ex1TP3

December 12, 2022

1 Exercício 1 (Model-Checking) - Trabalho Prático 3

Grupo 4: Carlos Costa-A94543Ruben Silva-A94633

2 Problema:

1. Pretende-se construir uma implementação simplificada do algoritmo "model checking" orientado aos interpolantes seguindo a estrutura apresentada nos apontamentos onde no passo (n, m) na impossibilidade de encontrar um interpolante invariante se dá ao utilizador a possibilidade de incrementar um dos índices n e m à sua escolha. Pretende-se aplicar este algoritmo ao problema da da multiplicação de inteiros positivos em `BitVec` (apresentado no TP2).

3 Análise do Problema

Este é um problema sobre a desenvolvimento do Model-Checking usando o "Control Flow Automaton (CFA)" do trabalho prático 2.

- $pc \in \mathbb{N}_0; \rightarrow$ Program Counter.
- $x \in \mathbb{N}_0; \rightarrow$ Variável "a" do input do utilizador.
- $y \in \mathbb{N}_0; \rightarrow$ Variável "b" do input do utilizador.
- $z \in \mathbb{N}_0; \rightarrow$ Variável que dará o output.
- $x' \in \mathbb{N}_0; \rightarrow$ Variável "a " que representa o próximo número do traço.
- $y' \in \mathbb{N}_0; \rightarrow$ Variável "b " que representa o próximo número do traço.
- $z' \in \mathbb{N}_0; \rightarrow$ Variável "z " que representa o próximo número do traço.
- $n \in \mathbb{N}_0; \rightarrow$ Representa a precisão dos bits.
- $maxN \in \mathbb{N}_0; \rightarrow$ Representa o número máximo possível com a precisão dada.
- $OFlow \in \mathbb{N}_0; \rightarrow$ Condição lógica do overflow.

4 Limitações e obrigações para o Control Flow Automation

1. **Condições iniciais:**
$$pc = 0 \wedge x = a \wedge y = b \wedge z = 0 \wedge b >= 0$$
2. **OFlow**
$$x >= 2^n$$
3. **Condição para parar o ciclo:**
$$y = 0 \wedge pc = 0 \wedge pc' = 1 \wedge x' = x \wedge y' = y \wedge z' = z$$

4. y**!=0 e Par:**

$$y! = 0 \wedge mod(y) = 0 \wedge x' = 2 * x \wedge y' = y/2 \wedge z' = z \wedge pc' = 3 \wedge pc = 0$$

5. y**!=0 e Impar:**

$$y! = 0 \wedge mod(y) = 1 \wedge x' = x \wedge y' = y - 1 \wedge z' = z + x \wedge pc' = 3 \wedge pc = 0$$

6. **Não fazer mais nada após o ciclo parar:**

$$((pc = 1 \wedge pc' = 1) \vee (pc = 2 \wedge pc' = 2)) \wedge x' = x \wedge y' = y \wedge z' = z$$

7. **Decidir se existe overflow ou não**

$$pc = 3 \wedge ((Oflow \wedge pc' = 2) \vee (!Oflow \wedge pc' = 3)) \wedge x' = x \wedge y' = y \wedge z' = z$$

5 Lógica do Model-Checking

O algoritmo de "model-checking" manipula as fórmulas $\mathsf{R}_n \equiv \mathsf{I} \wedge \mathsf{T}^n$ e $\mathsf{U}_m \equiv \mathsf{E} \wedge \mathsf{B}^m$ fazendo crescer os índices n, m de acordo com as seguintes regras

---

1. Inicia-se $n = 0$, $\mathsf{R}_0 = \mathsf{I}$ e $\mathsf{U}_0 = \mathsf{E}$.
2. No estado (n, m) tem-se a certeza que em todos os estados anteriores não foi detectada nenhuma justificação para a insegurança do SFOTS. Se $\mathsf{V}_{n,m} \equiv \mathsf{R}_n \wedge (X_n = Y_m) \wedge \mathsf{U}_m$ é satisfazível o sistema é inseguro e o algoritmo termina com a mensagem **unsafe**.
3. Se $\mathsf{V}_{n,m} \equiv \mathsf{R}_n \wedge (X_n = Y_m) \wedge \mathsf{U}_m$ for insatisfazível calcula-se C como o interpolante do par $(\mathsf{R}_n \wedge (X_n = Y_m), \mathsf{U}_m)$. Neste caso verificam-se as tautologias $\mathsf{R}_n \rightarrow C(X_n)$ e $\mathsf{U}_m \rightarrow \neg C(Y_m)$.
4. Testa-se a condição $\mathsf{SAT}(C \wedge \mathsf{T} \wedge \neg C') = \emptyset$ para verificar se C é um invariante de T; se for invariante então, pelo resultado anterior, sabe-se que $\mathsf{V}_{n',m'}$ é insatisfazível para todo $n' \geq n$ e $m' \geq n$. O algoritmo termina com a mensagem **safe**.
5. Se C não for invariante de T procura-se encontrar um majorante $S \supseteq C$ que verifique as condições do resultado referido: seja um invariante de T disjunto de U_m.
6. Se for possível encontrar tal majorante S então o algoritmo termina com a mensagem **safe**. Se não for possível encontrar o majorante pelo menos um dos índices n, m é incrementado, os valores das fórmulas R_n, U_m são actualizados e repete-se o processo a partir do passo 2.

