ex2TP3

December 12, 2022

1 Exercício 2 (Inversores) - Trabalho Prático 3

Grupo 4: Carlos Costa-A94543Ruben Silva-A94633

2 Problema:

1. O seguinte 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.
    1. Cada inversor tem um bit s de estado, inicializado com um valor aleatório.
    2. Cada inversor é regido pelas seguintes transformações $\textbf{invert}(in, out)$ $x \leftarrow \mathsf{read}(\mathtt{in})$
    $s \leftarrow \neg x \ \parallel \ s \leftarrow s \oplus x$
    $\mathsf{write}(\mathtt{out}, s)$
    3. A escolha neste comando é sempre determinística; isto é, em cada inversor a escolha do comando a executar é sempre a mesma. Porém qual é essa escolha é determinada aleatoriamente na inicializarão do sistema.
    4. O estado do sistema é um duplo definido pelos 4 bits s, e é inicializado com um vetor aleatório em $\{0, 1\}^4$.
    5. O sistema termina em ERRO quando o estado do sistema for $(0, 0, 0, 0)$.
2. Construa um SFOTS que descreva este sistema e implemente este sistema, numa abordagem BMC ("bouded model checker") num traço com n estados.
3. Verifique se o sistema é seguro usando BMC, k-indução ou model checking com interpolantes.

3 Análise do Problema

Este é um problema sobre um SFOTS para um sistema com 4 inversores (A, B, C, D) que recebem um bit como input de um outro inversor. Cada inversor tem um funcionamento definido no início do programa: * $pc \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Program Counter. * $a \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Estado do funcionamento do inversor "a". * $b \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Estado do funcionamento do inversor "b". * $c \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Estado do funcionamento do inversor "c". * $d \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Estado do funcionamento do inversor "d". * $a' \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Próximo estado do funcionamento do inversor "a ". * $b' \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Próximo estado do funcionamento do inversor "b ". * $c' \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Próximo estado do funcionamento do inversor "c ". * $d' \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Próximo estado do funcionamento do inversor "d ". * $x_a \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ bit do inversor "a". * $x_b \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ bit do inversor "b". * $x_c \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ bit do inversor "c". * $x_d \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ bit do inversor "d". * $x'_a \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ próximo bit do inversor "a". * $x'_b \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ próximo bit do inversor "b". * $x'_c \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ próximo bit do inversor "c". * $x'_d \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ próximo bit do inversor "d". * $x_{var} \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ bit do inversor atual. * $x'_{var} \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ próximo bit do inversor atual. * $x_{varIn} \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ bit de entrada do inversor atual. * $x'_{varIn} \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ próximo bit de entrada do inversor atual. * $aTc \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Função lógica do inversor "a" que recebe bit do inversor “c” * $bTa \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Função lógica do inversor “b” que recebe bit do inversor “a” * $dTb \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Função lógica do inversor “d” que recebe bit do inversor “b” * $cTd \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Função lógica do inversor “c” que recebe bit do inversor “d”

4 Limitações e obrigações para o Inversor

1. **Condições iniciais:**
   1. inversor01:

$$(a = 0 \land ((x_{var} = 1 \land x_{varIn} = 0) \lor (x_{var} = 0 \land x_{varIn} = 1)))$$

   2. inversor02:

$$(a = 1 \land ((x_{var} = 0 \land x_{varIn} + x_{var} = 0) \lor (x_{var} = 1 \land x_{varIn} + x_{var} = 1) \lor (x_{var} = 0 \land x_{varIn} + x_{var} = 2)))$$

5 Limitações e obrigações para o SFTOS

1. **Condições iniciais:**
   1. aTc:
$$(inversor_{ac} \land a' = a)$$
   2. bTa:
$$(inversor_{ba} \land b' = b)$$
   3. dTb:
$$(inversor_{db} \land d' = d)$$
   4. cTd:
$$(inversor_{cd} \land c' = c)$$
   5. transita01:
$$(aTc \land bTa \land dTb \land cTd \land pc = 0 \land pc' = pc)$$
   6. transita02:

$$(x'_a = x_a \land x'_b = x_b \land x'_c = x_c \land x'_d = x_d \land a' = a \land b' = b \land c' = c \land d' = d \land pc = 2 \land pc' = 2)$$

   7. error01:

$$(aTc \land bTa \land dTb \land cTd \land x_a = 0 \land x_b = 0 \land x_c = 0 \land x_d = 0 \land pc = 0 \land pc' = 2)$$

6 Implementaçao do Problema

Importar o solver 1. Importar o PySmt. 2. Importar o itertools. 3. Importar a função randint da libraria random.

[ ]:
```
import itertools
from pysmt.shortcuts import *
from pysmt.typing import INT
from pysmt import *
from random import randint
```

7 Resolver o codigo

Função "gerarDadosInicias" 1. Return de uma lista de 4 inteiros entre 0 e 1

[ ]:
```
def gerarDadosInicias():
    return [randint(0,1),randint(0,1),randint(0,1),randint(0,1)]
```

Função "genState" Esta função é responsável pela declaração de todas as variáveis que serão utilizadas no solver. 1. Parâmetros: 1. *vars* -> Conjunto de variáveis para o programa 2. *s* -> O Nome do traço pretendido 3. *i* -> um inteiro que será responsável por dar o nr às variaveis 2. Função: 1. Inicialmente criamos um dicionário para colocar todas as variáveis necessárias. 2. Criamos v variáveis com o input do utilizador. 3. Return do novo dicionário com as variáveis.

[ ]:
```
def genState(vars,s,i):
    state = {}
    for v in vars:
        state[v] = Symbol(v+'!'+s+str(i), INT)
    return state
```

Função "init" Esta função é responsável pela inicialização do primeiro estado do traço e algumas condições lógicas necessárias 1. Parâmetros: 1. *state* -> Primeiro estado do traço 2. *switch* -> Lista com a inicialização do funcionamente de cada inversor 3. $inicial_n$ -> Lista com a inicialização do bit de cada inversor 2. Return de um "And" com a seguinte condição lógica: $(pc = 0 \land a = switch_0 \land b = switch_1 \land c = switch_2 \land d = switch_3 \land xa = n_0 \land xb = n_1 \land xc = n_2 \land xd = n_3)$

[ ]:
```
def init(state, switch, inicial_n):
    #Inicializar, tanto cada variavel, como determinar o funcionamento de cada␣
↪função
    return And(Equals(state['a'], Int(switch[0])),
               Equals(state['b'], Int(switch[1])),
               Equals(state['c'], Int(switch[2])),
               Equals(state['d'], Int(switch[3])),
               Equals(state['xa'], Int(inicial_n[0])),
               Equals(state['xb'], Int(inicial_n[1])),
               Equals(state['xc'], Int(inicial_n[2])),
               Equals(state['xd'], Int(inicial_n[3])),
               Equals(state['pc'], Int(0)))
```

Função "error" Esta função é responsável por: dado um estado do programa, devolve um predicado do pySMT que testa se esse estado é um possível estado de erro do programa. 1. Parâmetros: 1. *state* -> Primeiro membro do dicionário principal da função 2. Return de um "And" com a seguinte condição lógica: $(pc = 2)$

[ ]:
```
def error(state):
    return And(Equals(state["pc"],Int(2)))
```

Função "inversor" Esta função é responsável pela criação das conexões lógicas necessárias para o FOTS fazer sentido e ser o pretendido 1. Parâmetros: 1. *curr* -> Membro atual do dicionário principal da função 2. *prox* -> Membro seguinte ao atual do dicionário principal da função 3. *var*

-> Varíavel do inversor atual 4. $varIn$ -> Varíavel que está a passar o bit à atual 2. Funções do inversor. Cada maneira de o inversor funcionar é determinado no inicio do programa e mantém-se durante toda a execução.. 1. inversor01: $(a = 0 \wedge ((x_{var} = 1 \wedge x_{varIn} = 0) \vee (x_{var} = 0 \wedge x_{varIn} = 1)))$ 2. inversor02: $(a = 1 \wedge ((x_{var} = 0 \wedge x_{varIn} + x_{var} = 0) \vee (x_{var} = 1 \wedge x_{varIn} + x_{var} = 1) \vee (x_{var} = 0 \wedge x_{varIn} + x_{var} = 2)))$ 3. Return de um "And" com a seguinte condição lógica: (\$ inversor01 inversor02\$)

[ ]:
```
def inversor(curr,prox, var, varIn):
    #var Representa o inversor atual
    #varIn representa o bit do inversor anterior

    #~x
    inversor01 = And(Equals(curr[var],Int(0)),
                     Or(And(Equals(prox["x"+var], Int(1)),␣
↪Equals(curr["x"+varIn],Int(0))),
                        And(Equals(prox["x"+var], Int(0)),␣
↪Equals(curr["x"+varIn],Int(1)))))

    #x OP s
    inversor02 = And(Equals(curr[var],Int(1)),
                    Or(And(Equals(prox["x"+var], Int(0)),␣
↪Equals(Plus(curr["x"+varIn],curr["x"+var]),Int(0))),
                        And(Equals(prox["x"+var], Int(1)),␣
↪Equals(Plus(curr["x"+varIn],curr["x"+var]),Int(1))),
                        And(Equals(prox["x"+var], Int(0)),␣
↪Equals(Plus(curr["x"+varIn],curr["x"+var]),Int(2)))))

    return Or(inversor01,inversor02)
```

