Utilisation de la programmation synchrone pour la spécification et la validation de services interactifs

Laya Madani

Université Joseph Fourier - Grenoble 1

Utilisation de la programmation synchrone

pour

la spécification et la validation de services interactifs

Thèse

présentée par

Laya MADANI

en vue de l'obtention du grade de

Docteur de l'Université Joseph Fourier Grenoble 1

Discipline Informatique

soutenue le 29 Octobre 2007 devant le jury composé de :

M. Yamine Ait Ameur	Professeur ENSMA	**Rapporteur**
M. Bruno d'Ausbourg	Chercheur ONERA	**Rapporteur**
Mme. Laurence Nigay	Professeur UJF	**Examinateur**
Mme. Catherine Oriat	Maître de Conférences INPG	**Examinateur**
M. Farid Ouabdesselam	Professeur UJF	**Examinateur**
M. Ioannis Parissis	Maître de Conférences UJF	**Examinateur**

Remerciements

Cette thèse a débuté au sein du Laboratoire Logiciels, Systèmes et Réseaux de l'Institut IMAG et s'est poursuivie au sein du Laboratoire d'Informatique de Grenoble. Je tiens à exprimer ma reconnaissance aux directeurs respectifs, M. Farid Ouabdesselam et Mme Brigitte Plateau, pour m'avoir accueillie et m'avoir fourni des conditions de travail exceptionnelles.

Je tiens, également, à exprimer mes plus sincères remerciements à :

M. Bruno d'Ausbourg, chercheur ONERA (Toulouse) et M. Yamine Ait Ameur, Professeur à l'École Nationale Supérieure de Mécanique et d'Aérotechnique (Poitiers) pour avoir accepté d'être les rapporteurs de ce travail ;

Mme Laurence Nigay, Professeur à l'Université Joseph Fourier, pour avoir accepté de présider le jury de cette thèse ;

M. Farid Ouabdesselam, Président et Professeur de l'université Joseph Fourier, qui m'a fait l'honneur d'être mon directeur de thèse ;

M. Ioannis Parissis, Maître de Conférences à l'Université Joseph Fourier, qui m'a guidée, dirigée, conseillée pendant mes années de thèse et m'a permis, par sa relecture attentive de mon manuscrit, d'en améliorer la qualité ;

Mme Catherine Oriat, Maître de Conférences à l'Institut National Polytechnique de Grenoble, qui a bien voulu accepter de participer au jury de cette thèse et qui m'a beaucoup apporté par sa lecture de mon manuscrit ;

Jullien Bouchet, pour l'étude de cas qu'il m'a fournie et qui est utilisée dans ce travail ;

Mme Lydie Du-Bousquet et Mme Sophie Dupuy-Chessa pour leurs conseils qui ont alimenté les réflexions de ce travail ;

Dima, Amal, Virginia, Abdesselam, Besnik et tous mes amis ;

Mes parents, Hakam, Ayah, Hebah et Eyas et toute ma famille pour leur soutien continu et toute leur affection.

Utilisation de la programmation synchrone pour la spécification et la validation de services interactifs

Résumé

Ce travail porte sur le test automatique de systèmes interactifs. L'approche proposée est basée sur des techniques de test de systèmes réactifs *synchrones*. Le comportement de systèmes synchrones, qui est constitué de cycles commençant par la lecture d'une entrée et finissant par l'émission d'une sortie, est sous certaines conditions, similaire à celui de systèmes interactifs. En particulier, nous étudions l'utilisation de Lutess, un environnement de test synchrone, pour valider les systèmes interactifs.

Nous montrons l'intérêt d'utiliser les techniques de test proposées par Lutess afin de générer des scénarios intéressants et nous illustrons leur utilisation sur une étude de cas (une application de réalité virtuelle mobile).

Nous avons également étudié la génération de données de test à partir d'arbres de tâches, qui peuvent être enrichis d'une spécification de profils opérationnels.

L'adaptation des techniques de test synchrone à la validation d'applications interactives multimodales est également étudiée, notamment en prenant en compte certaines propriétés concernant la multimodalité.

Using synchronous programming for specification and validation of interactive services

Abstract

This work deals with the automatic testing of interactive systems. The proposed approach is based on testing techniques for synchronous reactive software. The behaviour of synchronous systems, consisting of cycles starting by reading an input and ending by issuing an output, is to a certain extent similar to the one of interactive systems. In particular, we are using Lutess, a synchronous testing environment, to validate interactive systems.

We show the interest of applying the testing techniques proposed by Lutess in order to generate interesting scenarios and we illustrate their use on a case study (a mobile virtual reality application).

We also study the generation of test data from task trees, which can be enriched by operational profiles specification.

The adaptation of synchronous testing techniques for the validation of multimodal interactive systems is also studied, taking into account multimodality-related properties.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Contexte et motivation

Les applications interactives sont aujourd'hui présentes dans plusieurs domaines. Elles assurent l'accès à des services commerciaux divers comme les téléphones mobiles, les systèmes de réservation ou les services de télécommunication. Elles sont également présentes dans des systèmes critiques comme le contrôle de vol ou le contrôle de processus industriels.

Les systèmes interactifs deviennent de plus en plus puissants et le nombre de fonctions qu'ils offrent s'accroît sans cesse. En conséquence, ils sont de plus en plus complexes : d'une part la taille de systèmes à interfacer augmente, d'autre part l'évolution technologique introduit de nouvelles techniques d'interaction. Ainsi, les systèmes interactifs s'orientent actuellement vers l'utilisation de multiples modalités, c'est-à-dire de plusieurs moyens d'interaction, telles que les touches de clavier, les boutons de souris, la voix et le geste, permettant une affinité plus naturelle entre l'humain et le système, comme on peut le voir dans les applications mobiles de réalité virtuelle.

La complexification des systèmes interactifs augmente le risque d'erreurs qui peuvent être introduites pendant leur développement. Ces erreurs peuvent être liées aux aspects fonctionnels du système mais aussi diminuer son utilisabilité, qualité majeure des systèmes interactifs. En conséquence, leur développement requiert une validation rigoureuse.

Plusieurs méthodes ont été proposées pour valider et évaluer les systèmes

interactifs. Certaines font appel à des utilisateurs réels [1, 2] qui effectuent un ensemble prédéterminé de tâches pendant qu'un testeur enregistre leurs résultats. Ce dernier détermine si l'interface assure la complétion de tâches et enregistre le nombre d'erreurs. D'autres méthodes utilisent des retours de l'utilisateur sur l'interface par le biais d'entretiens, d'enquêtes et de surveillance, comme la méthode UPM [3] qui lorsqu'il se produit une erreur, déclenche un événement ("trigger"), afin de poser des questions sur l'utilisation de l'interface. Dans d'autres méthodes, l'évaluateur examine l'interface par rapport à un ensemble de critères et d'heuristiques prédéfinies, comme la méthode WebEval [4].

L'utilisation de méthodes formelles a également été suggérée pour la validation et la vérification de systèmes interactifs. Ces méthodes utilisent un modèle de l'application interactive ou de son interface. Ce modèle est spécifié avec un langage formel et les propriétés demandées sont vérifiées sur ce modèle. On peut citer parmi ces méthodes et de manière non exhaustive les approches LIM (Lotos Interactor Model) [5, 6], ICO (Interactive Cooperative Object) [7, 8, 9, 10, 11] qui est basée sur les réseaux de Petri et Lustre [12, 13, 14].

La méthode B a également été proposée dans ce cadre [15, 16, 17] ainsi que Z [18]. À partir d'un modèle abstrait de l'application, des raffinements et preuves successifs peuvent avoir lieu, enrichissant le modèle avec de nouvelles informations et de nouvelles propriétés. Après un nombre fini d'étapes, le processus de raffinement conduit à un modèle concret qui peut être implémenté. Ces approches permettent de vérifier les propriétés à différents niveaux d'abstraction.

Les méthodes formelles sont principalement utilisées pendant la phase de conception. Elles modélisent (le plus souvent partiellement) le système et prouvent des propriétés sur le modèle sans toutefois offrir de garantie de l'équivalence entre ce modèle et le système final. Pour ces raisons, l'étape de test reste un enjeu très important pour le développement des systèmes interactifs et complète les approches formelles de vérification.

Des méthodes de test automatique ont été proposées pour les systèmes interactifs. Ces méthodes utilisent un modèle pour générer les tests, comme la méthode proposée dans [19] qui utilise un modèle de tâches simplifié. La méthode de [20] s'appuie sur un modèle d'état finis à variables tandis qu'avec la méthode de [21] l'interface est modélisée au moyen d'un ensemble d'opérateurs hiérarchiques où, pour chaque opérateur, une précondition et une postcondition

doivent être définies.

Le travail de cette thèse s'inscrit dans le cadre du test automatique de systèmes interactifs et s'intéresse à l'étude de deux points :

- la génération automatique de tests
- et la validation automatique de propriétés.

Plus précisément, nous nous intéressons à l'application de techniques de test de logiciels réactifs *synchrones* (spécifiés dans un langage synchrone) à la validation de systèmes interactifs. Cet intérêt se justifie par les similitudes entre les deux types de systèmes. En effet, leurs comportements sont constitués de cycles "action-réaction" : les systèmes réactifs réagissent continûment à leur environnement physique à la vitesse de ce dernier tandis que les systèmes interactifs interagissent avec leur environnement, mais à leur vitesse propre (comme, par exemple, un système d'exploitation).

L'approche synchrone a déjà été utilisée pour la vérification de systèmes interactifs. Par exemple, Lustre a été proposée à des fins de spécification et vérification formelles d'interfaces graphiques [12, 13, 14], et la spécification, dans ce même langage, de services de télécommunication a servi à la validation de ces services [22].

L'approche synchrone a initialement été introduite pour la conception et l'implantation de systèmes réactifs. Les langages synchrones permettent de raisonner comme si le temps de réaction aux événements externes était nul (**hypothèse de synchronisme**), ce qui, en pratique, revient à s'assurer que le programme réagit assez vite pour percevoir toute évolution de son environnement. L'approche synchrone permet ainsi une modélisation plus simple de l'interaction entre un système interactif et son environnement, modélisée par des successions de cycles d'exécution (un cycle d'exécution est constitué d'une action et d'une réaction). En se basant sur cette modélisation, des propriétés peuvent aussi être facilement modélisées ce qui permet de procéder à une validation efficace.

Nous nous sommes plus particulièrement intéressés à l'outil de test Lutess. Cet outil requiert une spécification partielle du comportement de l'utilisateur de l'application écrite dans le langage synchrone Lustre, enrichie de directives de guidage. Nous avons ainsi étudié les moyens d'exprimer dans ce langage les spécifications nécessaires à la génération de tests de systèmes interactifs.

Cependant, ce modèle synchrone du comportement de l'utilisateur n'est pas

facile à construire pour les concepteurs des applications interactives, non familiers de ces langages. Pour cette raison, nous avons également étudié des méthodes de génération de données de test basées sur des modèles qui sont plus habituels dans le processus de développement des applications interactives, les arbres de tâches. Ces derniers décrivent les interactions entre l'application et l'utilisateur.

Nous nous sommes, enfin, intéressés à la validation d'applications multimodales. Un système multimodal dispose de plusieurs modalités (techniques d'interaction). Selon [23] une modalité est un couple constitué d'un dispositif physique et d'un langage d'interaction. Des exemples de modalités sont <microphone, pseudo langage naturel>, <souris, manipulation directe> ou <GPS, localisation en données GPS>. Dans un système interactif multimodal, plusieurs modalités peuvent être utilisées de manière indépendante ou combinée [24]. L'utilisation combinée de modalités est restreinte par des contraintes temporelles. La manière d'utiliser des modalités est caractérisée par quatre propriétés, appelées CARE [25, 26, 27] (Complémentarité, Assignation, Redondance et Équivalence). Nous avons ainsi étudié les moyens d'adapter les techniques de test synchrone à la validation de ce type d'applications en prenant en compte les propriétés CARE.

1.2 Contributions de la thèse

Test synchrone de systèmes interactifs

Nous montrons qu'il est possible de tester une application interactive au moyen d'un environnement de test de programmes synchrones comme Lutess [28]. En particulier, nous montrons que la connexion entre Lutess et un système interactif asynchrone est possible en introduisant un traducteur qui prend en charge la désynchronisation des événements d'entrée pour le système interactif et la synchronisation des événements de sortie de ce système.

Nous montrons également l'intérêt d'utiliser les techniques de test proposées par Lutess afin de générer des scénarios intéressants et nous illustrons leur utilisation sur une étude de cas (une application du domaine de réalité virtuelle mobile).

L'approche de test résultante est complètement automatique, car un oracle automatique, contenant les propriétés à valider par le système sous test, observe

les entrées et les sorties et produit des verdicts.

Génération de tests à partir d'arbres de tâches

Afin de rendre plus facile l'utilisation d'un outil comme Lutess, nous proposons une méthode permettant la génération automatique des données de test à partir d'arbres de tâches. L'arbre de tâches décrit l'interaction entre l'application interactive et l'utilisateur. Il comporte ainsi des informations sur le comportement de ce dernier. L'arbre de tâches est automatiquement analysé et le comportement de l'utilisateur est extrait sous la forme d'une machine d'états finis synchrone à entrées-sorties. Cette dernière est construite conformément à une sémantique formelle, définie à cet effet pour tous les opérateurs de l'arbre de tâches. Ce modèle est ensuite utilisé pour générer des tests simulant le comportement de l'utilisateur.

Nous avons également défini une méthode pour spécifier un profil opérationnel de l'utilisateur du système interactif. Spécifier un profil opérationnel consiste à affecter des probabilités à certains opérateurs de l'arbre de tâche. Ces probabilités sont traduites en des probabilités sur les transitions de l'automate synchrone associé. Ainsi, on obtient un automate probabiliste qui correspond à un profil opérationnel de l'utilisateur du système interactif. Cet automate est utilisé pour générer les tests.

Test synchrone de systèmes interactifs multimodaux

La connexion entre un système multimodal et l'outil de test synchrone Lutess nécessite la présence d'un traducteur qui se charge de la synchronisation des événements pour les modalités d'entrée. Nous montrons également comment Lutess permet de valider automatiquement la fusion entre les événements de différentes modalités en cas d'utilisation combinée de ces dernières. Cette validation automatique concerne la génération d'événements de différentes modalités proches temporellement et la construction d'un oracle automatique contenant les propriétés qui portent sur l'utilisation des modalités (propriétés CARE).

1.3 Organisation du document

Cette thèse est organisé en 2 parties. La première partie est dédiée à la validation de systèmes interactifs au moyen de l'approche synchrone. Dans cette partie, le chapitre 2 présente le domaine des systèmes interactifs et l'état de l'art sur leur modélisation et leur vérication. Le chapitre 3 présente l'outil Lutess et l'approche de test synchrone. Le test de systèmes interactifs avec l'approche synchrone est abordé dans le chapitre 4. Le chapitre 5 propose d'utiliser des arbres de tâches pour générer des tests. Nous définissons également dans ce chapitre des moyens de spécifier un profil opérationnel de l'utilisateur sur un arbre de tâches. La deuxième partie est dédiée à la validation de systèmes interactifs multimodaux. Les grandes caractéristiques de ces systèmes et l'état de l'art sur leur modélisation et leur validation sont présentés dans le chapitre 6. Le chapitre 7 présente notre proposition fondée sur l'utilisation de techniques de test synchrone pour ces systèmes. Dans le chapitre 8, nous donnons des perspectives et esquissons des axes de travail futur.

Première partie

Validation de systèmes interactifs

Chapitre 2

Modélisation et validation des systèmes interactifs

2.1 Introduction

Les systèmes interactifs deviennent de plus en plus puissants et ainsi de plus en plus complexes. Ils s'orientent actuellement vers l'utilisation de multiples modalités d'interaction telles que les boutons de souris, la voix et le geste, permettant une affinité plus naturelle entre l'homme et le système. La complexification de ces systèmes se traduit par une augmentation de la taille et de la complexité du code et augmente le risque d'erreurs pouvant être introduites pendant les différents étapes de développement. En conséquence, leur correction devient un enjeu très important et leur développement requiert une validation rigoureuse.

Les exigences attendues des systèmes interactifs s'expriment souvent en termes d'*utilisabilité*. L'utilisabilité d'un système interactif, selon Coutaz, caractérise sa capacité à permettre à l'utilisateur d'atteindre ses objectifs (obtenir un résultat correct ayant une qualité donnée) avec efficacité, en tout confort et sécurité. Elle est déterminée par deux notions principales : la **souplesse** et la **robustesse** de l'interaction [29]. Chacune de ces deux notions est définie par plusieurs propriétés.

La souplesse de l'interaction représente le degré des possibilités de choix offertes aussi bien à l'utilisateur qu'au système. Elle regroupe un ensemble de

propriétés dont :

- *Atteignabilité* : désigne la capacité du système à offrir à l'utilisateur la possibilité de naviguer dans l'ensemble des états observables du système. Formellement, un état q est atteignable à partir d'un état p s'il existe une suite de commandes $\{c_i\}$ qui permettent de passer de l'état p à l'état q. La longueur de la trajectoire d'interaction nécessaire pour passer de l'état p à l'état q offre une mesure de l'atteignabilité.
- *Interaction multifilaire (multithreading)* : désigne la capacité du système à permettre la réalisation de plusieurs tâches de manière entrelacée (par exemple édition de plusieurs documents à la fois). Le nombre de tâches que l'on peut mener de manière entrelacée ou parallèle en relation avec la surcharge cognitive de l'utilisateur permet de mesurer le degré d'interaction multifilaire offerte par un système interactif. Le parallélisme ou l'entrelacement des interactions peut être analysé à différents niveaux de granularité : niveau tâches, niveau actions (propriétés CARE - voir le chapitre 6), etc.
- *Multiplicité du rendu (représentation multiple d'un même concept)* : fait référence à la capacité du système à fournir plusieurs représentations pour un même concept. Par exemple, le concept de température peut être présenté sous forme d'un entier ou sous une forme analogique par l'intermédiaire d'un thermomètre.

La robustesse de l'interaction a pour objectif de prévenir les erreurs et d'augmenter les chances de succès de l'utilisateur. Elle est définie par un ensemble de propriétés dont :

- *Observabilité* : représente la capacité du système à rendre perceptible par l'utilisateur l'état pertinent du système. Elle représente par conséquent également la capacité pour l'utilisateur d'évaluer l'état actuel du système.
- *Insistance* : représente la capacité du système à forcer la perception de son état (que l'utilisateur devra percevoir).
- *Honnêteté* : désigne la capacité du système à rendre observable l'état du système sous une forme conforme à cet état et qui engendre une interprétation correcte de la part de l'utilisateur. Selon Abowd [18], cette propériété assure qu'à deux états internes distincts d'un objet correspondent des visualisations distinctes.

- *Curabilité* : indique la capacité pour l'utilisateur de corriger une situation non désirée.
- *Prévisibilité* : implique pour l'utilisateur la capacité de prévoir, pour un état donné, l'effet d'une action. Cette prévisibilité peut être obtenue grâce à la cohérence (conformité aux règles/usages). Il convient cependant de prendre en compte le fait que les règles/usages de l'utilisateur ne sont pas nécessairement celles du concepteur. La cohérence peut en général elle même s'obtenir par conformité à des normes d'IHM et par conformité à l'expérience de l'utilisateur dans le monde réel.

Plusieurs méthodes ont été réalisées pour valider et évaluer les systèmes interactifs et/ou leurs interfaces. Dans [30] ces méthodes sont classées comme suit :

- **L'évaluation expérimentale** : un évaluateur utilise des participants réels [1, 2] qui utilisent le système ou un prototype pour effectuer un ensemble prédéterminé de tâches pendant qu'un testeur enregistre leurs résultats de travail. Puis, le testeur utilise ces résultats pour déterminer comment l'interface assure la complétion de tâches, ainsi que d'autres mesures comme le nombre d'erreurs et le temps de la complétion d'une tâche.
- **L'inspection** : un évaluateur examine les aspects d'utilisabilité par rapport à un ensemble de critères et d'heuristiques, pour identifier les problèmes potentiels d'utilisabilité.
- **L'interrogatoire/entretien** : l'utilisateur émet des retours sur l'interface par le biais d'entretiens et de surveillances. L'évaluateur réalise des enquêtes pour obtenir des données supplémentaires après la réalisation du système pour définir des améliorations à apporter lors de réalisations futures.
- **L'analyse de modèles** : un évaluateur utilise des modèles de l'utilisateur et/ou de l'interface pour générer des prédictions d'utilisabilité.
- **La simulation** : un évaluateur utilise des modèles de l'utilisateur et/ou de l'interface pour imiter le comportement interactif des utilisateurs avec le système, et rapporte ainsi les résultats de l'interaction.

Dans cette thèse, on s'intéresse aux méthodes formelles pour l'évaluation des systèmes interactifs et de leurs interfaces. Ces méthodes utilisent des modèles du système et/ou de l'utilisateur pour **l'analyse** ou/et la **simulation**. Certaines d'entre elles sont utilisées pendant la phase de conception et d'autres sont appli-

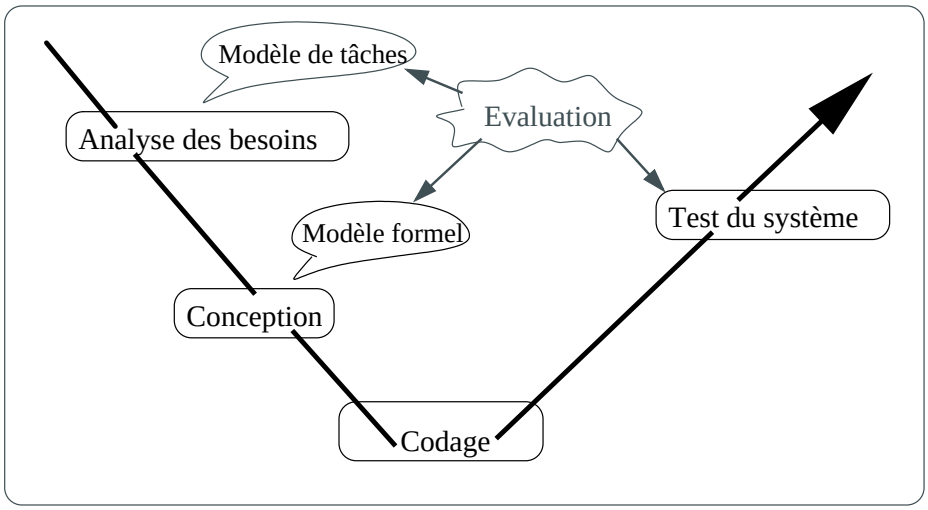

FIG. 2.1 – Le processus de développemnt des systèmes interactifs

quées après l'implémentation.

La figure 2.1 montre les étapes principales du processus du dévéloppement de systèmes interactifs [31, 32] .

Dans ce processus, on peut distinguer les étapes suivantes :

1. Analyse des besoins : dans cette étape, un modèle de tâches est spécifié ;
2. Conception : dans cette étape, des méthodes formelles peuvent être utilisées pour spécifier formellement l'application ou son interface et vérifier les propriétés sur la spécification ;
3. Codage : implémentation de l'application ;
4. Test et validation de l'application : dans cette étape, le test formel peut être utilisé pour valider les résultats attendus de l'application. La génération de tests est effectuée à partir d'un modèle spécifiant l'interaction. Ce modèle peut être une spécification formelle de l'application ou une déscription du comportement de l'utilisateur comme le modèle de tâches.

Dans ce chapitre, nous allons d'abord présenter des modèles et des méthodes utilisés pour la conception d'applications interactives (cf. paragraphe 2.2). Ensuite, nous allons présenter des méthodes de test pour ces applications (cf. paragraphe 2.3)

2.2 Modèles et méthodes de conception d'applications interactives

2.2.1 Modèles de tâches

Une tâche est un but que l'utilisateur vise à atteindre à l'aide d'un système interactif [33]. Ce but est assorti d'une procédure (ou plan) qui décrit les moyens pour l'atteindre [33].

Un modèle de tâches décrit l'interaction entre l'utilisateur et le système interactif en termes de tâches et sous-tâches. Dans ce modèle, les tâches sont représentées d'une manière hiérarchique [34] : une tâche est composée de sous-tâches liées par des opérateurs temporels. Cela signifie que le modèle de tâches fournit des informations sur les sous-tâches qui doivent être exécutées afin d'accomplir une autre tâche plus complexe, ainsi que des informations sur le comportement du système interactif : la structure hiérarchique d'une tâche montre les sous-tâches à exécuter afin de réaliser cette tâche (*what to do*) tandis que la représentation comportementale prend en compte les conditions pour réaliser la sous-tâche (*when to do things*) [34].

Plusieurs modèles de tâches ont été proposés : JSD (Jackson System Development) [35, 32], HTA (Hierarchical Task Analysis) [36], MAD (Méthode Analytique de Description de tâches)[37, 32], UAN (User Action Notation) [38, 39], CTT (Concur Task Tree) [40, 41] ... etc. Les opérateurs communs entre ces modèles sont la séquence, le choix et l'itération.

Parmi ces modèles, nous présentons CTT (*ConcurTaskTrees*) [40, 42, 41], une notation bien connue, intégrée dans un éditeur graphique qui permet de créer et sauvegarder un modèle en plusieurs formats. Nous nous sommes appuyés sur cette notation (cf. chapitre 5) pour proposer des techniques de génération de tests à partir de modèles de tâches.

2.2.1.1 Concur Task Trees CTT

CTT distingue quatre types de tâches :

1. Tâche usager : activité cognitive interne sans interaction avec le système, comme la réflexion pour résoudre un problème, la planification, la lecture

d'un message, etc...

2. Tâche application : réalisation par le système, comme l'affichage du résultat d'une requête, la production d'alerte, etc...
3. Tâche interactive : action usager avec un feedback immédiat par le système, comme l'édition d'un document.
4. Tâche abstraite : tâche composée d'autres sous-tâches.

Une tâche abstraite est composée de sous-tâches liées par des opérateurs temporels présentés ci-dessous :

1. **Activation (T1 >> T2)** : La tâche T2 est activée par l'exécution de la tâche T1.
2. **Activation avec passage d'information (T1 []>> T2)** : L'exécution de T1 active T2 en lui fournissant des informations.
3. **Choix (T1 [] T2)** : Une seule des deux tâches est exécutée, T1 ou T2.
4. **Entrelacement (Concurrence indépendante) (T1 ||| T2)** : Les actions des tâches T1 et T2 peuvent être effectuées dans un ordre quelconque, sans contrainte.
5. **Concurrence avec échange d'information (T1 |[]| T2)** : Les actions des tâches T1 et T2 peuvent être effectuées dans un ordre quelconque, mais elles doivent se synchroniser pour échanger de l'information.
6. **Désactivation (T1 [> T2)** : T1 est désactivée lorsque la première action de T2 est effectuée.
7. **Suspendre-reprendre (T1 |> T2)** : T2 peut interrompre l'exécution de T1 et lorsque elle est terminé, T1 reprend dans l'état ou elle a été interrompue.
8. **Itération (T1*)** : T1 est exécutée de manière itérative. L'itération continue jusqu'à ce qu'une autre tâche désactive la tâche itérative.
9. **Itération finie (T1(n))** : T1 doit être exécutée n fois.
10. **Tâche optionnelle ([T1])** : L'exécution de T1 est optionnelle.

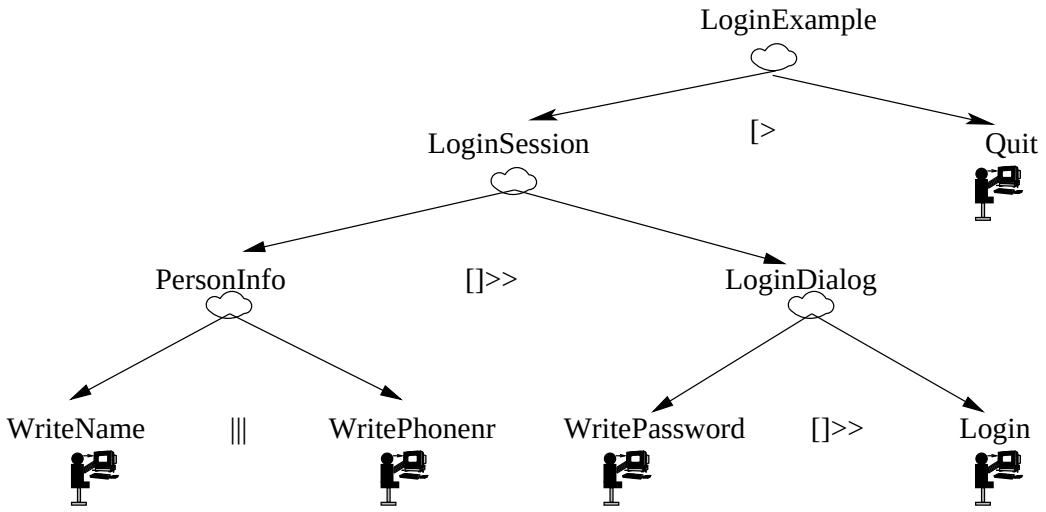

FIG. 2.2 – Exemple d'arbre de tâches CTT

Exemple

La figure 2.2 montre un exemple d'arbre de tâches dans la notation CTT. Dans cet arbre, la tâche "Quit" désactive la tache "LoginSession". "LoginSession" est une tâche abstraite (composée de sous-tâches). Cela signifie qu'après avoir realisé la tâche "Quit", on ne peut exécuter aucune autre action de la tâche "LoginSession". La tâche "PersonInfo" active la tâche "LoginDialog" et lui passe des informations. La tâche "PersonInfo" est composée de l'entrelacement entre les deux tâches "WriteName" et "WritePhonenr". La tâche "LoginDialog" est composée des tâches "WritePassword" et "Login". "WritePassword" doit être exécutée avant "Login" et elle lui fournit des informations.

2.2.2 L'approche LOTOS

La modélisation de l'interface avec LOTOS s'appuie sur le concept d'interacteur [5] pour structurer la description des interfaces. La figure 2.3 montre l'architecture d'un interacteur. Ce dernier est un processus qui décrit l'interaction comme une composition des quatre processus plus élémentaires :

1. *collection*, qui contient la description abstraite de l'apparence de l'interacteur. Sa fonctionnalité est d'interpréter la description abstraite des sorties

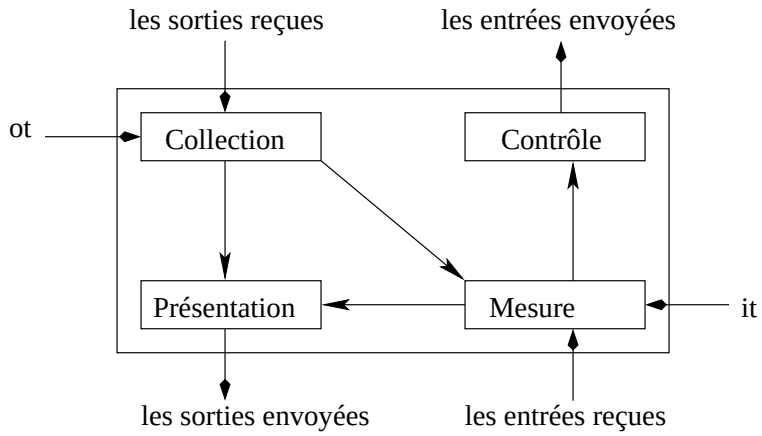

FIG. 2.3 – L'architecture d'un interacteur

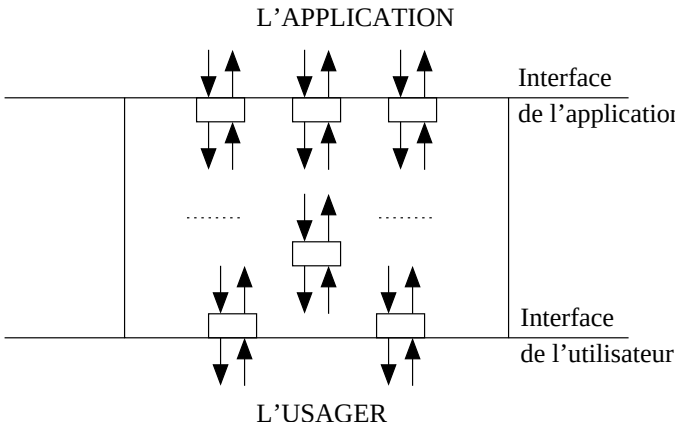

FIG. 2.4 – L'architecture d'interface Usager Système

reçues et de les transmettre à la *présentation* et à la *mesure* à l'arrivée du signal *ot* ;

2. *présentation*, qui produit l'apparence externe de l'interacteur ;
3. *mesure*, qui construit des données d'entrée plus abstraites et les transmet au *contrôle* à l'arrivée du signal *it* ;
4. *contrôle*, qui délivre les données d'entrée aux autres interacteurs ou à l'application.

L'interface est une composition d'interacteurs qui fonctionnent en parallèle [5, 39, 6]. Ces interacteurs échangent des événements pendant leur fonctionnement : les événements de sortie d'interacteur peuvent être les événement d'entrée

d'autres interacteurs (voir la figure 2.4). Cette vision permet de modéliser l'interface comme un réseau de processus communicants et parallèles, de la même manière que pour les systèmes concurrents. Ainsi, il a été proposé de modéliser l'interface avec un langage utilisé pour décrire les systèmes concurrents comme LOTOS.

Le modèle décrit dans le langage LOTOS sert à vérifier les propriétés d'interaction comme l'atteignabilité, réactivité, conformité. Les propriétés [43, 6] sont exprimées dans la logique ACTL (Action-Based Temporal Logic) et sont vérifiées par *model-checking* sur un modèle d'automate généré à partir de la spécification de l'interface en LOTOS.

Un des inconvénients de l'approche est que le modèle LOTOS est difficilement compréhensible. Ce modèle reflète bien l'architecture de l'interface mais pas le comportement dynamique de l'interaction (l'ensemble d'événements générés par l'utilisateur pour accéder aux fonctionnalités de l'application). C'est pourquoi, dans [44] il est proposé d'exprimer cette dynamique dans la notation CTT ConcurrTaskTree [40]. Ensuite, CTT est transformé automatiquement en spécification LOTOS qui sert à la vérification de l'interface.

2.2.3 ICO (Interactive Cooperative Objects)

Cette approche [7, 8, 9, 10, 11] est fondée sur les réseaux de Petri et le concept d'objet. Elle permet la spécification et la validation formelles d'une application interactive. La spécification formelle peut être utilisée pour la conception de l'application interactive finale.

La spécification formelle est réalisée dans un formalisme appelé "Objets Coopératifs Interactifs" ou ICO (en anglais "Interactive Cooperative Objects"). C'est un langage orienté objet où le comportement des objets est décrit par un réseau de Petri de haut niveau.

Le formalisme des Objets Coopératifs Interactifs

Un modèle ICO est une description formelle d'un système construit de plusieurs objets communicants. Le comportement des objets et leur communication sont décrits par des réseaux de Petri. Quand deux objets communiquent, l'un (le client) demande un service et l'autre (le serveur) exécute le service. Dans

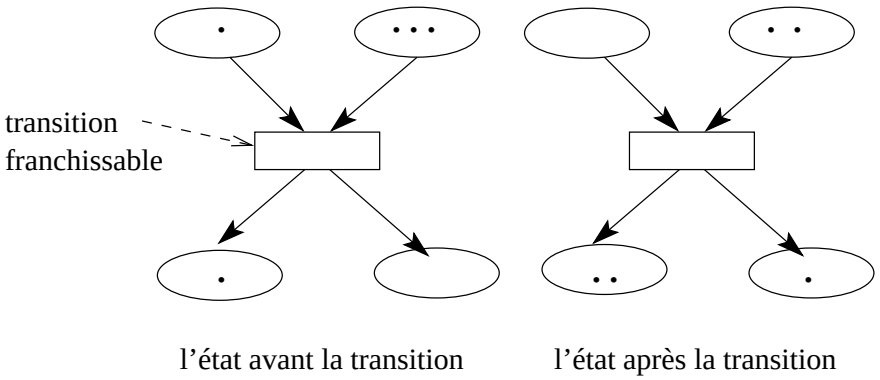

FIG. 2.5 – Un exemple de Réseau de Petri

le formalisme ICO, un objet est une entité décrite par quatre composants : le comportement, les services, l'état et la présentation.

Comportement : Le comportement d'un ICO décrit comment un objet réagit aux stimuli extérieurs en fonction de son état interne. Ce comportement est décrit par un réseau de Petri de haut niveau. Ce dernier est un graphe dont les nœuds sont soit des places (représentées graphiquement par des ellipses), soit des transitions (représentées par des rectangles). Places et transitions sont connectées par des arcs. Chaque place peut contenir un nombre quelconque de jetons. Les jetons peuvent porter des valeurs (string, integer, ou références aux autres objets). Pour que la transition soit franchissable, il faut que chacune de ses places d'entrée contienne au moins un jeton et que la précondition (si elle existe) soit vérifiée. L'occurrence d'une transition retire un jeton de chacune des places d'entrée et dépose un jeton dans chacune des places de sortie (figure 2.5).

Services : Un ICO offre un ensemble de services qui définissent l'interface de l'objet avec son environnement. L'environnement peut être les usagers ou d'autres objets de l'application. Chaque service est lié au moins à une transition et un service est disponible quand au moins une de ses transitions est franchissable. On distingue deux sortes de services : les services utilisateur (dans ce cas, l'environnement correspond aux usagers) et les services aux autres objets.

État : L'état d'un ICO est la distribution et les valeurs des jetons (appelé le marquage) dans les places du réseau de Petri correspondant (figure 2.5).

Présentation : La présentation d'un objet est son apparence externe. C'est un ensemble d'éléments d'interface (« widgets ») organisé dans un ensemble de fenêtres.

L'interaction usager→système a lieu uniquement au travers de ces widgets. Chaque action de l'utilisateur sur un widget peut déclencher un service utilisateur du modèle ICO. La relation entre services utilisateur et widgets est entièrement décrite par la *fonction d'activation* qui associe à chaque couple (widget, action de l'utilisateur) un service utilisateur :

$Act : Widgets \times Events \longrightarrow Services$

L'interaction système→usager est spécifiée par la *fonction de rendu* qui associe à chaque nœud (places ou transitions) un ensemble de widgets qui peuvent être utilisés pour rendre l'information à l'utilisateur :

$Rend : P \cup T \longrightarrow \mathcal{P}(Widgets)$

En général, le rendu est soit sur les places, soit sur les transitions [7].

Le rendu est sur les places quand le jeton entre (*TokenEntered*), sort (*TokenRemoved*) ou est accédé ou échangé par une transition bidirectionnelle (*TokenReset*) :

$Rend : P \times \{TokenEntered, TokenRemoved, TokenReset\} \longrightarrow \mathcal{P}(Widgets)$

Le rendu sur les transitions, dans la plupart de cas, aura lieu quand une action sur la transition commence (change la forme du curseur) et quand l'action se complète (retour dans la forme initiale du curseur).

L'application interactive est spécifiée formellement avec plusieurs objets coopératifs interactifs [9, 10]. Par exemple, si l'application est conçue selon l'architecture MVC (Modèle, Contrôleur, Vue), il y a trois sortes d'ICO [10] : le modèle, le contrôleur et la vue. Dans la classe ICO Modèle, les fonctions d'activation et de rendu sont vides. Dans la classe ICO Contrôleur, la fonction de rendu est vide. Dans la classe ICO Vue, la fonction d'activation est vide.

Ensuite, cette spécification formelle peut être exécutée, testée et vérifiée en utilisant les outils relatifs aux réseaux de Petri.

2.2.4 La méthode B

Dans [15, 16, 17], il est proposé d'utiliser la méthode B pour valider les tâches de l'utilisateur de systèmes interactifs. Dans ce cas, il s'agit de la validation des besoins de l'utilisateur à la phase de conception et spécification. Cette approche utilise le modèle de tâches CTT. Une tâche se décompose en sous tâches liées par les opérateurs CTT, elles-mêmes décomposées en d'autres sous-tâches...etc. On obtient ainsi un arbre de tâches.

Cette méthode est également proposée pour valider les systèmes interactifs critiques [45, 46]. Dans ce cas, la méthode B est utilisée à partir de la phase de spécification et conception et, éventuellement, jusqu'à l'étape de programmation.

Plus précisément, le modèle B [47] événementiel est proposé. Ce modèle est composé d'un ensemble d'événements atomiques décrits par des substitutions particulières (assignation, ANY, BEGIN et SELECT). Chaque événement E_v est déclenché si la garde P associée à cet événement est vraie. De plus, le modèle B contient un ensemble de propriétés (invariants, propriétés de sûreté, vivacité et atteignabilité) qui peuvent être validées pendant le développement grâce aux techniques de preuves associées à B. Finalement, les modèles peuvent être raffinés vers d'autres modèles B qui peuvent être enrichis par de nouveaux événements et de nouvelles propriétés. Le processus de raffinement conduit à la conception de l'interface de l'utilisateur après un nombre fini d'étapes de raffinement qui correspondent à différents niveaux d'abstraction.

