CS 6110 — Assignment 2
Gavin Gray u1040250

1. (25 pts) The execution of a set of Promela proctypes defines a *Kripke Structure* which is defined in Figure 22.1 (see the waveforms there) of Book-1: it is a way to specify how variables change over time. The proctype below defines the most general Kripke structure over two bits: all possible `x,y` waveforms are defined by this.

```
bit x,y; // initialized to 0
active proctype test() {  do :: x=!x :: y=!y od }
```

We are defining such a general Kripke structure to check whether Justice implies Compassion in a general sense or the other way (we know—or at least we believe—that Compassion implies Justice). Your tasks are below.

Compile a `never` automaton by giving the command

```
    spin -f " ! ((<>[]x -> []<>y) -> ([]<>x -> []<>y)) "
```

where `(<>[]x -> []<>y)` is justice and `([]<>x -> []<>y)` is compassion. Then insert this never automaton into the same file as the proctype and check for violations (Büchi acceptance happens, and you obtain an error trace). If you obtain an error trace, then argue that it was expected. Then explain the trace and conclude why it makes sense. Else (if no trace), then argue that it is the expected behavior.

Example: If you obtained the trace

```
Never claim moves to line 14 [((!(x)&&!(y)))]
Never claim moves to line 80 [((!(y)&&x))]
Never claim moves to line 46 [((!(x)&&!(y)))]
  <<<<<START OF CYCLE>>>>>
Never claim moves to line 29 [((!(y)&&x))]
Never claim moves to line 88 [((!(x)&&!(y)))]
Never claim moves to line 64 [((!(y)&&x))]
Never claim moves to line 35 [((!(x)&&!(y)))]
spin: trail ends after 14 steps
```

then you can say that the Kripke structure is as follows:

```
00 -> 10 -> 00 -[H:Head of Cycle] -> 10 -> 00 -> 10 -> 00 -> [H]
```

Note: We use the notation "10" to say `x=1` and `y=0`, and so on for the other pairs.

While I sort of asked the questions above, I'll be grading your answers to these specific questions below (that are now being asked in one cohesive way): For definiteness (and ease of grading), please answer the following with respect to the Kripke structure whose "ASCII drawing" I present above. (To clarify, please do obtain your own accepting trace and see if it matches mine. Let me know if it does not match—for some reason. But then, for the ease of grading, answer the questions below w.r.t. the Kripke structure I presented above.)

(a) (5 pts) Observe that the overall property checked is of the form `((a -> b) -> (c -> d))` where `a` is `<>[]x` (Eventually-Henceforth `x`). Is '`a`' true in this Kripke structure?

Student Response: For the following responses I will refer to the below error trace given by Spin:

```
Never claim moves to line 18 [((!(x)&&!(y)))]
Never claim moves to line 84 [((!(y)&&x))]
Never claim moves to line 50 [((!(x)&&!(y)))]
    <<<<<START OF CYCLE>>>>>
Never claim moves to line 33 [((!(y)&&x))]
Never claim moves to line 92 [((!(x)&&!(y)))]
Never claim moves to line 68 [((!(y)&&x))]
Never claim moves to line 39 [((!(x)&&!(y)))]
```

From this trace we can derive an accepting cycle of the Kripke structure which has the form:

```
HEAD: 10 -> 00 -> 10 -> 00
```

Given the statement `((<>[]x -> []<>y) -> ([]<>x -> []<>y))` it is trivial to see that it is of the form `(a -> b) -> (c -> d)`.

The variable 'a' in the statement corresponds to the statement: `<>[]x`. In the above cycle we can see that there does not exist a point in time t_i such that in all future moments t_{i+1} t is true. Therefore, we can say that 'a' is false.

(b) (5 pts) '`b`' and '`d`' are `[]<>y` (Infinitely-Often `y`). Are they true in this Kripke structure?

Student Response: In this Kripke structure we can see that y is false in every state. Therefore, we can say that `[]<>y` is false and 'b' and 'd' are consequently false.

(c) (5 pts) 'c' is [] <> x (Infinitely-Often x). Is it true in this Kripke structure?

Student Response: From the above cycle we can see that x is flickering between 0 and 1 on every state change. From this, we can assert that `[]<>x` is true. Therefore, 'c' is true.

