TP4 - Sistema de travagem ABS

Janeiro, 2022

Bruno Miguel Ferreira Fernandes - a95972

Hugo Filipe de Sá Rocha - a96463

Descrição do Problema

No contexto do sistema de travagem ABS (“Anti-Lock Breaking System”), pretende-se construir um autómato híbrido que descreva o sistema e que possa ser usado para verificar as suas propriedades dinâmicas.

1. A componente discreta do autómato contém os modos: `Start`, `Free`, `Stopping`, `Blocked` e `Stopped`. No modo `Start` inicia o funcionamento com os valores iniciais das velocidades. No modo `Free` não existe qualquer força de travagem. No modo `Stopping` aplica-se a força de travagem alta. No modo `Blocked` as rodas estão bloqueadas em relação ao corpo mas o veículo move-se (i.e. derrapa) com pequeno atrito ao solo. No modo `Stopped` o veículo está imobilizado.
2. A componente contínua do autómato usa variáveis contínuas V, v para descrever a `velocidade do corpo` e a `velocidade linear das rodas` ambas em relação so solo.
3. Assume-se que o sistema de travagem exerce uma força de atrito proporcional à diferença das duas velocidades. A dinâmica contínua, as equações de fluxo, está descrita abaixo.
4. Os “switchs” são a componente de projeto deste trabalho; cabe ao aluno definir quais devem ser de modo a que o sistema tenha um comportamento desejável: imobilize-se depressa e não “derrape” muito.
5. É imprescindível evitar que o sistema tenha “trajetórias de Zenão”. Isto é, sequências infinitas de transições entre dois modos em intervalos de tempo que tendem para zero mas nunca alcançam zero.

Descrição das transições

$$
\begin{gathered}
m = \mathrm{START} \wedge m' = \mathrm{FREE} \wedge v' = v \wedge \mathrm{V}' = \mathrm{V} \wedge timer' = \mathrm{timer} \wedge tempo' = \mathrm{tempo} \\
\vee \\
m = \mathrm{FREE} \wedge m' = \mathrm{FREE} \wedge v' = \mathrm{v} - \mathrm{a} * \mathrm{P} + \mathrm{c} * (\mathrm{V} - \mathrm{v}) \wedge \mathrm{V}' = \mathrm{V} - \mathrm{c} * (\mathrm{V} - \mathrm{v}) \wedge timer' \\
= \mathrm{timer} + (\mathrm{tempo}' - \mathrm{tempo}) \wedge tempo' = \mathrm{tempo} + 0.2 \wedge timer = \mathrm{timerlim} \\
\vee \\
m = \mathrm{FREE} \wedge m' = \mathrm{STOPPING} \wedge v' = v \wedge \mathrm{V}' = \mathrm{V} \wedge timer' = 0 \wedge tempo' = \mathrm{tempo} \wedge timer \\
\geq \mathrm{timerlim} \\
\vee \\
m = \mathrm{STOPPING} \wedge m' = \mathrm{STOPPING} \wedge v' = \mathrm{v} + ((-\mathrm{a}) * \mathrm{P}) + (\mathrm{cSTOPPING} * (\mathrm{V} - \mathrm{v})) \wedge \mathrm{V}' \\
= \mathrm{V} + (-\mathrm{cSTOPPING}) * (\mathrm{V} - \mathrm{v}) \wedge timer' = 0 \wedge tempo' = \mathrm{tempo} + 0.2 \\
\vee \\
m = \mathrm{STOPPING} \wedge m' = \mathrm{BLOCKED} \wedge v' = v \wedge \mathrm{V}' = \mathrm{V} \wedge \mathrm{v} = \mathrm{V} \wedge timer' = 0 \wedge tempo' = \mathrm{tempo} \\
\vee \\
m = \mathrm{BLOCKED} \wedge m' = \mathrm{BLOCKED} \wedge v' = \mathrm{v} - \mathrm{a} * \mathrm{P} \wedge \mathrm{V}' = \mathrm{V} \wedge timer' \\
= \mathrm{timer} + (\mathrm{tempo}' - \mathrm{tempo}) \wedge tempo' = \mathrm{tempo} + 0.2 \wedge timer \leq \mathrm{timerlim} \\
\vee \\
m = \mathrm{BLOCKED} \wedge m' = \mathrm{STOPPED} \wedge v' = 0 \wedge \mathrm{V}' = 0 \wedge timer' = 0 \wedge timer \geq \mathrm{timerlim} \wedge tempo' \\
= \mathrm{tempo} \\
\vee \\
m = \mathrm{BLOCKED} \wedge m' = \mathrm{FREE} \wedge v' = v \wedge \mathrm{V}' = \mathrm{V} \wedge timer' = 0 \wedge timer \geq \mathrm{timerlim} \wedge tempo' \\
= \mathrm{tempo} \\
\vee \\
m = \mathrm{STOPPED} \wedge m' = \mathrm{STOPPED} \wedge v' = 0 \wedge \mathrm{V}' = 0 \wedge timer' = 0 \wedge timer \geq 0 \wedge tempo' \\
= \mathrm{tempo} \\
\vee \\
m = \mathrm{QUALQUER} \wedge m' = \mathrm{STOPPED} \wedge v = 0 \wedge \mathrm{V} = 0 \wedge tempo' = \mathrm{tempo}
\end{gathered}
$$