Validation de tâches L'approche de validation proposée dans [15, 16, 17] consiste à décrire tous les opérateurs CTT avec le modèle B événementiel. Chaque décomposition d'une tâche supérieure dans l'arbre CTT correspond à un raffinement d'un événement B. Une tâche est décrite par un état initial et un état final et est raffinée par une séquence d'événements atomiques qui conduisent de l'état initial à l'état final. Le raffinement préserve toutes les propriétés de la tâche initiale. Ce processus est répété jusqu'aux événements atomiques. Quand tous les événements élémentaires du contrôleur de dialogue sont atteints par le raffi-

nement, le processus de la validation est terminé. Cette approche de validation présente deux intérêts. En premier lieu, la séquence des événements montre qu'il existe une séquence d'éléments simples pour implémenter une tâche abstraite, ce qui permet la validation de la tâche. En second lieu, si un ou plusieurs éléments simples sont manquants et/ou les obligations de preuve relatives aux événements élémentaires ne peuvent pas être prouvées, cela montre que la conception doit être modifiée ou complétée. Il s'agit donc d'une validation de la conception et de la décomposition architecturale.

Validation de systèmes interactifs critiques L'utilisation de la méthode B a également été proposée pour la vérification de la sûreté et l'utilisabilité de systèmes critiques interactifs [45, 46, 48]. Cette technique utilise un modèle orienté B et s'appuie sur la nouvelle architecture CAV (Contrôle, Abstraction, Vue), qui est un modèle hybride de deux modèles d'architecture de logiciel : MVC et PAC. Cette méthode peut garantir l'utilisabilité et la sûreté, en utilisant l'obligation de preuve, à partir de l'étape de spécification pendant le processus de développement, jusqu'à l'implémentation.

Pendant l'étape de la conception et spécification, l'application est modélisée selon l'architecture CAV. Chaque composant de l'architecture est modélisé avec une ou plusieurs machines abstraites du modèle B. Les besoins de l'application sont également identifiés. Ces besoins sont traduits en des propriétés qui doivent être vérifiées comme des invariants dans la spécification formelle. Cette dernière peut être modifiée pour vérifier ces invariants.

Ensuite, les machines abstraites sont raffinées en des machines IMPLEMENTATION programmées en pseudo-code BØ. Ces dernières peuvent être automatiquement transformées en code C. Les opérations de la machine IMPLEMENTATION respectent les invariants comme dans la machine abstraite. Des invariants entre les différentes machines de développement sont définis et des ensembles d'obligations de preuve sont générées. Ce code, complètement généré à l'aide d'une méthode formelle, est qualifié de sûr. Cependant, toute l'application ne peut pas être développée complètement de manière formelle, parce que les librairies de widgets ne sont pas développées avec des méthodes formelles. Les modules qui ne sont pas liés à la sécurité peuvent être validés avec le test.

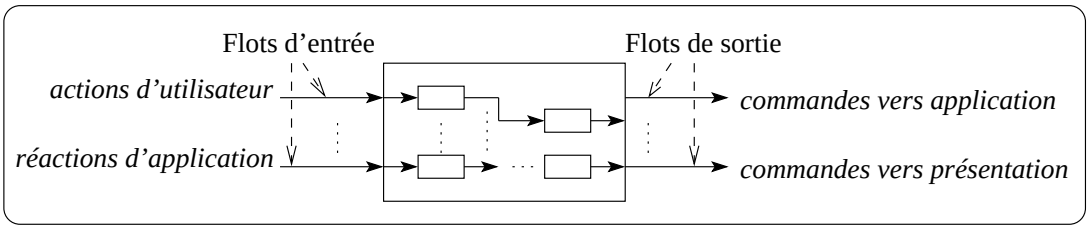

FIG. 2.6 – Interacteur à flots de données

2.2.5 L'approche Lustre

Cette approche [12, 13, 14] s'appuie sur le concept d'interacteur. L'interface est modélisée par un réseau d'interacteurs [49]. Un interacteur est un automate qui réagit à des actions d'entrée en modifiant son état et en générant des événements de sortie. Les actions d'entrée sont des actions de l'utilisateur (par exemple enfoncement d'une touche) et des réactions générées par l'application. Les événements de sortie sont des actions provenant de l'interacteur vers l'application et sa présentation. Ces actions correspondent à des commandes d'activation de traitements de données ou des commandes de visualisation. Un réseau d'interacteurs définit un interacteur plus complexe et se construit par l'intermédiaire d'un opérateur de composition parallèle des interacteurs. Un tel modèle d'interacteur décrit en fait un réseau de processus parallèles et communicants : les processus sont des automates et la communication entre les automates est assurée par la connexion des sorties de certains automates aux entrées d'autres automates.

Cette approche de vérification d'IHM adopte le formalisme Lustre pour la modélisation et la validation des interfaces Homme Machine. Le langage Lustre [50] est fondé sur le modèle à *flots de données*. Ce dernier représente un système à travers un réseau d'opérateurs agissant en parallèle sur leurs entrées. Dès que les entrées nécessaires à un opérateur sont disponibles, ce dernier calcule les sorties correspondantes. Les sorties des opérateurs peuvent être connectées aux entrées d'autres opérateurs. Les suites d'événements d'entrée et de sorties des opérateurs sont appelées « flots ».

Le modèle générique utilisé dans cette approche représente un interacteur comme un réseau d'opérateurs sur flots booléens (figure 2.6). Les flots d'entrée des opérateurs sont associés aux actions de l'utilisateur et des réactions générées par l'application. Les flots de sortie sont associés aux actions en direction de

l'application et de la présentation.

Le comportement des interacteurs en Lustre s'exprime sous la forme de dépendances entre flots de sortie et flots d'entrée. Lustre permet également la description modulaire des interacteurs (sous forme de nœuds).

L'approche a été appliquée à la production de spécifications d'interfaces à fenêtres. Le modèle Lustre de l'interface est extrait automatiquement à partir de la description de l'interface dans les langages UIL et C. Ce modèle est utilisé pour la vérification de propriétés à satisfaire par l'interface [51] au moyen de l'outil de model-checking Lesar [52]. L'analyse de propriétés sur le modèle Lustre permet d'améliorer la description de l'interface dans l'étape de conception.

Les propriétés vérifiées sur le modèle Lustre sont des propriétés génériques ou spécifiques. Les propriétés génériques sont automatiquement extraites comme *l'observabilité* et *l'honnêteté*, tandis qu'un éditeur graphique est utilisé pour exprimer des propriétés spécifiques.

2.2.6 Utilisation de modèles de Markov

Des modèles de Markov ont été utilisés pour tester l'utilisabilité d'Interfaces Homme Machine à la phase de conception [53]. L'approche est fondée sur l'utilisation de l'outil *Mathematica* pour définir des modèles de Markov pour des interfaces modélisables avec des machines d'états finis, comme l'interface de dispositifs physiques (téléphones portables, distributeurs de tickets) et les interfaces basées sur les événements de souris (sites internet). Pour ces interfaces, la création et l'analyse de chaînes de Markov sont automatisées, ce qui permet la comparaison quantitative entre plusieurs conceptions de l'interface.

L'interface à tester est modélisée avec une machine d'états finis à laquelle est associé un modèle de Markov spécifiant des probabilités pour les transitions entre les états. L'analyse du modèle de Markov fournit des aperçus de la conception de l'interface. Le nombre des étapes (transitions) nécessaires à l'utilisateur pour réaliser des tâches est une mesure simple et évidente de l'utilisabilité.

Le modèle de Markov est représenté par une matrice P, appelée matrice de probabilité de transitions, représentant la probabilité de transitions entre les états. Un élément $P_{i,j}$ de cette matrice est la probabilité d'effectuer une transition de l'état i à l'état j. Pour l'utilisateur qui utilise le dispositif aléatoirement (sans

aucune connaissance préalable), le modèle de Markov est utilisé pour calculer l'utilisabilité.

Les auteurs de [53] ont introduit le concept de "connaissance de l'utilisateur sur la manière d'utiliser le dispositif". Ils définissent "le facteur de la connaissance" de l'utilisateur k comme un nombre de l'intervalle [0..1]. Si k=*1*, la connaissance de l'utilisateur est égale à celle de concepteurs. Si k=*0*, l'utilisateur n'a pas de connaissance et se comporte aléatoirement sans préférence. L'approche montre l'impact de la connaissance de l'utilisateur sur l'utilisabilité.

Pour un but donné (transition d'un état à un autre), la matrice de probabilité de transition pour un usager avec une connaissance k est $kD + (1 - k)P$, où :

- D est la matrice de la connaissance parfaite pour réaliser ce but ($D_{i,j} = 1$ si l'action qui change l'état i à l'état j est optimale pour réaliser ce but, sinon $D_{i,j} = 0$). La modélisation de la connaissance générale d'un dispositif requiert une matrice pour chaque but.
- P est la matrice de probabilité de transitions. Dans cette matrice, pour chaque état, toutes les actions disponibles ont la même probabilité.

Pour un but donné, à partir de la matrice $kD+(1-k)P$, l'utilisabilité (le nombre des étapes pour réaliser ce but) est calculée automatiquement en fonction de la connaissance de l'utilisateur et le graphe correspondant est tracé.

En comparant les différents graphes d'utilisabilité pour différentes conceptions de l'interface, le concepteur peut choisir la conception qui rend le dispositif plus facile à utiliser.

2.3 Méthodes de test

2.3.1 Validation de dialogue avec analyse de tâche

Une méthode pour la validation de dialogue de systèmes interactifs est proposée dans [19]. Le principe de cette méthode est de générer l'ensemble complet de séquences possibles d'interaction à partir de l'analyse du modèle de tâches. Cet ensemble est inséré dans le composant Contrôleur de Dialogue de l'application. Les appels à partir du composant Dialogue vers le Noyau Fonctionnel sont interceptés et comparés avec la tâche de l'utilisateur.

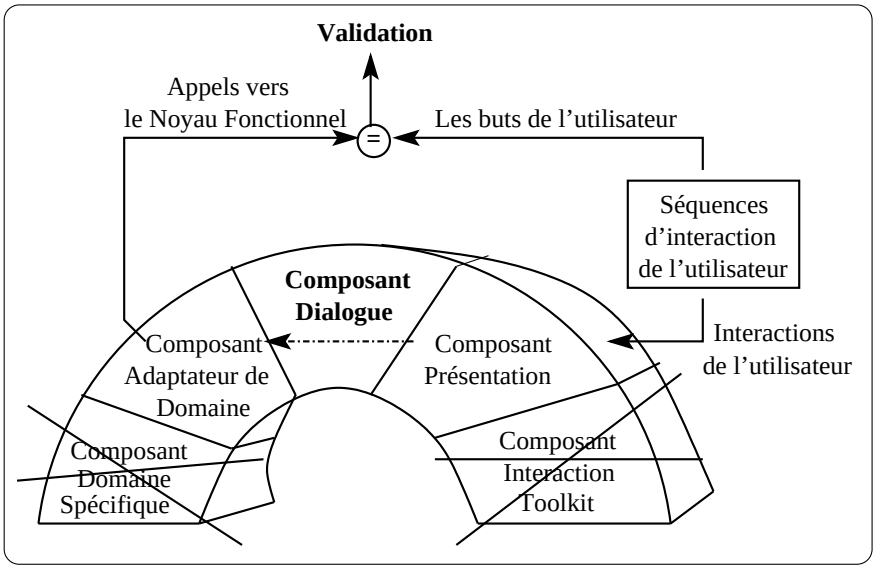

FIG. 2.7 – Principe de validation dans le modèle Arch

Génération de séquences de test : Le modèle de tâches considéré est une version simplifiée de HTA [36] et MAD [37]. Il ne permet ni la récursivité ni les boucles : l'analyse de ce modèle spécifie un nombre fini de séquences d'interaction. Le modèle est utilisé pour générer automatiquement toutes les séquences possibles d'interaction afin d'assurer la couverture complète de la validation de l'application.

Validation de propriétés : L'objectif est de vérifier que toutes les fonctions du système sont atteignables par l'utilisateur (*atteignabilité*) et sont cohérentes avec le domaine d'objets (*complétude de tâche*). Pour le premier objectif le modèle de tâches est utilisé. L'évaluation de la complétude nécessite de vérifier que les méthodes du Noyau Fonctionnel appelées par le Contrôleur de Dialogue pendant l'interaction sont cohérentes avec le but et les sous buts définis dans l'analyse des tâches.

Principe de validation de dialogue dans le modèle Arch : En supposant que le système est implémenté avec le modèle d'architecture Arch, comme le montre la figure 2.7, le principe de validation est de générer l'ensemble complet des séquences possibles d'interaction à partir de l'analyse des tâches. Cet

ensemble est ensuite injecté dans le composant Dialogue de l'application par le composant Présentation. L'application est exécutée et le composant Dialogue appelle le Noyau Fonctionnel par l'intermédiaire du composant Adaptateur de Domaine. Ces appels sont interceptés et comparés avec la tâche de l'utilisateur. S'ils concordent, la séquence d'interaction est supposée valide. Le composant Dialogue est considéré comme une boîte noire (il faut seulement respecter les interfaces des composants Présentation et Adaptateur de Domaine qui sont modifiés et réutilisés).

Cette méthode est proposée pour la validation des systèmes de CAO. Elle demande la construction manuelle du lien entre les tâches élémentaires et les fonctions du Noyau Fonctionnel. Les séquences d'interaction sont générées au niveau Composant Présentation.

2.3.2 Utilisation de machines d'états finis à variables

Avec cette méthode [20], l'interface de l'utilisateur est spécifiée comme une machine d'états finis à variables *VFSM* (v*ariable finite state machine*).

Une VFSM à n-variables $M_v = (S, I, O, T, \phi, V, \zeta)$ est définie formellement comme suit :

- S, I, O sont respectivement les ensembles d'états, d'entrées et de sorties ;
- V est un ensemble de variables. $V = \{V_1, V_2, ..., V_n\}$, où n est le nombre de variables, et V_i est l'ensemble de valeurs de i-*ème* variable ;
- T est la fonction de transition : $T : D_T \longrightarrow S$, où $D_T \subseteq S \times I \times V_1 \times V_2 \times ... \times V_n$;
- ϕ est la fonction de sortie : $\phi : D_T \longrightarrow O$;
- ζ est la fonction de transition de variables : $\zeta : D_T \longrightarrow V_1 \times V_2 \times ... \times V_n$.

Cette machine est transformée automatiquement en une machine d'états finis MEF, utilisée pour la génération selon la méthode W_p [54]. Cette méthode permet de détecter des différences entre le modèle MEF et l'implémentation. Les séquences de tests générées par cette méthode sont appliquées sur le modèle pour déterminer les séquences de sorties attendues.

L'exécution automatique de séquences de test requiert l'instrumentation de l'interface afin qu'elle puisse lire les séquences d'entrées et enregistrer les sorties pour une analyse ultérieure.

2.3.3 Utilisation d'une structure hiérarchique

L'approche proposée dans [21] utilise un modèle hiérarchique pour guider la génération de cas de test. Elle définit un ensemble d'opérateurs qui sont organisés de façon hiérarchique. Les opérateurs du niveau supérieur sont construits à partir d'autres plus simples. Les opérateurs simples correspondent aux actions d'utilisateur. Chaque opérateur possède une précondition qui doit être vraie avant l'exécution de l'opérateur, et une postcondition (conditions devant être vraie après la réalisation de l'action).

Le testeur spécifie un ensemble d'opérateurs, un état initial et un état final pour le planificateur (fondé sur les principes de l'intelligence artificielle). Ce dernier produit une séquence d'opérateurs qui change l'état initial en état final. Il génère les cas de test de niveaux supérieur d'abstraction. Le processus de composition conduit à des cas de test de bas niveau qui sont des événements utilisateur.

2.4 Conclusion

Dans ce chapitre, nous avons présenté des méthodes formelles pour la modélisation et la vérification de systèmes interactifs. Dans la plupart de ces méthodes, comme LIM (Lotos Interactor Model), ICO (Interactive Cooperative Object) ou Lustre, l'application interactive est spécifiée formellement comme un modèle abstrait. Les propriétés sont vérifiées sur ce modèle par des techniques traditionnelles de vérification formelle comme le model-checking. L'utilisation de la méthode B a également été suggérée pour assurer que les propriétés de l'application interactive sont préservées pendant le processus du raffinement. Dans tous les cas, la vérification requiert une démarche lourde de spécification que la plupart de concepteurs des applications interactives ne peuvent pas effectuer.

L'utilisation de chaînes de Markov a été suggérée initialement pour tester l'utilisabilité d'interfaces de dispositifs qui sont modélisés avec des machines d'états finis. Le modèle de Markov est ajouté à cette spécification et utilisé pour le test d'utilisabilité.

Ces méthodes formelles sont utilisées dans la phase de conception. Elles modélisent le système et prouvent les propriétés sur le modèle. Cependant, il n'y a

pas une garantie de l'équivalence entre ce modèle et le système final.

Pour ces raisons, il est important de bien tester les systèmes interactifs.

Il y a peu de méthodes pour tester automatiquement ces systèmes, nous en avons présenté certaines dans ce chapitre. La méthode de [19] permet de valider le composant Contrôleur de Dialogue pour les systèmes de CAO. Cette méthode utilise un arbre de tâches simple dans lequel il n'y a pas de boucles afin de générer des séquences finies de tests. Le testeur doit bien connaître l'application parce que il doit établir une correspondance entre chaque tâche élémentaire et la fonction associée du Noyau Fonctionnel. La méthode de [20] exige une spécification de l'interface avec une machine d'états finis à variables, ce qui n'est pas toujours facile. La méthode [21] est dédiée à la génération de cas de test. Le testeur doit spécifier l'interface avec un ensemble d'opérateurs hiérarchiques et définir pour chaque opérateur la précondition et la postcondition.

Chapitre 3

Lutess : Un outil de test de systèmes synchrones

Le travail de cette thèse s'inscrit dans le cadre du test automatique de systèmes interactifs. Plus précisément, nous souhaitons examiner l'adéquation de l'approche synchrone au test de programmes interactifs. Dans cette perspective, nous utilisons Lutess, un outil de test de systèmes synchrones.

Dans ce chapitre, nous présentons l'approche synchrone (paragraphe 3.1), le langage synchrone Lustre (paragraphe 3.2) et les principes de vérification de programmes Lustre (paragraphe 3.4). L'outil de test Lutess est enfin présenté dans le paragraphe 3.5.

3.1 L'approche synchrone

Un programme est dit *synchrone*, s'il vérifie 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 et l'émission de o_t est fait instantanément, au même instant t (figure 3.1).

En pratique, on considère qu'un logiciel a un comportement synchrone s'il réagit à son environnement avant toute évolution de ce dernier. Ainsi, si à l'instant t le logiciel reçoit les entrées i_t depuis son environnement externe, il émet les sor-

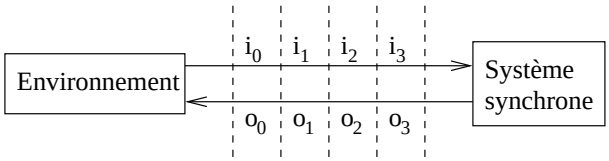

FIG. 3.1 – Un programme synchrone

ties o_t avant qu'une nouvelle entrée i_{t+1} soit disponible. Cette propriété permet de s'abstraire des problèmes de temporalité, très importants quand il s'agit de réaliser des *systèmes réactifs*.

Des langages de spécification et de programmation propres à cette approche ont été proposés, dont un représentant très connu est le langage Lustre, présenté dans le paragraphe suivant.

3.2 Le langage Lustre

Lustre [50, 55] est un langage synchrone, destiné à la spécification et la programmation de systèmes synchrones. Il peut être considéré à la fois comme une logique temporelle du passé [56] et comme un langage de programmation. C'est un langage flot de données : un flot est une séquence de valeurs couplée à une horloge qui indique à quel moment certaines valeurs apparaissent.

Un programme Lustre est structuré en nœuds. Un nœud Lustre est constitué d'un ensemble d'équations qui définissent ses variables de sortie comme des fonctions des variables d'entrée et des variables locales (figure 3.2). Il n'y a pas d'ordre entre les équations et chaque variable définit un flot.

```
node Programme(<entrées>) returns (<sorties>)
var
    <variables locales>
let
    <sorties>=f(<entrées>,<variables locales>);
tel
```

FIG. 3.2 – Structure syntaxique d'un programme Lustre

Une expression Lustre contient des constantes, des variables, des opérateurs logiques, arithmétiques ainsi que deux opérateurs spécifiques : "pre" et "->".

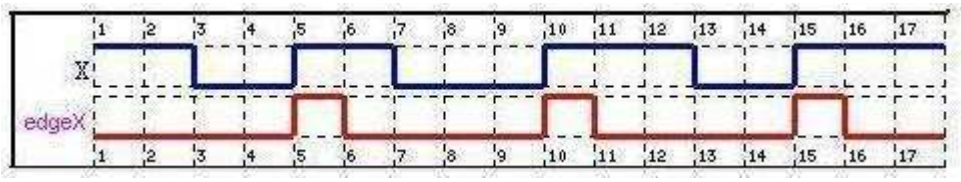

FIG. 3.3 – Un chronogramme représentant une trace de edge (X)

L'opérateur *précédent* (noté *pre*) donne accès à la dernière valeur qu'une expression vient de prendre (au top d'horloge précédent). L'opérateur *suivi par* (noté ->), est utilisé pour désigner la valeur initiale (à $t = 0$) des expressions :

- Si E est une expression définissant le flot $(e_0, e_1, ..., e_n, ...)$, $preE$ va dénoter le flot $(nil, e_0, e_1, ..., e_{n-1}, ...)$ où nil est une valeur indéfinie. En d'autres termes, $preE$ retourne, à un instant t, la valeur de l'expression E au moment $t - 1$.
- Si E et F sont des expressions dénotant, respectivement, les séquences $(e_0, e_1, e_2, ..., e_n, ...)$ et $(f_0, f_1, f_2, ..., f_n, ...)$, l'expression E->F définit le flot $(e_0, f_1, f_2, ..., f_n, ...)$.

Les nœuds Lustre sont réutilisables : une fois qu'un nœud est défini, il peut être ensuite utilisé dans d'autres nœuds comme tout autre opérateur.

Exemple d'un nœud Lustre : L'opérateur *edge* décrit le front montant d'un flot booléen. Cet opérateur retourne la valeur *varie* lorsque le flot booléen donné en paramètre passe de *faux* à *vrai*. Cet opérateur est implémenté avec le nœud Lustre suivant :

```
node edge (X : bool) returns (edgeX : bool) ;
let
   edgeX = false -> X and not pre (X) ;
tel
```

À l'instant initial, la sortie de cet opérateur prend la valeur *false* indépendemment de la valeur de X. À tous les instants suivants, edge (X) est *vrai* chaque fois X a été évalué à *false* à l'instant précédent et à *vrai* à l'instant courant (voir la figure 3.3).

Exemple de réutilisation d'un nœud : L'opérateur *countEdge* compte le nombre de fronts montants d'un flot X. Il est implémenté avec le nœud Lustre *countEdge* qui utilise le nœud *edge* :

```
node countEdge (X : bool) returns (countEdgeX : int) ;
let
   countEdgeX = (0->pre countEdgeX)+
                (if edge(X) then 1 else 0) ;
tel
```

Opérateurs temporels :

Les nœuds peuvent également servir à définir des opérateurs exprimant des invariants ou propriétés termporelles ce qui fait de Luste une logique temporelle du passé. Considérons par exemple, 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. À chaque instant, la sortie est vraie si seulement si l'entrée n'a jamais été vraie depuis le début de l'exécution du 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*).

Nous présentons ici la définition intuitive, puis en Lustre, des opérateurs qui sont souvent utilisés.

A, B et C étant des expressions booléennes Lustre quelconques :

- *always_from_to (A, B, C)* est *vrai* uniquement si A a toujours été *vrai* entre les deux derniers instants où B et C ont pris la valeur *vrai*. Cet opérateur s'assure que l'événement A s'est toujours produit entre les instants où l'événement B et l'événement C se sont produits.
- *once_from_to (A, B, C)* est *vrai* uniquement si A a été *vrai* au moins une fois entre les deux derniers instants où B et C ont pris la valeur *vrai*. Cet opérateur s'assure que l'événement A s'est produit au moins une fois entre les instants où l'événement B et l'événement C se sont produits.

- *once_since (A, B)* est *vrai* si la valeur de A a été *vrai* au moins une fois depuis le dernier instant où B a été *vrai*. Cet opérateur s'assure que l'événement A s'est produit au moins une fois depuis la dernière occurrence de l'événement B.
- *always_since (A, B)* est *vrai* si la valeur de A a toujours été *vrai* depuis le dernier instant où B a été *vrai*. Cet opérateur s'assure que l'événement A s'est toujours produit depuis la dernière occurrence de l'événement B.
- *after (A)* est *vrai* si la valeur de A a été *vrai* avant au moins une fois.
- *implies (A, B)* représente la valeur logique de l'implication : $A \Rightarrow B$.

L'expression en Lustre de ces opérateurs est réalisée à l'aide de nœuds dont la description complète [52, 57] est donnée dans la figure 3.4.

3.3 Exemple d'un programme Lustre

Nous reprenons ici un exemple connu [52] de système réactif contrôlant un aiguillage de tramways.

Dans chaque terminus de ligne de tramway, il y a une section spéciale ("U-turn") permettant aux trams de commuter d'une voie à une autre, afin de répartir dans le sens inverse (voir fig 3.5). Cette section est composée de trois voies A, B, C et un aiguillage S. En supposant que la voie entrante est A et la voie sortante est C, les trams passant de A à C doivent d'abord attendre que l'aiguillage connecte A avec B avant de transiter sur B et attendre également que S connecte B avec C avant de répartir vers C.

Le système qui contrôle cette section est appelé *UMS (U-turn section Management System)* (voir fig 3.6). Ce système reçoit en entrée les signaux suivants :

- **ack_AB** et **ack_BC** qui indique si l'aiguillage connecte actuellement A avec B ou B avec C.
- **on_A**, **on_B**, **on_C** qui sont trois capteurs, un pour chaque voie de la section. Ils sont actifs s'il y a un tram sur la voie correspondante.

Les signaux de sortie pour ce système sont :

- **do_AB** et **do_BC** qui sont des requêtes pour l'aiguillage lui demandant de connecter A avec B ou B avec C.
- **grant_access** et **grant_exit** qui sont des des autorisations de circulation

```
node always_from_to(A, B, C : bool)
          returns (alwaysAfromBtoC : bool);
let
    alwaysAfromBtoC = once_since(C,B)
                      or always_since(A,B);
tel

node once_from_to(A, B, C : bool)
         returns (onceAfromBtoC : bool);
let
   onceAfromBtoC = implies(C, once_since(A,B));
tel

node once_since(A, B : bool)
       returns (onceAsinceB : bool);
   let
      onceAsinceB = if B
                    then A
                    else (true -> (A or pre (onceAsinceB)));
tel

node always_since(A, B : bool)
         returns (alwaysAsinceB : bool);
let
   alwaysAsinceB = if never(B)
                   then true else if B then A
                   else (true -> A and pre (alwaysAsinceB));
tel

node after ( A : bool)
   returns ( afterA : bool)
let
    afterA = false -> pre (A or afterA);
tel

node implies(A, B : bool)
    returns (AimpliesB : bool);
let
   AimpliesB = not A or B;
tel
```

FIG. 3.4 – Opérateurs temporels définis en Lustre

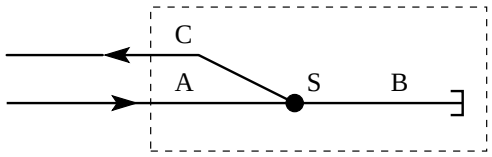

FIG. 3.5 – Une section U-tour de tramways

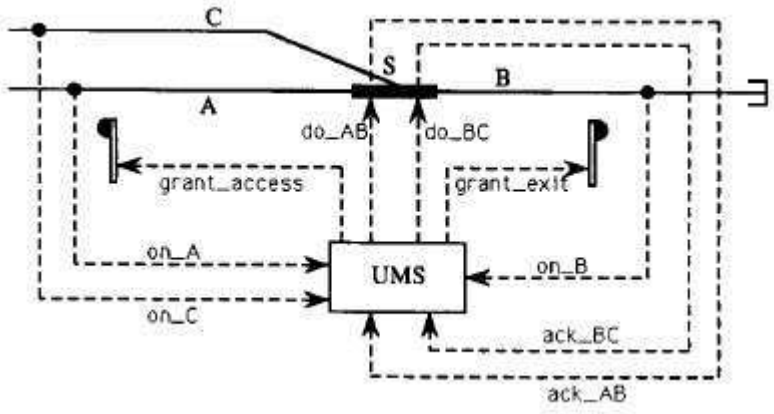

FIG. 3.6 – Le système UMS et son environnement

pour les trams. Elles sont matérialisées par des feux de circulation. L'implémentation de ce système en langage Lustre est présentée dans la figure 3.7, et le chronogramme de la trace d'une exécution de ce système est illustré dans la figure 3.8.

3.4 Vérification de programmes Lustre

Le langage Lustre pouvant être considéré comme une logique temporelle du passé, il peut servir pour exprimer des propriétés invariantes décrivant des comportements attendus du système à vérifier ou des assertions sur son environnement d'exécution.

3.4.1 Propriétés de sûreté

Les propriétés invariantes qu'on peut exprimer en Lustre sont les propriétés de sûreté. Dans le cas d'UMS (3.3), on a besoin de vérifier les propriétés [52] ci-dessous :

```
node UMS (on_A, on_B, on_C, ack_AB, ack_BC : bool)
     returns (grant_access, grant_exit, do_AB, do_BC : bool);
var empty_section, only_on_B : bool;
let
   grant_access = empty_section and ack_AB;
   grant_exit = only_on_B and ack_BC;
   do_AB = not ack_AB and empty_section;
   do_BC = not ack_BC and only_on_B;
   empty_section = not (on_A or on_B or on_C);
   only_on_B = on_B and not (on_A or on_C);
tel
```

FIG. 3.7 – Le programme Lustre pour le système UMS

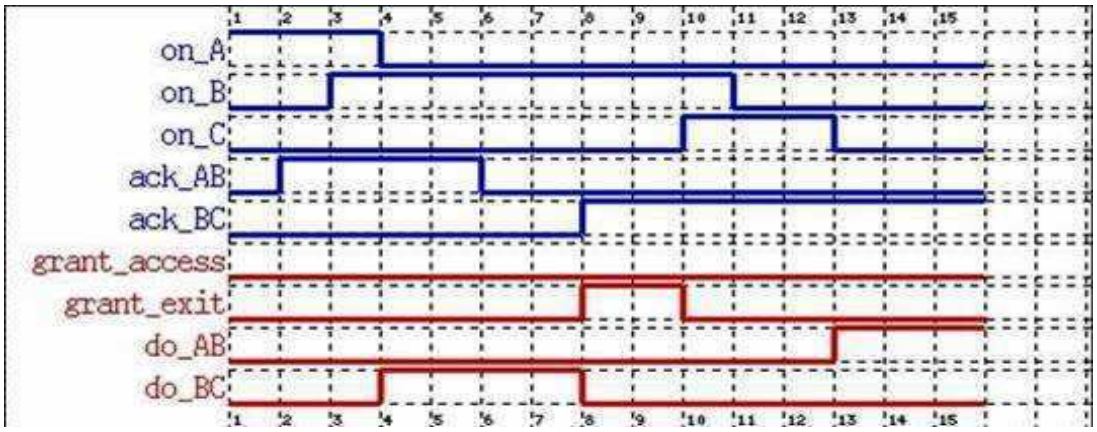

FIG. 3.8 – Un chronogramme de trace du système UMS

– Le tram peut accéder à la section seulement si cette dernière est vide :

```
no_collision = implies(grant_access, empty_section)
```

– Il ne faut pas demander à l'aiguillage deux connexions à la fois :

```
exclusive_req = not(do_AB and do_BC)
```

– L'aiguillage doit toujours connecter A et B entre l'instant où le tram est autorisé d'accéder à la section et l'instant où il arrive sur la voie B :

```
no_derail_AB = always_from_to (ack_AB,
                               grant_access, only_on_B)
```

– L'aiguillage doit toujours connecter B avec C entre l'instant où le tram est autorisé de quitter la section et l'instant où il la quitte :

```
no_derail_BC = always_from_to (ack_BC,
                                grant_exit, empty_section)
```

3.4.2 Spécification de l'environnement

Dans le processus de vérification de programmes Lustre, il est important de fournir une description du comportement de l'environnement du système à vérifier. Cette description est écrite à l'aide du mécanisme d'*assertions* en Lustre. Les assertions Lustre sont des propriétés invariantes supposées d'être toujours vraies.

Dans l'exemple UMS (3.3), les propriétés de l'environnement [52] sont les suivantes :

– L'aiguillage ne peut pas connecter A avec B et B avec C en même temps :

```
not (ack_AB and ack_BC)
```

– L'aiguillage reste stable dans sa position sauf si un signal lui demandant de changer d'état lui parvient :

```
always_from_to (ack_AB, ack_AB, do_BC)
always_from_to (ack_BC, ack_BC, do_AB)
```

– Initialement, il n'y a pas de trams dans la section :

```
empty_section->true
```

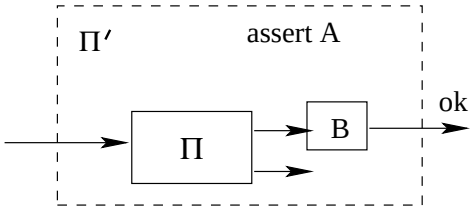

FIG. 3.9 – construire un programme de vérification

– Les trams obéissent aux feux de circulation :

```
true->implies(edge(not empty_section), pre grant_access)
true->implies(edge(on_C), pre grant_exit)
```

– Lorsque le tram quitte A, il est en B. Lorsque le tram quitte B, il est soit en A ou en C

```
implies(edge(not on_A), on_B)
implies(edge(not on_B), on_A or on_C)
```

3.4.3 Vérification

Étant donné un programme Π, des propriétés de sûreté exprimées par une expression booléenne B et des suppositions sur le comportement de l'environnement données par l'assertion A, un nouveau programme Π' peut être construit en mettant ensemble Π, B et l'assertion A comme dans la figure 3.9. Ainsi, pour vérifier les propriétés du programme Π, il faut vérifier que la seule sortie du programme Π' est toujours vraie pendant toute exécution du programme qui satisfait continûment l'assertion A.

La vérification formelle est réalisée en utilisant la technique de *model checking* sur une machine d'état finis correspondant à une abstraction booléenne du programme [52]. Les états atteignables doivent être examinés sur cette machine pour vérifier les propriétés du programme.

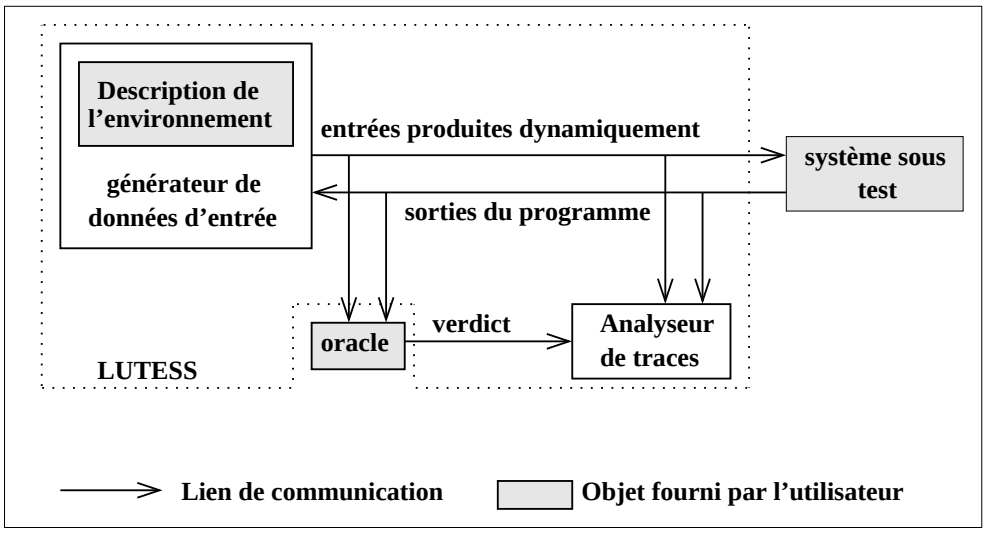

FIG. 3.10 – Architecture de Lutess

3.5 L'outil de test Lutess

Lutess est un outil de test de programmes synchrones [58].

Lutess permet de réaliser un test fonctionnel en boîte noire. La génération des entrées est effectuée « à la volée » : les entrées du programme pendant le test sont calculées en fonction des entrées et des sorties précédentes.

Les travaux sur le test des logiciels synchrones ayant abouti au développement de l'environnement Lutess [58, 28] avaient pour principale motivation la volonté d'apporter un moyen de vérification complémentaire à la preuve formelle par « model-checking » [52]. Les principes de spécification mis en œuvre sont donc semblables à cette dernière. Ainsi, Lutess requiert trois composants pour son fonctionnement (figure 3.10) :

1. un système sous test, qui est un programme exécutable synchrone,
2. un oracle, qui contient les propriétés de sûreté que le système sous test doit satisfaire,
3. une description de l'environnement, sous forme de contraintes que le générateur de données de test doit respecter.

Le système sous test doit être un programmes exécutable à entrées-sorties booléennes qui doit avoir un comportment synchrone. Ainsi, le programme exé-

```
node Oracle_UMS(on_A, on_B, on_C, ack_AB, ack_BC,
   grant_access, grant_exit, do_AB, do_BC : bool) returns (ok : bool)
var empty_section, only_on_B,
    no_collision, exclusive_req, no_derail_AB, no_derail_BC : bool ;
let
   ok = no_collision and exclusive_req
         and no_derail_AB and no_derail_BC ;
   no_collision = implies(grant_access, empty_section) ;
   exclusive_req = not(do_AB and do_BC) ;
   no_derail_AB = always_from_to_(ack_AB, grant_access, only_on_B) ;
   no_derail_BC = always_from_to_(ack_BC, grant_exit, empty_section) ;
   empty_section = not (on_A or on_B or on_C) ;
   only_on_B = on_B and not (on_A or on_C) ;
tel
```

FIG. 3.11 – L'oracle de UMS

cutable peut être écrit dans tout langage de programmation.

L'oracle est un programme exécutable et synchrone qui observe l'échange entre le générateur de données et le système sous test et fournit pour chaque pas de test un verdict. Pour cela, les variables d'entrée de l'oracle sont les variables d'entrée et de sortie du système sous test. L'oracle a ainsi une unique sortie booléenne qui correspond au verdict. À chaque pas de test, le verdict est calculé en fonction des échanges présents et passés entre le générateur et le système sous test.

La figure 3.11 montre l'oracle du système UMS (c.f. paragraphe 3.4.1).

La description de l'environnement se fait sous la forme d'un nœud Lustre spécial (*nœud de test*), excluant au moyen de propriétés invariantes les comportements irréalistes de l'environnement (opérateur *environment*) qui ne doivent pas être pris en compte pendant la génération de données de test. Le nœud de test est repéré par le mot clé *testnode* et correspond à une extension de la grammaire de Lustre. Ses entrées sont les sorties du programme sous test et ses sorties sont les entrées du programme sous test (fig 3.12).

```
testnode Env_UMS(grant_access, grant_exit, do_AB, do_BC : bool)
   returns (on_A, on_B, on_C, ack_AB, ack_BC : bool)
var empty_section : bool ;
let
   environment(not (ack_AB and ack_BC),
               always_from_to_(ack_AB, ack_AB, do_BC),
               always_from_to_(ack_BC, ack_BC, do_AB),
               empty_section->true,
               true->implies(edge(not empty_section),
                             pre grant_access),
               true->implies(edge(on_C), pre grant_exit),
               implies(edge(not on_A), on_B),
               implies(edge(not on_B), on_A or on_C)
             ) ;
   empty_section = not (on_A or on_B or on_C) ;
tel
```

FIG. 3.13 – L'environnement de UMS pour Lutess

```
testnode Environnement(<sorties du logiciel>)
   returns (<entrées du logiciel>)
var
   <variables locales>
let
   environment(C1,...,Cn) ;
   <définition des variables locales>
tel
```

FIG. 3.12 – Structure syntaxique d'un nœud de test

Les propriétés invariantes de l'environnement sont exprimées au moyen de formules Lustre. Chaque invariant est exprimé en fonction des variables d'entrée et en tenant compte des échanges passés entre le système sous test et son environnement. À chaque pas de test, ces invariants définissent un ensemble de vecteurs valides d'entrée pour le système sous test.

