# Spécifications formelles

TP CCS

**Résumé**

L'objectif est d'étudier la modélisation de petits problèmes avec CCS. Cela se fera avec un outil qui détermine si deux termes sont bisimilaires. On donnera donc un modèle abstrait (vu comme une spécification de haut niveau) et un ou des modèles concrets (vu comme des implantations) et on étudiera leur bisimulation.

## 1 Rappels

### 1.1 Notion d'équivalence

Deux systèmes peuvent être *équivalents* selon plusieurs définitions (de la plus restrictive à la plus libérale) :

- Bisimulation forte (vue en cours) ;
- Bisimulation faible (vue en cours) ;
- Équivalence de trace : les systèmes de transitions correspondants ont le même ensemble de traces (= d'exécutions).

L'équivalence de trace est la plus faible : $a.(b+c).0$ et $a.b.0 + a.c.0$ sont trace-équivalents mais ne sont pas bisimilaires. L'équivalence de trace est modérément adaptée pour comparer des systèmes (comparaison de l'exécution globale, contrairement à la bisimulation qui compare pas à pas) mais ses contre-exemples sont plus faciles à comprendre.

### 1.2 Contre-exemples

#### 1.2.1 Équivalence de trace

Un contre-exemple entre deux systèmes non trace-équivalents est une *séquence de transitions*, possible dans un système, impossible dans l'autre.

#### 1.2.2 Bisimulation

Un contre-exemple entre systèmes non bisimilaires est une échelle de carrés qui échoue. L'outil fournit une représentation équivalente en logique modale de Henessys-Milner (HML), parfois peu évidente. Deux termes sont non bisimilaires s'il existe une formule vraie dans l'un et fausse dans l'autre. Une formule est de la forme :

| | |
|---|---|
| $T$ | $true$ |
| $F$ | $false$ (= impossible) |
| $[[l]]\varphi$ | Pour toute transition par l'étiquette $l$, $\varphi$ est vrai |
| $\langle\langle l\rangle\rangle\varphi$ | Pour (au moins) une transition par l'étiquette $l$, $\varphi$ est vrai |

Par exemple, les deux termes $a.b.0 + a.c.0$ et $a.(b.0 + c.0)$ ne sont pas bisimilaires. Ils se distinguent par $\langle\langle a\rangle\rangle[[b]]F$ (il existe une transition $a$ telle qu'aucune transition $b$ ne soit ensuite

possible) qui est vrai dans le premier terme et faux dans le second ; ou par $[[a]]\langle\langle b\rangle\rangle T$ (toute transition $a$ peut être suivie par une transition $b$), qui est vrai dans le second terme et faux dans le premier.

### 1.3 Événements observables vs internes

Dans le système concret, il y a deux types de transitions :

- la transition $\tau$ correspondant à un envoi/réception synchrone de message (événements internes $\bar{a}$ et $a$) ;
- les autres transitions correspondants aux événements observables.
  La description abstraite se fait sur ces événements observables.

Comme la modélisation CCS ne distingue pas les messages et les événements, il faut masquer, au niveau global, la possibilité d'une transition sur un envoi seul (ou une réception seule).

Considérons le système modélisant deux processus A et B qui s'échangent des messages en alternance. Le système émet un événement observable tick après chaque aller-retour :

| *système concret* | *système abstrait* |
|---|---|
| $Internals \triangleq \{a, b\}$ | $SA \triangleq tick.SA$ |
| $A \triangleq \bar{a}.b.tick.A$ | |
| $B \triangleq a.\bar{b}.B$ | |
| $SC \triangleq (A\|B) \setminus Internals$ | |

On peut montrer que SC et SA sont faiblement bisimilaires. SC a besoin de masquer ces messages internes, ce qui se fait en interdisant ($\setminus$) les transitions sur un unique événement $a$, $b$, $\bar{a}$ ou $\bar{b}$. Ceci correspond au masquage des canaux par $\nu$ : $\nu a\ \nu b\ (A\|B)$.

