Projet VERBATIM

Sous-projet 2 – Lot 1

Test d'applications multimodales à l'aide de l'approche synchrone :
Fournitures :
« Expression de propriétés ergonomiques et fonctionnelles dans le formalisme synchrone »
et
« Guide de conception de propriétés de services multimodaux »

Version : 1.2

Date : septembre 2005

Laya Madani, Catherine Oriat, Ioannis Parissis

Laboratoire LSR-IMAG

BP 53 38041 Grenoble Cedex 9

www-lsr.imag.fr

LSR

Table des matières

1. Introduction

L'objectif du sous projet est d'étudier l'application de techniques de test de logiciels réactifs synchrones développées au laboratoire LSR-IMAG à la validation de services multimodaux. L'approche synchrone permet une modélisation plus simple des comportements et offre par ailleurs la possibilité de procéder à des simulations et tests automatiques. Nous nous appuyons sur le constat suivant : un service interactif multimodal peut, sous certaines conditions, être considéré comme un système réactif synchrone. En effet, s'il est vrai que la rapidité de réaction de systèmes interactifs à un événement externe est moins cruciale que dans le cas des systèmes traditionnellement qualifiés de synchrones, on peut toutefois constater que leurs comportements sont constitués de cycles "action-réaction". Ainsi, des interactions multimodales correspondant à des comportements des utilisateurs du service ainsi que certaines propriétés du service et de son contexte pourraient s'exprimer dans un formalisme synchrone à des fins de tests.

Ce document s'inscrit dans le premier objectif du sous-projet, à savoir la définition d'un formalisme inspiré du langage Lustre permettant de spécifier les comportements de l'utilisateur et du contexte ainsi que des propriétés du service à vérifier à l'aide de l'environnement Lutess. Il aborde plus particulièrement les points suivants :

- Définition du niveau d'abstraction de la spécification. Ce niveau va déterminer, en particulier, le type de données de test qui devraient être générées.
- Etude de l'adéquation du langage Lustre pour l'expression des propriétés à tester.
- Etude de l'adéquation des moyens de guidage fournis par Lutess pour l'expression de scénarios de test pertinents.

La structure de ce document reflète la démarche de travail que nous avons adoptée : après une brève présentation de l'approche de test proposée par l'environnement Lutess, nous présentons une étude de cas qui nous a guidé dans notre réflexion. La dernière partie du document revient sur les points les plus importants de cette étude en proposant des solutions pour chacun des points mentionnés ci-dessus.

2. Test de logiciels synchrones

2.1 Lutess : un environnement de test de programmes synchrones

Lutess [6] est un outil de test de programmes synchrones. Ces derniers vérifient l'hypothèse de synchronisme qui stipule que le calcul des sorties du programme à partir de ses entrées est instantané. En supposant que le temps est divisé en des instants discrets définis par une horloge globale, un programme synchrone, à un instant t, lit ses entrées i_t et calcule ses sorties o_t. L'hypothèse synchrone assure que le calcul de o_t est fait instantanément, au même instant.

Lutess permet la génération automatique de séquences d'entrée respectant les contraintes d'environnement du programme sous test. Ces contraintes correspondent à des hypothèses sur les comportements possibles de l'environnement. Lutess construit automatiquement un générateur de données de test et un harnais de test. Ce dernier lie le générateur, le programme sous test et l'oracle, coordonne leur exécution et enregistre les séquences d'entrée et de sortie et les verdicts associés d'oracle. Lutess suppose que le programme a un comportement synchrone et exige que les contraintes d'environnement soient écrites en langage Lustre.

Lustre [8] est un langage destiné à la spécification et la programmation de systèmes réactifs synchrones. Un programme Lustre est structuré en noeuds. Considérons le programme Lustre suivant :

```
node Never (A : bool) returns (never_A : bool);
   let
         never_A = not A -> (not A and pre (never_A));
   tel
```

Ce programme reçoit une entrée booléenne et a une unique sortie booléenne. A chaque instant, la sortie est vraie si seulement si l'entrée n'a jamais été vraie depuis le début de l'exécution de programme. Par exemple, le programme produit la séquence de sortie (true, true, true, false, false) en réponse à la séquence d'entrée (false, false, false, true, false).

Un nœud Lustre est constitué d'un ensemble d'équations qui définissent ses sorties comme des fonctions des entrées et des variables locales. Une expression Lustre contient des constantes, des variables, des opérateurs logiques, arithmétiques et des opérateurs spécifiques Lustre. Il y a deux opérateurs temporels Lustre: "pre" et "->". "pre" donne accès à la dernière valeur qu'une expression vient de prendre (au top d'horloge précédent). L'opérateur "->", « suivi par » , est utilisé pour désigner la valeur initiale (à t = 0) des expressions :

- Si E est une expression dénotant la séquence (e0, e1, ..., en, ...), **pre** E va dénoter la séquence (nil, e0, e1, ..., en-1, ...) où nil est une valeur indéfinie. En d'autres termes, pre E retourne, à un instant t, la valeur de l'expression E dans le moment t-1.
- Si E et F sont des expressions dénotant, respectivement, les séquences (e0, e1, e2, ..., en, ...) et (f0, f1, f2, ..., fn, ...), E -> F va dénoter la séquence (e0, f1, f2, ..., fn, ...).

Des opérateurs temporels exprimant des invariants ou propriétés peuvent être exprimés en Lustre. Par exemple, Once A From B To C spécifie que la propriété A doit être vraie au moins une fois entre les instants où B et C sont vrais.

Comme le montre la Figure 2-1, Lutess requiert, outre le logiciel exécutable sous test, la description de l'environnement et un oracle de test (un programme exécutable). Un générateur de données de test est automatiquement construit par Lutess à partir de la description de l'environnent écrite en Lustre.

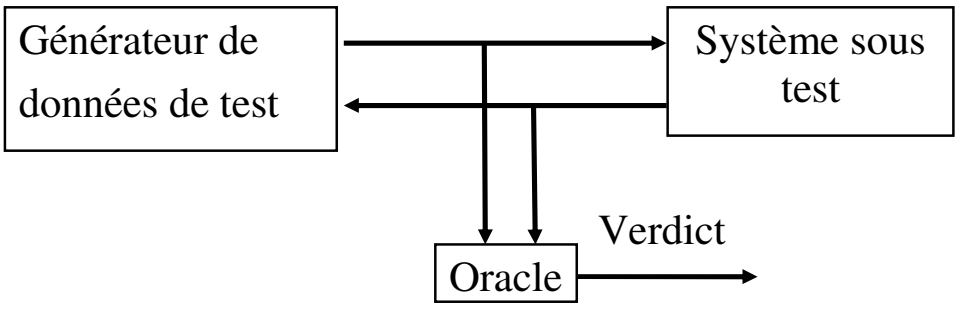

Figure 2-1: l'environnement Lutess

Le test est effectué par cycles successifs action-réaction. Le générateur engendre aléatoirement un vecteur d'entrée et l'envoie au logiciel, qui réagit en émettant un vecteur de sortie vers le générateur. Le générateur produit un nouveau vecteur d'entrée et le cycle est répété. L'oracle observe les entrées et les sorties du programme et détermine si les propriétés du logiciel sont violées.

A chaque cycle (ou pas), le générateur de Lutess choisit un vecteur d'entrée en supposant que la distribution de données est équiprobable. Cependant, l'utilisateur de Lutess peut aussi définir une distribution statique des entrées en associant des probabilités d'occurrence aux événements d'entrée. Il peut également spécifier des scénarios pour guider la génération des données de test.

3. Test synchrone d'applications multimodales

3.1 Exemple : spécification et validation d'une application multimodale avec l'approche synchrone

3.1.1 L'application Mémo

Memo [3] est un système multimodal qui permet d'annoter des localisations physiques avec des « post-it » digitaux. Les post-it peuvent être ensuite lus/portés/supprimés par d'autres utilisateurs mobiles. L'utilisateur de Memo de la Figure. 3-1 (gauche) est équipé d'un casque. Sa semi-transparence permet la fusion de données (les notes digitales) avec l'environnement réel comme dans la Figure. 3-1 (à droite). De plus, un GPS et un magnétomètre sont portés par l'utilisateur et permettent au système de calculer la localisation et l'orientation de ce dernier.

Dans Memo, trois tâches sont possibles en utilisant des modalités différentes :

1. changer d'orientation et de localisation de l'utilisateur mobile ; le système peut donc afficher sur le casque les notes visibles conformément à sa position courante et à son orientation,
2. manipuler (récupérer, placer ou supprimer) une note et
3. quitter le système.