La figure 3.13 montre la spécification de l'environnement du système UMS (c.f. paragraphe 3.4.2) pour Lutess.

Mise en œuvre du test

À partir de la description de l'environnement, Lutess construit un générateur de données d'entrée 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 de l'oracle de test pour une analyse éventuelle ultérieure.

Le test est effectué par des cycles successifs action-réaction. À chaque cycle (pas de test), le générateur produit un vecteur valide d'entrée qui respecte les contraintes de l'environnement. Ce vecteur est envoyé au logiciel sous test qui réagit en émettant un vecteur de sortie vers le générateur. Ce dernier produit un nouveau vecteur d'entrée et le cycle se répète. En parallèle, l'oracle observe les entrées et les sorties du programme sous test, détermine si les propriétés du logiciel sont violées et émet un verdict à chaque cycle de test. L'ensemble des données échangées et des verdicts sont récupérés et enregistrés sous forme de traces.

3.6 Les techniques de génération de tests en Lutess

3.6.1 Simulation de l'environnement en Lutess

Formellement, le nœud de test est transformé en un simulateur d'environnement qui est une une machine d'états finis

$$M_{env} = (S_{env}, s_{init_{env}}, V_i, V_o, env, trans_{env})$$

où :

- S_{env} est l'ensemble des états de l'environnement (ensemble de valeurs des variables d'état sve) ;
- $s_{init_{env}}$ est l'état initial de l'environnement ;
- i et o étant respectivement les ensembles de variables d'entrée et de sortie du logiciel sous test, V_i et V_o sont leurs ensembles de valeurs associés ;
- la fonction $env : S_{env} \times V_i \rightarrow \{false, true\}$ définit à tout moment t l'ensemble des valeurs des variables d'entrée i conformes à la spécification de l'environnement (i.e. ensemble de valeurs rendant *vraie* la fonction env dans l'état où se trouve l'environnement à l'instant t) ;
- la fonction de transition $trans_{env} : S_{env} \times V_i \times V_o \rightarrow S_{env}$ calcule l'état de l'environnement après chaque échange d'entrées-sorties avec le logiciel

synchrone.

L'algorithme de génération d'entrées choisit un vecteur valide d'entrée conforme à la spécification de l'environnement (une valeur des variables d'entrée rendant vraie la fonction de l'environnement *env*). Pour chaque état, tous les vecteurs d'entrée valides possèdent la même probabilité d'être sélectionnés. Ce mode de génération est appelé *simulation aléatoire de l'environnement* (c.f. paragraphe 3.6.2). Lutess fournit également d'autres stratégies de génération : les *profils opérationnels* (c.f. paragraphe 3.6.3) où un vecteur valide d'entrée est sélectionné en fonction d'une distribution spécifique des probabilités, les *schémas comportementaux* (c.f. paragraphe 3.6.4) où la génération d'entrée prend en compte ces schémas spécifiés par le testeur et le test guidé par des *propriétés de sûreté* (c.f. paragraphe 3.6.5) où les entrées sont potentiellement plus aptes à détecter des violations des propriétés.

Définition : La fonction d'environnement *env* définit pour chaque état $s \in S_{env}$ un ensemble de vecteurs valides d'entrée. On va noter cet ensemble $V_{i_{env}}$. Il est défini ci-dessous.

$$V_{i_{env}} = \{i \in V_i : env(s, i) = true\}$$

3.6.2 Test aléatoire

La génération aléatoire est utilisée par défaut dans Lutess. Elle consiste à choisir un vecteur valide d'entrée de l'ensemble de vecteurs valides d'entrée $V_{i_{env}}$ de manière équiprobable.

3.6.3 Test guidé par des profils opérationnels

Les profils opérationnels sont définis au moyen de probabilités d'occurrence d'événements d'entrée. Ces probabilités peuvent être conditionnelles ou inconditionnelles. Chaque probabilité conditionnelle est prise en compte pendant la génération si et seulement si sa condition associée est satisfaite.

Pendant la génération guidée par un profil opérationnel [59, 60, 61], comme la génération aléatoire, un vecteur valide d'entrée de l'ensemble $V_{i_{env}}$ est également

```
pas :          événements d'entrée et de sortie de UMS            :oracle
1   :  -    -    -    ack_AB -      grant_access -          -    - :true
2   :  on_A -    -    ack_AB -      -            -          -    - :true
3   :  on_A on_B -    ack_AB -      -            -          -    - :true
         ........
16  :  on_A -    -    -      ack_BC -            -          -    - :true
17  :  -    on_B -    -      ack_BC -            grant_exit -    - :true
18  :  -    on_B on_C -      ack_BC -            -          -    - :true
```

FIG. 3.14 – Un extrait d'une trace de test UMS avec un profil opérationel

choisi. Mais, au lieu d'une sélection équiprobale des vecteurs d'entrée, la sélection est effectuée en respectant une loi (élémentaire) de probabilité.

Le guidage par des profils opérationnels permet de rendre l'opération de test proche d'un profil d'usage. À titre d'illustration, dans l'exemple d'UMS, si on veut tester le système dans le cas où les trams traversent rapidement l'aiguillage, on peut spécifier une forte probabilité pour qu'un tram passe sur B (resp. sur C), s'il vient d'arriver sur A (resp. sur B)

```
proba (
    (on_B, 0.8, pre edge (on_A)),    ---(1)
    (on_C, 0.8, pre edge (on_B) )    ---(2)
    ) ;
```

La génération est effectuée en respectant ces probabilités conditionnelles et les contraintes de l'environnement. La figure 3.14 fournit un extrait d'une trace avec ce mode de la génération. On peut observer que dans le pas 1 de la trace, la section est vide ; ensuite un tram arrive sur A (événement on_A au pas 2) avant de traverser l'aiguillage pour aller sur B (événement on_B). De manière similaire pour les pas 16, 17, 18, on peut observer que le tram passe rapidement de B à C.

3.6.4 Test guidé par schémas comportementaux

En Lutess, la génération de données d'entrées peut également être guidée par des schémas comportementaux [22, 28]. Un schéma est une description d'une classe de comportements vérifiant certaines conditions. Plus précisément, les schémas comportementaux sont décrits à l'aide de deux listes de conditions :

– la liste des *"conditions d'instant"* (*cond*).

– la liste des *"conditions d'intervalle"* (*intercond*).

Pour chaque état de l'environnement, le générateur de test guidé par les schémas comportementaux partitionne l'ensemble de vecteurs valides d'entrée $V_{i_{env}}$ en trois classes :

– la classe C_P des vecteurs de *Progression* qui regroupe les vecteurs d'entrée satisfaisant la condition d'instant, ces vecteurs permettent de passer d'une étape du schéma comportemental à une autre ;
– la classe C_R des vecteurs de *Régression* qui est composée des vecteurs d'entrée mettant en défaut la condition d'intervalle correspondante : ces vecteurs bloquent la progression et nécessitent de recommencer le schéma depuis son début ;
– la classe C_N des vecteurs *Neutres* qui inclut les vecteurs d'entrée n'appartenant à aucune des deux autres classes : ces vecteurs ne font ni régresser, ni progresser le schéma comportemental.

Ensuite, une classe parmi ces trois classes est sélectionnée en accord avec une probabilité définie par le poids de chaque classe :

$\wp(C_P) = \frac{P_{C_P}}{P_{C_P}+P_{C_R}+P_{C_N}}$, $\wp(C_R) = \frac{P_{C_R}}{P_{C_P}+P_{C_R}+P_{C_N}}$, $\wp(C_N) = \frac{P_{C_N}}{P_{C_P}+P_{C_R}+P_{C_N}}$

où P_{C_P}, P_{C_R}, P_{C_N} représentent respectivement les poids des classes C_p, C_R, C_N définis par le testeur.

Lutess procède au choix aléatoire équiprobable d'un vecteur d'entrée dans la classe sélectionnée et l'envoie au système sous test.

Considérons à nouveau le système UMS. On peut spécifier le comportement suivant : les trams qui arrivent dans la section traversent toujours l'aiguillage pour aller en B (ils ne reculent pas). La formule correspondante pour spécifier ce schéma comportemental est la suivante :

```
cond (on_A and not on_B,on_B and on_A,on_B and not on_A) ;
intercond(true,true,true) ;
```

La première condition de ce schéma est l'arrivée d'un tram sur A, la deuxième condition est le passage de ce tram de l'aiguillage pour aller sur B et la dernière condition est l'arrivée de ce tram sur B. Les conditions de l'intervalle sont toujours vérifiées pour cet exemple.

3.6.5 Test guidé par les propriétés de sûreté

Le principe du test guidé par les propriétés de sûreté [62, 63] est de choisir un vecteur d'entrée tel que la valeur de vérité de la propriété de sûreté dépende des sorties calculées par le programme. Par exemple, si i est une entrée et o une sortie d'un programme devant satisfaire la propriété $i \Rightarrow o$, l'entrée $i = vrai$ doit être générée (sinon, la propriété est vraie quelle que soit la valeur de o). Le guidage par les propriétés de sûreté favorise la génération de l'entrée $i = vrai$, à condition que les contraintes d'environnement le permettent. De même, pour la propriété $true \rightarrow$ **pre** $i \Rightarrow o$ il faut engendrer l'entrée $i = vrai$ à l'instant courant pour que la propriété soit violable à l'instant suivant. D'une manière générale, ce type de guidage consiste à engendrer à un instant t des entrées qui peuvent mener le logiciel dans une situation où la propriété peut être violée à un instant $t + k$, k étant le nombre maximum d'instants à considérer.

Les propriétés de sûreté qui vont guider la sélection des vecteurs d'entrée sont décrites dans le nœud de test de manière similaire aux contraintes d'environnement et identifiées par l'opérateur *safety*. En termes de sémantique, il est nécessaire d'étendre le simulateur d'environnement défini plus haut de telle sorte que les propriétés de sûreté y soient intégrées. Nous parlerons alors de *simulateur guidé par les propriétés de sûreté*. Les propriétés de sûreté peuvent être assimilées à un programme synchrone et de ce fait être représentées par un automate d'états finis

$M_{prop} = (S_{prop}, s_{init_{prop}}, V_i, V_o, prop, trans_{prop})$

où la fonction $prop : S_{prop} \times V_i \times V_o \rightarrow \{faux, vrai\}$ définit à tout moment t la valeur de vérité de la propriété.

Étant donnée la machine associée à l'environnement

$M_{env} = (S_{env}, s_{init_{env}}, V_i, V_o, env, trans_{env})$,

le simulateur d'environnement guidé par les propriétés de sûreté peut être formellement défini par la machine

$M_{Ps} = (S_{Ps}, s_{init_{Ps}}, V_i, V_o, pert, trans_{Ps})$ où :

- $S_{Ps} = S_{env} \times S_{prop}$ est l'ensemble des états ;
- $s_{init_{Ps}} = (s_{init_{env}}, s_{init_{prop}})$ est l'état initial ;
- V_i et V_o sont les ensembles de valeurs de vecteurs d'entrées i et de sorties o ;

- $trans_{Ps} : S_{Ps} \times V_i \times V_o \rightarrow S_{Ps}$ est la nouvelle fonction de transition. Elle est définie par :
  $trans((s_{env}, s_{prop}), i, o) = (trans_{env}(s_{env}, i, o), trans_{prop}(s_{prop}, i, o))$.
- La fonction $pert : S_{Ps} \times V_i \rightarrow \{false, true\}$ définit les vecteurs d'entrées dits *pertinents* : $pert(s_{Ps}, i) = \exists o \in V_o \wedge (s_{Ps} = (s_{env}, s_{prop})) \wedge env(s_{env}, i) \wedge (\neg prop(s_{prop}, i, o) \vee \exists i' \in V_i \wedge pert(trans_{Ps}(s_{Ps}, i, o), i'))$.

La différence principale avec le simulateur d'environnement réside dans le calcul de l'ensemble des vecteurs *pertinents*. Ce sont les vecteurs qui respectent l'environnement et sont susceptibles de violer la propriété de sûreté sur un chemin de longueur comprise entre 1 et k. Dans [64], trois différentes stratégies de calcul sont définies. La *stratégie d'union* considère tous les chemins de longueur comprise entre 1 et k qui mènent le logiciel à une situation susceptible de violer la propriété de sûreté (autrement dit *état suspect*). La *stratégie d'intersection* ne considère que les vecteurs d'entrée qui font partie de tous les chemins de longueur comprise entre 1 et k qui mènent à un état suspect. La *stratégie paresseuse* choisit le vecteur qui mène le plus rapidement à un état suspect.

Dans l'exemple du système UMS, pour favoriser la génération de données testant la propriété *no_ collision = implies(grant_ access, empty_ section)*, on peut écrire :

```
safety (implies(grant_access, empty_section)) ;
```

Chapitre 4

Test synchrone de systèmes interactifs

4.1 Introduction

Dans ce chapitre nous étudions l'application des techniques de test de logiciels réactifs synchrones — présentées dans le chapitre précédent — à la validation de systèmes interactifs. Le langage synchrone Lustre a déjà été utilisé pour la spécification de systèmes interactifs [12, 13, 14] (cf. le paragraphe 2.2.5). En effet, il y a une similitude de comportement entre les systèmes interactifs et les systèmes qualifiés de synchrones, car, 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". En pratique, l'hypothèse de synchronisme est vérifiée si le logiciel est capable de prendre en compte toute évolution de son environnement externe. Ainsi, une application interactive peut être vue comme un programme synchrone si toutes les actions de l'utilisateur et autres stimuli externes sont pris en considération lors de son exécution.

L'approche synchrone permet une modélisation plus simple des comportements et offre par ailleurs la possibilité de procéder à des simulations et tests automatiques. L'outil Lutess qui a été présenté dans le chapitre précédent, a initialement été conçu pour tester les programmes synchrones écrits en Lustre. Cet

outil offre plusieurs techniques de test (test aléatoire, schémas comportementaux, profils opérationnels, ...). Ces techniques permettent de générer des séquences de tests qui favorisent l'apparition de certains comportements. Nous souhaitons ainsi étudier l'application de ces techniques sur les systèmes interactifs.

Les applications interactives s'orientent actuellement vers l'utilisation de multiples modalités : c'est-à-dire plusieurs techniques d'interaction comme les boutons de la souris et la voix. La validation de tels systèmes est également une question importante. Ces systèmes disposent de caractéristiques concernant la multimodalité, en plus de l'aspect interactif. Dans ce chapitre, nous étudions la validation de l'aspect interactif, tandis que la validation de l'aspect multimodal sera traitée plus tard (cf. chapitre 7).

Nous utilisons l'application interactive "Memo" dans l'ensemble du chapitre pour illustrer la problématique et les solutions proposées. Cette application est présentée dans le paragraphe 4.2.

Nous donnons ensuite une modélisation simple du comportement des applications interactives insistant sur l'interaction avec leur environnement (cf. paragraphe 4.3). Étant donné que ces systèmes ne sont pas synchrones, nous étudions ensuite l'adéquation de l'approche synchrone pour le test de ces systèmes. Pour cela, nous nous appuyons sur des travaux théoriques existants pour montrer que la synchronisation de leur trace est possible (cf. paragraphe 4.4). Enfin, nous détaillons l'utilisation de Lutess pour le test de systèmes interactifs dans le paragraphe 4.5.

4.2 L'application interactive Memo

Memo [65] est un système interactif multimodal. Dans ce chapitre nous ne tenons pas compte de la multimodalité (qui sera étudié dans la deuxième partie de cette thèse) mais nous nous concentrons sur les aspects fonctionnels.

Memo 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 4.1 (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 4.1 (à droite). De plus, un GPS

FIG. 4.1 – Gauche : un utilisateur de Memo, équipé d'un casque semi-transparent. 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.

et un magnétomètre sont portés par l'utilisateur et permettent au système de calculer la localisation et l'orientation de ce dernier. Trois tâches sont possibles :

- changer l'orientation et la 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 ;
- manipuler (récupérer, placer ou supprimer) une note ;
- quitter le système.

Considérons la tâche "manipulation d'une note" permettant à l'utilisateur mobile de 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 qu'il porte ou visible dans son environnement physique. Si l'utilisateur porte une note tout en voyant une autre note, la commande "remove" supprime la note dans le monde physique (la priorité est donnée à la manipulation de notes du 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.

4.3 Modélisation comportementale

En général, les systèmes interactifs attendent des entrées de l'utilisateur pour réagir. Autrement dit, ce sont des systèmes qui échangent des entrées et des sorties avec l'utilisateur.

En s'inspirant de [20] (cf. paragraphe 2.3.2) où l'interaction entre l'utilisateur est le système interactif est modélisée par une machine d'états finis à entrée/sortie, nous pouvons modéliser le comportement d'un système interactif comme suit :

$$\mathcal{B}_{\mathcal{IS}} =< I, O, S, Trans > \tag{4.1}$$

où :

- I est un ensemble d'entrées ;
- O est un ensemble de sorties ;
- S est un ensemble d'états ;
- $Trans$ est un ensemble de transitions :
  $Trans \subseteq S \times I \times (O \cup \{\perp\}) \times S$
  Le symbole $\perp$ signifie l'absence de sortie.

Exemple : modélisation de Memo

$$\mathcal{B}_{Memo} =< I_{Memo}, O_{Memo}, S_{Memo}, trans_{Memo} >$$

où :

- $I_{Memo} = \{move, turn, get, set, remove\}$ où
  move : l'utilisateur change sa localisation (quand il bouge)
  turn : l'utilisateur change son orientation (quand il tourne)
  get : l'utilisateur envoie une commande "get"
  set : l'utilisateur envoie une commande "set"
  remove : l'utilisateur envoie une commande "remove"
- $O_{Memo} = \{memoTaken, memoSet, memoRemoved\}$ où :
  memoTaken : l'application affiche le message "memo is taken"
  memoSet : l'application affiche le message "memo is set"

$memoRemoved$: l'application affiche le message "memo is removed"

- $S = \{s_1, s_2, s_3, s_4\}$ où :
  - $s_1 = memoDisplayed \wedge \neg memoCarried\}$
    état où l'application affiche une note et l'utilisateur ne porte pas de note
  - $s_2 = \neg memoDisplayed \wedge memoCarried$
    état où l'application n'affiche pas de note et l'utilisateur porte une note
  - $s_3 = memoDisplayed \wedge memoCarried$
    état où l'application affiche une note et l'utilisateur porte une note
  - $s_4 = \neg memoDisplayed \wedge \neg memoCarried$
    état où l'application n'affiche pas de note et l'utilisateur ne porte pas de note
- $trans(s_1, move, \bot, s_1)$, $trans(s_1, move, \bot, s_4)$,
  $trans(s_1, turn, \bot, s_1)$, $trans(s_1, turn, \bot, s_4)$,
  $trans(s_1, get, memoTaken, s_2)$, $trans(s_1, get, memoTaken, s_3)$,
  $trans(s_1, remove, memoRemoved, s_1)$, $trans(s_1, remove, memoRemoved, s_4)$,

  $trans(s_2, move, \bot, s_2)$, $trans(s_2, move, \bot, s_3)$,
  $trans(s_2, turn, \bot, s_2)$, $trans(s_2, turn, \bot, s_3)$,
  $trans(s_2, set, memoSet, s_1)$,
  $trans(s_2, remove, memoRemoved, s_4)$,

  $trans(s_3, move, \bot, s_2)$, $trans(s_3, move, \bot, s_3)$,
  $trans(s_3, turn, \bot, s_2)$, $trans(s_3, turn, \bot, s_3)$,
  $trans(s_3, set, memoSet, s_1)$,
  $trans(s_3, remove, memoRemoved, s_2)$, $trans(s_3, remove, memoRemoved, s_3)$,

  $trans(s_4, move, \bot, s_1)$, $trans(s_4, move, \bot, s_4)$,
  $trans(s_4, turn, \bot, s_1)$, $trans(s_4, turn, \bot, s_4)$.

4.4 Adéquation de l'approche synchrone

Dans [66], 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.

Afin de bien comprendre la manière de communiquer d'un système synchrone avec un système asynchrone, on peut se référer aux explications fournies dans [67] : un programme synchrone **P** progresse via des réactions :

```
run(P) = séquence des tuplets des événements
       = {(x_i(1))_{i=1,...,k}, (x_i(2))_{i=1,...,k}, .... }
```

Où $x_i(1)$ est l'événement x_i à l'instant 1.

Pendant une réaction, la décision peut être prise en testant l'absence de quelques signaux (dénotée par "$x = \perp$"). Par exemple, pour 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 $\perp$.

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) = tuplet de séquences des événements
       = (x_i(1), x_i(2), ...)_{i=1,...,k}
```

Dans le cas de l'exécution d'un système asynchrone, l'absence de valeurs n'a pas de sens ($\neg(x = \perp)$).

C'est pourquoi, désynchroniser une exécution est une application d'une séquence de tuplets de valeurs aux domaines étendus par $\perp$ dans un tuplet de séquences de valeurs présentes, une séquence étant associée à une variable (c.f. Figure 4.2).

Il est possible de construire une trace synchrone à partir d'une trace asynchrone, si on affecte à chaque signal s une horloge (variable) hs qui est toujours présente (voir la Figure 4.3). À 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.

Illustrons ces propos sur Memo. Un traducteur est introduit entre le côté synchrone et le côté asynchrone (cf. figure 4.4). 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, on affecte à

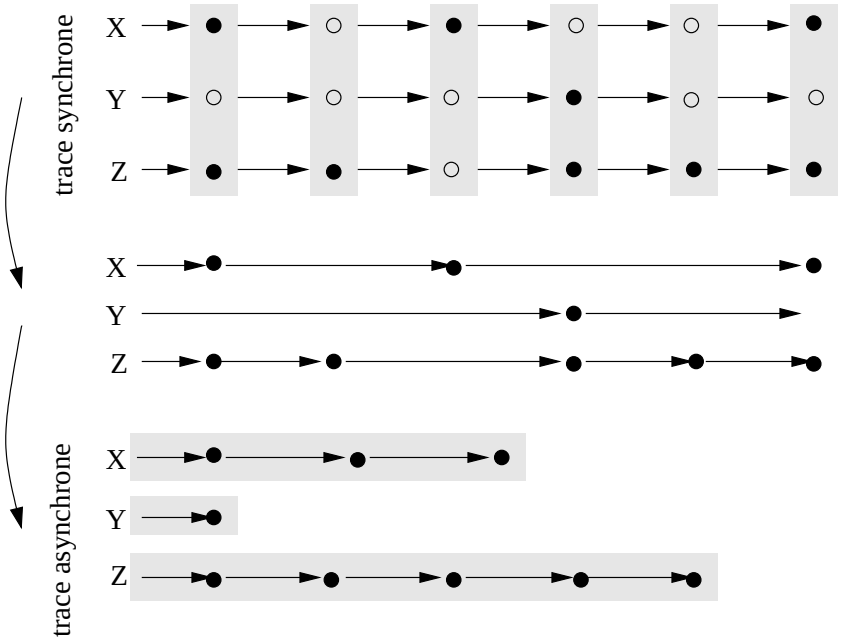

FIG. 4.2 – désynchronisation d'une trace synchrone

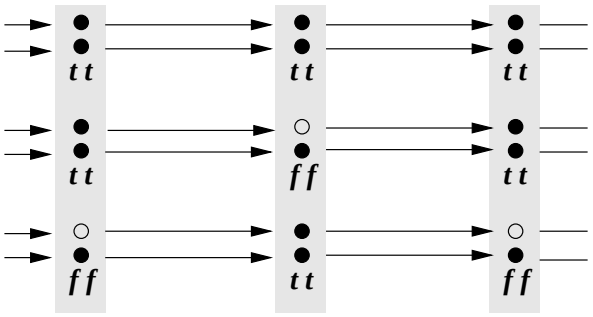

FIG. 4.3 – synchronisation d'une trace asynchrone

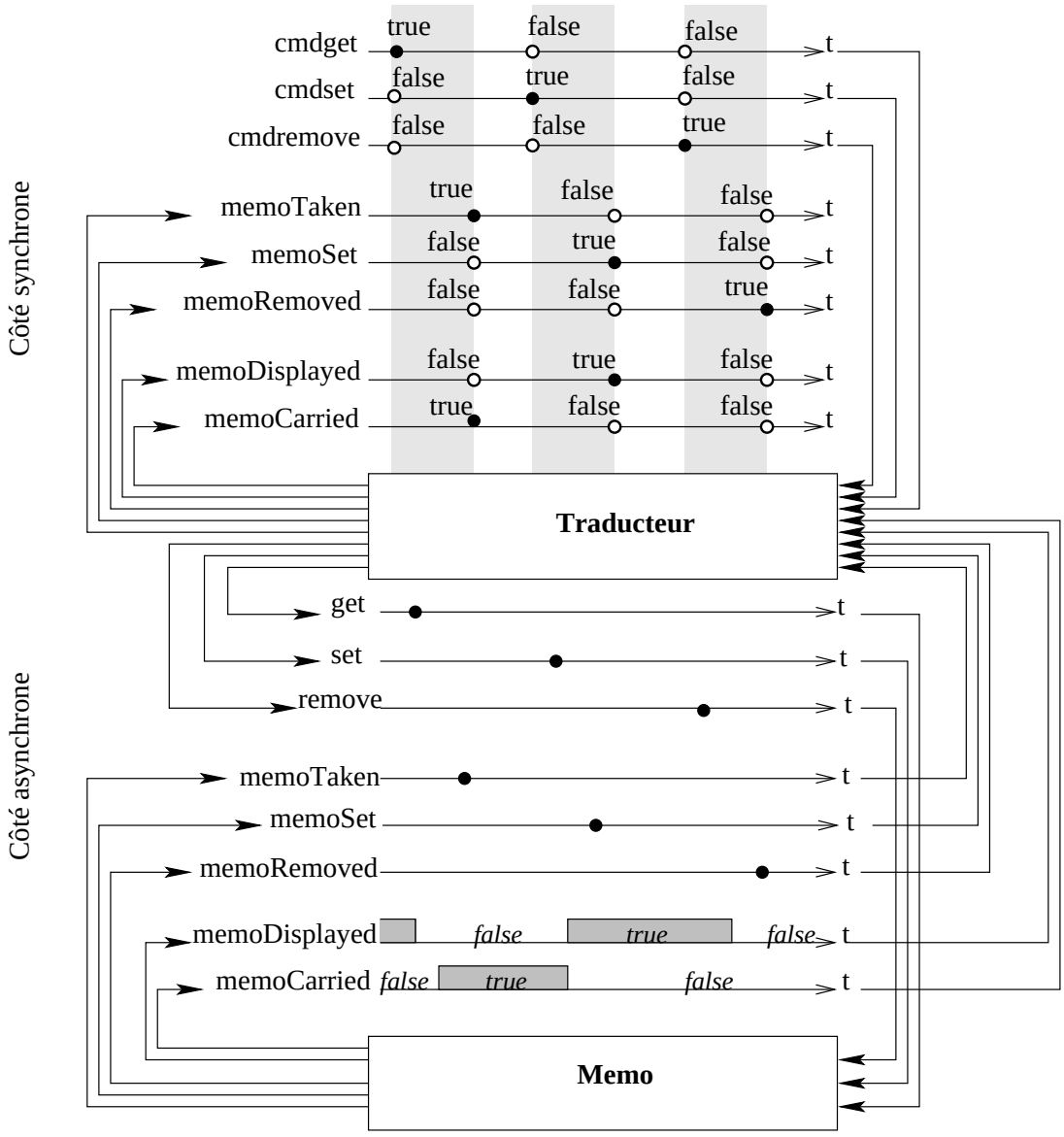

FIG. 4.4 – Transformation de trace de Memo en trace synchrone

chaque événement d'entrée ou de sortie une variable booléenne qui est toujours présente. Pour les entrées, à chaque cycle d'exécution, 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, sinon l'événement associé n'est pas présent. Ensuite, le traducteur attend les événements de sortie de l'application interactive. La durée d'attente est fixe pour chaque cycle. À la fin de chaque cycle, si un événement quelconque de sortie est arrivé pendant cet cycle, 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 arrivés dans cet cycle.

Les entrées de Memo sont : *"move"*, *"turn"*, *"get"*, *"set"*, *"remove"* (cf. paragraphe 4.3). Pour les événements d'entrée (*"get"*, *"set"*, *"remove"*), on a trois variables booléennes associées à ces événements : *cmdget, cmdset et cmdremove.* Pour les sorties de l'application, 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 : *memoTaken*, *memoSet* et *memoRemoved.*

Les deux entrées *"move"* et *"turn"* ne sont pas explicitement exprimées par l'utilisateur. En effet, les coordonnées sur les trois axes x, y et z sont changées avec le déplacement de l'utilisateur. De manière similaire, les trois angles d'orientation yaw, pitch et roll sont changés quand l'utilisateur tourne. Nous montrons comment nous traitons ces entrées dans les paragraphes 4.5.1.1 et 7.4).

Parfois, il est intéressant de récupérer les valeurs de certaines variables d'état de l'application interactive sous test (par exemple pour savoir si une note est portée par l'utilisateur). Une variable v_s est donc associée à chaque telle variable d'état s de l'application . À la fin de chaque cycle, le traducteur récupère la valeur de s et l'affecte à la variable v_s. Dans le cas de Memo, on a deux variables d'état booléennes *memoDisplayed* (qui signifie que l'application affiche une note sur le viseur) et *memoCarried* (qui signifie que l'utilisateur porte une note) comme le montre la figure 4.4.

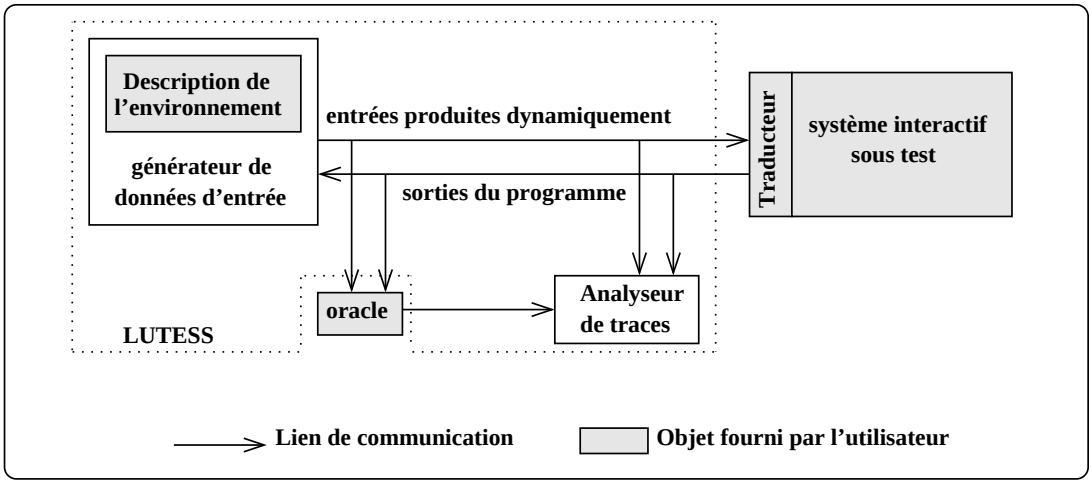

FIG. 4.5 – Test de système interactif avec Lutess

4.5 Test de systèmes interactifs avec Lutess

Pour tester un système interactif avec Lutess, trois entités sont nécessaires [68, 69] (cf. figure 4.5) :

1. Le système interactif comme un programme exécutable ainsi qu'un traducteur d'événements transformant les événements d'entrées et de sorties de l'application en événements booléens manipulés par Lutess.
2. La spécification en Lustre de l'oracle de test décrivant les propriétés à valider.
3. La spécification en Lustre de l'environnement externe du système interactif sous test, incluant éventuellement des directives de guidage.

4.5.1 Connexion Lutess-système interactif

Le traducteur est l'interface entre Lutess et le système interactif sous test. Comme on a montré dans la section 4.4, il prend en charge la désynchronisation des signaux d'entrée pour le système interactif et la synchronisation des signaux de sortie du système interactif.

Les signaux d'entrée et de sortie de l'application interactive sous test sont représentés sous la forme de flots booléens dans Lutess. Le traducteur traduit

les sorties booléennes de Lutess en entrée manipulées par le système interactif et inversement.

Le traducteur fonctionne selon l'algorithme suivant :

– Pour chaque cycle d'éxecution faire :

1. lire un vecteur de booléens générés par Lutess ;
2. envoyer les événements d'entrée correspondants à l'application interactive ;
3. attendre pour que l'application interactive réagisse (la durée d'attente dépend de l'application et est déterminée empiriquement) ;
4. récupérer les événements de sortie de l'application ;
5. envoyer les booléens correspondant à l'état et aux sorties de l'application à Lutess afin de générer un nouveau vecteur de booléens pour le cycle d'exécution suivant.

4.5.1.1 Exemple : Connexion de Memo à Lutess

Les entrées de Memo sont : $\{move, turn, get, set, remove\}$. Pour l'entrée *"move"*, l'utilisateur peut bouger dans l'espace, c'est-à-dire selon les trois axes x, y et z. De manière similaire pour l'entrée *"turn"*, l'utilisateur peut tourner avec les trois angles d'orientation yaw, pitch et roll.

Les entrées générées par Lutess sont les suivantes :

1. *Localisation* est un vecteur booléen indiquant le mouvement de l'utilisateur selon les directions x, y et z. À un instant donné, $Localisation[xplus] = true$ signifie que l'abscisse de l'utilisateur augmente. 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 selon 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. *cmdget*, *cmdset* et *cmdremove* sont des variables booléennes correspondant aux commandes *"get"*, *"set"* ou *"remove"*. À un instant, $cmdget = true$ indique que l'utilisateur envoie la commande *"get"* au système.

L'état de l'application Memo est observé par les cinq sorties booléennes suivantes :

1. *memoDisplayed*, qui est vrai quand au moins une note est affichée sur le viseur.
2. *memoCarried*, qui est vrai quand l'utilisateur porte une note.
3. *memoTaken*, qui est vrai si l'utilisateur vient de récupérer une note.
4. *memoSet*, qui est vrai si l'utilisateur vient de poser une note.
5. *memoRemoved*, qui est vrai si l'utilisateur vient de supprimer une note.

4.5.2 Spécification de l'oracle de test

4.5.2.1 Exemple : spécification de propriétés de Memo

Dans ce paragraphe nous présentons différentes propriétés de l'application Memo et la manière dont on peut les exprimer en Lustre afin de les intégrer dans un oracle automatique.

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 affichée (plus précisément, à un instant où une note est suffisamment proche de l'utilisateur pour être manipulée).

```
once_from_to(cmdget and pre memoDisplayed,
             memoDisplayed,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 memoDisplayed or pre memoCarried),
             memoDisplayed 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 tourné ou il a effectué une commande "get" ou "remove".

```
once_from_to((move or turn or cmdget or cmdremove),
                    memoDisplayed, not memoDisplayed)
```

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

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

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

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

– Seules les commandes "set" ou "remove" provoquent l'abandon d'une note portée.

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

4.5.2.2 Expressivité de Lustre

L'oracle de test consiste en des propriétés à valider par le système interactif. Dans le cas d'un système interactif, le but du test est de valider qu'une séquence d'événements produits par l'utilisateur (représentés ici 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.

Il est par contre impossible d'exprimer des propriétés de vivacité (« il est toujours possible de ... ») ainsi que des propriétés portant sur la valeur précise de données plus complexes (valeur de chaînes de caractères, par exemple).

4.5.3 Description de l'environnement et guidage

4.5.3.1 Exemple : spécification de l'environnement et du comportement de l'utilisateur de Memo

Lutess génère les entrées pour le système sous test en respectant des contraintes (invariants) 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 :

1. À un instant donné, l'utilisateur bouge ou manipule une note :

   ```
   (move or turn) xor (cmdget or cmdset or cmdremove )
   move = LIN_OR(6,Localisation)[1]
   turn = LIN_OR(6,Orientation)
   ```

   LIN_ OR(6, Localisation) (*LIN_ OR(6, Orientation)*) signifie que au moins une des six valeurs booléennes de localisation (orientation) est vraie.

2. L'utilisateur ne peut pas bouger sur un axe dans les deux directions en même temps. Les formules correspondantes sont les suivantes :

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

3. 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])
   ```

4. En ce qui concerne les commandes utilisateur servant à manipuler une note, on a trois commandes : "get", "set" et "remove". L'utilisateur ne peut pas envoyer plus d'une commande à un instant :

[1]–LIN_OR : un entier N, un tableau de N booléen→un booléen
–{LIN_OR(n,A) est vrai si au moins un élément de A est vrai}

```
node LIN_OR(const n :int ; A : bool^n) returns (OR : bool)
let
   OR = with n=1 then A[0]
        else A[0] or LIN_OR(n-1, A[1..n-1]) ;
tel
```

```
AtMostOne (cmdget, cmdset, cmdremove)[2]
```

4.5.3.2 Exemple : guidage de la génération de test de Memo

La simulation aléatoire des actions de l'utilisateur génère des entrées dans lesquelles chaque événement a la même probabilité d'arriver. Cela signifie que $Localisation[xminus]$ est généré autant de fois que $Localisation[xplus]$. 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 des profils opérationnels ou par des scénarios.

Profils opérationnels

Les profils opérationnels sont définis par des probabilités d'occurrence conditionnelles ou inconditionnelles des actions d'entrée. Les profils opérationnels peuvent être utilisés 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))
```

En associant des conditions aux probabilités, on peut spécifier qu'une commande "get" a une forte probabilité d'être générée quand l'application affiche une note sur le viseur :

```
proba ((cmdget, 0.8, pre memoDisplayed))
```

On peut également choisir des probabilités telles que l'utilisateur bouge très probablement quand il ne voit pas une note affichée, et fait très probablement une commande "get" sinon.

[2] AtMostOne(A, B, C) est *vrai* si au plus un élément parmi les variables A, B et C a la valeur *vrai*

1	-	-	-	-	-	-	-	mDis	-
2	-	-	-	-	-	-	get	-	mTak
3	-	z-	x+	-	y+	-	-	-	-
4	x-	-	-	z+	-	y-	-	-	-
5	-	-	x+	z+	y+	-	-	-	-
6	-	z-	x+	-	y+	-	-	-	-
7	-	z-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	mDis	-
9	-	-	-	-	-	-	get	-	mTak
10	x-	z-	-	-	-	-	-	-	-
11	-	z-	-	-	y+	-	-	-	-
12	x-	z-	-	-	-	y-	-	mDis	-
13	-	-	-	-	-	-	get	mDis	mTak
14	-	-	-	-	-	-	get	mDis	mTak
15	-	-	-	-	-	-	get	-	mTak
16	x-	z-	-	-	y+	-	-	mDis	-
17	-	-	-	-	-	-	get	-	mTak
18	x-	-	-	z+	y+	-	-	-	-
19	x-	z-	-	-	y+	-	-	-	-
20	-	-	x+	z+	y+	-	-	-	-

TAB. 4.1 – Un extrait de la trace de Memo avec guidage où :
"mDis" pour "memoDisplayed" et "mTak" pour "memoTaken"

```
proba( (cmdget, 0.9, pre memoDisplayed),
     (Localisation[xminus], 0.5, not pre memoDisplayed),
     (Localisation[zminus], 0.5, not pre memoDisplayed),
     (Localisation[xplus], 0.8, not pre memoDisplayed),
     (Localisation[zplus], 0.8, not pre memoDisplayed),
     (Localisation[yplus], 0.5, not pre memoDisplayed),
     (Localisation[yminus], 0.8, not pre memoDisplayed)) ;
```

Dans l'extrait de la trace illustré dans le tableau 4.1 , on peut observer que, quand il n'y a pas de note visible, l'utilisateur bouge, et quand une note est visible ("mDis" arrive dans le top précédent), l'utilisateur la récupère ("get", "mTak").

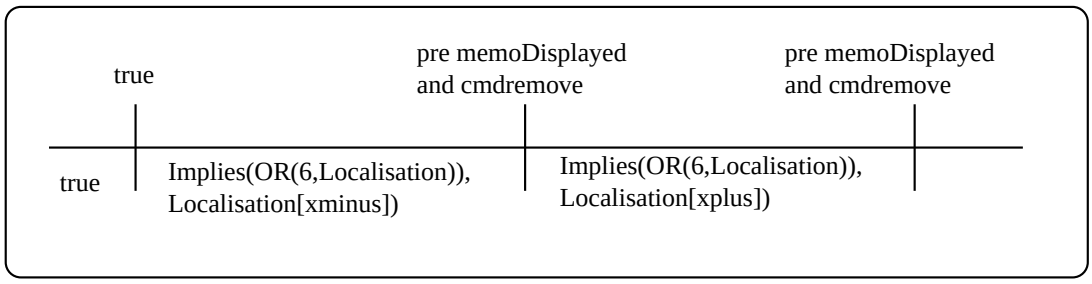

FIG. 4.6 – Exemple d'un scénario

Guidage par scénarios

Prenons l'exemple suivant de scénario : on considère quelques notes sur l'axe x ; l'utilisateur bouge selon l'axe x et supprime les notes ; quand l'utilisateur supprime une note, il change de direction (voir la figure 4.6). On décrit ce scénario de la façon suivante :

```
cond( true,
      pre memoDisplayed and cmdremove ,
      pre memoDisplayed and cmdremove)) ;
intercond(true,
      implies (LIN_OR (6, Localisation),
               Localisation[xminus]),     -- (1)
      implies (LIN_OR (6, Localisation),
               Localisation[xplus])) ;     -- (2)
```

LIN_OR(6, Localisation) signifie qu'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).

