Rapport d' Enseignement d'Intégration : Jeux adversariaux

Pierre-Emmanuel Seyeux, Julien Rosenberger, Gabriel Farago,
Damien Ouzillou, Ronan Villain

28/01/2022

1 Introduction

Dans cet Enseignement d'Intégration, on va s'intéresser à la validation de certaines propriétés d'un logiciel de gestion de capteurs en réseau. Nous ne rentrerons pas dans les détails d'implémentation de ce logiciel puisqu'un autre groupe s'occupe déjà de cette tâche, et on se penchera plutôt dans le détails de l'utilisation de la théorie des jeux pour assurer une couverture de test optimale.

2 Objectifs

Principe général du test, vocabulaire

Le test d'un logiciel peut prendre plusieurs formes : études de l'adéquation des entrées/sorties, performances, respects d'exigences,... L'intérêt est de valider le fonctionnement correct de l'objet à tester, et de vérifier sa conformité vis à vis du but initial recherché.
Le système réel à tester est appelé SUT, pour *System Under Test*, et l'ensemble des méthodes employées pour effectuer le test est appelé *harnais de test*.

On fait la distinction entre deux catégories indépendantes de test : la méthode statique (*offline*) qui se limite à l'étude de traces d'exécution (logs,...), et le test dynamique (*online*), qui a lieu simultanément pendant l'exécution du système.
On distingue aussi le niveau d'information dont on dispose en deux catégories : boîte noire ou boîte blanche, en fonction de notre connaissance des données et états internes du système.

Formalisation

L'implémentation du système de communication pour le réseau de capteur sans fil se fait sous Python. Nous avons décidé d'employer aussi ce langage pour réaliser notre couverture de test, en utilisant plus précisemment le module *unittest*.
On partira d'un graphe IOSM (Input Output State Machine) issu de la spécification pour réaliser une couverture de test de type modèle à automate.

Définition : IOSM

Un automate à entrées et sorties est un quadruplet (S, L, T, s_0) où:

- S est un ensemble fini non vide d'états dont s_0, état initial de l'automate, est un élément
- L est un alphabet fini non vide d'interactions
- $T \subset (S \times ((?, !.L) \cup \tau) \times S$ est l'ensemble des transitions de l'automate, une transition représentant le changement d'état de l'automate d'un état de départ à un état d'arrivé et étant associée à une action observable (émission ! ou réception ?) ou non (transition interne τ).

Exemple

On considère l'automate ci-dessous, qui modélise un distributeur automatique de boissons. L'utilisateur peut introduire une pièce dans la machine (input système "$?Coin$"). Au niveau d'abstraction de la modélisation, la réponse de la machine peut être soit "$!Bad$" ou "$!Good$". Dans le premier cas, la machine revient à son état initial; dans le deuxième de nouvelles actions sont disponible pour l'utilisateur et la machine attends qu'il entre soit "$?Soda$" soit "$?Cancel$". Dans le premier cas la machine peut soit valider la demande de boisson et revenir à son état initial, soit informer du manque de disponibilité et demander à l'utilisateur de refaire son choix.

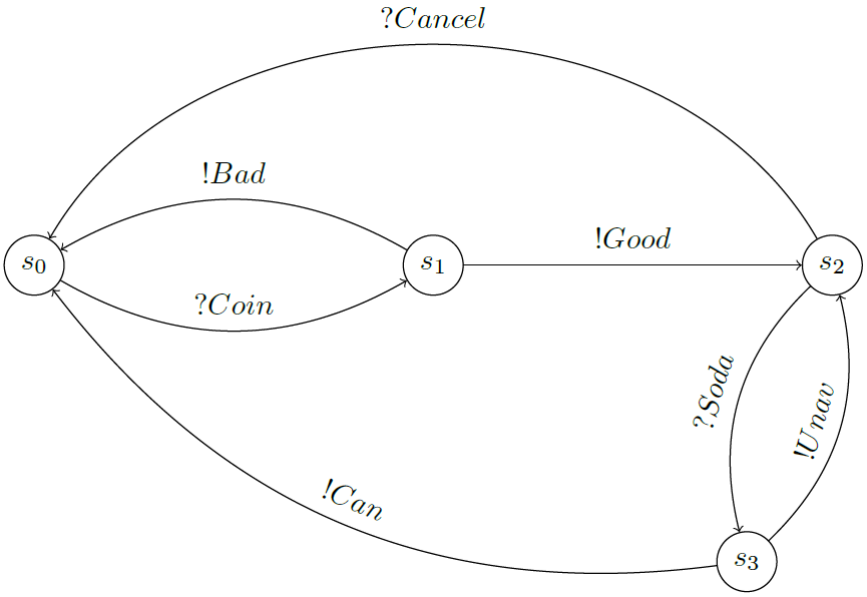


Figure 1: Graphe IOSM de notre exemple

Ce type de graphe met en exergue les échanges systèmes / environnement et la manière dont ils déterminent l'exécution du système. Les entrées fournies par l'environnement (ou le testeur) au système sont caractérisées par le symbole "?", tandis que les sorties fournies par le système à l'environnement / au testeur par "!". L'étude de ce type de modélisation va nous permettre de faire le lien avec la théorie des jeux.

Objectif de test

Au fur et à mesure de la phase de test, on va s'appuyer sur une série d'objectifs de test pour tester toutes les composantes du système.

Définition : Test Purpose

Un objectif de test TP est un IOSM acyclique. .

Pour notre exemple, l'ensemble des objectifs de test suivants suffit pour couvrir tous les comportements de l'automate.

