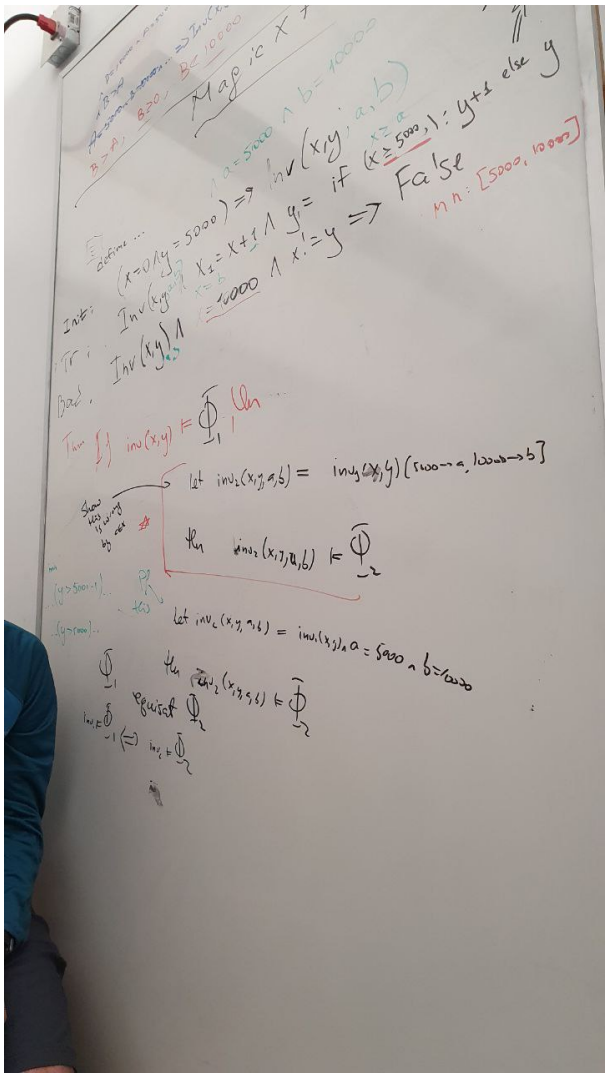
Prerequisite: During last weeks of summer I tried to show my technique to my advisor's collegue. During the conversation I got a task to prove 2 statement that will prove my technique. The picture of notes from this conversation I attach here. Everything on the blackboard was duplicated to this file. As an example was taken the file `s_split_01.smt2`



Proofs for equisat/non-equisat for variable substitution technique used in magicXform

In the file represented original transition system (ts0) and transformed transition system (ts1)

In [ ]:
```
import sys
sys.path.insert(1, '/Users/ekvashyn/Code/spacer-on-jupyter/src/')
from spacer_tutorial import *
import z3
z3.set_param(proof=True)
z3.set_param(model=True)
z3.set_html_mode(True)
```

In [ ]:
```
def mk_ts0():
    T = Ts('Ts0')
    x, x_out = T.add_var(z3.IntSort(), name='x')
    y, y_out = T.add_var(z3.IntSort(), name='y')
    T.Init = z3.And(x == 0, y == 5000)
    T.Tr = z3.And(x_out == x + 1, y_out == z3.If(x >= 5000, y+1, y))
    T.Bad = z3.And(x == 10000, x != y)
    return T

ts0 = mk_ts0()
HtmlStr(ts0)
```

Out[ ]:
Transition System: Ts0

Init: x = 0 ∧ y = 5000

Bad: x = 10000 ∧ x ≠ y

Tr: x' = x + 1 ∧ y' = If(x ≥ 5000, y + 1, y)

In [ ]:
```
def mk_ts1():
    T = Ts('Ts1')
    x, x_out = T.add_var(z3.IntSort(), name='x')
    y, y_out = T.add_var(z3.IntSort(), name='y')
    a, a_out = T.add_var(z3.IntSort(), name='a')
    b, b_out = T.add_var(z3.IntSort(), name='b')
    T.Init = z3.And(a == 5000, b == 10000, x == 0, y == a)
    T.Tr = z3.And(x_out == x + 1, y_out == z3.If(x >= a, y+1, y), a_out == a
    T.Bad = z3.And(x == b, x != y)
    return T

ts1 = mk_ts1()
HtmlStr(ts1)
```

