TP3

Daniel Francisco Texeira Andrade - A100057

Pedro André Ferreira Malainho - A100050

Problema 3

Enunciado

Considere de novo o 1º problema do TP2 relativo à descrição da cifra *A5/1* e o FOTS usando BitVec's que ai foi definido para a componente do gerador de chaves. Ignore a componente de geração final da chave e restrinja o modelo aos três LFSR's.

Sejam *X0*, *X1*, *X2* as variáveis que determinam os estados dos três LFSR's que ocorrem neste modelo. Como condição inicial e condição de erro use os predicados

$\mathsf{I} \;\equiv\; (\mathsf{X}_0 > 0) \,\land\, (\mathsf{X}_1 > 0) \,\land\, (\mathsf{X}_2 > 0) \quad \mathsf{e} \quad \mathsf{E} \;\equiv\; \neg\,\mathsf{I}$

a. Codifique em "z3" o SFOTS assim definido.

b. Use o algoritmos PDR "property directed reachability" (codifique-o ou use uma versão pré existente) e, com ele, tente provar a segurança deste modelo.

Referências

Github Model-Checking

Implementação

Imports

In [9]:
```
from z3 import BitVec, BitVecVal, And, Or, Not, Extract, Solver, unsat
```

Alinea a)

GenState Function

Generates a state dictionary for the given variables and state label.

`Paramentes:`

- `i:` The index of the state

`Returns:` A dictionary mapping variable names to their corresponding BitVec symbols.

Init1 function

Defines the initial conditions for the state.

`Parameters:`

- `state:` The current state containing variables.

`Returns:` A boolean expression representing the initial conditions.

Error1 function

Defines the error conditions based on the state.

`Parameters:`

- `state:` The current state containing variables.

`Returns:` A boolean expression representing the error conditions.

Trans1 function

Defines the transition relations between the current and next state.

`Parameters:`

- `curr:` The current state.
- `prox:` The next state.

`Returns:` A boolean expression representing the valid transitions.

In [10]:
```
class cifraOne:

    def __init__(self, r0mask=19, r1mask=22, r2mask=23, cbit0=8, cbit1=10, cbit2
        self.R0MASK = r0mask
        self.R1MASK = r1mask
        self.R2MASK = r2mask
        self.cbit0 = cbit0
        self.cbit1 = cbit1
        self.cbit2 = cbit2
        self.S0 = BitVec('S0', self.R0MASK)
        self.S1 = BitVec('S1', self.R1MASK)
        self.S2 = BitVec('S2', self.R2MASK)
```

```python
    def gen_state(self, i):
        return {
            'X0': BitVec(f'X0_{i}', self.R0MASK),
            'X1': BitVec(f'X1_{i}', self.R1MASK),
            'X2': BitVec(f'X2_{i}', self.R2MASK)
        }

    def init1(self, state):
        return And(
            state['X0'] > 0,
            state['X1'] > 0,
            state['X2'] > 0
        )

    def error1(self, state):
        return Not(self.init1(state))

    def trans1(self, curr, nxt):

        lfsr0, lfsr1, lfsr2 = curr['X0'], curr['X1'], curr['X2']
        nlfsr0, nlfsr1, nlfsr2 = nxt['X0'], nxt['X1'], nxt['X2']

        # Extract the control bits for majority function
        c0 = Extract(self.cbit0, self.cbit0, lfsr0) == 1
        c1 = Extract(self.cbit1, self.cbit1, lfsr1) == 1
        c2 = Extract(self.cbit2, self.cbit2, lfsr2) == 1

        # Compute the majority bit
        majority_bit = Or(And(c0, c1), Or(And(c1, c2), And(c0, c2)))

        # Compute the feedback bits and transitions
        feedback0 = lfsr0 & BitVecVal(self.S0, self.R0MASK)
        t0 = And(c0 == majority_bit,
                 nlfsr0 == ((lfsr0 << 1) + feedback0))

        feedback1 = lfsr1 & BitVecVal(self.S1, self.R1MASK)
        t1 = And(c1 == majority_bit,
                 nlfsr1 == ((lfsr1 << 1) + feedback1))

        feedback2 = lfsr2 & BitVecVal(self.S2, self.R2MASK)
        t2 = And(c2 == majority_bit,
                 nlfsr2 == ((lfsr2 << 1) + feedback2))

        # Combine the transitions based on majority function
        return Or(And(t0, t1), Or(And(t0, t2), And(t1, t2)))
```

Alinea b) -> 1ª Implementação

PDR Function

The main Property Directed Reachability algorithm implementation.