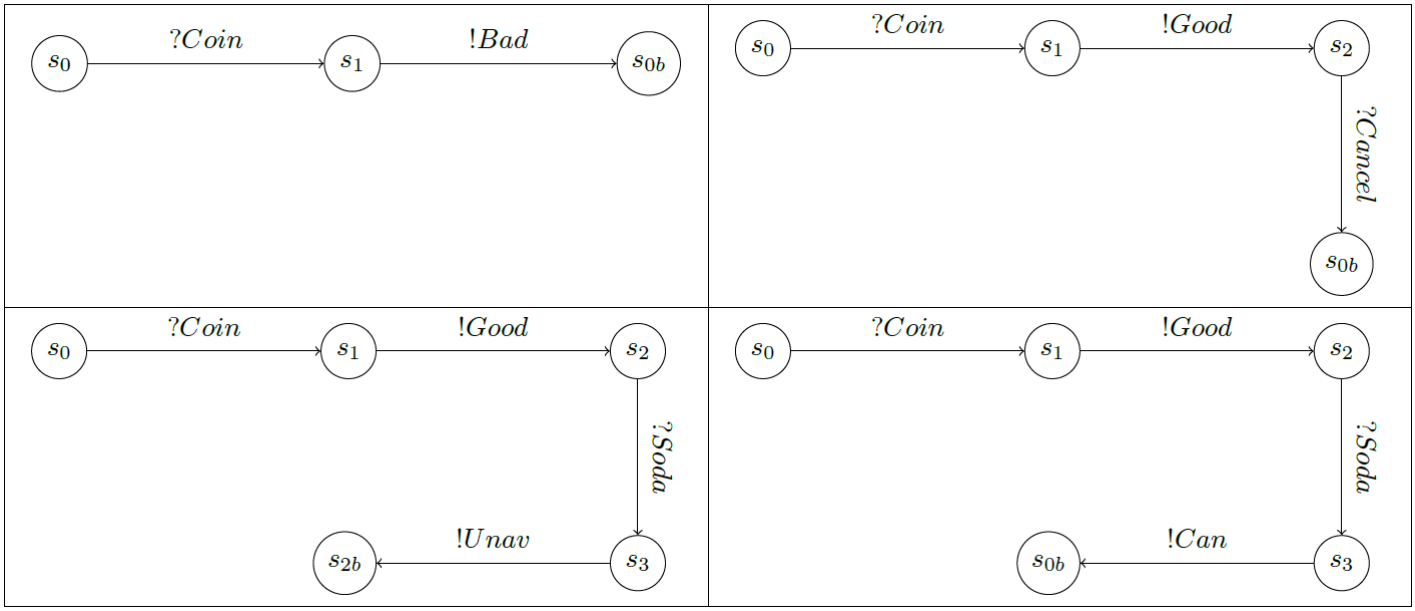


Figure 2: Converture complète par objectifs de test

Les exemples que nous avons donnés commencent tous par l'état initial s0 on peut cependant tout à fait avoir des objectifs de test se résumant simplement à des morceaux de chemins débutant à n'importe quel état comme par exemple :

Figure 3: Exemple d'objectif de test ne commençant pas par l'état initial

3 Réalisation pratique

Du graphe IOSM à la théorie des jeux

On peut systématiquement ré-exprimer des IOSM en tant que jeux à deux joueurs à information parfaite. En effet, dans le cadre du test, on peut considérer que les 2 joueurs sont le testeur et le système à tester, les coups joués par ces 2 joueurs étant : pour le testeur les inputs, et pour le système les outputs. Il suffit pour cela (si c'est nécessaire) d'intercaler des états et de créer des transitions silencieuses (τ) afin de transformer l'IOSM d'origine en un graphe biparti dirigé fini. En effet, sous cette forme, tout chemin alterne des états où l'on peut considérer que c'est "au testeur de jouer" et des états où c'est "au système de jouer".

On redonne à titre informatif la définition d'un jeu telle que vue en cours.

Définition : Jeu

Un jeu est un graphe biparti dirigé fini $G = (V, V_B, V_R, E)$ où:

- V est l'ensemble des sommets partitionnés en sommets bleus V_B et rouges V_R
- $E \subset V \times V$ est un ensemble d'arrêtes tel que:
  - $\forall\ (v, v') \in E$, on a $v \in V_B \Rightarrow v' \in V_R$ et $v \in V_R \Rightarrow v' \in V_B$ (bipartition)
  - $\forall\ v \in V,\ \exists\ v' \in V$ tel que $(v, v') \in E$ (chaque nœud a au moins une transition sortante)

.

Un partie est un mot (infini) de V^ω noté $x = v_0 v_1 ...$ tel que $\forall\ i \in \mathbb{N}, (v_i v_{i+1}) \in E$ et on pose comme convention $v_0 \in V_B$. Pour chaque nœud $v \in V$, si $v \in V_B$, alors c'est au joueur bleu de jouer et si $v \in V_R$, c'est au joueur rouge de jouer. On fixe un ensemble de parties dites *gagnantes*, qui le sont pour le joueur bleu. Le complémentaire de cette ensemble est l'ensemble des parties gagnantes pour le joueur rouge.
Une stratégie pour le joueur bleu est une fonction $f : (V_B.V_R)^*.V_B \to V_R$.
Une stratégie pour le joueur bleu est gagnante s'il gagne en la suivant (i.e. s'il gagne et que $\forall\ k \in \mathbb{N}, v_{2k} = f(v_0 ... v_{2k-1})$).

On aborde maintenant la méthode adoptée pour transformer une spécification IOSM en un tel jeu.

Définition : Jeu associé à une spécification IOSM

Soit $SUT = (S, L, T, s_0)$ une spécification IOSM. Le jeu $G = (V, V_B, V_R, E)$ qui lui est associé est le plus petit graphe biparti tel que:

- $(S \cup \{\bot\}) \subset V$
- $\{s \in S \mid \forall\ (s, t, s') \in T,\ \exists\ a \in L,\ t = !a\} \subset V_R$
- $V_B = V \backslash V_R$

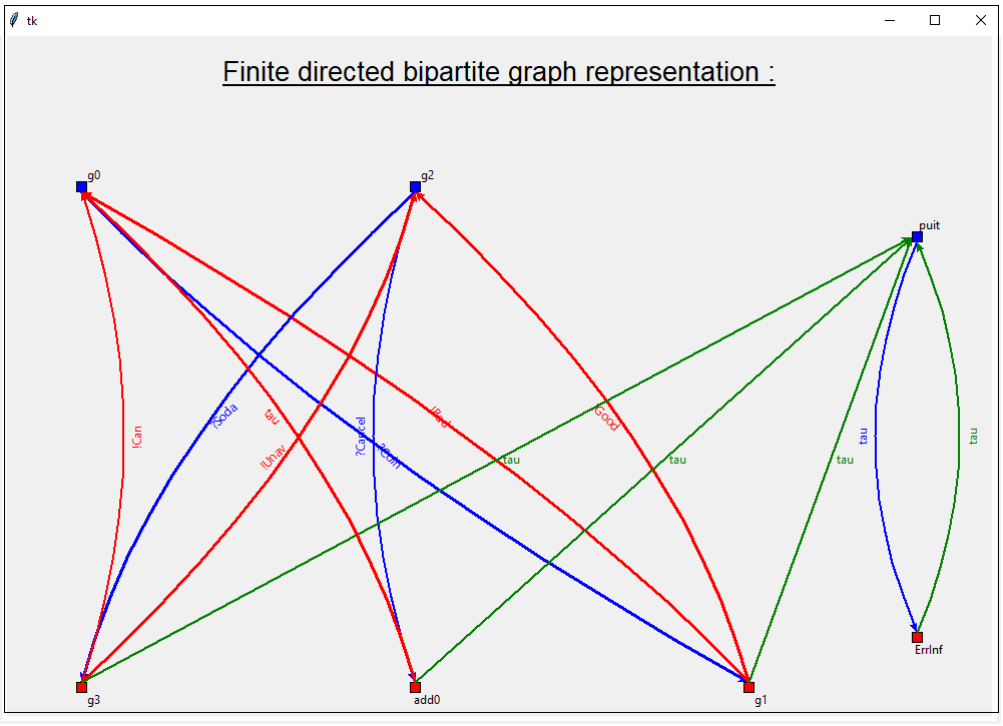
.

