Navios

January 17, 2021

1 Navios

Grupo 7:

- Luís Almeida A84180
- João Pedro Antunes A86813

- Considere um sistema híbrido formado por 4 autómatos híbridos: três navios (análogos aos do trabalho TP3) e um controlador. Neste sistema cada autómato desconhece o estado dos restantes e comunica com eles exclusivamente através de eventos.
  A propriedade de segurança é a mesma da do trabalho TP2: ausência de colisões entre navios. Para isso a área de navegação é dividida em setores e o controlador assegura que, em qualquer instante, cada sector não contém mais do que um navio. Assim

  1. Cada navio está, em cada estado, numa de três velocidades v possíveis: $10\,\mathtt{m/s}$ (high) , $1\,\mathtt{m/s}$ (low) e 0 (stop). As transições high $\leftrightarrow$ low têm uma duração mínima de $500\,\mathtt{seg}$; as transições low $\leftrightarrow$ stop têm uma duração mínima de $50\,\mathtt{seg}$.
  2. Cada navio está, em cada estado, numa rota $r \in \{0..23\}$; cada valor de r identifica um ângulo múltiplo de 15^o (também designado por `hora`).
  3. A área de navegação é dividida numa matriz $N \times N$ de setores quadrados com $1\,\mathtt{km}$ de lado. Cada setor setor é identificado por um par de índices $0 \leq \mathtt{linha}, \mathtt{coluna} < N$. Cada navio está, em cada estado, num único setor.
  4. O estado do controlador incluiu o seu setor e a sua velocidade.

- A navegação é determinada pelas seguintes regras:

  1. Um navio só muda de rota ou velocidade quando muda de setor.
  2. Quando um navio entra ou sai de um setor emite um evento, que identifica o navio e os setores envolvidos (de onde vem e para onde vai). Este evento sincroniza com o controlador que assim controla as mudanças de setor de cada navio.
  3. Se existir risco de dois navios estarem simultaneamente no mesmo setor, o controlador deve fazer que um deles mude de rota ou espere que o outro abandone esse setor.
  4. Dois navios em setores adjacentes estão ambos em velocidade low ou stop .

Pretende-se verificar, dada uma determinada posição inicial dos três navios, a seguinte propriedade de segurança: - Em qualquer traço e em qualquer estado, nenhum setor contém mais do que um navio.

1.1 FOTS de um Navio

O objetivo deste problema é a construção de um sistema híbrido constituído por 4 autómatos híbridos (3 navios e 1 controlador) e a posterior prova de uma propriedade de segurança sobre o sistema. Como tal, começa-se por construir o FOTS que modela o autómato híbrido de cada constituinte do sistema. Comecemos por definir o FOTS de cada navio.

[6]:
```
from z3 import *
from itertools import product

navios = 3

horas = range(24)

N = 20

vStop = 0
vLow = 0.001
vHigh = 0.01


vel, (STOP,LOW,HIGH,INIT) = EnumSort('vel', ('STOP','LOW','HIGH','INIT'))
```

Atendendo ao enunciado, verificamos que cada navio tem de possuir informação sobre as suas coordenadas num plano bidimensional (x, y), sobre a sua velocidade v, sobre o seu setor (lin, col), sobre a sua rota r e sobre o seu tempo $time$. Como tal, as variáveis discretas serão as seguintes:

- 3 velocidades: $INIT, HIGH, LOW$
- 24 horas (números inteiros de 0 a 23) que descrevem os ângulos de 0 a 345 em intervalos de 15
- $N \times N$ pares de coordenadas, que indicam o setor onde o navio se encontra

As variáveis contínuas serão:

- (x, y) pares de coordenadas que indicam a posição do navio no plano
- $time$ que indica o tempo do autómato híbrido

Apresenta-se de seguida a função que declara as variáveis de um FOTS que modela o comportamento de um navio:

[7]:
```
def declare_nav(i, navio):
    s = {}
    s['x'] = Real('x' + str(i) + str(navio))
    s['y'] = Real('y' + str(i) + str(navio) )
    s['v'] = Const('v' + str(i) + str(navio),vel)
    s['r'] = Int('r' + str(i) + str(navio))
    s['lin'] = Int('lin' + str(i) + str(navio))
    s['col'] = Int('col' + str(i) + str(navio))
    s['time'] = Real('time' + str(i) + str(navio))
    return s
```

Defina-se agora a função que inicializa o FOTS de um navio. Um navio poderá ter como posição inicial um qualquer par de coordenadas no plano desde que o seu setor seja uma entrada na matriz $N \times N$ que divide a área de navegação. A sua velocidade inicial terá de ser $INIT$, a sua rotação uma hora válida (um inteiro entre 0 e 23) e o tempo será inicializado a 0. Assim, define-se o predicado inicial do FOTS da seguinte forma:

$$lin \leq x \leq lin+1 \quad \wedge \quad col \leq y \leq col+1 \quad \wedge \quad v == INIT \quad \wedge$$
$$0 \leq r \leq 23 \quad \wedge \quad 0 \leq lin \leq N \quad \wedge \quad 0 \leq col \leq N \quad \wedge \quad time == 0$$

[8]:
```
def init_nav(s):
    return And(s['lin'] <= s['x'],s['x'] < s['lin']+1,
        s['col'] <= s['y'], s['y'] < s['col']+1,
        s['v'] == INIT, 0 <= s['r'] , s['r'] <= 23,
        0 <= s['lin'], s['lin'] <= N,
        0 <= s['col'], s['col'] <= N,
        s['time'] == 0)
```