Out[ ]:
Transition System: Ts1

Init: a = 5000 ∧ b = 10000 ∧ x = 0 ∧ y = a

Bad: x = b ∧ x ≠ y

Tr: x' = x + 1 ∧ y' = If(x ≥ a, y + 1, y) ∧ a' = a ∧ b' = b

In [ ]:
```
def vc_gen(T):
    '''Verification Condition (VC) for an Inductive Invariant'''
    Inv = z3.Function('Inv', *(T.sig() + [z3.BoolSort()]))

    InvPre = Inv(*T.pre_vars())
    InvPost = Inv(*T.post_vars())

    all_vars = T.all()
    vc_init = z3.ForAll(all_vars, z3.Implies(T.Init, InvPre))
    vc_ind = z3.ForAll(all_vars, z3.Implies(z3.And(InvPre, T.Tr), InvPost))
    vc_bad = z3.ForAll(all_vars, z3.Implies(z3.And(InvPre, T.Bad), z3.BoolVa
    return [vc_init, vc_ind, vc_bad], InvPre
```

In [ ]:
```
vc0, inv0 = vc_gen(ts0)
vc1, inv1 = vc_gen(ts1)
```

In [ ]:
```
chc_to_str(vc0)
```

Out[ ]: ∀x, y, x', y' : x = 0 ∧ y = 5000 ⇒ Inv(x, y)

∀x, y, x', y' :
Inv(x, y) ∧ x' = x + 1 ∧ y' = If(x ≥ 5000, y + 1, y) ⇒
Inv(x', y')

∀x, y, x', y' : Inv(x, y) ∧ x = 10000 ∧ x ≠ y ⇒ False

In [ ]:
```
chc_to_str(vc1)
```

Out[ ]: ∀x, y, a, b, x', y', a', b' :
a = 5000 ∧ b = 10000 ∧ x = 0 ∧ y = a ⇒ Inv(x, y, a, b)

∀x, y, a, b, x', y', a', b' :
Inv(x, y, a, b) ∧
x' = x + 1 ∧
y' = If(x ≥ a, y + 1, y) ∧
a' = a ∧
b' = b ⇒
Inv(x', y', a', b')

∀x, y, a, b, x', y', a', b' :
Inv(x, y, a, b) ∧ x = b ∧ x ≠ y ⇒ False

Invariants for those 2 systems locates below

Invariant 1

In [ ]:
```
HtmlStr(inv0)
```

Out[ ]: Inv(x, y)

In [ ]:
```
res0, answer0 = solve_horn(vc0, max_unfold=100)
res0
```

Out[ ]: **sat**

In [ ]:
```
res0
```

Out[ ]: **sat**

In [ ]:
```
answer0
```

Out[ ]: [Inv = [else → (¬(ν1 ≥ 5001) ∨ ¬(ν1 + -1·ν0 ≥ 1)) ∧ ¬(ν1 + -1·ν0 ≤ -1) ∧ ¬(ν1 ≤ 4999)]]

In [ ]:
```
answer0.eval(inv0)
```

Out[ ]: (¬(y ≥ 5001) ∨ ¬(y + -1·x ≥ 1)) ∧ ¬(y + -1·x ≤ -1) ∧ ¬(y ≤ 4999)

Invariant 2

In [ ]:
```
HtmlStr(inv1)
```

Out[ ]: Inv(x, y, a, b)

In [ ]:
```
res1, answer1 = solve_horn(vc1, max_unfold=100)
```

In [ ]:
```
res1
```

Out[ ]: **sat**

In [ ]:
```
answer1
```

Out[ ]: [Inv = [else → (¬(ν0 + -1·ν1 ≤ -1) ∨ ¬(ν0 + -1·ν2 ≥ 0)) ∧ ¬(ν2 + -1·ν1 ≥ 1) ∧ ¬(ν2 + -1·ν3 ≥ 0) ∧ ¬(ν0 + -1·ν1 ≥ 1) ∧ (¬(ν0 + -1·ν2 ≤ -1) ∨ ¬(ν2 + -1·ν1 ≤ -1))]]

In [ ]:
```
answer1.eval(inv1)
```

Out[ ]: (¬(x + -1·y ≤ -1) ∨ ¬(x + -1·a ≥ 0)) ∧ ¬(a + -1·y ≥ 1) ∧ ¬(a + -1·b ≥ 0) ∧ ¬(x + -1·y ≥ 1) ∧ (¬(x + -1·a ≤ -1) ∨ ¬(a + -1·y ≤ -1))