4.6 Conclusion

L'utilisation de Lutess pour le test de systèmes interactifs est fondée sur l'hypothèse suivante : une application interactive peut être assimilée, sous certaines conditions, à un programme synchrone. En basant sur cette hypothèse, nous utilisons Lutess pour tester automatiquement les systèmes interactifs.

Cette approche permet de valider la partie contrôle du système interactif. Elle permet de tester qu'une succession d'événements d'entrée conduit à une séquence adéquate d'événements de sortie, ce qui permet de valider la dynamique de l'interface.

En revanche, on ne peut pas tester des aspects plus sophistiqués liés à des informations textuelles, des images statiques ou des images vidéo (dans Memo, par exemple, on ne peut pas tester la propriété « quand un post-it est posé, alors son texte est le même que celui du dernier post-it récupéré ».

En ce qui concerne *l'utilisabilité* (cf. paragraphe 2.1), il est possible d'écrire des scénarios pour tester certaines des propriétés associées. Par exemple, pour tester l'atteignabilité, on peut spécifier un scénario décrivant les actions permettant de naviguer entre les différents états de l'application (dans Memo, l'action *"set"* change l'état de *"not memoCarried"* à *"memoCarried"*).

En résumé, l'utilisation de techniques de génération de Lutess permettent de simuler des scénarios de test pertinents. Par exemple, en utilisant un profil opérationnel et en s'inspirant de [53] (cf. paragraphe 2.2.6), l'application peut être testée avec différents profils (utilisateur expert, utilisateur naïf (comportement aléatoire)....).

Nous pouvons cependant noter que le test avec Lutess requiert une spécification partielle du comportement de l'utilisateur en Lustre. Ce modèle synchrone n'est pas facile à construire pour les concepteurs des applications interactives, non familiers de ce langage. Une manière de pallier cet inconvénient est de s'appuyer sur des modèles qui sont plus habituels dans le processus de développement des applications interactives, les *arbres de tâches*.

L'arbre de tâches est souvent utilisé par les concepteurs dans le processus de développement des applications interactives. Il décrit l'interaction entre l'application interactive et l'utilisateur. Il comporte ainsi des informations sur le comportement de ce dernier. L'utilisation de ce modèle en vue de générer des tests pour les systèmes interactifs fait l'objet du chapitre suivant.

Chapitre 5

Génération de tests à partir d'arbres de tâches

Dans l'approche basée sur l'utilisation de Lutess présentée dans le chapitre précédent, la description d'environnement est transformée en une machine d'états finis à Entrées/Sorties utilisée afin de générer les données de test. Dans ce chapitre, nous proposons d'utiliser l'arbre de tâches comme un langage de description d'environnement du système interactif et nous étudions la transformation de l'arbre de tâches en une machine d'états finis similaire à celle utilisée par Lutess. Cette machine pourra ainsi être exploitée par Lutess pour la génération de tests (voir la figure 5.1). Elle est construite conformément à une sémantique formelle définie à cet effet pour tous les opérateurs de l'arbre de tâche.

Nous étudions également l'enrichissement de l'arbre de tâches avec des probabilités afin de rendre possible la spécification de profils opérationnels directement

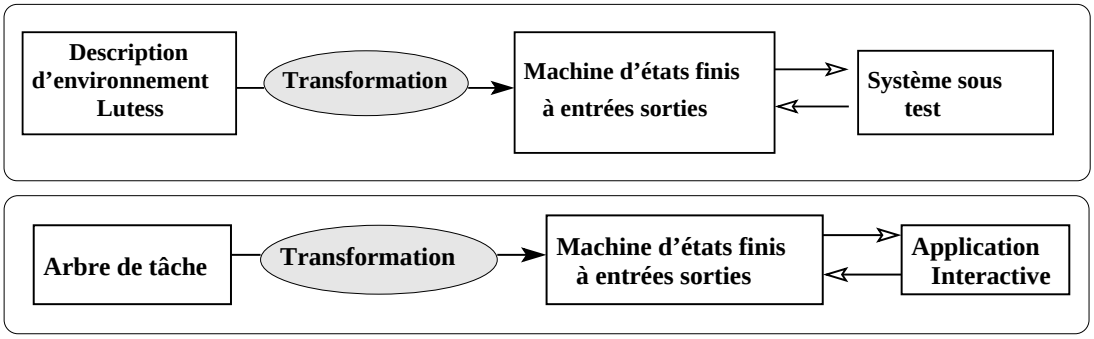

FIG. 5.1 – Test de Lutess et test à partir d'arbres de tâches

sur ce modèle. Ensuite, nous détaillons la manière dont ce modèle peut être exploité pour la génération de tests.

Comme nous l'avons annoncé au chapitre 2, nous avons retenu la notation CTT *Concur Task Trees*. Cette notation est riche en termes d'opérateurs et présente l'avantage d'être bien connue et outillée.

La notation CTT est inspirée de LOTOS. Dans [44] il est proposé de transformer CTT en spécification LOTOS qui est transformée en une machine d'états finis. Puis, des propriétés peuvent être vérifiées sur ce modèle par model-checking. Notre approche consiste plutôt en la transformation de CTT en un modèle qui sert à générer les données du test.

Nous détaillons d'abord l'extraction d'un modèle de l'utilisateur à partir d'arbre de tâches CTT (paragraphe 5.1). Ensuite, nous présentons le moyen de simuler ce modèle afin de générer des tests (paragraphe 5.2). Enfin, le paragraphe 5.3 expose la technique de test à partir de profils opérationnels spécifiés sur l'arbre de tâches CTT.

5.1 Extraction du modèle de l'utilisateur

5.1.1 Définitions et hypothèses préliminaires

Comme il a été mentionné dans le paragraphe 2.2.1.1, une tâche en notation CTT peut être une tâche usager, une tâche abstraite, une tâche application ou une tâche interactive.

Les tâches usager ne sont pas intéressantes du point de vue de la génération tests, car elles ne correspondent à aucune interaction avec le système (ni action ni réaction).

Nous assimilons une tâche application o à une machine d'états élémentaire à deux états, dont la seule transition consiste à passer de l'état initial à l'état final en émettant une sortie. Formellement, une tâche application est modélisée par la machine $M_o = (Q_o, qi_o, qf_o, I_o, O_o, trans_o)$ où :

- $Q_o = \{qi_o, qf_o\}$ est un ensemble d'états ; qi_o, qf_o sont respectivement l'état initial et l'état final de M_o
- $I_o = \{\mu\}$ est l'ensemble des entrées ; μ est une entrée vide (l'utilisateur ne procède à aucune action).

- $O_o = \{o\}$ est l'ensemble des sorties.
- $trans_o = \{qi_o \xrightarrow{\mu/o} qf_o\}$ est l'ensemble des transitions. La notation $qi_o \xrightarrow{\mu/o} qf_o$ signifie qu'il y a une transition de l'état qi_o vers l'état qf_o avec l'entrée vide μ et la sortie o.

La principale hypothèse de notre approche concerne les tâches interactives. Nous supposons qu'une tâche interactive est assimilée à une action élémentaire suivie d'une réaction. Elle est modélisée par une machine d'états finis à E/S avec une seule transition étiquetée avec cette action et sa réaction à partir de l'état initial vers l'état final. Cette modélisation doit être fournie en même temps que l'arbre des tâches.

Formellement, pour une tâche interactive T, on définit la machine d'états finis à E/S $M_T = (Q_T, qi_T, qf_T, I_T, O_T, trans_T)$ où :

- Q_T est un ensemble d'états.
- qi_T est l'état initial. C'est un état source.
- qf_T est l'état final. C'est un état puits.
- I_T est l'ensemble des entrées de l'application pour la tâche T.
- O_T est l'ensemble de sorties de l'application pour la tâche T.
- $trans_T \subseteq Q_T \times (I_T \cup \{\mu\}) \times O_T \times Q_T$ est l'ensemble de transitions de la tâche T. Si $(q_T, a, b, p_T) \in trans_T$, on écrit $q_T \xrightarrow{a/b} p_T$. On peut omettre l'entrée et la sortie de la transition : $q_T \xrightarrow{c} p_T$ signifie $c = a/b$.

On utilise la notation suivante : $I_T^\mu = I_T \cup \{\mu\}$.

Enfin, une tâche abstraite est composée d'autres sous-tâches liées par des opérateurs de CTT. Une machine à E/S peut aussi être associée à une tâche abstraite. Elle résulte de la composition des machines à E/S de ses sous-tâches comme expliqué dans le paragraphe suivant.

5.1.2 Transformation d'une tâche abstraite en une machine à E/S

Nous supposons que les ensembles d'états des machines associées à deux tâches distinctes de l'arbre de tâches sont disjoints. Cette hypothèse n'affecte pas la généralité de l'approche mais simplifie l'expression de la composition.

5.1.2.1 Notation

Pour une tâche T quelconque, on utilise les notations suivantes :
- $Q_T^{-fin} = Q_T \setminus \{qf_T\}$
- $Q_T^{-init} = Q_T \setminus \{qi_T\}$
- $Q_T^{-init-fin} = Q_T \setminus \{qi_T, qf_T\}$

5.1.2.2 Opérateur d'activation ($A>>B$ ou $A[]>>B$)

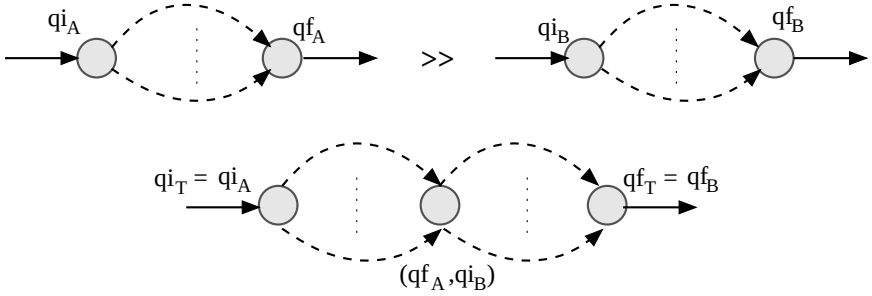

Figure 5.2: Sémantique de l'opérateur d'activation ">>"

Soit $T{=}A>>B$ ou $T{=}A[]>>B$. Nous considérons que les informations passant de la tâche A à la tâche B, lorsque l'opérateur $[]>>$ est utilisé, n'ont pas d'intérêt du point de vue de la génération de données de test, car cette communication est interne à l'application.

Puisque B commence dès que A termine, l'état final de A, qf_A, sera confondu avec l'état initial de B, qi_B, et ainsi on obtient un nouvel état (qf_A, qi_B). La figure 5.2 illustre la définition formelle suivante :

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A) >> M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A)[] >> M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$

$Q_T = Q_A^{-fin} \cup Q_B^{-init} \cup \{(qf_A, qi_B)\}$,
$qi_T = qi_A$, $qf_T = qf_B$,
$I_T = I_A \cup I_B$, $O_T = O_A \cup O_B$,
La relation $trans_T$ est définie comme suit :

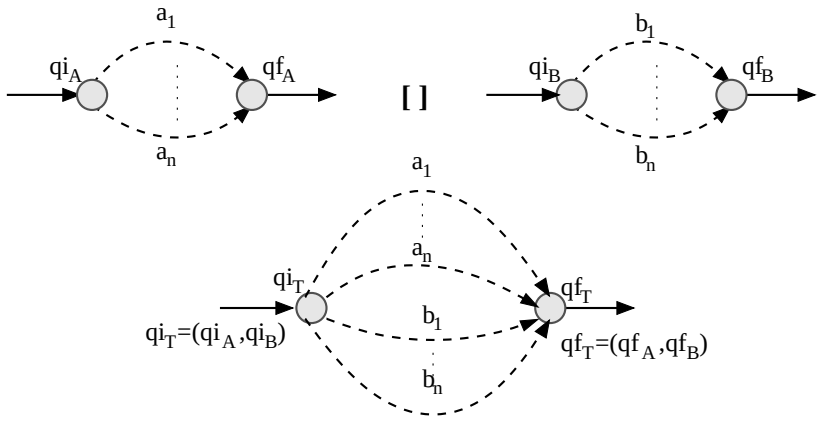

Figure 5.3: Sémantique de l'opérateur de choix "[]"

$$q \xrightarrow{c}_T s \text{ ssi } \begin{cases} soit \quad q \xrightarrow{c}_A s & et \quad q, s \in Q_A^{-fin} & & \\ soit \quad q \xrightarrow{c}_A qf_A & et \quad q \in Q_A^{-fin} & et \quad s = (qf_A, qi_B) & \\ soit \quad qi_B \xrightarrow{c}_B s & et \quad q = (qf_A, qi_B) & et \quad s \in Q_B^{-init} & \\ soit \quad q \xrightarrow{c}_B s & et \quad q, s \in Q_B^{-init} & & \end{cases}$$

5.1.2.3 Opérateur de Choix ($A[]B$)

Soit $T = A[]B$. T est réalisée en choisissant une tâche entre A et B. Ainsi, la tâche T commence lorsque soit A commence, ou soit B commence, et elle se termine lorsque la tâche choisie se termine. En conséquence, l'état initial de T, qi_T, sera la combinaison de deux états initiaux qi_A et qi_B ($qi_T = (qi_A, qi_B)$). De plus, l'état final de T, qf_T, sera également la combinaison de deux états finals ($qf_T = (qf_A, qf_B)$). La figure 5.3 illustre cette définition formelle :

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A)[]M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$

$Q_T = Q_A^{-init-fin} \cup Q_B^{-init-fin} \cup \{(qi_A, qi_B), (qf_A, qf_B)\}$,
$qi_T = (qi_A, qi_B)$, $qf_T = (qf_A, qf_B)$,
$I_T = I_A \cup I_B$, $O_T = O_A \cup O_B$,
La relation $trans_T$ est définie comme suit :

$$q \xrightarrow{c}_T s \text{ ssi} \begin{cases} soit \quad qi_A \xrightarrow{c}_A s & et \quad q = (qi_A, qi_B) & et \quad s \in Q_A^{-init-fin} \\ soit \quad q \xrightarrow{c}_A s & et \quad q, s \in Q_A^{-init-fin} & \\ soit \quad q \xrightarrow{c}_A qf_A & et \quad q \in Q_A^{-init-fin} & et \quad s = (qf_A, qf_B) \\ soit \quad qi_A \xrightarrow{c}_A qf_A & et \quad q = (qi_A, qi_B) & et \quad s = (qf_A, qf_B) \\ soit \quad qi_B \xrightarrow{c}_B s & et \quad q = (qi_A, qi_B) & et \quad s \in Q_B^{-init-fin} \\ soit \quad q \xrightarrow{c}_B s & et \quad q, s \in Q_B^{-init-fin} & \\ soit \quad q \xrightarrow{c}_B qf_B & et \quad q \in Q_B^{-init-fin} & et \quad s = (qf_A, qf_B) \\ soit \quad qi_B \xrightarrow{c}_B qf_B & et \quad q = (qi_A, qi_B) & et \quad s = (qf_A, qf_B) \end{cases}$$

5.1.2.4 Opérateur d'entrelacement (Concurrence indépendante) ($A \,|||\, B$)

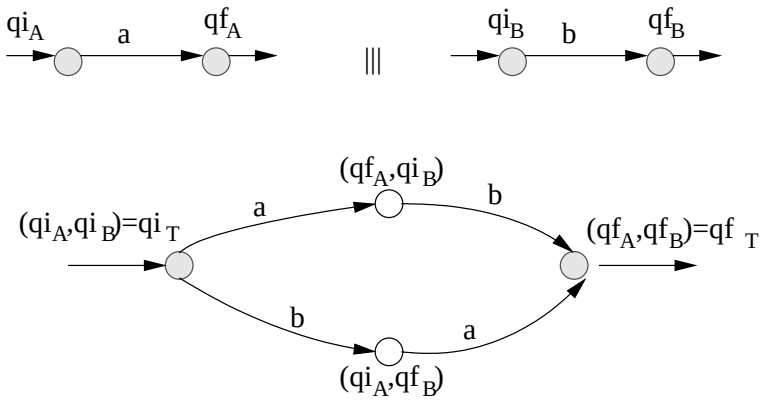

Figure 5.4: Un exemple de l'opérateur d'entrelacement "|||"

Soit $T = A \,|||\, B$. La figure 5.4 montre un exemple d'utilisation de la définition formelle de la machine résultante ci-dessous :

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A)|||M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$

$Q_T = Q_A \times Q_B$, $qi_T = (qi_A, qi_B)$, $qf_T = (qf_A, qf_B)$, $I_T = I_A \cup I_B$, $O_T = O_A \cup O_B$

La relation $trans_T$ est définie comme suit :

$$(q_A, q_B) \xrightarrow{c}_T (s_A, s_B) \; ssi \begin{cases} soit \quad (q_A \xrightarrow{c}_A s_A) & et \quad q_B = s_B \\ soit \quad (q_B \xrightarrow{c}_B s_B) & et \quad q_A = s_A \end{cases}$$

5.1.2.5 Opérateur de désactivation ($A[> B$)

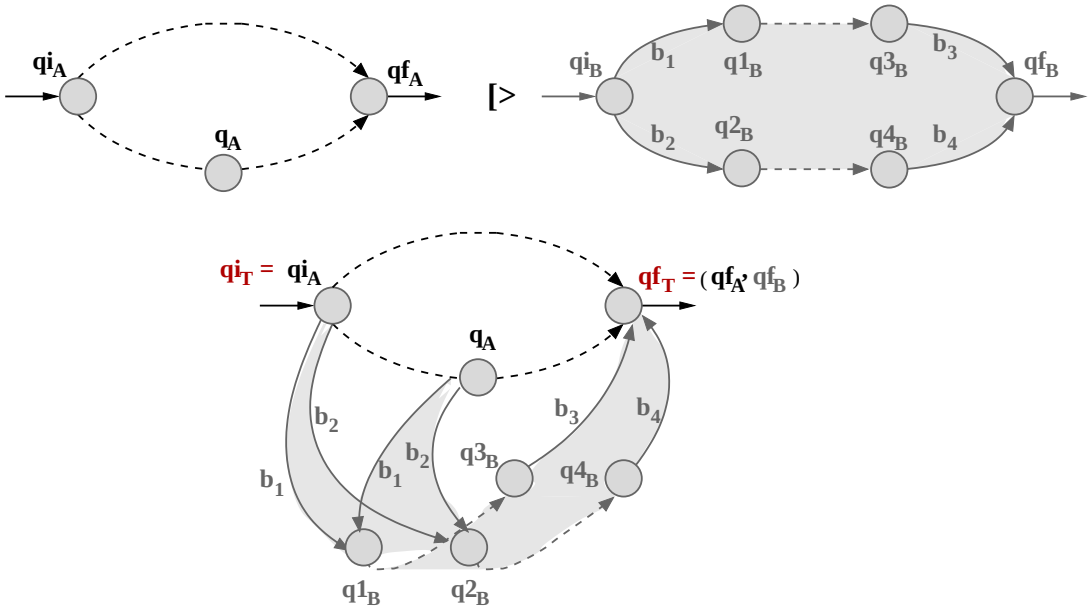

Figure 5.5: Sémantique de l'operateur de désactivation "[>"

Soit $T = A[> B$. Dans la machine à E/S de T, M_T, puisque la tâche T se termine lorsque soit A a terminé sans interruption, soit B s'est terminée après avoir interrompu A, l'état final de T, qf_T, sera la combinaison de deux états finals de A et de B ($qf_T = (qf_A, qf_B)$).

Dans la machine M_T, on a trois groupes de transitions :

- L'ensemble de transitions $trans_{A'}$. Les transitions dans cet ensemble correspondent aux transitions de M_A.
- L'ensemble de transitions $trans_{AB}$. Les transitions dans cet ensemble correspondent aux premières actions de la machine M_B qui peuvent interrompre la tâche A.
- L'ensemble de transitions $trans_{B'}$. Les transitions dans cet ensemble correspondent à la continuation de la tâche B après avoir interrompu A.

La figure 5.5 illustre la définition formelle suivante :

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A)[> M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$

$Q_T = Q_A^{-fin} \cup Q_B^{-init-fin} \cup \{(qf_A, qf_B)\}$,
$qi_T = qi_A$, $qf_T = (qf_A, qf_B)$,
$I_T = I_A \cup I_B$, $O_T = O_A \cup O_B$,
$trans_T = trans_{A'} \cup trans_{AB} \cup trans_{B'}$

Où la relation $trans_{A'} \subseteq Q_A^{-fin} \times I_A^{\mu} \times O_A \times (Q_A^{-fin} \cup (qf_A, qf_B))$ est définie ci-dessous :

$$q_A \xrightarrow{a}_{A'} q \text{ ssi } \begin{cases} soit \quad q_A \xrightarrow{a}_A q & et \quad q \in Q_A^{-fin} \\ soit \quad q_A \xrightarrow{a}_A qf_A & et \quad q = (qf_A, qf_B) \end{cases}$$

La relation $trans_{AB} \subseteq Q_A^{-fin} \times I_B^{\mu} \times O_B \times (Q_B^{-init-fin} \cup (qf_A, qf_B))$ est définie ci-dessous :

$$q_A \xrightarrow{b}_{AB} q \text{ ssi } \begin{cases} soit \quad qi_B \xrightarrow{b}_B q & et \quad q \in Q_B^{-init-fin} \\ soit \quad qi_B \xrightarrow{b}_B qf_B & et \quad q = (qf_A, qf_B) \end{cases}$$

La relation $trans_{B'} \subseteq Q_B^{-init-fin} \times I_B^{\mu} \times O_B \times (Q_B^{-init-fin} \cup (qf_A, qf_B))$ est définie ci-dessous :

$$q_B \xrightarrow{b}_{B'} q \text{ ssi } \begin{cases} soit \quad q_B \xrightarrow{b}_B q & et \quad q \in Q_B^{-init-fin} \\ soit \quad q_B \xrightarrow{b}_B qf_B & et \quad q = (qf_A, qf_B) \end{cases}$$

5.1.2.6 Opérateur suspendre-reprendre ($A \mid > B$)

Soit $T = A \mid > B$. Dans la machine M_T résultat de la composition, on a quatre groupes de transitions :

- $trans_A$: les transitions de la tâche A qui peut être suspendue par B.
- $trans_{AB}$: Les transitions correspondent aux premières actions de la machines M_B qui peuvent suspendre la tâche A en mémorisant l'état de départ (q_A, s_B).
- $trans_{B'}$: Les transitions dans cet ensemble correspondent à la continuation de la tâche B après avoir suspendu A en mémorisant l'état de départ.
- $trans_{T'}$: Les transitions à partir de l'état initial qi_T. Cet état a été ajouté parce que l'état initial d'une tâche est par définition un état source. En effet, à cause de l'opérateur "suspendre-reprendre", il y a des transitions vers l'état qi_A (ces transitions correspondent aux dernières actions de la tâche B qui peut suspendre la tâche A). L'état qi_T ajouté est un état source et les mêmes actions, qui peuvent s'exécuter à partir de l'état qi_A, peuvent également s'exécuter à partir de l'état qi_T.

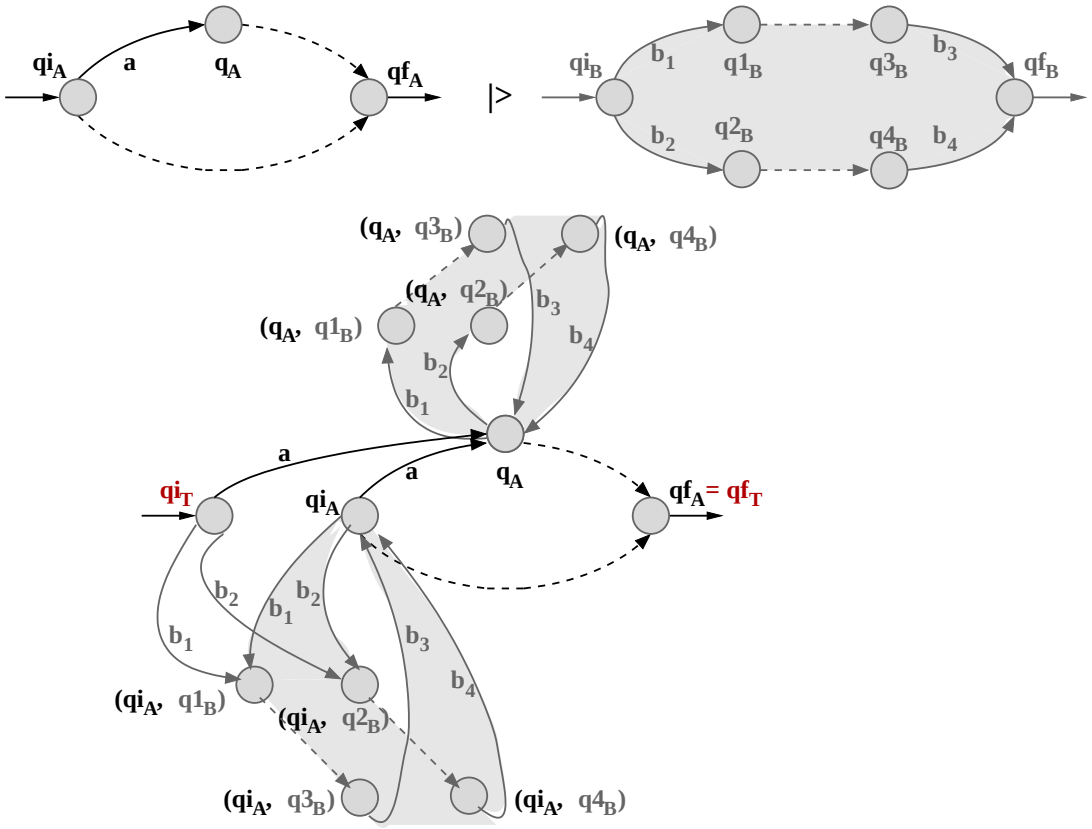

Figure 5.6: Sémantique de l'opérateur suspendre-reprendre "|>"

La figure 5.6 illustre la composition entre les machines à E/S de deux tâches A et B définie formellement comme suit :

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A) \mid > M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$

$Q_T = \{qi_T\} \cup Q_A \cup (Q_A^{-fin} \times Q_B^{-init-fin})$,
$qf_T = qf_A$,
$I_T = I_A \cup I_B$, $O_T = O_A \cup O_B$,
$trans_T = trans_A \cup trans_{AB} \cup trans_{B'} \cup trans_{T'}$,
Où la relation $trans_{AB} \subseteq Q_A^{-fin} \times I_B^{\mu} \times O_B \times (Q_A^{-fin} \cup (Q_A^{-fin} \times Q_B^{-init-fin}))$ est définie ci-dessous :
$q_A \xrightarrow{b}_{AB} (s_A, s_B) \quad ssi \quad qi_B \xrightarrow{b}_B s_B \; et \; q_A = s_A$

$q_A \xrightarrow{b}_{AB} s_A \quad ssi \quad qi_B \xrightarrow{b}_B qf_B \; et \; q_A = s_A$

Et la relation $trans_{B'} \subseteq (Q_A^{-fin} \times Q_B^{-init-fin}) \times I_B^{\mu} \times O_B \times (Q_A^{-fin} \cup (Q_A^{-fin} \times Q_B^{-init-fin}))$ est définie ci-dessous :

$(q_A, q_B) \xrightarrow{b}_{B'} (s_A, s_B) \quad ssi \quad q_B \xrightarrow{b}_B s_B \; et \; q_A = s_A$

$(q_A, q_B) \xrightarrow{b}_{B'} s_A \quad ssi \quad q_B \xrightarrow{b}_B qf_B \; et \; q_A = s_A$

Et la relation $trans_{T'} \subseteq \{qi_T\} \times I_T^{\mu} \times (Q_A \cup (\{qi_A\} \times Q_B^{-init-fin}))$ est définie ci-dessous :

$$qi_T \xrightarrow{c}_{T'} s_A \; ssi \begin{cases} soit \quad qi_A \xrightarrow{c}_A s_A \\ soit \quad qi_A \xrightarrow{c}_{AB} qi_A \quad et \quad s_A = qi_A \end{cases}$$

$qi_T \xrightarrow{b}_{T'} (qi_A, s_B) \; ssi \; qi_A \xrightarrow{b}_{AB} (qi_A, s_B) \; ssi \; qi_B \xrightarrow{b}_B s_B$

5.1.2.7 Opérateur d'itération (A^*)

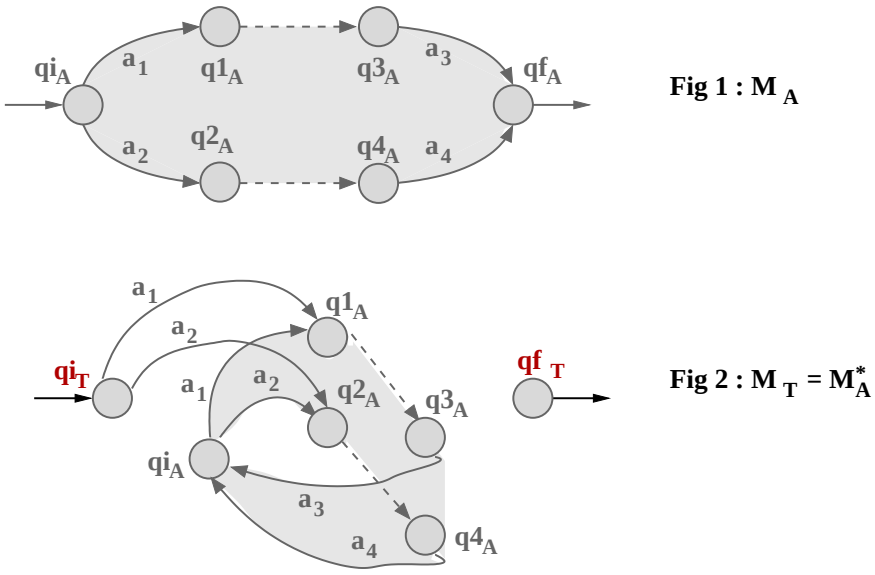

Figure 5.7: Sémantique de l'opérateur itération "*"

Soit $T = A^*$. Dans la machine à E/S correspondant à la tâche T, il y a deux types de transitions :

- $trans_{A'}$: transitions correspondant aux transitions de M_A après avoir remplacé l'état final par l'état initial.
- $trans_{T'}$: transitions à partir de l'état initial qi_T, ajouté parce que l'état initial d'une tâche est par définition un état source. Les mêmes actions, qui

peuvent s'exécuter à partir de l'état qi_A, peuvent également s'exécuter à partir de l'état qi_T.

La figure 5.7 donne une représentation graphique de la définition formelle suivante :

$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T) = M_A(Q_A, qi_A, qf_A, I_A, O_B, trans_A)^*$

$Q_T = Q_A^{-fin} \cup \{qi_T, qf_T\}$

$I_T = I_A$, $O_T = O_A$,

$trans_T = trans_{A'} \cup trans_{T'}$

Où la relation $trans_{A'} \subseteq Q_A^{-fin} \times I_A^{\mu} \times O_A \times Q_A^{-fin}$ est définie ci-dessous :

$$q_A \xrightarrow{a}_{A'} s_A \text{ ssi } \begin{cases} soit \quad q_A \xrightarrow{a}_A s_A \\ soit \quad q_A \xrightarrow{a}_A qf_A \quad et \quad s_A = qi_A \end{cases}$$

Et la relation $trans_{T'} \subseteq \{qi_T\} \times I_A^{\mu} \times O_A \times Q_A^{-fin}$ est définie ci-dessous :

$$qi_T \xrightarrow{a}_{T'} s_A \text{ ssi } qi_A \xrightarrow{a}_{A'} s_A \text{ ssi } \begin{cases} soit \quad qi_A \xrightarrow{a}_A s_A \\ soit \quad qi_A \xrightarrow{a}_A qf_A \quad et \quad s_A = qi_A \end{cases}$$

Notons que l'état final de la tâche itérative qf_T n'est pas atteignable, parce que, par définition, l'itération est infinie (elle continue jusqu'à ce qu'elle soit désactivée par une autre tâche).

5.1.2.8 Opérateur d'itération finie ($A(n)$)

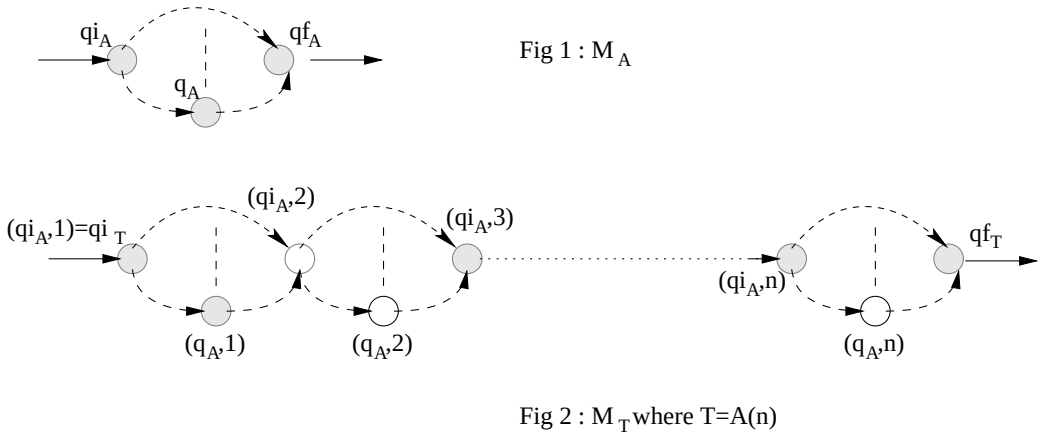

Figure 5.8: Sémantique de l'itération finie

On considère deux tâches A et T et un entier n, tels que : $T = A(n); n > 1$. Un compteur (i) est utilisé dans la définition de cette machine. Pendant la répétition

de la tâche A (quand $1 \leq i \leq n-1$), l'état final de A dans la *ième* itération sera remplacé par l'état initial dans la *(i+1)ème* itération.

La figure 5.8 illustre la définition formelle suivante :

$M_T(Q_T, qi_T, qf_T, I, O, trans_T) = M_A(Q_A, qi_A, qf_A, I, O, trans_A)(n)$

$Q_T = \{(q_A, i) \mid q_A \in Q_A^{-fin}, i \in [1..n]\} \cup \{qf_T\}$

$qi_T = (qi_A, 1)$

Où la relation $trans_T$ est définie ci-dessous :

$(q_A, i) \xrightarrow{a}_T (s_A, j)$ ssi $\begin{cases} soit \quad q_A \xrightarrow{a}_A s_A \quad et \quad i = j \\ soit \quad q_A \xrightarrow{a}_A qf_A \quad et \quad j = i+1 \; et \; s_A = qi_A \end{cases}$

$(q_A, i) \xrightarrow{a}_T qf_T$ ssi $q_A \xrightarrow{a}_A qf_A$ et $i = n$

5.1.2.9 La tâche optionnelle ($[A]$)

Les tâches optionnelles doivent être utilisées avec les opérateurs d'activation ($[A] >> B = (A >> B)[]B$) ou de concurrence ($[A]|||B = (A|||B)[]B$) [42, 41]. La machine de la tâche optionnelle est construite à partir des machines des autres tâches selon l'opérateur de la composition et en utilisant la sémantique définie ci-dessus.

5.1.2.10 Exemples

La figure 5.9 montre un exemple d'arbre de tâches avec la machine d'états finis à E/S associée. Dans cet arbre de tâches, toutes les feuilles sont des tâches interactives : *"WriteName", "WritePhonenr", "WritePassword", "Login" et "Quit"*. Chacune de ces tâches est représentée par une machine à deux états avec une seule transition dont l'action d'entrée est la tâche elle-même et la sortie n'est pas précisée.

En appliquant la sémantique des opérateurs définie précédemment, on obtient la machine à E/S illustrée dans la figure 5.9.

Considérons maintenant l'application interactive Memo (présentée dans le paragraphe 4.2). La figure 5.10 présente l'arbre des tâches de cette application dans la notation CTT. Cet arbre montre que l'utilisateur peut utiliser l'application itérativement (opérateur d'itération *) et peut être interrompu (opérateur de désactivation [>) par la tâche *"exit"*. La tâche *"use memo system"* est un choix entre les trois tâches suivantes : découvrir le terrain (*"explore the ground"*),

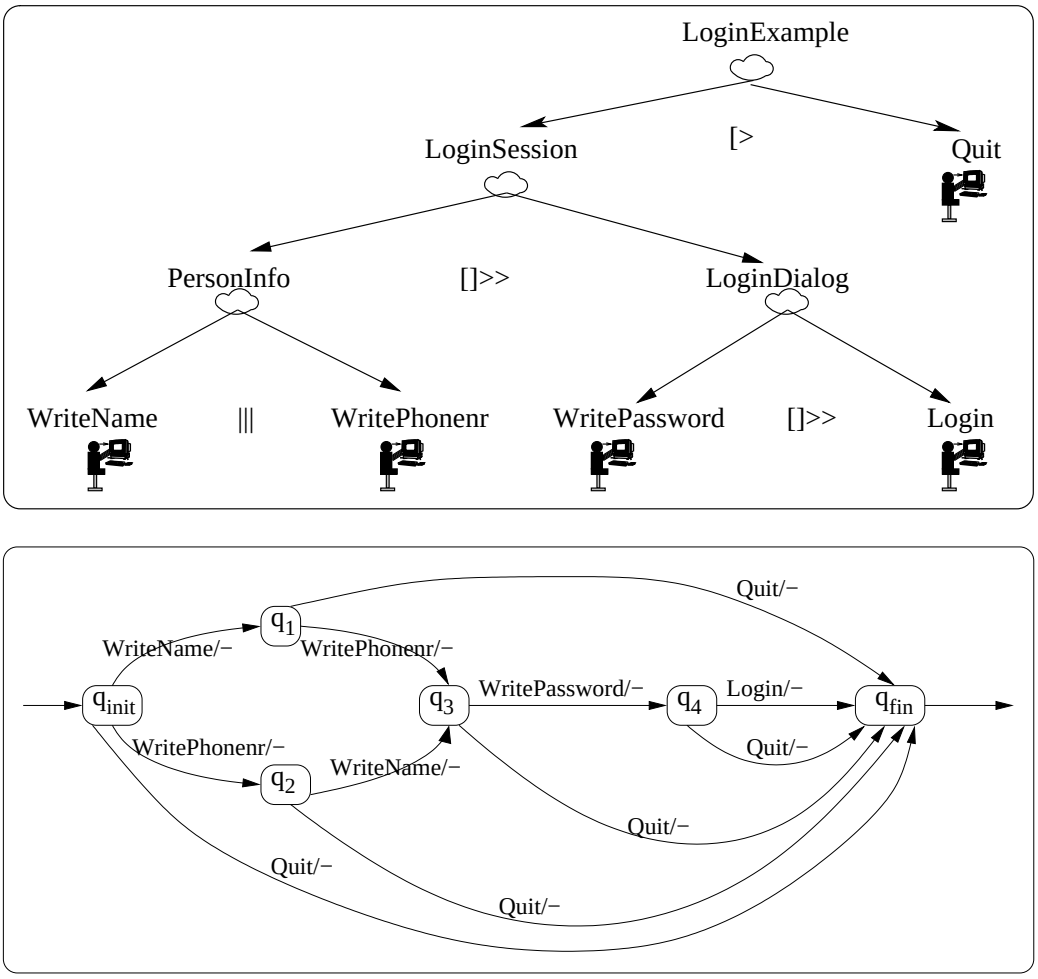

FIG. 5.9 – La machine à E/S pour l'arbre de tâche (Login Exemple)

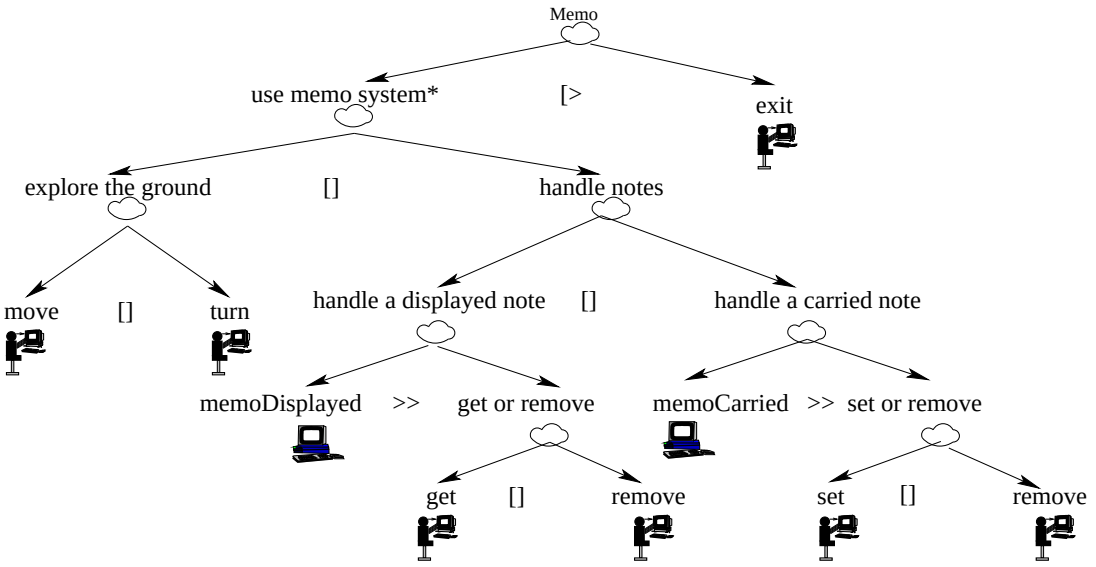

FIG. 5.10 – L'arbre de tâches de Memo

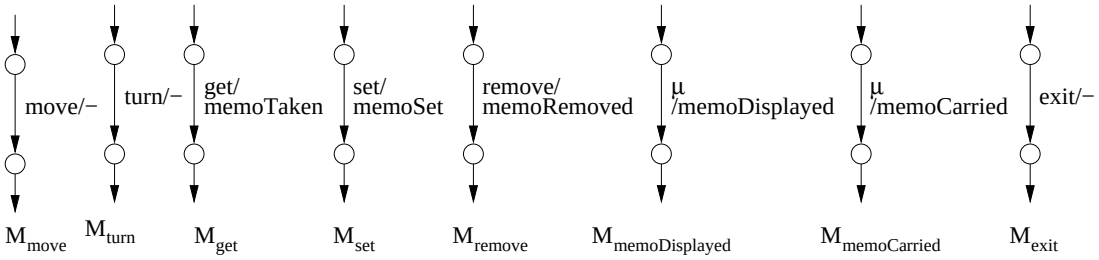

FIG. 5.11 – Les machines à E/S pour les tâches : *"move", "turn", "get", "set", "remove", "memoDisplayed", "memoCarried" et "exit"*

manipuler une note visible (*"handle a displayed note"*) ou manipuler une note portée (*"handle a carried note"*). Si le système affiche une note (*"memoDisplayed"*), l'utilisateur peut (opérateur d'activation $>>$) récupérer ou supprimer cette note. Si l'utilisateur est en train de porter une note (*"memoCarried"*), l'utilisateur peut la placer ou la supprimer.