Inicialização das Variáveis

In [142…

```
from pysmt.shortcuts import *
from pysmt.typing import *

global START
global FREE
global STOPPING
global BLOCKED
global STOPPED
START, FREE, STOPPING, BLOCKED, STOPPED = Int(0), Int(1), Int(2), Int(3), Int(4)
# numeraçao dos diferentes estados

global c
global cSTOPPING
c =  Real(0.1)
cSTOPPING = Real(1)

global P #peso
global a #constante de atrito
global timerlim #tempo maximo do timer
P, a, timerlim = Real(1000.0), Real(0.01), Real(0.2)

global aP
aP = a*P
```

```
global v0
v0 = Real(20.0)


def declare(i):
    s = {}
    s['v'] = Symbol('v'+str(i),REAL)
    s['m'] = Symbol('m'+str(i),INT)
    s['V'] = Symbol('V'+str(i),REAL)
    s['timer'] = Symbol('timer'+str(i), REAL)
    s['tempo'] = Symbol('tempo'+str(i), REAL)
    return s
```

Função `init` dado um estado do programa (um dicionário de variáveis), devolve um predicado do pySMT que testa se esse estado é um possível estado inicial do programa.

In [143…
```
def init(s):
    return And(Equals(s['v'], v0), Equals(s['V'], v0),
               Equals(s['m'], START),  Equals(s['timer'], Real(0)),
               Equals(s['tempo'], Real(0)), GE(s['v'], Real(0)),
               GE(s['V'], Real(0))
              )
```

A função `trans` que, dados dois estados do programa, devolve um predicado do pySMT que testa se é possível transitar do primeiro para o segundo estado