---

1. Provide cx for the statement: inv2(x,y,a,b) = inv(x,y) [5000->a, 10000->b]

- Invariant for the original benchmark is:
  Inv1(x,y) =

(¬(y ≥ 5001) ∨ ¬(y + -1·x ≥ 1)) ∧ ¬(y + -1·x ≤ -1) ∧ ¬(y ≤ 4999) =

(y ≥ 5001) => (y ≥ x + 1) ∧ y ≥ x ∧ y > 4999

- Invariant for the transformed benchmark is:
  Inv2(x,y,a,b) =

(¬(y ≥ 5001) ∨ x ≥ y) ∧ y ≥ a ∧ x ≤ y =

(y > a => x ≥ y) ∧ y ≥ a ∧ x ≤ y

We need to prove that Inv1(x,y)[5000->a, 10000->b] != Inv2(x,y,a,b) and provide a cx

Let's rewrite Inv1(x,y) in such way:

Inv1(x,y) = ((y ≥ 5001 => x>=y) ∧ y ≥ 5000 ∧ x ≤ y)

Inv1(x,y)[5000->a, 10000->b] = ((y ≥ 5001 => x≥y) ∧ y ≥ a ∧ x ≤ y)

Let's to find a cx using z3:

In [ ]:
```
from z3 import *

def check_is_sat(solver):
    if solver.check() == unsat:
        print("Invariant equivalency holds.")
    else:
        print("Invariant equivalency does not hold.")
        m = solver.model()
        print(m)
```

In [ ]:
```
def inv_gen(u,w):
    return And(
    Implies(w >= 5001, u >= w),
    w > 4999,
    u <= w)

a, b, x, y, x_prime, y_prime = Ints('a b x y x_prime y_prime')

init_constraints = And(
    x == 0,
    y == a  # If we will put y == 5000 (or a == 5000 before) the first solve
)

transition_constraints = And(
    x_prime == x + 1,
    y_prime == If(x >= a, y + 1, y) #y_prime == If(x >= 5000, y + 1, y) work
)
bad_state = And(x == b, x_prime != y_prime) #x == 10000 works

invariant_constraints = inv_gen(x,y)
invariant_prime_constraints = inv_gen(x_prime, y_prime)


solver = Solver()
solver.add(init_constraints)
solver.add(Not(invariant_constraints))

print("Check the initialization condition Init => Inv is valid:")
check_is_sat(solver)

solver2 = Solver()
solver2.add(transition_constraints)
solver2.add(invariant_constraints)
solver2.add(Not(invariant_prime_constraints))

print("Check the condition Inv ∧ Tr => Inv` is valid:")
check_is_sat(solver2)

solver3 = Solver()
```

```
solver3.add(invariant_constraints)
solver3.add(bad_state)
print("Check the initialization condition Inv ∧ Bad => False` is valid:")
check_is_sat(solver2)
```

```
Check the initialization condition Init => Inv is valid:
Invariant equivalency does not hold.
[x = 0, a = 5001, y = 5001]
Check the condition Inv ∧ Tr => Inv` is valid:
Invariant equivalency does not hold.
[x = 5000,
 y_prime = 5000,
 x_prime = 5001,
 a = 5001,
 y = 5000]
Check the initialization condition Inv ∧ Bad => False` is valid:
Invariant equivalency does not hold.
[x = 0, y_prime = 5001, x_prime = 1, y = 5000, a = 0]
```

2. Prove the statement:

inv2(x,y,a,b) = inv(x,y) ∧ a = 5000 ∧ b = 10000

- Ts0: `I0=Inv(x,y)`
- Ts1: `I2=Inv2(x,y,a,b)`

We aim to prove that `I0 ≡ I2`

In [ ]:
```
# vars for Ts0
x0, y0, x0_prime, y0_prime = Ints('x0 y0 x0_prime y0_prime')

# vars for Ts1
x1, y1, a1, b1, x1_prime, y1_prime = Ints('x1 y1 a1 b1 x1_prime y1_prime')

# vars for Ts2
x2, y2, a2, b2, x2_prime, y2_prime = Ints('x2 y2 a2 b2 x2_prime y2_prime')

solver = Solver()
solver2 = Solver()

# Define the invariants for Ts0 and Ts1
I0_0 = And(Implies(y0 > 5000, x0 >= y0), y0 >= 5000, x0 <= y0)
I0 = And(Implies(y0 > 5000, x0 + 1 > y0), y0 >= 5000, x0 - 1 < y0)
I1 = And(Implies(y1 >= a1, x1 >= y1), x1 <= y1, y1 >= a1)
I2 = And(Implies(a2-y2 <= -1, x2>y2-1),
        # Not(And(a2 -y2 <= -1, x2 - y2 <= -1)), // this is another variant
         x2 < y2 + 1,
         b2 - a2 >= 5000,
         a2 < y2 +1)

solver.add(Not(Implies(I0, I1)))
solver.add(Not(Implies(I1, I0)))

solver2.add(Not(Implies(I0, I2)))
solver2.add(Not(Implies(I2, I0)))
```

```
print(f"solver:")
check_is_sat(solver)
print()
print(f"solver2:")
check_is_sat(solver2)
print()
print("I0 == I1")
prove(I0 == I1)
print("I0 == I2")
prove(I0 == I2)
```

```
solver:
Invariant equivalency holds.

solver2:
Invariant equivalency holds.

I0 == I1
counterexample
[x0 = 0, a1 = 0, y0 = 5001, y1 = 0, x1 = 0]
I0 == I2
counterexample
[y2 = 0, x0 = 5001, y0 = 5001, a2 = -1, x2 = -1, b2 = 4999]
```

