CS 6110, Spring 2022, Assignment 1

Given 1/13/22 – Due 1/20/22 by 11:59 pm on Canvas

NAME: Jack Wilburn **UNID: u0999308**

Please try to use this .tex as your answer template, inserting your answers after each question. Ideally you must retain these frameboxs so that I can easily visually locate your answers. But other methods are fine, such as different color, sidebars, etc., so that I can easily locate your answers. In any case, please make my grading easy by starting answers to new questions on fresh pages.

Submission: Please submit a file `UNID_Lastname_Asg1.zip` with your solution on Canvas. See notes on discussions in class intro. Please enclose a PDF writeup and other supportive files (e.g., your Promela code) as a single ZIP.

Example: Your UNID is **u1122333**; then, you would submit:

- A single ZIP file `01122333_Lastname_Asg1.zip`
- When I unzip it, I must find:
  - A file `README.txt` (a README would be helpful to locate the files you are submitting).
  - A file `Asg1.pdf` – this is your main writeup of your answers.
  - A file `Otherfiles.OtherExt` (for all other files, try to name it as indicated).

1. (10 points) Read the first 27 pages of Bradley's book. Also read the material around "CNF-conversion using gates" (AKA Tseitin transformation, CEATL, 18.3.4). (Tseitin or Tseytin is/was a Russian scientist; see `https://en.wikipedia.org/wiki/Tseytin_transformation`.) (This procedure is present in Bradley's book; locate it there.) Why is this Make a glossary of concepts covered such as satisfiability, validity, contradiction, equisat, etc. Study the truth-table based (1.3.1) and semantic argument-based (1.3.2) methods. Summarize all these in neat bullets in your PDF answer. Try to create your initial answer as a cheat-sheet of about a page. Later transplant it to your own place (say, a GDoc) and maintain these concepts. (We could merge them one day perhaps.)

- propositional logic (PL)
- first-order logic (FOL)
- propositional calculus
- predicate calculus
- propositional variables
- Logical connectives/Boolean connectives
- unary vs binary operators
- antecedent and consequent
- atom: true/false
- literal: atom or not atom
- interpretation (assignment of bools to the variables)
- Truth table: a table showing the outcomes given various interpretations
- satisfiable: can be made true
- valid: always true
- semantic argument method
- modus ponens
- equivalence: if both directions of implication exist
- implication: same truth tables for all interpretations
- substitution: mapping from formula to formula
- Negation normal form (NNF)
- De Morgan’s Law
- disjunctive normal form: or of ands
- conjunctive normal form: and of ors
- equisatisfiable: F and F are equisatisfiable when F is satisfiable iff F is satisfiable
- Encodings

2. (30 pts) The formula

$$a \cdot b + c$$

is given to you. (Note that in the Python BDD the syntax is different.) Convert this formula to an equisat formula using Tseitin's transformation, following the procedure in CEATL. Call the variable you introduce at the output of the "and gate" as p and call the final output z. Now, make a copy of `BDD.ipynb` found inside `pbl/` of Jove. (Jove is at `https://github.com/ganeshutah/Jove.git`.) Get rid of all the material in this file before submission (you can keep it to look at it while developing your solution). Then just have one title page "**Understanding Equisat Versus Equivalence.**" Then have just two code cells:

- A code cell

```
EquiSat = '''
Var_Order : a,b,c,p,z
fGiven = (c|(a&b))
fTseitin = ...your Tseitin-converted result...
Main_Exp : fGiven OP fTseitin # OP is -> and <- in turn
```

- The second code cell begins with `buildBDDmain` and is to draw the above.

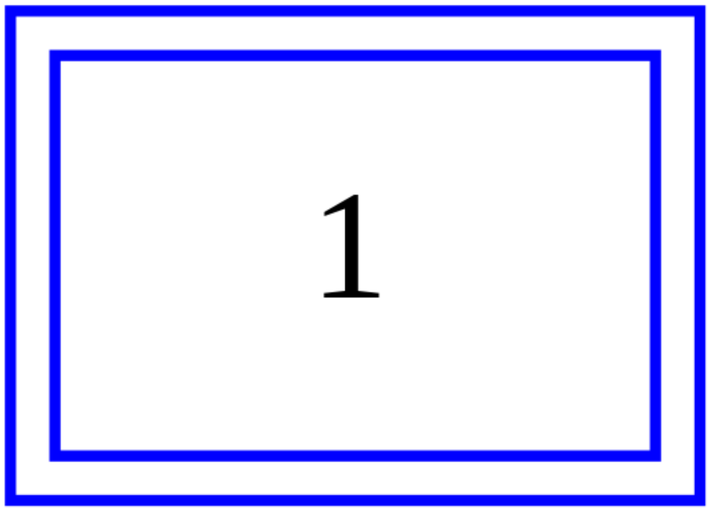
Now answer these questions:

(a) Which case (`->` or `<-`) gave you a "1" node and why?