Para encontrar um majorante S A parte crítica é o passo 5. Várias estratégias são possíveis (veremos algumas mais tarde). Uma solução possível é um algoritmo iterativo que tenta encontrar um invariante S pelos passos seguintes

1. S é inicializado com $C(X_n)$
2. Faz-se $A \equiv S(X_n) \wedge \mathsf{T}(X_n, Y_m)$ e verifica-se se $A \wedge U_m$ é insatisfazível. Se for satisfazível então não é possível encontrar o majorante e esta rotina termina sem sucesso.
3. Se $A \wedge U_m$ for insatisfazível calcula-se um novo interpolante $C(Y_m)$ deste par (A, U_m).
4. Se $C(X_n) \rightarrow S$ for tautologia, o invariante pretendido está encontrado.
5. Se $C(X_n) \rightarrow S$ não é tautologia, actualiza-se S com $S \vee C(X_n)$ e repete-se o processo a partir do passo (1).

6 Implementaçao do Problema

Importar o solver 1. Importar itertools 2. Importar pysmt

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

7 Resolver o codigo

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. *nomeTraco* -> O Nome do traço pretendido 3. *i* -> um inteiro que será responsável por dar o nr às variaveis 4. *nBits* -> Nr de Bits para o BitVec 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,nomeTraco,i):
    state = {}
    for v in vars:
        state[v] = Symbol(v+'!'+nomeTraco+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. *a* -> Variável do input do utilizador 3. *b* -> Variável do input do utilizador 4. *nBits* -> Nr de Bits para o BitVec 2. Return de um "And" com a seguinte condição lógica: $(pc = 0 \wedge x = a \wedge y = b \wedge z = 0 \wedge b >= 0)$

[ ]:
```
def init(state, a, b):
    return And(
        Equals(state['pc'], Int(0)),
        Equals(state['x'], Int(a)),
        Equals(state['y'], Int(b)),
        Equals(state['z'], 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):
    # pc=2
    return And(Equals(state['pc'] , Int(2)))
```

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. Inicialmente calculamos o número máximo de bits para conseguirmos fazer a condição

de overflow 2. Criamos as condições lógicas chamadas transita: 1. transita01: $(y = 0 \wedge pc = 0 \wedge pc' = 1 \wedge x' = x \wedge y' = y \wedge z' = z)$ 2. transita02: $(y! = 0 \wedge mod(y) = 0 \wedge x' = 2 * x \wedge y' = y/2 \wedge z' = z \wedge pc' = 3 \wedge pc = 0)$ 3. transita03: $(y! = 0 \wedge mod(y) = 1 \wedge x' = x \wedge y' = y - 1 \wedge z' = z + x \wedge pc' = 3 \wedge pc = 0)$ 4. transita04: $(((pc = 1 \wedge pc' = 1) \vee (pc = 2 \wedge pc' = 2)) \wedge x' = x \wedge y' = y \wedge z' = z)$ 5. transita05: $(pc = 3 \wedge ((Oflow \wedge pc' = 2) \vee (!Oflow \wedge pc' = 3)) \wedge x' = x \wedge y' = y \wedge z' = z)$ 3. Return de um "And" com a seguinte condição lógica: ($ transita01 transita02 transita03 transita04 transita05$)