`Returns:`

- `SAFE` if no error state can be reached.
- `UNSAFE` if an error state is reachable.

GetBadCube Function

Attempts to find a "bad cube".

`Returns:`

- `None` if no bad cube exists.
- A bad cube (set of symbolic constraints) otherwise.

Function

Attempts to block a bad cube from propagating backward through the transition system.

`Returns:`

- `True` if the bad cube is sucessfuly blocked.
- `False` otherwise.

ExpandToCube Function

Converts a model into a "cube", a set of constraints representing the state.

`Returns:`

- A symbolic conjunction (cube) representing the model.

Primed Function

Generates the "primed" version of a cube, representing the variables in the next state.

`Returns:`

- The primed version of the input cube.

In [11]:

```
cy = cifraOne()
decl_state = lambda i: cy.gen_state(i)
ini_state = cy.init1
error_state = cy.error1
transition = cy.trans1

def PDR():

    F = [Not(ini_state(decl_state(0)))]
    k = 0

    while True:

        bad = get_bad_cube(F[k], error_state(decl_state(k)))

        if bad is None:
```

```
            if F[k] == F[k - 1]:
                return "SAFE"
            else:

                F.append(False)
                k += 1
        else:

            if not bloqueio(F, bad, k):
                return "UNSAFE"


def get_bad_cube(frame, error_condition):

    solver = Solver()
    solver.add(Not(frame))
    solver.add(error_condition)
    if solver.check() == unsat:
        return None
    return expand_to_cube(solver.model())


def bloqueio(F, bad, k):

    cube = bad
    while k > 0:
        solver = Solver()

        curr_state = decl_state(k)
        next_state = decl_state(k + 1)
        solver.add(Not(F[k]))
        solver.add(transition(curr_state, next_state))
        solver.add(Not(cube))
        solver.add(primed(cube))
        if solver.check() == unsat:

            F[k] = And(F[k], Not(cube))
            return True
        else:
            cube = expand_to_cube(solver.model())
            k -= 1
    return False


def expand_to_cube(model):

    cube = []
    for d in model:
        value = model[d]

        if value.as_long() == 1:
            cube.append(d == 1)
        else:
            cube.append(d == 0)
    return And(cube)

def primed(cube):

    primed_cube = []
```

```python
    for var in cube:
        primed_var = f"{var}'"
        primed_cube.append(primed_var)
    return And(primed_cube)


def main():
    result = PDR()
    print(f"System is: {result}")


if __name__ == "__main__":
    main()
```

```
System is: SAFE
```

Alinea b) -> 2ª Implementação (Extensa)

NextVar Function

`Returns:` The 'next' of the given variable

AtTime function

Builds an SMT variable representing v at time t

TransitionSystem Class

Trivial representation of a Transition System.

PDR Class

`Functions:`

- `__init__`
- `Check_Property:`
  Property Directed Reachability approach without optimizations.
- `GetBadState:`
  Extracts a reachable state that intersects the negation of the property and the last current frame
- `Solve:`
  Provides a satisfiable assignment to the state variables that are consistent with the input formula
- `RecursiveBlock:`
  Blocks the cube at each frame, if possible. Returns True if the cube cannot be blocked.

- `Inductive:`
  Checks if last two frames are equivalent

BMCInduction Class

`Functions:`

- `__init__`
- `GetSubs:`
  Builds a map from x to x@i and from x' to x@(i+1), for all x in system.
- `GetUnrolling:`
  Unrolling of the transition relation from 0 to k.
- `GetSimplePath:`
  Simple path constraint for k-induction: each time encodes a different state
- `GetKHypothesis:`
  Hypothesis for k-induction: each state up to k-1 fulfills the property
- `GetBMC:`
  Returns the BMC encoding at step k
- `GetKInduction:`
  Returns the K-Induction encoding at step K
- `CheckProperty:`
  Interleaves BMC and K-Ind to verify the property.

In [12]:

```
from pysmt.shortcuts import Symbol, Not, And, Or, EqualsOrIff
from pysmt.shortcuts import is_sat, is_unsat, Solver, TRUE

def next_var(v):
    return Symbol("next(%s)" % v.symbol_name(), v.symbol_type())

def at_time(v, t):
    return Symbol("%s@%d" % (v.symbol_name(), t), v.symbol_type())

class TransitionSystem(object):
    def __init__(self, variables, init, trans):
        self.variables = variables
        self.init = init
        self.trans = trans

class PDR(object):
    def __init__(self, system):
        self.system = system
        self.frames = [system.init]
        self.solver = Solver()
        self.prime_map = dict([(v, next_var(v)) for v in self.system.variables])

    def check_property(self, prop):
        print("Checking property %s..." % prop)

        while True:
            cube = self.get_bad_state(prop)
            if cube is not None:
                # Blocking phase of a bad state
                if self.recursive_block(cube):
```

```
                    print("--> Bug found at step %d" % (len(self.frames)))
                    break
                else:
                    print("   [PDR] Cube blocked '%s'" % str(cube))
            else:
                # Checking if the last two frames are equivalent i.e., are induc
                if self.inductive():
                    print("--> The system is safe!")
                    break
                else:
                    print("   [PDR] Adding frame %d..." % (len(self.frames)))
                    self.frames.append(TRUE())

    def get_bad_state(self, prop):
        return self.solve(And(self.frames[-1], Not(prop)))

    def solve(self, formula):
        if self.solver.solve([formula]):
            return And([EqualsOrIff(v, self.solver.get_value(v)) for v in self.s
        return None

    def recursive_block(self, cube):
        for i in range(len(self.frames)-1, 0, -1):
            cubeprime = cube.substitute(dict([(v, next_var(v)) for v in self.sys
            cubepre = self.solve(And(self.frames[i-1], self.system.trans, Not(cu
            if cubepre is None:
                for j in range(1, i+1):
                    self.frames[j] = And(self.frames[j], Not(cube))
                return False
            cube = cubepre
        return True

    def inductive(self):
        if len(self.frames) > 1 and \
           self.solve(Not(EqualsOrIff(self.frames[-1], self.frames[-2]))) is Non
            return True
        return False

    def __del__(self):
        self.solver.exit()

class BMCInduction(object):

    def __init__(self, system):
        self.system = system

    def get_subs(self, i):
        subs_i = {}
        for v in self.system.variables:
            subs_i[v] = at_time(v, i)
            subs_i[next_var(v)] = at_time(v, i+1)
        return subs_i

    def get_unrolling(self, k):
        res = []
        for i in range(k+1):
            subs_i = self.get_subs(i)
            res.append(self.system.trans.substitute(subs_i))
        return And(res)
```

```
    def get_simple_path(self, k):
        res = []
        for i in range(k+1):
            subs_i = self.get_subs(i)
            for j in range(i+1, k+1):
                state = []
                subs_j = self.get_subs(j)
                for v in self.system.variables:
                    v_i = v.substitute(subs_i)
                    v_j = v.substitute(subs_j)
                    state.append(Not(EqualsOrIff(v_i, v_j)))
                res.append(Or(state))
        return And(res)

    def get_k_hypothesis(self, prop, k):
        res = []
        for i in range(k):
            subs_i = self.get_subs(i)
            res.append(prop.substitute(subs_i))
        return And(res)

    def get_bmc(self, prop, k):
        init_0 = self.system.init.substitute(self.get_subs(0))
