Master 1 d'Informatique. ***Langages de spécification de logiciel***

Exercice : Spécification d'un simple point d'accès en termes d'ASM et de SDL.

Définition du problème. Un point d'accès consiste en 2 processus : *Controller* (contrôleur) et *Door* (porte). Le contrôleur a 2 canaux pour communiquer avec l'environnement : l'un, dénoté *chNewCode* (*ch* provient de l'anglais *channel* qui signifie "canal"), sert pour que l'administrateur du point d'accès puisse entrer un code d'accès, and l'autre, dénoté *chEnvCntr*, sert pour qu'un utilisateur puisse ouvrir la porte s'il tape le bon code. Un code d'ouverture de porte est un entier plus grand de 1000. Le signal pour définir le code est *newCode* avec les valeurs du type entier (ou $undef$ qui est une constante spéciale pour désigner l'absence de signal). Cette valeur est gardée par le contrôleur comme la valeur de la fonction (variable) *codeDoor*, initialement égale à 0.

Avoir reçu un signal $code \neq undef$ le contrôleur retourne à l'environnement le signal $outOfService$ si $codeDoor \leq 1000$, le signal ERR si $code \neq codeDoor > 1000$ et le signal OK si $code = codeDoor > 1000$. Dans ce dernier cas le contrôleur envoie à *Door* le signal *unlock* via le canal *chCntrDoor*. Avoir reçu ce signal, la porte passe à l'état *doorUnlocked* de son état initial *doorLocked* . En même temps *Door* envoie à l'environnement le signal *unlocked*. Après l'état *doorUnlocked* la porte doit revenir à son état initial. Cela peut être déclenché soit par *timer* qui attend $\delta > 0$ temps soit par un signal en provenance de l'extérieur. Nous choisissons la deuxième possibilité. *Door* envoie un signal *step* au contrôleur lorsqu'elle passe à l'état *doorUnlocked*, et le contrôleur retourne à *Door* le signal *stepDoor* pour que *Door* revienne à son état initial.

Tâche. Spécifiez le système en termes d'ASM et en termes de SDL. Essayez d'assurer une correspondance proche entre ces deux spécifications.

Solution. Pour commencer on fait un diagramme d'UML pour mieux comprendre le fonctionnement et l'architecture du système en gros, par exemple voir Fig. 1 (c'est un diagramme de communication). Ce diagramme ne représente pas tous les signaux, mais il donne une vision suffisante pour avancer.

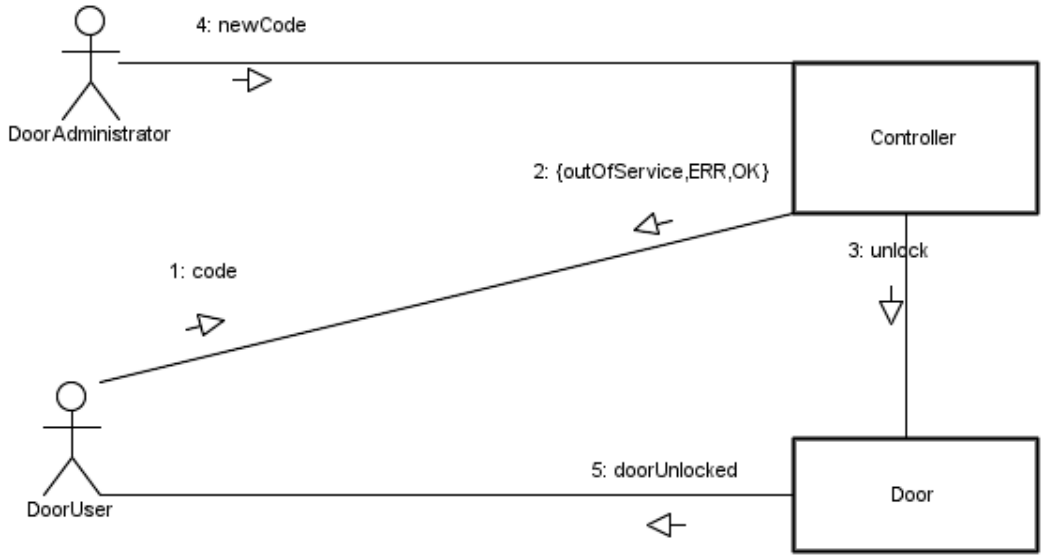

Fig. 1 – Point d'accès : diagramme de communication initial.

Ensuite, nous développons parallèlement une ASM et des diagrammes SDL. Les diagrammes

SDL sont commentés sur la page de cet exercice où il y a des liens pour les trouver en formats différents. On peut vite percevoir qu'il est plus facile d'avancer les diagrammes SDL.

Dans ce texte nous développons une ASM. Commençons avec les canaux compte tenu que nous voulons être proche des diagrammes SDL. On peut représenter dans notre ASM tous les canaux ou se limiter aux signaux. Ou on peut choisir une voie intermédiaire. Modélisons 4 canaux de 6 (le canal bi-directionnel SDL entre le contrôleur et la porte, est compté comme 2 canaux) pour voir la différence. La spécification nous indique les canaux suivants (la spécification de dit rien des canaux entre *Door* et *Controller* ; cette communication a besoin de 2 canaux de plus mentionnés ci-dessus) :

1. *chNewCode* un canal de l'environnement (de l'administrateur de porte) au contrôleur pour communiquer un nouveau code de la porte,
2. *chEnvCntr* un canal de l'environnement (de l'utilisateur) au contrôleur pour que l'utilisateur puisse ouvrir la porte,
3. *chCntrEnv* un canal du contrôleur à l'environnement (à l'utilisateur) qui retourne à l'utilisateur la réaction du contrôleur au code entré par l'utilisateur,
4. *chDoorEnv* un canal de la porte à l'environnement qui affiche que la porte est ouverte.

Les 4 canaux mentionnés sont uni-directionnels. Dans une vraie implémentation les canaux *chEnvCntr* et *chCntrEnv* seront probablement fusionnés en un canal bi-directionnel.

Vocabulaire.

Sortes : Les sortes et leur constantes sont définies ci-dessous dans les types de fonctions.

Fonctions :

- $chNewCode :\rightarrow \{signalNewCode, undef\} := undef$
  /* Fonction externe de *Controller*. La notation $:= undef$ donne la valeur initiale, idem ci-dessous. Si la valeur est $undef$ alors il n'y a pas de signal, sinon il y a un signal. Cependant la valeur *signalNewCode* n'a pas de signification spécifique, c'est un symbole différent de $undef$. */
- $newCode :\rightarrow \mathbb{Z} := 1$
  /* Voir le commentaire qui concerne *code* ci-dessous. Fonction externe de *Controller*. */
- $codeDoor :\rightarrow \mathbb{Z} := 0$ /* Fonction interne de *Controller*. */
- $chEnvCntr :\rightarrow \{signalCode, undef\} := undef$ /* Fonction externe de *Controller*. */
- $code :\rightarrow \mathbb{Z} := 0$
  /* Pour diminuer le nombre de drapeaux et pour faire *code* et *newCode* du type entier nous initialisons ces deux fonctions avec deux entiers différents. Fonction externe de *Controller*. */
- $chCntrEnv :\rightarrow \{signalCntrEnv, undef\} := undef$ /* Fonction interne de *Controller*. */
- $signalCntrEnvVal :\rightarrow \{outOfService, OK, ERR\}$
  /* Fonction interne de *Controller*. La valeur initiale de cette fonction n'est pas importante car cette fonction ne utiliser que dans les updates. */
- $chDoorEnv :\rightarrow \{signalDoorEnv, undef\} := undef$ /* Fonction interne de *Door*. */
- $signalDoorEnvVal :\rightarrow \{unlocked, undef\} := undef$ /* Fonction interne de *Door*. */
- $signalDoorCntr :\rightarrow \{step, undef\} := undef$
  /* Fonction interne de *Door*, externe pour *Controller*. */
- $signalCntrDoor :\rightarrow \{unlock, stepDoor, undef\} := undef$
  /* Fonction interne de *Controller*, externe pour *Door*. */
- $stateDoor :\rightarrow \{doorLocked, doorUnlocked\} := doorLocked$ /* Fonction interne de *Door*. */
- $stateCntr :\rightarrow \{init, stepDoor\} := init$ /* Fonction interne de *Controller*. */
- $branch :\rightarrow \{0, 1\}$ /* Fonction interne de *Controller*. La valeur initiale de cette fonction ne joue aucun rôle car avant chaque utilisation elle sera définie par **choose**. */