# 2 L'outil : The Edinburgh Concurrency Workbench

## 2.1 Commande

`/mnt/n7fs/tla/xccscwb`
ou `readline-editor /mnt/n7fs/tla/xccscwb`

## 2.2 Description d'un système

Un système est décrit avec (quasiment) la même syntaxe qu'utilisée en TD (cf exemples fournis). Un nom de processus doit commencer par une majuscule ; un nom d'événement commence par une minuscule, l'événement dual $\bar{a}$ s'écrit `'a`.

```
* Exemple élémentaire
agent A = a.b.0 + a.c.0;
agent B = a.C;
agent C = b.0 + c.0;
```

## 2.3 Comparaison de deux systèmes

### 2.3.1 Équivalence

`strongeq`$(S_1, S_2)$ ; bisimulation forte
`eq`$(S_1, S_2)$ ; bisimulation faible
`mayeq`$(S_1, S_2)$ ; équivalence de trace

### 2.3.2 Différence

`dfstrong`$(S_1, S_2)$ ; formule HML vraie dans $S_1$, fausse dans $S_2$, pour la bisimulation forte
`dfweak`$(S_1, S_2)$ ; idem pour la bisimulation faible
`dftrace`$(S_1, S_2)$ ; trace possible dans l'un, impossible dans l'autre

### 2.3.3 Autres commandes utiles

`help;` ou `help` *commande* ;
`input "fichier";` charge le fichier (remplacement des définitions déjà existantes)
`print;` affiche les agents définis
`obs`$(n, S)$ ; affiche toutes les séquences d'événements observables de longueur $n$
`sim S;` simulation d'un agent (évolution du terme)

### 2.3.4 Workflow

1. Décrire le système abstrait $S_a$ ;
2. Décrire le système concret $S_c$ ;
3. Vérifier s'ils sont trace-equivalents (`mayeq`). Utiliser `dftrace` pour comprendre et corriger la différence ;
4. Vérifier s'ils sont faiblement bisimilaires (`eq`). Utiliser `dfweak` pour tenter de comprendre la différence. Utiliser `obs` pour trouver la différence.

# 3 Exercices

## 3.1 Exemple : les compteurs

Il s'agit des compteurs $C_1$, $C_2$ et $C_3$ vus en TD. Intégralement fourni dans `compteurs.ccs`.

```
* Compteurs Ci vus en TD

agent C1 = S10;
agent S10 = plus.S11;
agent S11 = moins.S10;

agent C2 = S20;
agent S20 = plus.S21;
agent S21 = moins.S20 + plus.S22;
agent S22 = moins.S21;

agent C3 = S30;
agent S30 = plus.S31;
agent S31 = moins.S30 + plus.S32;
agent S32 = moins.S31 + plus.S33;
agent S33 = moins.S32;

eq(C2, C1 | C1);       * true : (weakly) bisimilar
eq(C3, C1 | C2);       * true : (weakly) bisimilar
eq(C3, C1 | C1 | C1);  * true : (weakly) bisimilar
strongeq(C3, C1 | C1 | C1);  * true : all three are also strongly bisimilar

pb(S20, S10 | S10);    * largest weak bisimulation
                       * (S20, S10|S10), (S21, S10|S11, S11|S10), (S22, S11|S11)
```

## 3.2 Communication client/serveur

On part de l'exercice vu en TD. On fournit le squelette du système (cf fichier `socket.ccs`). On souhaite enregistrer les événements de connexion et de déconnexion des client. Pour cela, on va ajouter des événements observables `login` et `logout`. Dans le cas où le serveur est unique, on doit donc avoir une alternance d'événements `login` puis `logout`, autant que de clients.

### Questions

1. Enrichir le système concret (`ServeurSimple`, `Client`...) pour avoir les événements observables `login` et `logout`.
2. Considérer le cas où il n'y a qu'un seul client (`Systeme1`). Donner un terme correspondant au système abstrait, où seuls les événements `login` et `logout` apparaissent et vérifier s'il est bisimilaire avec le système concret (faiblement `eq`, fortement `strongeq`).
3. Même question dans le cas où il y a un nombre fixe (3) de clients (`Systeme3`).
4. Même question dans le cas où il y a un nombre arbitraire de clients (`SystemeArbitraire`).

### 3.3 Protocole du bit alterné

**Description du protocole**

Un émetteur souhaite transmettre des messages à un récepteur, en utilisant un canal non fiable (perte possible du message). Pour cela, à chaque réception de message, le récepteur renvoie un acquittement (acknowledge), qui peut aussi se perdre. Quand l'émetteur n'a pas reçu l'acquittement après un délai fixé (timeout), il réémet le message (cf figure 1).