        prop_k = prop.substitute(self.get_subs(k))
        return And(self.get_unrolling(k), init_0, Not(prop_k))

    def get_k_induction(self, prop, k):
        subs_k = self.get_subs(k)
        prop_k = prop.substitute(subs_k)
        return And(self.get_unrolling(k),
                   self.get_k_hypothesis(prop, k),
                   self.get_simple_path(k),
                   Not(prop_k))

    def check_property(self, prop):
        print("Checking property %s..." % prop)
        for b in range(100):
            f = self.get_bmc(prop, b)
            print("   [BMC]    Checking bound %d..." % (b+1))
            if is_sat(f):
                print("--> Bug found at step %d" % (b+1))
                return

            f = self.get_k_induction(prop, b)
            print("   [K-IND]  Checking bound %d..." % (b+1))
            if is_unsat(f):
                print("--> The system is safe!")
                return


def cifraTwo():

    from pysmt.shortcuts import Equals, BVAnd, BVOr, BVExtract, BVAdd, BV
    from pysmt.typing import BVType

    R0MASK, R1MASK, R2MASK = 19, 22, 23
    cbit0, cbit1, cbit2 = 8, 10, 10

    lfsr0 = Symbol("lfsr0", BVType(R0MASK))
    lfsr1 = Symbol("lfsr1", BVType(R1MASK))
```

```python
    lfsr2 = Symbol("lfsr2", BVType(R2MASK))

    nlfsr0 = next_var(lfsr0)
    nlfsr1 = next_var(lfsr1)
    nlfsr2 = next_var(lfsr2)

    variables = [lfsr0, lfsr1, lfsr2]

    c0 = BVExtract(lfsr0, cbit0, cbit0)
    c1 = BVExtract(lfsr1, cbit1, cbit1)
    c2 = BVExtract(lfsr2, cbit2, cbit2)
    majority_bit = BVOr(BVAnd(c0, c1), BVOr(BVAnd(c1, c2), BVAnd(c0, c2)))

    feedback0 = BVAnd(lfsr0, BV(1, R0MASK))
    t0 = And(Equals(c0, majority_bit),
             Equals(nlfsr0, BVAdd(lfsr0 << 1, feedback0)))

    feedback1 = BVAnd(lfsr1, BV(1, R1MASK))
    t1 = And(Equals(c1, majority_bit),
             Equals(nlfsr1, BVAdd(lfsr1 << 1, feedback1)))

    feedback2 = BVAnd(lfsr2, BV(1, R2MASK))
    t2 = And(Equals(c2, majority_bit),
             Equals(nlfsr2, BVAdd(lfsr2 << 1, feedback2)))

    trans1 = Or(And(t0, t1), Or(And(t0, t2), And(t1, t2)))

    # Initial conditions: All LFSRs are greater than zero
    init = And(lfsr0 > BV(0, R0MASK),
               lfsr1 > BV(0, R1MASK),
               lfsr2 > BV(0, R2MASK))

    # True invariant: All LFSRs are not zero
    true_prop = And(Not(Equals(lfsr0, BV(0, R0MASK))),
                    Not(Equals(lfsr1, BV(0, R1MASK))),
                    Not(Equals(lfsr2, BV(0, R2MASK))))

    # False invariant: All LFSRs are zero or overflow
    false_prop = Not(true_prop)

    return (
        TransitionSystem(variables, init, trans1),
        [true_prop, false_prop]
    )


def main():
    example = cifraTwo()

    bmcind = BMCInduction(example[0])
    pdr = PDR(example[0])

    for prop in example[1]:
        bmcind.check_property(prop)
        pdr.check_property(prop)
        print("")

if __name__ == "__main__":
    main()
```

```
Checking property ((! (lfsr0 = 0_19)) & (! (lfsr1 = 0_22)) & (! (lfsr2 = 0_2
3)))...
   [BMC]    Checking bound 1...
   [K-IND]  Checking bound 1...
   [BMC]    Checking bound 2...
--> Bug found at step 2
Checking property ((! (lfsr0 = 0_19)) & (! (lfsr1 = 0_22)) & (! (lfsr2 = 0_2
3)))...
   [PDR] Adding frame 1...
   [PDR] Cube blocked '((lfsr0 = 2_19) & (lfsr1 = 2_22) & (lfsr2 = 0_23))'
--> Bug found at step 2

Checking property (! ((! (lfsr0 = 0_19)) & (! (lfsr1 = 0_22)) & (! (lfsr2 = 0_2
3))))...
   [BMC]    Checking bound 1...
--> Bug found at step 1
Checking property (! ((! (lfsr0 = 0_19)) & (! (lfsr1 = 0_22)) & (! (lfsr2 = 0_2
3))))...
   [PDR] Cube blocked '((lfsr0 = 3_19) & (lfsr1 = 1_22) & (lfsr2 = 2_23))'
   [PDR] Cube blocked '((lfsr0 = 2_19) & (lfsr1 = 2_22) & (lfsr2 = 3_23))'
--> Bug found at step 2
```