[ ]:
```
def trans(curr, prox, nBits):
    overflow = GE(curr["x"], Pow(Int(2), Int(nBits-1)))
    odd = Equals(Div(Minus(curr["y"],Int(1)),Int(2)), Div(curr["y"], Int(2)))

    #Parar o ciclo
    transita01 = And(Equals(curr['y'], Int(0)),
                     Equals(curr['pc'], Int(0)),
                     Equals(prox["x"], curr["x"]),
                     Equals(prox["y"], curr["y"]),
                     Equals(prox["z"], curr["z"]),
                     Equals(prox["pc"], Int(1)))

    #even=PAR
    transita02 = And(NotEquals(curr['y'], Int(0)),
                     Not(odd),
                     Equals(prox['x'], Times(curr['x'],Int(2))),
                     Equals(prox["y"],Div(curr['y'], Int(2))),
                     Equals(prox['z'], curr['z']),
                     Equals(prox["pc"], Int(3)),
                     Equals(curr["pc"], Int(0)))

    #odd=Impar
    transita03 = And(NotEquals(curr['y'], Int(0)),
                     odd,
                     Equals(prox['x'], curr['x']),
                     Equals(prox['y'], Minus(curr['y'], Int(1))),
                     Equals(prox['z'], Plus(curr['z'], curr['x'])),
                     Equals(prox["pc"],Int(3)), Equals(curr["pc"], Int(0)))


    # Manter o código parado
    transita04 = And(Or(And(Equals(curr["pc"], Int(1)), Equals(prox["pc"],␣
↪Int(1))),
                        And(Equals(curr["pc"], Int(2)), Equals(prox["pc"],␣
↪Int(2)))),
                        Equals(prox["y"],curr["y"]),
                        Equals(prox["x"],curr["x"]),
                        Equals(prox["z"],curr["z"]))

    # Detetar overflow
```

```
    transita05 = And(Equals(prox["y"],curr["y"]),
                     Equals(prox["x"],curr["x"]),
                     Equals(prox["z"],curr["z"]),
                     Equals(curr["pc"],Int(3)),
                     Or(And(overflow, Equals(prox["pc"], Int(2))),␣
 ↪And(Not(overflow), Equals(prox["pc"], Int(0)))))

    return Or(transita01, transita02, transita03, transita04, transita05)
```

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.

[ ]:
```
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])
```

Função "invert" Esta função recebe a função python que codifica a relação de transição e devolve a relação e transição inversa. 1. Parâmetros: 1. *trans* -> Função transição 3. Return da inversão

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

Função "model_checking" Esta é a função principal e é a que irar juntar as funções todas e gerar o traço pretendido e com ele tabelar o output 1. Parâmetros: 1. *vars* -> Variáveis do algoritmo 2. *genState* -> Função genState 3. *init* -> Função init 4. *trans* -> Função trans 5. *error* -> Função error 6. *nBits* -> Precisão de bits (input do utilizador) 7. *a* -> Variável *a* (input do utilizador) 8. *b* -> Variável *b* (input do utilizador) 2. Função: 1. Iniciamos o Solver 2. Criamos o Traço A e B 3. Estabelecemos a ordem do par (n,m) 4. Resolver o Problema

[ ]:
```
def model_checking(vars, genState, init, trans, error, nBits, a, b, N, M):
    nBits = nBits+1
    with Solver(name="z3") as s:

        # 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], nBits) for i in range(n)])
            I = init(traceA[0], a ,b)
            Rn = And(I, Tn)

            #Criar Bm
            Bm = And([invert(trans, nBits)(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)

            #Passo 3.1
            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:
                    #O Utilizador escolher qual indice incrementar
                    escolha = input("Interpolante não existe, qual deseja␣
↪incrementar:m,n")
                    if escolha=="m":
                        m+=1
                    if escolha == "n":
                        n+=1
                    continue
                #Passo 3.2
                C0 = rename(C, traceA[0])
                C1 = rename(C, traceA[1])
                T = trans(traceA[0], traceA[1], nBits)

                #Passo4
                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], nBits))
                        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
```

Exemplo 1 (Possível) Exemplo simples

[ ]:
```
a = 5
b = 5
k = 15
N = 20
M = 20
vars = ['pc', 'x', 'y','z']
model_checking(vars,genState, init, trans,error, k, a, b, N,M)
```

```
---------------------------------------------------------------------------
NoSolverAvailableError                    Traceback (most recent call last)
Cell In [9], line 7
      5 M = 20
      6 vars = ['pc', 'x', 'y','z']
----> 7 model_checking(vars,genState, init, trans,error, k, a, b, N,M)

Cell In [8], line 34, in model_checking(vars, genState, init, trans, error,␣
 ↪nBits, a, b, N, M)
     32     return
     33 else:
---> 34     C = binary_interpolant(And(Rn, same(traceA[n], traceB[m])), Um)
     35     #Interpolante nao existe
     36     if C is None:
     37         #O Utilizador escolher qual indice incrementar

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
```