Prenons la tâche 'manipulation d'une note': l'utilisateur mobile peut récupérer une note. Cette note est portée pendant le déplacement de l'utilisateur. Ce dernier ne peut porter qu'une seule note à la fois. Il peut par contre placer une note portée. L'utilisateur peut également supprimer une note portée ou visible dans son environnement physique. Si l'utilisateur porte une note et voit aussi une autre note, la commande "remove" supprime la note dans le monde physique (la priorité est donnée à la manipulation de notes au monde physique). Si l'utilisateur porte une note et n'en voit pas d'autre, alors la commande "remove" a pour effet la suppression de la note portée.

Dans Memo, il y a cinq modalités actives et passives pour les entrées. Les modalités actives sont utilisées pour passer une commande vers l'ordinateur (e.g. une commande vocale). En considérant une modalité [10] comme un couple d'un dispositif physique et un langage, les trois modalités actives dans Memo sont: (Souris, Commandes de Bouton), (Microphone, Commandes Vocales) et (Clavier, Commandes de Clavier). Ces modalités sont utilisées par l'utilisateur pour manipuler une note et pour quitter le système. Les modalités passives sont utilisées pour calculer l'information qui n'est pas explicitement exprimée par l'utilisateur à l'ordinateur, comme le suivi de l'oeil dans la manifestation « mets ça là » (Put that there) [1] ou la localisation de l'utilisateur mobile de Memo. Les deux modalités passives dans Memo sont: (Magnétomètre, les trois angles d'orientation en radians) et (capteur de Localisation GPS, 3D localisation). Ces modalités passives sont utilisées pour afficher les notes sur le casque semi-transparent, et pour sélectionner une note.


Figure. 3-1: ***Gauche : un utilisateur de Memo, équipé d'un casque semi-transparent, et qui tient une souris. Le clavier n'était pas utilisé dans cette version.***

Droite: une vue à travers le casque semi-transparent. L'utilisateur mobile est devant le bâtiment d'informatique de l'université de Grenoble et peut voir deux notes digitales.

3.1.2 Besoins de spécification

Afin de tester Memo dans l'environnement Lutess, nous devons fournir les éléments suivants :

- Le système sous test (Memo) comme un programme exécutable. Un traducteur d'événements doit être ajouté pour traduire les événements d'entrée et de sortie de l'application interactive en événements booléens manipulés par Lutess. Le paragraphe 3.1.3 présente une implémentation permettant de connecter Lutess avec Memo.
- Un oracle contenant les propriétés que l'application Memo doit vérifier. Dans le paragraphe 3.1.4 nous présentons un oracle exprimé en Lustre.
- La spécification Lustre de l'environnement externe. Cette spécification correspond au comportement de l'utilisateur. Le paragraphe 3.1.4 est consacré à cette spécification pour le cas de Memo.

Avec les éléments ci-dessus, Lutess peut tester de manière aléatoire Memo. Des comportements plus spécifiques peuvent également être spécifiés à l'aide de plusieurs directives de guidage qui peuvent être considérées comme un complément de spécification de l'environnement. Le paragraphe 3.1.6 se concentre sur deux stratégies de guidage: les profils opérationnels (probabilités affectées aux entrées) et les schémas comportementaux (scénarios).

Nous présentons également (paragraphe 3.1.7) des extraits de résultats produits par cette expérimentation.

3.1.3 Connexion de Memo à Lutess

La classe Java MemoLutess est l'interface entre Memo et Lutess. Elle traduit les sorties de Lutess en entrées de Memo et inversement. La multimodalité étant une caractéristique forte de l'application, on impose à Lutess de générer des événements correspondant à chacune des modalités. Les entrées générées par Lutess et reçues par les composants de Memo sont les suivantes :

1. *Localisation* est un vecteur booléen indiquant le mouvement de l'utilisateur selon les directions des *x*, *y* et *z*. A un instant, Localisation[plus]= true signifie que l'abscisse de l'utilisateur augmente. Pour le cas de Memo, nous avons fixé (empiriquement) la valeur de l'augmentation ou de la diminution.
2. *Orientation* est un vecteur booléen indiquant les changements d'orientation de l'utilisateur avec les trois angles d'orientation : yaw, pitch et roll. Orientation[pitchplus]= true signifie que l'utilisateur baisse la tête d'un angle fixe prédéfini de manière empirique..
3. *Mouse, Keyboard* et *Speech* sont des vecteurs booléens correspondant aux commandes get, set ou remove spécifiées en utilisant la souris, le clavier ou la parole. A un instant,

Mouse[get]= true indique que l'utilisateur vient de cliquer sur le bouton de la souris correspondant à la commande get.

L'état de l'application Memo et les événements de sortie sont observés par les cinq sorties booléennes suivantes :

1. *memoSeen*, qui est vrai quand au moins une note est visible.
2. *memoCarried*, qui est vrai quand l'utilisateur porte une note.
3. *memoTaken,* qui est vrai si l'utilisateur a pris une note au cycle précédent.
4. *memoSet,* qui est vrai si l'utilisateur a posé une note au cycle précédent.
5. *memoRemoved,* qui est vrai si l'utilisateur a supprimé une note au cycle précédent.

La classe MemoLutess contient un constructeur qui crée une nouvelle instance de l'application Memo. La méthode principale (main) crée une nouvelle instance de MemoLutess et lie cette instance avec Lutess.

```
/* Main method */
static public main(String[] args) {
   MemoLutess m = new MemoLutess();
   m.connectLutess(); }
```

La méthode connectLutess se compose d'une boucle infinie. La boucle, dans chaque étape (1) lit un vecteur de booléens généré par Lutess, (2) envoie les événements correspondants à Memo, (3) attend que Memo exécute la commande, (4) récupère l'état courant et les événements de sortie de Memo, (5) et finalement écrit les booléens correspondant à l'état et aux sorties de Memo qui serons lus par Lutess afin de générer un nouveau vecteur de booléen pour le cycle d'exécution suivant.

```
/* Main interaction loop */
void connectLutess() {
  while (true) {
    readInputs();             // Read the sequence produced by Lutess
    memoApp.sendEvents() ;    // Send corresponding events to Memo
    wait(500);                // wait for Memo to execute the commands
    memoApp.getState() ;      // Get the new state of Memo
    writeOutputs();}}         // Write the obtained sequence
```

Finalement, l'orientation et la localisation de l'utilisateur sont récupérées avec des modalités passives : les données correspondant à l'orientation et à la localisation de l'utilisateur doivent être continuellement envoyées à Memo. C'est pourquoi on utilise un "thread" Java pour l'orientation :

```
  class OrientationThread extends Thread {
 public void run() {
   while (true) {
     memoApp.sendOrientation(); // Send the orientation to Memo
     wait(5); }}}
```

3.1.4 Spécification de l'environnement et du comportement de l'utilisateur

Lutess génère les entrées pour le système sous test en respectant des contraintes définissant des hypothèses sur l'environnement externe du système sous test, c'est-à-dire, pour le cas de Memo, le comportement de l'utilisateur. Par exemple, l'utilisateur ne peut pas bouger sur un

axe dans les deux directions en même temps. Les formules correspondantes sont les suivantes :

```
not (Localization[xminus] and Localization[xplus])
not (Localization[yminus] and Localization[yplus])
not (Localization[zminus] and Localization[zplus])
```

De même, l'utilisateur ne peut pas tourner autour d'un axe dans les deux directions à la fois :

```
not (Orientation[yawminus] and Orientation[yawplus])
not (Orientation[pitchminus] and Orientation[pitchplus])
not (Orientation[rollminus] and Orientation[rollplus])
```

En ce qui concerne les commandes utilisateur, étant donné que Lutess envoie les entrées à Memo au niveau de la modalité, une donnée par modalité peut être uniquement envoyée à un instant donné. On a trois commandes (get, set, remove) associées à la souris, le clavier ou la parole. On a donc les formules suivantes :

```
AtMostOne(3,Mouse)
AtMostOne(3,Keyboard)
AtMostOne(3,Speech).
```

3.1.5 Spécification de propriétés fonctionnelles

Nous spécifions ici les propriétés souhaitées de l'application qui se retrouvent au sein de l'oracle de test.

