

I. Algorithme de Tarjan et composantes fortement connexes :

Pour trouver les attracteurs, la bibliothèque PyBoolNet utilise la fonction **compute_attractors_tarjan**, cette fonction fait appelle à la fonction **strongly_connected_components** de la bibliothèque networkx en passant par la fonction **digraph2condensationgraph**.

```
def compute_attractors_tarjan(s):
    """
    Parameters:
    s: str, state transition graph
    Returns:
    steadystates: list of sets of str, the steady states
    cyclic: list of sets of str, the cyclic attractors
    """
    condensation_graph = PyBoolNet.utility.Digraph2CondensationGraph(s)
    steadystates = []
    cyclic = []

    for scc in condensation_graph.nodes():
        if not list(condensation_graph.successors(scc)):
            steadystates.append(scc)
        else:
            cyclic.append(scc)

    return steadystates, cyclic
```

Retourne les différents attracteurs du réseau de booléens.
La variable **cyclic** représente les attracteurs cycliques et la variable **steadystates** représente les attracteurs simples (non-cycliques).

```
def digraph2condensationgraph(digraph):
    """
    Creates the condensation graph from a digraph.
    The condensation graph is similar to the scc graph but it replaces cascades between
    SCCs by single edges.
    """
    parameters:
    digraph: networkx.Digraph, directed graph
    Returns:
    condensation_graph: networkx.Digraph, the condensation graph
    """
    sccs = sorted([tuple(sorted(scc)) for scc in networkx.strongly_connected_components(digraph)])
    cascades = []
    for c in sccs:
        for e in sccs:
            if (len(c) > 1 and len(e) > 1) and not digraph.has_edge(c[0], e[0]):
                cascades.append((c, e))

    graph = networkx.Digraph()
    graph.add_nodes_from(sccs)
    graph.add_edges_from(cascades)

    # graph is a copy of digraph with edges leading noncyclic components removed
    # will use graph to decide if there is a cascade path between u and v (i.e. edge in graph)
    graph = networkx.Digraph(digraph.edges())

    for u, v in itertools.product(sccs, sccs):
        if u == v:
            continue
        graph = Digraph.copy()
        for k in noncascades:
            if not k.startswith(u) and not k.endswith(v):
                graph.remove_node(k)

        if has_path(graph, u, v):
            graph.add_edge(u, v)

    # annotate each node with its depth in the hierarchy and an integer ID
    for id, target in enumerate(sccs):
        depth = 1
        for source in networkx.ancestors(target):
            for g in networkx.all_hybrid_paths(graph, source, target):
                depth = max(depth, len(g))
            graph.node[target]['depth'] = depth
            graph.node[target]['id'] = id

    return graph
```

Utilisation de Tarjan

```
def strongly_connected_components(G):
    """Generate nodes in strongly connected components of graph.

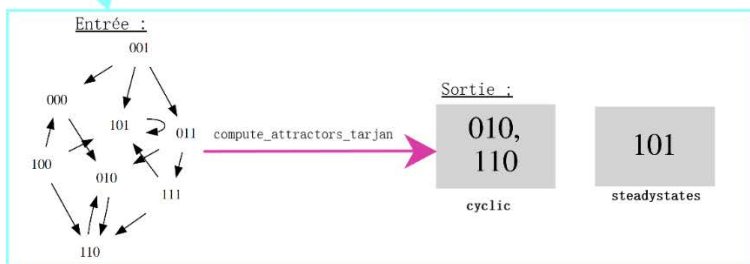
    Parameters:
    G: NetworkX Graph
    A directed graph.

    Returns:
    comp: generator of sets
    A generator of sets of nodes, one for each strongly connected component of G.

    Raises:
    NetworkXNotImplemented
    If G is undirected.

    Notes:
    Uses Tarjan's algorithm[1], with Nuutila's modifications[2].
    Nonrecursive version of algorithm.
    """
    preorder = []
    lowlink = {}
    scc_found = set()
    scc_queue = []
    i = 0 # Preorder counter

    for source in G:
        if source not in scc_found:
            queue = [source]
            while queue:
                v = queue[-1]
                if v not in preorder:
                    i = i + 1
                    preorder[v] = i
                    done = True
                    for w in G[v]:
                        if w not in preorder:
                            queue.append(w)
                            done = False
                            break
                    if done:
                        lowlink[v] = preorder[v]
                        for w in G[v]:
                            if w not in scc_found:
                                if preorder[w] > preorder[v]:
                                    lowlink[v] = min(lowlink[v], lowlink[w])
                                else:
                                    lowlink[v] = min(lowlink[v], preorder[w])
                        queue.pop()
                        if lowlink[v] == preorder[v]:
                            scc = set()
                            while scc_queue and preorder[scc_queue[-1]] > preorder[v]:
                                k = scc_queue.pop()
                                scc.add(k)
                            scc_found.update(scc)
                            yield scc
                        else:
                            scc_queue.append(v)
```



Ainsi, à la fin de l'exécution, on récupère les différents attracteurs du réseau : les **steadystates** et les **cyclic**. Les premiers représentent les attracteurs « simples » tandis que les seconds sont les attracteurs cycliques.

Pour obtenir les composantes fortement connexes, on utilise une version non-récursive de l'algorithme de Nuutila, qui est une amélioration de l'algorithme de Tarjan.

Algorithme de Tarjan :

```
fonction tarjan(graphe G)
    num := 0
    P := pile vide
    partition := ensemble vide

    fonction parcours(sommet v)
        v.num := num
        v.numAccessible := num
        num := num + 1
        P.push(v), v.dansP := oui

        // Parcours récursif
        pour chaque w successeur de v
            si w.num n'est pas défini
                parcours(w)
            v.numAccessible := min(v.numAccessible, w.numAccessible)
            sinon si w.dansP = oui
                v.numAccessible := min(v.numAccessible, w.num)

        si v.numAccessible = v.num
            // v est une racine, on calcule la composante fortement connexe associée
            C := ensemble vide
            répéter
                w := P.pop(), w.dansP := non
                ajouter w à C
            tant que w différent de v
                ajouter C à partition
        fin de fonction

    pour chaque sommet v de G
        si v.num n'est pas défini
            parcours(v)

    renvoyer partition
fin de fonction
```

Algorithme de Nuutila :

```
1. procedure COMP-TC(v);
2. begin
3.   root(v) := v; C(v) := Nil;
4.   PUSH(v, nstack);
5.   hsaved(v) := HEIGHT(cstack);
6.   for each vertex w such that (v, w) ∈ E do begin
7.     if w is not already visited then COMP-TC(w);
8.     if C(w) = Nil then root(v) := MIN(root(v), root(w))
9.     else if (v, w) is not a forward edge then
10.      PUSH(C(w), cstack);
11.   end;
12.   if root(v) = v then begin
13.     create a new component C;
14.     if TOP(nstack) = v then Succ(C) := ∅
15.     else Succ(C) := {C};
16.     while HEIGHT(cstack) ≠ hsaved(v) do begin
17.       X := POP(cstack);
18.       if X ∉ Succ(C) then Succ(C) := Succ(C) ∪ {X} ∪ Succ(X);
19.     end;
20.     repeat
21.       w := POP(nstack);
22.       C(w) := C;
23.       insert w into component C;
24.     until w = v
25.   end
26. end;
27. begin /* Main program */
28.   nstack := ∅; cstack := ∅;
29.   for each vertex v ∈ V do
30.     if v is not already visited then COMP-TC(v)
31. end.
```

Différence entre l'algorithme de Tarjan et celui de Nuutila :

La principale différence entre l'algorithme de Tarjan et celui modifié par Nuutila réside dans **l'amélioration de la fermeture transitive** (*transitive closure* en anglais).

La détection des composantes fortement connexes utilise l'algorithme de recherche en profondeur (DFS). Cependant, au lieu de construire partiellement les ensembles de successeurs des composantes puis de les combiner ou de les construire après avoir détecté complètement la composante, ces derniers sont créés pendant la recherche. Cela **évite des opérations** inutiles et notamment de scanner le graphe plusieurs fois.

L'algorithme de fermeture transitive utilisé dans cette modification génère un unique ensemble de successeur pour chaque composante pendant la recherche en profondeur.

Pour ce faire l'algorithme de Nuutila utilise une **pile auxiliaire** qui lui permet de collecter les composantes adjacentes qui seront nécessaires plus tard pour la construction de l'ensemble. Ainsi le graphe n'est **scanné qu'une seule fois**.

La trame principale de la détection des composantes fortement connexes est cependant la même que celle de Tarjan.

La pile auxiliaire (citée précédemment) utilisée dans l'algorithme est nommé **cstack**. Outre le fait d'éviter de scanner 2 fois les arrêtes partants des sommets de la composante parcourue, elle permet aussi de **stocker les composantes de manière consécutive** dans la pile et ainsi en cas de problème de mémoire, le parcours de la pile nécessite moins d'opérations de pagination que l'accès à la liste d'adjacence contenant les sommets des composants. (Cf. l'article original en pièce jointe).

Une fois qu'on a obtenu le graphe des composantes fortement connexes grâce à l'algorithme de Nuutila et qu'on l'a transformé en **condensation graph** (à l'aide de la méthode de networkx), on regarde **chaque nœud** du graphe, s'il n'a **pas de successeur** on en déduit que c'est un attracteur. On regarde alors s'il est **seul** ou s'il est **composé de plusieurs nœuds**, afin de le classer (**cyclic** ou **steadystates**).

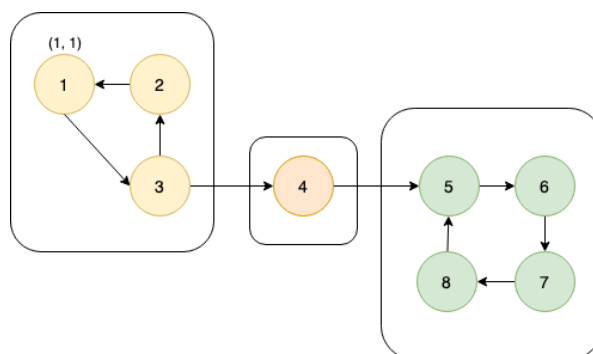
Implémentation de l'algorithme de Nuutila dans networkx :

Tout d'abord, il est important de préciser que les **deux algorithmes** (version récursive et non-récursive) ont été implémentés dans networkx. Cependant, nous **n'étudierons que la version non-récursive** car c'est celle choisie par les développeurs de PyBoolNet.

4 tableaux sont utilisés dans cet algorithme :

- **preorder** : il s'agit d'un **dictionnaire qui stocke les numéros des nœuds** en clé et leur position (avec l'algorithme de parcours en profondeur : DFS) en valeur
- **lowlink** : il s'agit d'un **dictionnaire qui stocke en clé le numéro du nœud dont les successeurs sont déjà dans le preorder**. Sa valeur initiale sera la position actuelle donnée par le preorder. Puis sa valeur sera mise à jour afin de garder en mémoire la composante fortement connexe dans laquelle il se situe. (Commence par 1)
- **scc_queue** : il s'agit d'une **pile qui stocke les éléments ne formant pas encore une composante** fortement connexe (la source n'est pas encore trouvée). A la fin de l'algorithme cette pile est vide.
- **scc_found** : il s'agit d'une **pile qui stocke tous les éléments faisant partie d'une composante** fortement connexe. Elle est mise à jour en piochant dans les éléments de scc_queue lorsque le nœud source est trouvé.

Ci-dessous, un exemple du fonctionnement de l'algorithme sur ce graphe :





Vidéo exemple de l'algorithme de Nuutila non récursif.

II. Utilisation de NuSMV dans PyBoolNet :

Les appels vers NuSMV se font dans la classe **ModelChecking** de PyboolNet. En effet, on peut trouver des références à NuSMV dans la classe Commitment mais il ne s'agit que d'appels vers des méthodes de la classe ModelChecking (counter_mc compte les appels/exécutions faites vers ses méthodes).

Prenons l'exemple de la méthode **check_primes_with_acceptingstates** citée dans l'article du créateur de PyBoolNet car c'est celle-ci qui nous intéresse pour la suite de nos recherches.

On y apprend que NuSMV est exécuté via un sous-processus (**subprocess**) dont la ligne de commande est générée en fonction des paramètres de la méthode.

On peut également voir qu'un **fichier temporaire** (accessible via /tmp sous UNIX) **est créé**. D'abord vide, il sera ensuite rempli à l'aide de la méthode **primes2smv** dont le but est de convertir le réseau booléen (transformé sous forme de primes) en format SMV.

Ensuite, le **processus exécute la ligne de commande** sur le fichier créé.

[Attention : Le fichier SMV n'est pas directement récupérable car il est supprimé après avoir été analysé par NuSMV et il n'y a pas de paramètres pour empêcher sa suppression.]

```
def check_primes_with_acceptingstates(Primes, Update, InitialStates, CTLSpec, DynamicReorder=True, ConeOfInfluence=True, Silent=True):
    """
    The accepting states are a dictionary with the following keywords:
    * 'INIT': a Boolean expression for the initial states, or 'None', see note below
    * 'INIT_SIZE': integer number of initial states, or 'None', see note below
    * 'ACCEPTING': a Boolean expression for the accepting states
    * 'ACCEPTING_SIZE': integer number of accepting states
    * 'INITACCEPTING': a Boolean expression for the intersection of initial and accepting states, or 'None', see note below
    * 'INITACCEPTING_SIZE': integer number of states in the intersection of initial and accepting states, or 'None', see note below

    **arguments**
    * 'Primes': prime implicants
    * 'Update' (str): the update strategy, either "synchronous", "asynchronous" or "mixed"
    * 'InitialStates' (str): a ref: installation nusmv expression for the initial states, including the keyword 'INIT'
    * 'CTLSpec' (str): a ref: installation nusmv formula, including the keyword 'CTLSPEC'
    * 'DynamicReorder' (bool): enables dynamic reordering of variables (*-dynamic*)
    * 'ConeOfInfluence' (bool): enables cone of influence reduction using *-coi*
    * 'Silent' (bool): print infos to screen

    **returns**
    * 'Answer, AcceptingStates' (bool, dict): result of query with accepting states
    """
    assert(CTLSpec[-7] == "CTLSPEC")
    print_warning_accstates_bug(Primes, CTLSpec)
    cmd = ["OD_NUSMV"]
    cmd += ['-dcx']
    cmd += ['-a', 'print']

    if DynamicReorder:
        cmd += ['-dynamic']
    if ConeOfInfluence:
        cmd += ['-coi']

    # enforced to ensure accepting states are correct
    cmd += ['-df']

    tmpfile = tempfile.NamedTemporaryFile(delete=False, prefix="pyboolnet_")
    tmpfname = tmpfile.name
    if not Silent:
        print("created %s" % tmpfname)
    tmpfile.close()
    smvfile = primes2smv(Primes, Update, InitialStates, CTLSpec, FnameSMV=tmpfname, Silent=True)
    cmd += [tmpfname]

    try:
        proc = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    except Exception:
        print("could not start process for nusmv")
        print("cmd: %s" % ' '.join(cmd))
        raise Exception

    out, err = proc.communicate()
    out = out.decode()

    if os.path.isfile(tmpfname):
        os.remove(tmpfname)

    return nusmv_handle(cmd, proc, out, err, DisableCounterExamples=True, AcceptingStates=True)
```

Génération de la ligne de commande du **subprocess**

Création du **fichier temporaire**

Remplissage du fichier temporaire avec le **.smv**

Exécution de la ligne de commande et du **subprocess**

Suppression du fichier temporaire

Interprétation de la réponse de NuSMV

Pour finir, la méthode `nusmv_handle` est appelée lors du retour de la méthode afin d'interpréter la réponse NuSMV.

```
def nusmv_handle(Command, Process, Output, Error, DisableCounterExamples, AcceptingStates):
    """
    **arguments**:
    * *Command* (list): list of commands that was used to call subprocess.Popen.
      It is only used for creating an error message if the process does not finish correctly.
    * *Process* (subprocess.Popen): the object returned by subprocess.Popen
    * *Output* (Popen.communicate): the object returned by Popen.communicate
    * *DisableCounterExamples* (bool): whether a counterexample should be returned if the query is false
    * *AcceptingStates* (bool): whether information about the accepting states should be returned

    **returns**:
    * *Answer* (bool): whether NuSMV returns "is true"
    * *CounterExample* (list): a counterexample if NuSMV returns "is false" and DisableCounterExamples==False.
    * *AcceptingStates* (dict): information about the accepting states, if *DisableAcceptingStates==False*.
    """

    if Process.returncode == 0:
        if Output.count('specification')>1:
            print('SMV file contains more than one CTL or LTL specification.')
            raise Exception

        if 'is false' in Output:
            answer = False
        elif 'is true' in Output:
            answer = True
        else:
            print(Output)
            print(Error)
            print('NuSMV output does not respond with "is false" or "is true".')
            raise Exception

    else:
        print(Output)
        print(Error)
        print('NuSMV did not respond with return code 0')
        print('command: %s' % ' '.join(Command))
        raise Exception

    if DisableCounterExamples and not AcceptingStates:
        return answer

    result = [answer]

    if not DisableCounterExamples:
        counterex = output2counterexample(NuSMVOutput=Output)
        result.append(counterex)

    if AcceptingStates:
        accepting = output2acceptingstates(NuSMVOutput=Output)
        result.append(accepting)

    return tuple(result)
```