(Insert framebox here and answer.)

The direction fTseitin => fGiven gave the one node. This is expected since Tseitin SAT implies that the given is SAT. Essentially this shows that it's equisat, not equivalent.

(b) Within the case where you did not get a "0" node, insert that BDD which was non-0 here.



(c) for all the paths to the “0” node, explain why that path exists (this is where equisat differed from equivalence). Write out your answer in neat bulletted steps per path.

Path 1:
●$a = 1$
●$b = 1$
●$p = 0$

Path 2:
●$a = 1$
●$b = 1$
●$p = 1$
●$z = 0$

Path 3:
●$a = 1$
●$b = 0$
●$c = 1$
●$p = 1$

Path 4:
●$a = 1$
●$b = 0$
●$c = 1$
●$p = 0$
●$z = 0$

Path 5:
●$a = 0$
●$c = 1$
●$p = 1$

Path 6:
●$a = 0$
●$c = 1$
●$p = 0$
●$z = 0$

3. (30 pts) First, begin reading Ben-Ari's SPIN book and get some practice following the commands there.

   Run the program in CEATL, Exercises 21.4 (Page 396) on "Bubble sorting," and see how the bug is discovered (the "sortedness assertion" fails). Run this code under SPIN, listing the commands and flags you used. Insert your run-results in a framebox.

   Fully explain all uses of nondeterminism in this Promela model. Fully explain also the use of data-abstraction (i.e., we got away with using "0" and "1" in sorting; is that representative-enough?) Write a good para arguing that (modulo modeling-errors which are assumed not to be there) this model-checking ended-up verifying the sorting for the size you considered. Can you extend your argument to say that this verification is good for arrays of *any* size? Write out the bullets of answers.

   (Make my life easy; write clear concise bulleted arguments of 1-2 pages.)

   I ran this program in spin on the command line as (see first line below for invocation). This puts an upper limit on the number of iterations (1 million, way overkill for this, but a useful limit for other programs) and compiles and runs the code, all in one.
   Here are the results:

```
[jwilburn@nzxtarch homework1]$ spin -u1000000 -run bubble_sort.pml
pan:1: assertion violated 0 (at depth 11)
pan: wrote bubble_sort.pml.trail

(Spin Version 6.5.2 -- 6 December 2019)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 20 byte, depth reached 20, errors: 1
       44 states, stored
        7 states, matched
       51 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.002       equivalent memory usage for states (stored*(State-vector + overhead))
    0.287       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage



pan: elapsed time 0 seconds
```

Nondeterminism:
- Initializing the array

Data abstraction:
- Use 1 and 0 for sorted or not
- Good enough to do 0 or 1 for sorted, since each item has it’s correct place. It’s a binary outcome.
- Verifying the sorting:

Yes, this model did verify the sorting, since the the data abstraction holds and the model a direct translation of the pseudo-code written in the text. It ran the exact same steps as detailed and failed.
- Extension:

Yes, since the model failed on this small array, it has the potential to fail on arrays of any size.