- $\forall\ v \in V_R, (v, \bot) \in E$
- $\forall (s, t, s') \in T$, on a:
  - $(s, s') \in E$ si s et s' ne sont pas de la même couleur dans G
  - sinon on intercale un nouveau nœud de la bonne couleur

L'implémentation de cet algorithme en Python est donnée en annexe A.
On constate que c'est au joueur rouge de choisir les sorties (émissions !a) du SUT tandis que c'est au joueur bleu de choisir les entrées (réceptions ?a).
Dans le cadre d'un test, le joueur rouge correspond donc au SUT tandis que le joueur bleu correspond au testeur.
L'état additionnel $\bot$, vers lequel on crée des transitions afin qu'il soit accessible depuis tous les noeuds rouges représentes la détection d'une non-conformité. Depuis tout noeud rouge, une sorties du SUT est attendue. Le testeur peut déclarer non conformité si quelque chose d'inattendu est émis (par exemple un message non décrit par l'automate...).

Pour plus de clarté, nous avons développé un affichage graphique permettant de visualier

en couleur la séparation du graphe IOSM en deux sous-familles de sommets correpondants aux deux joueurs, et dont l'union est un graphe FDB.
Le code de cet outil est disponible en annexe B, et voici le résultat obtenu pour le graphe d'exemple que nous avons rencontré précedemment.



Du graphe FDB à l'objectif de test

On va désormais chercher à relier un objectif de test au graphe FDB que nous avons obtenu précedemment. Pour cela, on va s'intéresser à la construction d'une stratégie gagnante, puis de relier le sommet initial t de cet objectif de test à l'entrée du système par la méthode des ensembles emboités.

On donne en Annexe C le code de génération d'objectifs de test aléatoires, ainsi que le code d'une fonction de vérification d'existence d'une séquence dans le graphe IOSM, pour vérifier la validité des objectifs de test que l'équipe 1 est susceptible de nous transmettre.

Il s'agit désormais de trouver une partie dans laquelle on "atteint t", t étant le noeud de départ de l'objectif de test dans graphe biparti. Pour se faire, on construit une suite croissante d'ensembles de sommets W_j^i . On part du sommet t et on parcours le graphe à l'envers jusqu'à retrouver s_0. Au vu de la nature bipartite du graphe, on doit avoir une méthode flexible, capable de calculer le "pas précédent" en fonction de l'acteur à qui c'est au tour de jouer.

Définition : Construction de la stratégie gagnante

On pose $W_0^0 = t$.
Et, $\forall\ i, j$:

$$W_j^{i+1} = W_j^i \cup \{p \in V_B \mid \exists\ q \in W_j^i : (p,q) \in E\} \cup \{p \in V_R \mid \forall\ q \in V_B : ((p,q) \in E) \Rightarrow (q \in W_j^i)\}$$

Et, sachant que $k_j = min\{i \mid W_j^{i+1} = W_j^i\}$:

$$W_{j+1}^0 = W_j^{k_j} \cup \left\{ p \in V_R \;\middle|\; \begin{array}{l} \exists\ q \in V_B \\ \exists\ r \in V_B \end{array} \;\middle|\; \begin{array}{c} (p,q) \in E \\ (p,r) \in E \\ q \in W_j^{k_j} \\ r \notin W_j^{k_j} \end{array} \right\}$$

Dans la définition ci-dessus on définit 2 types de "pas" qu'on peut schématiser comme étant des "pas horizon- taux" pour passer des W_j^i aux W_j^{i+1} et des "pas verticaux" pour passer des $W_j^{k_j}$ (une fois le parcours horizontal stable à k_j) aux W_{j+1}^0.

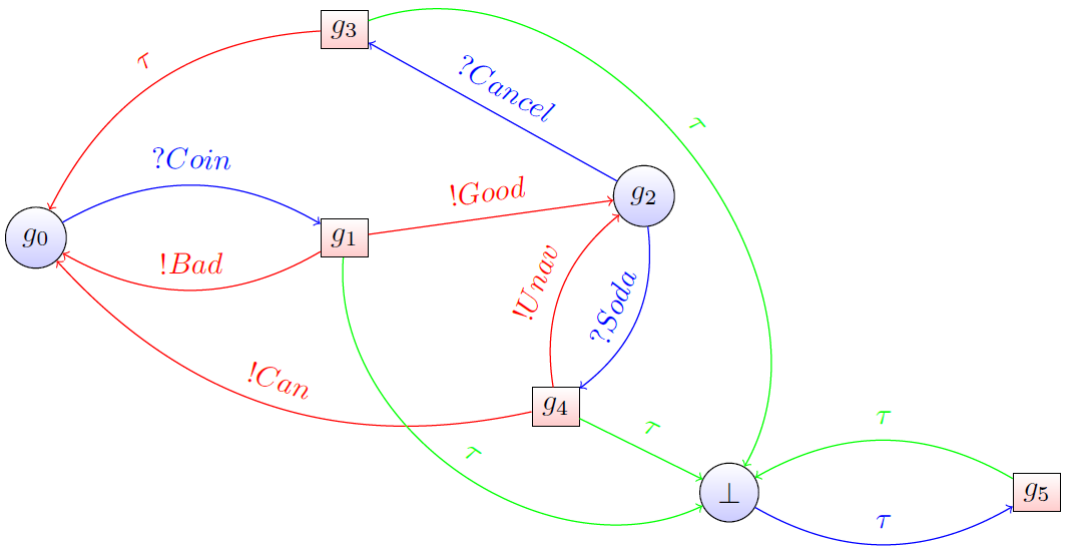
Dans les "pas horizontaux", on calcule W_j^{i+1}:

- on conserve un ensemble existant W_j^i et on l'agrandit:
- en y ajoutant tous les ancêtres bleus (joueur testeur) des nœud rouges de l'ensemble précédent W_j^i
- en y ajoutant une partie des ancêtres rouges (joueur système), dont les descendants font partie de l'ensemble précédent W_j^i

Cette construction permet au joueur testeur de contraindre le système dans l'ensemble de sommets emboîtés en lui enlevant l'accès aux transitions permettant d'en sortir.