Vamos agora definir a função de transição entre estados do FOTS. Como é sabido, temos sempre 2 tipos de transições, *timed* e *untimed*. As transições *timed* são aquelas em que não existem alterações nas variáveis discretas (os modos do autómato híbrido não são alterados), ou seja, no nosso FOTS teremos:

$$lin' == lin \wedge col' == col \wedge r' == r \wedge v' == v$$

No entanto, nestas transições as coordenadas do navio vão ser alteradas consoante o seu deslocamento (não pode ocorrer alteração dos setores).Sabemos que a variável V de velocidade do navio pode ser interpretada como uma constante interna a cada estado *HIGH*, *LOW* e *STOP*, assim, o deslocamento total do navio seria representado por,

$$\Delta = v \cdot |t_1 - t_0|$$

com V constante a 0.001 ou 0.01, e Δ o deslocamento total do navio. O deslocamento em cada componente, no entanto, será dado por uma relação entre Δ e os valores de seno e cosseno do ângulo associado à rota do navio, que é descrita pelo seguinte sistema de equações:

$$\begin{cases} \Delta_x = \Delta \cdot cos(\rho) \\ \\ \Delta_y = \Delta \cdot sin(\rho) \end{cases}$$

sendo ρ a inclinação do navio, também constante consoante o estado discreto "ativo" do autómato de cada navio. Sendo assim, o flow em cada modo será representado por:

$$lin \leq x \leq lin+1 \quad \wedge \quad col \leq y \leq col+1$$

Pois a partir do momento que este predicado deixa de ser verdadeiro para um determinado modo (setor) dá-se uma transição *untimed* em que o setor é alterado.

Definimos então no *z3* a ocorrência de uma transição *timed*.

[ ]:
```
def deltaXY(now,prox,vel):
    p = []
    for angle in horas:
        angVer = (now['r'] == angle)
        deltaX = (prox['x'] - now['x'] == cosseno(angle*15)*vel*(prox['time'] -␣
 ↪now['time']))
        deltaY = (prox['y'] - now['y'] == seno(angle*15)*vel*(prox['time'] -␣
 ↪now['time']))
        deltaLin = [prox['lin'] <= prox['x'], prox['x'] < prox['lin']+1]
        deltaCol = [prox['col'] <= prox['y'], prox['y'] < prox['col']+1]
        p.append(And(angVer,deltaX,deltaY,*deltaLin,*deltaCol))
    return Or(p)


def cosseno(ang):
    return math.cos(math.radians(ang))

def seno(ang):
    return math.sin(math.radians(ang))

def timed_nav(curr,prox):
    timed = [] # As transições timed não permitem mudança de setor

    # STOP -> STOP
    timed.append(And(prox['lin'] == curr['lin'], prox['col'] == curr['col'],␣
 ↪curr['v'] == STOP,
                 prox['v'] == STOP, prox['time'] > curr['time'],prox['r'] ==␣
 ↪curr['r'],deltaXY(curr,prox,vStop)))

    # LOW -> LOW
    timed.append(And(prox['lin'] == curr['lin'], prox['col'] == curr['col'],␣
 ↪curr['v'] == LOW,
                 prox['v'] == LOW, prox['time'] > curr['time'],prox['r'] ==␣
 ↪curr['r'],deltaXY(curr,prox,vLow)))

    # HIGH -> HIGH
    timed.append(And(prox['lin'] == curr['lin'], prox['col'] == curr['col'],␣
 ↪curr['v'] == HIGH,
                 prox['v'] == HIGH, prox['time'] > curr['time'],prox['r'] ==␣
 ↪curr['r'],deltaXY(curr,prox,vHigh)))

    return Or(timed)
```

Precisamos agora de definir as transições *untimed* do FOTS. Estas são as transições que, ao contrário das transições *timed*, permitem alterações aos modos do navio (variáveis discretas). Serão aquelas que vão permitir a um navio mudar de setor e alterar a sua velocidade. De acordo com o enunciado, um navio apenas pode mudar de velocidade e de rotação quando muda de setor, pelo que podemos dividir as transições *untimed* em dois tipos:

- Transições em que a velocidade e a rotação podem ser alteradas
- Transições em que a velocidade e a rotação não são alteradas

De acordo com o enunciado considera-se que um navio apenas pode alterar a sua rota na velocidade *LOW*.Vamos admitir que um navio apenas pode transitar entre um setor de cada vez, ou seja:

$$(lin' == lin + 1 \vee lin' == lin - 1 \vee lin' == lin) \wedge$$
$$(col' == col + 1 \vee col' == col - 1 \vee col' == col) \wedge \neg(lin' == lin \wedge col' == col)$$

No entanto, apenas podemos adicionar estas restrições nas transições em que não existe alteração da velocidade do navio, pois existe um *delay* quando o navio altera a sua velocidade e portanto ele poderá deslocar-se entre setores até finalizar a mudança de velocidade. Como tal, teremos de ser mais flexíveis nesta situação e permitir que o navio transite para o setor que o seu deslocamento indicar. De seguida apresentam-se todas as transições *untimed* possíveis no FOTS:

[ ]:
```
def untimed_nav(curr,prox):
    untimed = []

    #Alteração da velocidade inicial para qualquer outra é instantânea

    # INIT -> HIGH
    untimed.append(And(prox['r'] == curr['r'], prox['lin'] == curr['lin'],␣
↪prox['col'] == curr['col'],
                curr['v'] == INIT, prox['v'] == HIGH, prox['x'] == curr['x'],␣
↪prox['y'] == curr['y'],
                prox['time'] == curr['time']))

    #INIT -> LOW
    untimed.append(And(prox['r'] == curr['r'], prox['lin'] == curr['lin'],␣
↪prox['col'] == curr['col'],
                curr['v'] == INIT, prox['v'] == LOW, prox['x'] == curr['x'],␣
↪prox['y'] == curr['y'],
                prox['time'] == curr['time']))

    #INIT -> STOP
    untimed.append(And(prox['r'] == curr['r'], prox['lin'] == curr['lin'],␣
↪prox['col'] == curr['col'],
                curr['v'] == INIT, prox['v'] == STOP, prox['x'] == curr['x'],␣
↪prox['y'] == curr['y'],
                prox['time'] == curr['time']))
```

```
   #Transições em que existe alteração de velocidade

   #STOP -> LOW
   untimed.append(And(prox['r'] == curr['r'],
               curr['v'] == STOP, prox['v'] == LOW, deltaXY(curr,prox,vStop),␣
↪prox['time'] >= curr['time'] + 50,
               prox['x'] >= 0, prox['y'] >= 0, prox['x'] <= N, prox['y'] <= N))

   #LOW -> HIGH
   untimed.append(And(prox['r'] == curr['r'],
                      curr['v'] == LOW, prox['v'] == HIGH,␣
↪deltaXY(curr,prox,vLow),
                      prox['time'] >= curr['time'] + 500, prox['x'] >= 0,
                      prox['y'] >= 0, prox['x'] <= N, prox['y'] <= N))

   #HIGH -> LOW
   untimed.append(And(prox['r'] == curr['r'], curr['v'] == HIGH,
                  prox['v'] == LOW, deltaXY(curr,prox,vHigh), prox['time'] >=␣
↪curr['time'] + 500, prox['x'] >= 0,
                     prox['y'] >= 0, prox['x'] <= N, prox['y'] <= N))

   #LOW -> STOP
   untimed.append(And(prox['r'] == curr['r'],
               curr['v'] == LOW, prox['v'] == STOP, deltaXY(curr,prox,vLow),␣
↪prox['time'] >= curr['time'] + 50,
               prox['x'] >= 0, prox['y'] >= 0, prox['x'] <= N, prox['y'] <= N))

   #Transições onde não existe mudança de velocidade

   #STOP -> STOP (Igual à transição timed porque o x,y,r,setor não mudam)
   untimed.append(And(prox['r'] == curr['r'],
               curr['v'] == STOP, prox['v'] == STOP, deltaXY(curr,prox,vStop),␣
↪prox['time'] > curr['time'],
               prox['x'] >= 0, prox['y'] >= 0, prox['x'] <= N, prox['y'] <= N))

   #LOW -> LOW (Aqui o navio pode mudar ou manter a sua rota)
   untimed.append(And(Or(prox['r'] == curr['r']+1,prox['r'] == curr['r']-1,␣
↪prox['r'] == curr['r']),
               Or(prox['lin'] == (curr['lin'])+1, prox['lin'] ==␣
↪(curr['lin'])-1, prox['lin'] == curr['lin']),
               Or(prox['col'] == (curr['col'])+1, prox['col'] ==␣
↪(curr['col'])-1, prox['col'] == curr['col']),
               Not(And(prox['lin'] == curr['lin'], prox['col'] == curr['col'])),
               curr['v'] == LOW, prox['v'] == LOW, deltaXY(curr,prox,vLow),
```

```
                prox['time'] > curr['time'], prox['x'] >= 0, prox['y'] >= 0,␣
↪prox['x'] <= N, prox['y'] <= N))

    #HIGH -> HIGH
    untimed.append(And(prox['r'] == curr['r'],
                Or(prox['lin'] == (curr['lin'])+1, prox['lin'] ==␣
↪(curr['lin'])-1, prox['lin'] == curr['lin']),
                Or(prox['col'] == (curr['col'])+1, prox['col'] ==␣
↪(curr['col'])-1, prox['col'] == curr['col']),
                Not(And(prox['lin'] == curr['lin'], prox['col'] == curr['col'])),
                curr['v'] == HIGH, prox['v'] == HIGH, deltaXY(curr,prox,vHigh),
                prox['time'] > curr['time'], prox['x'] >= 0, prox['y'] >= 0,␣
↪prox['x'] <= N, prox['y'] <= N))

    return Or(untimed)
```

De seguida apresenta-se a função de transição, que indica que o navio pode seguir uma transição *timed* ou uma transição *untimed*.

[9]:
```
def trans(curr,prox):
    return Or(timed_nav(curr,prox),untimed_nav(curr,prox))

def gera_traco(declare_nav,init_nav,trans,k):
    s = Solver()
    d = {}
    state = [declare_nav(i,0) for i in range(k)]
    s.add(init_nav(state[0]))
    for i in range(k - 1):
        s.add(trans(state[i], state[i + 1]))
    if s.check() == sat:
        m = s.model()
        for i in range(k):
            print("Estado", i)
            for x in state[i]:
                if state[i][x].sort() != RealSort():
                    print(x, "=", m[state[i][x]])
                else:
                    print(x, '=', float(m[state[i][x]].numerator_as_long())/
↪float(m[state[i][x]].denominator_as_long()))


gera_traco(declare_nav,init_nav,trans,10)
```

```
Estado 0
x = 1.9808012701892221
y = 20.0
v = INIT
```

```
r = 21
lin = 1
col = 20
time = 0.0
Estado 1
x = 1.9808012701892221
y = 20.0
v = LOW
r = 21
lin = 1
col = 20
time = 0.0
Estado 2
x = 2.95
y = 19.030801270189222
v = LOW
r = 20
lin = 2
col = 19
time = 1370.6539883331789
Estado 3
x = 2.975
y = 18.9875
v = STOP
r = 20
lin = 2
col = 18
time = 1420.6539883331789
Estado 4
x = 2.975
y = 18.9875
v = LOW
r = 20
lin = 2
col = 18
time = 1470.6539883331789
Estado 5
x = 3.0
y = 18.94419872981078
v = STOP
r = 20
lin = 3
col = 18
time = 1520.6539883331789
Estado 6
x = 3.0
y = 18.94419872981078
v = STOP
```

```
r = 20
lin = 3
col = 18
time = 1520.666488333179
Estado 7
x = 3.0
y = 18.94419872981078
v = LOW
r = 20
lin = 3
col = 18
time = 1570.6664883331787
Estado 8
x = 12.9875
y = 1.6453412892166204
v = HIGH
r = 20
lin = 12
col = 1
time = 21545.666488333176
Estado 9
x = 13.937438236238021
y = 0.0
v = HIGH
r = 20
lin = 13
col = 0
time = 21735.65413558078
```

1.2 FOTS do Controlador

Adicionalmente, o nosso sistema híbrido terá também um autómato referente ao controlador, o que também torna relevante a construção de um FOTS que modele o seu comportamento. Segundo o enunciado, o controlador irá possuir informação relativa a cada navio do sistema, como o seu setor, a sua velocidade e a sua rotação, assim como um tempo próprio. O controlador intervirá como um agente que determinará toda a navegação de cada navio, para poder evitar colisões. Em termos de variáveis discretas, o controlador possuirá:

- Pares de coordenadas $N \times N$, um por cada navio, que representarão o seu setor
- A velocidade atual de cada navio
- A rotação de cada navio

A única variável contínua do FOTS do Controlador será o tempo. Define-se então a função que declara as variáveis do Controlador:

[10]:
```
def declare_cont(i):
    state = {}

    for r in range(navios):
```

```
        state[r] = {}
        state[r]['lin'] = Int(str(i) + str(r) + ' lin')
        state[r]['col'] = Int(str(i) + str(r) + ' col')
        state[r]['vel'] = Const('v' + str(i) + str(r),vel)
        state[r]['r'] = Int(str(i) + str(r) + 'rot')

    state['time'] = Real(str(i) + 'time')

    return state
```

Para definirmos o predicado inicial do Controlador basta pensarmos no que acontece no predicado inicial de cada navio. Cada navio é inicializado com um setor válido, com uma rotação válida e com a velocidade $INIT$. Basta inicializarmos adicionalmente o tempo do Controlador com o valor 0. Como tal, o predicado que inicializará o FOTS do Controlador será:

$$nav_v == INIT \quad \wedge \quad 0 \leq nav_r \leq 23 \quad \wedge \quad 0 \leq nav_{lin} \leq N \quad \wedge \quad 0 \leq nav_{col} \leq N \quad \wedge \quad time == 0 \quad \forall nav$$

O Controlador não poderá no entanto permitir que exista mais do que um navio no mesmo setor. Portanto adiciona-se a seguinte restrição:

$$(navA_{lin} \neq navB_{lin} \vee navA_{col} \neq navB_{col}), \quad \forall navA, navB$$

[11]:
```
def init_cont(state):
    ini = []

    for r in range(navios):
        ini.append(0 <= state[r]['lin'])
        ini.append(state[r]['lin'] <= N)
        ini.append(0 <= state[r]['col'])
        ini.append(state[r]['col'] <= N)
        ini.append(state[r]['vel'] == INIT)
        ini.append(0 <= state[r]['r'])
        ini.append(state[r]['r'] <= 23)

    ini.append(state['time'] == 0)

    for n in range(navios):
        for i in range(navios):
            if n != i:
                ini.append(Or(state[n]['lin'] != state[i]['lin'],␣
↪state[n]['col'] != state[i]['col']))

    return And(ini)
```

Precisamos agora de definir a função de transição entre estados do FOTS do Controlador. Tem-se então 2 tipos de transições, *timed* e *untimed*. Conforme referido anteriormente, as transições

timed são aquelas onde não existe alteração dos modos do FOTS, pelo que vão corresponder às transições onde não existe alteração do setor de um navio. Podemos então definir as transições timed possíveis de um navio no Controlador da seguinte forma:

[ ]:
```
def timed_cont_nav(curr,prox,n):
    t = []

    t.append(And(prox[n]['lin'] == curr[n]['lin'],prox[n]['col'] ==␣
↪curr[n]['col'], curr[n]['vel'] == HIGH,
             prox[n]['vel'] == curr[n]['vel'],prox['time'] > curr['time'],␣
↪Not(adjacent(prox,n))))

    t.append(And(prox[n]['lin'] == curr[n]['lin'],prox[n]['col'] ==␣
↪curr[n]['col'], curr[n]['vel'] == LOW,
             prox[n]['vel'] == curr[n]['vel'], prox['time'] > curr['time']))

    t.append(And(prox[n]['lin'] == curr[n]['lin'],prox[n]['col'] ==␣
↪curr[n]['col'], curr[n]['vel'] == STOP,
             prox[n]['vel'] == curr[n]['vel'], prox['time'] > curr['time']))

    return And(t)
```

Note-se que segundo o enunciado sempre que existam navios em setores adjacentes estes têm de estar em velocidade *LOW* ou *STOP*, pelo que apenas é possível seguir uma transição *timed* em que a velocidade é *HIGH* sempre que o navio não tenha outros navios em setores adjacentes. Como tal, apresenta-se o seguinte predicado que permite ao Controlador detetar esta situação:

[12]:
```
def adjacent(state,n1):
    pred = []
    for n in range(navios):
        if n != n1:
            pred.append(And(state[n]['lin'] >= state[n1]['lin']-1,
                            state[n]['lin'] <= state[n1]['lin']+1,
                            state[n]['col'] >= state[n1]['col']-1,
                            state[n]['col'] <= state[n1]['col']+1))
    return Or(pred)
```

Falta-nos então definir as transições *untimed* do Controlador. Conforme referido anteriormente, podemos classifica-las em 2 tipos consoante o navio altere (ou não) a sua velocidade. Comecemos por definir o predicado que impede que exista mais do que um navio no mesmo setor:

$$(navA_{lin} \neq navB_{lin} \vee navA_{col} \neq navB_{col}), \quad \forall navA, navB$$

Este predicado terá de se verificar em todos os estados do FOTS e será impossível de seguir uma transição para um estado que não o verifique. Sendo assim, todas as transições *untimed* terão de o englobar.

[ ]:
```
def no_collis(state,n1):
    pred = []
    for n in range(navios):
        if n != n1:
            pred.append(And(state[n]['lin'] == state[n1]['lin'],
                            state[n]['col'] == state[n1]['col']))
    return Not(Or(pred))
```

Note-se que no enunciado é referido que, sempre que exista risco de colisão, o Controlador deve fazer com que apenas um dos navios altere a sua rota ou com que um dos navios altere a sua velocidade para *STOP*. Sendo assim, no máximo apenas 1 dos seguintes predicados será verdadeiro:

$$nav_{r'} \neq nav_r \wedge (nav_v \neq STOP \wedge nav_{v'} == STOP) \quad \forall nav$$

[13]:
```
def col_prevent(curr,prox):
    pred = []
    l = [prox[n]['r'] != curr[n]['r'] for n in range(navios)]
    v = [And(curr[n]['vel'] != STOP, prox[n]['vel'] == STOP) for n in␣
 ↪range(navios)]

    pred.append(AtMost(*l,1))
    pred.append(AtMost(*v,1))

    return And(pred)
```

As transições *untimed* do FOTS do Controlador seguem a lógica das transições *untimed* do FOTS do navio, ou seja, será permitido fazer transições *untimed* apenas quando é alterado o setor de um navio. Para conservarmos a restrição que indica que navios em setores adjacentes tenham de estar em velocidade *LOW* ou *STOP*, adicionamos o predicado *adjacent* para todas as transições que indiquem que a próxima velocidade do navio é *LOW* ou *STOP*, e adicionamos a negação do mesmo predicado para todas as transições que indiquem que a próxima velocidade do navio é *HIGH*. Sendo assim, e tendo em conta o *delay* de cada alteração de velocidade de um navio, apresentam-se as transições *untimed* de um navio no Controlador:

[ ]:
```
def untimed_cont_nav(curr,prox,nav):
    t = []

    #INIT -> HIGH
    t.append(And(curr[nav]['lin'] == prox[nav]['lin'], curr[nav]['col'] ==␣
 ↪prox[nav]['col'],
                 Not(adjacent(prox,nav)), curr[nav]['vel'] == INIT,␣
 ↪prox[nav]['vel'] == HIGH,
                 prox['time'] == curr['time'], prox[nav]['lin'] >= 0,␣
 ↪prox[nav]['lin'] <= N,
                 prox[nav]['col'] >= 0, prox[nav]['col'] <= N,␣
 ↪no_collis(prox,nav)))
```

```
   #INIT -> LOW
   t.append(And(curr[nav]['lin'] == prox[nav]['lin'], curr[nav]['col'] ==␣
↪prox[nav]['col'],
                adjacent(prox,nav), curr[nav]['vel'] == INIT, prox[nav]['vel']␣
↪== LOW,
                prox['time'] == curr['time'], prox[nav]['lin'] >= 0,␣
↪prox[nav]['lin'] <= N,
                prox[nav]['col'] >= 0, prox[nav]['col'] <= N))

   #INIT -> STOP
   t.append(And(curr[nav]['lin'] == prox[nav]['lin'], curr[nav]['col'] ==␣
↪prox[nav]['col'],
                adjacent(prox,nav), curr[nav]['vel'] == INIT, prox[nav]['vel']␣
↪== STOP,
                prox['time'] == curr['time'], prox[nav]['lin'] >= 0,␣
↪prox[nav]['lin'] <= N,
                prox[nav]['col'] >= 0, prox[nav]['col'] <= N))

   #Transições que permitem mudança de velocidade

   #HIGH -> LOW
   t.append(And(Not(And(curr[nav]['lin'] == prox[nav]['lin'], curr[nav]['col']␣
↪== prox[nav]['col'])),
                adjacent(prox,nav),curr[nav]['vel'] == HIGH, prox[nav]['vel']␣
↪== LOW,
                prox['time'] >= curr['time'] + 500, prox[nav]['lin'] >= 0,␣
↪prox[nav]['lin'] <= N,
                prox[nav]['col'] >= 0, prox[nav]['col'] <= N))

   #LOW -> STOP
   t.append(And(Not(And(curr[nav]['lin'] == prox[nav]['lin'], curr[nav]['col']␣
↪== prox[nav]['col'])),
                adjacent(prox,nav), curr[nav]['vel'] == LOW, prox[nav]['vel']␣
↪== STOP,
                prox['time'] >= curr['time'] + 50, prox[nav]['lin'] >= 0,␣
↪prox[nav]['lin'] <= N,
                prox[nav]['col'] >= 0, prox[nav]['col'] <= N))

   #LOW -> HIGH
   t.append(And(Not(And(curr[nav]['lin'] == prox[nav]['lin'], curr[nav]['col']␣
↪== prox[nav]['col'])),
                Not(adjacent(prox,nav)), curr[nav]['vel'] == LOW,␣
↪prox[nav]['vel'] == HIGH,
                prox['time'] >= curr['time'] + 500, prox[nav]['lin'] >= 0,␣
↪prox[nav]['lin'] <= N,
```

```
                prox[nav]['col'] >= 0, prox[nav]['col'] <= N))

    #Transições que não permitem mudança de velocidade

     #LOW -> LOW
    t.append(And(Not(And(curr[nav]['lin'] == prox[nav]['lin'], curr[nav]['col']␣
↪== prox[nav]['col'])),
                adjacent(prox,nav), curr[nav]['vel'] == LOW, prox[nav]['vel']␣
↪== LOW,
                prox['time'] > curr['time'], prox[nav]['lin'] >= 0,␣
↪prox[nav]['lin'] <= N,
                prox[nav]['col'] >= 0, prox[nav]['col'] <= N))

    #HIGH -> HIGH
    t.append(And(Not(And(curr[nav]['lin'] == prox[nav]['lin'], curr[nav]['col']␣
↪== prox[nav]['col'])),
                Not(adjacent(prox,nav)), curr[nav]['vel'] == HIGH,␣
↪prox[nav]['vel'] == HIGH,
                prox['time'] > curr['time'], prox[nav]['lin'] >= 0,␣
↪prox[nav]['lin'] <= N,
                prox[nav]['col'] >= 0, prox[nav]['col'] <= N))


    return And(Or(t),col_prevent(curr,prox),no_collis(prox,nav))
```