4. (40 pts) Figure 21.2 gives you one version of how “system automata” and “property automata” are used—this matches my video-recording of Jan 11th. Run this example under SPIN, and produce a violation trace showing that the never automaton “accepts.” This reveals a liveness violation.

```
    <<<<<START OF CYCLE>>>>>
MSC: ~G line 56
368:    proc  - (never_0:1) dining-philosophers.pml:56 (state 7)       [(!(progress))]

Never claim moves to line 56    [(!(progress))]
369:    proc  5 (phil:1) dining-philosophers.pml:20 (state -)   [values: 4!release]
369:    proc  5 (phil:1) dining-philosophers.pml:20 (state 18)  [lf!release]

370:    proc  4 (fork:1) dining-philosophers.pml:30 (state -)   [values: 4?release]
370:    proc  4 (fork:1) dining-philosophers.pml:30 (state 5)   [rp?release]

371:    proc  - (never_0:1) dining-philosophers.pml:56 (state 7)       [(!(progress))]

372:    proc  5 (phil:1) dining-philosophers.pml:7 (state -)    [values: 4!are_you_free]
372:    proc  5 (phil:1) dining-philosophers.pml:7 (state 1)    [lf!are_you_free]

373:    proc  4 (fork:1) dining-philosophers.pml:27 (state -)   [values: 4?are_you_free]
373:    proc  4 (fork:1) dining-philosophers.pml:27 (state 1)   [rp?are_you_free]

374:    proc  - (never_0:1) dining-philosophers.pml:56 (state 7)       [(!(progress))]

375:    proc  4 (fork:1) dining-philosophers.pml:27 (state -)   [values: 4!yes]
375:    proc  4 (fork:1) dining-philosophers.pml:27 (state 2)   [rp!yes]

376:    proc  5 (phil:1) dining-philosophers.pml:9 (state -)    [values: 4?yes]
376:    proc  5 (phil:1) dining-philosophers.pml:9 (state 2)    [lf?yes]

377:    proc  - (never_0:1) dining-philosophers.pml:56 (state 7)       [(!(progress))]

378:    proc  5 (phil:1) dining-philosophers.pml:16 (state -)   [values: 5!are_you_free]
378:    proc  5 (phil:1) dining-philosophers.pml:16 (state 10)  [rf!are_you_free]

379:    proc  6 (fork:1) dining-philosophers.pml:29 (state -)   [values: 5?are_you_free]
379:    proc  6 (fork:1) dining-philosophers.pml:29 (state 3)   [lp?are_you_free]

380:    proc  - (never_0:1) dining-philosophers.pml:56 (state 7)       [(!(progress))]

381:    proc  6 (fork:1) dining-philosophers.pml:29 (state -)   [values: 5!no]
381:    proc  6 (fork:1) dining-philosophers.pml:29 (state 4)   [lp!no]

382:    proc  5 (phil:1) dining-philosophers.pml:20 (state -)   [values: 5?no]
382:    proc  5 (phil:1) dining-philosophers.pml:20 (state 17)  [rf?no]
```

This trace shows that the automaton gets stuck in a loop (the never acceptance loop), and is unable to progress.