Dans cet arbre de tâches, la tâche abstraite *"Memo"* est définie à l'aide de six tâches interactives : *"get", "set", "remove", "move", "turn", "exit"* et deux tâches application : *"memoDisplayed", "memoCarried"*. Ces tâches sont modélisées par les machines à E/S illustrées par la figure 5.11. Lorsque l'utilisateur envoie la commande *"get"* sur une note affichée au système Memo, l'application

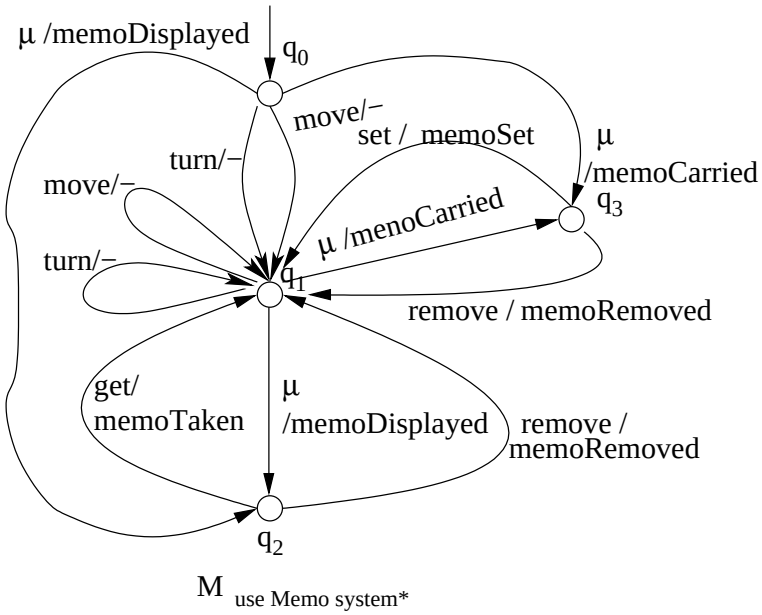

FIG. 5.12 – La machine à E/S pour la tâche *"use Memo system*"*

réagit en envoyant le message *"memo is taken"* et la note disparaît du terrain. Quand l'utilisateur supprime une note (portée ou affichée), le système envoie le message *"memo is removed"*. Enfin, lorsque l'utilisateur porte une note (*"memo-Carried"*) et envoie la commande *"set"*, le système envoie le message *"memo is set"* et la note est reposée sur le terrain.

En appliquant la sémantique des opérateurs définies précédemment, on obtient la machine à E/S pour la tâche *"use Memo system*"*, illustrée dans la figure 5.12.

La machine à E/S de Memo est illustrée par la figure 5.13. Cette machine est obtenue à partir de $M_{\text{use Memo system*}}$ (figure 5.12) en ajoutant pour chaque état une transition étiquetée par l'action *"exit"* vers l'état final q_5.

5.2 Simulation de la machine à E/S

L'utilisation d'un système interactif implique l'échange continu d'entrées et de sorties avec ce système (voir le paragraphe 4.3). En d'autres mots, l'utilisation d'un système interactif peut continuer jusqu'à l'infini sauf si l'utilisateur déclenche une action qui interrompt cette utilisation comme l'action *"exit"*. Dans

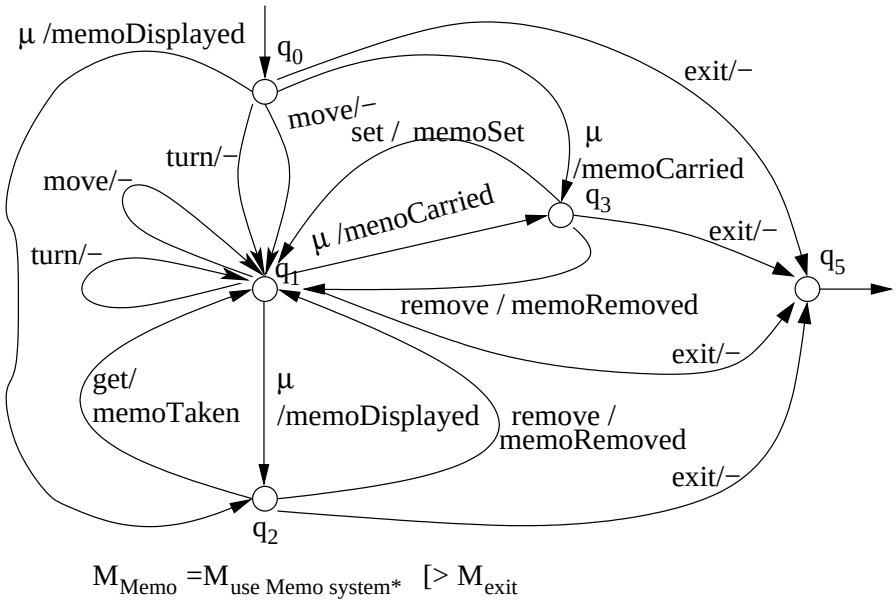

FIG. 5.13 – La machine à E/S pour l'arbre de tâches de Memo

l'exemple de Memo, on simule la machine à E/S de la tâche abstraite *"use Memo système*"* illustrée par la figure 5.12.

La simulation du comportement de l'utilisateur, nécessaire pour engendrer les séquences de tests, peut se faire sur la machine à E/S associée à l'arbre de tâches. Cette machine est semblable à celle utilisée par Lutess à l'exception de l'absence d'entrée ou de sortie sur certaines transitions. Nous proposons dans ce paragraphe une adaptation de l'algorithme de génération à l'aide de deux fonctions beh_T et $pTrans_T$ définies ci-dessous :

Définition : $beh_T : Q_T \longrightarrow 2^{I_T^\mu}$ où
$beh_T(q) = \{i \in I_T^\mu \,|\, \exists o, p,\ q \xrightarrow{i/o}_T p\}$ est l'ensemble de toutes les entrées valides de l'application à l'état q.

Définition : $pTrans_T : Q_T \times I_T^\mu \times 2^{O_T} \longrightarrow 2^{Q_T}$ où
$pTrans_T(q, i, os) = \{p \,|\ q \xrightarrow{i/o}_T p\,, o \in os\}$ est l'ensemble des états d'arrivée des transitions issues de l'état q ayant i pour entrée et dont la sortie est dans os.

La génération de données de test est illustrée par l'algorithme 1. La fonction *random*, appliquée à un ensemble fini, retourne un élément aléatoire de cet ensemble. Suivant cet algorithme, à chaque état une entrée aléatoire est choisie ($i \leftarrow random(beh_T(q))$ parmi les entrées sur les transitions issues de

Algorithm 1

```
1.  var
2.      preq, q ∈ Q_T, pFollowingq ∈ 2^{Q_T}, i ∈ (I_T ∪ {μ}),
3.      oset ∈ 2^{O_T}
4.  begin
5.      q ← qi_T
6.      while (beh_T(q) ≠ φ)
7.          oset ← ∅
8.          i ← random(beh_T(q))
9.          if (i ≠ μ) then write(i)
10.         wait(C)
11.         read(oset)
12.         preq ← q
13.         pFollowingq ← pTrans_T(preq, i, oset)
14.         if (pFollowingq ≠ ∅)
15.             then q ← random(pFollowingq)
16.             else q ← preq
17.     end while
18. end
```

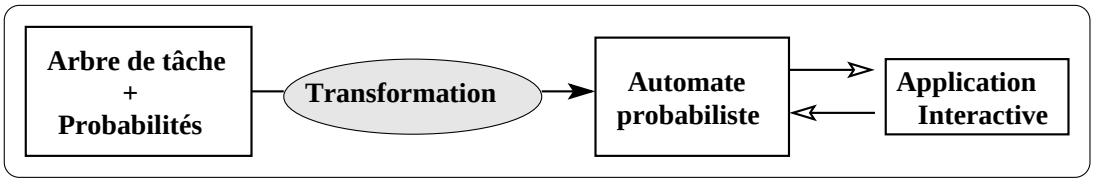

FIG. 5.14 – Test avec le profil opérationnel spécifié par l'arbre de tâche

cet état. L'entrée choisie est envoyée à l'application interactive ($write(i)$). Puis, les sorties de l'application sont lues ($read(oset)$). L'ensemble des états successeurs possibles est calculé en fonction de l'entrée envoyée et les sorties lues ($pFollowingq \leftarrow pTrans_T(preq, i, oset)$). Un état aléatoire est choisi dans cet ensemble ($q \leftarrow random(pFollowingq)$), s'il n'est pas vide, sinon, l'état précédent est maintenu ($q \leftarrow preq$), et ainsi de suite.

5.3 Utilisation de profils opérationnels

Spécifier un profil opérationnel consiste à donner des probabilités à certains opérateurs de l'arbre de tâches. Ces probabilités seront ensuite traduites en des

probabilités sur les transitions de l'automate associé. Ainsi, on obtient un automate probabiliste utilisé pour générer des tests (cf. figure 5.14).

Cette manière de procéder est proche de travaux étendant LOTOS, comme, par exemple, dans [70, 71] où des probabilités ont été ajoutées sur les opérateurs de choix, parallélisme, "hidding". Mais, contrairement à ces travaux, nous nous concentrons sur le comportement de l'environnement en essayant de prendre en compte tous les opérateurs. À cet effet, nous définissons une sémantique formelle permettant d'obtenir un modèle dont les données de test sont générées.

Dans ce paragraphe, nous présentons d'abord la manière de spécifier un profil opérationnel (cf. paragraphe 5.3.1) en proposant une extension de CTT. Le paragraphe 5.3.2 expose la transformation de cette spécification en un automate probabiliste. La simulation de cette machine afin de générer les tests pour les systèmes interactifs fait l'objet du paragraphe 5.3.3.

5.3.1 Spécification de profils opérationnels

Le profil opérationnel de l'utilisateur du système interactif est spécifié en définissant des probabilités sur certains opérateurs de l'arbre de tâches. Les règles pour spécifier ces probabilités dépendent des opérateurs.

Opérateur d'activation >>

$$T = A >> B$$

Cela n'a pas de sens d'associer une probabilité à l'opérateur d'activation. En effet, l'exécution de la tâche T implique l'exécution de la tâche A suivi de celle de la tâche B. En d'autres mots, on n'a pas de choix : la première tâche doit être exécutée afin d'activer la deuxième tâche.

Opérateur de choix []

$$T = A[]_{pr_A, pr_B} B$$

où $pr_A + pr_B = 1$

La tâche T est exécutée si seulement si une de ses sous-tâches (A, B) est exécutée. Dans ce cas, pour spécifier un profil opérationnel, nous devons spécifier des probabilités pour chacune de ces sous-tâches (A, B) en respectant la contrainte : $pr_A + pr_B = 1$.

Opérateur d'entrelacement $|||$

$$T = A|||_{pr_{ActA}, pr_{ActB}} B$$

où $pr_{ActA} + pr_{ActB} = 1$.

Pour exécuter la tâche T , il faut que toutes les sous-tâches soient exécutées, mais on n'exécute qu'une seule action de ces sous tâches à la fois. Pour cela, on peut spécifier des probabilités entre les actions de ces sous-tâches en respectant la contrainte : $pr_{ActA} + pr_{ActB} = 1$. Cela signifie qu'à chaque état d'avancement dans la tâche T, la probabilité d'exécuter une action de la tâche A est pr_{ActA} et la probabilité d'exécuter une action de la tâche B est pr_{ActB}. Cette distribution de la probabilité est vérifiée quand il y a une possibilité d'exécuter des actions de toutes les sous-tâches. Par exemple, si les actions de la tâche A ne sont plus disponibles (arrivé dans l'état $(qf_A \times q_B)$), alors la probabilité d'exécuter une action de la tâche B sera 1.

Opérateur d'itération $*$

$$T = A^*$$

On ne peut pas spécifier de probabilité pour une tâche itérative : l'exécution de la tâche T demande l'itération infinie de la tâche A.

Opérateur de désactivation $[>$

$$T = A[>_{pr_{des}} B$$

où $pr_{des} \leq 1$. Dans chaque état de la tâche A , la probabilité de l'interruption par la tâche B est pr_{des}.

Opérateur suspendre reprendre | >

$$T = A| >_{pr_{sus}} B$$

où $pr_{sus} \leq 1$, ce qui signifie que pour chaque état de la tâche A, la probabilité de suspendre la tâche A par la tâche B sera pr_{sus}.

Opérateur d'itération finie

$$T = A(n) \ , \ n > 1$$

Comme pour l'itération, on ne peut pas spécifier de probabilité pour cet opérateur.

Tâche optionnelle

$$[A]_{pr_A}$$

Cela signifie que cette tâche n'est pas obligatoire afin d'exécuter le sur-tâche. Spécifier un profil opérationnel pour une tâche A optionnelle revient à spécifier une probabilité pour que cette tâche soit exécutée, en respectant la contrainte $pr_A \leq 1$.

5.3.2 Du profil opérationnel vers un automate probabiliste

On considère un profil opérationnel défini sur l'arbre de tâche CTT. À partir de ce profil opérationnel, nous souhaitons construire un automate probabiliste.

Soit $M_T = (Q_T, qi_T, qf_T, I_T, O_T, trans_T)$ l'automate correspondant à la tâche T. On définit l'automate probabiliste correspondant à la tâche T : $MP_T = (M_T, P_T)$ où P_T est la fonction $P_T : trans_T \rightarrow [0..1]$ en imposant la contrainte :

$$\forall q \in Q_T \setminus \{qf_T\} \ : \ \sum_{c,q'} P_T(q \xrightarrow{c}_T q') = 1 \tag{5.1}$$

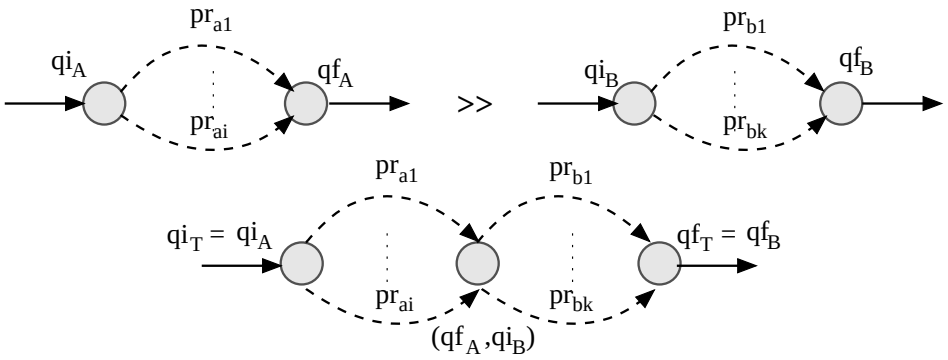

FIG. 5.15 – L'automate probabiliste résultat de l'opérateur d'activation

(la somme des probabilités des transitions partant de cet état est égale à un).

Supposons qu'on a les tâches A et B dont les automates probabilistes sont : $MP_A = (M_A, P_A)$, $MP_B = (M_B, P_B)$ respectivement, chacun de ces automates vérifiant la contrainte 5.1. L'automate résultat de leur composition avec un opérateur de (CTT+probabilité) vérifie également cette contrainte.

5.3.2.1 Opérateur d'activation >>

$$T = A >> B$$

$$MP_T(M_T, P_T) = MP_A(M_A, P_A) >> MP_B(M_B, P_B)$$

- $M_T = M_A >> M_B$
- P_T est défini ci-dessous (cf. figure 5.15).

$$P_T(q \xrightarrow{c}_T s) = \begin{cases} P_A(q \xrightarrow{c}_A s) & si \quad q, s \in Q_A^{-fin} \\ P_A(q \xrightarrow{c}_A qf_A) & si \quad q \in Q_A^{-fin} \quad et \quad s = (qf_A, qi_B) \\ P_B(qi_B \xrightarrow{c}_B s) & si \quad q = (qf_A, qi_B) \quad et \quad s \in Q_B^{-init} \\ P_B(q \xrightarrow{c}_B s) & si \quad q, s \in Q_B^{-init} \end{cases}$$

5.3.2.2 Opérateur de choix []

$$T = A[]_{pr_A, pr_B} B$$

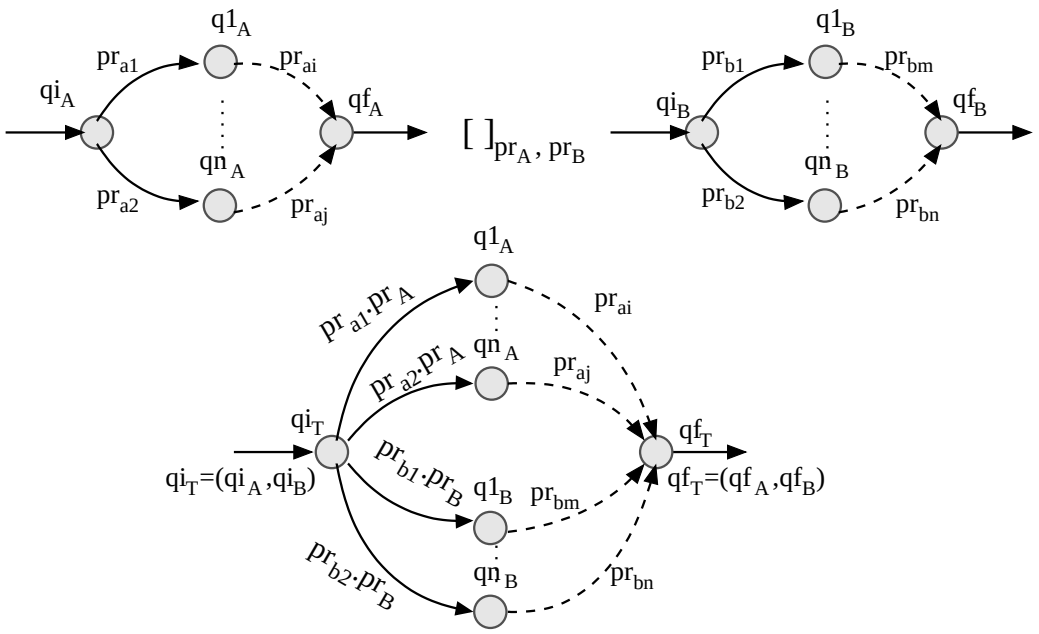

FIG. 5.16 – L'automate probabiliste résultat de l'opérateur de choix

ou $pr_A + pr_B = 1$

$$MP_T(M_T, P_T) = MP_A(M_A, P_A)[]_{pr_A, pr_B} MP_B(M_B, P_B)$$

- $M_T = M_A[]M_B$
- P_T est défini comme suit (cf. figure 5.16) :

$$P_T(q \xrightarrow{c}_T s) = \begin{cases} P_A(qi_A \xrightarrow{c}_A s) \times pr_A & si \quad q = (qi_A, qi_B) \quad et \quad s \in Q_A^{-init-fin} \\ P_A(q \xrightarrow{c}_A s) & si \quad q, s \in Q_A^{-init-fin} \\ P_A(q \xrightarrow{c}_A qf_A) & si \quad q \in Q_A^{-init-fin} \quad et \quad s = (qf_A, qf_B) \\ P_A(qi_A \xrightarrow{c}_A qf_A) \times pr_A & si \quad q = (qi_A, qi_B) \quad et \quad s = (qf_A, qf_B) \\ & et \quad qi_A \xrightarrow{c}_A qf_A \\ P_B(qi_B \xrightarrow{c}_B s) \times pr_B & si \quad q = (qi_A, qi_B) \quad et \quad s \in Q_B^{-init-fin} \\ P_B(q \xrightarrow{c}_B s) & si \quad q, s \in Q_B^{-init-fin} \\ P_B(q \xrightarrow{c}_B qf_B) & si \quad q \in Q_B^{-init-fin} \quad et \quad s = (qf_A, qf_B) \\ P_B(qi_B \xrightarrow{c}_B qf_B) \times pr_B & si \quad q = (qi_A, qi_B) \quad et \quad s = (qf_A, qf_B) \\ & et \quad qi_B \xrightarrow{c}_B qf_B \end{cases}$$

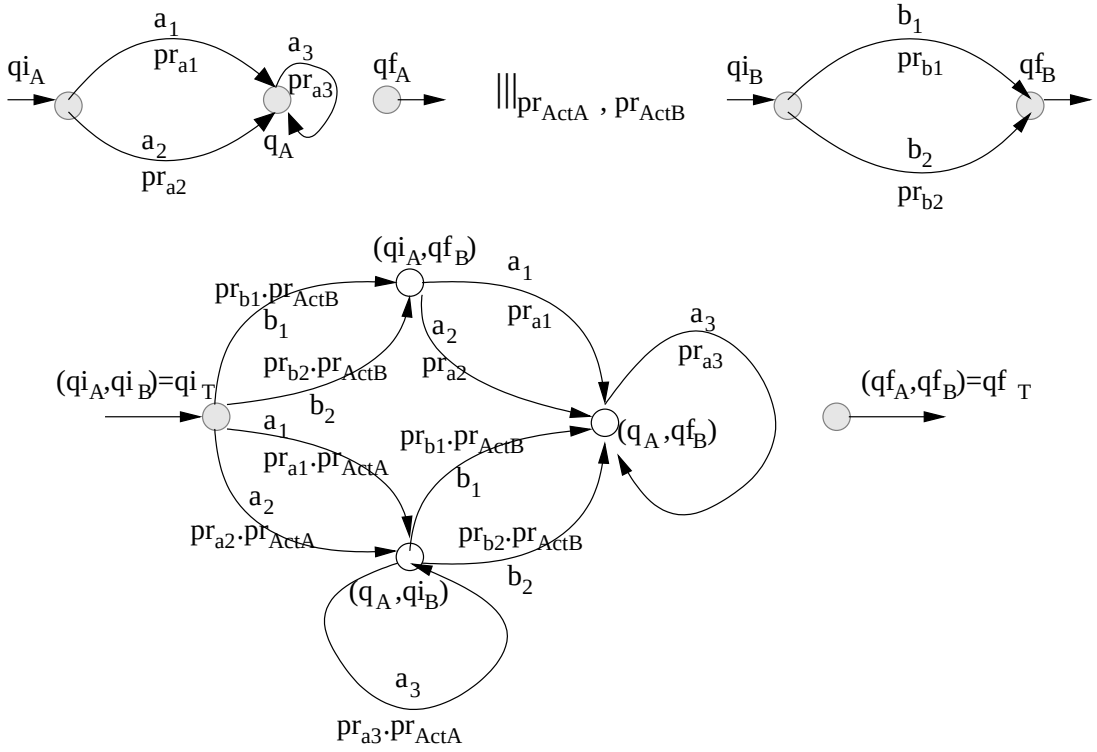

FIG. 5.17 – Un exemple d'automate probabiliste résultat de l'opérateur d'entrelacement

5.3.2.3 Opérateur d'entrelacement |||

$$T = A|||_{pr_{ActA}, pr_{ActB}} B$$

où $pr_{ActA} + pr_{ActB} = 1$

$$MP_T(M_T, P_T) = MP_A(M_A, P_A)|||_{pr_{ActA}, pr_{ActB}} MP_B(M_B, P_B)$$

- $M_T = M_A|||M_B$
- P_T est défini comme suit (cf. figure 5.17) :

$$P_T((q_A, q_B) \xrightarrow{c}_T (s_A, s_B)) = \begin{cases} pr_{ActA} \times P_A(q_A \xrightarrow{c}_A s_A) & si \quad q_B = s_B \neq qf_B \\ & et \quad q_A \xrightarrow{c}_A s_A \\ P_A(q_A \xrightarrow{c}_A s_A) & si \quad q_B = s_B = qf_B \\ & et \quad q_A \xrightarrow{c}_A s_A \\ pr_{ActB} \times P_B(q_B \xrightarrow{c}_B s_B) & si \quad q_A = s_A \neq qf_A \\ & et \quad q_B \xrightarrow{c}_B s_B \\ P_B(q_B \xrightarrow{c}_B s_B) & si \quad q_A = s_A = qf_A \\ & et \quad q_B \xrightarrow{c}_B s_B \end{cases}$$

5.3.2.4 Opérateur de désactivation [>

$$T = A[>_{pr_{des}} B$$

où $pr_{des} \leq 1$, ce qui signifie que dans chaque état de la tâche A, la probabilité d'interruption par la tâche B est pr_{des}

$$MP_T(M_T, P_T) = MP_A(M_A, P_A)[>_{pr_{des}} MP_B(M_B, P_B)$$

- $M_T = M_A[> M_B$
- Étant donné qu'on a trois ensembles de transitions ($trans_T = trans_{A'} \cup trans_{AB} \cup trans_{B'}$) (définis dans le paragraphe 5.1.2.5), on définit P_T pour chaque ensemble (cf. figure 5.18) :
  - Pour les transitions de l'ensemble $trans_{A'}$:

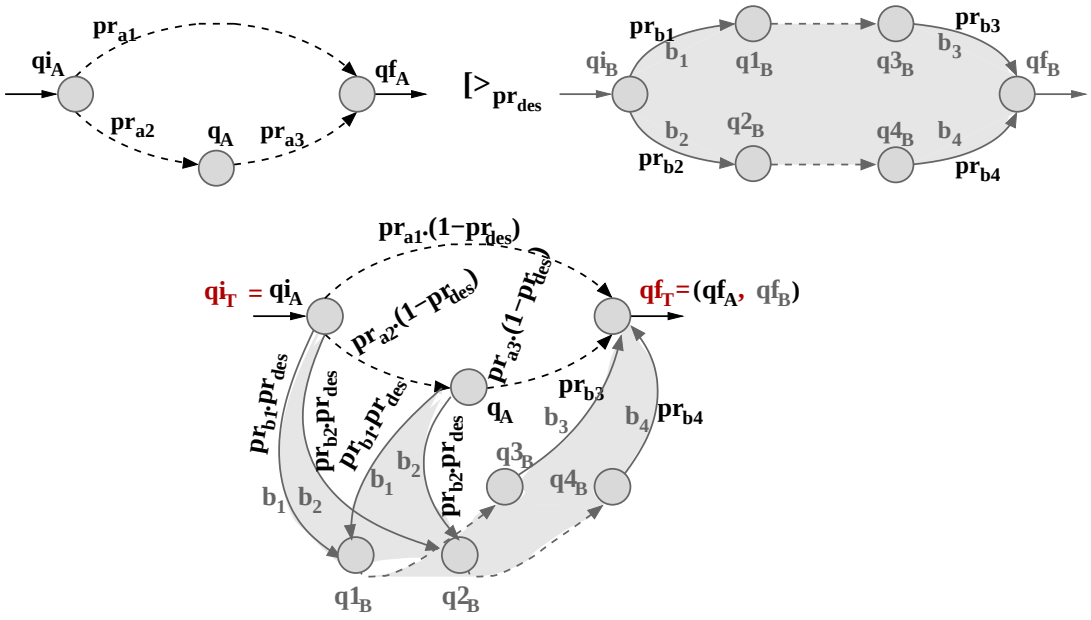

FIG. 5.18 – L'automate probabiliste résultat de l'opérateur de désactivation

$$P_T(q_A \xrightarrow{a}_{A'} q) = \begin{cases} (1-pr_{des}) \times P_A(q_A \xrightarrow{a}_A q) & si \quad q \in Q_A^{-fin} \\ (1-pr_{des}) \times P_A(q_A \xrightarrow{a}_A qf_A) & si \quad q = (qf_A, qf_B) \end{cases}$$

– Pour les transitions de l'ensemble $trans_{AB}$:

$$P_T(q_A \xrightarrow{b}_{AB} q) = \begin{cases} pr_{des} \times P_B(qi_B \xrightarrow{b}_B q) & si \quad q \in Q_B^{-init-fin} \\ pr_{des} \times P_B(qi_B \xrightarrow{b}_B qf_B) & si \quad q = (qf_A, qf_B) \end{cases}$$

– Pour les transitions de l'ensemble $trans_{B'}$:

$$P_T(q_B \xrightarrow{b}_{B'} q) = \begin{cases} P_B(q_B \xrightarrow{b}_B q) & si \quad q \in Q_B^{-init-fin} \\ P_B(q_B \xrightarrow{b}_B qf_B) & si \quad q = (qf_A, qf_B) \end{cases}$$

5.3.2.5 Opérateur suspendre reprendre$|>$

$$T = A|>_{pr_{sus}} B$$

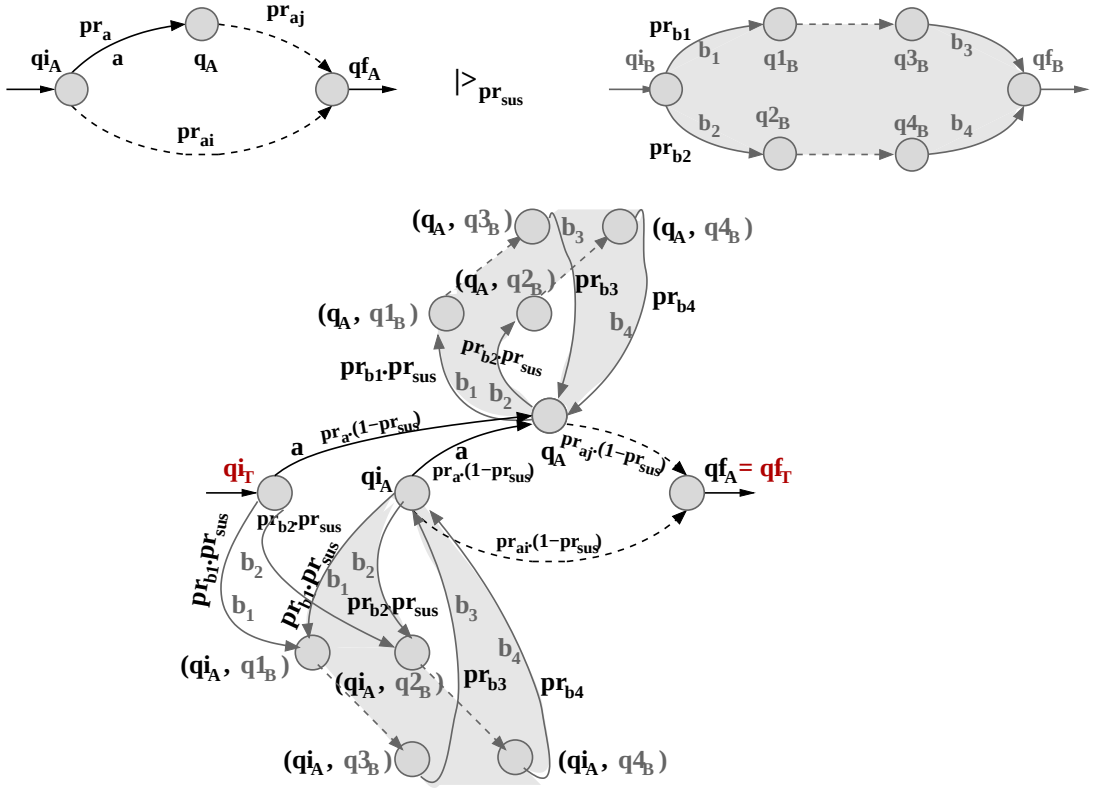

FIG. 5.19 – L'automate probabiliste résultat de l'opérateur suspendre reprendre

où $pr_{sus} \leq 1$, ce qui signifie que dans chaque état de la tâche A, la probabilité de suspendre la tâche A par la tâche B est pr_{sus}.

$$MP_T(M_T, P_T) = MP_A(M_A, P_A)| >_{pr_{sus}} MP_B(M_B, P_B)$$

- $M_T = M_A| > M_B$
- Étant donné qu'on a quatre ensembles de transitions ($trans_T = trans_A \cup trans_{AB} \cup trans_{B'} \cup trans_{T'}$) (définis dans le paragraphe 5.1.2.6), on définit P_T pour chaque ensemble (cf. figure 5.19) :
  - Pour les transitions de l'ensemble $trans_A$:

$$P_T(q_A \xrightarrow{a}_A s_A) = (1 - pr_{sus}) \times P_A(q_A \xrightarrow{a}_A s_A)$$

  - Pour les transitions de l'ensemble $trans_{AB}$:

$$P_T(q_A \xrightarrow{b}_{AB} (q_A, s_B)) = pr_{sus} \times P_B(qi_B \xrightarrow{b}_B s_B)$$

$$P_T(q_A \xrightarrow{b}_{AB} q_A) = pr_{sus} \times P_B(qi_B \xrightarrow{b}_B qf_B)$$

  - Pour les transtions de l'ensemble $trans_{B'}$:

$$P_T((q_A, q_B) \xrightarrow{b}_{B'} (q_A, s_B)) = P_B(q_B \xrightarrow{b}_B s_B)$$

$$P_T((q_A, q_B) \xrightarrow{b}_{B'} q_A) = P_B(q_B \xrightarrow{b}_B qf_B)$$

  - Pour les transitions de l'ensemble $trans_{T'}$:

$$P_T(qi_T \xrightarrow{c}_{T'} s_A) = \begin{cases} (1 - pr_{sus}) \times P_A(qi_A \xrightarrow{c}_A s_A) & si \quad s_A \neq qi_A \\ pr_{sus} \times P_B(qi_B \xrightarrow{c}_B qf_B) & si \quad s_A = qi_A \end{cases}$$

$$P_T(qi_T \xrightarrow{b}_{T'} (qi_A, s_B)) = pr_{sus} \times P_B(qi_B \xrightarrow{b}_B s_B)$$

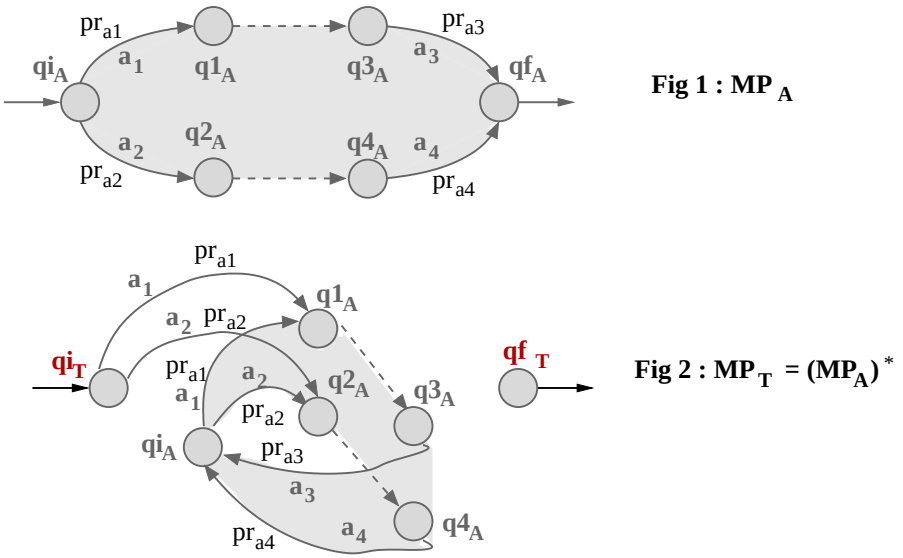

FIG. 5.20 – L'automate probabiliste résultat de l'opérateur d'itération

5.3.2.6 Opérateur d'itération *

$$T = A^*$$

$$MP_T(M_T, P_T) = MP_A(M_A, P_A)^*$$

- $M_T = (M_A)^*$
- Étant donné qu'on a deux ensembles de transitions ($trans_T = trans_{A'} \cup trans_{T'}$) définis dans le paragraphe 5.1.2.7, on va définir P_T pour chaque ensemble (cf. figure 5.20) :
  - Pour les transitions de l'ensemble $trans_{A'}$:

$$P_T(q_A \xrightarrow{a}_{A'} s_A) = \begin{cases} P_A(q_A \xrightarrow{a}_A s_A) & si \quad s_A \neq qi_A \\ P_A(q_A \xrightarrow{a}_A qf_A) & si \quad s_A = qi_A \end{cases}$$

  - Pour les transitions de l'ensemble $trans_{T'}$:

$$P_T(qi_T \xrightarrow{a}_{T'} s_A) = \begin{cases} P_A(qi_A \xrightarrow{a}_A s_A) & si \quad s_A \neq qi_A \\ P_A(qi_A \xrightarrow{a}_A qf_A) & si \quad s_A = qi_A \end{cases}$$

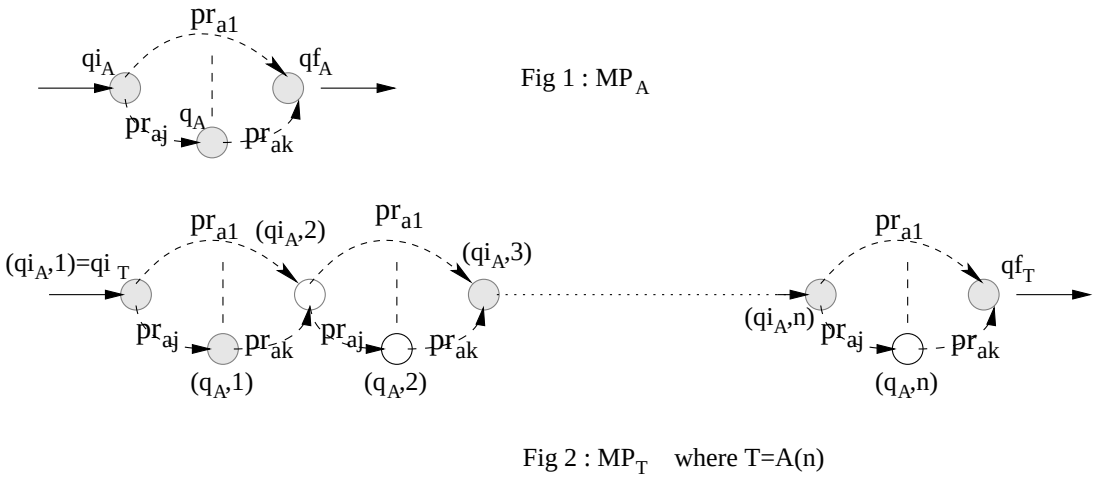

FIG. 5.21 – L'automate probabiliste résultat de l'opérateur d'itération finie

5.3.2.7 Opérateur d'itération finie

$$T = A(n)\ ,\ n > 1$$

L'exécution de la tâche T demande l'itération de n fois de la tâche A

$$MP_T(M_T, P_T) = MP_A(M_A, P_A)(n)$$

- $M_T = (M_A)(n)$
- P_T est défini comme suit (cf. figure 5.21) :

$$P_T((q_A, i) \xrightarrow{a}_T (s_A, j)) = \begin{cases} P_A(q_A \xrightarrow{a}_A s_A) & si \quad i = j \\ P_A(q_A \xrightarrow{a}_A qf_A) & si \quad j = i + 1 \, et \, s_A = qi_A \end{cases}$$

$$P_T((q_A, n) \xrightarrow{a}_T qf_T) = P_A(q_A \xrightarrow{a}_A qf_A)$$

5.3.2.8 Tâche optionnelle

$T = [A]_{pr_A}$ ou $pr_A \leq 1$

Dans le cas d'utilisation avec l'opérateur d'activation :

$$[A]_{pr_A} >> B = (A >> B)[]_{pr_A,(1-pr_A)}B$$

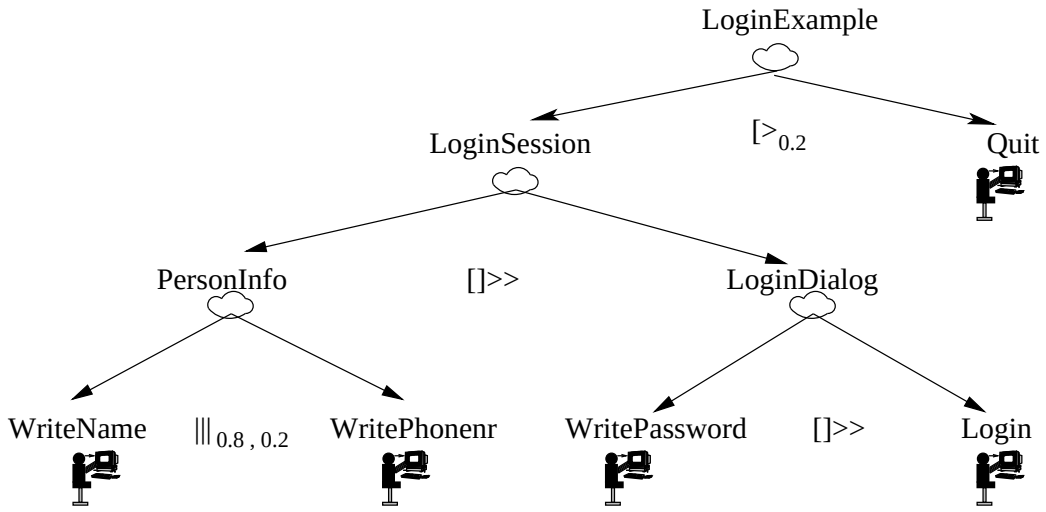

FIG. 5.22 – Un exemple de spécification de profil opérationnel sur l'arbre de tâche

Dans le cas d'utilisation avec l'opérateur d'entrelacement :

$$[A]_{pr_A}|||_{pr_{Act[A]},pr_{ActB}}B = (A|||_{pr_{Act[A]},pr_{ActB}}B)[]_{pr_A,(1-pr_A)}B$$

5.3.2.9 Exemple

La figure 5.22 illustre une spécification de profil opérationnel sur l'arbre de tâches et la figure 5.23 montre l'automate probabiliste associé.

