MASTER EN SCIENCES INFORMATIQUES

Description de l'interaction multimodale :extension du langage SMUIML

Tallier, Max

Award date:
2017

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2016 – 2017

Description de l'interaction multimodale : extension du langage SMUIML

Tallier Max

Promoteur : Dumas Bruno
(Signature pour approbation du dépôt - REE art. 40)

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

1 Résumé

Au cours des années, le potentiel de création de liens de communication entre l'homme et la machine n'a fait qu'évoluer. Nous nous trouvons face à une quantité relativement importante d'outils permettant la communication entre ces deux acteurs. Pour pouvoir organiser et faciliter les échanges, les langages de modélisation fournissent de puissants outils permettant la création de communication par interfaces multimodales.

Il existe de nombreux langages de description, chacun fonctionnant de manière différente et proposant des options de modélisation spécifiques.
Ce travail se concentrera sur un langage en particulier :
SMUIML (Synchronized User Interface Modelling Language) est un langage de modélisation développé par Bruno Dumas, professeur à la faculté d'informatique à l'Université de Namur.
La représentation de la modélisation d'une interaction avec l'ordinateur se fait sous la forme d'une machine à états. Cet automate dispose d'une traduction en XML.
Ce langage, basé sur une architecture de type CARE [8], est principalement orienté vers la gestion des modalités en entrée et permet au système de les gérer soit en parallèle l'une de l'autre, soit de manière séquentielle. Malheureusement, son développement a été mis en suspens et n'a pas pu bénéficier de nouvelles évolutions depuis quelques années.

Profitant du travail lié à ce mémoire, B. Dumas souhaite redémarrer le développement de son langage et lui apporter des modifications structurelles pour pouvoir, dans un futur plus ou moins proche, l'utiliser dans le cadre de son travail académique.
De plus, une boite à outils graphique, HephaisTK [6], a été créée afin de venir en aide à l'utilisateur souhaitant réaliser des interactions avec l'ordinateur. De même que son langage de référence, son développement est suspendu depuis plusieurs années.

L'objet de ce mémoire est d'une part de répertorier et d'expliquer les modifications réalisées sur SMUIML [5] et d'autre part d'ouvrir des pistes de réflexion sur les modifications à venir sur ce langage.

2 Abstract

Over the years, the potential for creating communication links between man and machine only evolved continually. We face a large number of communication tools between these two actors. In order to organize and facilitate exchanges, modelling languages provide powerful tools to create communication through multimodal interface.

There are many descriptives languages, each operating in different ways and with specific modelling options.
This work will focus on one of those languages :
The SMUIML (Synchronized User Interface Modelling Language) is a modelling language developed by Bruno Dumas, professor at the Faculty of Computer Sciences at the University of Namur.
The modelling representation of the multimodal interaction is done in form of a state machine and this controller features an XML translator.
This language is based on a CARE architecture [8] and is mainly oriented towards the input modalities management and allows a parallel management or sequential management. Unfortunately, its development has been put on hold and did not benefit from new developments in recent years.

Taking advantage of the work associated with this dissertation, B. Dumas wishes to restart the development of this language and make structural changes in order to be able to use it in the framework of his academic work, in a quite close future.
In addition, a graphical toolkit, HephaisTK [6], was created to help users who want to create some interactions with the computer. Like its reference language, its development has been put on hold.

The purpose of this work is to list and explain the modifications made on SMUIML [5] and on the other hand to open up some lines of discussion for the future modifications of this language.

3 Remerciements

Je tiens à remercier toutes les personnes qui ont permis la réalisation de ce mémoire.

Je remercie plus particulièrement Bruno Dumas, mon promoteur depuis deux ans, pour sa patience et sa pédagogie dans l'aboutissement ultime de ce travail.

Je remercie aussi ma compagne qui m'a soutenu durant ces quelques années d'étude.

Je remercie également mes testeurs pour leurs évaluations objectives et pertinentes.

Et finalement, je remercie mes relecteurs pour leur disponibilité et leur indulgence.