D'abord, on désire valider que les notes sont récupérées, placées ou supprimées uniquement par des commandes adéquates:

- Après avoir vu une note et avant de la récupérer, l'utilisateur doit faire une commande "get" à un instant où une note est vue (plus précisément, à un instant où une note est suffisamment proche de l'utilisateur pour être manipulée).

```
once_from_to(cmdget and pre memoSeen, memoSeen, memoTaken)
```

- Entre l'instant où l'utilisateur voit ou porte une note et l'instant où une note est supprimée, l'utilisateur doit faire une commande "remove" .

```
once_from_to(cmdremove and (pre memoSeen or pre memoCarried),
             memoSeen or memoCarried, memoRemoved)
```

On souhaite également valider que l'état du système Memo ne peut changer que suite à l'arrivée d'événements d'entrée adéquats :

- Entre l'instant où l'utilisateur voit une note et l'instant où il ne voit plus de note, l'utilisateur a bougé ou il a efffectué une commande "get" or "remove".

```
once_from_to((move or cmdget or cmdremove) and pre memoSeen,
             memoSeen, not memoSeen)
```

- Entre l'instant où aucune note n'est visible et l'instant où une note est visible, l'utilisateur a bougé ou a il a executé une commande "set".

```
once_from_to(move or (cmdset and pre memoCarried),
             not memoSeen, memoSeen)
```

- Si l'utilisateur porte une note, alors une commande "get" est arrivée auparavant.

```
once_from_to(cmdget and pre memoSeen, not memoCarried, memoCarried)
```

– Seules les commandes "set" ou "remove" peuvent faire que l'utilisateur porte une note.

```
once_from_to(cmdset or cmdremove, memoCarried, not memoCarried)
```

3.1.6 Guidage de la génération de test

La simulation aléatoire des actions de l'utilisateur génère des entrées dans lesquelles chaque événement a la même probabilité d'arriver. Cela signifie que Localization[xminus] est généré autant de fois que Localization[xplus]. En conséquence, la position de l'utilisateur change difficilement. Pour tester l'application d'une manière plus réaliste, la génération de données peut être guidée par les profils opérationnels ou par les scénarios.

Les profils opérationnels sont définis par des probabilités conditionnelles ou non conditionnelles d'occurrence des entrées. Les probabilités non conditionnelles peuvent être utilisées pour forcer la simulation à correspondre à un cas particulier. Par exemple, si l'on souhaite que l'utilisateur tourne le plus souvent la tête vers la droite, on pourrait écrire :

```
proba((Orientation[yawminus], 0.80), (Orientation[yawplus], 0.01),
      (Orientation[pitchminus], 0.01), (Orientation[pitchplus], 0.01),
      (Orientation[rollminus], 0.01), (Orientation[rollplus, 0.01))
```

Des conditions peuvent être associées aux probabilités. A un instant, on peut spécifier qu'une commande "get" a une forte probabilité d'être générée quand l'utilisateur voit une note :

```
proba((Mouse[get], 0.8, pre memoSeen),
      (Keyboard[get], 0.8, pre memoSeen),
      (Speech[get], 0.8, pre memoSeen))
```

On peut aussi spécifier que, quand il n'y a pas une note visible, l'utilisateur va probablement bouger :

```
proba((Orientation[yawminus], 0.9, not pre MemoSeen),… )
```

Une autre possibilité offerte par Lutess est de guider la génération en spécifiant des scénarios qui correspondent à des classes de comportements. Un scénario est une séquence d'actions et de conditions qui doivent être vérifiées entre deux actions successives. Pendant la génération de test, les entrées correspondant au scénario ont une probabilité forte d'être générées. Par exemple, dans le scénario de la Figure 3-2, l'utilisateur effectue deux fois la commande "get", puis une commande "set" et finalement une commande "remove". Et entre les deux premiers "get", l'utilisateur ne fait pas de commande "set" ; et entre les deux commandes "get" et "set", il n'y a pas de commande "get".

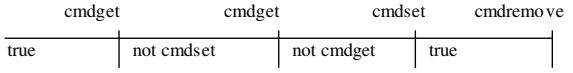

Figure 3-2: Exemple de scénario.

Ce scénario peut être exprimé en Lutess :

```
cond(      (Mouse[get] or Keyboard[get] or Speech[get]),
           (Mouse[get] or Keyboard[get] or Speech[get]),
           (Mouse[set] or Keyboard[set] or Speech[set]),
           (Mouse[remove] or Keyboard[remove] or Speech[remove]));
intercond( true,
           not(Mouse[set] or Keyboard[set] or Speech[set]),
```

```
        not(Mouse[get] or Keyboard[get] or Speech[get]),
        true);
```

Cas particulier de la fusion entre les événements (CARE)

Quand deux modalités sont utilisées avec le mode redondance ou le mode complémentarité [20], les événements correspondants doivent arriver dans la même fenêtre temporelle afin d'effectuer la fusion entre eux et aboutir à la tâche associée. Supposons que T est la durée de cette fenêtre temporelle (c'est un paramètre du système interactif), et que C est la durée d'un cycle d'exécution du générateur de Lutess (c'est le temps qui sépare la génération de deux entrées successives). C est déterminé empiriquement et il est constant pour un type donné de génération. C'est pourquoi, si N = T div C, alors N est le nombre de cycles exécutés dans la fenêtre temporelle. Si on impose que la probabilité de l'occurrence d'un événement d'entrée est 1/N, alors cet événement arrivera environ une fois chaque fois qu'on aura exécuté N cycles (c'est-à-dire environ une fois par fenêtre temporelle, d'une durée T). En conséquence, si on désire qu'un événement d'entrée soit généré dans la fenêtre temporelle, on spécifie que sa probabilité d'occurrence est supérieure ou égale à 1/N.

Par exemple, pour que les deux commandes Mouse[get] et Speech[get] soient générés dans cet ordre et dans la même fenêtre temporelle, on peut écrire :

```
proba(Speech[get],
      1/N,
      after(Mouse[get]) and pre always_since(not Speech[get], Mouse[get]));
```

En effet, cette formule signifie que si au moins un événement Mouse[get] a été généré dans le passé et l'événement Speech[get] n'est pas encore généré depuis la dernière occurrence de Mouse[get], alors la probabilité d'occurrence de Speech[get] est égal à 1/N. En effet, la fenêtre temporelle commence à l'instant de la dernière occurrence de Mouse[get] et dure N tops. Speech[get] va donc très probablement arriver au moins une fois avant la fin de la fenêtre temporelle.

Le résultat expérimental présenté ci-dessous utilise ce principe afin de valider les propriétés CARE pour le système interactif Memo.

3.1.7 Résultats de test

Simulation aléatoire

Nous avons testé Memo avec la simulation aléatoire (c'est-à-dire la génération aléatoire d'entrées validant la spécification de l'environnement de la section 3.1.4). Figure 3-3 montre un extrait de la trace résultat. La dernière colonne contient la valeur de l'oracle (1 signifie "true"). On utilise les abréviations suivantes :

- (*ya*, *p*, *r*) pour yaw, pitch et roll respectivement,
- (*mG*, *mS*, *mR*) pour Mouse[get], Mouse[set] et Mouse[remove] respectivement,
- de même (*kG*, *kS*, *kR*) pour la modalité "keyboard" et (*sG*, *sS*, *sR*) pour "speech",
- *Se* pour memoSeen, *Car* pour memoCarried,
- *Tak* pour memoTaken et *Rem* pour memoRemoved

```
x- z- -  -  -  -  -   ya+ -  p+ -  r+ -  -  -  -  -  -  -  -  -   -  Se -  -  -  1
-  -  -  -  -  -  -   -   -  -  -  -  -  mG -  -  -  -  -  -  -  -   -  Se Car Tak -  1
-  -  -  z+ y+ -  -   ya+ p- -  -  -  -  -  -  -  -  -  -  -  -   -  Se Car -  -  1
-  -  -  -  -  -  -   -   -  -  -  -  -  -  -  kG -  -  -  -  -   sR -  -  Tak Rem 1
x- z- -  -  -  y- -   -   -  -  r- -  -  -  -  -  -  -  -  -  -   -  Se -  -  -  1
-  -  -  -  -  -  -   -   -  -  -  -  -  -  mS -  -  -  -  -  -   sR Se -  -  Rem 1
x- z- -  -  y+ -  ya- -   p- -  -  -  -  -  -  -  -  -  -  -  -   -  Se -  -  -  1
-  -  -  -  -  -  -   -   -  -  -  -  -  mS -  -  kS -  sG -  -   -  -  Car Tak -  1
-  -  x+ -  y+ -  ya- -   -  -  -  r+ -  -  -  -  -  -  -  -  -   -  -  Car -  -  1
-  -  -  -  -  -  -   -   -  -  -  -  mG -  -  kG -  -  sG -  -   -  -  Car -  -  1
x- -  -  -  -  y- -   -   p- -  r- -  -  -  -  -  -  -  -  -  -   -  -  Car -  -  1
-  -  -  -  -  -  -   -   -  -  -  -  -  -  mR -  -  kR -  -  -   sR -  -  -  Rem 1
```

Figure 3-3 : Un extrait d'une simulation aléatoire de Memo

Utilisation de profils opérationnels

Nous avons aussi testé Memo en utilisant des profils opérationnels. Comme les résultas ci-dessous le montrent, nous nous sommes concentrés sur les propriétés CARE. La *complementarité* a été beaucoup testée car la manipulation d'une note implique deux fusions à accomplir : une première fusion est effectuée pour préciser la note sélectionnée (fusion de la localisation et de l'orientation). Puis une deuxième fusion est effectuée pour associer la note sélectionnée et une commande (en utilisant la parole, le clavier ou la souris). Les traces, qui sont présentées ci-dessous, montrent que l'*équivalence* a été testée considérablement : Les trois modalités équivalentes, la souris, le clavier et la parole, sont fréquemment simulées. La *redondance* a été testée à la troisième expérimentation. L'assignation n'a pas été testée, car elle concerne seulement la commande "exit" dans Memo.