Função "trans" Esta função é responsável pela criação das conexões lógicas necessárias para o FOTS fazer sentido e ser o pretendido 1. Parâmetros: 1. $curr$ -> Membro atual do dicionário principal da função 2. $prox$ -> Membro seguinte ao atual do dicionário principal da função 2. Função: 1. Criamos as condições lógicas de cada inversor individualmente: 1. aTc: $(inversor_{ac} \wedge a' = a)$ 2. bTa: $(inversor_{ba} \wedge b' = b)$ 3. dTb: $(inversor_{db} \wedge d' = d)$ 4. cTd: $(inversor_{cd} \wedge c' = c)$ 2. Criamos as condições lógicas chamadas transita: 1. transita01: $(aTc \wedge bTa \wedge dTb \wedge cTd \wedge pc = 0 \wedge pc' = pc)$ 2. transita02: $(x'_a = x_a \wedge x'_b = x_b \wedge x'_c = x_c \wedge x'_d = x_d \wedge a' = a \wedge b' = b \wedge c' = c \wedge d' = d \wedge pc = 2 \wedge pc' = 2)$ 3. erro01: $(aTc \wedge bTa \wedge dTb \wedge cTd \wedge x_a = 0 \wedge x_b = 0 \wedge x_c = 0 \wedge x_d = 0 \wedge pc = 0 \wedge pc' = 2)$ 3. Return de um "And" com a seguinte condição lógica: (\$ transita01 transita02 transita03\$)

[ ]:
```
def trans(curr, prox):
    #esta 4 variáveis contem as condições lógica de cada inversor
    aTc = And(inversor(curr,prox,"a","c"), Equals(prox["a"],curr["a"]))
    bTa = And(inversor(curr,prox,"b","a"), Equals(prox["b"],curr["b"]))
    dTb = And(inversor(curr,prox,"d","b"), Equals(prox["d"],curr["d"]))
    cTd = And(inversor(curr,prox,"c","d"), Equals(prox["c"],curr["c"]))

    #transita01 é responsável pela execução padrão do código
    transita01 = And(aTc,bTa,dTb,cTd,
```

```
                        Equals(curr["pc"],Int(0)),
                        Equals(prox["pc"],curr["pc"]))


    #transita03 é responsável por manter o código parado caso passe pelo estado␣
↪de erro
    transita02 = And(Equals(prox["xa"],curr["xa"]),
                        Equals(prox["xb"],curr["xb"]),
                        Equals(prox["xc"],curr["xc"]),
                        Equals(prox["xd"],curr["xd"]),
                        Equals(prox["a"],curr["a"]),
                        Equals(prox["b"],curr["b"]),
                        Equals(prox["c"],curr["c"]),
                        Equals(prox["d"],curr["d"]),
                        Equals(curr["pc"] , Int(2)),
                        Equals(prox["pc"] , Int(2)))

    #transita02 é responsável por marcar os erros -> (0,0,0,0)
    error01 = And(aTc,bTa,dTb,cTd,
                        Equals(curr["xa"] , Int(0)),
                        Equals(curr["xb"] , Int(0)),
                        Equals(curr["xc"] , Int(0)),
                        Equals(curr["xd"] , Int(0)),
                        Equals(curr["pc"] , Int(0)),
                        Equals(prox["pc"] , Int(2)))
    return Or(transita01,transita02, error01)
```

Função "bmc" Esta é a função para testar o algoritmo em BMC e gerar o traço pretendido e com ele tabelar o output 1. Parâmetros: 1. *vars* -> Variáveis do algoritmo 2. *init* -> Função init 3. *trans* -> Função trans 4. *error* -> Função error 5. *n* -> Quantidade de estados (input do utilizador) 2. Função: 1. Iniciamos o Solver 2. Criamos uma lista com o estado inicial e o bit de cada inversor 3. Geramos o traço 5. Geramos o estado inicial do traço (I) 6. Geramos o resto do traço sem o init (Tks) 7. Resolvemos o problema com: $I \wedge Tks$