Diagramming, and Running SPIN on a terminal

You must attempt to draw message-sequence charts on a notepad or ipad to debug effectively. Please submit these with the assignment. The URL `http://spinroot.com/spin/Man/Spin.html` gives you info on SPIN's flags. There is more documentation on the **spinroot** page.

```
spin -a yourfile.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 66796

(Once the pan binary is generated),

GET pan HELP AS FOLLOWS:

[ganesh@thinmac Examples]$ ./pan --help
saw option --
Spin Version 6.4.5 -- 1 January 2016
Valid Options are:
-a,-l,-f  -> are disabled by -DSAFETY
-A  ignore assert() violations
-b  consider it an error to exceed the depth-limit
-cN stop at Nth error (defaults to -c1)
-D  print state tables in dot-format and stop
-d  print state tables and stop
-e  create trails for all errors
-E  ignore invalid end states
-hN use different hash-seed N:0..499 (defaults to -h0)
-hash generate a random hash-polynomial for -h0 (see also -rhash)
      using a seed set with -RSn (default 12345)
-i  search for shortest path to error
-I  like -i, but approximate and faster
-J  reverse eval order of nested unlesses
-mN max depth N steps (default=10k)
-n  no listing of unreached states
-QN set time-limit on execution of N minutes
-q  require empty chans in valid end states
-r  read and execute trail - can add -v,-n,-PN,-g,-C
-r trailfilename  read and execute trail in file
-rN read and execute N-th error trail
-C  read and execute trail - columnated output (can add -v,-n)
-r -PN read and execute trail - restrict trail output to proc N
-g  read and execute trail + msc gui support
-S  silent replay: only user defined printfs show
-RSn use randomization seed n
-rhash use random hash-polynomial and randomly choose -p_rotateN, -p_permute, or p_reverse
-T  create trail files in read-only mode
-t_reverse  reverse order in which transitions are explored
-tsuf replace .trail with .suf on trailfiles
-V  print SPIN version number
-v  verbose -- filenames in unreached state listing
-wN hashtable of 2^N entries (defaults to -w24)
-x  do not overwrite an existing trail file

options -r, -C, -PN, -g, and -S can optionally be followed by
a filename argument, as in '-r filename', naming the trailfile
        [ganesh@thinmac Examples]$


== One tries to catch bugs at the most shallow depth ==
== SPIN also has a BFS mode and also a depth minimization mode ==

[ganesh@thinmac Examples]$ ./pan -m10000  <== search depth
```

```
=== LOOK AT HOW ERRORS ARE SUMMARIZED AND REPORTED ===
=== You MUST read these bugs and warnings carefully! ===
=== Also read the statistics carefully! ===

pan:1: invalid end state (at depth 20)
pan: wrote atomicphil.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim          - (not selected)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid end states +

State-vector 108 byte, depth reached 23, errors: 1        <==
       12 states, stored
        2 states, matched
       14 transitions (= stored+matched)
        5 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.002 equivalent memory usage for states (stored*(State-vector + overhead))
    0.291 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  128.730 total actual memory usage

== THIS IS ERROR-TRAIL SIMULATON BELOW - understand all these flags! ==

[ganesh@thinmac Examples]$ spin -p -r -s -c  -t atomicphil.pml <-- just an example
proc 0 = :init:
using statement merging
Starting phil with pid 1 <-- just an example showing what to expect, is below
proc 1 = phil
  1: proc  0 (:init::1) atomicphil.pml:28 (state 1) [(run phil(p0,v0,p2,v2))]
Starting phil with pid 2
proc 2 = phil
  2: proc  0 (:init::1) atomicphil.pml:32 (state 2) [(run phil(p1,v1,p0,v0))]
Starting phil with pid 3
proc 3 = phil
  3: proc  0 (:init::1) atomicphil.pml:36 (state 3) [(run phil(p2,v2,p1,v1))]
Starting fork with pid 4
proc 4 = fork
  4: proc  0 (:init::1) atomicphil.pml:39 (state 4) [(run fork(p0,v0))]
Starting fork with pid 5
proc 5 = fork
  5: proc  0 (:init::1) atomicphil.pml:41 (state 5) [(run fork(p1,v1))]
Starting fork with pid 6
proc 6 = fork
  6: proc  0 (:init::1) atomicphil.pml:43 (state 6) [(run fork(p2,v2))]
q\p   0   1   2   3   4   5   6
  5   .   .   .   lfp!0
  7: proc  3 (phil:1) atomicphil.pml:4 (state 1) [lfp!0]
  5   .   .   .   .   .   .   p?0
  8: proc  6 (fork:1) atomicphil.pml:10 (state 1) [p?0]
  3   .   .   .   rfp!0
  9: proc  3 (phil:1) atomicphil.pml:4 (state 2) [rfp!0]
  3   .   .   .   .   .   p?0
 10: proc  5 (fork:1) atomicphil.pml:10 (state 1) [p?0]
                  Eating 11: proc  3 (phil:1) atomicphil.pml:4 (state 3) [printf('Eating')]
  6   .   .   .   lfv!0
 12: proc  3 (phil:1) atomicphil.pml:4 (state 4) [lfv!0]
  6   .   .   .   .   .   .   v?0
 13: proc  6 (fork:1) atomicphil.pml:11 (state 2) [v?0]
  1   .   lfp!0
 14: proc  1 (phil:1) atomicphil.pml:4 (state 1) [lfp!0]
  1   .   .   .   .   p?0
 15: proc  4 (fork:1) atomicphil.pml:10 (state 1) [p?0]
  4   .   .   .   rfv!0
 16: proc  3 (phil:1) atomicphil.pml:4 (state 5) [rfv!0]
  4   .   .   .   .   .   v?0
 17: proc  5 (fork:1) atomicphil.pml:11 (state 2) [v?0]
  5   .   .   .   lfp!0
 18: proc  3 (phil:1) atomicphil.pml:4 (state 1) [lfp!0]
  5   .   .   .   .   .   .   p?0
 19: proc  6 (fork:1) atomicphil.pml:10 (state 1) [p?0]
  3   .   .   lfp!0
 20: proc  2 (phil:1) atomicphil.pml:4 (state 1) [lfp!0]
  3   .   .   .   .   .   p?0
 21: proc  5 (fork:1) atomicphil.pml:10 (state 1) [p?0]
spin: trail ends after 21 steps
-------------
final state:
-------------
#processes: 7
 21: proc  6 (fork:1) atomicphil.pml:11 (state 2)
 21: proc  5 (fork:1) atomicphil.pml:11 (state 2)
 21: proc  4 (fork:1) atomicphil.pml:11 (state 2)
 21: proc  3 (phil:1) atomicphil.pml:4 (state 2)
 21: proc  2 (phil:1) atomicphil.pml:4 (state 2)
 21: proc  1 (phil:1) atomicphil.pml:4 (state 2)
 21: proc  0 (:init::1) atomicphil.pml:46 (state 8) <valid end state>
7 processes created

=== NOW FIND OUT WHY THE ABOVE IS DESCRIBING A DEADLOCK ===
=== Drawing a diagram will help                         ===
```