Logic Synthesis & Verification, Fall 2025

National Taiwan University

Programming Assignment 2

Due on 11/30 23:59 on GitHub.
Updated 11/4.

Submission Guidelines. Please develop your code under `src/ext-lsv`, and do not modify any code outside `src/ext-lsv`, and we will only copy your files under `src/ext-lsv` for evaluation. You are asked to submit your assignments by creating pull requests to your own branch. To avoid plagiarism, please push files and create pull requests in the last three hours (21:002̃3:59) before the deadline. See GitHub page (https://github.com/NTU-ALComLab/LSV-PA) for more details.

1 [Unateness Checking with BDD] (50%)

Given a circuit C in BDD, a primary output y_k, and a primary input x_i, write a procedure in ABC to check whether the function of y_k is binate, positive unate, negative unate in x_i, or independent of x_i. If the function of y_k is binate in x_i, show that it is indeed binate by giving some input patterns.

Integrate this procedure into ABC (under `src/ext-lsv/`), so that after reading in a circuit (by command "`read`") and transforming it into BDD (by command "`collapse`"), running the command "`lsv_unate_bdd`" would invoke your code. The command should have the following format.

```
lsv_unate_bdd <k> <i>
```

where k is a primary output index starting from 0, and i is a primary input index starting from 0.

If the function of y_k is positive unate, negative unate in x_i, or is independent of x_i, print one of the following three lines:

```
positive unate
negative unate
independent
```

Otherwise, print "`binate`" and show that it is binate in the following format.

```
binate
<pattern 1>
<pattern 2>
```

Patterns 1 and 2 are primary input assignments where all the primary inputs, except x_i, are assigned as 0 or 1. For x_i, please use "-" to represent its value. Cofactoring the function of y_k with respect to the cube of Pattern 1 (respectively, Pattern 2) produces a new function equal to x_i (respectively, $\neg x_i$).

For example, consider the checking for function $y_0 = (x_0 \oplus x_1) \vee x_2$ on variable x_0. The command should output the following information:

```
abc 01> lsv_unate_bdd 0 2
positive unate
abc 02> lsv_unate_bdd 0 0
binate
-00
-10
```

Hint 1. When operating BDDs, remember to use `Cudd_Ref` when you create a BDD node and use `Cudd_RecursiveDeref` when you dereference a BDD node. This helps to avoid `cuddGarbageCollect` errors. Here is an example that shows how to use these commands.

```
Ddnode* cube = Cudd_ReadOne(manager);
for (int i = 0; i < n; ++i) {
    Ddnode* var = Cudd_bddIthVar(manager, i);
    Cudd_Ref(var);
    Ddnode* new_cube = Cudd_bddAnd(manager, cube, var);
    Cudd_Ref(new_cube);
    Cudd_RecursiveDeref(manager, cube);
    Cudd_RecursiveDeref(manager, var);
    cube = new_cube;
}
```

Hint 2. There is already a built-in command "`print_unate`" in ABC. You are welcome to refer to the code, but you have to write your own procedure.

Hint 3: More tips, FAQs, and future updates will be posted on the GitHub wiki page (https://github.com/NTU-ALComLab/LSV-PA/wiki/PA2-2025)

2 [Unateness Checking with SAT] (50%)

Repeat Exercise 1 with all conditions being the same except that the circuit C is in the form of AIG (by commands "`read`" and "`strash`"). Use the SAT solver to check whether the output pin y_k is binate, positive unate, negative unate, in x_i, or independent of x_i. Your procedure should implement the command in the following format.

`lsv_unate_sat` <*k*> <*i*>

where k is a primary output index starting from 0, and i is a primary input index starting from 0.

The output format is the same as in Exercise 1.

Hint 1. You can follow the following steps.

(1) Use `Abc_NtkCreateCone` to extract the cone of y_k.

(2) Use `Abc_NtkToDar` to derive a corresponding AIG circuit.

(3) Use `sat_solver_new` to initialize an SAT solver.

(4) Use `Cnf_Derive` to obtain the corresponding CNF formula C_A, which depends on variables $v_1, \ldots, v_n$.

(5) Use `Cnf_DataWriteIntoSolverInt` to add the CNF to the SAT solver.

(6) Use `Cnf_DataLift` to create another CNF formula C_B that depends on different input variables $v_{n+1}, \ldots, v_{2n}$. Again, add the CNF to the SAT solver.

(7) For each input x_t of the circuit, find its corresponding CNF variables $v_A(t)$ in C_A and $v_B(t)$ in C_B. Set $v_A(t) = v_B(t)\, \forall t \notin \{i\}$. This step can be done by adding the corresponding clauses to the SAT solver.

(8) Use `sat_solver_solve` to solve the SAT problem with some assumptions on variable values.

(9) If y_k is binate in x_i, use `sat_solver_var_value` to obtain the satisfying assignment, which can be used to derive the counterexample.

Hint 2. To use `Abc_NtkToDar` and `Cnf_Derive` functions, you should include the following code.

```
#include "sat/cnf/cnf.h"
extern "C"{
    Aig_Man_t* Abc_NtkToDar( Abc_Ntk_t * pNtk, int fExors, int fRegisters );
}
```

Hint 3. The variable orders in CNF differ from the ones in AIG. For a pointer `pObj` in `Aig_Obj_t*` type, you can use `pCnf->pVarNums[pObj->ID]` to find its variable index in `pCnf`. If the pointer `pObj` is from the original network in `Abc_Obj_t*` type, to make it have the same ID as its counterpart in AIG, make sure you set the last parameter of `Abc_NtkCreateCone` to 1 in step (1), so that all input variables are always included.

Hint 4. You can refer to our GitHub page (https://github.com/NTU-ALComLab/LSV-PA/wiki/Reasoning-with-SAT-solvers) for more details about using SAT solvers in ABC.