There I tried 2 approaches to prove that invariants equisat, but got 2 different answers. I believe it's not.

Below I tried to fit invariant of a transformed benchmark to initial one and see whether it will take it, but I tried 3 different invariants (because solve_horn returned me >1 solution), but non of them worked. Probably I miss something

In [ ]:
```
a, b, x, y, x_prime, y_prime = Ints('a b x y x_prime y_prime')

init_constraints = And(
    # a == 5000,
    # b == 10000,
    x == 0,
    y == 5000
)

transition_constraints = And(
    x_prime == x + 1,
    y_prime == If(x >= 5000, y + 1, y)
)
bad_state = And(x == 10000, x_prime != y_prime)

I0 = And(Implies(y > 5000, x + 1 > y), y >= 5000, x - 1 < y)
I0_prime = And(Implies(y_prime > 5000, x_prime + 1 > y_prime),
               y_prime >= 5000, x_prime - 1 < y_prime)

I1 = And(Implies(y >= a, x >= y), x <= y, y >= a)
I1_prime = And(Implies(y_prime >= a, x_prime >= y_prime), x_prime <= y_prime

I2 = And(Implies(a-y <= -1, x>y-1),
```

```
        # Not(And(a - y <= -1, x - y <= -1)),
         x < y + 1,
         b - a > 4999,
         a - 1 < y)
I2_prime = And(
    Implies(a-y_prime <= -1, x_prime>y_prime-1),
        # Not(And(a -y_prime <= -1, x_prime - y_prime <= -1)),
         x_prime < y_prime + 1,
         b - a > 4999,
         a - 1 < y_prime)

I3 = And(Or(Not(x - y <= -1), Not(x - a >= 0)),
         Not(a - y >= 1),
         b - a > 4999,
         Not(x - y >= 1),
         Or(Not(x - a <= -1), Not(a - y <= -1)))

I3_prime = And(Or(Not(x_prime - y_prime <= -1), Not(x_prime - a >= 0)),
         Not(a - y_prime >= 1),
         b - a > 4999,
         Not(x_prime - y_prime >= 1),
         Or(Not(x_prime - a <= -1), Not(a - y_prime <= -1)))

# (¬(x + -1·y ≤ -1) ∨ ¬(x + -1·a ≥ 0)) ∧
# ¬(a + -1·y ≥ 1) ∧ ¬(a + -1·b ≥ 0) ∧
# ¬(x + -1·y ≥ 1) ∧
# (¬(x + -1·a ≤ -1) ∨ ¬(a + -1·y ≤ -1))

solver = Solver()
solver.add(init_constraints)
solver.add(Not(I3))

print("Check the initialization condition Init => Inv is valid:")
check_is_sat(solver)

solver2 = Solver()
solver2.add(transition_constraints)
solver2.add(I3)
solver2.add(Not(I3_prime))

print("Check the condition Inv ∧ Tr => Inv` is valid:")
check_is_sat(solver2)

solver3 = Solver()
solver3.add(I3)
solver3.add(bad_state)
print("Check the initialization condition Inv ∧ Bad => False` is valid:")
check_is_sat(solver2)
```

```
Check the initialization condition Init => Inv is valid:
Invariant equivalency does not hold.
[y = 5000, b = 5001, x = 0, a = 1]
Check the condition Inv ∧ Tr => Inv` is valid:
Invariant equivalency does not hold.
[a = 5002,
 x = 5000,
 y_prime = 5003,
 b = 10002,
 x_prime = 5001,
 y = 5002]
Check the initialization condition Inv ∧ Bad => False` is valid:
Invariant equivalency does not hold.
[a = 5002,
 x = 5000,
 y_prime = 5003,
 y = 5002,
 x_prime = 5001,
 b = 10002]
```