Tendo as transições *timed* e *untimed* definidas, define-se a função de transição entre estados do FOTS como sendo:

$$\forall nav, \quad timed(nav) \vee untimed(nav)$$

[14]:
```
def trans(curr,prox):
    actions = []

    for n in range(navios):
        actions.
↪append(Or(timed_cont_nav(curr,prox,n),untimed_cont_nav(curr,prox,n)))

    return And(actions)
```

De seguida apresentam-se algumas funções que geram traços de execução do FOTS assim como provam que, em todos os estados, não existe mais do que um navio no mesmo setor, e que navios em setores adjacentes estão em velocidade *LOW* ou *STOP*

[18]:
```
def gera_traco(declare_cont,init_cont,trans,k):
    trace = [declare_cont(i) for i in range(k)]
    s = Solver()
```

```
    s.add(init_cont(trace[0]))

    for i in range(k-1):
        s.add(trans(trace[i],trace[i+1]))

    r = s.check()
    if r == sat:
        m = s.model()
        for i in range(k):
            print("==========\n\n\nstate: ",i)
            for v in trace[i]:
                if v != 'time':
                    for h in trace[i][v]:
                        if trace[i][v][h].sort() != RealSort():
                            print(v,h, "=", m[trace[i][v][h]])
                        else:
                            print(v,h, '=', float(m[trace[i][v][h]].
 ↪numerator_as_long())/float(m[trace[i][v][h]].denominator_as_long()))
                    print("\n")
                else:
                    print(v, '=', float(m[trace[i][v]].numerator_as_long())/
 ↪float(m[trace[i][v]].denominator_as_long()))
        return

    print('UNSAT')
    return

gera_traco(declare_cont,init_cont,trans,6)

def bmc_always(declare,init,trans,inv,K):
    for k in range(1,K+1):
        s = Solver()
        trace = [declare_cont(i) for i in range(k)]
        s.add(init_cont(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()
            for i in range(k):
                print("==========\n\n\nstate: ",i)
                for v in trace[i]:
                    if v != 'time':
                        for h in trace[i][v]:
                            if trace[i][v][h].sort() != RealSort():
```

```
                                print(v,h, "=", m[trace[i][v][h]])
                            else:
                                print(v,h, '=', float(m[trace[i][v][h]].
 ↪numerator_as_long())/float(m[trace[i][v][h]].denominator_as_long()))
                        print("\n")
                    else:
                        print(v, '=', float(m[trace[i][v]].numerator_as_long())/
 ↪float(m[trace[i][v]].denominator_as_long()))
            return

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

def adj(n1,n2,state):
    return And(state[n2]['lin'] >= state[n1]['lin']-1,
                            state[n2]['lin'] <= state[n1]['lin']+1,
                            state[n2]['col'] >= state[n1]['col']-1,
                            state[n2]['col'] <= state[n1]['col']+1)

def prop_adj(state):
    t = []

    for n in range(navios):
        for i in range(navios):
            if i != n:
                t.append(Implies(And(adj(n,i,state),state[n]['vel'] != INIT,␣
 ↪state[i]['vel'] != INIT),And(Or(state[n]['vel'] == LOW, state[n]['vel'] ==␣
 ↪STOP),
                                                        Or(state[i]['vel'] == LOW,␣
 ↪state[i]['vel'] == STOP))))
    return And(t)

bmc_always(declare,init,trans,prop_adj,15)

def prop_no_collision(state):
    t = []

    for n in range(navios):
        for i in range(navios):
            if i != n:
                t.append(Or(state[n]['lin'] != state[i]['lin'], state[n]['col'] !
 ↪= state[i]['col']))

    return And(t)

bmc_always(declare_cont,init_cont,trans,prop_no_collision,15)
```

```
==========


state:  0
0 lin = 1
0 col = 2
0 vel = INIT
0 r = 0


1 lin = 0
1 col = 0
1 vel = INIT
1 r = 0


2 lin = 0
2 col = 3
2 vel = INIT
2 r = 0


time = 0.0
==========


state:  1
0 lin = 1
0 col = 2
0 vel = LOW
0 r = 1


1 lin = 0
1 col = 0
1 vel = HIGH
1 r = 0


2 lin = 0
2 col = 3
2 vel = LOW
2 r = 0


time = 0.0
==========
```

```
state:  2
0 lin = 3
0 col = 3
0 vel = HIGH
0 r = 2


1 lin = 0
1 col = 1
1 vel = LOW
1 r = 0


2 lin = 1
2 col = 2
2 vel = LOW
2 r = 0


time = 500.0
==========


state:  3
0 lin = 2
0 col = 2
0 vel = LOW
0 r = 3


1 lin = 5
1 col = 0
1 vel = HIGH
1 r = 0


2 lin = 3
2 col = 1
2 vel = LOW
2 r = 0


time = 1000.0
==========


state:  4
```

```
0 lin = 2
0 col = 4
0 vel = LOW
0 r = 3


1 lin = 0
1 col = 1
1 vel = HIGH
1 r = 1


2 lin = 3
2 col = 3
2 vel = LOW
2 r = 0


time = 1000.5
==========


state:  5
0 lin = 3
0 col = 2
0 vel = LOW
0 r = 4


1 lin = 5
1 col = 2
1 vel = LOW
1 r = 1


2 lin = 4
2 col = 1
2 vel = STOP
2 r = 0


time = 1500.5
Property is valid up to traces of length 15
Property is valid up to traces of length 15
```