(d) (5 pts) Is the overall property true? Explain in detail why it is true! The overall property is true if the never automaton generated by the above spin -f command does not produce an accepting cycle.

Student Response: Again, we can say that the overall property of the never automata is `!((a -> b) -> (c -> d))`. Substituting the logical values for the above sections we can say that this property simplifies as follows:

```
!((false -> false) -> (true -> false))
!( true -> false )
!(false)
true
```

(e) (5 pts) Was the overall property falsified (as evidenced by model checking with the never automaton being accepted)? Explain in detail why it is false!

Yes the never automaton entered into an accepting state indicating that the property `((<>[]x -> []<>y) -> ([]<>x -> []<>y))` was falsified. This can be checked by running the program `just_impl_comp.pml` included in the zip archive. This was as expected and it is intuitive as to why this is the case.

We can observe that the predicate `<>[]x` is stronger than `[]<>x`. If the first (stronger) predicate implies `[]<>y`, it is not necessarily true that the second (weaker) predicate implies the same.

2. (10 pts) Now, verify the proctype via this `never` automaton:

```
spin -f " ! (([]<>x -> []<>y) -> (<>[]x -> []<>y)) "
```

Did you get a violation and was that expected? Why? Explain in detail.

Student Response: When running the program (included in the zip as `comp_impl_just.pml`) we can see that under spin the never automaton is never accepted, and therefore, no errors are given. This is expected and can be easily explained.

The predicate `<>[]x` is much stronger than `[]<>x`. Therefore, if the weaker predicate implies that `[]<>y` then it must also be true that the stronger predicate implies the same.

3. (10 pts) Convert the formula `((a->b)->(c->d))` (call it `Orig`) directly without the Tseitin transformations, but by converting `->` using the "Not-a-or-b" rule, and then use distributive law till you get the final CNF (call it `Fin`). Show using the BDD tool that `Orig` and `Fin` are logically equivalent. (Please do not simplify the CNF. Present a suitably shrunken "1" BDD node screenshot as your answer.) Here is the outline of how you will go about:

```
Var_Order: a b c d
Orig = ((a->b)->(c->d))
Fin = ...what you obtain after conversion...
Main_Exp: Orig <-> Fin
```

Student Response: The following transformation can be made from Orig to Fin using conditional laws:

$$Orig = ((a \rightarrow b) \rightarrow (c \rightarrow d)) \quad (1)$$

$$\neg(a \wedge \neg b) \rightarrow \neg(c \wedge \neg d) \quad (2)$$

$$\neg((\neg(a \wedge \neg b)) \wedge \neg(\neg(c \wedge \neg d))) \quad (3)$$

$$\neg((\neg a \vee b)) \wedge (c \wedge \neg d))) \quad (4)$$

$$(a \wedge \neg b) \vee \neg c \vee d \quad (5)$$

$$((a \vee \neg c) \wedge (\neg b \vee \neg c)) \vee d \quad (6)$$

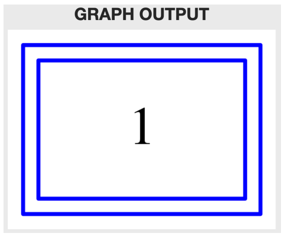
$$(d \vee (a \vee \neg c)) \wedge (d \vee (\neg b \vee \neg c)) \quad (7)$$

$$Fin = (a \vee \neg c \vee d) \wedge (\neg b \vee \neg c \vee d) \quad (8)$$

The following text can by pasted into the BDD tool:

```
Var_Order : a b c d
Orig = ((a -> b) -> (c -> d))
Fin = (a | ~c | d) & (~b | ~c | d)
Main_Exp : Orig <=> Fin
```

This program gives us the expected "1" BDD node as seen here:



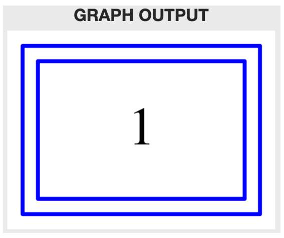
4. (15 pts) Now, take `Orig` and build a BDD for it. Obtain the CNF for it by reading the paths leading to the "0" node as I explained in Lecture-3. Specifically, by way of a *hypothetical example*,

   - Suppose there is a path labeled by `a` and `!b`
   - Suppose there is a path labeled by `!a` and `c`

   then the CNF must be `(!a+b).(a+!c)`. Call the CNF you obtain `FinB` (standing for "Final as per BDD"). Show using the BDD tool that `Orig` and `FinB` are logically equivalent.

   Student Response: Reading the path to the "0" node in the BDD leads to the following CNF: $(a \vee \neg c \vee d) \wedge (\neg a \vee \neg b \vee \neg c \vee d)$ which is the same as the one obtained in the previous question (maybe I misunderstood the previous question). Again, putting in the command:

```
Var_Order : a b c d
Orig = ((a -> b) -> (c -> d))
Finb = (a | ~c | d) & (~b | ~c | d)
Main_Exp : Orig <=> Finb
```

   outputs the expected "1" BDD node as seen here: 

GRAPH OUTPUT

1

5. (20 pts) Now, take `Orig` and write it as `Temp` below.

```
Temp = ( (a & !b) | (!c | d) )
```

Then apply Tseitin transformation to `Temp` by the following steps:

- Introduce a variable `p` as follows: `p = (a & !b)`
- Turn `=` into a conjunction of `p -> (a & !b)` and `(a & !b) -> p`. Convert these all the way to CNF.
- Introduce a variable `q` as follows: `q = (!c | d) )`. Then turn it into a conjunction of implications as above, and obtain a CNF.
- Finally, introduce a variable `z` and set `z = (p | q)`, and convert to a CNF.
- You will obtain a collection of clauses—call them `T1` through `T9`—that are conjoined. The final clause you need is the "output" `z`. A full derivation is in Section 18.3.4 of Book-1. The final CNF is as shown just before Section 18.3.5 of Book-1. Call this final CNF conjunction `FinT` (for "Final Tseitin"). Now your task is to compare `Orig` and `FinT`.

  Here are the questions I'll be grading:

  1. (15 pts) Obtain the Tseitin transformation as requested above.
  2. (5 pts) Which is more general: `Orig` or `FinT`? Formula `A` is more general than formula `B` if `(A->B)` is valid. Do this validity checking using BDDs.

Student Response: We can first simplify the original formula `( (a -> b) -> (c -> d) )` as follows:

$$((a \to b) \to (c \to d)) \tag{9}$$

$$((!a|b) \to (!c|d)) \tag{10}$$

$$(!(!a|b)|(!c|d)) \tag{11}$$

$$((a\&!b)|(!c|d)) \tag{12}$$

Given this formula we can follow the Tseitin Transformation as follows:

$$\text{Step 1} \tag{13}$$

$$p = (a\&!b) \tag{14}$$

$$p \rightarrow (a\&!b) \equiv !p|(a\&!b) \equiv (!p|a)\&(!p|!b) \tag{15}$$

$$(a\&!b) \rightarrow p \equiv (!a|b)|p \equiv (p|!a|b) \tag{16}$$

$$\text{Step 2} \tag{17}$$

$$q = (!c|d) \tag{18}$$

$$q \rightarrow (!c|d) \equiv !q|(!c|d) \equiv (!q|!c|d) \tag{19}$$

$$(!c|d) \rightarrow q \equiv (c\&!d)|q \equiv (q|c)\&(q|!d) \tag{20}$$

$$\text{Step 3} \tag{21}$$

$$z = (p|q) \tag{22}$$

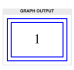
$$z \rightarrow (p|q) \equiv !z|(p|q) \equiv (!z|p|q) \tag{23}$$

$$(p|q) \rightarrow z \equiv (!p\&!q)|z \equiv (z|!p)\&(z|!q) \tag{24}$$

$$\tag{25}$$

From the above transformation we can turn this into an input for the BDD solver as follows:

```
Var_Order : a b c d p q z
Orig = ((a -> b) -> (c -> d))
Fin = (a | !c | d) & (!b | !c | d)
T1 = (!p | a)
T2 = (!p | !b)
T3 = (!a | b | p)
T4 = (!q | !c | d)
T5 = (c | q)
T6 = (!d | q)
T7 = (!z | p | q)
T8 = (!p | z)
T9 = (!q | z)
FinT = (T1 & T2 & T3 & T4 & T5 & T6 & T7 & T8 & T9 & z)
Main_Exp : FinT -> Orig
```

The expression `Main_Exp : FinT -> Orig` shows that in fact `FinT` is more general than Orig because the output of this program shows 

6. (5 pts) Install Z3 (`pip install z3-solver`). Encode the SEND MORE MONEY problem. The best link I know is `https://ericpony.github.io/z3py-tutorial/guide-examples.htm` but it is also discussed in the book "SAT and SMT By Example" by Yurichev (linked in the table on the Canvas Home Page). Let the most significant digit (MSD) constraint be that it be non-zero (full encoding in Lec3's slides). Try to obtain more than one solution using ideas such as in `https://stackoverflow.com/questions/13395391/z3-finding-all-satisfying-models` (there could be others) that rule out previously seen solutions. If more than one solution isn't obtainable, then allow the MSD to be 0, and then try.

   Student Response: You will find the related code at the end of this PDF as well as included in the zip archive for the assignment (`send_money.py`).

   When the MSD is constrainted to > 0 there is only one unique solution which is as follows:

```
S = 9                      9567
E = 5                    + 1085
N = 6                    ------
D = 7                     10652
M = 1
O = 0
R = 8
Y = 2
```

   However, if we allow the MSD to be ≥ 0 then there are 25 possible solutions (which I will not list here).

7. REMOVED

8. (10 pts) **(This entire problem calls for a traditional manual math proof typeset in Latex; no tool usage needed.)**

   Some Definitions:

   - $A =_{equisat} B$ if and only if $\exists\sigma :\ \sigma \models A \Leftrightarrow \sigma \models B$.
   - Here, σ is a variable assignment (a setting of Boolean variables to true/false), and "$\sigma \models X$" means "formula X is true under σ."

   Suppose we are given the DNF formula `Orig = (ab+cd)`. Suppose someone claims that the following is EquiSAT to it:
   `EquiMaybe = (!a+!b+p)(!p+a)(!p+b)(!c+!d+q)(!q+c)(!q+d)`

   1. (1 pts) What method are they presumably following to obtain the plausible `EquiMaybe` above? One sentence.
   2. (6 pts) Prove that it is not equisat. Write all the steps neatly.

      HINTS: To show that A is not equisat B, one approach is this. Do a case analysis as follows:

      (a) Assume A is SAT, and show that for the same variable assignment, B is also SAT.
      (b) Assume A is UNSAT, and show that for the same variable assignment, B could be SAT.
   3. (3 pts) Add the missing clauses in this conversion, and now argue that it is equisat.
      HINT: In the case analysis hinted above, you should be able to prove that when A is UNSAT, then B must also emerge UNSAT.

   Student Response: It can be noted that the method followed to obtain `EquiMaybe` is the Tseitin Transformation. This is evident by the expansion of the expression `(a&b)` being expanded out with the introduction of a new variable p and similarly with `(c&d)` and q.

   Let's say that Orig is SAT, and give the variable assignment $\sigma_0 =$ `{a & b & c & d = 1}`. Under this same assignment, we can see that `EquiMaybe` is also SAT regardless of the assignment to p and q. Therefore, $\sigma_0 \models Orig$ and $\sigma_0 \models EquiMaybe$

   Now, let's assume that Orig is UNSAT and give the variables the following assignment $\sigma_1 =$`{ a=0,b=1,c=1,d=0 }`. For this assignment, `EquiMaybe` is still SAT, therefore, we can say that $Orig \neq_{equisat} EquiMaybe$.

   If `EquiMaybe` was with the Tseitin Transformation we can finish this transformation by adding in the missing clauses to get the formula

```
EquiSAT = (a  | !p)
        & (b  | !p)
        & (!a | !b | p)
        & (c  | !q)
        & (d  | !q)
        & (!c | !d | q)
        & (!z | p  | q)
        & (!p | z)
        & (!q | z)
        & z
```

Now, if you take the variable ordering σ_1 we can see that both Orig and EquiSAT are forced to be UNSAT. Similarly, if you take the variable ordering σ_0 you can see that both must come out SAT [1].

[1] I will stop here because I am not entirely sure how a formal proof of equisatisfiability should look, my apologies.