TABLE DES MATIÈRES

TABLE DES FIGURES

4 Glossaire

CARE : Complementarity, Assignement, Redundancy, Equivalence ;

CASE : Concurrent, Alternate, Synergistic, Exclusive ;

MMMM : Multimodal Man Machine Model ;
(anciennement appelé MUSE : Multimodal User System Equivalence ;)

MVC : Modèle- Vue- Contrôleur ;

OS : Operating System ;

SMUIML : Synchronized User Interface Modelling Language ;

TTN : Time To Next ;

TW : Time Within ;

UIDL : User Interface Definition Language ;

Wireframe : Maquette fonctionnelle ;

XML : Extensible Marckup Language ;

CHAPITRE

1

INTRODUCTION

1 Contexte

Depuis Richard Bolt et son exemple du Put-That-There [9], les interactions multimodales n'ont fait que se développer, s'adaptant aux nouveautés technologiques de ces dernières années.
De nombreux langages de modélisation ont été créés dans le but d'améliorer la communication entre l'homme et la machine.
Bruno Dumas, professeur à l'UNamur, a développé le sien, SMUIML [5].
Ce langage a été créé en 2008 et a été développé durant plusieurs années.
Malheureusement celui-ci a dû être mis entre parenthèses pour diverses raisons et par conséquent a connu peu d'évolutions ces dernières années.
Ce langage possède déjà une base structurelle solide composée de plusieurs éléments correspondant à son architecture actuelle, le modèle CARE [8].
Afin d'aider à la création d'interactions, un « toolkit » fut développé, HephaïsTK [6], mais son développement a été mis en veille également.

Dans ces conditions, peut-on intégrer de nouvelles évolutions techniques à ce langage ? Permettent-elles d'améliorer le langage ? Peut-on réfléchir à une prise en charge contextuelle de l'interaction, c'est-à-dire modifier l'interaction suivant des événements ponctuels propres à l'environnement ambiant de l'utilisateur ?
Si les modifications sont concluantes, HephaisTK, conviendra-t-il toujours à SMUIML ou devra-t-il également subir des modifications ? Nous tâcherons de répondre à ces questions tout au long de ce travail.

2 Motivation

Ce travail est effectué dans le cadre du mémoire pour l'obtention du master 60 en Sciences Informatiques.
Il est la poursuite de l'état de l'art effectué dans le cadre du projet personnel de troisième baccalauréat en Sciences Informatiques [4] à l'Université de Namur.
Cet état de l'art avait pour but d'explorer le monde des langages de modélisation d'interaction multimodale et de faire une étude comparative des différentes analyses disponibles sur ce sujet.
Ce travail est réalisé dans le but d'améliorer SMUIML à l'aide de certaines innovations structurelles.

3 Méthodologie

Afin de réaliser ce travail, nous avons utilisé la méthode de conception de système vue lors du cours d'interaction Homme-Machine [1].

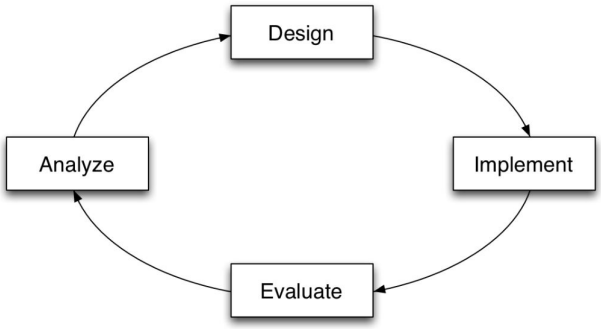


FIGURE 1.1: ≪ UX Process Wheel ≫ ou cycle de vie de l'expérience utilisateur [1].

Dans un premier temps, nous commencerons par la phase d'analyse en réalisant un recueil de critiques et de besoins émis par divers chercheurs ou utilisateurs.
Ces demandes vont permettre d'orienter les modifications qu'il faudra apporter au langage SMUIML.
Dans un deuxième temps, nous aborderons la phase de design. Elle consistera à prendre en compte plusieurs remarques faites à l'encontre du langage afin de penser un nouveau système de référence.
Par faute de temps, la phase d'implémentation a été remplacée par un prototypage graphique sous la forme d'un wireframe cliquable.
Nous terminerons par la phase d'évaluation, où le prototype sera testé par un échantillon choisi d'utilisateurs.

4 Structure du document

Avant d'entamer le début de l'analyse présentée dans le chapitre précédent, nous ferons un rappel théorique sur la multimodalité et ses langages de modélisation.
Le chapitre 2 aborde la question de la multimodalité, ce qu'implique cette notion et comment nous pouvons l'aborder dans le sujet qui nous intéresse.
Le chapitre 3 se penche sur le concept des langages de modélisation. Quels sont les différents types de langages que nous pouvons rencontrer ? Sont-ils équivalents entre eux ?
Le chapitre 4 présente le langage de modélisation SMUIML ("Synchronized Multimodal User Interfaces Modelling Language") de B. Dumas, l'instigateur de ce travail. Nous découvrirons l'architecture existante de ce langage, ainsi que ses qualités et ses défauts. Nous présenterons brièvement HephaisTK, l'éditeur graphique (UIDL) de SMUIML.
Le chapitre 5 présente l'évolution que nous avons fait parcourir au langage SMUIML et illustre une nouvelle implémentation d'un éditeur graphique.
Et dans le dernier chapitre, nous donnerons nos conclusions et nous introduirons les futures tâches à réaliser pour assurer la continuité de ce projet.

CHAPITRE

2

DESCRIPTION DES INTERACTIONS MULTIMODALES

Dans cette partie du document, nous allons décrire en quoi consiste une interaction utilisant la multimodalité, ainsi que les différentes architectures que nous pouvons retrouver derrière celle-ci.

1 Multimodalité

Avec les évolutions technologiques des dernières années, les interactions homme/ machine dites WIMPs (Windows, Icons, Menus, Pointing) laissent plutôt la place à une nouvelle façon de communiquer avec la machine.
Cette manière d'interagir passe nécessairement par la multimodalité. L'utilisateur doit transmettre ses informations par un ou plusieurs canaux de communication, incluant la voix, le geste, l'écriture,etc.
Le but de cette démarche est de créer une communication homme-machine plus naturelle, se rapprochant du dialogue entre hommes.
Afin d'arriver à ce genre d'interactions le chemin est long et de nombreuses ressources doivent être utilisées.
Dans la figure 2.1 [10] est modélisé graphiquement un exemple de communication homme-machine. Nous remarquons que celle-ci est divisée en deux parties, une concernant l'humain, en haut, et l'autre la machine, en bas.
Chaque partie est également divisée en quatre sections symétriques.
Elles représentent une étape différente dans la communication établie entre l'homme et la machine.

Quand l'homme veut transmettre une information à la machine, il doit passer par différentes étapes :

- L'état décisionnel : il s'agit de la préparation consciente ou inconsciente de l'information par l'utilisateur.
- L'état d'action : l'information est transmise à l'ordinateur par le biais de la parole, de gestes ou d'expressions faciales.
- L'état de perception : l'information est perçue par l'ordinateur via les différents canaux d'entrée.
- L'état d'interprétation : l'ordinateur donne une signification aux différents stimuli perçus. Ici se situe le mécanisme de fusion.

L'ordinateur répond à l'homme par les mêmes étapes :

- L'état computationnel : une action est prise par l'ordinateur suivant la logique et les règles de dialogue données par le développeur. Ici se situe le mécanisme de fission.
- L'état d'action : une réponse est générée et émise par l'ordinateur vers l'utilisateur.
- L'état de perception : la réponse est reçue par l'utilisateur.
- L'état d'interprétation : l'utilisateur donne une signification à la réponse reçue de l'ordinateur.

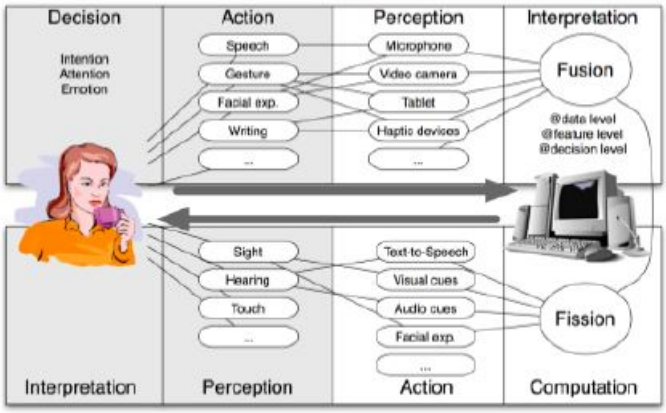


FIGURE 2.1: Représentation de la boucle de communication multimodale [10].