Pour que le récepteur puisse distinguer entre un nouveau message et un message réémis dont l'acquittement avait été perdu, il est nécessaire de numéroter les messages (distinction entre le message numéro $N$ et le message $N+1$).

Si le canal est FIFO, l'écart de numérotation (message attendu, message effectivement reçu) ne peut dépasser 1, et il suffit de transmettre, en plus du message, un seul bit pair/impair, qui est inversé à chaque transmission.

**Modèle abstrait**

Le modèle abstrait est (où les événements `irecu/precu` correspondent à la réception d'un message impair/pair) :

```
set Observables = { irecu, precu };  * les événements observés
agent AbstractSystem = irecu.precu.AbstractSystem;
```

**Modèles concrets à faire**

1. Pas de réseau, le système est constitué uniquement de l'émetteur et du récepteur qui se transmettent directement les messages :

   ```
   set Internals = { i, p, ack };
   agent System1 = (Sender | Receiver) \ Internals;
   ```

   Compléter ce modèle de manière qu'il soit bisimilaire avec le système abstrait.

2. Réseau fiable à une case : l'émetteur donne le message au réseau, et le réseau donne le message au récepteur. Il faut introduire de nouveaux messages internes pour les interactions entre émetteur et réseau, et entre réseau et récepteur.

   ```
   set Internals = { sendi, deliveri, sendp, deliverp, sendack, deliverack };
   agent System2 = (Sender | Receiver | Network ) \ Internals;
   ```

3. Réseau avec perte : la perte est non déterministe. Le réseau émet un événement `timeout` en cas de perte d'un message applicatif ou d'un acquittement. Sur timeout, l'émetteur doit réémettre le même message que précédemment.

   ```
   set Internals = { sendi, deliveri, sendp, deliverp, sendack, deliverack, timeout };
   agent System3 = (Sender | Receiver | Network ) \ Internals;
   ```

**Indication**

Il est vivement conseillé de faire l'automate de chacun des composants (émetteur, récepteur, réseau) pour comprendre leur évolution.

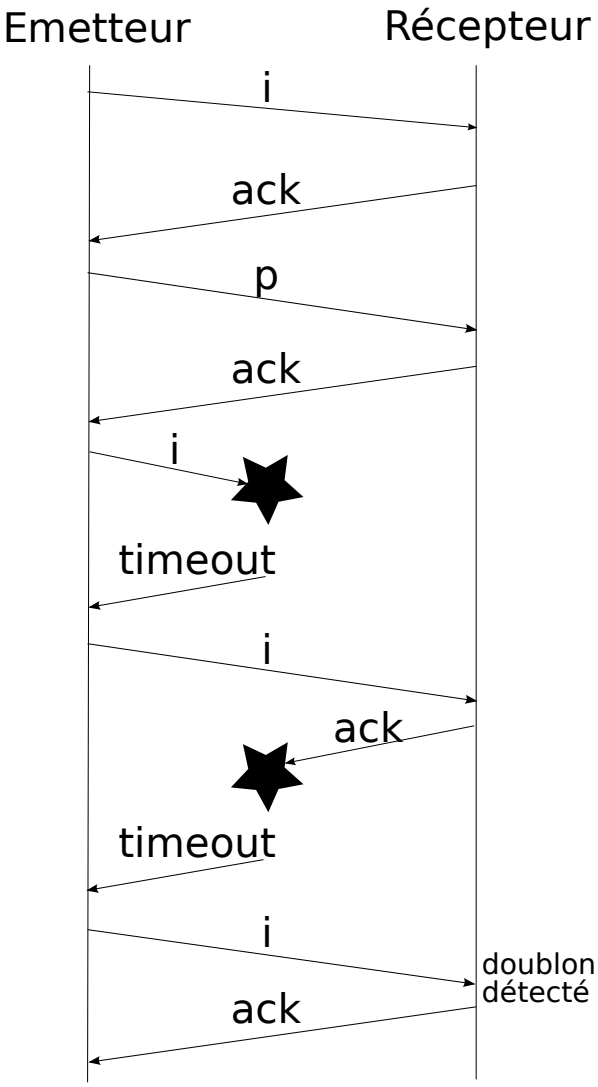


FIGURE 1 – Exemple d'exécution du protocole du bit alterné