TP2

Grupo 9

Exercicio 1

Este exercício consiste na alteração das funções bmc_always e bmc_eventually,desenvolvidas na aula, tirando partido partido das técnicas de solving incremental do Z3(usando push e pop)

In [26]:
```
def minha_always(declare,init,trans,inv,K):
    s = Solver()
    trace = []
    trace.append(declare(0))
    #verificação do estado inicial
    s.add(init(trace[0]))
    s.push()

    #verificação do invariante. Negamos o que queremos provar, se der sat então é porque falha
    s.add(Not(inv(trace[0])))
    if s.check() == sat:
        print("Propriedade não válida")
        return

    else:
        s.pop()
        for i in range(1,K+1):
            trace.append(declare(i))

            #verificação da transição entre 2 estados
            s.add(trans(trace[i-1],trace[i]))
            s.push()
            s.add(Not(inv(trace[i])))
            if s.check() == sat:
                print("Propriedade não é válida.")
                print("Um traço em que falha é o seguinte:")
                m = s.model()
                print("State:\t",i)
                print("pc =\t",m[trace[i]['pc']])
                print("x =\t",m[trace[i]['x']])
                print("m =\t",m[trace[i]['m']])
                print("n =\t",m[trace[i]['n']])
                print("r =\t",m[trace[i]['r']])
                print("y =\t",m[trace[i]['y']])
                return
            else:
                s.pop()

        print("A propriedade é válida em traços de comprimento até " + str(K))
```

In [27]:
```
def minha_eventually(declare,init,trans,prop,bound):
    s = Solver()
    trace = []
    trace.append(declare(0))

    #verificação do estado inicial
    s.add(init(trace[0]))

    for i in range(1,bound):
        trace.append(declare(i))
        s.add(trans(trace[i-1],trace[i]))
        s.push()

        s.add(Not(prop(trace[i])))

        #restrição para forçar a existencia do loop
        s.add(Or([trans(trace[i-1],trace[j]) for j in range(i) ]))

        if s.check() == sat:
            m = s.model()
            for j in range(i):
                if(m.eval(trans(trace[i-1],trace[j]))):
                    print ("O loop começa aqui")
                    m = s.model()
                    print("State:\t",j)
                    print("pc =\t",m[trace[j]['pc']])
                    print("x =\t",m[trace[j]['x']])
                    print("m =\t",m[trace[j]['m']])
                    print("n =\t",m[trace[j]['n']])
                    print("r =\t",m[trace[j]['r']])
                    print("y =\t",m[trace[j]['y']])
            return
        else:
            s.pop()


    print("A propriedade [F prop] é válida em traços de comprimento até " + str(bound))
```

Exercício 2

Consideramos o seguinte programa de multiplicação de 2 inteiros de precisão limitada a 16 bits

Consideramos o seguinte programa de multiplicação de 2 inteiros de precisão limitada a 16 bits

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
while y > 0:
    if y & 1 == 1:
        y = y-1
        r = r+x
    x = x<<1
    y = y>>1
assert r == m * n
```

Definindo as funções `init` (que verifica se um estado é inicial) e `trans` (que verifica se é possível transitar entre 2 quaisquer estados), podemos usar um SMT solver (nomeadamente o Z3) para, por exemplo, gerar uma possível execução de $k-1$ passos do programa (em que $k > 0$). Para tal precisamos de criar k cópias das variáveis que caracterizam o estado do FOTS e depois impor que a primeira cópia satisfaz o predicado inicial e que cada par de cópias consecutivas satisfazem o predicado de transição.

A função declare cria a i-ésima cópia das variáveis de estado, agrupadas num dicionário que nos permite aceder às mesmas pelo nome.

In [28]:
```python
from z3 import *

def declare(i):
    state = {}
    state['pc'] = BitVec('pc',16)
    state['m'] = BitVec('m',16)
    state['n'] = BitVec('n',16)
    state['r'] = BitVec('r',16)
    state['x'] = BitVec('x',16)
    state['y'] = BitVec('y',16)
    return state
```

Definimos agora um invariante do sistema, que varia conforme a componente pc de um estado. Caso pc tenha os valores 0 ou 6 o invariante vai ser diferente do que se o pc tiver valores entre 1 e 5

In [29]:
```python
def inv(state):
    l = state['pc']
    if l == 0 or l == 6:
        return And(state['x'] * state['y'] + state['r'] == state['m'] * state['n'])
    else:
        return (state['y'] >= 0)
```

Agora definimos a inicialização do modelo. O programa no início começa com o pc a zero e o valor da variável x tem de ser igual ao valor da variável m e maior ou igual que zero, quanto à variável y tem de ser igual à variável n e maior ou igual a zero, por fim a variável r tem de ser zero

In [30]:
```python
def init(state):
    return And(state['m'] >= 0,state['n'] >= 0,state['x'] == state['m'],state['y'] == state['n'],state['pc'] == 0,state['r'] == 0)
```

Agora passamos a tratar da função de transição que, dados dois possíveis estados do programa, devolva um predicado Z3 que testa se é possível transitar do primeiro para o segundo:

In [31]:
```python
def trans(curr,prox):
    x = curr['x']
    y = curr['y']
    r = curr['r']
    pc = curr['pc']
    nx = prox['x']
    ny = prox['y']
    nr = prox['r']
    npc = prox['pc']