1.3 Sistema Híbrido

Tendo os FOTS de cada Navio e do Controlador construídos, torna-se bastante simples construir o sistema híbrido resultante da composição simples destes autómatos. Como foi referido no início do problema, este sistema terá 3 navios e 1 controlador. Sendo assim, aproveitam-se as funções de declaração de variáveis dos FOTS apresentadas anteriormente para declarar as variáveis do sistema híbrido:

[ ]:
```
def declare(i):
    state = {}

    for n in range(navios):
        state[n] = declare_nav(i,n)

    state['controlador'] = declare_cont(i)

    return state
```

O predicado *init* do FOTS do sistema híbrido será a aplicação do predicado *init* de cada autómato híbrido aplicado ao conjunto de variáveis do sistema que o representam. Sendo assim, podemos definir este predicado do sistema híbrido da seguinte forma:

$$init_{nav}(nav1) \quad \wedge \quad init_{nav}(nav2) \quad \wedge \quad init_{nav}(nav3) \quad \wedge \quad init_{cont}(controlador)$$

Não nos podemos esquecer no entanto de sincronizar os setores de cada navio no controlador com os setores de cada navio correspondente, bem como as suas rotas. Assim, temos também:

$$lin_{nav} == lin_{controlador_{nav}} \wedge col_{nav} == col_{controlador_{nav}}, \forall nav$$

$$r_{nav} == r_{controlador_{nav}} \wedge r_{nav} == r_{controlador_{nav}}, \forall nav$$

[19]:
```
def init(state):

    init_navs = And([init_nav(state[n]) for n in range(navios)])

    init_conts = init_cont(state['controlador'])

    sectors = And([And(state[n]['lin'] == state['controlador'][n]['lin'],
                   state[n]['col'] == state['controlador'][n]['col']) for n in␣
 ↪range(navios)])

    routes = And([state[n]['r'] == state['controlador'][n]['r'] for n in␣
 ↪range(navios)])

    return And(init_navs,init_conts,sectors,routes)
```

As transições num sistema híbrido serão a sincronização das transições entre os autómatos do sistema. No nosso caso, os eventos e associados às transições em todos os autómatos são as mudanças de setor dos navios. Uma condição necessária ao sincronismo entre transições é a igualdade de duração das mesmas, portanto em todas as transições teremos o predicado:

$$t'_{nav} - t_{nav} == t'_{controlador_{nav}} - t_{controlador_{nav}}, \quad \forall nav$$

Esta condição obriga a que todas as transições de um navio sincronizem com o controlador. No entanto, temos também de sincronizar a informação relativa à velocidade, aos setores e à rotação de um navio com o controlador, de modo que este último coordene as várias mudanças de setor de cada navio do sistema. Assim, os predicados:

$$v'_{nav} == v'_{controlador_{nav}}, \quad \forall nav$$

$$r'_{nav} == r'_{controlador_{nav}}, \quad \forall nav$$

terão de estar presentes em todas as transições. Vamos ter 3 tipos de sincronismo, *untimed*, *timed* e *mixed*.

$$untimed \equiv \bigvee_{e \in E} \quad untimed_{nav1,e} \wedge untimed_{nav2,e} \wedge untimed_{nav3,e} \wedge untimed_{controlador,nav1,e} \\ \wedge untimed_{controlador,nav2,e} \wedge untimed_{controlador,nav3,e}$$

$$timed \equiv \bigvee_{e \in E} \quad timed_{nav1,e} \wedge timed_{nav2,e} \wedge timed_{nav3,e} \wedge timed_{controlador,nav1,e} \\ \wedge timed_{controlador,nav2,e} \wedge timed_{controlador,nav3,e}$$

Ora, os sincronismos *mixed* como o próprio nome indica, serão todas as combinações possíveis de transições *timed* e *untimed* nos diferentes autómatos do sistema. Um exemplo seria:

$$timed_{nav1,e} \wedge untimed_{nav2,e} \wedge untimed_{nav3,e} \wedge timed_{controlador,nav1,e} \wedge \\ untimed_{controlador,nav2,e} \wedge untimed_{controlador,nav3,e}$$