Première expérimentation

Dans cette expérimentation, nous avons choisi les probabilités telles que l'utilisateur bouge très probablement quand il ne voit pas une note, et fait très probablement une commande "get" sinon.

```
proba(
   (Mouse[get], 0.9, pre memoSeen),
   (Clavier[get], 0.7, pre memoSeen),
   (Speech[get], 0.5, pre memoSeen),
   (Localisation[xminus], 0.5, not pre memoSeen),
   (Localisation[zminus], 0.5, not pre memoSeen),
   (Localisation[xplus],  0.8, not pre memoSeen),
   (Localisation[zplus],  0.8, not pre memoSeen),
   (Localisation[yplus],  0.5, not pre memoSeen),
   (Localisation[yminus], 0.8, not pre memoSeen));
```

Dans l'extrait de la trace ci-dessous, on peut observer que, quand il n'y a pas une note visible, l'utilisateur bouge, et quand une note est visible (*Se* arrive dans le top précédent), l'utilisateur la récupère (*Tak*).

```
1      :  -  -  -  -  -  -  -  -  -  Se -
2      :  -  -  -  -  -  -  -  mG kG sG -  Tak
3      :  -  z- x+ -  y+ -  -  -  -  -  -
4      :  x- -  -  z+ -  y- -  -  -  -  -
5      :  -  -  x+ z+ y+ -  -  -  -  -  -
6      :  -  z- x+ -  y+ -  -  -  -  -  -
7      :  -  z- -  -  -  -  -  -  -  -  -
8      :  -  -  -  -  -  -  -  -  -  Se -
9      :  -  -  -  -  -  -  mG -  sG -  Tak
```

```
10      :  x- z- -  -  -  -  -  -  -  -  -
11      :  -  z- -  -  y+ -  -  -  -  -  -
12      :  x- z- -  -  -  y- -  -  -  Se -
13      :  -  -  -  -  -  -  -  mG kG sG Se Tak
14      :  -  -  -  -  -  -  -  mG -  -  Se Tak
15      :  -  -  -  -  -  -  -  mG kG sG -  Tak
16      :  x- z- -  -  y+ -  -  -  -  Se -
17      :  -  -  -  -  -  -  -  mG kG -  -  Tak
18      :  x- -  -  z+ y+ -  -  -  -  -  -
19      :  x- z- -  -  y+ -  -  -  -  -  -
20      :  -  -  x+ z+ y+ -  -  -  -  -  -
21      :  x- z- -  -  y+ -  -  -  -  -  -
22      :  -  z- x+ -  -  y- -  -  -  -  -
23      :  -  -  x+ z+ y+ -  -  -  -  -  -
```

Deuxième expérimentation

Pour cette expérimentation, nous avons préliminairement mis quelques notes sur l'axe *x*. L'utilisateur, qui bouge selon cet axe, doit supprimer les notes. Nous avons choisi une forte probabilité pour que l'utilisateur continue à bouger dans la même direction. Mais quand il voit une note, il la supprime probablement. A la fin, quand une note est supprimée, l'utilisateur change sa direction avec une forte probabilité.

```
proba(
  (Localisation[xplus], 0.9, pre always_since(
    not Localisation[xminus], Localisation[xplus])),--(1)
  (Localisation[xplus], 1, pre always_since(
    not Localisation[xplus],
    Localisation[xminus]) and pre memoRemoved),     --(2)
  (Localisation[xminus], 0.9, pre always_since(
    not Localisation[xplus], Localisation[xminus])),--(3)
  (Localisation[xminus], 1, pre always_since(
    not Localisation[xminus],
    Localisation[xplus]) and pre memoRemoved),      --(4)
  (Mouse[remove], 0.9, pre memoSeen ),              --(5)
  (Clavier[remove], 0.9, pre memoSeen),             --(6)
  (Speech[remove], 0.9, pre memoSeen));             --(7)
```

La ligne (1) signifie qu'il y a une forte probabilité pour que l'utilisateur bouge dans la direction (x+), si celui-ci n'a pas bougé dans la direction (x-) depuis la dernière occurrence de (x+) (le dernier mouvement est x+). La ligne (2) signifie qu'il y a une forte probabilité pour que l'utilisateur change de direction quand une note vient d'être supprimée. Les lignes (3), (4) sont similaires à (1), (2).

Dans l'extrait de la trace résultat ci-dessous, on peut observer que l'utilisateur bouge souvent dans la même direction jusqu'à ce qu'il trouve une note (pas 64: événement memoSeen *Se*). Ensuite, il la supprime (pas 65: événement memoRemoved *Rem*), Puis il change de direction… etc.

```
53      :  -  -  mR kR sR -  -
```

```
54      :  -  x+ -  -  -  -  -
55      :  -  -  mR kR -  -  -
56      :  -  x+ -  -  -  -  -
57      :  -  -  -  kR -  -  -
58      :  -  x+ -  -  -  -  -
59      :  -  -  -  -  -  -  -
60      :  -  x+ -  -  -  -  -
61      :  -  -  -  -  -  -  -
62      :  -  x+ -  -  -  -  -
63      :  -  -  mR -  -  -  -
64      :  -  x+ -  -  -  Se -
65      :  -  -  -  kR sR -  Rem
66      :  x- -  -  -  -  -  -
67      :  -  -  -  -  sR -  -
68      :  x- -  -  -  -  -  -
69      :  -  -  mR -  sR -  -
70      :  -  x+ -  -  -  -  -
71      :  -  -  -  -  -  -  -
72      :  -  x+ -  -  -  -  -
73      :  -  -  -  -  -  -  -
74      :  -  x+ -  -  -  -  -
75      :  -  -  -  -  -  -  -
76      :  -  x+ -  -  -  -  -
77      :  -  -  -  kR -  -  -
78      :  -  x+ -  -  -  Se -
79      :  -  -  -  -  -  -  -
80      :  -  x+ -  -  -  -  -
81      :  -  -  -  -  sR -  -
82      :  -  x+ -  -  -  Se -
83      :  -  -  mR kR sR -  Rem
```

Troisième expérimentation

Cet exemple a pour but de valider l'utilisation redondante de deux modalités : la souris et la parole. Nous avons reconfiguré l'application Memo avec le mode redondance entre la souris et la parole. Avec ce mode, pour exécuter une commande, un événement de chacune des modalités redondantes est nécessaire et les deux événements doivent arriver dans la même fenêtre temporelle. Nous avons fait le test de plusieurs manières :

- D'abord nous avons spécifié une forte probabilité d'effectuer une commande "get" avec la souris et la parole quand une note est visible.

```
proba (  (Speech[get], 0.9, pre memoSeen),
         (Mouse[get],  0.9, pre memoSeen));
```

Voici un extrait de la trace résultat.

```
1       :  -  -  Se -   -
2       :  -  -  Se -   -
```

```
3       :  mG sG Se Car Tak
4       :  -  -  Se Car -
5       :  mG -  Se Car -
```

On commence le test dans un état où il n'y a aucune note près de l'utilisateur. Le pas 2 contient l'événement memoSeen (*Se*), indiquant qu'une ou plusieurs notes sont près de l'utilisateur. Dans le pas 4, les deux événements simultanés mouseGet and speechGet (*mG* and *sG*) provoquent, à cause de la redondance, la récupération d'une note (*Tak*). Alors, une note est toujours visible (*Se*), et l'utilisateur porte une note (*Car*). On observe que dans le pas 5, le seul événement mouseGet (*mG*) ne provoque aucune réaction, parce qu'avec ce mode, on a besoin de deux événements redondants pour effectuer une tâche.

