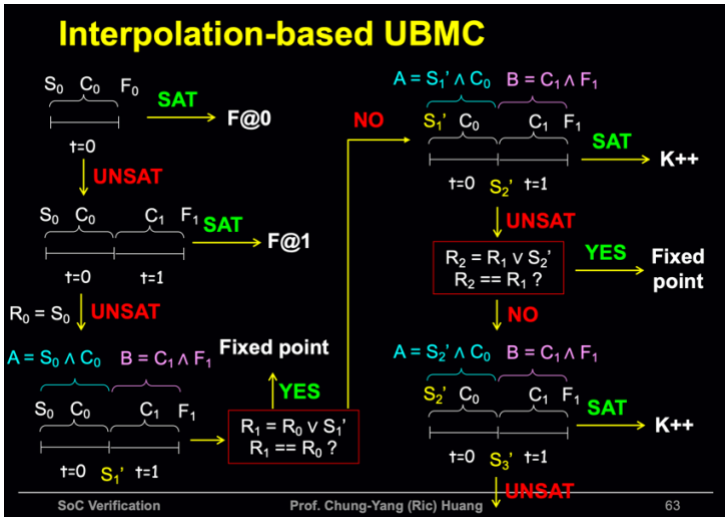
PART II: Take care the first timeframe (i.e., Timeframe 0):

The first thing I check is if the initial state S_0 reach $\sim P$. If yes, the monitor is unsafe; else, I construct C_0 by looping all the latches. Since the basic interpolation- based algorithm separates the onset part $A = S_0 \wedge C_0$ and offset part B at S_1, all these constructions belong to onset.

PART III: Start the ITP verification loop:

As taught in the lecture slides, I basically understand and follow the same procedure.





This part is identical to the above figure. Because of the implementation method merely follows the lines, I will only point out a few important points where I encountered errors while implementing this algorithm & not shown in the figure.

To begin, I implement $R \equiv R' = R \vee S'$ for checking fixed point. In this case, I constructed R' by creating V3AndGate(ntk, ~$\boldsymbol{R}$', ~$\boldsymbol{S}$', ~$\boldsymbol{R}$), S' and R, which represent interpolation and origin reach states, respectively.

After that, I produced an equivalent, which is neither exclusive nor as follows:

```
V3NetId posForm = _ntk -> createNet();
createV3AndGate(_ntk, posForm, newReachState, reachState);
V3NetId negForm = _ntk -> createNet();
createV3AndGate(_ntk, negForm, ~newReachState, ~reachState);
V3NetId nOR = _ntk -> createNet();
createV3AndGate(_ntk, nOR, ~negForm, ~posForm);

_ptrMinisat->resizeNtkData( _ntk->getNetSize() - oriNetSize);

satSolver -> assumeRelease();
satSolver -> addBoundedVerifyData(nOR, 0);
satSolver -> assumeProperty(nOR, false, 0);
```

The second and most difficult part is obtaining interpolation. The above approach suggests that we conduct interpolation while entering the loop, but this has a significant flaw. The repeat 2 loop, as described in the lecture slides, will compute if fixed point, which requires adding bounded verify data to the SAT solver; however, the two interpolation parameters should not be dependent on exclusive gates, so I move the line of computing interpolation before checking fixed point. Last but not least, depth should increase when the inner loop approaches fault, which is a fictitious counter example. Prior to K_curr++, I must assert P to the current depth.

```
if(!(satSolver->assump_solve()))
{
   interpolation = getItp();
   int oriNetSize = _ntk -> getNetSize();
   S_ID = getNumClauses();
   V3NetId newNegReachState = _ntk -> createNet();

   createV3AndGate(_ntk, newNegReachState, ~interpolation, ~reachState);
   V3NetId newReachState = ~newNegReachState;
```

Calculate ITP before computing $R' = R \vee S'$

The result of ref and mine is presented in above table. Note that execution time may change due to the loading of CPU, and the usages of memory are almost the same between ref and mine. Detailed results are saved in each directory.

tests/
- basic/
  - a.log
  - b.log
  - c.log
- hwmcc/
  - unsat.log
  - sat.log
- vending/
  - (HW1 version) vending.log
- vending-abstracted/
  - (HW3 abstracted version) vending-abstracted.log