Apresenta-se então a função de transição no sistema híbrido, que traduz estes sincronismos

[22]:
```
def trans(curr,prox):

    sectors = And([And(prox[n]['lin'] == prox['controlador'][n]['lin'],
                   prox[n]['col'] == prox['controlador'][n]['col']) for n in␣
 ↪range(navios)])

    time = And([prox[n]['time'] - curr[n]['time'] == prox['controlador']['time']␣
 ↪- curr['controlador']['time'] for n in range(navios)])
```

```
   speed = And([prox[n]['v'] == prox['controlador'][n]['vel'] for n in␣
↪range(navios)])

   routes = And([prox[n]['r'] == prox['controlador'][n]['r'] for n in␣
↪range(navios)])

   untimed = []

   untimed += [untimed_nav(curr[n],prox[n]) for n in range(navios)]
   untimed += [untimed_cont_nav(curr['controlador'],prox['controlador'],n) for␣
↪n in range(navios)]

   untimed = And(untimed)


   timed = []

   timed += [timed_nav(curr[n],prox[n]) for n in range(navios)]
   timed += [timed_cont_nav(curr['controlador'],prox['controlador'],n) for n in␣
↪range(navios)]

   timed = And(timed)

   mixed = []
   mixed_modes = ["".join(x) for x in product(['t','u'], repeat=navios) if 't'␣
↪in x and 'u' in x]

   for mode in mixed_modes:
       l = []
       for n in range(navios):

           if mode[n] == 'u':
               l.append(untimed_nav(curr[n],prox[n]))
               l.
↪append(untimed_cont_nav(curr['controlador'],prox['controlador'],n))

           elif mode[n] == 't':
               l.append(timed_nav(curr[n],prox[n]))
               l.
↪append(timed_cont_nav(curr['controlador'],prox['controlador'],n))

       mixed.append(And(l))

   mixed = Or(mixed)

   return And(Or(untimed,timed,mixed),sectors,time,speed)
```

Resta agora provar que no nosso FOTS que representa o sistema híbrido não existe nenhum setor onde navegam mais do que 1 barco. Para tal, vamos usar uma abordagem *BMC* e provar que esta propriedade é válida em todos os estados de execução, ou seja, é uma propriedade de segurança.

[23]:
```
def gera_traco(declare,init,trans,k):
    trace = [declare(i) for i in range(k)]
    s = Solver()

    s.add(init(trace[0]))

    for i in range(k-1):
        s.add(trans(trace[i],trace[i+1]))

    r = s.check()
    if r == sat:
        m = s.model()
        for i in range(k):
            print("==========\n\n\nstate: ",i)
            for v in trace[i]:
                if v != 'time' and v != 'controlador':
                    for h in trace[i][v]:
                        if trace[i][v][h].sort() != RealSort():
                            print(v,h, "=", m[trace[i][v][h]])
                        else:
                            print(v,h, '=', float(m[trace[i][v][h]].
 ↪numerator_as_long())/float(m[trace[i][v][h]].denominator_as_long()))
                    print("\n")
                elif v == 'time':
                    print(v, '=', float(m[trace[i][v]].numerator_as_long())/
 ↪float(m[trace[i][v]].denominator_as_long()))

                else:
                    for n in trace[i][v]:
                        if n != 'time':
                            for var in trace[i][v][n]:
                                if trace[i][v][n][var].sort() != RealSort():
                                    print(v,n,var, "=", m[trace[i][v][n][var]])
                        elif n == 'time':
                            print(n, '=', float(m[trace[i][v][n]].
 ↪numerator_as_long())/float(m[trace[i][v][n]].denominator_as_long()))

        return

    print('UNSAT')
    return

def bmc_always(declare,init,trans,inv,K):
```

```
    for k in range(1,K+1):
        s = Solver()
        trace = [declare(i) for i in range(k)]
        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()
            for i in range(k):
                print("==========\n\n\nstate: ",i)
                for v in trace[i]:
                    if v != 'time' and v != 'controlador':
                        for h in trace[i][v]:
                            if trace[i][v][h].sort() != RealSort():
                                print(v,h, "=", m[trace[i][v][h]])
                            else:
                                print(v,h, '=', float(m[trace[i][v][h]].
 ↪numerator_as_long())/float(m[trace[i][v][h]].denominator_as_long()))
                        print("\n")
                    elif v == 'time':
                        print(v, '=', float(m[trace[i][v]].numerator_as_long())/
 ↪float(m[trace[i][v]].denominator_as_long()))

                    else:
                        for n in trace[i][v]:
                            if n != 'time':
                                for var in trace[i][v][n]:
                                    if trace[i][v][n][var].sort() != RealSort():
                                        print(v,n,var, "=",␣
 ↪m[trace[i][v][n][var]])
                            elif n == 'time':
                                print(n, '=', float(m[trace[i][v][n]].
 ↪numerator_as_long())/float(m[trace[i][v][n]].denominator_as_long()))

            return

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

#gera_traco(declare,init,trans,8)

def prop_no_collision(state):
    t = []

    for n in range(navios):
```

```
        for i in range(navios):
            if i != n:
                t.append(Or(state['controlador'][n]['lin'] !=␣
↪state['controlador'][i]['lin'], state['controlador'][n]['col'] !=␣
↪state['controlador'][i]['col']))

    return And(t)

bmc_always(declare,init,trans,prop_no_collision,7)
```

```
Property is valid up to traces of length 7
```