Nous définissons notre ASM $AccessP1$ d'une façon modulaire. Elle est une ASM distribuée synchrone (en fait, on peut la traiter comme asynchrone, mais dans ce cas on perd la correspondance exacte avec notre diagramme SDL). L'ASM $AccessP1$ consiste en 2 agents :

$$AccessP1 = (Controller, Door)$$

Le premier agent représente le contrôleur. Pour éviter le traitement simultané de 2 signaux *newCode* et *code* nous introduisons un choix non-déterministe à l'aide de la fonction *branch*. Cette fonction a 2 valeurs qui correspondent à 2 branches : l'une qui traite *newCode* (la valeur 0) et l'autre qui traite *code* (la valeur 1). Ce choix non-déterministe concerne seulement le cas où les deux signaux arrivent en même temps. L'activité du contrôleur dans l'état *init* est représentée par 2 règles (sous-programmes) $branchNesCode$ et $branchCode$ qui décrivent respectivement le traitement du signaux *newCode* et *code*. Les diagrammes SDL donnent une vision graphique de nos programmes ASM modulo quelques détails secondaires. Pour faciliter la lecture des programmes nous utilisons des notations redondantes, par exemple, pour la composition séquentielle nous écrivons $(R_1 \,\mathbf{seq}\, R_2 \,\mathbf{seq}\, \ldots \,\mathbf{seq}\, R_n)$ ou nous ajoutons **endif** pour 'fermer' **if** lointain.

main $Controller =$

1 : **choose** $branch$ **in** $\{0, 1\}$ **do**
2 : (**if** $stateCntr = init$
3 : **then**
4 : **if** $branch = 0$ **then** $branchNewCode()$
5 : **seq**
6 : $branchCode()$
7 : **seq**
8 : **if** $branch = 1$ **then** $branchNewCode()$
)
9 : **if** $stateCntr = stateDoor \wedge signalDoorCntr = step$
10 : **then**
11 : ($signalCntrDoor := stepDoor$
12 : **seq**
13 : [$stateCntr := init$ **par** $signalCntrDoor := undef$]
)

$branchNewCode() =$
if $chNewCode \neq undef \wedge codeDoor \neq newCode$
then $codeDoor := newCode$
/* La condition $codeDoor \neq newCode$ dans la garde joue en rôle de drapeau pour éviter l'exécution de cet opérateur en continue durant l'existence du signal *newCode* dans le canal $chNewCode$*/

```
  branchCode() =
1 :  if code ≠ undef ∧ y = 0 ∧ code > 0
2 :   then
       (
3 :     y := code
        seq
4 :     if codeDoor ≤ 1000
5 :      then
6 :       (chCntrEnv := signalCntrEnv
7 :        seq
8 :        [signalCntrEnvVal := outOfService par y := 0]
9 :      else openDoor()
10 :        )
       )
```

/*Si $codeDoor \leq 1000$ le programme remet y à sa valeur initiale 0 ce qui n'est pas explicité dans la définition du problème. */

```
  openDoor() =
1 :  if y = codeDoor
2 :   then
       [
3 :     (chCntrEnv := signalCntrEnv
4 :      seq
5 :      signalCntrEnvVal := OK)
6 :     par
7 :      signalCntrDoor := unlock
       ]
8 :   else
9 :    (chCntrEnv := signalCntrEnv
10 :    seq
11 :     signalCntrEnvVal := ERR)
12 :  endif
13 :  seq
14 :  [stateCntr := init par y := 0]
```

/* Pour la lisibilité nous avons ajouté **endif** pour indiquer la fin du premier **if**.*/

```
  main Door =
1 :  if stateDoor = doorLocked ∧ signalCntrDoor = unlock
2 :  then
3 :    signalDoorCntr := step
4 :    seq
5 :    chDoorEnv := signalDoorEnv
6 :    seq
7 :    signalDoorEnvVal := doorUnlocked
8 :    seq
9 :    stateDoor = doorUnlocked
10 :  endif
11 :  par
12 :  if stateDoor = doorUnlocked ∧ signalCntrDoor = stepDoor
13 :  then stateDoor := doorLocked
```