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

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

        self.transition_constraints = And(
            self.x_prime == self.x + 1,
            self.y_prime == If(self.x >= 5000, self.y + 1, self.y))

        self.bad_state = And(self.x == 10000, self.x != self.y)

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

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

        self.I2 = And(Implies(self.a - self.y <= -1, self.x > self.y - 1),
                      self.x < self.y + 1,
                      self.b - self.a > 4999,
                      self.a - 1 < self.y)
        self.I2_prime = And(
            Implies(self.a - self.y_prime <= -1, self.x_prime > self.y_prime
            self.x_prime < self.y_prime + 1,
            self.b - self.a > 4999,
            self.a - 1 < self.y_prime)

        self.I3 = And(Or(Not(self.x - self.y <= -1), Not(self.x - self.a >=
                      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 <=

        self.I3_prime = And(Or(Not(self.x_prime - self.y_prime <= -1), Not(s
```

```
                                Not(self.a - self.y_prime >= 1),
                                self.b - self.a > 4999,
                                Not(self.x_prime - self.y_prime >= 1),
                                Or(Not(self.x_prime - self.a <= -1), Not(self.a

    def prove_solver(self, solver):
        return solver.check() == unsat

    def check_init_inv(self, inv):
        solver = Solver()
        solver.add(self.init_constraints)
        solver.add(Not(inv))
        result = self.prove_solver(solver)
        print(f"Init => Inv: {result}")
        return result

    def check_inv_transition(self, inv, inv_prime):
        solver = Solver()
        solver.add(self.transition_constraints)
        solver.add(inv)
        solver.add(Not(inv_prime))
        result = self.prove_solver(solver)
        print(f"Inv ∧ Tr => Inv`: {result}")
        return result

    def check_init_bad_false(self, inv):
        solver = Solver()
        solver.add(inv)
        solver.add(self.bad_state)
        result = self.prove_solver(solver)
        print(f"Inv ∧ Bad => False: {result}")
        return result

    def check_i0(self):
        init = self.check_init_inv(self.I0)
        tr = self.check_inv_transition(self.I0, self.I0_prime)
        bad = self.check_init_bad_false(self.I0)

    def check_i1(self):
        init = self.check_init_inv(self.I1)
        tr = self.check_inv_transition(self.I1, self.I1_prime)
        bad = self.check_init_bad_false(self.I1)

    def check_i2(self):
        init = self.check_init_inv(self.I2)
        tr = self.check_inv_transition(self.I2, self.I2_prime)
        bad = self.check_init_bad_false(self.I2)

    def check_i3(self):
        init = self.check_init_inv(self.I3)
        tr = self.check_inv_transition(self.I3, self.I3_prime)
        bad = self.check_init_bad_false(self.I3)


# Example usage:
```

```python
checker = InvariantChecker()

print("I0:")
checker.check_i0()
print()

print("I1:")
checker.check_i1()
print()

print("I2:")
checker.check_i2()
print()

print("I3:")
checker.check_i3()
print()
```

```
I0:
Init => Inv: True
Inv ∧ Tr => Inv`: True
Inv ∧ Bad => False: True

I1:
Init => Inv: False
Inv ∧ Tr => Inv`: False
Inv ∧ Bad => False: True

I2:
Init => Inv: False
Inv ∧ Tr => Inv`: False
Inv ∧ Bad => False: False

I3:
Init => Inv: False
Inv ∧ Tr => Inv`: False
Inv ∧ Bad => False: False

Inv ∧ Bad => False: False
```