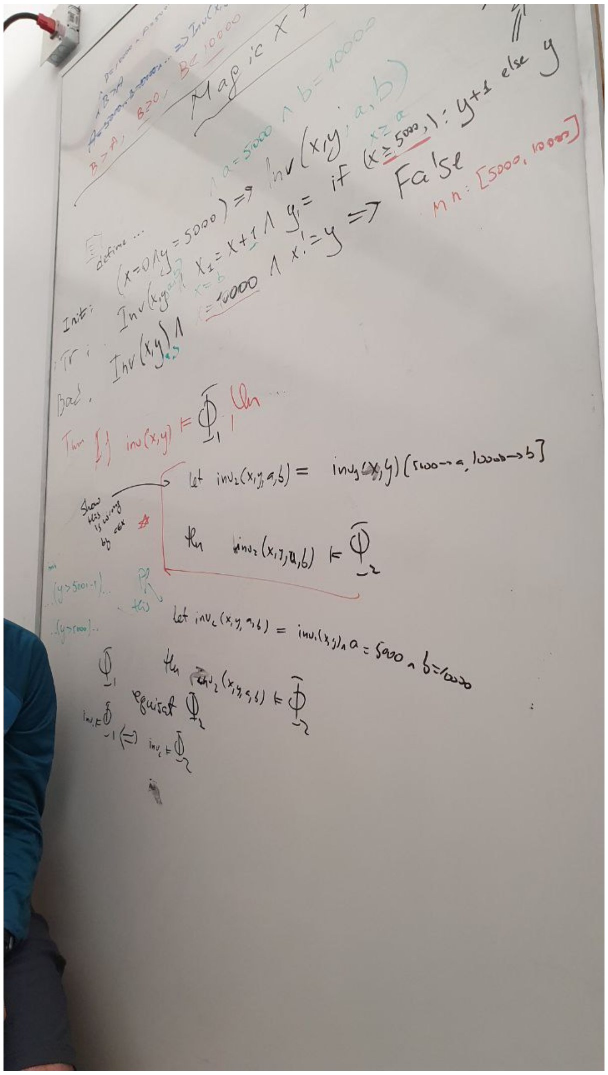
Prerequisite: During last weeks of summer I tried to show my technique to my advisor's collegue. During the conversation I got a task to prove 2 statement that will prove my technique. The picture of notes from this conversation I attach here. Everything on the blackboard was duplicated to this file. 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 [ ]:
```python
import sys
sys.path.insert(1, '/Users/ekvashyn/Code/spacer-on-jupyter/src/')
from spacer_tutorial import *
from z3 import *
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: $x = 0 \wedge y = 5000$

Bad: $x = 10000 \wedge x \neq y$

Tr: $x' = x + 1 \wedge y' = \mathrm{If}(x \geq 5000, y + 1, y)$

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: $a = 5000 \wedge b = 10000 \wedge x = 0 \wedge y = a$

Bad: $x = b \wedge x \neq y$

Tr: $x' = x + 1 \wedge y' = \mathrm{If}(x \geq a, y + 1, y) \wedge a' = a \wedge b' = b$

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[ ]: $\forall x, y, x', y' : x = 0 \land y = 5000 \Rightarrow Inv(x, y)$

$\forall x, y, x', y'$:
$Inv(x, y) \land x' = x + 1 \land y' = If(x \geq 5000, y + 1, y) \Rightarrow$
$Inv(x', y')$

$\forall x, y, x', y' : Inv(x, y) \land x = 10000 \land x \neq y \Rightarrow False$

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

Out[ ]: $\forall x, y, a, b, x', y', a', b'$:
$a = 5000 \land b = 10000 \land x = 0 \land y = a \Rightarrow Inv(x, y, a, b)$

$\forall x, y, a, b, x', y', a', b'$:
$Inv(x, y, a, b) \land$
$x' = x + 1 \land$
$y' = If(x \geq a, y + 1, y) \land$
$a' = a \land$
$b' = b \Rightarrow$
$Inv(x', y', a', b')$

$\forall x, y, a, b, x', y', a', b'$:
$Inv(x, y, a, b) \land x = b \land x \neq y \Rightarrow False$

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 → (¬(v_1 ≥ 5001) ∨ ¬(v_1 + -1·v_0 ≥ 1)) ∧ ¬(v_1 + -1·v_0 ≤ -1) ∧ ¬(v_1 ≤ 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 → (¬(v_0 + -1·v_1 ≤ -1) ∨ ¬(v_0 + -1·v_2 ≥ 0)) ∧ ¬(v_2 + -1·v_1 ≥ 1) ∧ ¬(v_2 + -1·v_3 ≥ 0) ∧ ¬(v_0 + -1·v_1 ≥ 1) ∧ (¬(v_0 + -1·v_2 ≤ -1) ∨ ¬(v_2 + -1·v_1 ≤ -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 [ ]:

```
class TransitionSystem:
    def __init__(self):
        self.a, self.b, self.x, self.y, self.x_prime, self.y_prime = Ints('a

    def inv_gen(self, u, w):
        '''Returns an invariant with contrains that don't have number 5000 e
        return And(
            Implies(w >= 5001, u >= w),
            w > 4999,
            u <= w)

    def init_constraints(self):
        return And(self.x == 0, self.y == self.a)

    def transition_constraints(self):
        return And(
            self.x_prime == self.x + 1,
            self.y_prime == If(self.x >= self.a, self.y + 1, self.y))

    def bad_state(self):
        return And(self.x == self.b, self.x_prime != self.y_prime)

    def invariant_constraints(self):
        return self.inv_gen(self.x, self.y)

    def invariant_prime_constraints(self):
        return self.inv_gen(self.x_prime, self.y_prime)

    def prove_solver(self, solver):
        checked = solver.check() == unsat
        if checked:
            print("Invariant equivalency holds.")
        else:
            print("Invariant equivalency does not hold.\nCounterexample:")
            print(solver.model())
            print()
        return checked

    def prove_init_impl_inv(self):
        print("Check the initialization condition Init => Inv is valid:")
        solver = Solver()
        solver.add(self.init_constraints())
        solver.add(Not(self.invariant_constraints()))
        self.prove_solver(solver)

    def prove_inv_tr_impl_inv_p(self):
        print("Check the condition Inv ∧ Tr => Inv` is valid:")
        solver2 = Solver()
        solver2.add(self.transition_constraints())
```

```
        solver2.add(self.invariant_constraints())
        solver2.add(Not(self.invariant_prime_constraints()))
        self.prove_solver(solver2)

    def prove_inv_bad_impl_false(self):
        print("Check the initialization condition Inv ∧ Bad => False` is val
        solver3 = Solver()
        solver3.add(self.invariant_constraints())
        solver3.add(self.bad_state())
        self.prove_solver(solver3)


    def check_conditions(self):
        self.prove_init_impl_inv()
        self.prove_inv_tr_impl_inv_p()
        self.prove_inv_bad_impl_false()

ts = TransitionSystem()
ts.check_conditions()
```

```
Check the initialization condition Init => Inv is valid:
Invariant equivalency does not hold.
Counterexample:
[x = 0, a = 5001, y = 5001]

Check the condition Inv ∧ Tr => Inv` is valid:
Invariant equivalency does not hold.
Counterexample:
[x = 4999,
 y_prime = 5001,
 x_prime = 5000,
 a = 4999,
 y = 5000]