Dans cet arbre, la tâche "LoginExample" est composée des tâches "LoginSession" et "Quit". La tâche "Quit" peut interrompre la tâche "LoginSession" avec la probabilité $''0.2''$. Nous allons étudier dans ce cas la probabilité de la tâche "LoginSession", pr_{LS}, et de la tâche "Quit", pr_Q. La somme de pr_{LS} et pr_Q doit être *un*.

Avant de calculer ces probabilités, on définit un chemin comme une suite de transitions dont la probabilité d'exécution est le produit des probabilités des transitions qui le composent.

Chaque tâche sur l'arbre de tâche peut être exécutée par un chemin parmi un ensemble de chemins possibles. La probabilité d'une tâche est la somme des probabilités de tous les chemins possibles qui conduisent à son exécution.

Supposons que les transitions sont dénotées comme suit :

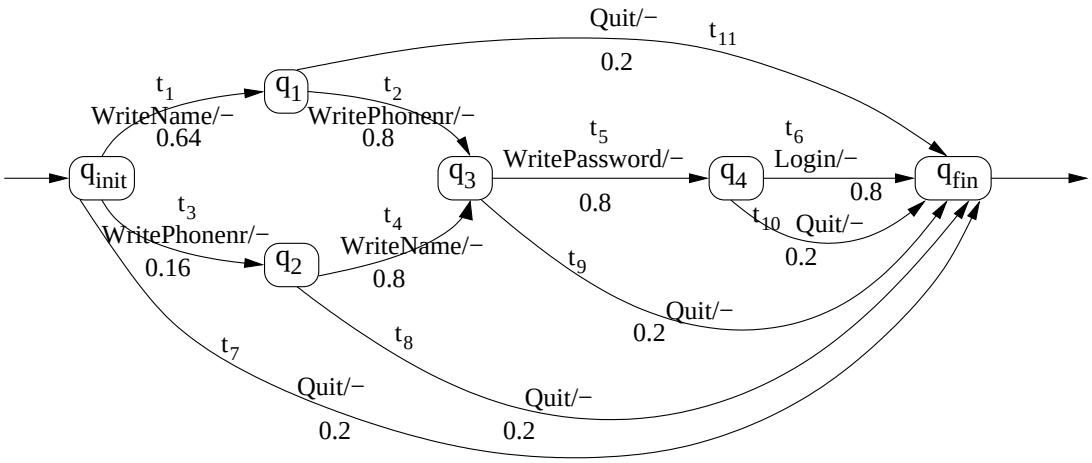

FIG. 5.23 – Un exemple d'un automate probabiliste

$$t_1 = q_{init} \overset{writeName/-}{\longrightarrow} q_1 \quad t_2 = q_1 \overset{writePhonenr/-}{\longrightarrow} q_3 \quad t_3 = q_{init} \overset{writePhonenr/-}{\longrightarrow} q_2$$
$$t_4 = q_2 \overset{writeName/-}{\longrightarrow} q_3 \quad t_5 = q_3 \overset{writePassword/-}{\longrightarrow} q_4 \quad t_6 = q_4 \overset{Login/-}{\longrightarrow} q_{fin}$$
$$t_7 = q_{init} \overset{Quit/-}{\longrightarrow} q_{fin} \quad t_8 = q_2 \overset{Quit/-}{\longrightarrow} q_{fin} \quad t_9 = q_3 \overset{Quit/-}{\longrightarrow} q_{fin},$$
$$t_{10} = q_4 \overset{Quit/-}{\longrightarrow} q_{fin} \quad t_{11} = q_1 \overset{Quit/-}{\longrightarrow} q_{fin}$$

Les probabilités pr_{LS} et pr_Q sont calculées comme suit :

$$\begin{aligned} pr_{LS} &= (P(t_1) * P(t_2) * P(t_5) * P(t_6)) \\ &\quad +(P(t_3) * P(t_4) * P(t_5) * P(t_6)) = 0.4096 \end{aligned}$$

$$\begin{aligned} pr_Q &= (P(t_7)) \\ &\quad +((P(t_3) * P(t_8)) \\ &\quad +((P(t_1) * P(t_{11})) \\ &\quad +(P(t_1) * P(t_2) * P(t_9)) + (P(t_3) * P(t_4) * P(t_9)) \\ &\quad +(P(t_1) * P(t_2) * P(t_5) * P(t_{10})) + (P(t_3) * P(t_4) * P(t_5) * P(t_{10})) \\ &\quad = 0.5904 \end{aligned}$$

Nous constatons que malgré la probabilité faible de la désactivation, la probabilité de la tâche désactivant *"Quit"* est importante, car la désactivation peut apparaître dans tout état de la tâche *"LoginSesion"*.

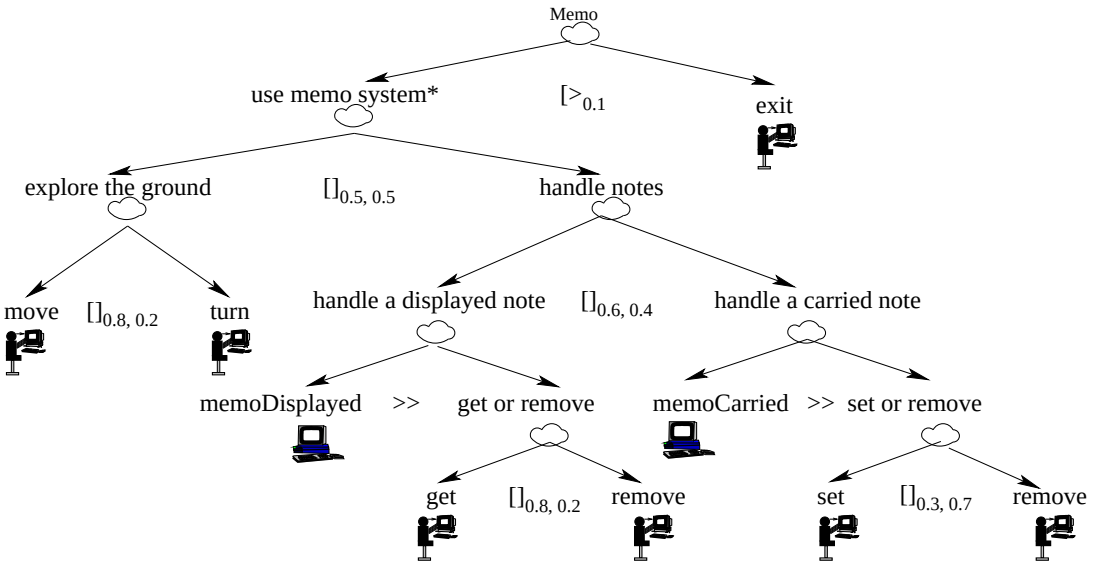

FIG. 5.24 – L'arbre de tâches CTT étendue avec des probabilité pour Memo

5.3.2.10 Exemple

La figure 5.24 montre l'arbre de tâches CTT étendu pour Memo. Le profil opérationnel spécifié par cet arbre de tâches est le suivant :

- pour chaque état pendant l'utilisation de Memo, la probabilité de sortir de l'application est de 0.1 ;
- il n'y a pas de préférence entre les deux tâches découvrir le terrain et manipuler les notes ;
- l'utilisateur préfère bouger (probabilité 0.8) que tourner (probabilité 0.2) pour découvrir le terrain ;
- l'utilisateur préfère manipuler les notes affichées (probabilité 0.6) que les notes portées (probabilité 0.4) ;
- si le système affiche une note, l'utilisateur préfère la récupérer (probabilité 0.8) que la supprimer (probabilité 0.2) ;
- si l'utilisateur porte une note, il préfère la supprimer (probabilité 0.7) que la poser (probabilité 0.3).

À partir des automates probabilistes des tâches élémentaires du système Memo qui sont illustrés dans la figure 5.25 et en appliquant les règles de composition définis précédemment sur le sous-arbre de tâches qui représente la tâche *"use memo system*"*, on obtient l'automate probabiliste de cette tâche, illustré

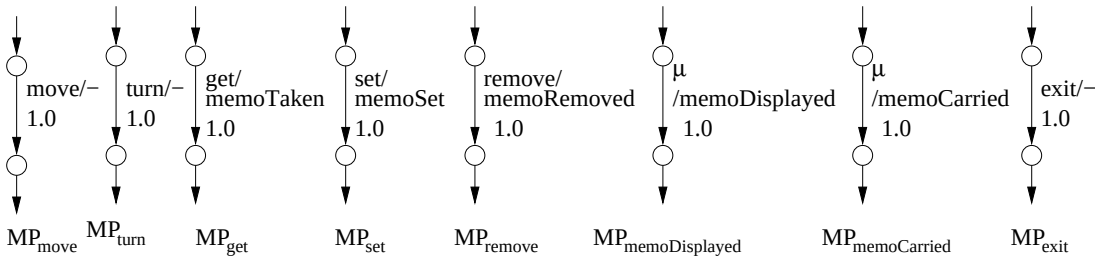

FIG. 5.25 – Les automates probabilistes pour les tâches : *"move", "turn", "get", "set", "remove", "memoDisplayed", "memoCarried"* et *"exit"*

dans la figure 5.26.

Ainsi, en appliquant la règle définie dans le paragraphe 5.3.2.4 sur la tâche *"Memo = use memo system* $[>_{0.1}$ exit"*, l'automate probabiliste de cette tâche de la figure 5.27 est obtenu.

Dans cette figure, à partir de chaque état de l'automate probabiliste de la figure 5.26, il y a une transition étiquetée par l'action *"exit/-"* et avec la probabilité *"0.1"* vers l'état final q_5 qui est atteignable uniquement par cette action. Les probabilités des autres transitions qui sont à l'origine de la machine probabiliste de la tâche *"use memo system*"* sont multipliées par la valeur *(0.9 = 1.0 - 0.1)*.

5.3.3 Simulation de la machine probabiliste à E/S

Afin de générer les données de test en respectant le profil opérationnel défini par un arbre de tâches CTT étendu par des probabilités, on simule la machine probabiliste correspondante. Pour simuler une machine probabiliste $MP_T = (M_T, P_T)$, on utilise les deux fonctions beh_T (retourne l'ensemble de toutes les entrées valides de l'application pour un état donné) et $pTrans_T$ (retourne l'ensemble de toutes les transitions possibles pour un état et un ensemble de sorties donnés) qui sont définies dans le paragraphe 5.2. On utilise également les fonctions définies ci-dessous.

Définition : Une distribution de probabilités sur un ensemble A d'éléments, dénotée par $DistProb_A$ est un ensemble de couples $< el, pr >$ tels que $el \in A$, *pr est un réel* $\in [0..1]$, avec la contrainte :

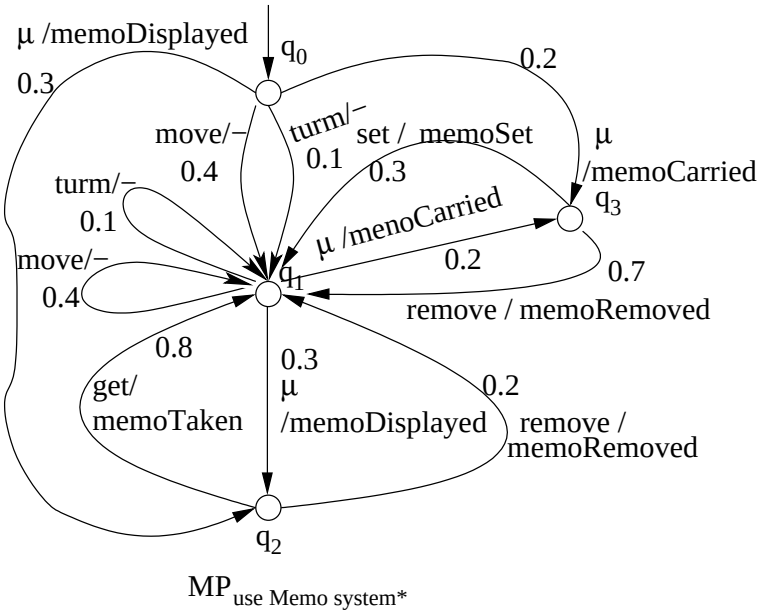

FIG. 5.26 – L'automate probabiliste de la tâche *"use memo system*"*

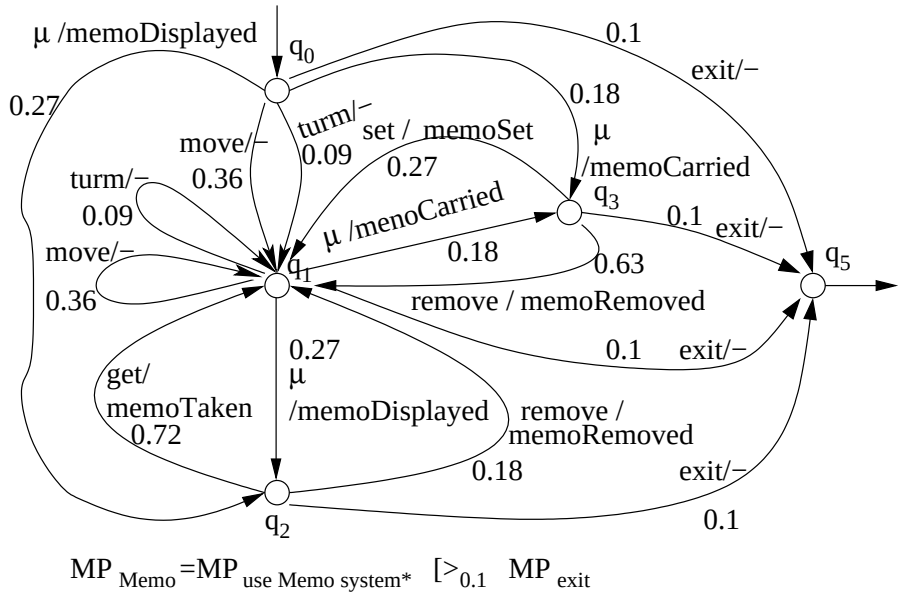

FIG. 5.27 – L'automate probabiliste de la tâche *"Memo = use memo system* $[>_{0.1}$ exit"*

$$\sum_{<el,pr>\in DistProb_A} pr = 1 \tag{5.2}$$

Si A est vide, alors $DistProb_A$ est également vide.

Définition : Soit E un ensemble d'éléments. $\mathcal{P}(E)$ est l'ensemble de tous les sous-ensembles de E. On appelle $ENSDistProb_{\mathcal{P}(E)}$ l'ensemble de toutes les distributions possibles de probabilités sur tous les sous-ensembles de E. En d'autres termes, un élément de $ENSDistProb_{\mathcal{P}(E)}$ est une distribution de probabilité $DistProb_A$ sur un ensemble $A \in \mathcal{P}(E)$.

Définition : $DistProbEn : Q_T \longrightarrow ENSDistProb_{\mathcal{P}(I_T^\mu)}$ où :

$DistProbEn(q) = DistProb_{beh_T(q)}$

$= \{< i, pr > | i \in beh_T(q), pr = \sum_{o,r} P_T(q \xrightarrow{i/o} r)\}$ est la distribution de probabilités des entrées valides dans l'état q.

Définition : $DistProbTrans : Q_T \times I_T^\mu \times \mathcal{P}(O_T) \longrightarrow ENSDistProb_{\mathcal{P}(Q_T)}$ où :

$DistProbTrans(q, i, oset) = DistProb_{pTrans_T(q,i,oset)} = \{< q', pr > | q' \in pTrans_T(q, i, oset), pr = \frac{\sum_{o \in oset} P_T(q \xrightarrow{i/o} q')}{\sum_{r \in pTrans_T(q,i,oset), o \in oset} P_T(q \xrightarrow{i/o} r)}\}$ est la distribution de probabilités des transitions possibles pour l'état q pour l'entrée i et l'ensemble de sorties $oset$.

Définition : $tirage : ENSDistProb_{\mathcal{P}(E)} \longrightarrow E$ où :

$tirage(DistProb_A)$ retourne un élément de A en respectant la distribution de probabilités $DistProb_A$

La génération de données de test est illustrée par l'algorithme 2. Suivant cet algorithme, dans chaque état, une entrée est choisie en respectant les probabilités des transitions issues de cet état ($i \leftarrow tirage(DistProbEn(q))$). L'entrée choisie est envoyée à l'application interactive. Puis, les sorties de l'application sont lues ($read(oset)$). L'ensemble des états successeurs possibles est calculé en fonction de l'entrée envoyée et les sorties lues ($pFollowingq \leftarrow pTrans_T(preq, i, oset)$). De cet ensemble, un état est choisi en respectant les probabilités des transitions associées ($q \leftarrow tirage(DistProbTrans(preq, i, oset))$), et ainsi de suite.

Algorithm 2

```
var
    preq, q ∈ Q_T, pFollowingq ∈ 2^{Q_T}, i ∈ (I_T ∪ {μ}),
    oset ∈ 2^{O_T}
begin
    q ← qi_T
    while ( beh_T(q) ≠ φ )
            oset ← ∅
            i ← tirage(DistProbEn(q))
            if (i ≠ μ) then write(i)
            wait(C)
            read(oset)
            preq ← q
            pFollowingq ← pTrans_T(preq, i, oset)
            if (pFollowingq ≠ ∅)
                  then q ← tirage(DistProbTrans(preq, i, oset))
                  else q ← preq
    end while
end
```

5.4 Conclusion

Aujourd'hui, les modèles sont utilisés de plus en plus dans le processus du développement de systèmes.

La génération de tests à partir de l'arbre de tâches permet aux concepteurs de systèmes interactifs de valider leurs systèmes à partir d'un modèle habituel dans le processus du développement. En ajoutant des probabilités aux tâches du modèle, on peut générer automatiquement des tests avec différents profils de l'utilisateur.

Une extension intéressante de ce travail serait l'utilisation du guidage par des propriétés de sûreté. Dans ce cas, le générateur favorisera la génération de données qui conduisent dans un état où la propriété de sûreté peut être violée.

Deuxième partie

Validation de systèmes interactifs multimodaux

Chapitre 6

Systèmes multimodaux

Aujourd'hui, les applications interactives s'orientent vers de multiples techniques d'interaction, autres que les touches de clavier et les boutons de la souris, telles que la voix ou bien le geste, en particulier dans le cadre des applications mobiles de réalité virtuelle.

Ces techniques d'interaction peuvent être utilisées de manière indépendante ou synergique. Ainsi, de nouveaux aspects de l'interaction doivent être considérés, comme la fusion d'information et la nature de différentes contraintes dans le cas de l'utilisation synergique de plusieurs modalités. Ces aspects doivent être considérés pour la conception, la modélisation et la validation des applications interactives.

Dans cette deuxième partie de notre travail, on s'intéresse aux particularités de la validation des applications multimodales et nous étudions les extensions des techniques présentées précédemment nécessaires à la prise en compte des caractéristiques de ces applications.

Certaines approches formelles ont été proposées pour modéliser et valider des applications multimodales. Nous en exposons certaines (paragraphe 6.2) après avoir présenté les concepts les plus importants de l'interaction multimodale (paragraphe 6.1).

6.1 Interaction multimodale

6.1.1 Modalité

D'après [72], une technique d'interaction est définie soit comme un dispositif physique d'entrée/sortie, soit comme un langage d'interaction, soit comme le couplage d'un dispositif et d'un langage d'interaction. Le choix entre ces trois vues dépend du contexte et des besoins de l'analyse. Pour nos travaux, nous reprenons la définition suivante de la technique d'interaction (le concept de modalité est assimilé à cette définition) [23] : "Une technique d'interaction ou une modalité est un couple formé d'un dispositif physique d et d'un langage d'interaction L : <d,L>". Un dispositif physique est un objet du système qui acquiert (dispositif d'entrée) ou distribue (dispositif de sortie) de l'information. Des exemples de dispositifs sont le clavier, la souris, le microphone ou l'écran. Le langage d'interaction se définit par un ensemble d'expressions bien définies (i.e., un ensemble conventionnel de symboles) qui porte des sens. La génération d'un symbole, ou d'un ensemble de symboles, provient d'actions sur les dispositifs physiques. Dans l'application MATIS [23] (Multimodal Airline Travel Information System), qui est un système d'information multimodal pour les transports aériens, le langage pseudo-naturel et la manipulation directe sont des exemples de langages d'interaction.

Cette définition s'applique aux modalités d'interaction en entrée utilisées par l'utilisateur pour interagir avec le système ainsi que les modalités d'interaction en sortie, qui véhiculent des informations du système vers l'utilisateur.

Des exemples de modalités dans MATIS incluent :

- la modalité d'entrée "Parole" se définit par le couple <microphone, langage pseudo-naturel NL>, où NL est défini par une grammaire spécifique ;
- la modalité d'entrée "langage naturel écrit" se définit par le couple <clavier, langage pseudo-naturel NL> ;
- la modalité "entrée graphique" se définit par <souris, manipulation directe> et
- la modalité "sortie graphique" correspond au couple <écran, tables> (les horaires de vol retournées par MATIS sont toujours présentées par une table).

Nous distinguons deux classes de modalités : les modalités actives et les modalités passives.

Une modalité d'entrée est active quand l'utilisateur doit réaliser une action explicite avec un dispositif en vue de spécifier une commande au système comme la parole <microphone, langage pseudo-naturel> ou les interfaces graphiques manipulables <souris, la manipulation directe>.

Une modalité est passive quand le dispositif associé ne requiert pas l'attention ni d'action explicite de l'utilisateur comme la capture par un GPS de la localisation d'un utilisateur, la modalité passive correspondante étant décrite par le couple <GPS, localisation en données GPS>. De nombreuses interfaces (p.ex. interfaces sensibles au contexte "perceptual user interfaces") exploitent des modalités d'entrée passives afin de rendre l'interaction plus robuste et efficace. De même, de nombreux systèmes de réalité augmentée sur supports mobiles reposent sur des modalités de sortie passives exploitant un casque semi-transparent.

6.1.2 Multimodalité

Un système interactif multimodal [23] dispose d'au moins deux modalités pour un sens donné (entrée ou sortie) comme la parole et une interface manipulable par la souris. Plusieurs modalités peuvent être indépendantes ou combinées [24], si une fusion entre les différents types de données associées à ces modalités a lieu.

L'utilisation combinée de modalités est restreinte par des contraintes temporelles que nous présentons dans le paragraphe suivant.

6.1.2.1 Aspects temporels de l'usage combiné de modalités

Plusieurs modalités peuvent être utilisées de manière séquentielle ou parallèle [26, 73] dans une fenêtre temporelle (intervalle de temps) TW.

Les modalités d'un ensemble M sont utilisées de manière **parallèle**, si elles sont utilisées au même instant. Les modalités d'un ensemble M sont utilisées **séquentiellement** dans une fenêtre temporelle TW, si il y a au plus une modalité active à chaque instant et si toutes ces modalités sont utilisées dans TW.

La fenêtre temporelle, le parallélisme et le séquencement expriment une contrainte sur l'espace de l'interaction. L'absence de contrainte temporelle revient à considérer que la durée de la fenêtre temporelle est infinie.

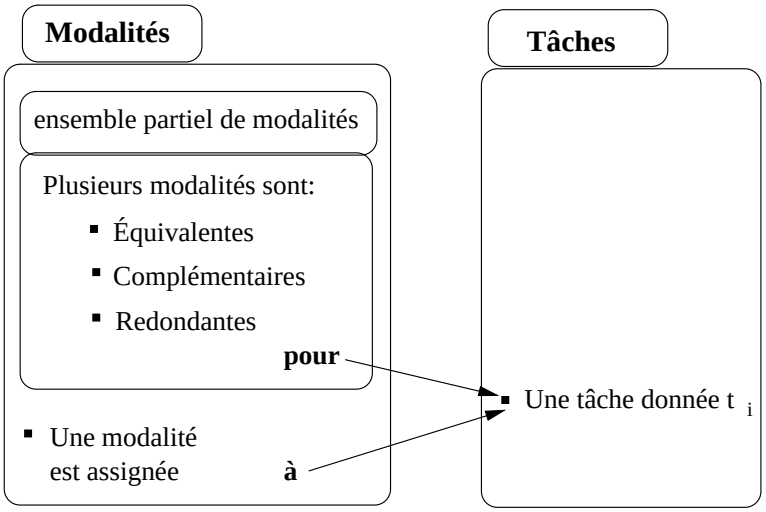

FIG. 6.1 – Les propriétés CARE

6.1.2.2 Propriétés CARE

Dans une application interactive multimodale, les propriétés CARE [25, 26, 27] définissent quatre manières d'utiliser des modalités : la Complémentarité, l'Assignation, la Redondance et l'Équivalence (voir la figure 6.1).

Assignation Une modalité M est dite assignée à un ensemble partiel T de tâches, si chaque tâche $t \in T$ ne peut être provoquée que par la modalité M.

Équivalence Les modalités d'un ensemble M sont équivalentes pour la tâche t, si t peut être provoquée par l'une quelconque des modalités de M.

Complémentarité Les modalités d'un ensemble M sont complémentaires pour la tâche t, si, pour exécuter t, toutes ces modalités doivent être utilisées simultanément ou de manière séquentielle restreinte par une fenêtre temporelle.

Redondance Les modalités d'un ensemble M sont redondantes pour la tâche t, si, pour exécuter t, la même information doit être émise par chacune de ces modalités simultanément ou de manière séquentielle restreinte par une fenêtre temporelle.

6.2 Vérification/validation de systèmes multimodaux

6.2.1 L'approche ICO

L'approche ICO [7, 8, 9, 10, 11], présentée dans le paragraphe (2.2.3), a été étendue pour prendre également en compte les systèmes multimodaux [11, 74].

Nous rappelons que cette approche utilise un formalisme appelé «Objets Coopératifs Interactifs» ou ICO («Interactive Cooperative Objects»), fondé sur les concepts de l'approche orientée-objet et les réseaux de Petri de haut niveau.

Pour spécifier les systèmes multimodaux avec les ICO [11] les extensions suivantes ont été proposées :

- Ajout du temps : un ensemble d'extensions temporelles a été ajouté à la description formelle d'ICO [8] afin de pouvoir manipuler des contraintes temporelles (le temps a été ajouté aux transitions, sur les places et sur les jetons).
- Un mécanisme de communication par production et consommation d'événements.
- Un mécanisme de structuration sur l'utilisation de «transducers», de manière à pouvoir traiter des événements de plus ou moins haut niveau.
- Un mécanisme plus général de rendu de manière à pouvoir modéliser le comportement des médias de sortie en fonction de l'état des différents réseaux qui modélisent l'application.

Afin de modéliser une application multimodale avec les ICO, cette dernière est décomposée en différents niveaux logiques, qui communiquent entre eux de façon asynchrone par diffusion. Par exemple, dans [11] chaque dispositif d'entrée est modélisé avec un ICO qui prend en entrée les actions sur un média et produit des événements logiques utilisables par le contrôleur de dialogue. Ce dernier réagit en fonction de son état et reproduit les événements de rendu qui sont capturées par des ICO de rendu logique et/ou physique, représentant les médias de sortie.

6.2.2 Algèbre de processus pour la multimodalité

Une méthode de modélisation et de vérification de systèmes interactifs multimodaux est proposée dans [75, 76, 77, 78]. Cette approche se concentre sur les modalités d'entrée et leur fusion et consiste à définir un modèle formel pour le système interactif multimodal et les propriétés.

L'expression du système est inspirée de l'algèbre de processus dont la sémantique est donnée par des systèmes de transitions étiquetées, tandis que les propriétés sont exprimées soit par des systèmes de transitions décrivant les comportements souhaités soit par des expressions d'une logique comme CTL (Computational Tree Logic).

Description du système La syntaxe du langage décrivant l'interaction multimodale en entrée est donnée par des grammaires issues de l'algèbre de processus classique. La règle S est utilisée pour générer le modèle en fonction des tâches utilisateurs à un niveau élevé d'abstraction jusqu'au niveau des tâches élémentaires qui sont les énoncés. Les énoncés sont les tâches qui déclenchent une fonction élémentaire du noyau fonctionnel de l'application. La règle E génère les énoncés en composant des actions de l'ensemble $A = \bigcup_{i=1}^{n} A_{m_i}$ où A_{m_i} est l'ensemble des événements produits par la modalité m_i. Un élément de A est dénoté par e. A_{Set} dénote un sous-ensemble quelconque de A .

Les opérateurs suivants de composition sont définis : [], >>, |||, ||, et ; pour le choix, la séquence, l'entrelacement, le parallélisme et le préfixage. De plus, δ est un terme qui ne fait rien. La sémantique opérationnelle de ces opérateurs est donnée dans le tableau 6.1.

Description formelle du système multimodal La modélisation d'interfaces multimodales proposée en [75, 76, 77, 78] repose sur les trois critères suivants [24, 23, 79] :

- production des énoncés : séquentielle ou parallèle ;
- usage des modalités : exclusif ou simultané ;
- nombre de modalités par énoncé.

La combinaison de ces critères produit plusieurs types de multimodalité. Nous présentons ici à titre d'exemple trois types :

δ	$\delta \nrightarrow$	
;	$e; P \xrightarrow{e} P$	
[]	$\frac{P \xrightarrow{e} P'}{P[]Q \xrightarrow{e} P'}$	$\frac{Q \xrightarrow{e} Q'}{P[]Q \xrightarrow{e} Q'}$
>>	$\frac{P \xrightarrow{e} P' \, et \, P' \neq \delta}{P>>Q \xrightarrow{e} P'>>Q}$	$\frac{P \xrightarrow{e} P' \, et \, P' = \delta}{P>>Q \xrightarrow{e} Q}$
|||	$\frac{P \xrightarrow{e} P'}{P\|\|\|Q \xrightarrow{e} P'\|\|\|Q}$	$\frac{Q \xrightarrow{e} Q'}{P\|\|\|Q \xrightarrow{e} P\|\|\|Q'}$
||	$\frac{P \xrightarrow{e} P'}{P\|\|Q \xrightarrow{e} P'\|\|Q}$ $\frac{P \xrightarrow{e_1} P', Q \xrightarrow{e_2} Q' \, avec \, modalit(e_1) \neq modalit(e_2)}{P\|\|Q \xrightarrow{(e_1,e_2)} P'\|\|Q'}$	$\frac{Q \xrightarrow{e} Q'}{P\|\|Q \xrightarrow{e} P\|\|Q'}$

TAB. 6.1 – La sémantique d'opérateurs de la composition au moyen de règles de la forme $\frac{prémisses}{conclusion}$

1. Alternée : plusieurs modalités peuvent être utilisées alternativement pour produire un énoncé. Les énoncés sont produits de manière séquentielle :
   $S ::= S[]S \mid S >> S \mid E$ choix ou séquence entre les énoncés
   $E ::= e; E \mid \delta \, avec \, e \in A_{Set}$ événements issus de plusieurs modalités en séquence
2. Synergique : les énoncés sont produits de manière séquentielle mais plusieurs modalités peuvent être utilisées dans un même énoncé et de manière parallèle.
   $S ::= S[]S \mid S >> S \mid E$ choix ou séquence entre les énoncés
   $E ::= e; E \mid e|||E \mid e||E \mid \delta \, avec \, e \in A_{Set}$ événements issus de plusieurs modalités en séquence ou en parallèle
3. Parallèle exclusive : plusieurs énoncés indépendants peuvent être produits en parallèle. Une seule modalité est utilisée pour chaque énoncé et à un instant donné une seule modalité est active.
   $S ::= S[]S \mid S >> S \mid S|||S \mid E$ choix, séquence ou entrelacement entre les énoncés
   $E ::= e; E \mid \delta \, avec \, e \in A_{m_i}$ événements issus d'une seule modalité

6.2.3 L'approche B

Dans le paragraphe 2.2.4, nous avons vu que la méthode B a été utilisée pour la validation de systèmes interactifs. Cette méthode a également été utilisée pour la vérification et la spécification de systèmes interactifs multimodaux [80, 81].

La validation de systèmes multimodaux en B événementiel suit une démarche de conception descendante fondée sur le raffinement. Le modèle abstrait racine déclare des événements de haut niveau. Puis, il est raffiné par l'introduction de nouveaux événements qui précisent le modèle abstrait initial.

Quatre scénarios de conception d'IHM multimodales fondée sur la preuve sont identifiés [80]. Chacun décrit une implantation de la composition des composants du noyau fonctionnel et de la présentation pour définir le Contrôleur de dialogue :

- description des raffinements B du noyau fonctionnel puis introduction des événements de la présentation par raffinement ;
- description des raffinements B de la présentation puis introduction des événements du noyau fonctionnel par raffinement ;
- composition des événements du noyau fonctionnel et de la présentation dans le même modèle B ;
- description des raffinements B de la présentation avec un noyau fonctionnel abstrait.

Dans [81] le scénario 4 a été retenu, ce qui permet une vérification formelle modulaire.

Quatre modèles ont été définis, chacun raffinant le précédent. Le premier introduit les événements de haut niveau. Le deuxième présente les événements de la présentation ainsi que leur synchronisation avec le Contrôleur de Dialogue. Le troisième introduit les interactions multi-modales avec les différentes possibilités d'interaction. Enfin, le quatrième introduit concrètement les modalités et décompose les interactions multi-modales en événements d'interaction atomiques de base. Ces modèles font tous abstraction du noyau fonctionnel et ne font que rendre compte du fait que des événements abstraits du noyau fonctionnel ont été déclenchés.

La vérification de propriétés CARE est effectuée dans le quatrième modèle où les événements d'interaction atomiques en provenance de différentes modalités sont introduits. Chaque propriété CARE est décrite par une tâche CTT. Les opérateurs CTT sont codés dans le modèle B événementiel [17], ce qui permet la validation de ces propriétés par construction.

Les propriétés CARE sont décrites avec les opérateurs CTT comme suit [81] :

Étant données deux modalités différentes, x et y étant deux événements appartenant respectivement à ces modalités, une équivalence entre ces modalités

implique l'implantation de la tâche $choix\_x\_y = x[]y$; la redondance implique l'implantation de la tâche $redondance\_x\_y = x||y$ et la complémentarité implique l'implantation de la tâche $complement\_x\_y = x||y$.

Chapitre 7

Test synchrone de systèmes interactifs multimodaux

7.1 Introduction

Dans le chapitre précédent nous avons vu que les systèmes interactifs multimodaux disposent de caractéristiques propres telles que les propriétés CARE (cf. paragraphe 6.1.2.2). Dans ce chapitre, nous étudions les extensions de l'approche de validation basée sur l'utilisation de Lutess proposée dans le chapitre 4 nécessaires à la prise en compte de la multimodalité [68, 69] en entrée.

Les systèmes multimodaux peuvent recevoir des événements de différents niveaux d'abstraction ce qui a un impact sur la génération de tests. Le paragraphe 7.2 traite de cet aspect et précise le choix du niveau d'abstraction que nous avons effectué.

La multimodalité implique que plusieurs événements peuvent survenir simultanément. Nous montrons comment les systèmes multimodaux interagissent avec leur environnement dans le paragraphe 7.3, puis comment l'approche synchrone peut être adaptée à l'interaction avec les systèmes multimodaux (cf. paragraphe 7.4).

La validation de ces systèmes avec Lutess est exposée dans le paragraphe 7.5. Nous montrons en particulier, que les propriétés CARE peuvent être exprimées en Lustre (cf. paragraphe 7.5.1.1). La validation de ces propriétés nécessite la génération d'événements de modalités différentes proches temporellement (cf.

paragraphe 6.1.2.2). La technique à utiliser pour générer ces événements dans la même fenêtre temporelle afin de valider ces propriétés est présentée dans le paragraphe 7.5.2.

Reprenons l'exemple de l'application Memo [65], présenté dans le chapitre 4. On rappelle que Memo est un système 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 (cf. paragraphe 4.2).

Trois tâches sont possibles en utilisant des modalités différentes :

- changer l'orientation et la localisation de l'utilisateur mobile ;
- manipuler (récupérer, placer ou supprimer) une note ;
- quitter le système.

Dans Memo, il y a cinq modalités actives ou passives pour les entrées. Les modalités actives sont utilisées pour passer une commande vers l'ordinateur (p.ex. une commande vocale). En considérant une modalité [23] comme un couple d'un dispositif physique et un langage, les trois modalités actives de 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, comme le suivi de l'oeil dans la manifestation « mets ça là » (put that there) [23] 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 sont utilisées pour déterminer la position de l'utilisateur afin que l'application puisse afficher les notes sur le casque semi-transparent et permettre à l'utilisateur de sélectionner une note.

7.2 Niveaux d'abstraction pour le test

.

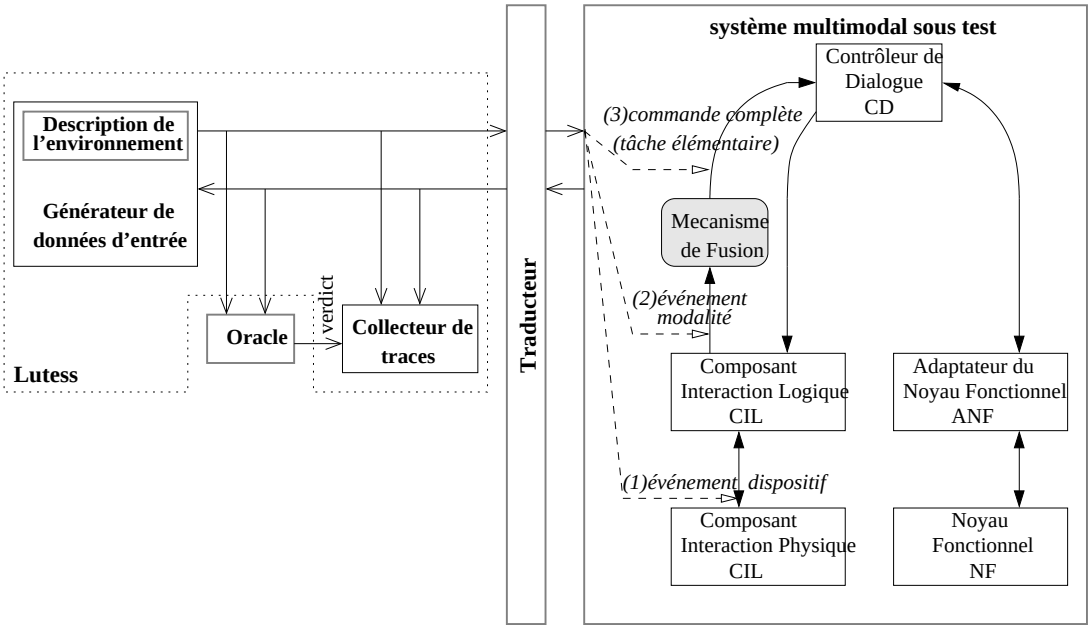

FIG. 7.1 – Connexion Lutess-système multimodal organisé selon le modèle PAC-Amodeus

La connexion d'un système multimodal avec Lutess nécessite la définition du niveau d'abstraction des événements échangés. En fait, le niveau d'abstraction de ces événements détermine le composant du système multimodal qui sera connecté avec Lutess. Cela nous amène à faire des hypothèses sur l'architecture du système multimodal à tester. On considère que ce dernier est construit selon le modèle PAC-Amodeus [23]. Selon ce modèle, un système multimodal est composé de cinq composants principaux et dispose d'un Mécanisme de Fusion des événements de plusieurs modalités (cf. figure 7.1). Le Noyau Fonctionnel représente le fonctionnement conceptuel du système et implémente des concepts dans un domaine précis. L'Adaptateur de Noyau Fonctionnel est l'intermédiaire entre le Contrôleur de Dialogue et les concepts implémentés dans le Noyau Fonctionnel. Le Contrôleur de Dialogue est responsable de la succession des tâches, contrôle l'enchaînement des états liés à l'interaction et réalise la dynamique de l'interface. Le Composant Interaction Logique est l'intermédiaire entre les événements logiques et les événements de niveau dispositif. Les événements logiques issues du Composant Interaction Logique vers le Mécanisme de Fusion sont des événements de niveau modalité. Enfin, le Composant Interaction Physique fournit

l'interaction physique avec l'utilisateur et dépend des dispositifs physiques.

Puisque notre méthode se concentre sur le test d'interaction multimodale en entrée, trois composants de PAC-Amodeus sont concernés : Composant Interaction Physique, Composant Interaction Logique et Mécanisme de Fusion.

Cela revient à identifier trois niveaux d'abstraction possibles pour les événements générés :

1. Simulation du Composant Interaction Physique : les événements générés doivent être envoyés au Composant Interaction Logique. Dans ce cas, on génère des événements de bas niveau (dispositif) tels que les clics souris.
2. Simulation du Composant Interaction Physique et Logique : les événements générés doivent être envoyés au Mécanisme de Fusion. Par conséquent, on génère des événements correspondent aux modalités (dispositif et langage), comme, par exemple, <mouse, empty trash> ou <speech, empty trash>. Puisque dans ce chapitre on s'intéresse à tester la multimodalité en entrée, nous avons opté pour cette solution. En effet, ce choix permet de conserver une connaissance fine des événements d'entrée et de leurs modalités.

   Il existe deux possibilités pour les événements de sortie :

   – Les événements de sortie sont récupérés en sortie du Contrôleur de Dialogue vers le Composant Interaction Logique (sortie). Cela permet de tester l'ensemble de composants du Mécanisme de Fusion, Contrôleur de Dialogue, Adaptateur de Noyau Fonctionnel et Noyau Fonctionnel. Les propriétés qui peuvent être validées par ce type de connexion sont les propriétés du mécanisme de fusion (propriétés CARE) ainsi que les propriétés fonctionnelles. Cette connexion a été choisie pour tester l'aspect multimodal de Memo.
   – Les événements de sortie correspondent aux commandes complètes et sont récupérés en sortie du Mécanisme de Fusion vers le Contrôleur de Dialogue. Cela permet de tester uniquement le Mécanisme de Fusion et les propriétés qui peuvent être validées par ce type de connexion sont les propriétés CARE.
3. Simulation du Mécanisme de Fusion : les événements générés doivent être envoyés au Contrôleur de Dialogue et correspondent à des commandes complètes indépendantes des modalités utilisées pour les spécifier (par exemple

<empty trash>).

Il existe deux possibilités pour les événements de sortie :

- Les événements de sortie sont récupérés en sortie du Contrôleur de Dialogue vers le Composant Interaction Logique (sortie). Ce choix permet de tester l'ensemble de composants Contrôleur de Dialogue, Adaptateur de Noyau Fonctionnel et Noyau Fonctionnel. Avec cette connexion, les propriétés fonctionnelles (le contrôle d'enchaînement d'états, la succession de tâches, la dynamique de l'interface ...) de l'application interactive sont validées. Cette connexion a été choisie pour le test des propriétés fonctionnelles de Memo (c.f. chapitre 4).
- Les événements de sortie sont récupérés en sortie du Contrôleur de Dialogue vers l'Adaptateur de Noyau Fonctionnel. Cette connexion permet de tester uniquement le Contrôleur de Dialogue. Les propriétés qui peuvent être validées sont des propriétés fonctionnelles.

7.3 Modélisation comportementale

Un système interactif multimodal peut interagir avec plusieurs événements d'entrée et de sortie au même instant.

Nous modélisons son comportement par n-uplets :

$$\mathcal{B}_{\mathcal{MMIS}} =< (I_{M_i})_{i\in\{1,..,n\}}, (O_{M_j})_{j\in\{1,..,m\}}, S, trans > \tag{7.1}$$

où :

- I_{M_i} est l'ensemble d'actions d'une modalité d'entrée M_i et n est le nombre de modalités d'entrée ;
- O_{M_j} est l'ensemble d'actions d'une modalité de sortie M_j et m est le nombre de modalités de sortie ;
- S est un ensemble d'états ;
- $Trans$ est un ensemble de transitions :
  $Trans \subseteq S \times D_I \times (D_O \cup \{\perp\}) \times S$
  Où :
  - $D_I \subseteq \bigcup_{I\in 2^{\{1,..,n\}}} \prod_{i\in I} I_{M_i}$
  - $D_O \subseteq \bigcup_{I\in 2^{\{1,..,m\}}} \prod_{i\in I} O_{M_i}$

– le symbole $\perp$ signifie l'absence de sortie.

Cela signifie qu'à chaque instant, le système interactif multimodal peut recevoir plusieurs événements issus de différentes modalités d'entrée et peut envoyer plusieurs événements issus de différentes modalités de sortie.

7.3.1 Exemple : modélisation de Memo

$$\mathcal{B}_{Memo} = < (I_{Localisation}, I_{Orientation}, I_{Mouse}, I_{Keyboard}, I_{Speech}), (O_{Screen}, O_{Voice}), S, trans >$$

où :

– $I_{Localisation}, I_{Orientation}, I_{Mouse}, I_{Keyboard}, I_{Speech}$ sont les ensembles d'actions des modalités d'entrée :

  – $I_{Localisation} = \{move\}$
  Localisation est une modalité passive dont le langage est constitué des trois dimensions de la localisation. Ces dimensions changent quand l'utilisateur mobile se déplace ;

  – $I_{Orientation} = \{turn\}$
  De manière similaire *Orientation* est une modalité passive dont le langage est consitué des trois angles d'orientation.

  – $I_{Mouse} = \{get_{Mouse}, set_{Mouse}, remove_{Mouse}\}$: ensemble des actions de la modalité active *"Souris"* ;

  – $I_{Keyboard} = \{get_{Keyboard}, set_{Keyboard}, remove_{keyboard}\}$: ensemble des actions de la modalité active *"Keyboard"* ;

  – $I_{Speech} = \{get_{Speech}, set_{Speech}, remove_{Speech}\}$: ensemble des actions de la modalité active *"Voix"* ;

– O_{Screen}, O_{Voice} sont les ensembles d'actions des modalités de sortie :

  – $O_{Screen} = \{memoTaken_{Screen}, memoSet_{Screen}, memoRemoved_{Screen}\}$
  où $memoTaken_{Screen}, memoSet_{Screen}, memoRemoved_{Screen}$ correspondent à des messages affichés par l'application : *"memo is taken"*, *"memo is set"*, *"memo is removed"* lorsque l'utilisateur vient de récupérer une note, vient de poser une note ou vient de supprimer une note ;

  – $O_{Voice} = \{memoTaken_{Voice}, memoSet_{Voice}, memoRemoved_{Voice}\}$
  où $memoTaken_{Voice}$, $memoSet_{Voice}$, $memoRemoved_{Voice}$ correspondent

à des messages énoncés par l'application : *"memo is taken"*, *"memo is set"*, *"memo is removed"* lorsque l'utilisateur vient de récupérer une note, vient de poser une note ou vient de supprimer une note ;

- $S = \{s_1, s_2, s_3, s_4\}$ où :
  - $s_1 = memoDisplayed \wedge \neg memoCarried\}$: état où l'application affiche au moins une note et l'utilisateur ne porte pas de note
  - $s_2 = \neg memoDisplayed \wedge memoCarried$: état où l'application n'affiche pas une note et l'utilisateur porte une note
  - $s_3 = memoDisplayed \wedge memoCarried$: état où l'application affiche au moins une note et l'utilisateur porte une note
  - $s_4 = \neg memoDisplayed \wedge \neg memoCarried$: état où l'application n'affiche pas une note et l'utilisateur ne porte pas de note
- Pour la relation *trans*, nous allons présenter à titre d'exemple les transitions sortantes de s_1
  - $trans(s_1, move, \perp, s_1)$, $trans(s_1, move, \perp, s_4)$,
  - $trans(s_1, turn, \perp, s_1)$, $trans(s_1, turn, \perp, s_4)$,
  - $trans(s_1, (move, turn), \perp, s_1)$, $trans(s_1, (move, turn), \perp, s_4)$,
  - $trans(s_1, cmdget, (memoTaken_{Screen}, memoTaken_{Voice}), s_2)$,
  - $trans(s_1, cmdget, (memoTaken_{Screen}, memoTaken_{Voice}), s_3)$,
  - $trans(s_1, cmdremove, (memoRemoved_{Screen}, memoRemoved_{Voice}), s_1)$,
  - $trans(s_1, cmdremove, (memoRemoved_{Screen}, memoRemoved_{Voice}), s_4)$,

    où :
    - Si Memo est configuré avec *la Redondance* entre les deux modalités *(Mouse, Speech)* :
      - $cmdget = (get_{Mouse}, get_{Speech})$
      - $cmdremove = (remove_{Mouse}, remove_{Speech})$
    - Si Memo est configuré avec *l'Équivalence* entre les trois modalités *(Mouse, Keyboard, Speech)* :
      - $cmdget = get_{Mouse} \mid get_{Keyboard} \mid get_{Speech}$
      - $cmdremove = remove_{Mouse} \mid remove_{Keyboard} \mid remove_{Speech}$
    - .....
  - ....

7.4 Adéquation de l'approche synchrone

Dans le paragraphe 4.4 nous avons montré qu'il est possible de synchroniser une trace asynchrone. Pour tester un système interactif avec l'approche synchrone, nous avons introduit un traducteur qui est l'intermédiaire entre le côté synchrone et le côté asynchrone. Il prend en charge la désynchronisation des événements d'entrée du système interactif et la synchronisation de ses événements de sortie.

Dans le cas d'un système interactif multimodal, ce dernier peut recevoir plusieurs événements d'entrée au même instant (cf. paragraphe 7.3), issus de modalités différentes. Donc, à chaque instant, le traducteur doit prendre en charge la lecture des événements concernant toutes les modalités pour les désynchroniser et les envoyer au système sous test par les modalités adéquates. On distingue deux types d'événements d'entrée :

- les événements d'entrée pour les modalités actives et
- les événements d'entrée pour les modalités passives.

Les modalités actives sont utilisées pour passer une commande vers l'ordinateur. Du côté synchrone du traducteur, une variable booléenne est associée à chaque événement d'entrée de chaque modalité active. À chaque cycle, les valeurs des variables booléennes associées aux événements d'entrée des modalités actives 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. Memo a trois modalités actives : (Souris, Commandes de Bouton), (Microphone, Commandes Vocales) et (Clavier, Commandes de Clavier). Les événements qui peuvent passer par chacune de ces modalités sont : *"get"*, *"set"* et *"remove"* (cf. paragraphe 7.3.1). C'est pourquoi on a trois variables booléennes associées à chaque modalité (cf. figure 7.2) : Mouse[get], Mouse[set], Mouse[remove], Keyboard[get], Keyboard[set], Keyboard[remove], Speech[get], Speech[set] et Speech[remove]. À un instant donné, Mouse[get]= *true* indique que l'utilisateur vient de cliquer sur le bouton de la souris correspondant à la commande *"get"*.

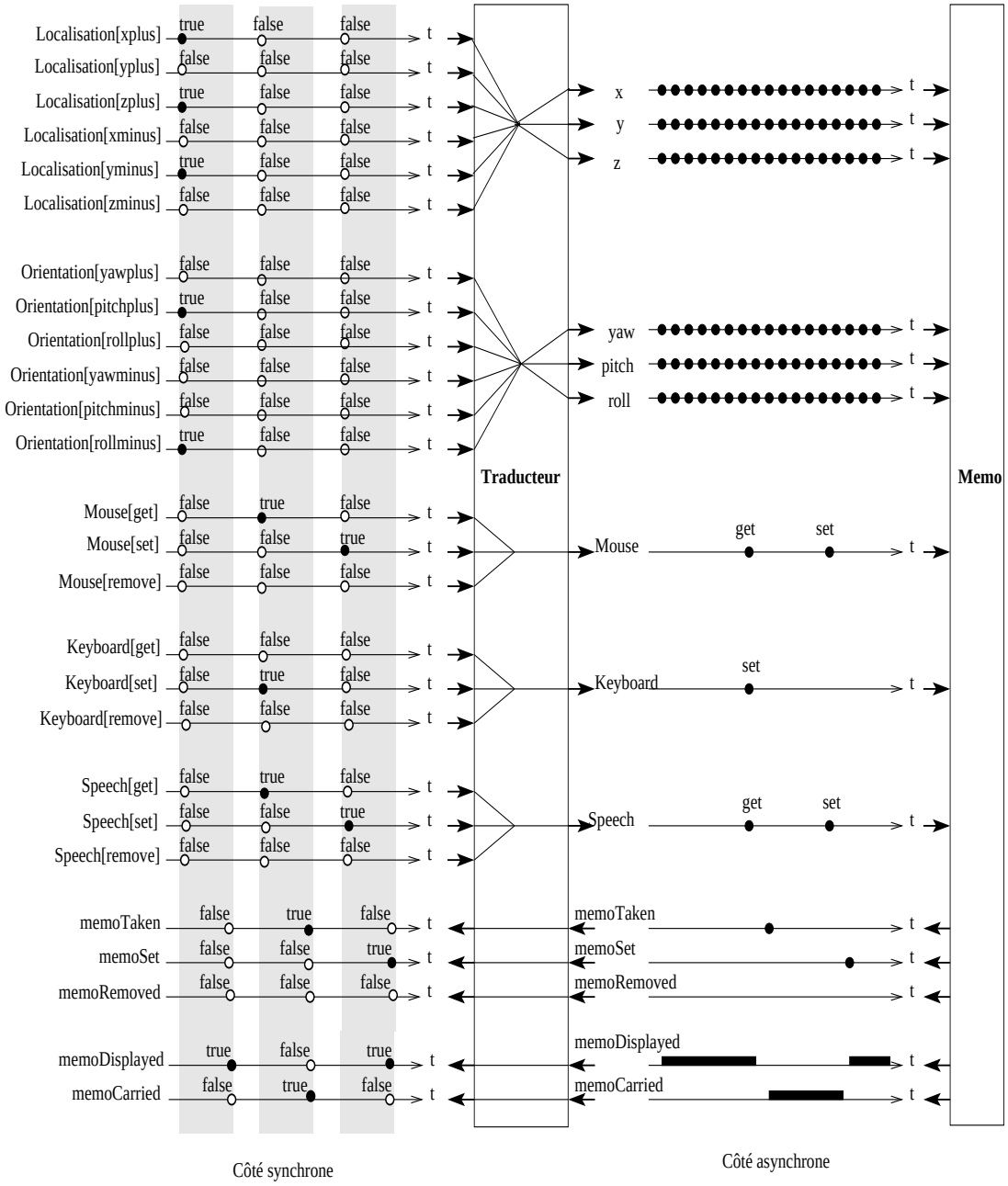

FIG. 7.2 – Adaptation de l'approche synchrone pour le test de systèmes multi-modaux

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 ou la localisation de l'utilisateur mobile de Memo.

À chaque cycle, le traducteur (synchrone - asynchrone) lit les événements synchrones qui modélisent les actions concernant les modalités passives et modifie ses données pour ces modalités. Il envoie également ces données continuellement au système interactif multimodal.

Illustrons ces propos sur Memo, qui possède deux modalités passives : (capteur de Localisation GPS, 3D localisation) et (magnétomètre, les trois angles d'orientation en radians).

Prenons la première modalité concernant la localisation. Les données de cette modalité sont les trois dimensions de localisation. La seule action qui a un effet sur ces données est le déplacement de l'utilisateur (voir le paragraphe 7.3.1). L'utilisateur peut bouger dans l'espace selon les trois axes x, y et z. Il y a plusieurs manières de modéliser ce déplacement :

- avec trois variables numériques (dx, dy, dz) signifiant le déplacement selon les trois axes (x, y, z) et qui sont toujours présentes (chaque cycle) ; l'évaluation (0,0,0) des ces trois variables signifie l'absence de mouvement de l'utilisateur ou
- avec six variables booléennes ($Localisation[xplus]$, $Localisation[yplus]$, $Localisation[zplus]$, $Localisation[xminus]$, $Localisation[yminus]$, $Localisation[zminus]$). À un instant donné, $Localisation[xplus] = true$ signifie que l'abscisse de l'utilisateur augmente. La valeur de l'augmentation ou de la diminution est prédéfinie dans le traducteur.

Dans notre expérimentation (figure 7.2) nous avons choisi la deuxième solution, Lutess ne manipulant pas de données numériques (une extension de Lutess est en cours permettant la prise en compte de telles données [82, 83]).

À chaque cycle, le traducteur lit les valeurs de variables synchrones d'entrée modélisant le déplacement de l'utilisateur, et selon ces valeurs il modifie les 3 dimensions (x, y, z) de la localisation. Les valeurs initiales de ces dimensions sont prédéfinies dans le traducteur. Le traducteur envoie également les valeurs (x, y, z) continuellement à Memo.

De manière similaire, la modalité d'entrée (magnétomètre, les trois angles d'orientation en radians) est adaptée pour le test synchrone comme le montre la fi-

gure 7.2. Les actions synchrones pour cette modalité sont : ($Orientation[yawplus]$, $Orientation[pitchplus]$, $Orientation[rollplus]$, $Orientation[yawminus]$, $Orientation[pitchminus]$, $Orientation[rollminus]$). À un instant donné, $Orientation[pitchplus] = true$ signifie que l'utilisateur baisse la tête d'un angle fixe prédéfini dans le traducteur. Les données de cette modalité, envoyées continuellement à Memo, sont les trois angles d'orientation (yaw, roll, pitch).

Evénements de sortie : Comme on l'a expliqué dans le paragraphe 4.4, à la fin de chaque cycle, si un événement quelconque de sortie arrive pendant cet cycle, le traducteur donne la valeur *"true"* à la variable booléenne associée et donne la valeur *"false"* aux variables booléennes associées aux autres événements. Dans Memo, on a trois événements de sortie (*"memoTaken"*, *"memoSet"* et *"memoRemoved"*) où :

$memoTaken = memoTaken_{Screen} \wedge memoTaken_{Voice}$

$memoSet = memoSet_{Screen} \wedge memoSet_{Voice}$

$memoRemoved = memoRemoved_{Screen} \wedge memoRemoved_{Voice}$

Etat de l'application : Une variable v_s est associée à chaque variable d'état de l'application s. À la fin de chaque cycle, le traducteur récupère la valeur de cette variable s et donne cette valeur à la variable. Dans Memo, on a deux variables d'état booléennes *memoDisplayed* et *memoCarried* (cf. figure 7.2).

7.5 Test de systèmes interactifs multimodaux avec Lutess

Dans les applications multimodales, plusieurs modalités peuvent être utilisées de manière séquentielle ou parallèle [26, 73] dans une fenêtre temporelle TW. Or, Lustre ne fournit pas la notion de temps physique. Pour cela, si C est la durée moyenne d'un cycle d'exécution du générateur de Lutess (c'est le temps qui sépare la génération de deux entrées successives), la fenêtre temporelle en Lustre est spécifiée comme le nombre de cycles d'exécution discrets $N = TW \; div \; C$.

7.5.1 Spécification de l'oracle de test

L'oracle de test consiste en des propriétés à valider par le système interactif multimodal sous test. Pour les systèmes interactifs multimodaux, il y a deux types de propriétés intéressantes à valider : les propriétés fonctionnelles et les propriétés concernant l'interaction multimodale. Des exemples de propriétés fonctionnelles ont été fournies dans le paragraphe 4.5.2.1. Dans ce paragraphe, on se concentre sur les besoins concernant la multimodalité et on considère les propriétés CARE et les relations temporelles définies dans le paragraphe 6.1.2.2. Nous allons montrer que ces propriétés peuvent être exprimées avec des expressions Lustre et peuvent donc être incluses dans l'oracle de test [69, 84].

7.5.1.1 Propriétés CARE en Lustre

La figure 7.3 montre des opérateurs temporels utilisés pour exprimer les propriétés CARE en Lustre présentées dans cette section.

Équivalence

Deux modalités $M1$ et $M2$ sont équivalentes par rapport à un ensemble T de tâches, si chaque tâche $t \in T$ peut être provoquée par un événement issu de $M1$ ou de $M2$. En d'autres mots, une de ces modalités peut remplacer l'autre pour provoquer les tâches de T.

Supposons que $eAM1$ est un événement issu de $M1$, $eAM2$ un événement issu de $M2$. $eAM1$ ou $eAM2$ peut solliciter la tâche $tA \in T$. Alors on peut exprimer l'équivalence entre les deux événements $eAM1$, $eAM2$ déclenchant la tâche tA comme suit :

```
once_from_to (eAM1 or eAM2 , not tA , tA)
```

Complémentarité et Redondance

Les deux propriétés *Redondance* et *Complémentarité* décrivent l'utilisation combinée de modalités. Cette utilisation doit survenir dans la même fenêtre temporelle TW.

```
--lastOccurrence (A) retourne le nombre de cycles passés
--depuis la dernière fois la valeur de A a été vrai si A a été
--vrai au moins une fois avant, sinon -1
node lastOccurrence ( A : bool) returns ( NB : int)
let
    NB = if A then 0
         else  if after (A) then (1+ pre NB)
               else -1 ;
tel

--atMostOne_since (A,B) est vrai si A a été vrai au plus
--une fois depuis le dernier instant où B a été vrai
node atMostOne_since (A, B : bool) returns (res :bool)
let
   res = B
         or always_since(not A, B)
         or (A  and (true -> pre always_since(not A, B)))
         or (not A and (true -> pre res)) ;
tel

--always_since_ticks( A,N) est vrai si A est toujours vrai depuis
--N instants (tops d'horloge)
node always_since_ticks( A : bool ; N :int)
               returns ( os : bool)
let
   os =  with N=0 then A
         else ( A-> A and pre always_since_ticks(A, N-1)) ;
tel

--atMostOne_since_ticks( A,N) est vrai si A a été vrai au plus
--une fois depuis N instants (tops d'horloge)
node atMostOne_since_ticks( A : bool ; N :int)
                returns ( ast : bool)
let
   ast = with N=0 then true
         else (A and (true->
                        pre always_since_ticks(not A, N-1))
              or
              not A and (true->
                           pre atMostOne_since_ticks(A,N-1))
              ) ;
tel
```

FIG. 7.3 – Opérateurs temporelles utilisés dans l'expression de propriétés CARE

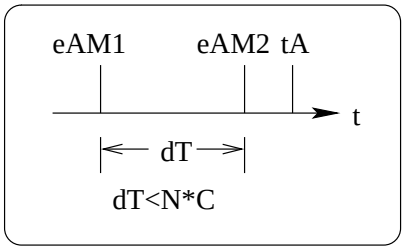

FIG. 7.4 – La propriété Redondance

Complémentarité La complémentarité entre deux modalités $M1$ et $M2$ par rapport à un ensemble T de tâches signifie que pour effectuer une tâche $tAB \in T$, il faut deux événements $eAM1$, $eBM2$, chaque événement complètant l'autre pour provoquer la tâche tAB. Ces deux événements doivent être proches temporellement (dans la même fenêtre temporelle de durée TW) issus de ces deux modalités, c'est-à-dire : $abs(time(eAM1) - time(eBM2)) < TW$.

Supposons que N est le nombre de cycles discrets exécutés pendant la durée complète de la fenêtre temporelle (calculée comme dans le paragraphe 7.5), la propriété de Complémentarité en Lustre est :

```
implies (tAB,
        (after(eAM1) or eAM1)
         and (after(eBM2) or eBM2)
         and
         abs (lastOccurrence(eAM1)
              -lastOccurrence(eBM2))<= N
         and atMostOne_since(tAB, eAM1)
         and atMostOne_since(tAB, eBM2)
         ) ;
```

Cette propriété signifie qu'une exécution d'une tâche tAB implique d'abord l'existence de deux événements $eAM1$, $eBM2$ issus de $M1$, $M2$, que la distance temporelle discrète entre les deux événements est inférieure à N et qu'il y a une seule tâche tAB qui s'exécute le cas échéant.

Redondance La propriété Redondance ressemble à la Complémentarité, mais dans le cas de la Redondance, les deux événements issus de deux modalités

doivent avoir le même sens, tandis que pour la Complémentarité un événement complète l'autre pour provoquer la tâche correspondante.

On dit que deux modalités $M1$ et $M2$ sont redondantes par rapport à un ensemble partiel T de tâches, si chaque tâche $tA \in T$ doit être déclenchée par deux événements $eAM1$, $eAM2$ portant le même sens et issus de $M1$ et de $M2$ simultanément ou séquentiellement dans la même fenêtre temporelle dont la durée est TW. $(abs(time(eAM1) - time(eAM2)) < TW)$. Si les deux événements $eAM1$, $eAM2$ ont lieu, une seule tâche tA sera exécutée. La propriété de Redondance peut être écrite en Lustre comme suit :

```
implies (tA,
        (after(eAM1) or eAM1)
        and (after(eAM2) or eAM2)
        and
        abs (lastOccurrence(eAM1)
             - lastOccurrence(eAM2))<= N
        and
        atMostOne_since(tA, eAM1)
        and atMostOne_since(tA, eAM2)
        ) ;
```

Cette propriété signifie qu'une exécution d'une tâche tA implique d'abord l'existence de deux événements portant le même sens $eAM1$, $eAM2$ issus de deux modalités redondantes $M1$, $M2$, que la distance temporelle entre les deux événements est inférieure à $N * C$ et qu'il y a une seule tâche tA qui s'exécute pour les deux événements (voir la figure 7.4).

Assignation

On dit qu'une modalité M est une assignation pour effectuer un ensemble partiel T de tâches, quand chaque tâche $t \in T$ ne peut être provoquée que par une tâche issue de M. Supposons que la tâche système $tA \in T$ ne peut être provoquée que par l'événement eAM issue de M. Cette propriété peut être écrite en Lustre comme suit :

```
once_from_to (eAM , not tA , tA) ;
```

Équivalence-Redondance

La propriété d'Équivalence-Redondance combine l'Équivalence et la Redondance. Pour deux événements portant le même sens $eAM1$, $eAM2$ issus de $M1$ et $M2$ dans la même fenêtre temporelle, il y a une seule tâche tA qui s'exécute. Cependant, un seul événement peut provoquer l'exécution de la tâche tA :

```
OnceFromTo ( eAM1 or eAM2 , not tA , tA )  -- équivalence
and
implies(tA, atMostOne_since_ticks(tA,N) );   -- redondance.
```

7.5.1.2 Exemple de propriétés CARE

Si l'application Memo est configurée avec la redondance entre les deux modalités *"Mouse"* et *"Speech"*, pour valider qu'une note est récupérée uniquement après l'occurrence de deux événements redondants *Mouse[get]* et *Speech[get]*, on peut écrire l'oracle de test suivant :

```
node MemoOracle(-- les entrées et les sorties de l'application)
returns(propertyOK :bool) ;
let propertyOK = implies
                 (memoTaken,
                (after(Mouse[get]) or Mouse[get])
                 and (after(Speech[get]) or Speech[get])
                 and
                 abs(lastOccurrence(Mouse[get])-
                     lastOccurrence(Speech[get]))<= N
                 and
                 atMostOne_since(memoTaken, Mouse[get])
                 and atMostOne_since(memoTaken, Speech[get])) ;
tel
```

Le nœud précédent retourne la valeur *true* si

- l'événement *"memoTaken"* est provoqué uniquement par l'occurrence des deux événements redondants *"Mouse[get]"* et *"Speech[get]"*,
- arrivant dans la même fenêtre temporelle (de durée discrète N),
- et s'il y a une seule occurrence de *"memoTaken"* pour les deux événements *"Mouse[get]"* et *"Speech[get]"*.

7.5.2 Génération de données de test

Pour tester la multimodalité d'une application multimodale sous test, Lutess doit génère des événements correspondants aux modalités (cf. paragraphe 7.2). Les contraintes d'environnement porteront sur ces événements. Par exemple, pour les modalités actives (cf. paragraphe 6.1.1), une commande par modalité peut être uniquement envoyée à un instant donné.

En utilisant la technique de guidage par les profils opérationnels, on peut générer des événements dans la même fenêtre temporelle. Cela permet de tester l'utilisation synergique de modalités (les propriétés Redondance et Complémentarité). Pour générer des événements dans la même fenêtre temporelle, on associe aux ces événements une probabilité calculée à partir de la durée de la fenêtre temporelle. Supposons que N est le nombre de cycles discrets exécutés pendant la durée complète de la fenêtre temporelle (calculée comme dans la section 7.5). Si on impose que la probabilité de l'occurrence d'un événement d'entrée est $1/N$, alors cet événement arrivera environ une fois tous les N cycles (c'est-à-dire environ une fois dans la fenêtre temporelle). 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 événements A et B soient générés dans cet ordre et dans la même fenêtre temporelle, on peut écrire :

$$proba(B,\, 1/N,\, after(A)\, and\, pre\, always\_since(not\, B,\, A)); \qquad (7.2)$$

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

7.5.2.1 Exemple : L'environnement de Memo

Dans le cas du test de la multimodalité de Memo, les contraintes de l'environnement sont les mêmes que celles du paragraphe 4.5.3.1, sauf en ce qui concerne

la contrainte (4) 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 [1] suivantes :

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

7.5.2.2 Exemple : guidage de génération de Memo

Dans le paragraphe 4.5.3.2, nous avons vu comment on peut utiliser des techniques de guidage pour obtenir des scénarios de test intéressants. Dans ce paragraphe nous allons voir comment on peut utiliser ces techniques afin d'obtenir des scénarios intéressants pour tester la multimodalité.

Dans les traces présentées dans cette section, on va utiliser les abréviations suivantes :

- *MsGet, MsRem, SpGet, SpRem* pour *Mouse[get], Mouse[remove], Speech[get], Speech[remove]* ;
- *mDis, mCar, mTak, mRem* pour *memoDisplayed, memoCarried, memoTaken, memoRemoved*

Utilisation de profils opérationnels

Nous avons utilisé des profils opérationnels afin de tester les propriétés CARE de Memo. La complémentarité 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 présentées ci-dessous montrent que l'équivalence a été testée considérablement : les trois modalités

[1] *Mouse* est un tableau de trois éléments booléens. Les indices de ce tableau sont *"get", "set"* et *"remove"* : *AtMostOne(3, Mouse)* signifie que au plus un élément du tableau *Mouse* a la valeur *vrai*.

1	-	-	-	-	-
2	-	-	mDis	-	-
3	MsGet	SpGet	mDis	mCar	mTak
4	-	-	mDis	mCar	-
5	MsGet	-	mDis	mCar	-

TAB. 7.1 – Un extrait de la trace de Memo

équivalentes, la souris, le clavier et la parole, sont fréquemment simulées. La redondance a également été testée. L'assignation n'a pas été testée, car elle concerne seulement la commande "exit" dans Memo.

Validation de la redondance Pour valider l'utilisation redondante de deux modalités, la souris et la parole, nous avons reconfiguré l'application Memo avec le mode redondance. 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 affichée sur le viseur.

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

Le tableau 7.1 montre un extrait de la trace résultante. On commence le test dans un état où il n'y a aucune note près de l'utilisateur. Le pas 2 contient l'événement $memoDisplayed(mDis)$, indiquant qu'une ou plusieurs notes sont près de l'utilisateur. Dans le pas 3, les deux événements simultanés *mouseGet* et *speechGet* $(MsGet, SpGet)$ provoquent, à cause de la redondance, la récupération d'une note $(mTak)$. Alors, une note est toujours visible $(mDis)$, et l'utilisateur porte une note $(mCar)$. On observe que dans le pas 5, le seul événement $mouseGet(MsGet)$ ne provoque aucune réaction, parce qu'avec ce mode, deux événements redondants sont nécessaires pour effectuer une tâche.

– Supposons que la situation suivante doit être testée : quand une note est affichée sur le viseur, 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 a utilisé la formule 7.2 : on donne une probabilité pr pour 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 mêmes 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.

$$
\begin{array}{l}
proba(\\
(Speech[get], \quad pr, \quad pre\, memoDisplayed\ and\ after(Mouse[get])\ and \\
\qquad\qquad\qquad\quad pre\ always\_since(not\ Speech[get], Mouse[get]) \\
), \\
(Mouse[get], \quad 0, \quad pre\, memoDisplayed\ and\ after(Mouse[get])\ and \\
\qquad\qquad\qquad\quad pre\ always\_since(not\ Speech[get], Mouse[get]) \\
));
\end{array}
\tag{7.3}
$$

On va illustrer le choix de la valeur pr. Supposons que TW = 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'entrée 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 la dernière occurrence de *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 :

Le tableau 7.2 montre un extrait de la trace résultante pour pr = 0.2. On peut noter que la distance temporelle entre les deux événements *mouseGet* et *speechGet (MsGet, SpGet)* est égale à 8 cycles (8000 ms), qui est plus long que TW, et donc aucune tâche n'est exécutée.

Le tableau 7.3 montre un extrait de la trace résultat pour pr = 0.8. On peut noter que dans cette trace les événements $MsGet$ et $SpGet$ sont plus proches que au cas où pr = 0.2. On peut également observer que les deux événements

1	MsGet	-	mDis	-	-
2	-	-	mDis	-	-
3	-	-	mDis	-	-
4	-	-	mDis	-	-
5	-	-	mDis	-	-
6	-	-	mDis	-	-
7	-	-	mDis	-	-
8	-	-	mDis	-	-
9	-	SpGet	mDis	-	-
10	-	-	mDis	-	-
11	-	SpGet	mDis	-	-
12	-	-	mDis	-	-

TAB. 7.2 – Un extrait de la trace de Memo (mode Redondance Mouse-Speech) avec un guidage défini avec la formule 7.3 pour pr = 0.2

146	-	-	mDis	-	-
147	MsGet	-	mDis	-	-
148	-	-	mDis	-	-
149	-	SpGet	mDis	mCar	mTak
150	-	-	mDis	mCar	-

TAB. 7.3 – Un extrait de la trace de Memo avec un guidage défini avec la formule 7.3 pour pr = 0.8

7	-	-	mDis	-	-
8	MsGet	-	mDis	mCar	mTak
9	-	-	mDis	mCar	-
10	-	-	mDis	mCar	-
11	-	SpGet	mDis	mCar	-
12	-	-	mDis	mCar	-

TAB. 7.4 – Un extrait de la trace de Memo (mode Équivalence-Redondance Mouse-Speech) avec un guidage défini avec la formule 7.3
pour pr = 0.8 and N = 5000/1000 = 5 cycles

MsGet et *SpGet*, qui arrivent dans la même fenêtre temporelle, provoquent la prise d'une note *mTak*.

Validation de la Équivalence-Redondance : Dans cette expérimentation, on utilise la même formule de celle de l'expérimentation précédente (la formule 7.3), mais on a reconfiguré le système avec le mode «Équivalence-Redondance». Avec ce mode, le système utilise les deux modes Équivalence et Redondance en même temps : pour deux événements apparaissant dans la même fenêtre temporelle et portant la même information, il y a seulement une seule tâche exécutée (mode Redondance). Cependant, un seul événement peut provoquer l'exécution d'une tâche (mode Équivalence).

Le tableau 7.4 montre un extrait de la trace résultat avec pr = 0.8 and N = 5000/1000 = 5 cycles. On peut observer que dans l'étape 8, à cause de l'équivalence, l'événement *Mouse[Get] (MsGet)* provoque la capture d'une note. Ensuite l'événement *Speech[Get] (SpGet)* à l'étape 11 est ignoré, à cause de la redondance (la distance temporelle entre ces deux événements est de 3 cycles, donc moins élevée que la durée de la fenêtre temporelle (5 cycles)).

Guidage par schéma (scénario)

On considère un scénario qui 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 (*pre memoDisplayed*). L'utilisateur récupère une note (note 1) de manière redondante, en utilisant la souris et la parole au même instant. Ensuite, l'utilisateur supprime une note en utilisant aussi la souris

1	-	-	-	-	mDis	-	-	-
2	MsGet	-	SpGet	-	mDis	mCar	mTak	-
3	-	MsRem	-	-	mDis	mCar	-	-
4	-	-	-	SpRem	mDis	mCar	-	-
5	-	-	-	-	-	mCar	-	mRem

TAB. 7.5 – Un extrait de la trace de Memo

et la parole, à deux instants différents appartenant à la même fenêtre temporelle. Le scénario est exprimé comme suit :

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

Le tableau 7.5 montre un extrait de la trace résultat. Dans cette trace, le premier pas contient l'événement *memoDisplayed(mDis)*, signifiant qu'une ou plusieurs notes sont proches de l'utilisateur. Au deuxième pas, les deux événements simultanés *Mouse[get]* et *Speech[get] (MsGet, SpGet)* provoquent la capture d'une note (événement *mTak* au pas 2). L'événement *memoDisplayed* (*mDis*) a toujours la valeur *vraie*. Ce qui signifie qu'une autre note est visible. Les pas 3 et 4 contiennent les événements Mouse[removed] et Speech[removed] (*MsRem* et *SpRem*) qui provoquent la suppression de la note visible (l'événement *mRem* dans le cinquième pas), parce que les deux événements (*MsRem*, *SpRem*) appartiennent à la même fenêtre temporelle.

7.5.3 Conclusion

Tester les systèmes interactifs multimodaux avec Lutess nécessite de spécifier l'environnement de l'application de sorte à pouvoir générer des événements au niveau de la modalité. L'aspect multimodal est pris en compte essentiellement

au moyen des propriétés CARE (*Complémentarité, Assignation, Redondance* et *Équivalence*). Nous avons proposé une expression de ces propriétés en Lustre, ce qui permet de les inclure dans un oracle automatique. De plus, en utilisant les profils opérationnels on a montré qu'il est possible de générer des événements de modalités synergiques, proches temporellement (dans la même fenêtre temporelle). Pour cela, on s'est basé sur l'hypothèse que la durée de la fenêtre temporelle peut se définir par le nombre (discret) de cycles d'exécution. Cela permet de valider fonctionnelement le moteur de fusion mais est insuffisant si l'on s'intéresse de manière très précise au temps. Dans une telle perspective, le modèle de temps discret de Lustre (et de Lutess) serait insuffisant (il faudrait un modèle prenant explicitement en compte le temps comme cela est fait dans l'approche ICO présentée dans le paragraphe 6.2.1).

Il en est toujours que l'utilisation de Lutess telle quelle est fastidieuse, car le formalisme de description des directives de génération de tests est d'un niveau difficilement abordable par des concepteurs d'applications interactives. De manière similaire au chapitre 5, on propose donc dans le paragraphe suivant quelques perspectives sur la génération de tests à partir d'arbre de tâches pour les systèmes multimodaux.

7.6 Vers une génération à partir de CTT incluant la multimodalité

Afin d'utiliser l'arbre de tâches pour générer les événements pour les systèmes interactifs multimodaux, deux solutions sont envisageables :

1. Descendre dans l'arbre de tâches jusqu'au niveau des modalités [81] (cf. paragraphe 6.2.3).
2. Concevoir un autre modèle pour spécifier le moteur du fusion qui décrit la relation entre les tâches élémentaires de l'arbre de tâches et les événements de différentes modalités.

Avoir deux modèles (l'arbre de tâches et le moteur du fusion) ressemble à l'approche de la modélisation présentée dans le paragraphe 6.2.2 (algèbre de processus pour les systèmes multimodaux). Dans ce travail, les énoncés sont compa-

rables aux tâches élémentaires de l'arbre de tâches et la production des énoncés à partir d'événements de différentes modalités est comparable au moteur du fusion.

Pour la première solution qui consiste à descendre dans l'arbre de tâches jusqu'à l'arrivée au niveau d'événements de modalités on peut distinguer trois cas :

1. Cas où on a la propriété *Équivalence* entre les deux modalités m_1 et m_2 afin de solliciter un ensemble de tâches T. $Events_{m_1}$ et $Events_{m_2}$ sont respectivement les deux ensembles des événements de m_1 et m_2. $tA \in T$ est une tâche élémentaire de l'arbre de tâches. Grâce à *l'Équivalence,* il existe $eA_1 \in Events_{m_1}$, $eA_2 \in Events_{m_2}$ tel que la tâche tA peut être provoquée soit par l'événement eA_1, soit par eA_2 (cf. paragraphe 6.1.2.2). Pour descendre dans l'arbre de tâches jusqu'à l'arrivée au niveau de la modalité et en utilisant les opérateurs de CTT , on peut écrire $tA = eA_1 [] eA_2$.
2. Cas où on a la propriété *Assignation* de la modalité m pour effectuer un ensemble T de tâches. $tA \in T$ est une tâche élémentaire de l'arbre de tâches. Grâce à *l'Assignation,* il existe $eA \in Events_m$ tel que la tâche tA est provoquée uniquement par l'événement eA (cf. paragraphe 6.1.2.2). Pour descendre dans l'arbre de tâches jusqu'à l'arrivée au niveau événements modalité, on peut écrire $tA = eA$.
3. Cas où on a la propriété *Redondance* (ou Complémentarité) entre m_1 et m_2 pour un ensemble de tâches T. $tA \in T$ est une tâche élémentaire de l'arbre de tâches. Grâce à la *Redondance,* il existe $eA_1 \in Events_{m_1}$, $eA_2 \in Events_{m_2}$ tel que la tâche tA ne peut être provoquée que par les deux événements eA_1 et eA_2 issus simultanément ou séquentiellement dans la même fenêtre temporelle (cf. paragraphe 6.1.2.2). Donc, il faut trouver l'opérateur adéquat qui exprime ce fait. Par exemple, si on utilise un opérateur de parallélisme ("||") comme dans [81] (cf. paragraphe 6.2.3), on peut écrire $tA = eA_1 || eA_2$.

Pour la deuxième solution qui consiste à concevoir un autre modèle $\mathcal{MF}$ pour spécifier le *Moteur de Fusion* décrivant la relation entre les tâches élémentaires de l'arbre de tâches et les événements de différentes modalités d'entrée,

considérons par exemple le modèle suivant :

$$\mathcal{MF} =< \mathbb{T}, \mathbb{M}, (Events_{m_i})_{i\in\{1..n\}}, \approx, \mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{E} >$$

où :

- $\mathbb{T}$ est l'ensemble de tâches élémentaires de l'arbre de tâches pour l'application interactive multimodale à tester (on va dénoter T pour un sous-ensemble de $\mathbb{T}$) ;
- $\mathbb{M} = \{m_1, m_2, ...m_n\}$ est un ensemble de modalités d'entrée (on va dénoter M pour un sous-ensemble de $\mathbb{M}$) ;
- $Events_{m_1}, Events_{m_2}, ..., Events_{m_n}$ sont les événements de modalités m_1, m_2, ..., m_n ;
- $\approx \subseteq (\bigcup_{i\in\{1,..,n\}} \mathbb{T} \times Events_{m_i}) \cup (\bigcup_{i\in\{1,..,n\}} Events_{m_i} \times \mathbb{T}) \cup (\bigcup_{i,j\in\{1,..,n\}} Events_{m_i} \times Events_{m_j})$
  est une relation d'équivalence entre les tâches élémentaires et les événements associés aux modalités. Cette relation identifie les événements qui ont la même signification. Par exemple, pour l'application multimodale Memo [65] (cf. les paragraphes 7.1 et 7.3.1), on peut écrire :
  - $get \approx get_{Mouse} \approx get_{Keyboard} \approx get_{Speech}$
  - $set \approx set_{Mouse} \approx set_{Keyboard} \approx set_{Speech}$
  - $remove \approx remove_{Mouse} \approx remove_{Keyboard} \approx remove_{Speech}$

  où *get*, *set*, *remove* sont des tâches élémentaires de l'arbre de tâches ....etc
  Sur cette relation d'équivalence, on a la contrainte suivante :
  - $\forall m \in \mathbb{M}, \forall t \in \mathbb{T}, \forall e1_m, e2_m \in Events_m : e1_m \approx t, e2_m \approx t \Rightarrow e1_m = e2_m$
    Cette contrainte signifie que pour un ensemble d'événements d'une modalité quelconque, il n'y a pas deux événements qui ont la même signification.
- $\mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{E} \subseteq 2^{\mathbb{M}} \times 2^{\mathbb{T}}$ sont quatre relations de *Complémentarité*, *Assignation, Redondance* et *Équivalence* avec les deux contraintes :

  1. $(M, T) \in \mathcal{A} \Rightarrow card(M) = 1$
     où $card(M)$ est le nombre d'éléments dans l'ensemble M (cela signifie que chaque élément de la relation *Assignation* lie une seule modalité avec un sous-ensemble de tâches).
  2. Supposons que $\mathcal{C} \cup \mathcal{A} \cup \mathcal{R} \cup \mathcal{E} = \{(M_1, T_1), (M_2, T_2), ..., (M_k, T_k)\}$

alors $T_1 \cup T_2 \cup .... \cup T_k = \mathbb{T}$
et *si* $i \neq j$ *alors* $T_i \cap T_j = \phi$

(cela signifie que les quatre relations $\mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{E}$ forment une partition par rapport à l'ensemble de tâches élémentaires $\mathbb{T}$).
Chaque élément de ces relations lie un sous-ensemble de tâches à un sous-ensemble de modalités concernées. Par exemple, si Memo est configuré avec la relation d'équivalence entre les trois modalités (*Mouse*, *Keyboard* et *Speech*) pour l'ensemble de tâches {*get*, *set*, *remove*} on peut écrire :

$$\{Mouse, Keyboard, Speech\}\ \mathcal{E}\ \{get, set, remove\}$$

Dans l'application Memo, la modalité *clavier* (*Keyboard*) est une assignation pour la tâche *exit* :

$$\{Keyboard\}\ \mathcal{A}\ \{exit\}$$

- Le modèle de fusion $\mathcal{MF}$ doit également vérifier la contrainte suivante :
  - $\forall t \in \mathbb{T}$, on a plusieurs cas selon t :
    - Si $t \in T : (M, T) \in \mathcal{C}$ alors $\forall m_i \in M, Events_{m_i} \neq \phi$
    - Si $t \in T : (M, T) \in \mathcal{A}$ alors $M = \{m\}, \exists t_m \in Events_m : t \approx t_m$
    - si $t \in T : (M, T) \in \mathcal{R}$ alors $\forall m_i \in M, \exists t_{m_i} \in Events_{m_i} : t \approx t_{m_i}$
    - si $t \in T : (M, T) \in \mathcal{E}$ alors $\forall m_i \in M, \exists t_{m_i} \in Events_{m_i} : t \approx t_{m_i}$

Génération de données du test

Le modèle $MEF = (Q, qi, qf, I, O, trans)$ (cf. paragraphe 5.1.1) en lequel nous avons transformé l'arbre de tâches ne peut pas être utilisé tel quel pour générer les données de test pour les systèmes multimodaux. En effet, dans ce modèle il y a un seul événement d'entrée étiquetant chaque transition ($trans \subseteq Q \times (I \cup \{\mu\}) \times O \times Q$), donc un seul événement d'entrée à générer à chaque pas de test. Mais, pour les systèmes multimodaux, le générateur doit avoir la possibilité de générer plusieurs événements d'entrée à la fois.

Si on descend dans l'arbre de tâches jusqu'au niveau des modalités, on peut transformer cet arbre de tâches en utilisant la même sémantique que nous avons utilisé dans le paragraphe 5.1.2, en une machine d'états finis similaire mais avec

des transitions permettant plusieurs événements d'entrée/sortie à la fois ($trans \subseteq Q \times 2^I \times 2^O \times Q$).

Si on utilise le $\mathcal{MF}$ *(Moteur de Fusion)* spécifié précédemment avec l'arbre de tâches pour générer les données du test au niveau modalité, on peut suivre la technique suivante :

1. Utiliser la machine d'états finis à E/S $MEF = (Q, qi, qf, I, O, trans)$ issue de la transformation de l'arbre de tâches pour générer les données de test au niveau tâches élémentaires de l'arbre de tâches, comme nous l'avons fait au chapitre 5. Dans ce cas, chaque événement généré est une tâche élémentaire.
2. Pour chaque tâche élémentaire t générée, on utilise $\mathcal{MF}$ pour générer les événements de modalités associées comme suit :
   (a) Parmi les relations $\mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{E}$, chercher la relation qui concerne t (voir les contraintes de $\mathcal{MF}$ spécifiées précédemment) :
      i. Si $t \in T, (M, T) \in \mathcal{C}$ alors en utilisant toutes les modalités $m_i \in M$, un événement quelconque $e \in Evenets_{m_i}$ de chaque modalité doit être généré afin de produire la tâche t.
      ii. Si $t \in T, (M, T) \in \mathcal{A}$ alors en utilisant la seule modalité $m \in M$, un événement $t_m \in Events_m$ équivalent ($t \approx t_m$) de cette modalité doit être généré afin de produire la tâche t.
      iii. si $t \in T, (M, T) \in \mathcal{R}$ alors en utilisant toutes les modalités $m_i \in M$, un événement $t_{m_i} \in Evenets_{m_i}$ équivalent ($t \approx t_{m_i}$) de chaque modalité doit être généré afin de produire la tâche t.
      iv. si $t \in T, (M, T) \in \mathcal{E}$ alors en utilisant une modalité quelconque $m_i \in M$, un seul événement $t_{m_i} \in Evenets_{m_i}$ équivalent ($t \approx t_{m_i}$) doit être généré afin de produire la tâche t.

7.6.1 Profil opérationnel pour les systèmes multimodaux

Afin de spécifier un profil opérationnel de l'utilisateur pour l'aspect multimodal, nous proposons de distribuer les probabilités entre les différentes modalités.

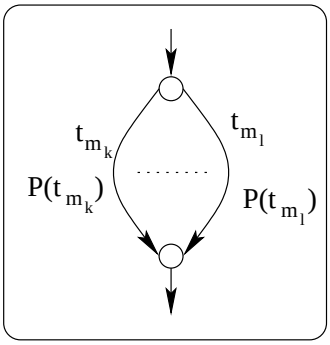

FIG. 7.5 – Distribution de probabilités au cas de l'Équivalence

Considérons un ensemble de modalité $\mathbb{M} = \{m_1, m_2, ..., m_n\}$, avec une distribution de probabilité $pr_{m_1}, pr_{m_2}, ..., pr_{m_n}$, où pr_{m_i} est la probabilité d'utiliser la modalité m_i. On a également la contrainte : $\sum_{m_i \in \mathbb{M}} pr_{m_i} = 1$.

Pendant la génération de données de test, on a plusieurs cas selon la tâche élémentaire t générée par le simulateur de l'arbre de tâches (on suppose que la deuxième solution du paragraphe précédent est utilisée) :

- Si $t \in T$ *et* $(M, T) \in \mathcal{A}$, alors la seule modalité dans l'ensemble M doit être utilisée comme nous avons montré dans le paragraphe précédent.
- Si $t \in T$ *et* $(M, T) \in \mathcal{E}$, la distribution de probabilité entre les modalités doit être respectée en choisissant un seul événement d'une modalité de l'ensemble M comme suit :
  Supposons que $M = \{m_k, ..., m_l\} \subseteq \mathbb{M}$ et $t \approx t_{m_k} \approx .. \approx t_{m_l}$. La probabilité d'un événement t_{m_j} (dénotée $P(t_{m_j})$) est calculée comme suit :
  $P(t_{m_j}) = \frac{pr_{m_j}}{\sum_{m_i \in M} pr_{m_i}}$
  La tâche t est exécutée en choisissant un événement parmi les événements $t_{m_k}, ..., t_{m_l}$ en respectant les probabilités associées à ces événements comme le montre la figure 7.5.
- Si $t \in T$ *et* $(M, T) \in \mathcal{C} \cup \mathcal{R}$, en nous appuyant sur le paragraphe 7.5.2 et en nous inspirant des travaux autour de LOTOS probabiliste [71, 70] et autour des automates probabilistes [85] on peut utiliser les probabilités des modalités afin de générer les événements redondants ou complémentaires dans la même fenêtre temporelle mais pas forcément au même instant.
  Supposons que $M = \{m_1, m_2\}$, et les deux événements $e_1 \in Events_{m_1}$, $e_2 \in Events_{m_2}$ doivent être générés dans la même fenêtre temporelle de

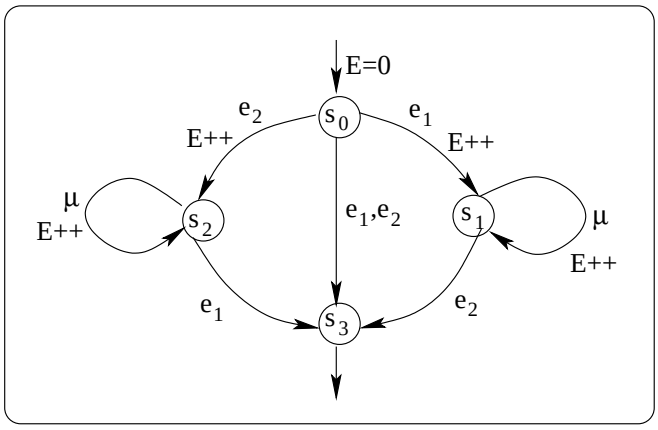

FIG. 7.6 – La génération au cas de Redondance (Complémentarité) entre deux événements

durée W afin d'exécuter la tâche t. C étant la durée d'un cycle d'exécution, $N = W/C$ est le nombre de cycles d'exécution dans la fenêtre temporelle. Nous supposons dans la suite que $W \geq C \Rightarrow N \geq 1$.

On a trois possibilités pour générer e_1, e_2 :

1. Générer e_1, e_2 au même instant.
2. Générer e_1 et ensuite générer e_2 dans un des N cycles suivants (donc les deux événements e_1, e_2 sont générés dans la même fenêtre temporelle).
3. Générer e_2 et ensuite générer e_1 dans un de N cycles suivants.

Pour générer la tâche t, on adopte le comportement spécifié dans la figure 7.6 (rappelons que μ est l'action vide de l'utilisateur). Dans ce modèle, le variable E compte le nombre des étapes (transitions) depuis l'état initial s_0. La valeur de E est dans l'intervalle $[0..N]$. Les probabilités des transitions dépendent de la valeur de E : en effet, plus la valeur de E est élevée, plus la probabilité de générer le deuxième événement — qui n'a pas encore été généré — est élevée.

À l'état s_0, $E = 0$ les probabilités des transitions sortantes sont :

- $P(s_0 \xrightarrow{(e_1,e_2)} s_3)_{E=0} = \frac{pr_{m_1}.pr_{m_2}}{pr_{m_1}+pr_{m_2}+pr_{m_1}.pr_{m_2}}$
- $P(s_0 \xrightarrow{e_1} s_1)_{E=0} = \frac{pr_{m_1}}{pr_{m_1}+pr_{m_2}+pr_{m_1}.pr_{m_2}}$
- $P(s_0 \xrightarrow{e_2} s_2)_{E=0} = \frac{pr_{m_2}}{pr_{m_1}+pr_{m_2}+pr_{m_1}.pr_{m_2}}$

Aux états s_1, s_2 les probabilités des transitions sortantes dépendent du nombre d'étapes depuis l'état s_0 (la valeur de la variable E) :

Si $E = 1$, on a :

- $P(s_1 \xrightarrow{e_2} s_3)_{E=1} = (pr_{e_2})_{E=1}$, où $(pr_{e_2})_{E=1}$ est une valeur aléatoire dans l'intervalle $[\frac{1}{N}..1]$
- $P(s_1 \xrightarrow{\mu} s_1)_{E=1} = 1 - (pr_{e_2})_{E=1}$
- $P(s_2 \xrightarrow{e_1} s_3)_{E=1} = (pr_{e_1})_{E=1}$, où $(pr_{e_1})_{E=1}$ est une valeur aléatoire dans l'intervalle $[\frac{1}{N}..1]$
- $P(s_2 \xrightarrow{\mu} s_2)_{E=1} = 1 - (pr_{e_1})_{E=1}$

Si $E = 2$, on a :

- $P(s_1 \xrightarrow{e_2} s_3)_{E=2} = (pr_{e_2})_{E=2}$, où $(pr_{e_2})_{E=2}$ est une valeur aléatoire dans l'intervalle $[\frac{1}{N-(E-1)}..1]$
- $P(s_1 \xrightarrow{\mu} s_1)_{E=2} = 1 - (pr_{e_2})_{E=2}$
- $P(s_2 \xrightarrow{e_1} s_3)_{E=2} = (pr_{e_1})_{E=2}$, où $(pr_{e_1})_{E=2}$ est une valeur aléatoire dans l'intervalle $[\frac{1}{N-(E-1)}..1]$
- $P(s_2 \xrightarrow{\mu} s_2)_{E=2} = 1 - (pr_{e_1})_{E=2}$
- .....

Si $E = N$,on a :

- $P(s_1 \xrightarrow{e_2} s_3)_{E=N} = 1$
- $P(s_1 \xrightarrow{\mu} s_1)_{E=N} = 0$
- $P(s_2 \xrightarrow{e_1} s_3)_{E=N} = 1$
- $P(s_2 \xrightarrow{\mu} s_2)_{E=N} = 0$

Chapitre 8

Conclusion et travaux futures

8.1 Bilan de la thèse

Le travail que nous venons de présenter s'inscrit dans le cadre de développement de techniques formelles pour la validation de systèmes interactifs. Il vient enrichir et compléter une famille de techniques de génération de données de test pour les systèmes interactifs.

Dans cet objectif, nous avons proposé d'utiliser l'approche synchrone et l'outil de test Lutess. Cet outil propose plusieurs techniques de génération de données : test aléatoire, profils opérationnels ...

Nous avons proposé une technique d'adaptation de l'approche de test synchrone aux systèmes interactifs moyennant la construction d'un traducteur qui prend en charge la synchronisation des événements issus du système interactif sous test et la désynchronisation des événements synchrones générés du Lutess.

L'utilisation de différentes techniques de génération de Lutess permet d'engendrer des scénarios de test intéressants ainsi que de valider des propriétés qu'on peut exprimer en Lustre dans un oracle de test (en général des propriétés fonctionnelles, liées au contrôle).

Nous avons également utilisé l'approche synchrone pour tester des systèmes interactifs multimodaux en entrée. Nous avons ainsi proposé une adaptation de cette approche prenant en compte les événements de différentes modalités. L'aspect multimodal de ces systèmes s'exprime au moyen des propriétés CARE (*Complémentarité, Assignation, Redondance* et *Équivalence*). Nous avons exprimé ces

propriétés en Lustre et nous avons proposé une technique basée sur l'utilisation d'un profil opérationnel pour générer les événements de modalités synergiques, proches temporellement (dans la même fenêtre temporelle).

Afin de générer les données de test pour les systèmes interactifs, le comportement de l'utilisateur doit être spécifié par des propriétés invariantes Lustre. Étant donné que ce modèle synchrone du comportement de l'utilisateur n'est pas facile à construire pour les concepteurs des applications interactives, non familiers de ces langages, nous avons proposé d'utiliser l'arbre de tâches. L'arbre de tâches est transformé en une machine d'états finis à Entrée/Sortie comparable à celle utilisée par Lutess afin de générer les tests. Nous avons également enrichi cet arbre avec des probabilités afin de permettre la spécification et puis la génération de données du test avec des profils opérationnels.

8.2 Perspectives

Nous donnons dans ce paragraphe quelques perspectives qui nous semblent intéressantes :

1. Étudier la génération à partir de l'arbre de tâches guidée par des propriétés de sûreté comme nous l'avons proposé dans la conclusion du chapitre 5 (cf. paragraphe 5.4). Dans ce cas, le générateur favorisera la génération de données qui conduisent dans un état où la propriété de sûreté peut être violée (état suspect). En effet, les propriétés de sûreté peuvent être assimilées à un programme synchrone et peuvent être modélisées par un automate d'état finis (cf. paragraphe 3.6.5). À un état donné de cet automate, les entrées pertinentes sont les entrées qui mènent à un état suspect. Lors de la génération guidée par des propriétés de sûreté, une entrée est choisie parmi ces entrées pertinentes.

   Étant donné que la description de l'environnement de Lutess est assimilée à une machine d'états finis et l'arbre de tâches est également transformé en une machine d'états finis, alors il semble possible de s'inspirer des travaux relatifs à la génération de test guidée par des propriétés de sûreté [62, 63, 64] et l'adapter pour la génération à partir d'arbre de tâches.

2. Dans le chapitre 7, nous avons proposé quelques perspectives sur la géné-

ration de tests à partir d'arbres de tâches pour les systèmes multimodaux (cf. paragraphe 7.6). Afin de générer des tests pour ces systèmes, il faut spécifier l'aspect multimodal de ces systèmes. Nous avons proposé deux solutions : une qui consiste à descendre dans l'arbre de tâches jusqu'au niveau des modalités et une autre qui consiste à concevoir un autre modèle du moteur de fusion qui décrit la relation entre les tâches élémentaires de l'arbre de tâches et les événements de différentes modalités. Dans le paragraphe 7.6, nous avons présenté une version préliminaire d'un modèle de moteur de fusion.

L'avantage de la première solution est la simplicité pour le testeur, tandis que la deuxième solution demande l'utilisation d'un langage supplémentaire pour spécifier le moteur de fusion.

Dans le paragraphe 7.6.1, nous avons également proposé de distribuer des probabilités entre les différentes modalités d'entrée comme un moyen de spécification de profils opérationnels. Pendant la génération de données de test, cela permet de favoriser une modalité en cas d'*Équivalence* entre plusieurs modalités. Dans le cas de la *Redondance* ou de la *Complémentarité*, la spécification de tels profils opérationnels permet de donner plusieurs possibilités afin de générer des événements synergiques dans la même fenêtre temporelle. Nous avons exposé un modèle d'un automate probabiliste pour générer deux événements synergique, qu'on pourrait imaginer de généraliser pour un nombre quelconque d'événements.

Bibliographie

[1] Nielsen J. *Usability Engineering.* MA : Academic Press, Boston, 1993.

[2] Shneiderman B. *Designing the user interface : strategies for effective human-computer interaction.* Reading, MA : Addison-Wesley, 1998.

[3] Abelow D. Automating feedback on software product use. CASE Trends, December 1993.

[4] Scapin D., Leulier C., Vanderdonckt J., Mariage C., Bastien C., France C., Palanque P., and Bastide R. A framework for organizing web usability guidelines. In *the Sixth Conference on Human Factors and the Web*, 2000.

[5] F. Paternò and G. Faconti. On the use of LOTOS to describe graphical interaction. In *HCI'92 : Proceedings of the conference on People and computers VII*, pages 155–173, New York, NY, USA, 1993. Cambridge University Press.

[6] F.Paternò. Definition of user interface properties using action-based temporal logic. In *the Fifth International Conference Software Engineering and Knowledge Engineering*, pages 314–319, S.Francisco, June 1993.

[7] Rémi Bastide, Philippe A. Palanque, Duc-Hoa Le, and Jaime Munoz. Integrating rendering specifications into a formalism for the design of interactive systems. In *DSV-IS*, pages 171–190, 1998.

[8] Xavier Lacaze, Philippe A. Palanque, David Navarre, and Rémi Bastide. Performance evaluation as a tool for quantitative assessment of complexity of interactive systems. In *DSV-IS*, pages 208–222, 2002.

[9] Ph. Palanque and R. Bastide. *Spécifications formelles pour l'ingénierie des interfaces homme machine, Revue Techniques et Sciences Informatiques*, volume 14, pages 473–500. 1995.

[10] David Navarre, Philippe A. Palanque, Rémi Bastide, and Ousmane Sy. Structuring interactive systems specifications for executability and prototypability. In *DSV-IS*, pages 97–119, 2000.

[11] Amélie Schyn, David Navarre, Philippe Palanque, and Luciana Porcher Nedel. Description formelle d'une technique d'interaction multimodale dans une application de réalité virtuelle immersive. In *IHM*, Caen, France, November 2003.

[12] Bruno d'Ausbourg. Contribution à la définition de systèmes de confiance et de systèmes utilisables. HDR, Avril 2001.

[13] Pierre Roché. *Modélisation et Vérification d'Interface Homme-Machine*. PhD thesis, ENSAE/CERT, Toulouse, France, 1998.

[14] Irina Sessitskaia. *Apport des techniques d'abstraction pour la vérification des interfaces Homme-Machine*. PhD thesis, ONERA, Toulouse, France, 2002.

[15] Yamine Aït Ameur, Mickaël Baron, and Patrick Girard. Formal validation of HCI user tasks. In *Software Engineering Research and Practice*, pages 732–738, 2003.

[16] Yamine Ait-Ameur and Mickael Baron. Bridging the gap between formal and experimental validation approaches in HCI systems design : use of the event B proof technique. In *ISOLA*, 2004.

[17] Yamine Ait-Ameur, Mickael Baron, and Nadjet Kamel. Encoding a process algebra using the event B method. Application to the validation of user interfaces. In *ISOLA*, 2005.

[18] Abowd G.D. *Formal Aspects of Human-Computer Interaction*. PhD thesis, Oxford University Computing Laboratory, Oxford, UK, 1991.

[19] Francis Jambon, Patrick Girard, and Yohann Boisdron. Dialogue validation from task analysis. In David J. Duke and Angel R. Puerta, editors, *DSV-IS*, pages 205–224. Springer, 1999.

[20] Richard K. Shehady and Daniel P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *FTCS '97 : Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, page 80, Washington, DC, USA, 1997. IEEE Computer Society.

[21] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Trans. Softw. Eng.*, 27(2) :144–155, 2001.

[22] Nicolas Zuanon. *Test de spécification de services de télécommunication*. PhD thesis, Université Joseph Fourrier, Grenoble, France, Juin 2000.

[23] Laurence Nigay and Joëlle Coutaz. A generic platform for addressing the multimodal challenge. In *CHI*, pages 98–105. ACM Press, 1995.

[24] Laurence Nigay and Joëlle Coutaz. A design space for multimodal systems : concurrent processing and data fusion. In A. Henderson E. Hollnagel T. White S. Ashlung, K. Mullet, editor, *INTERCHI*, pages 172–178, Amsterdam, avril 1993. ACM New York.

[25] Laurence Nigay and Joëlle Coutaz. *Intelligence and Multimodality in Multimedia Interfaces : Research and Applications*, chapter Nine : Multifeature Systems : The CARE Properties and Their Impact on Software Design. John Lee, AAAI Press.

[26] Joëlle Coutaz, Laurence Nigay, Daniel Salber, Ann Blandford, Jon May, and Richard M. Young. Four easy pieces for assessing the usability of multimodal interaction : the CARE properties. In *INTERACT*, pages 115–120. Chapman & Hall, 1995.

[27] Coutaz J. and Nigay L. Les propriétés "CARE" dans les interfaces multimodales. In *IHM'94*, pages 7–14, Lille, Décembre 1994.

[28] L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanon. Lutess : a testing environment for synchronous software. In *Tool Support for System Specification, Development and Verification*, pages 48–61. Advances in Computer Science. Springer Verlag, June 1998.

[29] Gregory D. Abowd, Joëlle Coutaz, and Laurence Nigay. Structuring the space of interactive system properties. In James A. Larson and Claus Unger, editors, *Engineering for Human-Computer Interaction*, volume A-18 of *IFIP Transactions*, pages 113–129. North-Holland, 1992.

[30] Melody Y. Ivory and Marti A Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.*, 33(4) :470–516, 2001.

[31] Joelle Coutaz. Modélisation en interaction homme-machine. Cours DEA Informatique Système Communication 2002-2003.

[32] Sandrine Balbo. *Evaluation ergonomique des interfaces utilisateur : Un pas vers l'automatisation.* PhD thesis, Université Joseph Fourrier, Grenoble, France, Septembre 1994.

[33] V. Normand. *Le modèle SIROCO : de la spécification conceptuelle des interfaces utilisateur à leur réalisation.* PhD thesis, Université Joseph Fourrier, Grenoble, France, 1992.

[34] A. Dittmar. More precise descriptions of temporal relations within task models. In *DS-VIS*, pages 151–168, 2000.

[35] Lim K.Y., Long J.B., and Silcock N. Instanciation of task analysis in a structured method for user interface design. In *the 11th Interdisciplinary Workshop on Informatics and Psychology : Task Analysis in Human-Computer Interaction*, Scharding, June 1992.

[36] Shepherd A. Analysis and training in information technology tasks. In D. Diaper, editor, *Task Analysis for Human-Computer Interaction*, pages 15–55, Chichester, USA : Ellis Horwood, 1989.

[37] Scapin D. and Pierret-Goldbreich C. Towards a method for task description : MAD. In L. Berlinguet and D. Berthelette, editors, *Proceedings of Work with Display Units (WWU '89)*, pages 27–34. North-Holland : Elsevier Science, 1989.

[38] H. Rex Hartson and Philip D. Gray. Temporal aspects of tasks in the user action notation. *Human-Computer Interaction*, 7(1) :1–45, 1992.

[39] J. Coutaz, G. P. Faconti, F. Paternò, L. Nigay, and D. Salber. A Comparison of Approaches for Specifying Multi-Modal Interactive Systems. In *Proceedings of the ERCIM Workshop on Multimodal Human-Computer Interaction*, pages 165–174, November 1993.

[40] Giulio Mori, Fabio Paternò, and Carmen Santoro. Ctte : Support for developing and analyzing task models for interactive system design. *IEEE Trans. Software Eng.*, 28(8) :797–813, 2002.

[41] Paternò F., Ballardin G., Conte E., Mancini C., and Mori G. Specification of notation of task modelling methodology. Technical report, CNUCE-C.N.R., Pisa, Italy, January 2000.

[42] Andreas Lecerof and Fabio Paternò. Automatic support for usability evaluation. *IEEE Trans. Software Eng.*, 24(10) :863–888, 1998.

[43] Fabio Paternò. A formal approach to the evaluation of interactive systems. *SIGCHI Bull.*, 26(2) :69–73, 1994.

[44] Fabio Paternò and Carmen Santoro. Integrating model checking and HCI tools to help designers verify user interface properties. In *DSV-IS*, pages 135–150, 2000.

[45] Francis Jambon, Patrick Girard, and Yamine Aït Ameur. Interactive system safety and usability enforced with the development process. In *EHCI*, pages 39–56, 2001.

[46] Francis Jambon. From formal specifications to secure implementations. In *CADUI*, pages 51–62, 2002.

[47] J.-R. Abrial. *The B-book : assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[48] Patrick Girard, Mickaël Baron, and Francis Jambon. Integrating formal approaches in human-computer interaction,. In *INTERACT 2003, workshop "Closing the Gaps : Software Engineering and HCI".*

[49] David J. Duke and Michael D. Harrison. Abstract interaction objects. *Comput. Graph. Forum*, 12(3) :25–36, 1993.

[50] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.

[51] Bruno d'Ausbourg. Using model checking for the automatic validation of user interface systems. In *DSV-IS*, pages 242–260, 1998.

[52] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Software Eng.*, 18(9) :785–793, 1992.

[53] Harold Thimbleby, Paul Cairns, and Matt Jones. Usability analysis with markov models. *ACM Trans. Comput.-Hum. Interact.*, 8(2) :99–132, 2001.

[54] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6) :591–603, 1991.

[55] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre : A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[56] Daniel Pilaud and Nicolas Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In Mathai Joseph, editor, *FTRTFT*, volume 331 of *Lecture Notes in Computer Science*, pages 99–110. Springer, 1988.

[57] C. Ratel. *Définition et réalisation d'un outil de vérification formelle de programmes Lustre : Le système Lesar*. PhD thesis, Université Joseph Fourrier, Grenoble, France, Juin 1992.

[58] Ioannis Parissis. *Test de logiciel synchrones spécifiés en Lustre*. PhD thesis, Université Joseph Fourrier, Grenoble, France, Septembre 1996.

[59] Lydie Du Bousquet. *Test Fonctionnel Statistique de Systèmes Spécifiés en Lustre*. PhD thesis, Université Joseph Fourrier, Grenoble, France, Septembre 1999.

[60] F. Ouabdesselam and I. Parissis. Constructing operational profiles for synchronous critical software. In *6th International Symposium on Software Reliability Engineering*, pages 286–293, Toulouse, France, october 1995.

[61] L. du Bousquet, F. Ouabdesselam, and J.-L. Richier. Expressing and implementing operational profiles for reactive software validation. In *9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, november 1998.

[62] Ioannis Parissis and Jérôme Vassy. Thoroughness of specification-based testing of synchronous programs. In *ISSRE*, pages 191–202. IEEE Computer Society, 2003.

[63] I. Parissis and J. Vassy. Strategies for automated specification-based testing of synchronous software. In *16th International Conference on Automated Software Engineering*, San Diego, USA, November 2001. IEEE.

[64] Jérôme Vassy. *Génération automatique de cas de test guidée par les propriétés de sûreté*. PhD thesis, Université Joseph Fourrier, Grenoble, France, Octobre 2004.

[65] J. Bouchet and L. Nigay. ICARE : a component-based approach for the design and development of multimodal interfaces. In *CHI Extended Abstracts*, pages 1325–1328, 2004.

[66] Nicolas Halbwachs and Siwar Baghdadi. Synchronous modelling of asynchronous systems. In *EMSOFT*, pages 240–251, 2002.

[67] Albert Benveniste, Benoît Caillaud, and Paul Le Guernic. From synchrony to asynchrony. In *CONCUR*, pages 162–177, 1999.

[68] Laya Madani, Catherine Oriat, Ioannis Parissis, Jullien Bouchet, and Laurence Nigay. Synchronous testing of multimodal systems : An operational profile-based approach. In *ISSRE*, pages 325–334. IEEE Computer Society, 2005.

[69] J. Bouchet, L. Madani, L. Nigay, C. Oriat, and I. Parissis. Formal testing of multimodal interactive systems. In *EIS'2007 Engineering Interactive Systems*, Salamanca, Spain, March 2007.

[70] Manuel Núñez and David de Frutos-Escrig. Testing semantics for probabilistic lotos. In Gregor von Bochmann, Rachida Dssouli, and Omar Rafiq, editors, *FORTE*, volume 43 of *IFIP Conference Proceedings*, pages 367–382. Chapman & Hall, 1995.

[71] Joost-Pieter Katoen, Rom Langerak, and Diego Latella. Modeling systems by probabilistic process algebra : an event structures approach. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *FORTE*, volume C-22 of *IFIP Transactions*, pages 253–268. North-Holland, 1993.

[72] Laurence Nigay and Joëlle Coutaz. *Espaces conceptuels pour l'interaction multimédia et multimodale*, volume 15, N 9 of *TSI, numéro spécial Multimédia et Collecticiel*, pages 1195–1225. AFCET and Hermes, 1996.

[73] Laurence Nigay, Francis Jambon, and Joëlle Coutaz. Formal specification of multimodality. In *CHI'95 Workshop on Formal Specification of User Interfaces*, Denver, May 1995.

[74] Philippe A. Palanque and Amélie Schyn. A model-based approach for engineering multimodal interactive systems. In Matthias Rauterberg, Marino Menozzi, and Janet Wesson, editors, *INTERACT*. IOS Press, 2003.

[75] Yamine Aït Ameur and Nadjet Kamel. A generic formal specification of fusion of modalities in a multimodal HCI. In René Jacquart, editor, *IFIP Congress Topical Sessions*, pages 415–420. Kluwer, 2004.

[76] Nadjet Kamel. Modélisation et vérification formelle des IHM multimodales. In *RJC-IHM 2004. Rencontres Jeunes chercheurs en Interaction Homme-Machine. AFIHM.*

[77] Nadjet Kamel and Yamine Ait-Ameur. Modèle formel général pour le traitement d'interactions multimodales. In *IHM*, pages 219–222, Namur, Belgique, 2004. ACM.

[78] Nadjet Kamel and Yamine Ait-Ameur. *Mise en oeuvre d'IHM multimodales dans un système de CAO. Une approche fondée sur les méthodes formelles*, volume 1/3, pages 235–256. 2005.

[79] Bellik Y. *Interfaces Multimodales : concepts, modèles et architecture.* PhD thesis, LIMSI- Université d'Orsay, 1995.

[80] Yamine Ait-Ameur, Idir Ait-Sadoune, and Mickael Baron. Étude et comparaison de scénarios de développements formels d'interfaces multi-modales fondés sur la preuve et le raffinement. In *MOSIM - 6ème Conférence Francophone de Modélisation et Simulation. Modélisation, Optimisation et Simulation des Systèmes Défis et Opportunités*, Rabat, Maroc, Avril 2006.

[81] Yamine Ait-Ameur, Idir Ait-Sadoune, Jean-Marc Mota, and Mickael Baron. Validation et vérification formelles de systèmes interactifs multi-modaux fondées sur la preuve. In *IHM*, pages 123–130, Montreal, Canada, 2006. ACM.

[82] Besnik Seljimi and Ioannis Parissis. Using CLP to automatically generate test sequences for synchronous programs with numeric inputs and outputs. In *ISSRE*, pages 105–116. IEEE Computer Society, 2006.

[83] Besnik Seljimi and Ioannis Parissis. Test de logiciel synchrones : apports de la programmation par contraintes. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, pages 143–160, Namur, Belgium, June 2007.

[84] Laya Madani, Catherine Oriat, and Ioannis Parissis. « expression de propriétés ergonomiques et fonctionnelles dans le formalisme synchrone » et « guide de conception de propriétés de services multimodaux. Technical report, Fourniture du projet RNRT VERBATIM, Décembre 2005.

[85] Jeremy Sproston. probabilistic Timed Systems. In *the 6th school on MOdeling and VErifying parallel Processes (MOVEP'04)*, 2004.

Utilisation de la programmation synchrone pour la spécification et la validation de services interactifs

Résumé

Ce travail porte sur le test automatique de systèmes interactifs. L'approche proposée est basée sur des techniques de test de systèmes réactifs *synchrones*. Le comportement de systèmes synchrones, qui est constitué de cycles commençant par la lecture d'une entrée et finissant par l'émission d'une sortie, est sous certaines conditions, similaire à celui de systèmes interactifs. En particulier, nous étudions l'utilisation de Lutess, un environnement de test synchrone, pour valider les systèmes interactifs.

Nous montrons l'intérêt d'utiliser les techniques de test proposées par Lutess afin de générer des scénarios intéressants et nous illustrons leur utilisation sur une étude de cas (une application de réalité virtuelle mobile).

Nous avons également étudié la génération de données de test à partir d'arbres de tâches, qui peuvent être enrichis d'une spécification de profils opérationnels.

L'adaptation des techniques de test synchrone à la validation d'applications interactives multimodales est également étudiée, notamment en prenant en compte certaines propriétés concernant la multimodalité.

Using synchronous programming for specification and validation of interactive services

Abstract

This work deals with the automatic testing of interactive systems. The proposed approach is based on testing techniques for synchronous reactive software. The behaviour of synchronous systems, consisting of cycles starting by reading an input and ending by issuing an output, is to a certain extent similar to the one of interactive systems. In particular, we are using Lutess, a synchronous testing environment, to validate interactive systems.

We show the interest of applying the testing techniques proposed by Lutess in order to generate interesting scenarios and we illustrate their use on a case study (a mobile virtual reality application).

We also study the generation of test data from task trees, which can be enriched by operational profiles specification.

The adaptation of synchronous testing techniques for the validation of multimodal interactive systems is also studied, taking into account multimodality-related properties.