In [144…
```
def trans(curr, prox):
    ge_zero = And(GE(prox['v'], Real(-0.001)), GE(prox['V'], Real(-0.001)),
                  GE(prox['timer'], Real(-0.001)), GE(prox['tempo'], Real(-0.001) ))


    StartToFree = And(Equals(curr['m'], START), Equals(prox['m'], FREE),
                      Equals(prox['v'], curr['v']), Equals(prox['V'], curr['V']),
                      Equals(prox['timer'], curr['timer']),
                      Equals(prox['tempo'], curr['tempo']),
                      ge_zero
                     )

    FreeToFree = And(Equals(curr['m'], FREE), Equals(prox['m'], FREE),
                     Equals(prox['v'], curr['v'] + (-a)*P + c*(curr['V'] - curr['v'])),
                     Equals(prox['V'], curr['V'] + (-c)*(curr['V'] - curr['v'])),
                     Equals(prox['timer'], curr['timer']+(prox['tempo']-curr['tempo'])),
                     Equals(prox['tempo'], curr['tempo']+0.2),
                     LE(prox['timer'], timerlim),
                     ge_zero
                     )

    FreeToStopping = And(Equals(curr['m'], FREE), Equals(prox['m'], STOPPING),
                         Equals(prox['v'], curr['v'])  ,
                         Equals(prox['V'], curr['V']) ,
                         Equals(prox['timer'], Real(0)),
                         Equals(prox['tempo'], curr['tempo']),
                         GE(curr['timer'], timerlim),
                         ge_zero
                     )

    StoppingToStopping = And(Equals(curr['m'], STOPPING), Equals(prox['m'], STOPPING),
                             Equals(prox['v'], curr['v'] +
                             ((-a)*P) + (cSTOPPING*(curr['V'] - curr['v']))),
                             Equals(prox['V'], curr['V'] +
                             (-cSTOPPING)*(curr['V'] - curr['v'])),
                             Equals(prox['timer'], Real(0)),
                             Equals(prox['tempo'], curr['tempo']+0.2),
```

```
                        ge_zero
                       )

    StoppingToBlocked = And(Equals(curr['m'], STOPPING), Equals(prox['m'], BLOCKED),
                        Equals(prox['v'], curr['v']),
                        Equals(prox['V'], curr['V']),
                        Equals(curr['V'], curr['v']),
                        Equals(prox['timer'], Real(0)),
                        Equals(prox['tempo'], curr['tempo']),
                        ge_zero
                       )


    BlockedToBlocked = And(Equals(curr['m'], BLOCKED), Equals(prox['m'], BLOCKED),
                      Equals(prox['v'], curr['v'] + (-a)*P),
                      Equals(prox['V'], curr['V']),
                      Equals(prox['timer'], curr['timer']+(prox['tempo']-curr['tempo'])),
                      Equals(prox['tempo'], curr['tempo']+0.2),
                      LE(curr['timer'], timerlim),
                      ge_zero
                      )

    BlockedToStopped = And(Equals(curr['m'], BLOCKED), Equals(prox['m'], STOPPED),
                        Equals(prox['v'], Real(0)),
                        Equals(prox['V'], Real(0)) ,
                        Equals(prox['timer'], Real(0)),
                        GE(curr['timer'], timerlim),
                        Equals(prox['tempo'], curr['tempo']),
                        ge_zero
                       )

    BlockedToFree =    And(Equals(curr['m'], BLOCKED), Equals(prox['m'], FREE),
                        Equals(prox['v'], curr['v']),
                        Equals(prox['V'], curr['V']) ,
                        Equals(prox['timer'], Real(0)),
                        GE(curr['timer'], timerlim),
                        Equals(prox['tempo'], curr['tempo']),
                        ge_zero
                       )

    StoppedToStopped = And(Equals(curr['m'], STOPPED), Equals(prox['m'], STOPPED),
                        Equals(prox['v'], Real(0))  ,
                        Equals(prox['V'], Real(0)) ,
                        Equals(prox['timer'], Real(0)),
                        GE(curr['timer'], Real(0)),
                        Equals(prox['tempo'], curr['tempo']),
                        ge_zero
                       )

    EverywhereToStopped = And(Equals(prox['m'], STOPPED),
                        Equals(curr['v'], Real(0))  ,
                        Equals(curr['V'], Real(0)) ,
                        Equals(prox['tempo'], curr['tempo']),
                        ge_zero
                       )

    return Or(StartToFree, FreeToFree, FreeToStopping, StoppingToStopping,
              StoppingToBlocked, BlockedToBlocked, BlockedToStopped, BlockedToFree,
              StoppedToStopped, EverywhereToStopped)
```

A função `gera_traco`, dada uma função que gera uma cópia das variáveis do estado, um predicado que testa se um estado é inicial, um predicado que testa se um par de estados é uma transição válida, e um número positivo k gera um possível traço de execução do programa de tamanho k.

