Trabalho3

December 13, 2020

1 Trabalho 3

1.1 Sistemas Dinâmicos por FOTS ("First Order Transition System")

O nosso sistema dinâmico denota 4 inversores (A, B, C, D) que lêm um bit num canal input e escrevem num canal output uma transformação desse bit.

Começamos por declarar os nossos estados cada um com uma variável s que os identifica.

[73]:
```
from z3 import *

def declare(i):
    state = {}
    state['s'] = Int('s'+str(i))
    return state
```

Posteriormente, definimos o nosso estado inicial por:

$$s = 0 \vee s = 1$$

[74]:
```
def init(state):
    # state = 0 or state = 1
    return Or(state['s'] == 0, state['s'] == 1)
```

As transições possíveis no FOTS são caracterizadas pelo seguinte predicado:

$$(s = 0 \wedge s' = s) \vee (s = 1 \wedge s' = s - 1)$$

Este predicado é uma disjunção de todas as possíveis transições que podem ocorrer no programa.

[75]:
```
def trans(curr, prox):
    trans01 = And(curr['s'] == 0, prox['s'] == curr['s'])

    trans02 = And(curr['s'] == 1, prox['s'] == curr['s'] - 1)

    return Or(trans01, trans02)
```

[94]:
```
def kinduction_always(declare,init,trans,inv,K):
    # completar
    for k in range(2,K+2):
```

```
        trace = [declare(i) for i in range(k+1)]

        # provar caso base (os primeiros k estados)
        s = Solver()
        s.add(init(trace[0]))
        for i in range(k-1):
            s.add(trans(trace[i], trace[i+1]))

        s.add(Not(inv(trace[k-1])))

        r = s.check()

        if r == sat:
            m = s.model()
            print(" A propriedade falha no caso base começado em: ")
            for v in trace[0]:
                print(v, '=', m[trace[0][v]])
            return
        if r != unsat:
            return



        # provar caso indutivo
        s = Solver()
        for i in range(k):
            s.add(trans(trace[i], trace[i+1]))
            s.add(inv(trace[i]))
        s.add(Not(inv(trace[k])))

        r = s.check()

        if r==sat:
            m = s.model()
            print("A propriedade falha no passo k indutivo que começa em")
            for v in trace[0]:
                print(v, 'm', m[trace[0][v]])
            return

        if r == unsat:
            print("A propriedade verifica-se.")

def equalToZero(state):
    return (state['s'] == 0)
kinduction_always(declare,init,trans,equalToZero,4)
```

```
A propriedade verifica-se.
```

```
A propriedade verifica-se.
A propriedade verifica-se.
A propriedade verifica-se.
```

[ ]:

1.2 Sistema Híbrido

Os automatos hibridos sao modelos de sistemas ciber-fisicos que são particularmente uteis porque podem ser descritos por um FOTS e verificado com um SMT "solver".

Começamos por definir os modos e atribuímos valores constantes às variáveis que são enunciadas.

Criámos duas matrizes: uma matriz 3x2 (s_discrete), uma linha por navio e uma coluna para a rota e outra para a velocidade; Outra matriz 3x3 (s_continuous) com 3 linhas (uma por navio) e um coluna para cada componente x, y e t.

[116]:
```
import math

Mode,(Start, Colisao, Hight) = EnumSort('Mode', ('Start', 'Collision', 'High'))

low_velocity = 1
high_velocity = 10
angle = 15
r = 3
x = 5
y = 7
```

[117]:
```
def declare(i):
    s = {}
    s_discrete = {}
    s_continuous = {}

    for i in range(3):
        s_discrete[i] = {}
        s_discrete[i]['r'] = Real('r' + str(i))
        s_discrete[i]['v'] = Real('v' + str(i))

    for i in range(3):
        s_continuous[i] = {}
        s_continuous[i]['x'] = Real('x' + str(i))
        s_continuous[i]['y'] = Real('y' + str(i))
        s_continuous[i]['t'] = Real('t' + str(i))


    s['m'] = Const('m' + str(i), Mode)

    #print(s_continuous)
```

```
    return (s, s_discrete, s_continuous)
```

[128]:
```
def init(state):
    ship1 = And(state[2][0]['x'] == x, state[2][0]['y'] == y, state[2][0]['t']␣
↪== 0,
                state[1][0]['v'] == high_velocity, state[1][0]['r'] ==␣
↪(state[2][0]['x'] / math.cos(angle)))

    ship2 = And(state[2][1]['x'] == x + 10, state[2][1]['y'] == y + 20,␣
↪state[2][1]['t'] == 0,
                state[1][1]['v'] == high_velocity, state[1][1]['r'] ==␣
↪(state[2][1]['x'] / math.cos(angle*2)))

    ship3 = And(state[2][2]['x'] == x + 50, state[2][2]['y'] == y + 10,␣
↪state[2][2]['t'] == 0,
                state[1][2]['v'] == high_velocity, state[1][2]['r'] ==␣
↪(state[2][2]['x'] / math.cos(angle*3)))

    return And(state[0]['m'] == Start, ship1, ship2, ship3)
```