Directory graph

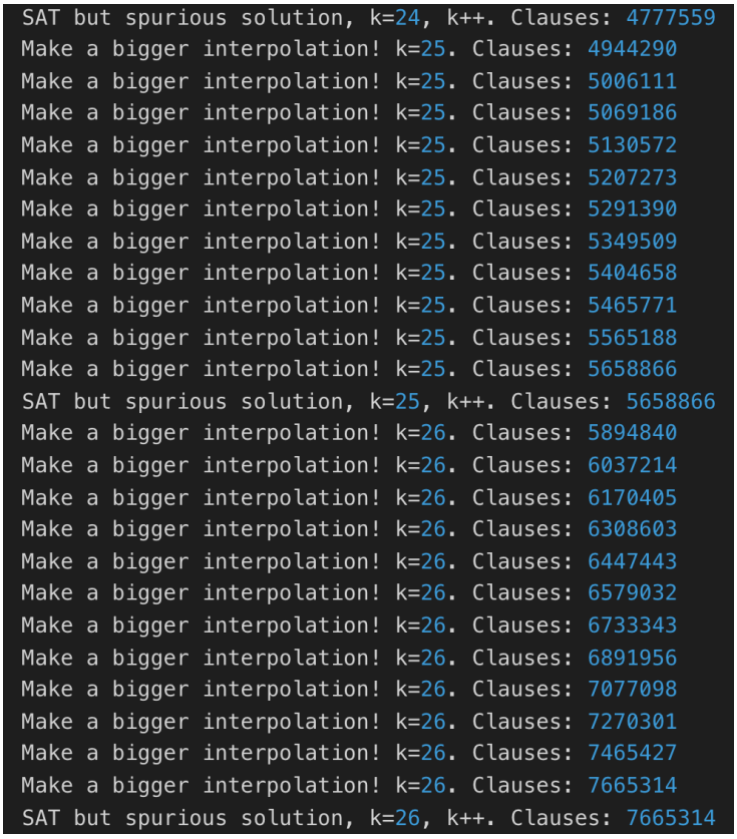
II) Verification results

Tests	Monitor / File	Ref	My	Ref_time	My_time	Ref memory usage	My memory usage(Mb)
SAT	6s307rb06	14	14	2.63	9.67	76.92	76.91
	6s374b029	10	10	88.87	177.1	309.9	311.2
	6s388b07	0	0	0.03	0.11	309.9	311.2
	abp4pold	17	17	6.84	51.21	309.9	311.2
	bob9234spec7neg	512	512	155.9	793.5	309.9	311.2
	bobpci215	10	10	4.06	32.97	309.9	311.2
UNSAT	6s136	safe	safe	13.98	72.46	98.45	118.7
	6s206rb025	safe	safe	1.64	6.63	157.1	126.5
	6s221rb18	safe	safe	26.02	86.71	287.6	288.6
	6s326rb02	safe	safe	11.58	14.45	287.6	288.6
	6s327rb10	safe	safe	0.33	1.45	287.6	288.6
	6s380b129	safe	safe	0.95	3.99	287.6	288.6
	6s6	safe	safe	15.91	48.1	287.6	288.6
	pdtpmsfpmult	safe	safe	1.12	>10 min	287.6	
	pj2018	safe	safe	26.74		287.6	
a	z1	safe	safe	0.02	0.07	3.691	3.68
	z2	safe	safe	0	0.05	3.766	3.891
	z3	safe	safe	0.01	0.06	3.82	3.891
	z4	safe	safe	0.01	0.02	3.805	3.766
b	p1	safe	safe	0.01	0.06	4.07	4.062
	p2	safe	safe	0.02	0.89	4.4492	5.98
	p3	42	42	7.11	103.5	25.86	58.18
c	z0	0	0	0	0	3.301	3.297
	z1	5	5	0	0.01	3.43	3.426
	z2	6	6	0	0.01	3.43	3.426
	z3	safe	safe	0.01	0	3.43	3.426
vending.v	p	9	9	0.45	1.8	11.77	12.38
	checkforinitialized	1	1	0.01	0.04	11.77	12.38
	checkcorrectchange	9	9	0.39	2.13	11.22	11.86
vending-fixed.v	p	Runtime exceeds 10 minutes					
	checkforinitialized	safe	safe	0.04	0.15	11.22	11.86
	checkcorrectchange	Runtime exceeds 10 minutes					
vending-abstracted.v	p	17	17	1.38	7.78	11.34	7.164
	checkforinitialized	1	1	0	0.01	11.34	7.164
	checkcorrectchange	17	17	1.43	7.92	10.89	10.12
vending-abstracted-fixed.v	p	safe	safe	0.27	1.27	10.89	10.12
	checkforinitialized	safe	safe	0	0.02	10.89	10.12
	checkcorrectchange	safe	safe	0.22	1.26	10.89	10.12

Observations: My program is able to achieve exactly the same correctness as ref program.
From the above table, we can observe that in most cases, ref code is faster than my code, but in **pdtpmsfpmult**, ref code takes only 1.15 second, which is truly amazing but perplexing, and my runtime exceeds more than 10 minutes This difference, I believe, is due to the ref program's upgraded algorithm. We can notice that two programs are comparable in aspects like basic and SAT. My program generally uses less memory.

Nonetheless, I'm looking at the relationship between clauses and speed. It can be boiled down to two main points. First, as the clauses become more complex, the time may increase exponentially (or not in linear & square). Because it has 3,251,454 clauses, **Pdtpmsfpmult** takes much longer than others. Second, **bob9234spec7neg** demonstrates that even though the counter example is not that large (k=512), it must take a long time. To improve performance, consider using reset or discard clauses in routines to reduce the amount of clauses and speed up the program.

The program failed when checking the correct counter example for the vending machine, the original design in HW1. I tried the abstracted version to check the correctness, the result seems good in above table. Therefore, I try to investigate what's wrong.



Interpolation produces a significant number of clauses, as shown in the diagram above. Take, for example, k=25, 26. There are 881,307 clauses in k=25 and 2,006,448 clauses in k=26. Worse, when k was increased to 27, practically all of these clauses became obsolete. They were not, however, discarded. Because of this, the program is unable to compute the algorithm. We should include a method to remove certain unnecessary clauses from the program to reduce the SAT solver's workload.

III. Comparison with the ref program and other model checkers

In the previous section, I compared my program to the ref program. These sizes are incompatible with other model checkers, such as those based on BDD. We will get an error that says BDD Support Size is Smaller Than Current Design Required!! when running unsat.dofile and sat.dofile. As a result, the model checkers fail for the entire BDD checker or the combination of BDD and SAT solver.