– Supposons que la situation suivante doive être testée : Quand une note est visible, une commande "get" avec la souris et une commande "get" avec la parole sont générées dans la même fenêtre temporelle, mais pas au même instant. Pour cela, on donne une probabilité *pr* de générer `Speech[get]` quand une note est visible et `Speech[get]` n'est pas encore généré depuis la dernière occurrence de `Mouse[get]`. Dans les même conditions, on spécifie aussi une probabilité 0 pour l'événement `Mouse[get]` afin de ne pas générer les deux événements `Speech[get]` et `Mouse[get]` au même instant.

```
proba (
  (Speech[get], pr, pre memoSeen and
   after (Mouse[get]) and
   pre always_since(not Speech[get], Mouse[get])),
  (Mouse[get], 0, pre memoSeen and
   after (Mouse[get]) and
   pre always_since(not Speech[get], Mouse[get])) );
```

On va illustrer comment on choisit la valeur *pr* (voir section0). Supposons que T=5000ms est la durée de la fenêtre temporelle et que C=1000ms est la durée du cycle d'exécution (i.e. la fréquence des événements d'entrées générés par Lutess). Si *pr* = 1000/5000 = 0.2, alors `Speech[get]` va arriver environ une fois tous les 5 cycles d'exécution quand la pré-condition est "true" (`Mouse[get]` est arrivé et `Speech[get]` n'est pas arrivé depuis `Mouse[get]`). Si on souhaite que les deux événements soient plus proches, on doit augmenter *pr*. On a testé cet exemple avec des différentes valeurs de *pr* :

- Un extrait de la trace résultat est ci-dessous pour *pr* = 0.2.

```
1       :  mG -  Se -   -
2       :  -  -  Se -   -
3       :  -  -  Se -   -                T
4       :  -  -  Se -   -
5       :  -  -  Se -   -
6       :  -  -  Se -   -
7       :  -  -  Se -   -
8       :  -  -  Se -   -
9       :  -  sG Se -   -
10      :  -  -  Se -   -
```

```
11      :  -  sG Se -   -
12      :  -  -  Se -   -
13      :  -  -  Se -   -
14      :  -  -  Se -   -
```

La distance temporelle entre les deux événements mouseGet et speechGet (*mG*, *sG*) est égale à 8 cycles (8000 ms), qui est plus long que T, et donc aucune tâche n'est exécutée. En regardant la trace complète, on peut observer que la distance moyenne entre *mG* et *sG* est environ 5 cycles (5000ms).

- Voici un extrait de la trace résultat pour *pr* = 0.8:

```
145     :  -  -  Se -   -
146     :  -  -  Se -   -
147     :  mG -  Se -   -
148     :  -  -  Se -   -
149     :  -  sG Se Car Tak                T
150     :  -  -  Se Car -
151     :  -  -  Se Car -
```

Les événements *mG* et *sG* sont plus proches. On peut observer que les deux événements *mG* et *sG*, qui arrivent dans la même fenêtre temporelle, provoquent la prise d'une note, l'autre note restant dans le champ visuel.

Quatrième expérimentation

Dans cette expérimentation, on utilise les mêmes formules de celles de l'expérimentation précédente. Mais, on a reconfiguré le système avec le mode « RedondanceEquivalence ». Avec ce mode, le système utilise les deux modes Redondance et Equivalence en même temps : Pour deux événements apparaissent dans la même fenêtre temporelle et portant la même information, il y a seulement une tâche exécutée (mode Redondance). Cependant, un seul événement peut provoquer l'exécution d'une tâche (mode Equivalence).

D'abord, nous avons choisi les probabilités comme indiqué ci-dessous.

```
proba (  (Speech[get], 0.9, pre memoSeen),
         (Mouse(get],  0.9, pre memoSeen) );
```

Un extrait de la trace résultat est ci-dessous pour N = T/C = 10 000/1000 = 10

```
…
10      :  -  -  Se -   -
11      :  mG sG Se Car Tak
12      :  -  -  Se Car -
13      :  mG -  Se Car -
14      :  -  -  Se Car -
15      :  mG sG Se Car -
```

```
    16      :  -  -  Se Car -                    T
    17      :  mG sG Se Car -
    18      :  -  -  Se Car -
    19      :  mG sG Se Car -
    20      :  -  -  Se Car -
    21      :  mG -  -  Car Tak
    22      :  -  -  -  Car -
    23      :  -  -  Se -   -
    24      :  -  -  Se -   -
    25      :  mG sG Se -   -
    26      :  -  -  Se -   -
    27      :  mG sG Se -   -
    28      :  -  -  Se -   -
    29      :  mG sG Se -   -
```

Comme dans l'exemple précédent, nous avons mis deux notes proches de l'utilisateur avant le test. L'étape11 montre que, à cause du mode redondance, les deux événements simultanés mouseGet et speechGet (*mG* et *sG*) provoquent la capture d'une note (*Tak*). En conséquence, une note est toujours visible (*Se*), et une note est portée (*Car*). Ensuite, tous les événements "get" dans la même fenêtre temporelle sont ignorés. Dans l'étape 21, à cause du mode équivalence, le seul événement mouseGet (*mG*) provoque la capture d'une note numérique.

Supposons que l'on veut valider le mécanisme de fusion. Pour cela, nous avons choisi la formule de probabilités comme celle de la troisième expérimentation : on assigne une probabilité *pr* pour `Speech[get]` quand une note est visible et `Speech[get]` n'est pas encore arrivé depuis la dernière occurrence de `Mouse[get]`. On assigne la probabilité 0 à `Mouse[get]` dans ces conditions pour éviter l'occurrence de deux événements au même instant.

```
proba (
(Speech[get], pr, pre memoSeen and
   after(Mouse[get]) and
   pre always_since(not Speech[get], Mouse[get])),
(Mouse[get], 0, pre memoSeen and
   after(Mouse[get] and
   pre always_since(not Speech[get], Mouse[get])) );
```

Un extrait de la trace résultat est juste au-dessous avec *pr* = 0.8 and N = 5000/1000 = 5 cycles:

```
     1      :  -  -  -  -   -
     2      :  -  -  Se -   -
     3      :  -  -  Se -   -
     4      :  -  -  Se -   -
     5      :  -  -  Se -   -
     6      :  -  -  Se -   -
     7      :  -  -  Se -   -
     8      :  mG -  Se Car Tak
     9      :  -  -  Se Car -
```

```
10      :  -  -  Se Car -
11      :  -  sG Se Car -
12      :  -  -  Se Car -
13      :  -  sG -  Car Tak
14      :  -  -  -  Car -
15      :  -  -  -  Car -
16      :  -  -  -  Car -
17      :  -  sG Se -   -
18      :  -  -  Se -   -
19      :  -  -  Se -   -
20      :  -  -  Se -   -
21      :  mG -  Se -   Tak
22      :  -  -  Se -   -
23      :  -  sG Se -   -
24      :  -  -  Se -   -
```

Dans l'étape 8, à cause de l'équivalence, l'événement mouseGet (*mG*) provoque la capture d'une note. Ensuite l'événement speechGet (*sG*) à l'étape 11 est ignoré, à cause de la redondance, etc.

Guidage par schéma (scénario)

Premier scénario

Ce scénario décrit une utilisation redondante de deux modalités: la souris et la parole. On commence le scénario dans un état ou deux notes sont proches de l'utilisateur. L'utilisateur d'abord récupère une note (note 1), en utilisant la souris et la parole au même instant. Après, l'utilisateur supprime une note en utilisant aussi la souris et la parole, à deux instants différents (mais proches). Le résultat sera que, la note dans l'environnement physique (note 2) sera supprimée et l'utilisateur continue à porter la note 1. En fait, la priorité est donnée à la manipulation de notes dans l'environnement physique. Le scénario est le suivant :

```
cond( pre memoSeen and (Speech[get] and Mouse[get]) and
               not (Speech[remove] or Mouse[remove]),
               Mouse[remove] and not Speech[remove],
               Speech[remove] and not Mouse[remove]);
intercond(  true, not Speech[remove], not Mouse[remove]);
```

Un extrait de la trace résultat est juste au-dessous.

```
1    -    -    -    -    Se   -    -   -          1
2    mG   -    sG   -    Se   Car  Tak -          1
3    -    mR   -    -    Se   Car  -   -          1
4    -    -    -    sR   Se   Car  -   -          1
5    -    -    -    -    -    Car  -   Rem        1
```

Dans cette trace, le premier pas contient l'événement memoSeen(*Se*), signifiant qu'une ou plusieurs notes sont proches de l'utilisateur. Au deuxième pas, les deux événements simultanés mouseGet et speechGet (*mG* et *sG*) provoquent la capture d'une note. Les pas 3 et 4 contiennent les événements mouseRemoved et speechRemoved (*mR* et *sR*) qui provoquent la capture de la note vue (l'événement *Rem* dans le cinquième pas).

Deuxième scénario

Dans ce scénario, on considère quelques notes sur l'axe *x*. Pendant ce scénario, l'utilisateur bouge selon l'axe *x* et supprime les notes. Quand l'utilisateur supprime une note, il change de direction. On décrit ce scénario de la façon suivante.

```
cond(     true,
          pre (memoSeen or memoCarried) and
             (Mouse[remove] or Keyboard[remove] or Speech[remove]),
          pre (memoSeen or memoCarried) and
             (Mouse[remove] or Keyboard[remove] or Speech[remove]));
intercond(  true,
          Implies (OR (6, Localization), Localization[xminus]), -- (1)
          Implies (OR (6, Localization), Localization[xplus])); -- (2)
```

`OR(6, Localization)` signifie que au moins une des six valeurs booléennes de localisation est vraie. La condition 1 (respectivement la condition 2) signifie que chaque fois que l'utilisateur essaie de bouger, il est obligé de bouger vers la gauche (respectivement vers la droite).

Un extrait de la trace résultat est juste au-dessous.

1	–	–	mG	–	–	kR	sG	–	–	–	–
2	–	–	mG	–	kG	–	sG	–	–	–	–
3	–	x+	–	–	–	–	–	–	–	–	–
4	–	x+	–	–	–	–	–	–	–	–	–
5	–	–	mG	–	–	kR	sG	–	–	–	–
6	–	–	mG	–	–	–	–	sR	–	–	–
7	–	x+	–	–	–	–	–	–	–	–	–
8	–	x+	–	–	–	–	–	–	Se	–	–
9	–	–	–	–	–	–	–	sR	–	–	Rem

Comme spécifié par le scénario, cette trace illustre la génération d'événements de localisation, correspondant au mouvement dans la direction (*x+*), jusqu'à trouver une note proche de l'utilisateur (*Se* pas 8). Puis, la note est supprimée en utilisant la parole (*sR* et *Rem* au pas 9). On peut observer que les commandes remove *kR* et *sR* pas 1, 5 et 6, qui apparaissent quand il n'y a pas de note visible (*Se = false*), n'a aucun effet.

3.2 Test de systèmes interactifs multimodaux avec Lutess

Dans ce paragraphe nous présentons d'une manière générale et synthétique les principes de spécification et de guidage que nous proposons et qui ont été, en partie, appliqués au cas de Mémo dans le paragraphe précédent.

3.2.1 Adéquation de l'approche synchrone

En pratique, l'hypothèse de synchronisme est vérifiée si le logiciel est capable de prendre en compte toute l'évolution de son environnement externe. Ainsi, une application interactive peut être vue comme un programme synchrone tant que toutes les actions de l'utilisateur et autres stimuli externes sont pris en considération lors de son exécution.

Dans [23], il est montré qu'un système asynchrone peut être modélisé avec un modèle synchrone, en laissant les processus "bégayer", ou "rester silencieux" d'une manière

indéterministe. En fait, plusieurs langages synchrones fournissent une manière de "bloquer" ou "geler" un processus:

- En Esterel, l'expression "`suspend P when S`" empêche **P** de la réaction quand le signal **S** est présent.
- En Lustre, les sorties sont « gelées » à leurs dernières valeurs en utilisant l'opérateur "`current`".
- Aux circuits synchrones, cela peut être réalisé par la désactivation de l'horloge.

Afin de bien comprendre la manière de communiquer d'un système synchrone avec un système asynchrone, on peut se référer aux explications fournies dans [22] [21] : un programme synchrone **P** progresse via des réactions :

$$run(P) = \text{séquence des tuplets des événements}$$

$$= \{(x_i(1))_{i=1,\ldots,k}, (x_i(2))_{i=1,\ldots,k}, \ldots\}$$

Pendant une réaction, la décision peut être prise en testant l'absence de quelques signaux (dénotée par *"x=⊥"*). Par exemple, à l'instruction `y = current x` en Lustre, y est la dernière valeur de x si x n'est pas présent.

Alors, on peut dire qu'une exécution d'un système synchrone est une séquence de tuplets de valeurs dans des domaines étendus par le symbole ⊥.

Dans le cas des systèmes asynchrones, il n'y a pas de réaction, ni horloge globale. Pour chaque variable, on sait uniquement la séquence ordonnée des valeurs présentes. Ainsi, une exécution d'un système asynchrone est un tuplet de séquences des valeurs présentes :

$$run(P) = \text{tuplet de séquences des événements}$$

$$= (\{x_i(1), x_i(2), \ldots\})_{i=1,\ldots,k}$$

On peut dire autrement qu'une exécution d'un système asynchrone est un tuplet de canaux. L'absence de valeurs n'a pas de sens (⌐ [x=⊥]).

C'est pourquoi, désynchroniser une exécution est une application d'une séquence de tuplets de valeurs aux domaines étendus par ⊥ dans un tuplet de séquences de valeurs présentes, une séquence étant associée à une variable (cf. Figure 3-4).

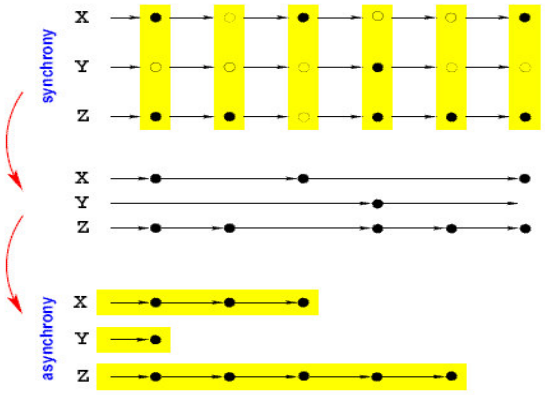

Figure 3-4: désynchronisation d'une trace synchrone

Cependant, est-ce que c'est possible de construire une trace synchrone à partir d'une trace asynchrone?

En effet, c'est possible si on affecte à chaque signal s une horloge (variable) hs qui est toujours présente (voir la Figure 3-5). A chaque cycle de l'horloge globale, on examine l'existence de chaque signal : si un signal quelconque s existe, alors l'horloge hs sera vraie.

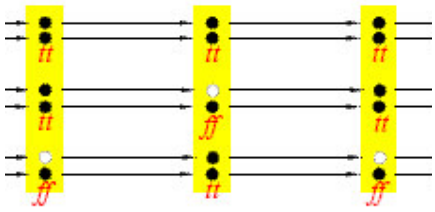

Figure 3-5: synchronisation d'une trace asynchrone

Illustrons ces propos sur l'étude de cas Memo. Sur la Figure 3-6, un traducteur est l'intermédiaire entre le côté synchrone et le côté asynchrone. Il prend en charge la désynchronisation des événements d'entrée pour le système interactif et la synchronisation des événements de sortie du système interactif.

Du côté synchrone, comme on a mentionné avant, on affecte à chaque événement d'entrée ou de sortie une variable booléenne qui est toujours présente. Pour les entrées, à chaque cycle, les valeurs des variables booléennes associées aux événements d'entrée sont examinées par le traducteur: une valeur "true" pour une variable booléenne signifie que l'événement associé est présent et il sera envoyé au système interactif par la modalité adéquate, sinon l'événement associé n'est pas présent. Ensuite, le traducteur attend les événements de sortie de l'application interactive sur les modalités de sortie. Quand un événement quelconque de sortie est présent sur une modalité de sortie, le traducteur donne la valeur "true" à la variable booléenne associée et donne la valeur "false" aux variables booléennes associées aux événements qui ne sont pas présents.

Dans la Figure 3-6, on ne considère qu'une modalité d'entrée pour Memo (la souris) et une seule modalité de sortie (l'écran). Les événements qui peuvent passer par la modalité "Mouse" sont : "get", "set" et "remove". C'est pourquoi, on a trois variables booléennes associées à ces événements: mouse[get], mouse[set] et mouse[remove]. Sur le canal (modalité) de sortie "Screen", on peut recevoir trois sortes de messages écrits : "memo is taken" (cela correspond à l'événement "memoTaken"), "memo is set" (cela correspond à l'événement "memoSet") et "memo is removed" (cela correspond à l'événement "memoRemoved"). En conséquence, on a trois variables booléennes associées à ces événements: screen[memoTaken], screen[memoSet] et screen[memoRemoved].

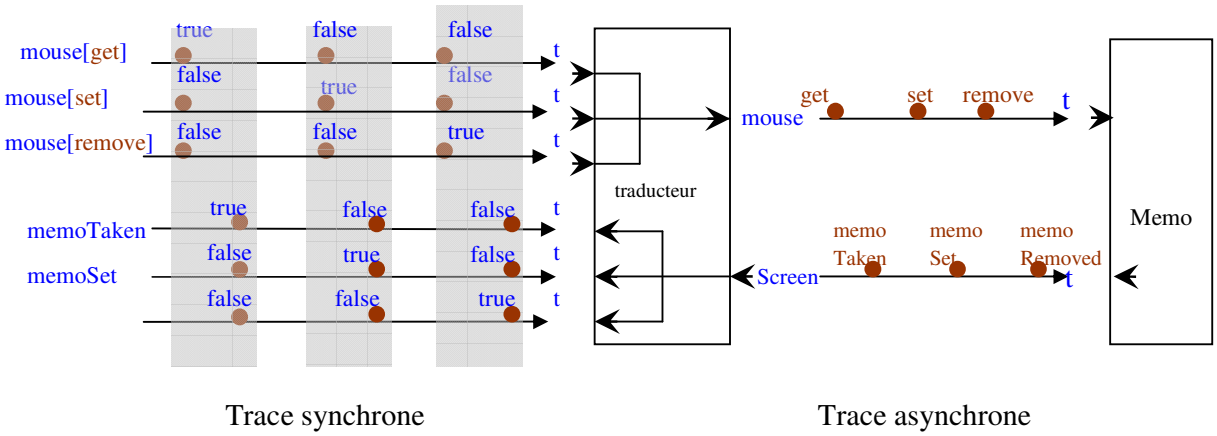

Figure 3-6: transformation de trace de Memo en trace synchrone

3.2.2 Architecture technique

Choix du niveau d'abstraction

La connexion d'un système multimodal avec Lutess nécessite la définition d'un niveau d'abstraction des événements générés par Lutess. En fait, le niveau d'abstraction des événements détermine quel composant dans le système multimodal sera connecté avec Lutess. Si on considère l'architecture logicielle PAC-Amodeus [2], [10], Lutess pourrait être connecté à trois composants au choix, comme le montre la Figure 3-3-7.

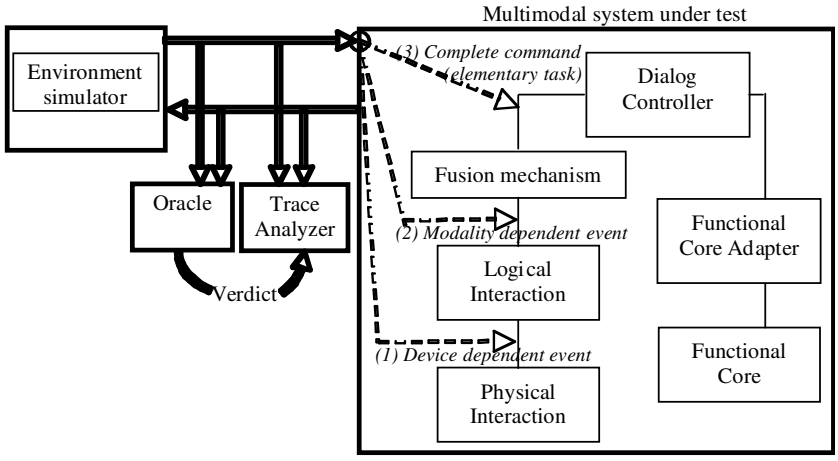

Figure 3-3-7 : ***trois solutions pour brancher un système multimodal construit avec l'architecture PAC-Amodeus avec Lutess.***

La première solution est de connecter Lutess au composant Interaction Logique. En conséquence, Lutess va envoyer des séquences d'événements de bas niveau (dispositif) au système multimodal sous test. Par exemple dans le cas de l'application multi-modale Memo, qui permet de manipuler des notes virtuelles, Lutess enverrait à Memo des événements correspondant aux clics des boutons de la souris. Une deuxième solution est de connecter Lutess au mécanisme de fusion. Par conséquent, les événements générés par Lutess

correspondent aux modalités (dispositif et langage). Par exemple, pour tester Memo, Lutess peut envoyer des événements comme <Mouse-get> ou <speech-remove>. Une troisième solution est de connecter Lutess avec le Contrôleur de Dialogue. Les événements envoyés par Lutess au système multimodal seront alors des commandes complètes comme<remove note3>. Dans le cas de Memo, nous avons opté pour la deuxième solution (mécanisme de fusion) : en effet, ce choix a permis de conserver une connaissance fine des événements et de leurs modalités.

Enfin, le composant d'interaction physique ne peut pas interagir avec Lutess, parce qu'il ne génère pas d'actions physiques.

Cas particulier de l'architecture ICARE

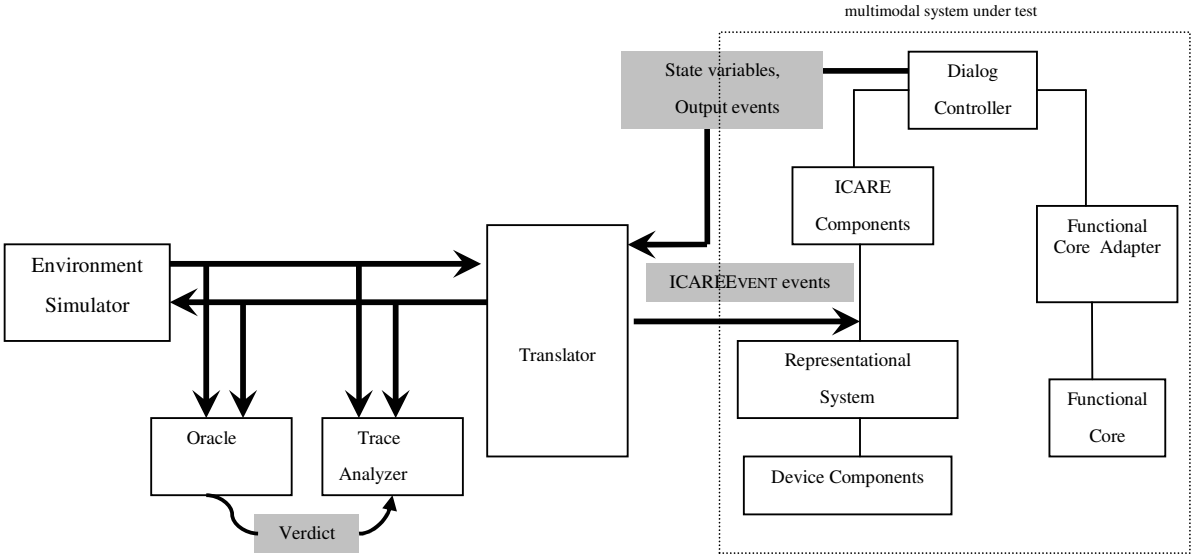

Figure 3-8: la communication entre Lutess et le système multimodal construit avec ICARE

La Figure 3-8 montre l'architecture adoptée pour faire communiquer Lutess et un système multimodal construit avec les composants ICARE [2] [3]. Lutess envoie les événements d'entrée au niveau modalité <langage, dispositif>. Les événements de sortie et les variables d'état sont récupérés du contrôleur de dialogue. Cela permet de tester le moteur de la fusion et la partie contrôle de l'application.

Le traducteur est l'intermédiaire entre l'environnement du test Lutess et le système multimodal. Il traduit les événements synchrones booléens générés par Lutess en des événements compréhensibles par le système interactif (dans notre cas événements de type ICAREEvent), et inversement pour les événements de sortie et les variables d'état du système.

Si on veut tester les composants du Système Représentationnel de l'application, on peut modifier le traducteur afin qu'il envoie les événements d'entrées de niveau dispositif (comme un événement correspondant au clic du bouton gauche de la souris). Ces événements sont envoyés aux composants du Système Représentationnel.

3.2.3 Expression de propriétés

Dans ce paragraphe nous présentons différents types de propriétés étudiées dans le cadre de l'étude de cas et la manière dont on peut les exprimer en Lustre afin de les intégrer dans un oracle automatique. Il ne s'agit pas d'une liste exhaustive de tous types de propriétés exprimables. Toutefois, nous avons, en particulier, traité le cas des propriétés CARE qui sont propres aux applications multimodales.

Propriétés CARE

Equivalence

On dit que deux modalités M1 et M2 sont équivalentes par rapport à un ensemble T de tâches du système, si chaque tache $t \in T$ peut être provoquée par une tâche issue de M1 ou de M2 (voir [20]). En d'autres mots, une de ces modalités peut remplacer l'autre pour provoquer les tâches T du système. Supposons que TA_{M1} une tâche issue de M1, TA_{M2} une tâche issue de M2. TA_{M1} ou TA_{M2} peut solliciter la tâche système $TA_S \in T$. Alors on peut exprimer l'équivalence entre les deux tâches TA_{M1}, TA_{M2} déclenchant la tâche TA_S comme suit :

OnceFromTo (TA_{M1} or TA_{M2} , not TA_S , TA_S).

Redondance

On dit que deux modalités M1 et M2 sont redondantes par rapport à un ensemble partiel T de tâches de système, si chaque tache système $t \in T$ doit être déclenchée par deux tâches identiques issues de M1 et de M2 simultanément ou séquentiellement dans la même fenêtre temporelle dont la durée est W (abs(time(TA_{M1}) - time(TA_{M2}))<W). Si les deux tâches TA_{M1} , TA_{M2} ont lieu, une seule tâche système TA_S sera exécutée.

Supposons que C est la durée d'un cycle d'exécution, alors N = W div C sera le nombre d'instants discrets dans la fenêtre temporelle. Dans ce cas la propriété de redondance en Lustre sera :

Implies (TA_S,

onceSinceTicks(TA_{M1},2*N) and onceSinceTicks(TA_{M2},2*N)

and abs (lastOccurrence(TA_{M1})- lastOccurrence(TA_{M2}))<= N

and atMostOneSince(TA_S, TA_{M1}) and atMostOneSince(TA_S, TA_{M2})

);

Cette propriété signifie qu'une exécution d'une tâche système TA_S implique d'abord l'existence de deux tâches identiques TA_{M1}, TA_{M2} issues de deux modalités redondantes M1, M2 (depuis une durée inférieure à 2*N*C) , que la distance temporelle entre les deux tâches est inférieure à N*C et qu'il y a une seule tâche système TA_S qui s'exécute (voir la Figure 3-9).

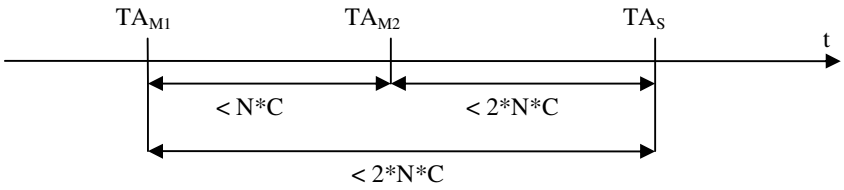

Figure 3-9

Complémentarité

La complémentarité entre deux modalités M1 et M2 par rapport à un ensemble T de tâches de système signifie que : pour effectuer une tâche système $TAB_S \in T$, il faut deux tâches (chacune complète l'autre) TA_{M1} , TB_{M2} proches temporellement (dans la même fenêtre temporelle de durée W) issues de ces deux modalités, c'est-à-dire : abs(time(TA_{M1}) - time(TB_{M2}))<W.

Si C est la durée d'un cycle d'exécution, et N = W div C le nombre d'instants dans la fenêtre temporelle :

Implies (TAB_S,

onceSinceTicks(TA_{M1},2*N) and onceSinceTicks(TB_{M2},2*N)

and abs (lastOccurrence(TA_{M1})- lastOccurrence(TB_{M2}))<= N

and atMostOneSince(TAB_S, TA_{M1}) and atMostOneSince(TAB_S, TB_{M2})

);

Cette propriété signifie qu'une exécution d'une tâche système TAB_S implique d'abord l'existence de deux tâches TA_{M1}, TB_{M2} issues de M1, M2 (depuis une durée inférieure à 2*N) , que la distance temporelle entre les deux tâches est inférieure à N et qu'il y a une seule tâche système TAB_S qui s'exécute le cas échéant (voir la Figure 3-10).

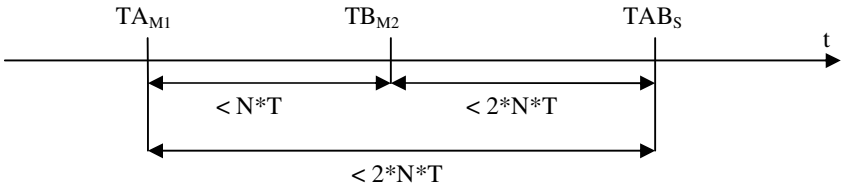

Figure 3-10

Assignation

On dit qu'une modalité M est une assignation pour effectuer un ensemble partiel T de tâches de système, quand chaque tâche système $t \in T$ ne peut être provoquée que par une tâche issue de M (r.f. [20]). Supposons que la tâche système $TA_S \in T$ ne peut être provoquée que par la tâche TA_M issue de M.

OnceFromTo (TA_M , not TA_S , TA_S);

Equivalence-redondance

La propriété d'équivalence-redondance comporte les propriétés d'équivalence et de redondance à la fois. Pour deux événements identiques TA_{M1}, TA_{M2} issues de M1 et M2 dans la même fenêtre temporelle, il y a une seule tâche système qui s'exécute. Cependant, un seul événement peut provoquer l'exécution de la tâche système :

OnceFromTo (TA_{M1} or TA_{M2} , not TA_S , TA_S) and -- équivalence

Implies(TA_S, atMostOnceSinceTicks(TA_S,N)); -- redondance.

Propriétés fonctionnelles

Lutess est plus particulièrement conçu pour tester la partie contrôle du système et, plus précisément, la capacité du système de transformer correctement une séquence d'événements d'entrée en sorties adéquates. Dans le cas d'un système interactif multimodal, le but du test est de valider qu'une séquence d'événements produits par l'utilisateur (représentés par des événements booléens) donne lieu à une séquence appropriée d'événements de sortie. Dans la mesure où ces propriétés peuvent s'exprimer dans une logique temporelle du passé, leur écriture en tant que formules Lustre est possible (cf. 3.1.4). Il est par contre impossible d'exprimer des propriétés de vivacité (« il est toujours possible de … »).

3.2.4 Guidage

Dans le cadre de l'expérimentation sur l'application Mémo, nous avons constaté que l'utilisation de probabilités et des scénarios a permis de reproduire des comportements de l'utilisateur correspondant à des utilisations réalistes de l'application.

Une utilisation intéressante du guidage par probabilités a été la génération de données permettant de tester la fusion entre modalités : en calculant la probabilité qu'on doit affecter aux différents événements, on arrive à produire des séquences dans lesquelles les événements à fusionner sont compris dans la même fenêtre temporelle (ce qui permet de valider la fusion).

4. Références bibliographiques

[1] Bolt, R. Put That There: Voice and Gesture at the Graphics Interface. Proc. of SIGGRAPH'80. ACM Press (1980) 262-270.

[2] Bouchet, J., Nigay, L., & Ganille, T. ICARE Software Components for Rapidly Developing Multimodal Interfaces. Proc. of ICMI'04. ACM Press (2004) 251-258.

[3] Bouchet, J., Nigay, L. ICARE: A Component-Based Approach for the Design and Development of Multimodal Interfaces. Proc. of CHI'04 extended abstract. ACM Press (2004) 1325-1328.

[4] Coutaz, J., Nigay, L., Salber, D., Blandford, A., May, J., & Young, R. Four Easy Pieces for Assessing the Usability of Multimodal Interaction: The CARE properties. Proc. Of INTERACT'95. Chapman et Hall (1995) 115-120.

[5] d'Ausbourg, B. Using Model Checking for the Automatic Validation of User Interfaces Systems. Proc. of DSVIS'98. Springer Verlag (1998) 242-260.

[6] du Bousquet, L., Ouabdesselam, F., Richier, J.-L., & Zuanon, N. Lutess: a Specification Driven Testing Environment for Synchronous Software. Proc. of ICSE'99. ACM Press (1999) 267-276.

[7] Duke, D., Harrison, M. Abstract Interaction Objects. Proc. of Eurographics'93. North Holland (1993) 25-36.

[8] Halbwachs, N. Synchronous programming of reactive systems, a tutorial and commented bibliography. Proc. of CAV'98, LNCS 1427. Springer Verlag (1998) 1-16.

[9] L. Madani, L. Nigay, I. Parissis. Testing the care properties of multimodal applications by means of a synchronous approach. Proc. of IASTED Int'l Conference on Software Engineering, (2005).

[10] J. Musa. Operational Profiles in Software-Reliability Engineering. IEEE Software (1993), 14–32.

[11] Nigay, L., Coutaz, J. A Generic Platform for Addressing the Multimodal Challenge. Proc. of CHI'95. ACM Press (1995) 98-105.

[12] F. Ouabdesselam, I. Parissis. Constructing Operational Profiles for Synchronous Critical Software. Proc. of 6th Int'l Symposium on Software Reliability Engineering (1995).

[13] Palanque, P., Bastide, R. Verification of Interactive Software by Analysis of its Formal Specification. Proc. of INTERACT'95. Chapman et Hall (1995) 191-197.

[14] Parissis, I., Ouabdesselam, F. Specification-based Testing of Synchronous Software. Proc. of ACM SIGSOFT Fourth Symposium on the Foundations of Software Engineering. ACM Press (1996) 127-134.

[15] I. Parissis, J. Vassy. Thoroughness of Specification-Based Testing of Synchronous Programs. Proc. of 14th. IEEE International Symposium on Software Reliability Engineering (2003) 191-202.

[16] Paterno, F., Faconti, G. On the Use of LOTOS to Describe Graphical Interaction. Proc. of HCI'92. Cambridge University Press (1992) 155-173.

[17] J. Whittaker. Markov chain techniques for software testing and reliability analysis. Thesis, University of Tenessee (1992).

[18] D. Woit. Specifying Operational Profiles for Modules. Proc. of the International Symposium on Software Testing and Analysis (1993) 2–10.

[19] Zouinar, M. et al. Multimodal Interaction on Mobile Artefacts. Chapter 4 in Communicating with smart objects. Hermes Penton Science/Kogan Page Science (2003).

[20] Nigay, L., Coutaz, J. Les propriétés “CARE” dans les interfaces multimodales.

[21] *http://www.irisa.fr/distribcom/benveniste/pub/B_al99.html*.

[22] A. Benveniste, B. Caillaud and P. Le Guernic. ``From synchrony to asynchrony." In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99,* Concurrency Theory, 10th International Conference, vol. 1664 of *Lecture Notes in Computer Science,* 162-177. Springer V., 1999.

[23] N. Halbwachs, S. Baghdadi Synchronous Modelling of Asynchronous Systems, 2002