On donne en Annexe D le code Python permettant la pondération des sommets du graphe IOSM telle que décrite ci-dessus.

Application sur l'exemple



On a alors la construction suivante :

$W_0^0 = \{g_4\}$, $W_1^0 = \{g_4, g_2\}$
$W_0^1 = \{g_4, g_2, g_1\}$, $W_1^1 = \{g_4, g_2, g_1, g_0\}$
$W_0^2 = \{g_4, g_2, g_1, g_0, g_3\}$
...

Notion de rang pour la formalisation d'une stratégie

Définition :

Le rang d'un sommet p est :

$$rank(p) = W_{j_{min}}^{i_{min}}$$

avec:

- $i_{min} = min(i \mid \exists\ j, p \in W_i^j)$
- $j_{min} = min(j \mid p \in W_{i_{min}}^j)$

On a $rank(p) = (\infty, \infty)$ si on ne peux pas l'atteindre par construction.

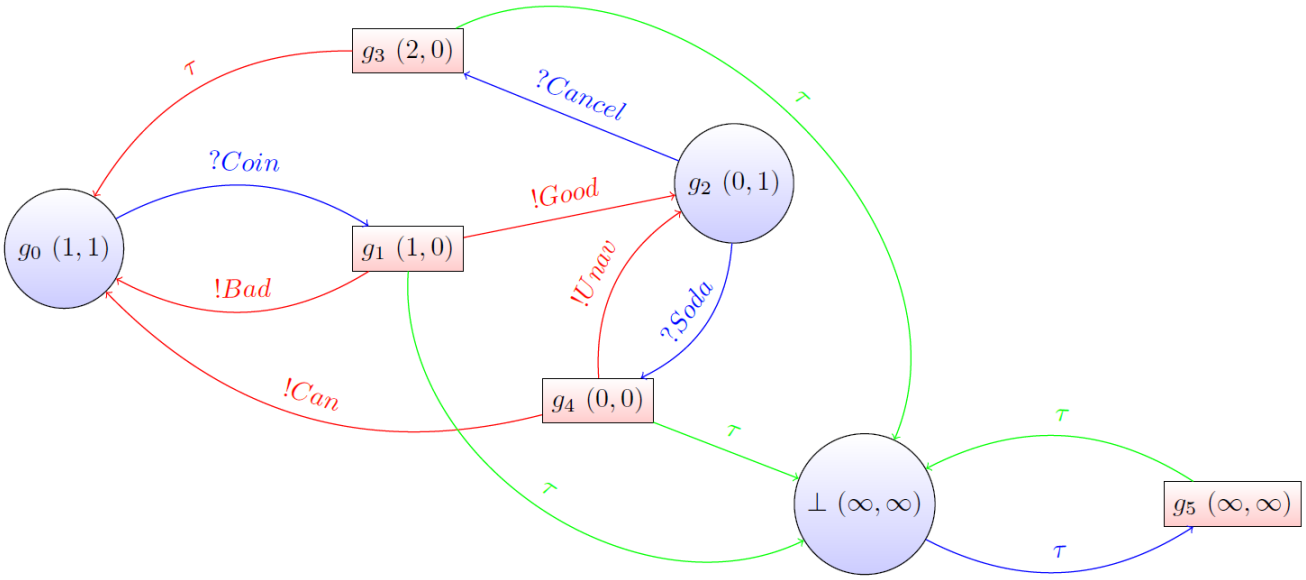
La stratégie proposée pour le joueur bleu (le testeur) est alors de "faire décroître le rang" c'est-à-dire de toujours choisir (quand c'est son tour) un sommet rouge de rang inférieur par rapport à l'ordre lexicographique gauche qui se définit ainsi :

$$(\forall\ (j, j'), i < i' \Rightarrow (i, j) < (i', j')) \bigwedge (\forall\ i, j < j' \Rightarrow (i, j) < (i, j'))$$

Cette application pratique de la recherche du plus petit point fixe issue du théorème de Tarski présente l'intérêt de permettre une implémentation par récurrence, ce qui dans le monde de la résolution numérique est très utile.

L'implémentation Python de cette recherche de chemin dans le graphe est donnée en Annexe E. On y retrouve une fonction permettant de transfomer le dictionnaire des pondérations pour permettre la descente dans le graphe, ainsi qu'une fonction de recherche de chemin "gagnant".

Application sur l'exemple



Le principe théorique donné précedemment donne une stratégie ne dépendant que du nœud actuel et des pondérations des nœuds accessibles:

$$f((V_B \cdot V_R)^* \cdot g_0) = g_1$$
$$f((V_B \cdot V_R)^* \cdot g_2) = g_4$$
$$f((V_B \cdot V_R)^* \cdot \bot) = g_5$$

Implémentation de l'intéraction SUT-testeur

Le code de l'Annexe F décrit l'intéraction entre le SUT et le testeur (représenté par une instance de la classe IATesteur). Ce dernier est chargé de suivre à quel nœud du graphe l'interaction en est, d'identifier diverses erreurs, et d'arrêter le programme lorsque le temps imparti est écoulé ou que l'objectif de test est atteint. La fonction testSeries est celle à lancer pour vérifier que les objectifs insérés dans lObj sont atteignables. Lorsque cette liste est vide, NObj sont créés de profondeur maximale DObj.
Il est à remarquer que le testeur agit bien sur une boîte noire. En effet, l'instance d'IATesteur n'agit pas directement dans les loop de la machine ; seulement la sortie a été adaptée pour pouvoir interagir à l'aide de l'instance d'IOController impliquée.
Voici une sortie enregistrée par une instance d'IOController à la fin d'un test :

```
1   [('g3', '!CAN'), ('g0', '?COIN'), ('g1', '!GOOD'), ('g2', '?SODA')]
2
3   COIN
4
5   GOOD
6
7   SODA
8
9   CAN
10
11  COIN
12
13  GOOD
14
15  TEST SUCCEEDS
```

4 Conclusion

Nous avons donc réussi à atteindre l'objectif de départ, à savoir réaliser un test dynamique du système. La méthode développée ici sur un cas plus simple (le distributeur automatique) est généralisable au problème du groupe : la distribution d'une mise à jour de capteurs au goutte à goutte.

Difficultés rencontrées

Les attendus n'étaient pas très évidents. On manquait peut-être d'un énoncé, même simple, commun au groupe et éventuellement segmenté pour les trois sous-groupes.
Le groupe manquait de coordination. Nous aurions dû mettre en place un représentant par sous-groupe et un représentant du groupe chargé de lier le travail de tous et de mettre en avant les tâches à exécuter.

Annexe : Implémentations Python

A Graphe IOSM vers graphe biparti dirigé fini (FDB)

```python
import copy

def graph_to_game(IOSM_graph):
    """graph_to_game transforms an IOSM (Input Output State Machine) graph
    into a finite directed bipartite graph.

    Args:
        IOSM_graph (tuple): IOSM graph, with structure (S, L, T, s0). S is a
                            set of vertexes, L the transition language set
                            and T a dictionnary of transitions between the
                            vertexes.
                            Structure of T is {'input vertex': [(output
                            vertex, transition name)]}.
                            s0 is always equal to S[0].

    Returns:
        tuple:   FDB_graph: finite directed bipartite graph corresponding to
                 the given IOSM graph, with structure (V, Vb, Vr, L, E, s0).
                 V is a list of vertexes, Vb a list of vertexes belonging to
                 blue player (Vb included in V), Vr a list of vertexes
                 belonging to red player (Vr included in V), L the
                 transition language set, E a list of edges, and s0
                 equal to S[0]. Structure of E is (input vertex,
                 transition name, output vertex).
    """

    _, L, T, s0 = IOSM_graph
    E = copy.deepcopy(T)

    color = {s0: 0}
    next = [s0]
    i = 0
    while len(next) > 0:
        n = next.pop()
        if n in E.keys():
            for neighbor, trans in E[n]:
                if neighbor not in color:
                    next.append(neighbor)
                    color[neighbor] = 1-color[n]
                elif color[neighbor] == color[n]:
                    new_node = "add"+str(i)

                    E[n].append((new_node, trans))
                    E[new_node] = [(neighbor, "tau")]
                    color[new_node] = 1-color[n]
                    i += 1

                    E[n].remove((neighbor, trans))

    Vb = set([x for x in color.keys() if color[x] == 0])
    Vr = set([x for x in color.keys() if color[x] == 1])

    FDB_graph = (Vb | Vr, Vb, Vr, L, E, s0)
    return FDB_graph
```

```python


def obj_to_FDB_graph(obj, FDB_graph):
    '''
    Converts obj to the FDB_graph without creating another list

    Variables
        - obj : test objective to complete, a list of [(first vertex,
                transition name of L)] extracted from IOSM_graph
        - FDB_graph : finite directed bipartite graph corresponding to the
                      given IOSM graph, with structure (V, Vb, Vr, L, E, s0)
    Returns :
        obj with vertices and "tau" transitions added when FDB_graph was
        built.
    '''
    (_, _, _, _, E, _) = FDB_graph
    for i, (node, trans) in enumerate(obj):
        for neigh, tr_node_to_neigh in E[node]:
            if tr_node_to_neigh == trans:
                if neigh[:3] == "add":
                    obj.insert(i+1, (neigh, "tau"))
    return obj


def add_fictif_point_init(ptAdded, transAdded, FDB_graph):
    '''
    Add a fictive point ptAdded that goes through transAdded to FDB_graph
    [-1]

    Remark : no need to link this red vertex to "puit"

    Variables :
        - ptAdded : name of the vertex created (str)
        - transAdded : name of the transition created from ptAdded to
                       FDB_graph[-1] (with transAdded[0] == "?")
        - FDB_graph :  finite directed bipartite graph with structure (V, Vb
                       , Vr, L, E, s0)

    Returns
        - new_graph : finite directed bipartite graph with structure
                      (V|{ptAdded}, Vb, Vr|{ptAdded}, L1, E1, s0), only
                      ptAdded and transAdded are created
    '''
    assert transAdded[0] == "!"

    V, Vb, Vr, L, E, s0 = FDB_graph
    V |= {ptAdded}
    Vr |= {ptAdded}
    L |= {transAdded[1:]}

    E[ptAdded] = [(s0, transAdded)]

    return (V, Vb, Vr, L, E, s0)

```

B Visualisation graphique du graphe FDB

```python
import tkinter as tk
```

```
import math

from matplotlib.pyplot import arrow, fill
from numpy import angle


def draw_FDB_graph(FDB_graph, l = 1000, h = 800):
    """draw_FDB_graph provides a graphical representation of the finite
       directed bipartite graph using the package Tkinter.

    Args:
        FDB_graph (tuple): a finite directed bipartite graph to be displayed
        l (int, optional): length of tkinter window. Defaults to 1000.
        h (int, optional): height of tkinter window. Defaults to 800.
    """

    w = tk.Tk()
    c = tk.Canvas(w, width=l, height=h)

    (S, Vb, Vr, L, E, s0) = FDB_graph
    N = len(S)

    x_b, x_r, y_b, y_r = l//(2*(N-1)), l//(2*(N-1)), 150, 650

    c.create_text(500, 35, font=("Purisa", 20, 'underline'), text="Finite
   directed bipartite graph representation :")

    Vb_indexed, Vr_indexed = {}, {}
    for vertex in S:
        if vertex == "puit":
            if "puit" in Vb:
                Vb_indexed["puit"] = l -  l//(2*(N-1))
            else:
                Vr_indexed["puit"] = l -  l//(2*(N-1))
            draw_vertex(c, l -  l//(2*(N-1)), 200, vertex, "blue")
        elif vertex == "ErrInf":
            if "ErrInf" in Vb:
                Vb_indexed["ErrInf"] = l -  l//(2*(N-1))
            else:
                Vr_indexed["ErrInf"] = l -  l//(2*(N-1))
            draw_vertex(c, l -  l//(2*(N-1)), 600, vertex, 'red')
        elif vertex in Vb:
            Vb_indexed[vertex] = x_b
            draw_vertex(c, x_b, y_b, vertex, 'blue')
            x_b += l//(N//2)
        elif vertex in Vr:
            Vr_indexed[vertex] = x_r
            draw_vertex(c, x_r, y_r, vertex, 'red')
            x_r += l//(N//2)

    for edge in E:
        for neighbour in E[edge]:
            input_vertex, label, output_vertex = edge, neighbour[1],
            neighbour[0]
            if output_vertex not in ["puit", "ErrInf"] and input_vertex not
            in ["puit", "ErrInf"]:
                if input_vertex in Vb:
                    x_0, x_1 = Vb_indexed[input_vertex], Vr_indexed[
                               output_vertex]
                    draw_edge(c, l, N, x_0, x_1, y_b, y_r, label, 'blue')
```

```
                elif input_vertex in Vr:
                    x_0, x_1 = Vr_indexed[input_vertex], Vb_indexed[
                                output_vertex]
                    draw_edge(c, l, N, x_0, x_1, y_r, y_b, label, 'red')
            else:
                if output_vertex == "puit":
                    if input_vertex == "ErrInf":
                        x_0 = l - l//(2*(N-1))
                        draw_edge(c, l, N, l - l//(2*(N-1)), l - l//(2*(N
                        -1)), 600, 200, 'tau', 'green')
                    elif input_vertex in Vb:
                        x_0 = Vb_indexed[input_vertex]
                        draw_tau(c, l, N, x_0, y_b + 5)
                    elif input_vertex in Vr:
                        x_0 = Vr_indexed[input_vertex]
                        draw_tau(c, l, N, x_0, y_r - 5)
                else:
                    draw_edge(c, l, N, l - l//(2*(N-1)), l - l//(2*(N-1)),
                    200, 600, 'tau', 'blue')



    c.pack()
    w.mainloop()


def draw_vertex(c, x, y, name, color):
    c.create_rectangle(x-5, y-5, x+5, y+5, fill=color)
    if color == 'blue':
        c.create_text(x+12, y-12, text=name)
    else:
        c.create_text(x+12, y+12, text=name)

def draw_edge(c, l, N, x_0, x_1, y_0, y_1, label, color):
    if x_0 == x_1:
        if color == 'blue':
            c.create_line(x_0, y_0 + 5, (x_0 + x_1)//2-l//(2*(N - 1)),
            400, x_1, y_1-5, arrow=tk.LAST, fill= color, smooth=1, width=2)
            c.create_text((x_0 + x_1)//2 - l//(2*(N+2)), 400, text = label,
            fill= color, angle=90)
        else:
            c.create_line(x_0, y_0 - 5, (x_0 + x_1)//2 + l//(2*(N-1)), 400
            ,x_1, y_1+5, arrow=tk.LAST, fill= color, smooth=1, width=2)
            c.create_text((x_0 + x_1)//2 + l//(2*(N+2)), 400, text = label,
            fill= color, angle=90)
    else:
        if color == 'blue':
            c.create_line(x_0, y_0 + 5, (x_0 + x_1)//2 - l//(2*(N-2)), 400,
            x_1, y_1-5, arrow=tk.LAST, fill= color, smooth=1, width=3)
            if x_0 < x_1:
                x_test, y_test, orientation = (x_0 + x_1)//2 - 25 , 420, -45
            else:
                x_test, y_test, orientation = (x_0 + x_1)//2 - 25, 380, 45

            c.create_text(x_test , y_test, text = label, fill= color, angle
             = orientation)
        else:
            c.create_line(x_0, y_0 - 5, (x_0 + x_1)//2 + l//(2*(N-2)), 400,
            x_1, y_1+5, arrow=tk.LAST, fill= color, smooth=1, width=3)
            if x_0 < x_1:
```

```
                x_test, y_test, orientation = (x_0 + x_1)//2 + 25, 420, 45
            else:
                x_test, y_test, orientation = (x_0 + x_1)//2 + 25, 380, -45
            c.create_text(x_test, y_test, text = label, fill= color, angle =
            orientation)

def draw_tau(c, l, N, x_0, y_0):
    c.create_line(x_0, y_0,  l -  l//(2*(N-1)) - 5, 200, arrow=tk.LAST, fill
    ="green", smooth=1, width = 2)
    c.create_text((x_0 + l -  l//(2*(N-1))) // 2 + 12, (y_0 + 200) // 2,
    text="tau", fill = "green")
```

C Génération aléatoire d'objectifs de test

```
import random

def randomObjectifTest(IOSM_graph, N):
    """
    Prend en entree un graphe IOSM et un entier N qui represente la taille
    du chemin voulu. Return un objectif de test aleatoire de taille N.
    """
    S, _, E, _ = IOSM_graph
    objTest = []
    c_node = random.choice(list(S))
    for _ in range(N):
        next_node, c_trans = random.choice(E[c_node])
        objTest.append((c_node, c_trans))
        c_node = next_node
    return objTest




def existChemin(c, graph, IOSM_is_graph_type=True):
    """
    Are the test objective c and graph consistent together

    Variables :
        - c (list): test objective, a list like [(s0,trans name from s0), (
                    s1, trans name from s1)]
        - graph (tuple): an IOSM or FDB grap
        - IOSM_is_graph_type (bool) : indicate the type of the graph, if
                                      false, graph is an FDB graph
    Returns :
        boolean indicating if the objective c and the graph graph are
        consistent together
    """
    if IOSM_is_graph_type:
        S, _, T, _ = graph
    else:
        # graph type is FDB (V,Vb,Vr,L,E,s0)
        S, _, _, _, T, _ = graph
    for i, (node, trans) in enumerate(c):
        if not(node in S):
            return False
        else:
            found_trans = False
            for neigh, neigh_trans in T[node]:
```

```
                if neigh_trans == trans:
                    if found_trans:
                        return False
                    if i <= len(c)-2 and neigh != c[i+1][0]:
                        return False
                    found_trans = True
    return True
```

D Pondération du graphe IOSM par méthode des W_j^i

```
import math
import transform

def dico_W(t, FDB_graph):
    '''
    BFS

    Variables:
        - t = vertex we want to end (str)
        - FDB_graph = (V,Vb,Vr,L,E,s0) a bipartite graph with "puit" and
                                        "ErrInf" in E.keys()

    Returns :
        dict_W = dictionnary with key a node and value a tuple (i,j) where i
    ,j are those of Wij
    '''
    (_, Vb, _, _, E, _) = FDB_graph
    dict_W = {t: (0, 0)}
    lnext = [t]
    # passed_through = {t: True}
    E = dico_reverse(E)
    while len(lnext):
        c_node = lnext.pop(0)
        (i, j) = dict_W[c_node]
        for neigh, _ in E[c_node]:
            if neigh in Vb:
                if not(neigh in dict_W) or dict_W[neigh][0] > i+1:
                    dict_W[neigh] = i+1, j
                    lnext.append(neigh)
            else:
                if not(neigh in dict_W) or dict_W[neigh][0] > 0:
                    dict_W[neigh] = 0, j+1
                    lnext.append(neigh)
    return dict_W
```

E Recherche de stratégie "gagnante"

```
import math
import transform

def dico_reverse(dico):
    '''
    Exchanges directions in a directed graph

    Variable :

```

```
        dico : a dictionary with key : a node, value : the list of tuples
               (nodes the key is directly linked to, the transfer string)

    returns :
        dico_rev : dico with reverse directions
    '''
    dico_rev = {}
    for key in dico:
        for neigh, n_trans in dico[key]:
            if not(neigh in dico_rev):
                dico_rev[neigh] = [(key, n_trans)]
            else:
                dico_rev[neigh].append((key, n_trans))
    return dico_rev


def choose_transition(s, graph, dict_W):
    '''
    Return the best path from s to t

    Variables :
        - s = beginning vertex, s in Vb (str)
        - graph = (V,Vb,Vr,L, E,s0) a bipartite graph
        - dict_W = dictionary returned by dico_W(,,)

    Return :
        best_neigh, best_trans : blue player has to choose
    '''
    (_, _, _, _, E, _) = graph

    best_j, best_neigh, best_trans = math.inf, None, None
    for neigh, trans in E[s]:
        # print("W",dict_W)
        if dict_W[neigh][1] <= best_j:
            best_j, best_neigh, best_trans = dict_W[neigh][1], neigh, trans
    return best_neigh, best_trans
```

F Testeur Online

```
from choice_to_make import choose_transition, dico_W
from objective_creation import existChemin
from transform import graph_to_game, obj_to_FDB_graph, add_fictif_point_init
from objective_creation import randomObjectifTest
import os
import sys
import datetime
import random

# start to read at start_machine(...)
# command() of IOController to change()


def simulate_non_determinism():
    '''
    Change it for a denser tree (with more than 2 choices max per vertex)
    '''
    return random.choice([True, False])