Check the initialization condition Inv ∧ Bad => False` is valid:
Invariant equivalency does not hold.
Counterexample:
[b = 0, x_prime = 1, x = 0, y = 5000, y_prime = 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 [ ]:
```
class InvariantEquivalenceChecker:
    def __init__(self):
        self.x, self.y, self.a, self.b, self.x_prime, self.y_prime = Ints('x

    def inv_for_original_benchmark(self):
        '''(y > 5000 => x ≥ y) ∧ y ≥ 5000 ∧ x ≤ y'''
```

```
        return And(Implies(self.y > 5000, self.x >= self.y), self.y >= 5000,

    def z3_prove(self, claim):
        """Stolen from z3 codebase part. Return boolean value of result
        Try to prove the given claim.

        This is a simple function for creating demonstrations.  It tries to
        `claim` by showing the negation is unsatisfiable.

        >>> p, q = Bools('p q')
        >>> prove(Not(And(p, q)) == Or(Not(p), Not(q)))
        true
        """
        s = Solver()
        s.add(Not(claim))
        return s.check() == unsat

    def weak_prove_eq(self, fml1, fml2):
        s1 = Solver()
        s1.add(Not(Implies(fml1, fml2)))
        s1.add(Not(Implies(fml2, fml1)))
        return s1.check() == unsat

    def strong_prove_eq(self, fml1, fml2):
        s1 = Solver()
        s1.add(Not(Implies(fml1, fml2)))
        s2 = Solver()
        s2.add(Not(Implies(fml2, fml1)))
        print(f"fml1 => fml2: {s1.check() == unsat}")
        print(f"fml2 => fml1: {s2.check() == unsat}")
        return s1.check() == unsat and s2.check() == unsat

    # Provers

    def prove_inv_equi_inv(self):
        print("Check whether inv_for_original_benchmark is equisat to itself
        inv = self.inv_for_original_benchmark()
        inv_2 = And(Implies(self.y > 5000, self.x + 1 > self.y), self.y >= 5
        print(f"Strong prove: = {self.strong_prove_eq(inv, inv_2)}")
        print(f"Weak prove: = {self.weak_prove_eq(inv, inv_2)}")
        print(f"z3 prove: = {self.z3_prove(inv == inv_2)}")
        print()

    def prove_inv_equi_inv_2(self):
        '''(y > a => x ≥ y) ∧ y ≥ a ∧ x ≤ y'''
        print("Check whether inv_for_original_benchmark is equisat to inv_fo
        inv = self.inv_for_original_benchmark()
        I1 = And(Implies(self.y > self.a, self.x >= self.y), self.x <= self.
        print(f"Strong prove: = {self.strong_prove_eq(inv, I1)}")
        print(f"Weak prove: = {self.weak_prove_eq(inv, I1)}")
        print(f"z3 prove: = {self.z3_prove(inv == I1)}")
        print()

    def prove_inv_equi_inv_3(self):
        print("Check whether inv_for_original_benchmark is equisat to inv_fo
        inv = self.inv_for_original_benchmark()
```

```
        I2 = And(Implies(self.a - self.y <= -1, self.x > self.y - 1),
                 self.x < self.y + 1,
                 self.b - self.a >= 5000,
                 self.a < self.y + 1)
        print(f"Strong prove: = {self.strong_prove_eq(inv, I2)}")
        print(f"Weak prove: = {self.weak_prove_eq(inv, I2)}")
        print(f"z3 prove: = {self.z3_prove(inv == I2)}")
        print()

    def prove_inv_equi_inv_4(self):
        print("Check whether inv_for_original_benchmark is equisat to inv_fo
        inv = self.inv_for_original_benchmark()
        I2 = And(Or(Not(self.x - self.y <= -1), Not(self.x - self.a >= 0)),
            Not(self.a - self.y >= 1),
            self.b - self.a > 4999,
            Not(self.x - self.y >= 1),
            Or(Not(self.x - self.a <= -1), Not(self.a - self.y <= -1)))
        print(f"Strong prove: = {self.strong_prove_eq(inv, I2)}")
        print(f"Weak prove: = {self.weak_prove_eq(inv, I2)}")
        print(f"z3 prove: = {self.z3_prove(inv == I2)}")
        print()


    def check_invariant_equivalence(self):
        self.prove_inv_equi_inv()
        self.prove_inv_equi_inv_2()
        self.prove_inv_equi_inv_3()
        self.prove_inv_equi_inv_4()

iec = InvariantEquivalenceChecker()
iec.check_invariant_equivalence()
```

```
Check whether inv_for_original_benchmark is equisat to itself:
fml1 => fml2: True
fml2 => fml1: True
Strong prove: = True
Weak prove: = True
z3 prove: = True

Check whether inv_for_original_benchmark is equisat to inv_for_transfromed_b
enchmark 2:
fml1 => fml2: False
fml2 => fml1: False
Strong prove: = False
Weak prove: = True
z3 prove: = False

Check whether inv_for_original_benchmark is equisat to inv_for_transfromed_b
enchmark 2:
fml1 => fml2: False
fml2 => fml1: False
Strong prove: = False
Weak prove: = True
z3 prove: = False

Check whether inv_for_original_benchmark is equisat to inv_for_transfromed_b
enchmark 2:
fml1 => fml2: False
fml2 => fml1: False
Strong prove: = False
Weak prove: = True
z3 prove: = False
```

So, according to z3, I tried to show Inv(x,y) <==> Inv2(x,y,a,b) (moreover tried 3 output SPACER invariants) but got a result that theory about equisat answers is not true