"Parser" de
réponse nuSMV

La méthode `output2acceptingstates` permet de « parser » la réponse de NuSMV et de renvoyer le résultat sous la forme voulue par les développeurs de PyBoolNet.

Il est important de souligner qu'il existe aussi une méthode `check_smv_with_acceptingstates` qui permet directement de travailler sur un fichier SMV fourni en paramètre de la fonction et qui se comportera comme la méthode `check_primes_with_acceptingstates`.

Limite de PyBoolNet :

How well does PyBoolNet scale with large networks?

"Depends on what you want to do with it and what "large" is. You can compute steady states for networks with 600 components easily, unless the update functions are particularly evil. You can do model checking for 50 components easily. You can draw interaction graphs or state transition graphs for 3000 nodes easily."

Hklarner (<https://github.com/hklarner/PyBoolNet/issues/9>)

III. Etude de nuSMV-a :

Nous avons testé nuSMV-a sur l'exemple 4.2 (Sémaphore) disponible dans le tutoriel de nuSMV.

```

MODULE main

VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);

ASSIGN
  init(semaphore) := FALSE;

SPEC AG ! (proc1.state = critical & proc2.state = critical)
SPEC AG (proc1.state = entering -> AF proc1.state = critical)

MODULE user(semaphore)
VAR
  state : {idle, entering, critical, exiting};

ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle, entering};
      state = entering & !semaphore : critical;
      state = critical : {critical, exiting};
      state = exiting : idle;
      TRUE : state;
    esac;
  next(semaphore) :=
    case
      state = entering : TRUE;
      state = exiting : FALSE;
      TRUE : semaphore;
    esac;

FAIRNESS
  running

```

Or, contrairement à nuSMV, nuSMV-a ne peut **pas prendre en charge plusieurs SPEC** en un seul fichier.

```

>>> print(PyBoolNet.ModelChecking.check_smv_with_acceptingstates("semaphore.smv"
))
SMV file contains more than one CTL or LTL specification.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "/usr/local/lib/python3.6/dist-packages/PyBoolNet/ModelChecking.py", line
385, in check_smv_with_acceptingstates
    return nusmv_handle(cmd, proc, out, err, DisableCounterExamples=True, Accept
ingStates=True)
    File "/usr/local/lib/python3.6/dist-packages/PyBoolNet/ModelChecking.py", line
722, in nusmv_handle
    raise Exception
Exception

```

Nous avons donc **divisé** cet exemple en deux fichiers différents. La **première assertion est censée être vraie**, la **seconde est censée être fausse**.

```

MODULE main

VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);

ASSIGN
  init(semaphore) := FALSE;

SPEC AG ! (proc1.state = critical & proc2.state = critical)

MODULE user(semaphore)
VAR
  state : {idle, entering, critical, exiting};

ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle, entering};
      state = entering & !semaphore : critical;
      state = critical : {critical, exiting};
      state = exiting : idle;
      TRUE : state;
    esac;
  next(semaphore) :=
    case
      state = entering : TRUE;
      state = exiting : FALSE;
      TRUE : semaphore;
    esac;

FAIRNESS
  running

```

```

MODULE main

VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);

ASSIGN
  init(semaphore) := FALSE;

SPEC AG (proc1.state = entering -> AF proc1.state = critical)

MODULE user(semaphore)
VAR
  state : {idle, entering, critical, exiting};

ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle, entering};
      state = entering & !semaphore : critical;
      state = critical : {critical, exiting};
      state = exiting : idle;
      TRUE : state;
    esac;
  next(semaphore) :=
    case
      state = entering : TRUE;
      state = exiting : FALSE;
      TRUE : semaphore;
    esac;

FAIRNESS
  running

```


Dans un premier temps, nous avons testé les deux CTL avec la méthode `check_smv_with_acceptingstates`.