[ ]:
```
def bmc(vars,init,trans,error,n):
    #Escolhe aleatoriamente entre
    # (~x) 0
    # (s OP x) 1
    inicial_n = gerarDadosInicias()

    #Escolher aleatoriamente os bits de cada inversor
    switch =gerarDadosInicias()

    with Solver(name="z3") as s:

        #Abordagem BMC
        for n1 in range(1,n+1):
```

```
            #Gerar o Traço inicial
            X = [genState(vars,'X',i) for i in range(n1)]   # cria n+1 estados␣
 ↪(com etiqueta X)

            #Criar o Estado inicial e colocar-lo numa Constante
            I = init(X[0], switch, inicial_n)

            #Gerar o Traço
            Tks = [trans(X[i],X[i+1]) for i in range(n1-1)]

            #Testar o Traço com a constante que tem o estado inicial
            if s.solve([I,And(Tks)]):      # testa se I /\ T^n  é satisfazível
                if (n1 == n):
                    for i in range(n1):
                        if s.get_value(X[i]["pc"])==Int(2):
                            print("Error")
                            return
                        print("Estado: "+str(i))
                        for v in X[i]:
                            print("          ",v,'=',s.get_value(X[i][v]))
```

Funções de auxílio Para auxiliar na implementação deste algoritmo, começamos por definir três funções. 1. A função *baseName* cria o nome base de uma fórmula 2. A função *rename* renomeia uma fórmula (sobre um estado) de acordo com um dado estado. 3. A função *same* testa se dois estados são iguais. 4. A função *invert* codifica a relação de transição e devolve a relação e transição inversa

[ ]:
```
def baseName(s):
    return ''.join(list(itertools.takewhile(lambda x: x!='!', s)))

def rename(form,state):
    vs = get_free_variables(form)
    pairs = [ (x,state[baseName(x.symbol_name())]) for x in vs ]
    return form.substitute(dict(pairs))

def same(state1,state2):
    return And([Equals(state1[x],state2[x]) for x in state1])

def invert(trans):
    return (lambda c, p: trans(p,c))
```

Função "model_checking" Esta é para verificar a segurança do codigo 1. Parâmetros: 1. *vars* -> Variáveis do algoritmo 2. *genState* -> Função genState 3. *init* -> Função init 4. *trans* -> Função trans 5. *error* -> Função error 6. *n* -> Quantidade de estados (input do utilizador) 2. Função: 1. Segue a lógica da criação dos interpolantes usado no exercício anterior

[ ]:
```python
def model_checking(vars, genState, init, trans, error, N, M):
    with Solver(name="z3") as s:

            switch =gerarDadosInicias()
            inicial_n = gerarDadosInicias()

            # Declarar todas as variaveis em cada traço especifico
            traceA = [genState(vars, 'A', i) for i in range(N+1)]
            traceB = [genState(vars, 'B', i) for i in range(M+1)]

            # Estabelecer a ordem pela qual os pares (n,m) vão surgir. Por␣
↪exemplo:
            order = sorted([(a, b) for a in range(1, N+1)
                              for b in range(1, M+1)], key=lambda tup:␣
↪tup[0]+tup[1])

            #Resolver o problema
            #Passo 1 e 2
            for (n, m) in order:
                #Criar Rn
                Tn = And([trans(traceA[i], traceA[i+1]) for i in range(n)])
                I = init(traceA[0], switch, inicial_n)
                Rn = And(I, Tn)

                #Criar Bm
                Bm = And([invert(trans)(traceB[i], traceB[i+1]) for i in␣
↪range(m)])
                E = error(traceB[0])
                Um = And(E, Bm)

                #Criar Vnm
                Vnm = And(Rn, same(traceA[n], traceB[m]), Um)

                if s.solve([Vnm]):
                    print("unsafe")
                    return
                else:
                    C = binary_interpolant(And(Rn, same(traceA[n], traceB[m])),␣
↪Um)
                    #Interpolante nao existe
                    if C is None:
                        break
                    C0 = rename(C, traceA[0])
                    C1 = rename(C, traceA[1])
                    T = trans(traceA[0], traceA[1])

                    if not s.solve([C0, T, Not(C1)]):
```

```
                        print("safe")
                        return
                    else:
                        S = rename(C, traceA[n])
                        while True:
                            A = And(S, trans(traceA[n], traceB[m]))
                            if s.solve([A, Um]):
                                print("Nao é possivel encontrar um majorante")
                                break
                            else:
                                Cnew = binary_interpolant(A, Um)
                                Cn = rename(Cnew, traceA[n])
                                if s.solve([Cn, Not(S)]):
                                    S = Or(S, Cn)
                                else:
                                    print("safe")
                                    return
```