In [145…

```
def gera_traco(declare,init,trans,k):
    l = ['START', 'FREE', 'STOPPING', 'BLOCKED', ' STOPPED']
    with Solver(name="z3") as s:

        #cria k cópias do estado
        trace = [declare(i) for i in range(k+1)]
        #cria o traço
        s.add_assertion( init(trace[0]) )
        for i in range(k):
            s.add_assertion(trans(trace[i], trace[i+1]))

        if s.solve():
            for i in range(k+1):
                print()
                print("Estado",i)
                m = s.get_py_value( trace[i]['m'])
                print(l[m])
                for v in trace[i]:
                    if v not in ['m']:
                        aux = s.get_py_value(trace[i][v])
                        if aux < 0:
                            aux = aux*aux
                        print(f'{v} -> ' + '%.2f' % float(aux))
        else:
            print('unsat')

gera_traco(declare,init,trans,10)

print('\n')
```

```
Estado 0
START
v -> 20.00
V -> 20.00
timer -> 0.00
tempo -> 0.00

Estado 1
FREE
v -> 20.00
V -> 20.00
timer -> 0.00
tempo -> 0.00

Estado 2
FREE
v -> 10.00
V -> 20.00
timer -> 0.20
tempo -> 0.20

Estado 3
STOPPING
v -> 10.00
V -> 20.00
timer -> 0.00
tempo -> 0.20

Estado 4
STOPPING
v -> 10.00
V -> 10.00
timer -> 0.00
tempo -> 0.40

Estado 5
```

```
BLOCKED
v -> 10.00
V -> 10.00
timer -> 0.00
tempo -> 0.40

Estado 6
BLOCKED
v -> 0.00
V -> 10.00
timer -> 0.20
tempo -> 0.60

Estado 7
 STOPPED
v -> 0.00
V -> 0.00
timer -> 0.00
tempo -> 0.60

Estado 8
 STOPPED
v -> 0.00
V -> 0.00
timer -> 0.20
tempo -> 0.60

Estado 9
 STOPPED
v -> 0.00
V -> 0.00
timer -> 0.00
tempo -> 0.60

Estado 10
 STOPPED
v -> 0.00
V -> 0.00
timer -> 0.00
tempo -> 0.60
```

A seguinte função `bmc_alwaysVD`, dada uma função que gera uma cópia das variáveis do estado, um predicado que testa se um estado é inicial, um predicado que testa se um par de estados é uma transição válida, um invariante a verificar, e um número positivo de passos K, averigua se o candidato a invariante é ou não válido.

In [146…

```python
def bmc_alwaysVD(declare,init,trans,inv,K):

    for k in range(K+1):
        with Solver(name="z3") as s:
            trace = [declare(i) for i in range(k+1)]

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

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


            s.add_assertion(Not(And([inv(trace[i], trace[i+1]) for i in range(k)])))

            if s.solve():
                print("A velocidade aumentou em algum passo.")
```

```
                return

            print(f'A velocidade diminuiu em todos os {K} passos.')
            return

def VDiminui(state1, state2):
    return GE(state1['V'], state2['V'])

bmc_alwaysVD(declare,init,trans,VDiminui,10)
```

```
A velocidade diminuiu em todos os 10 passos.
```

In [147…

```
global T
T = 1.5

def bmc_alwaysIM(declare,init,trans,inv,K):

    for k in range(1,K+1):
        with Solver(name="z3") as s:
            trace = [declare(i) for i in range(k+1)]

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

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


            s.add_assertion(Not(And(inv(trace[k]))))

            if s.solve():
                print(f'O veículo imobilizou-se em {float(T)} segundos ou mais.')
                return

    print(f'O veículo imobilizou-se em menos de {float(T)} segundos.')

def imobilizaT(state):
    return LE(state['tempo'], Real(T))

bmc_alwaysIM(declare,init,trans,imobilizaT, 10)
```

```
O veículo imobilizou-se em menos de 1.5 segundos.
```