On obtient alors,

- Pour la première CTL :

```
Test sur le fichier : semaphoreCTL1.smv

(True, {'INIT': '!(semaphore | (proc1.state.1 | (proc1.state.0 | (proc2.state.1 | (proc2.state.0))))', 'INIT_SIZE': 1, 'ACCEPTING': '!(semaphore & (proc1.state.0 & (proc2.state.0)) | !semaphore & (proc1.state.0 | (proc2.state.0)))', 'ACCEPTING_SIZE': 16, 'INITACCEPTING': '!(semaphore | (proc1.state.1 | (proc1.state.0 | (proc2.state.1 | (proc2.state.0))))', 'INITACCEPTING_SIZE': 1})
```

- Pour la seconde CTL :

```
Test sur le fichier : semaphoreCTL2.smv

(False, {'INIT': '!(semaphore | (proc1.state.1 | (proc1.state.0 | (proc2.state.1 | (proc2.state.0))))', 'INIT_SIZE': 1, 'ACCEPTING': 'FALSE', 'ACCEPTING_SIZE': 0, 'INITACCEPTING': 'FALSE', 'INITACCEPTING_SIZE': 0})
```

Puis, nous avons écrit un **programme python pour exécuter nuSMV-a** sur ces deux exemples. Nous avons utilisé `check_svm_with_acceptingstates` sur le premier fichier puisque la CTL est vraie, et qu'ainsi il y a des *acceptingstates*. Et utiliser la fonction `check_svm_with_counterexample` pour la CTL fausse, puisqu'elle devrait avoir des contre-exemples. Cette méthode est censée nous renvoyer un contre-exemple pour **prouver que la CTL est fausse**.

```
import PyBoolNet.ModelChecking as pm

def check_smv(FnameSMV):

    print("Test sur le fichier :", FnameSMV, "\n")
    if (pm.check_smv(FnameSMV)):
        return pm.check_smv_with_acceptingstates(FnameSMV)

    return pm.check_smv_with_counterexample(FnameSMV)

print(check_smv("semaphoreCTL1.smv"))
print(check_smv("semaphoreCTL2.smv"))
```

Pourtant, la fonction `check_svm_with_counterexample` renvoie une erreur, puisqu'apparemment une partie de la réponse de nuSMV est `None`, ce qui n'est pas géré par le *parser* de PyBoolNet.

```
Test sur le fichier : semaphoreCTL1.smv

(True, {'INIT': '!(semaphore | (proc1.state.1 | (proc1.state.0 | (proc2.state.1 | (proc2.state.0))))', 'INIT_SIZE': 1, 'ACCEPTING': '!(semaphore & (proc1.state.0 & (proc2.state.0)) | !semaphore & (proc1.state.0 | (proc2.state.0)))', 'ACCEPTING_SIZE': 16, 'INITACCEPTING': '!(semaphore | (proc1.state.1 | (proc1.state.0 | (proc2.state.1 | (proc2.state.0))))', 'INITACCEPTING_SIZE': 1})
Test sur le fichier : semaphoreCTL2.smv

Traceback (most recent call last):
  File "off_test_lol.py", line 12, in <module>
    print(check_smv("semaphoreCTL2.smv"))
  File "off_test_lol.py", line 9, in check_smv
    return pm.check_smv_with_counterexample(FnameSMV)
  File "/usr/local/lib/python3.6/dist-packages/PyBoolNet/ModelChecking.py", line 336, in check_smv_with_counterexample
    return nusmv_handle(cmd, proc, out, err, DisableCounterExamples=False, AcceptingStates=False)
  File "/usr/local/lib/python3.6/dist-packages/PyBoolNet/ModelChecking.py", line 747, in nusmv_handle
    counterex = output2counterexample(NuSMVOutput=Output)
  File "/usr/local/lib/python3.6/dist-packages/PyBoolNet/ModelChecking.py", line 600, in output2counterexample
    assert(value in ['TRUE','FALSE'])
AssertionError
```