[ ]:
```
vars = ["a","b","c","d","pc", "xa","xb","xc","xd"]
n = 8   #tamanho do traço
bmc(vars, init, trans, error, n)
```

```
Estado: 0
           a = 1
           b = 1
           c = 1
           d = 0
           pc = 0
           xa = 0
           xb = 0
           xc = 0
           xd = 1
Estado: 1
           a = 1
           b = 1
           c = 1
           d = 0
           pc = 0
           xa = 0
           xb = 0
           xc = 1
           xd = 1
Estado: 2
           a = 1
           b = 1
           c = 1
           d = 0
           pc = 0
```

```
        xa = 1
        xb = 0
        xc = 0
        xd = 1
Estado: 3
        a = 1
        b = 1
        c = 1
        d = 0
        pc = 0
        xa = 1
        xb = 1
        xc = 1
        xd = 1
Estado: 4
        a = 1
        b = 1
        c = 1
        d = 0
        pc = 0
        xa = 0
        xb = 0
        xc = 0
        xd = 0
Estado: 5
        a = 1
        b = 1
        c = 1
        d = 0
        pc = 0
        xa = 0
        xb = 0
        xc = 0
        xd = 1
Estado: 6
        a = 1
        b = 1
        c = 1
        d = 0
        pc = 0
        xa = 0
        xb = 0
        xc = 1
        xd = 1
Estado: 7
        a = 1
        b = 1
        c = 1
```

```
d = 0
pc = 0
xa = 1
xb = 0
xc = 0
xd = 1
```

[ ]:
```
vars = ["a","b","c","d","pc", "xa","xb","xc","xd"]
n = 40
m=40
model_checking(vars, genState, init, trans, error, n, m)
```

```
---------------------------------------------------------------------------
NoSolverAvailableError                    Traceback (most recent call last)
Cell In [12], line 4
      2 n = 40
      3 m=40
----> 4 model_checking(vars, genState, init, trans, error, n, m)

Cell In [10], line 36, in model_checking(vars, genState, init, trans, error, N,
 ↪M)
     34     return
     35 else:
---> 36     C = binary_interpolant(And(Rn, same(traceA[n], traceB[m])), Um)
     37     #Interpolante nao existe
     38     if C is None:

File c:
 ↪\Users\ruben\AppData\Local\Programs\Python\Python310\lib\site-packages\pysmt\shortcuts.
 ↪py:1153, in binary_interpolant(formula_a, formula_b, solver_name, logic)
   1149         warnings.warn("Warning: Contextualizing formula during "
   1150                       "binary_interpolant")
   1151         formulas[i] = env.formula_manager.normalize(f)
-> 1153 return env.factory.binary_interpolant(formulas[0], formulas[1],
   1154                                       solver_name=solver_name,
   1155                                       logic=logic)

File c:
 ↪\Users\ruben\AppData\Local\Programs\Python\Python310\lib\site-packages\pysmt\factory.
 ↪py:562, in Factory.binary_interpolant(self, formula_a, formula_b, solver_name␣
 ↪logic)
    559     _And = self.environment.formula_manager.And
    560     logic = get_logic(_And(formula_a, formula_b))
--> 562 with self.Interpolator(name=solver_name, logic=logic) as itp:
    563     return itp.binary_interpolant(formula_a, formula_b)

File c:
 ↪\Users\ruben\AppData\Local\Programs\Python\Python310\lib\site-packages\pysmt\factory.
 ↪py:452, in Factory.Interpolator(self, name, logic)
```

```
    451 def Interpolator(self, name=None, logic=None):
--> 452     return self.get_interpolator(name=name, logic=logic)

File c:
 ↪\Users\ruben\AppData\Local\Programs\Python\Python310\lib\site-packages\pysmt\factory.
 ↪py:132, in Factory.get_interpolator(self, name, logic)
    130 def get_interpolator(self, name=None, logic=None):
    131     SolverClass, closer_logic = \
--> 132        self._get_solver_class(solver_list=self._all_interpolators,
    133                               solver_type="Interpolator",
    134                               preference_list=self.
 ↪interpolation_preference_list,
    135                               default_logic=self.
 ↪_default_interpolation_logic,
    136                               name=name,
    137                               logic=logic)
    139     return SolverClass(environment=self.environment,
    140                        logic=closer_logic)

File c:
 ↪\Users\ruben\AppData\Local\Programs\Python\Python310\lib\site-packages\pysmt\factory.
 ↪py:146, in Factory._get_solver_class(self, solver_list, solver_type,␣
 ↪preference_list, default_logic, name, logic)
    143 def _get_solver_class(self, solver_list, solver_type, preference_list,
    144                       default_logic, name=None, logic=None):
    145     if len(solver_list) == 0:
--> 146         raise NoSolverAvailableError("No %s is available" % solver_type)
    148     logic = convert_logic_from_string(logic)
    149     if name is not None:

NoSolverAvailableError: No Interpolator is available
```