[129]:
```
# todas as possibilidades dos navios colidirem, ou seja,
# navio 0 com navio1, navio0 com navio3 e navio2 com navio3

def collisionsContinuous(state):
    # colisão entre o navio2 com navio0
    collision1 = And((state[2][1]['x'] - state[2][0]['x']) <= r,
                (state[2][1]['y'] - state[2][0]['y']) <= r,
                (state[2][1]['t'] - state[2][0]['t']) <= r/high_velocity)

    # colisão entre o navio3 com navio0
    collision2 = And((state[2][2]['x'] - state[2][0]['x']) <= r,
                (state[2][2]['y'] - state[2][0]['y']) <= r,
                (state[2][2]['t'] - state[2][0]['t']) <= r/high_velocity)

    # colisão entre o navio 3 com navio 2
    collision3 = And((state[2][2]['x'] - state[2][1]['x']) <= r,
                (state[2][2]['y'] - state[2][1]['y']) <= r,
                (state[2][2]['t'] - state[2][1]['t']) <= r/high_velocity)

    return Or(collision1, collision2, collision3)
```

[130]:
```
def collisionsDiscrete(state):
    # a velocidade de ambos os navios decresce para o valor mais baixo␣
↪low_velocity
    # e o navio0 desvia-se a bombordo (-15 graus) e o navio1 desvia-se␣
↪estibordo (+15graus)
```

```
    collision1 = And(state[1][1]['v'] == low_velocity, state[1][0]['v'] ==␣
↪low_velocity,
                     state[1][0]['r'] == state[2][0]['x']/math.cos(angle - 15),
                     state[1][1]['r'] == state[2][1]['x']/math.cos(angle + 15))

    # a velocidade de ambos os navios decresce para o valor mais baixo␣
↪low_velocity
    # o navio0 desvia-se a estibordo (+15 graus) e o navio2 desvia-se a␣
↪bombordo(-15 graus)
    collision2 = And(state[1][2]['v'] == low_velocity, state[1][0]['v'] ==␣
↪low_velocity,
                     state[1][0]['r'] == state[2][0]['x']/math.cos(angle + 15),
                     state[1][2]['r'] == state[2][2]['x']/math.cos(angle - 15))

    # a velocidade de ambos os navios decresce para o valor mais baixo␣
↪low_velocity
    # o navio1 desvia-se a estibordo (-15 graus) e o navio2 desvia-se a␣
↪bombordo(+15 graus)
    collision3 = And(state[1][2]['v'] == low_velocity, state[1][1]['v'] ==␣
↪low_velocity,
                     state[1][1]['r'] == state[2][1]['x']/math.cos(angle - 15),
                     state[1][2]['r'] == state[2][2]['x']/math.cos(angle + 15))

    return Or(collision1, collision2, collision3)
```

[131]:
```
# os navios após a colisão retornam à velocidade original
def originalVelocity(state):
    return And(state[1][0]['v'] == high_velocity, state[1][1]['v'] ==␣
↪high_velocity,
               state[1][2]['v'] == high_velocity)
```

[132]:
```
# navegação dos navios, vão mudar de coordenadas
def sailing(state):
    ship1 = And(state[2][0]['x'] == x + 15, state[2][0]['y'] == y - 10,␣
↪state[2][0]['t'] == 5)

    ship2 = And(state[2][1]['x'] == x - 10, state[2][1]['y'] == y + 20,␣
↪state[2][1]['t'] == 10)

    ship3 = And(state[2][2]['x'] == x + 50, state[2][2]['y'] == y - 30,␣
↪state[2][2]['t'] == 15)

    return And(ship1, ship2, ship3)
```

[133]:
```
def trans(curr, prox):
    # untimed
```

```
    # transição do start para a velocidade High
    startHigh = And(curr[0]['m'] == Start, prox[0]['m'] == High, prox[1] ==␣
↪curr[1], prox[2] == curr[2])


    highCollision = And(curr[0]['m'] == High, prox[0]['m'] == Collision,␣
↪prox[2] == collisionsContinuous(curr[2]),
                        prox[1] == collisionsDiscrete(curr[1]))

    collisionHigh = And(curr[0]['m'] == Collision, prox[0]['m'] == High,␣
↪prox[1] == originalVelocity(curr[1]))

    # timed
    highHigh = And(curr[0]['m'] == High, prox[0]['m'] == curr[0]['m'], prox[2]␣
↪== sailing(curr[2]))

    return Or(startHigh, highCollision, collisionHigh, highHigh)
```

[151]:
```
def bmc_always(declare,init,trans,inv,K):
    for k in range(1,K+1):
        s = Solver()
        # it's like an authomata
        # declare all k to states
        trace = [declare(i) for i in range(k)]
        #print(trace)

        # initializate state 0
        s.add(init(trace[0]))


        # create a link between two spaces
        for i in range(k-1):
            s.add(trans(trace[0][i], trace[0][i+1]))

        #s.add(trace[0][k-1])


        if s.check() == sat:
            m = s.model()
            print(m)
            for i in range(k):
                print(i)
                for v in trace[0][i]:
                    print(v,'=', m[trace[0][i][v]])
            return
```

```
    print ("Property is valid up to traces of length "+str(K))


bmc_always(declare,init,trans,positive,5)
```

```
[r2 = 550000000000000000/5253219888177297,
 v2 = 10,
 t2 = 0,
 y2 = 17,
 x2 = 55,
 r1 = 300000000000000000/3085028997751681,
 v1 = 10,
 t1 = 0,
 y1 = 27,
 x1 = 15,
 r0 = -12500000000000000/1899219782147053,
 v0 = 10,
 t0 = 0,
 y0 = 7,
 x0 = 5,
 m2 = Start]
0
m = Start
```

[ ]: