CentraleSupélec

Enseignement d'Intégration
Jeux adversariaux

Introduction au test & lien proposé avec la théorie des jeux

Erwan Mahé	`Laboratoire LICIA, CEA-List`
Pascale Le Gall	`Laboratoire MICS, CentraleSupélec`
Marc Aiguier	`Laboratoire MICS, CentraleSupélec`
Arnault Lapître	`Laboratoire LECS, CEA-List`
Boutheïna Bannour	`Laboratoire LECS, CEA-List`

- *Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes (MICS)*
  CentraleSupélec - Plateau de Moulon
  9 rue Joliot-Curie
  F-91192 Gif-sur-Yvette Cedex
- *Laboratoire d'Intégration de Systèmes et des Technologies (LIST)*
  Centre d'intégration Nano-INNOV
  8 avenue de la Vauve
  91120 PALAISEAU

Résumé – Dans ce document nous introduisons la problématique du test de systèmes logiciels et cyber-physiques (sytèmes embarqués, analogique-digital ...). Nous y proposons une exploration des liens pouvant être faits entre tests basés sur les modèles et théorie des jeux. Le but est de pouvoir exploiter des concepts et résultats de théorie des jeux dans le cadre du test. Nous nous somme basés pour cette approche sur l'article suivant : [2].

Keywords – conformance testing, model-based testing, test purposes, test generation, oracle, state-machine models, input-output state machines, game theory, game strategies

Contents

1 Test et test basé sur les modèles (MBT)

Dans cette section nous allons rapidement définir le test système. Nous décrierons de manière concise cette discipline et nous introduirons certains concepts et éléments de vocabulaire importants.

1.1 Le test : principe général et définitions

1.1.1 Définitions

Définition générale

En toute généralité le test consiste en la vérification de certaines propriétés d'un système. Ces propriétés peuvent correspondre à des exigences fonctionnelles (adéquation entrées/sorties, stimuli/observations, variables intermédiaires ...) comme non fonctionnelles (performances, sécurité, documentation, respect d'interfaces ...) mais doivent toujours pouvoir se réduire à l'évaluation d'un booléen.

Le test peut s'appliquer à de nombreux types de systèmes; programmes informatiques, systèmes cyber-physiques (Cyber Physical Systems (CPS) par exemple en aéronautique, automobile, systèmes embarqués ...), qui peuvent être temporisés ou non, communiquer de manière discrète (évènements d'émissions et de réceptions ponctuels), continue ou encore hybride, être déterministe dans leur exécution ou non ...

Vocabulaire

On désigne le système soumis à la procédure de test par l'acronyme SUT pour System Under Test (système sous test).

On parle de harnais de test (test harness) pour désigner l'outillage permettant aux tests d'être effectués. Il peut s'agir d'un banc de test concret pour les sytèmes physiques ou bien d'outils logiciels (librairie de test, tests unitaires, simulateurs, ...) pour les systèmes logiciels.

Distinction statique (offline) / dynamique (online)

Le test peut être dynamique, c'est à dire qu'il accompagne (éventuellement guide) une exécution du SUT, ou bien statique, i.e. portant sur une trace d'exécution (logs, ...) obtenue au préalable i.e. issue d'une exécution passée du SUT. Les tests dynamiques peuvent tout aussi bien caractériser des exécutions finies (en temps, en nombres d'évènements ...) comme infinies (par exemple en garantissant ne jamais atteindre un état, ou revenir infiniment souvent dans un état ...).

Distinction boîte blanche/noire

Dans les tests en boîte blanche on peut évaluer des données internes (normalement non accessibles aux utilisateurs finaux) au SUT au cours du test. Typiquement, il s'agirait pour ces données internes de variables intermédiaires pour les programmes informatiques (dont les valeurs sont capturés par exemple via des points d'arrêt avec un débogueur interactif ...) ou bien de données issues d'un outillage en banc de test (capteurs de température, pression ...) pour certains CPS; ces données n'étant normalement pas accessible dans le produit fini.

Dans les tests en boîte noire au contraire, les moyens d'intéragir avec le SUT (entrées et sorties) se limitent à l'interface prévue pour l'utilisateur final.

Restriction du problème

Nous nous intéresserons uniquement par la suite à des systèmes communicant de manière discrète et à des tests dynamiques en boîte noire. Cette "restriction" permet déjà de tester la plupart des systèmes embarqués et protocoles de communication utilisés dans l'industrie. Dans ce papier, nous nous intéresserons plus particulièrement à l'application de la théorie des jeux pour ce type de test. Toutefois, nous ne ferons ici qu'effleurer le sujet; nous nous limiterons à des systèmes centralisé (par oppositions aux systèmes distribués et/ou multi-threadés) et à des modèles comportementaux simplistes (transitions non gardées ...).

Brique de base du test

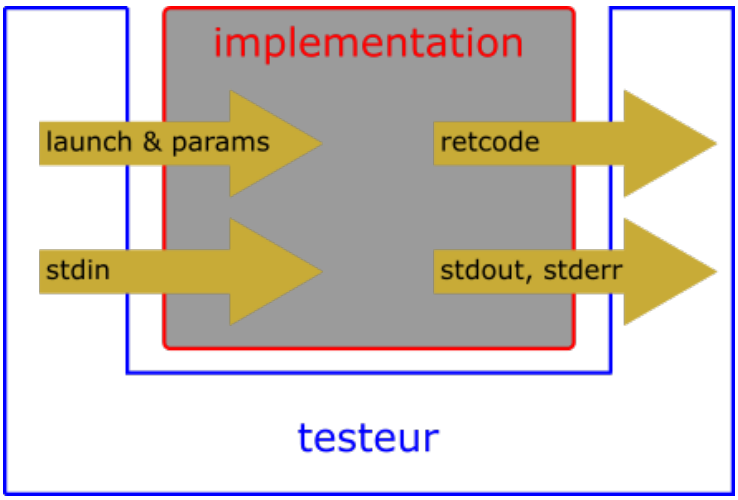
Rappelons que notre restriction signifie que le résultat des tests dépends uniquement de leur définition et de comparaisons faîtes sur les interactions (entrées et sorties) entre le système et son environnement; un test pouvant porter sur une séquence finie ou infinie (du moins dans le cadre théorique) de telles entrées et sorties.

La brique de base de ce genre de test est donc une relation du type suivant (explication en langage naturel) : "SI *j'envoie* **x** *au système* ALORS *je dois recevoir* **y** *du système*"

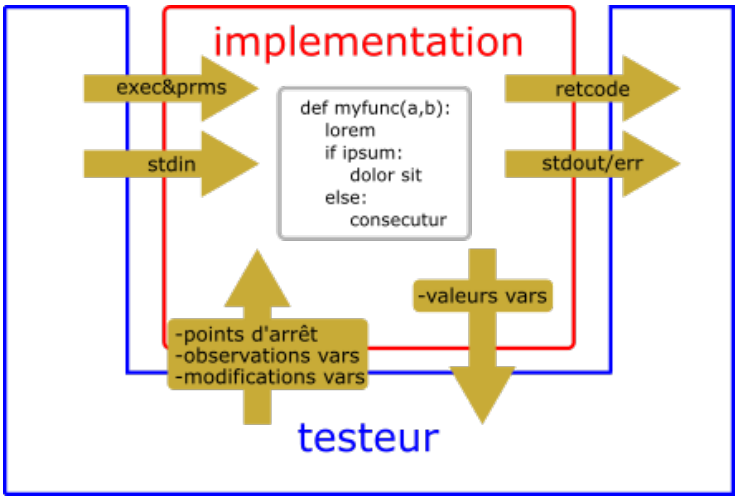
1.1.2 Schémas explicatifs (cas généraux)

Dans les schémas qui suivent nous donnons les catégories générales dans lesquelles on peut trouver ces valeurs **x** (entrées; qui vont du testeur vers l'implémentation) et **y** (sorties; qui vont de l'implémentation au testeur) dans les cas de systèmes logiciels ou CPS, qu'ils soient testés en boîte noire ou blanche.

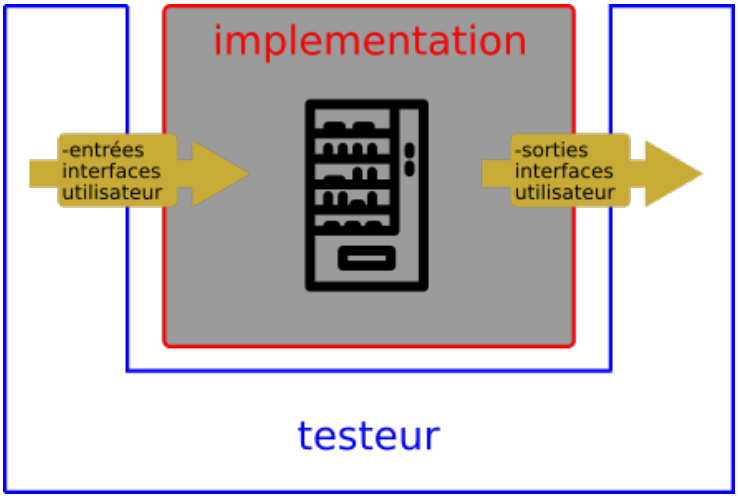
Test boîte noire d'un système logiciel



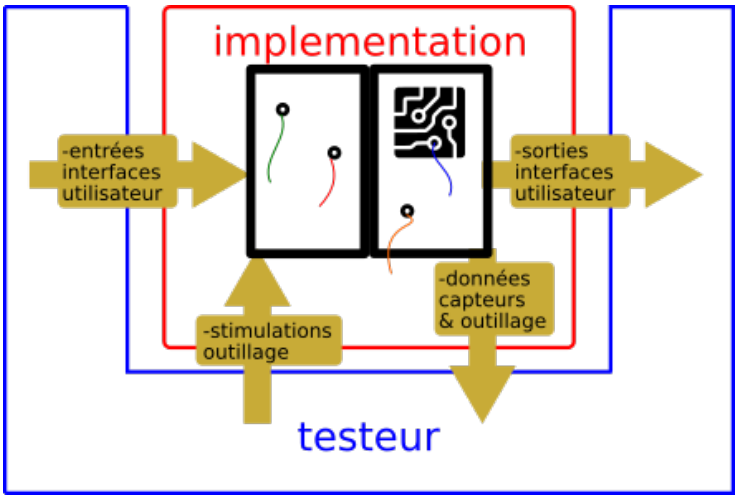
Test boîte blanche d'un système logiciel



Test boîte noire d'un système cyber-physique



Test boîte blanche d'un système cyber-physique



Nous savons maintenant comment se passe un test mais nous ne savons pas quoi tester. Le concept de sélection de test va nous permettre d'introduire cette problématique. Nous nous intéresserons aussi à cette occasion au test basé sur les modèles.

1.2 Principe de sélection des tests

Dans cette section nous évoquerons la nécessité pratique de sélectionner un sous ensemble fini de tests et nous donnerons quelques pistes sur la manière dont cela peut être fait à l'aide d'exemples triviaux.

Exemple simple sur un programme informatique

Considérons un programme informatique prenant en entrée un entier e et retournant un booléen s valant $True$ si $e > 0$ et $False$ sinon (ceci est la spécification en langage naturel du programme).

Pour vérifier la spécification, selon le langage de programmation utilisé et les limitations de la machine, la combinatoire des entrées possibles (même en supposant le bon typage des entrées) pourrais nécessiter une très grande quantité de tests (rédhibitoire et/ou extrêmement superflus, voir ci-dessous). Se pose donc la problématique de la sélection (pertinente) des tests.

```
1  import unittest
2
3  class Myf_Tests(unittest.TestCase):
4
5      "..."
6
7      def test_T_m1(self):
8          assert( myf(-1) == False )
9
10     def test_T_0(self):
11         assert( myf(0) == False )
12
13     def test_T_p1(self):
14         assert( myf(1) == True )
15
16     "..."
17
18 if __name__ == '__main__':
19     unittest.main()
```

Si on veut une sélection finie / de taille raisonnable, sans information supplémentaire on ne peut rien faire de mieux qu'une sélection aléatoire. Pour aller plus loin, il nous faut un modèle comportemental du système. Un tel modèle, associé à un critère de sélection nous permettra alors de définir un recouvrement de l'espace des données d'entrées. Il suffira alors de choisir un ensemble de valeurs dans chacune des zones du recouvrement pour satisfaire le critère de sélection.

Dans le cas de systèmes logiciels, dans certains projets où l'on dispose du code source (ou bytecode) du programme, on peut s'en servir directement comme modèle comportemental. L'usage de ce modèle est souvent associé à un critère de couverture de code/branche permettant une sélection (considérée comme plus pertinente) des tests.

Supposons que l'on ait le code Python ci-dessous. Pour satisfaire le critère de couverture de code il est nécessaire et suffisant que l'on ait juste assez de cas de test pour que l'exécution "passe" par toutes les "lignes de code".

On identifie "à la main" les 2 branches de notre modèle. Chacune est associée à des contraintes sur les entrées du système; contraintes qui définissent une partition de l'espace des données. On peut alors définir un test pour chacune des branches (ici on a pris des valeurs aléatoires dans chacun des domaines).

```python
def is_pos(x):
    if x > 0:
        return True # branche 0
    if x < 0:
        return False # branche 1

# ======================================================================
import unittest

class Myf_Tests(unittest.TestCase):

    def test_branch_0(self):
        assert( is_pos(578) == True )

    def test_branch_1(self):
        assert( is_pos(-38) == False )

if __name__ == '__main__':
    unittest.main()
```

Il existe des outils pour évaluer la couverture de test, c'est-à-dire qu'ils donnent le pourcentage de lignes de code couvertes par un ensemble de tests. Avec Python on peut par exemple utiliser l'outil "coverage" comme décrit dans la Figure 1.

```
PS C:\> coverage run .\myf.py
..
--------------------------------
Ran 2 tests in 0.001s

OK
PS C:\> coverage report
Name     Stmts   Miss  Cover
--------------------------------
myf.py      13      0   100%
--------------------------------
TOTAL       13      0   100%
```

(a) Coverage aves les deux tests

```
PS C:\> coverage run .\myf.py
.
--------------------------------
Ran 1 test in 0.001s

OK
PS C:\> coverage report
Name     Stmts   Miss  Cover
--------------------------------
myf.py      11      2    82%
--------------------------------
TOTAL       11      2    82%
```

(b) Coverage avec un seul test

Figure 1: Utiliser coverage sur l'exemple

La sélection par critère de couverture ne discrimine pas sur la diversité des sorties possibles ni sur la propriété vérifiée en sortie mais sur la diversité des chemins qui peuvent être pris dans le code (voir exemple ci-dessous). Chaque "branche" correspondant à un chemin possible de sorte à ce que l'ensemble des branches recouvrent l'intégralité du code. On verra plus tard dans ce document un concept similaire à ces "branche" dans le code : le concept d'objectifs de test (Test Purpose) qui sont des chemins (ou sous-graphes) d'un modèle à automate représentant le système.

```python
def myf(e1, e2):
    if e1:
        return 0     # BRANCH_0 : (e1)
    else:
        if e2 > 5:
            return 1 # BRANCH_1 : (!e1)/\(e2>5)
        else:
            return 0 # BRANCH_2 : (!e1)/\(e2<=5)
```

```
9
10 # ====================================================================
11 import unittest
12
13 class Myf_Tests(unittest.TestCase):
14
15     def test_branch_0(self):
16         assert( myf(True, 0) != 3 )
17
18     def test_branch_1(self):
19         assert( myf(False, 6) != 3 )
20
21     def test_branch_2(self):
22         assert( myf(False, 5) != 3 )
```

On voit toutefois que ce procédé a certains problèmes :

- la "couverture" ne suffit parfois pas à trouver des bugs à l'intérieur des plages de valeurs couvertes (à moins d'avoir de la chance) : dans le premier exemple on en a un pour la valeur "0". En effet $is\_pos(0)$ retourne "None".
- utiliser le code comme modèle n'est pas toujours possible (code non disponible, systèmes en partie analogiques ...), ni désirable (certains types d'erreurs non discernables, différent niveau d'abstraction voulu, ...)

1.3 Modèles à automates

1.3.1 Définition

On se propose de faire du test basé sur les modèles avec des modèles à automate. Comparé au code qu'on a utilisé comme modèle précédemment, les modèles à automate peuvent être utilisés à différents niveaux d'abstraction et pour traduire directement des spécifications fonctionnelles. On peut aussi facilement s'en servir pour faire de la génération automatique de test en boite noire; et, sous réserve de la définition d'une relation de conformité, pour faire du test de conformité.

Il existe différents langages pour écrire des modèles à automate ((E)FSM[1], IO(L/S)TS[2] ...). Nous nous limiterons ici à l'utilisation d'un langage très simple : IOSM (Input Output State Machine). Nous choisisons ici ce formalisme, comme cela a été fait dans [2] et [3], afin d'introduire une application de la théorie des jeux au test de conformité.

Définition : IOSM

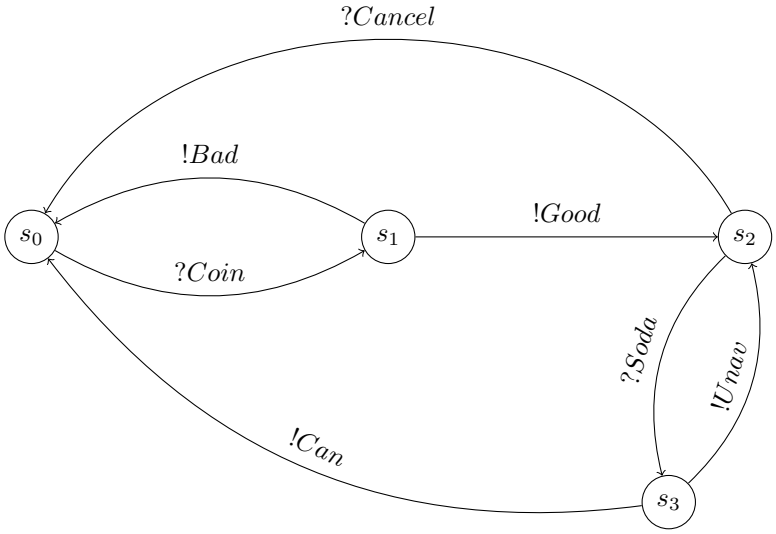
Un automate à entrées et sorties est un quadruplet (S, L, T, s_0) où :

- 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 (on considérera des chaînes de caractères alphabétiques pour les exemples)
- $T \subset (S \times ((\{?, !\}.L) \cup \{\tau\}) \times S$ 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 reception ?) ou non (transition interne τ).

Exemple

[1] (Extended) Finite State Machine
[2] Input Outut (Labelled / Symbolic) Transition Systems

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.



Dans cette modélisation, l'intégralité du comportement attendu est contenu dans le seul dessin de l'automate; on n'a pas de variable caché par exemple gardant compte du stock de boissons, etc. Le tirage des transitions est non-déterministe, et, pour chaque transition sortante d'un état, de même probabilité. Il n'y a pas de notion de temps. La notion de mémoire se résume à la connaissance de l'état "actuel" de l'automate.

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, comme on le verra dans la suite de cet enseignement d'intégration, de faire le lien avec la théorie des jeux.

1.3.2 Objectif de test

De la même manière que l'on puisse parcourir des chemins dans un code source, on peut parcourir des chemins dans un modèle à automate. De tels chemins, lorsqu'ils décrivent un test à faire, sont appelés des "objectifs de test" (Test Purpose).

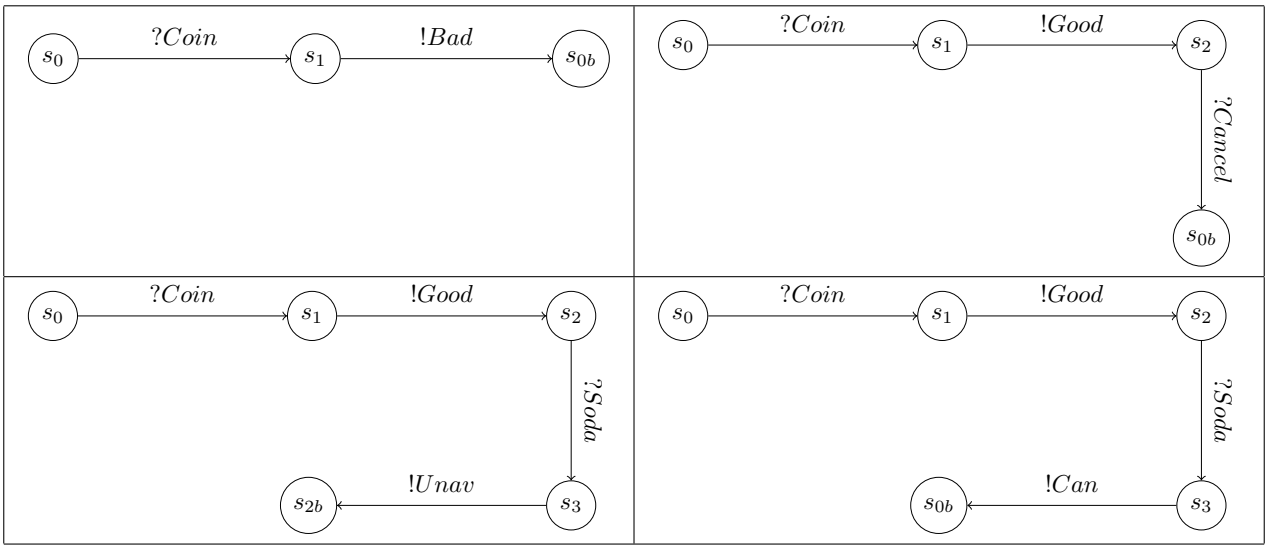
On donne la définition formelle suivante des objectifs de test (inspirée de [1]).

Définition : Test Purpose

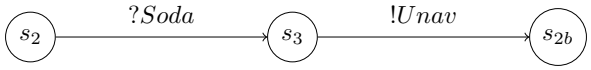
Un objectif de test TP est un IOSM acyclique

Exemple

Aussi, on peut utiliser un critère de sélection par couverture afin de s'assurer de la "couverture" de l'automate. Avec l'exemple précédent, on aurait les objectifs de test suivants qui, ensemble, couvriraient les comportements de l'automate :



Les exemples que nous avons donnés commencent tous par l'état initial s_0 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 :



1.4 Concepts de Model-Based Testing

Nous avons vu que l'utilisation de modèles peut servir à la sélection de test via par exemple la définition d'objectifs de test en fonction d'un critère de couverture. Cette pratique s'inscrit dans le cadre du test basé sur les modèles. Nous allons ici introduire quelques notions importantes liées au test basé sur les modèles.

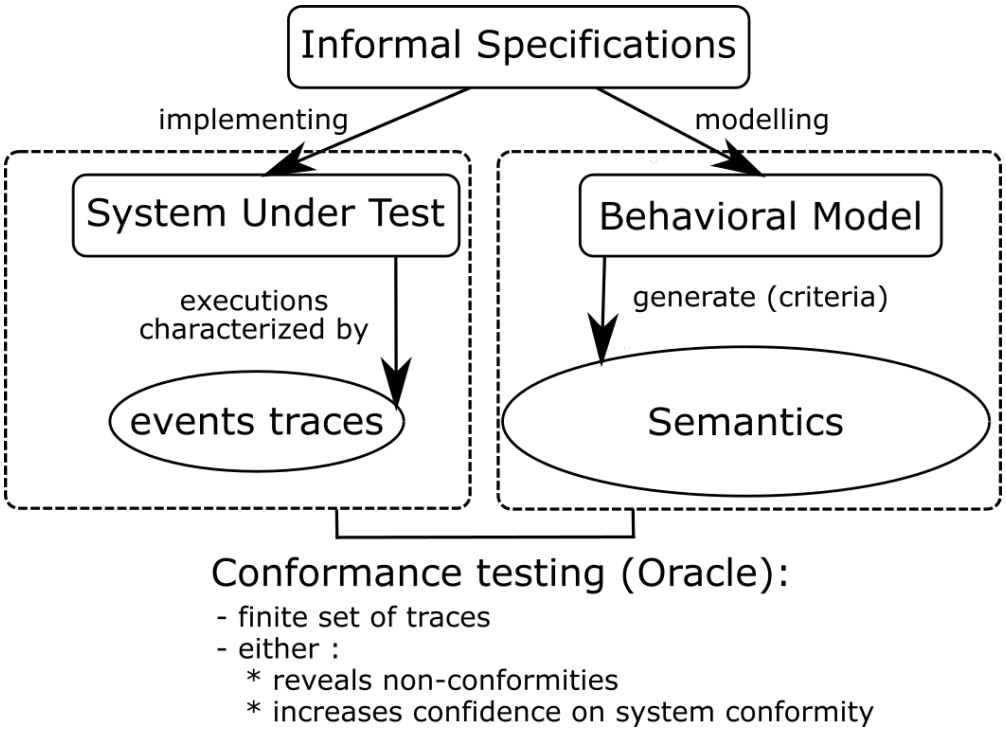
1.4.1 Test de conformité et Oracle

Un modèle utilisé en MBT est construit en utilisant un langage formel (pouvant éventuellement être associé à une représentation graphique comme les modèles à automates que nous avons vu). La théorie des langages formel est un domaine des mathématiques qui donne des résultats intéressants entre autre sur les mots constructibles à partir d'un alphabet et d'opérations données. En représentant par un tel alphabet les entrées/sorties exprimables par un système, on peut analyser les traces d'exécution du système comme un mot ayant été généré avec cet alphabet, et ainsi tirer parti des résultats de la théorie des langages formels.

Les "traces" d'exécution du système correspondent à des séquences (temporisées ou non) d'évènements pouvant être des réceptions ou émissions de données, et, dans le cas du test en boîte blanche pouvant aussi correspondre à l'évaluation de données internes.

Typiquement on se sert d'outils informatiques pour modéliser un système selon ses spécifications (ceci pouvant se faire aussi bien au préalable, en parallèle ou à posteriori du développement du système réel). A partir de ce modèle, on calculera une sémantique (celle ci pouvant être pré-calculée ou bien calculée à la volée et en fonction des besoins et des critères retenus). Un outil généralement désigné par le terme Oracle[3] va ensuite permettre de comparer des traces issues d'exécutions du système à cette sémantique, et de formuler des verdicts en conséquence.

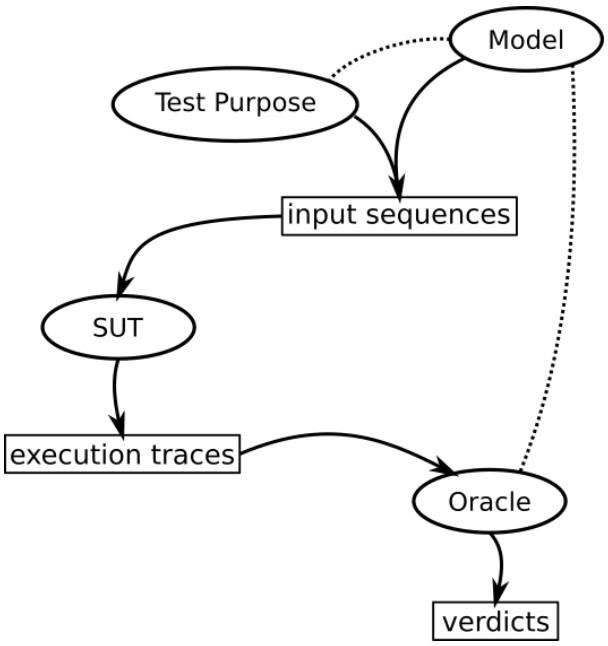
[3] William Howden, 1978



1.4.2 Test off-line vs test on-line

On peut se servir d'un modèle, d'une sémantique et d'un oracle pour analyser statiquement des traces issues d'une exécution s'étant déjà déroulée et terminée; on parle de test off-line. Ceci permet de faire une analyse à posteriori d'une exécution qui peut avoir eu lieu dans un cadre d'utilisation normale (test passif, maintenance, analyse d'erreur remontée par utilisateurs etc.), ou bien dans le cadre de batteries de tests automatiques.

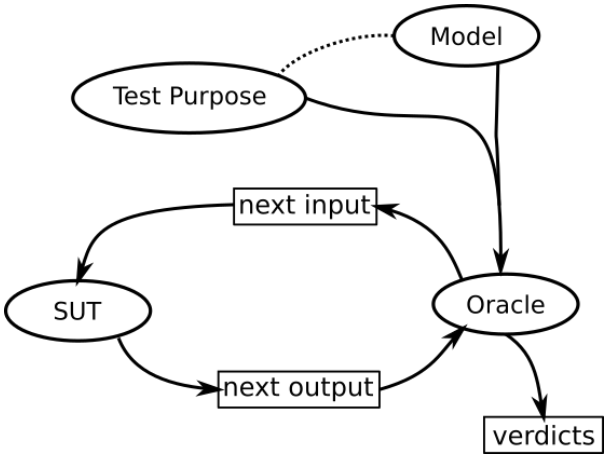
Ci-dessous, à titre indicatif, un workflow typique qui pourrait être catégorisé comme du "off-line testing".



A l'inverse, on peut se servir des outils de MBT afin de guider l'exécution du système en cours de test de sorte à ce qu'il atteigne un certain objectif (le suivi d'un scénario de test donné malgré des aléas non déterministes par exemple). La capacité d'un test à trouver des erreurs dépends en effet de 2 facteurs : les données d'entrées du

test (inputs envoyés au système) et l'oracle. Utiliser des données d'entrées pré-calculée ne permet pas toujours de rejoindre un objectif de test à cause de non déterminismes propres à certains systèmes. Il est ainsi pertinent de pouvoir calculer à la volée les entrées nécessaires à la révélation de certains chemins et éventuellement d'erreurs. Ce calcul à la volée, en se servant du modèle, est au coeur de ce qu'on appelle le test on-line.

Ci-dessous, à titre indicatif, un workflow typique qui pourrait être catégorisé comme du "on-line testing".



2 Application à la théorie des jeux : jeux adversariaux

2.1 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".

Nous donnons ci-dessous une définition formelle d'un "jeu".

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 noeud a au moins une transition sortante)

Une partie est un mot (infini) de V^ω $x = v_0 v_1 ...$ tel que $\forall i \in \mathbb{N}$ $(v_i, v_{i+1}) \in E$ et l'on a par convention $v_0 \in V_B$
Pour chaque noeud v, si $v \in V_B$ alors c'est au joeur bleu de jouer et si $v \in V_R$ c'est au joueur rouge de jouer.
On fixe un ensemble de parties dîtes gagnantes, qui le sont pour le joueur bleu. Le complémentaire de cet 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 si il gagne en la suivant (i.e. s'il gagne et que $\forall k \in \mathbb{N}$ $v_{2k} = f(v_0 ... v_{2k-1})$).

Dans la définition ci-dessus :

- le joueur bleu représente le testeur et le joueur rouge le système sous test
- il n'y a pas de puits dans le graphe (tous les nœuds ont au moins une transition sortante). Cela fait qu'on soit toujours garanti de pouvoir continuer à jouer et d'avoir un mot infini.

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 :

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

Il vient que c'est au joueur rouge de choisir les sorties (émissions $!a$) du SUT tandis que c'est au joeur bleu de choisir les entrées (reception $?a$).

Dans le cadre d'un test, le joueur rouge corresponds donc au SUT tandis que le joueur bleu corresponds au testeur.

L'état additionel $\bot$, 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 sortie du SUT est attendue. Le testeur peut déclarer non conformité si qqc d'inattendu est émis (par exemple un message non décrit par l'automate...).

Définition : Jeu associé à un IOSM et à un objectif de test

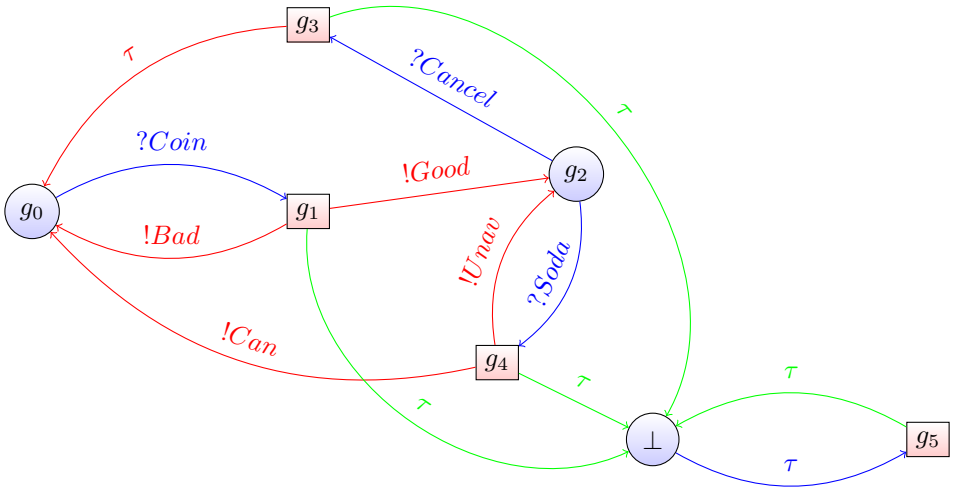
Il s'agit en fait de considérer que les parties gagnantes du jeu associé à l'IOSM sont :

$$\left\{v_0.V^*.\bot.V^\omega\right\} \cup \Big(\bigcup_{x \in Br(TP)} \{v_0.V^*.x.V^*.v_0.V^\omega\} \Big)$$

Où $Br(TP)$ représente l'ensemble des branches de TP. Un test purpose étant un IOSM acyclique, on considère comme branches tous les chemins finissant par un sommet n'ayant pas de transition sortante.

Exemple

La transformation en jeu de notre IOSM donne alors le graphe suivant. Notons que les 4 transitions en vert ont été mises de cette couleur pour des raisons de lisibilité (elles constituent l'application du 4ème point dans la définition d'un jeu associé à un IOSM) mais devraient être rouges car ce sont des transitions sortantes d'états où c'est au tour du SUT de jouer.



2.2 Atteignabilité et stratégies gagnantes

Maintenant que nous avons défini ce qu'est un jeu associé à un IOSM et à un objectif de test, et que nous connaissons quelles sont les parties gagnantes pour le testeur (qui vise soit à détecter une non-conformité, soit à parcourir une branche de l'objectif entièrement), il nous faut pouvoir mettre au point (de manière systématique) une stratégie gagnante.

Nous avons vu que les parties gagnantes sont de la forme :

$$\left\{v_0.V^*.\perp.V^\omega\right\} \cup \Big(\bigcup_{x \in Br(TP)} \{v_0.V^*.x.V^*.v_0.V^\omega\} \Big)$$

On identifie une "brique élémentaire" (c'est-à-dire un "jeu élémentaire") pour développer une stratégie gagnante : il s'agit du jeu d'atteignabilité.

Définition : Jeu d'atteignabilité pour un IOSM

Il s'agit d'un jeu associé à un IOSM dont les parties gagnantes sont de la forme :

$$v_0.V^*.t.V^\omega \quad \text{où} \quad t \in V$$

$\perp$ est un cas particulier; on considère que tout jeux d'atteignabilité est aussi un jeu d'atteignabilité sur $\perp$ c'est-à-dire que le joueur bleu (testeur) gagne toujours s'il atteint $\perp$ même si on considère un jeu sur $t \neq \perp$.

2.2.1 Construction de la suite d'ensembles emboités

Théorie

Il s'agit de trouver une partie dans laquelle on "atteint t", t étant un noeud du graphe biparti. Pour se faire, on construit une suite croissante d'ensembles de sommets W_i^j. 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, $\forall i, j$:

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

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

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

Dans la définition ci-dessus on définit 2 types de "pas" qu'on peut schématiser comme étant des "pas horizontaux" 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} :

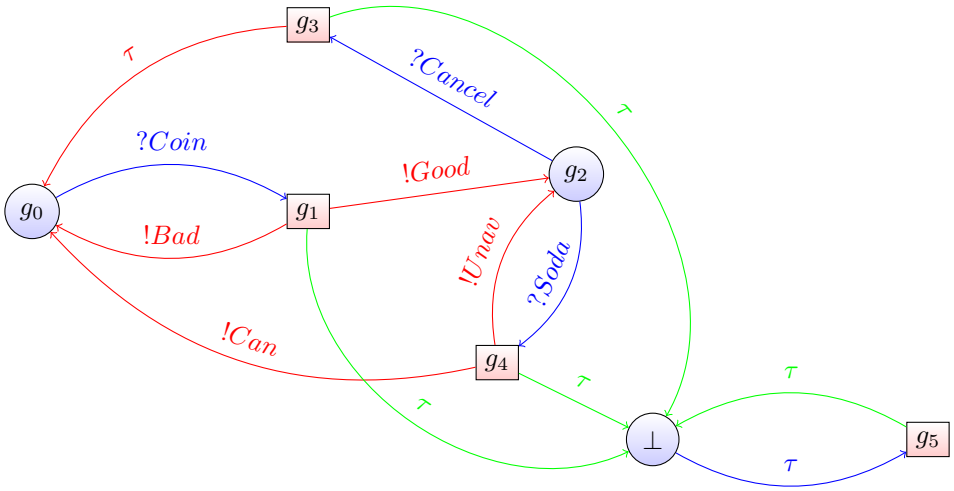
- on conserve un ensemble existant W_j^i (voir (1) dans la définition) et on l'agrandit :
- en y ajoutant tous les ancêtres bleus (contrôlés par le testeur) (ancêtres de noeuds rouges de l'ensemble précédent W_j^i)
- en y ajoutant une partie des ancêtres rouges (contrôlés par le système), ceux pour lesquels tous les descendants sont déjà dans l'ensemble précédent W_j^i

Comme on le verra après, cela permet que, dans les conditions de test, le testeur soit en mesure de forcer le système à l'intérieur de ces ensembles emboîtés, qui forment un "piège" pour ce dernier (puisqu'il est construit de sorte à ce que le joueur rouge n'y ait pas accès à des transitions permettant d'en sortir).

Dans les "pas verticaux", on calcule W_{j+1}^0 en ajoutant à $W_j^{k_j}$ des ancêtres rouges d'éléments de $W_j^{k_j}$ tels qu'ils ont au moins 1 descendant qui n'est pas dans $W_j^{k_j}$. Ceci permet de "sauter" certains blocages dans l'exploration du graphe, aboutissant à ce que tout le graphe soit couvert par ces ensembles emboités.

Application sur l'exemple

Reprenons notre exemple. Supposons qu'on s'intéresse à un jeu d'atteignabilité sur le sommet g_4.



On a alors la construction suivante :

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

2.2.2 Notion de rang et formalisation de la stratégie

Théorie

On donne la définition suivante :

Définition : Rang d'un sommet dans un jeu d'atteignabilité

Le rang d'un sommet p est :

$$rank(p) = (min\{i | p \in W_i^j\}, min\{j | p \in W_i^j\})$$

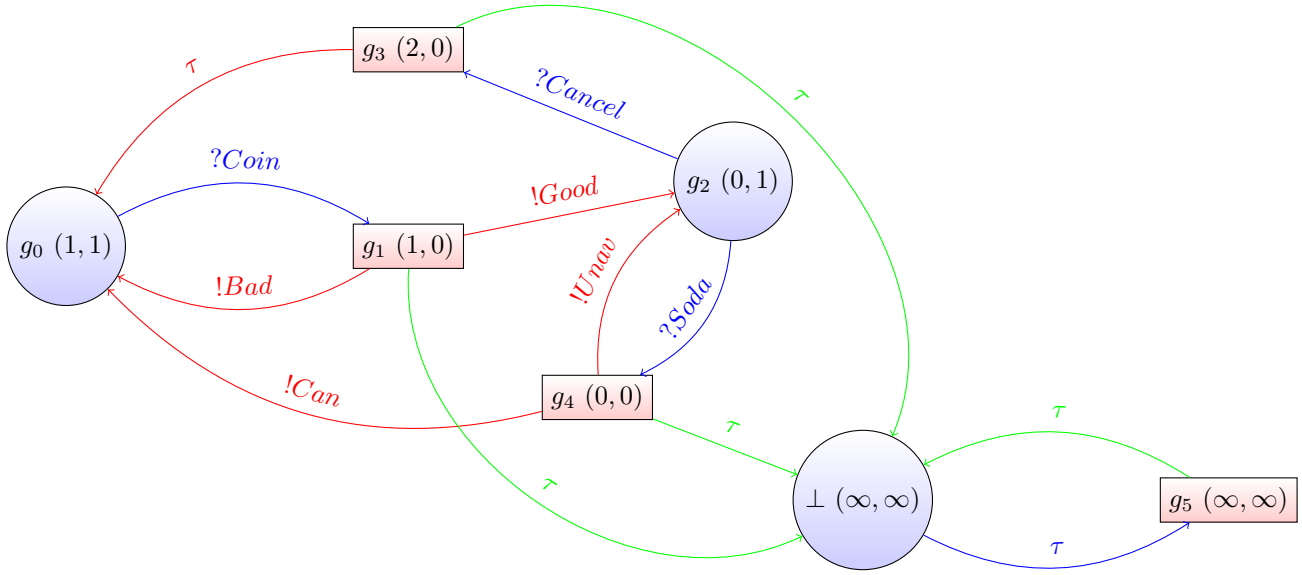
On a $rank(p) = (\infty, \infty)$ si on ne peux pas l'atteindre par construction.

La stratégie proposée pour le joueur bleu (le testeur) est alors de "faire décroitre 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 :

$$\Big(\forall (j, j'), i < i' \Rightarrow (i, j) < (i', j')\Big) \wedge \Big(\forall i, j < j' \Rightarrow (i, j) < (i, j')\Big)$$

Application sur l'exemple

On annote chaque noeud avec leur rang :



Cela se traduit par une stratégie sans mémoire (i.e. qui ne dépend que du dernier noeud visité) qui est la suivante :

$$f((V_B.V_R)^*.g_0) = g_1$$
$$f((V_B.V_R)^*.g_2) = g_4$$
$$f((V_B.V_R)^*.\perp) = g_5$$

2.2.3 Preuve que la stratégie est gagnante

Le fait que notre modèle de construction de stratégie produise des stratégies gagnantes dépends du fait que l'implémentation c'est-à-dire le joueur rouge, joue de manière équitable (il ne cherche pas à gagner). Nous en donnant la définition suivante :

Définition : Partie équitable

Soient $\pi = s_1...s_n...$ une partie, et $Inf(\pi) = \{s | s \in^\infty \pi\}$ où $\in^\infty$ signifie "apparaît infiniment".
Soit l'ensemble des couples composés d'un sommet rouge r_i et de l'ensemble des sommets bleus b_{i_k} qui lui sont adjacents :

$$\left\{(r_i, B_i = \{b_{i_1}, ..., b_{i_j}\}) \;\middle|\; \forall k \in [1, j]\ (r_i, b_{i_k}) \in E\right\}$$

Cet ensemble étant inclus dans $V_R \times \mathcal{P}(V_B)$
La partie π est équitable si $\forall i$ on a $(r_i \in Inf(\pi)) \Rightarrow (B_i \subset Inf(\pi))$

Ceci signifie que si l'implémentation a infiniment souvent la main en étant dans l'état r_i, alors tous ses successeurs sont également infiniment souvent représenté, ceci signifiant que les choix faits par l'implémentation sont en moyenne équirépartis.

La proposition à prouver est que la stratégie de "faire décroitre le rang" est gagnante pour le joueur bleu (testeur) dans un jeu d'atteignabilité à condition que le joueur rouge (implémentation) joue suivant une stratégie équitable.

Preuve : Faire décroitre le rang est une stratégie gagnante

On procède par l'absurde.
Soit π une partie gagnante pour le joueur rouge (i.e. l'implémentation). Soient $r_1, ..., r_n$ les éléments minimaux de $Inf(\pi) \cap V_R$ par rapport à leur rang suivant l'ordre lexicographique gauche. Il existe un rang i minimal tel que ces éléments appartiennent à W_0^i.
Par définition de l'équité, chacun des r_i étant connecté à au moins un sommet $b \in W_{k_i-1}^{i-1} \cap V_B$, il existe par construction un tel b dans $Inf(\pi)$.
Il y a donc un sommet rouge r (celui choisi par la stratégie du joueur bleu en b) tel que $r \in Inf(\pi)$. Ce sommet r serait alors de rang strictement inferieur à $(k_i - 1, i - 1)$. Ceci contredit la minimalité de i.

Nous avons donc pu définir un jeu et une stratégie permettant à un testeur de diriger le système vers un objectif de test se limitant à un sommet t unique. Dans la suite, nous étendrons ceci à des objectifs de tests constitués de branches entières.

2.3 Jeu de conformité

Contrairement au jeu d'atteignabilité, le jeu de conformité peut être associé à un objectif de test constitué de plusieurs sommets adjacents (constituant un sous-graphe non cyclique du graphe associé au système).

Nous avons vu qu'un tel jeu admet des parties gagnantes de la forme :

$$\left\{v_0.V^*.\bot.V^\omega\right\} \cup \left(\bigcup_{x \in Br(TP)} \{v_0.V^*.x.V^*.v_0.V^\omega\} \right)$$

Afin de définir une stratégie gagnante correspondant aux parties de la forme $v_0.V^*.x.V^*.v_0.V^\omega$, nous divisons le jeu en trois parties :

- le préambule qui correspond à un jeu d'atteignabilité pour le sommet tp_0, racine de l'objectif de test TP
- le corps, où l'on adoptera une stratégie consistant à suivre pas à pas l'objectif de test (et si le joueur rouge quitte le TP, on se remettra dans le cas du préambule afin de rejoindre de nouveau tp_0)
- le postambule, où l'on cherchera à rejoindre l'état initial $v0$ avec un jeu d'atteignabilité afin de réinitialiser le système

Nous vous invitons à lire les articles originaux afin d'approfondir le sujet : [2] et [3].

3 Application à un système réel

Un des objectifs de cet EI est que vous puissize implémenter un outillage de test basé sur la théorie des jeux comme présenté dans la section précédente. Cet outillage pourra être utilisé sur un système sous test (qui peut être un petit programme réel ou bien qui simule un système physique) de votre conception.

Afin de vous mettre dans le bain nous vous proposons (rien d'obligatoire) de tester le petit programme donné ci-dessous qui est une implémentation de l'exemple que nous avons utilisé tout au long de ce document.

```
1 import random
2
3 def simulate_non_determinism():
4     return random.choice([True, False])
5
6 class SystemSignatureViolation(Exception): pass
7
8 def can_loop(io_controller):
9     ev_name = io_controller.get_input()
```

```
    if(ev_name == "SODA"):
        g = simulate_non_determinism()
        if g:
            print("CAN")
            return False
        else:
            print("UNAV")
            return True
    elif(ev_name == "CANCEL"):
        return False
    else:
        raise SystemSignatureViolation()

def coin_loop(io_controller):
    ev_name = io_controller.get_input()
    if(ev_name == "COIN"):
        g = simulate_non_determinism()
        if g:
            print("GOOD")
            while(g):
                g=can_loop(io_controller)
        else:
            print("BAD")
    else:
        raise SystemSignatureViolation()

def start_machine(io_controller):
    while(True):
        coin_loop(io_controller)

import sys,os
import datetime

class IOController(object):
    def __init__(self):
        self.terminal = sys.stdout
        current_time = datetime.datetime.now().strftime("%Y%m%d%H%M%S")
        self.log = open("boisson_%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):
        got = input()
        if not got.endswith(os.linesep):
            self.log.write(got + os.linesep)
        else:
            self.log.write(got)
        return got

if __name__ == '__main__':
    io_controller = IOController()
    try:
        start_machine(io_controller)
    except SystemSignatureViolation:
        io_controller.write_output("ABORT")
```

References

[1] Jean Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, pages 348–359, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[2] S. Ramangalahy. *Strategies for Conformance Testing*. Forschungsberichte des Max-Planck-Instituts für Informatik. MPI Informatik, Bibliothek & Dokumentation, 1998.

[3] S. Ramangalahy, P. Le Gall, and T. Jéron. Une application de la théorie des jeux au test de conformité. In A. Schaff, editor, *Colloque Francophone sur l'Ingénierie des Protocoles, CFIP 99, Nancy, France*. Hermès, April 1999.