    a = And(pc == 0,y > 0,npc == 1,ny == y,nx == x,nr == r)
    b = And(pc == 1, y & 1 == 1,npc == 2,ny == y,nx == x,nr == r)
    h = And(pc == 1, y & 1 == 0, npc == 4, nx == x, nr == r, ny == y)
    c = And(pc == 2,npc == 3,ny == y-1,nr == r,nx == x )
    d = And(pc == 3,npc == 4, ny == y, nx == x, nr == r+x)
    e = And(pc == 4,npc == 5, nx == 2*x,ny == y, nr == r)
    f = And(pc == 5, npc == 0, nx == x, nr == r, ny == y/2)
    g = And(pc == 0, y == 0, npc == 6, nx == x, ny == y, nr == r)
    i = And(pc == 6, ny == y, nx == x, nr == r, npc == 6)


    return Or(a,b,h,c,d,e,f,g,i)
```

Correção Parcial

Vamos usar a função minha_always para fazer a verificação parcial do programa, usando o invariante acima definido, as funções init e trans e também um K = 20 para definir o número de estados em que estamos a procurar.

In [32]:
```python
minha_always(declare,init,trans,inv,30)
```

```
A propriedade é válida em traços de comprimento até 30
```

Verificação da terminação do programa

Podemos verificar a terminação de um programa definindo uma condição que nos diz que eventualmente o programa termina

No caso do programa acima, podemos usar como variante $pc = 6$

No caso do programa acima , podemos usar como variante $pc = 6$.

In [33]:
```
def variante(state):
    return (state['pc'])

def terminacao(state):
    return variante(state) == 6
```

In [34]:
```
minha_eventually(declare,init,trans,terminacao,20)
```

```
A propriedade [F prop] é válida em traços de comprimento até 20
```

Eficiência

Em termos de eficiência, a estratégia que usa o push e o pop é muito mais eficiente. Porque o metodo lecionado na aula fazia sucessivamente a verificação de estados que já foram verificados e isso, para além de desnecesário, acaba por tornar o programa menos efeciente. Usando o push e o pop não precisamos de voltar a verificar o que já foi verificado, tornando o programa mais rápido(mais eficiente). Usando a função time() é possivel verificar a diferença entre as duas estratégias. Com um k=20 verificou-se uma diferença de 3 segundos enquanto que com um k=30 a diferença foi de 9 segundos. Como tal facilmente se conclui que quanto maior o k utilizado, maior será intervalo entre o tempo de execução dos dois algoritmos, e mais vantajoso será usar a alternativa que tira partido das técnicas de solving incremental do Z3 (push/pop).

Exercício 3

Considerando o mesmo programa do exercicio 2, para se provar as duas proprieadades do exercicio anterior, utilizaremos um procedimento de verificação alternativo, a k-indução.

In [35]:
```
def kinducao(declare,init,trans,inv,k):
    s = Solver()

    #criar k copias do estado, no traço
    trace = []
    for i in range(k+1):
        trace.append(declare(i))

    s.add(init(trace[0]))
    #testar se é válido para k estados
    for i in range(k-1):
        s.add(trans(trace[i],trace[i+1]))
    s.add(Or( [ Not(inv(trace[i])) for i in range(k) ]))

    if s.check() == sat:
        m = s.model()
        print("A propriedade falha nos seguintes casos de base")
        for i in range(k):
            if m.eval(inv(trace[i])):
                print("pc =\t",m[trace[i]['pc']].as_long())
                print("x =\t",m[trace[i]['x']])
        return

    #testar inv no passo 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])))

    if s.check() == sat:
        m = s.model()
        print("A propriedade falha nos seguintes casos de base")
        for i in range(k):
            if m.eval(inv(trace[i])):
                print("pc =\t",m[trace[i]['pc']].as_long())
                print("x =\t",m[trace[i]['x']])
        return

    print("A propriedade é válida")
```

Correção Parcial

Vamos usar a função k-inducao para fazer a verificação parcial do programa, usando o invariante acima definido, as funções init e trans e também um $K = 3$ para definir o número de estados em que estamos a procurar.

In [36]:
```
kinducao(declare,init,trans,inv,2)
```

```
A propriedade é válida
```

Verificação da terminação do programa

Para provarmos que o programa termina precisamos de encontrar um variante V, sendo que:

- O variante é sempre positivo
- O variante descresce ou atinge o valor 0
- Quando o variante é 0 verifica-se necessariamente ϕ

Para o programa acima, a terminação pode ser expressa por $F\,(G\ \mathrm{pc} = 6)$

In [37]:
```
def variante(state):
    return If(state['pc']==6,0,state['y']*4 + 6 - state['pc'])
```

```python
def positivo(state):
    return variante(state) >= 0

kinducao(declare,init,trans,positivo,2)

def decrescente(state0):
    state1 = declare(-1)
    state2 = declare(-2)
    state3 = declare(-3)
    state4 = declare(-4)
    return ForAll(list(state2.values()),Implies(And(trans(state0,state1),trans(state1,state2),trans(state2,state3),trans
(state3,state4)),Or(variante(state4)<variante(state0),variante(state4)==0)))

kinducao(declare,init,trans,decrescente,2)

def util(state):
    return Implies(variante(state)==0, state['pc']==6)

kinducao(declare,init,trans,util,2)
```

```
A propriedade é válida
A propriedade é válida
A propriedade é válida
```

Observou-se que k=2 foi o menor valor possivel para se provar por k-indução as propriedades em questão.