```

```
20
21 class SystemSignatureViolation(Exception):
22     pass
23
24
25 class IATesteurError(Exception):
26     pass
27
28
29 class TestOver(Exception):
30     pass
31
32
33 class ObjectiveNotCompatible(Exception):
34     pass
35
36 class TimeOut(Exception):
37     pass
38
39 def can_loop(io_controller, adv, position=None, FDB_graph=None):
40     if adv.testOver():
41         raise TestOver()
42     ev_name = io_controller.get_input(adv)
43     if(ev_name == "SODA"):
44         g = simulate_non_determinism()
45         if g:
46             car = "CAN"
47             io_controller.transition = car
48             io_controller.write_output(car)
49             return False
50         else:
51             car = "UNAV"
52             io_controller.transition = car
53             io_controller.write_output(car)
54             return True
55     elif(ev_name == "CANCEL"):
56         if adv.name == "IATesteur":
57             # only matters to share tau artificial output in this case
58             car = "tau"
59             io_controller.transition = car
60             io_controller.write_output(car)
61         return False
62     else:
63         raise SystemSignatureViolation()
64
65
66 def coin_loop(io_controller, adv):
67     if adv.testOver():
68         raise TestOver()
69
70     ev_name = io_controller.get_input(adv)
71     if(ev_name == "COIN"):
72         g = simulate_non_determinism()
73         if g:
74             car = 'GOOD'
75             io_controller.transition = car
76             io_controller.write_output(car)
77             while(g):
78                 g = can_loop(io_controller, adv)
79         else:
```

```
            car = "BAD"
            io_controller.transition = car
            io_controller.write_output(car)
    else:
        raise SystemSignatureViolation()


def start_machine(io_controller, adv):
    '''
    Variables :
     - io_controller : SUT to test
     - adv : test operator
    '''

    while(True):
        coin_loop(io_controller, adv)


class IOController(object):
    '''
    Modified in order to inform the test where the system is
    '''

    def __init__(self, transAdded):
        self.transition = transAdded[1:]
        self.terminal = sys.stdout
        current_time = datetime.datetime.now().strftime("%Y_%m_%d_%H_%M_%S")
        self.log = open("boisson_record_%s.log" % current_time, "a")

    def write_output(self, message):
        if not message.endswith(os.linesep):
            message += os.linesep
        # self.terminal.write(message)
        self.log.write(message)

    def get_input(self, adv):
        got = adv.command(self.transition)
        if not got.endswith(os.linesep):
            self.log.write(got + os.linesep)
        else:
            self.log.write(got)
        return got


class HumanPlayer(object):
    def __init__(self):
        self.name = "human"
        self.transition = None
        self.Obj_Reached = False
        self.position = None

    def command(self, transition):
        print(transition)
        self.transition = input()
        return self.transition

    def testOver(self):
        return False
```

```python
def deduce_next_position(pos, ext_trans, FDB_graph):
    '''

    Returns :
        - next_pos : the vertex at the end of the transition starting from pos named ext_trans
    '''
    # set the new position
    found_trans = False
    next_pos = None
    bool_tau = (ext_trans[1:] == "tau")
    if bool_tau:
        ext_trans = ext_trans[1:]
    for neigh, trans in FDB_graph[4][pos]:
        if trans == ext_trans:
            if bool_tau:
                return neigh
            elif found_trans:
                # force unique possibility
                raise SystemSignatureViolation()
            found_trans = True
            next_pos = neigh
    if next_pos == None:
        print("IATesteur is blocked on a vertex.")
        raise IATesteurError()
    return next_pos


class IATesteur(object):
    '''
    Aims to test online another system with its results compared to a FDB graph where transition are strings with the first caracter being ! or ?
    '''

    def __init__(self, obj, FDB_graph, ptAdded, Nmax_transitions):
        '''
        Variables :
            - obj : test objective to complete, a list of [(first vertex,
                    transition name of L)] extracted from IOSM_graph
            - FDB_grah (tuple) : finite directed bipartite graph with
                                 structure (V, Vb, Vr, L, E, s0)
            - ptAdded (str) : the fictive point added before the beginning
                              for the initialisation
        '''
        self.name = "IATesteur"
        self.position = ptAdded  # fictive point
        self.counter = 0
        self.Nmax_transitions = Nmax_transitions
        self.FDB_graph = FDB_graph
        new_obj = obj_to_FDB_graph(obj, self.FDB_graph)
        if existChemin(new_obj, FDB_graph, IOSM_is_graph_type=False):
            self.objective = new_obj
        else:
            print("obj_to_FDB_graph is not working")
            raise ObjectiveNotCompatible()
        self.completion = []
        self.dict_W = {}  # to change
        self.Obj_Reached = (self.completion == self.objective)

        # for empty objective
```

```
198         if self.testOver():
199             raise TestOver()
200 
201     def testOver(self):
202         return self.Obj_Reached and self.position == self.FDB_graph[-1]
203 
204     def update_comp(self, pos, trans):
205         '''
206         1) DO NOT Change position of IATesteur
207         2) Add (pos,trans) to self.completion if next element in objective
208         3) Set self.completion = [] if the test gets out of the objective
209         4) Set self.dict_W = {} if IATesteur is in the objective
210 
211         Variables :
212             -pos : the vertex from which tran starts (str) (ex: "g0")
213             -trans : the transition name one of the player just passed
214                      through (str) (ex : "!COIN")
215         '''
216 
217         if not(self.Obj_Reached):
218             # second  == to adapt if the punctuation is kept or not
219             if pos == self.objective[len(self.completion)][0] and trans ==
    self.objective[len(self.completion)][1]:
220                 if not(len(self.completion)):
221                     # not to go in here too often, first time you go in
    completion
222                     self.dict_W = {}
223                 self.completion.append((pos, trans))
224             else:
225                 if len(self.dict_W.keys()):
226                     # first time you get out of the objective
227                     self.dict_W = dico_W(self.objective[0][0], self.
    FDB_graph)
228                 self.completion = []
229             self.Obj_Reached = (self.completion == self.objective)
230 
231     def command(self, io_transition):
232         '''
233         Returns a command to complete IATesteur objective, it needs
234         position, completion, dict_W, Obj_Reached updated by the SUT,
235         while playing.
236         '''
237         self.counter +=1
238 
239         if self.counter > self.Nmax_transitions:
240             raise TimeOut()
241 
242         obj = self.objective
243         dict_W = self.dict_W
244 
245         # set if the objective is reached
246         if io_transition == "tau":
247             self.update_comp(self.position, io_transition)
248         else:
249             self.update_comp(self.position, "!"+io_transition)
250 
251         if self.testOver():
252             raise TestOver()
253 
254         # set new position
```

```
255         self.position = deduce_next_position(
256             self.position, "!"+io_transition, self.FDB_graph)
257 
258         trans_next = None
259         # continue the completion of the objective, once its completion has
260         # begun
261         if not(self.Obj_Reached) and self.position == obj[len(self.
    completion)][0]:
262             trans_next = obj[len(self.completion)][1]
263             pos_next = deduce_next_position(
264                 self.position, trans_next, self.FDB_graph)
265             self.update_comp(self.position, trans_next)
266             self.position = pos_next
267 
268         # get to the first point of objective, when it is not reached
269         elif not(self.Obj_Reached) and not(len(self.completion)):
270             if not(len(dict_W)):
271                 dict_W = dico_W(obj[0][0], self.FDB_graph)
272             # not empty anymore
273             pos_next, trans_next = choose_transition(
274                 self.position, self.FDB_graph, dict_W)
275             # in order (current position, trans_next), (pos_next, ...) if
276             # alright with objective
277             self.update_comp(self.position, trans_next)
278             self.position = pos_next
279 
280         # return to the first point of departure, once the objective is
281         # completed
282         if self.Obj_Reached and self.position != self.FDB_graph[-1]:
283             if not(len(dict_W)):
284                 dict_W = dico_W(self.FDB_graph[-1], self.FDB_graph)
285             pos_next, trans_next = choose_transition(
286                 self.position, self.FDB_graph, dict_W)
287             # no need to update as the method is to reach the completion of
288             # the objective
289             self.position = pos_next
290 
291         if self.testOver():
292             raise TestOver()
293 
294         if trans_next == None:
295             raise IATesteurError()
296         self.transition = trans_next[1:]
297         return trans_next[1:]
298 
299 
300 def test_series(IOSM_graph, Nmax_transitions = 40, l_obj=[], N_obj=2, D_obj
    =6):
301     '''
302     It checks each test objective of l_obj on an SUT (in this .py file, a
    vending machine).
303     If no objective is given, it randomly creates N_obj test objectives, of
    length from 1 to D_obj
304 
305     Variables :
306         - IOSM_graph (tuple): IOSM graph, with structure (S, L, T, s0). S is
307                               a set of vertexes, L the transition language
308                               set and T a dictionnary of transitions between
309                               the vertexes. Structure of T is {'input
310                               vertex': [(output vertex, transition name)]}.
```

```
                                s0 is always equal to S[0].
        - l_obj (list): a list of test objectives, each test objective being
                        a list of the form [(node0, transition from node0),
                        ...]
        - N_obj (int): the lenght of the list of objectives automatically
                       created if l_obj is not given or empty
        - D_obj (int): the maximum length of an objective of the
                       automatically created objectives
    '''
    FDB_graph = graph_to_game(IOSM_graph)
    ptAdded = "fictPoint"
    transAdded = "!FICTRANS"
    FDB_graph = add_fictif_point_init(ptAdded, transAdded, FDB_graph)
    print("deduced FDB graph  :")
    print("V, Vb, Vr : ", FDB_graph[:3])
    print("L : ", FDB_graph[3])
    print("E : ", FDB_graph[4])
    print("s0 : ", FDB_graph[5])
    io_controller = IOController(transAdded)

    if not len(l_obj):
        for _ in range(N_obj):
            d = random.randint(1, D_obj)
            l_obj.append(randomObjectifTest(IOSM_graph, d))

    for obj in l_obj:
        if not existChemin(obj, IOSM_graph):
            io_controller.write_output(str(obj))
            raise ObjectiveNotCompatible()
        test_obj(obj, io_controller, ptAdded, transAdded, FDB_graph)


def test_obj(obj, io_controller, ptAdded, transAdded, FDB_graph,
    Nmax_transitions = 40):
    adv = IATesteur(obj, FDB_graph, ptAdded, Nmax_transitions)
    io_controller.transition = transAdded[1:]
    io_controller.write_output(str(obj))
    try:
        start_machine(io_controller, adv)
    except SystemSignatureViolation:
        io_controller.write_output("ABORT")
    except TestOver:
        io_controller.write_output("TEST SUCCEEDS")
    except IATesteurError:
        io_controller.write_output(
            "IATesteur FAILED, IT DID NOT FIND THE NEXT POSITION OR
            TRANSITION")
    except ObjectiveNotCompatible:
        io_controller.write_output(str(obj))
        io_controller.write_output(
            "THE OBJECTIVE AND THE IOSM GRAPH ARE NOT CONSISTENT TOGETHER")
    except TimeOut:
        io_controller.write_output(str(obj))
        io_controller.write_output("THE TEST TOOK TOO LONG")

def run_human():
    io_controller = IOController("")
    adv = HumanPlayer()
    try:
        start_machine(io_controller, adv)
```

```
370     except SystemSignatureViolation:
371         io_controller.write_output("ABORT")
```