Nous partons du postulat que la partie gauche du schéma 2.1 pense par elle même et n'a pas besoin d'une implémentation d'intelligence spécifique.
Toute l'intelligence machine doit se trouver dans la partie droite du schéma 2.1 afin que l'ordinateur comprenne au mieux le ou les messages que l'utilisateur veut lui faire passer mais également qu'il puisse fournir la réponse la plus adéquate possible.

2 Architecture et composantes clés

Le cœur de l'interaction vu dans la section précédente se situe au niveau de l'état computationnel.

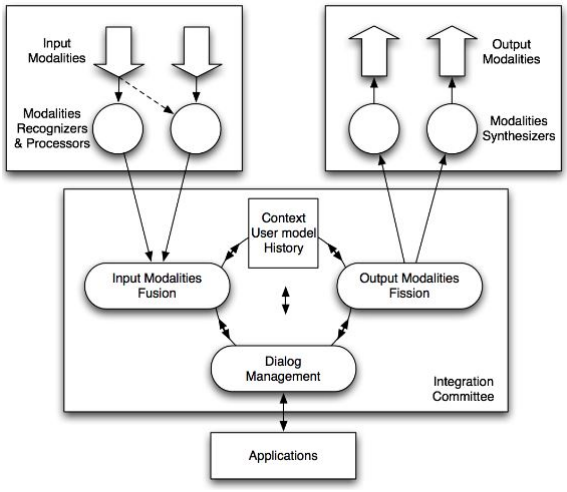


FIGURE 2.2: Architecture d'un système multimodal et son comité d'intégration [10].

Dans la figure 2.2 [10], nous avons la représentation graphique de l'état computationel composée de trois parties distinctes, les modalités d'entrée, le comité d'intégration et les modalités de sortie.
Le comité d'intégration a pour but d'interpréter et de répondre au message envoyé par l'utilisateur. Il est composé de quatre éléments, le moteur de fusion, le module de fission, le gestionnaire de dialogue et le gestionnaire de contexte.
Dans un premier temps, les signaux envoyés par l'utilisateur via les modalités d'entrée vont être perçus par le système. Ceux-ci sont transmis au moteur de fusion qui a pour tâche de donner une interprétation à ces signaux. L'interprétation est ensuite envoyée au gestionnaire de dialogue qui détermine le degré de dialogue, les transitions et les actions à effectuer. Le gestionnaire transmet ses conclusions au module de fission qui retourne la réponse à l'utilisateur en utilisant la modalité la plus adéquate.
Le gestionnaire de contexte, quant à lui, informe les trois autres composants de tout changement de contexte afin d'adapter l'interprétation du message ou la réponse.
La fusion et la fission font partie des points clés de l'intégration multimodale. Sans un organe d'interprétation (fusion) et un organe de communication (fission), il serait quasiment impossible de créer ce type de communication avec l'ordinateur.
Ce mécanisme de fission doit fournir une réponse cohérente et contextuelle aux informations que l'utilisateur a envoyé lors de la communication. Pour ce faire elle est divisée en trois étapes :

- La construction du message.
- La sélection du canal de sortie.
- La construction d'un résultat synchronisé et cohérent.

Dans ce travail nous nous concentrerons davantage sur le mécanisme de fusion.

2.1 Fusion

Le mécanisme de fusion dispose de plusieurs architectures proposant de multiples alternatives quant à la prise en charge de la multimodalité.
Nous allons détailler ces différentes architectures.

le modèle CASE, présenté par J. Coutaz et L. Nigay en 1993 [2]

Le modèle CASE fut le premier modèle à avoir été présenté comme architecture pour la fusion multimodale.
Il a la particularité de se concentrer principalement sur la gestion et la combinaison des modalités entre elles au niveau du moteur de fusion, donc dans la partie intelligente du mécanisme qui régit la communication.
L'acronyme CASE signifie Concurrent, Alternate, Synergistic et Exclusive.
Il décrit la manière dont les commandes peuvent être interprétées par un système combinant les quatre propriétés de base pour la fusion des modalités en entrée.

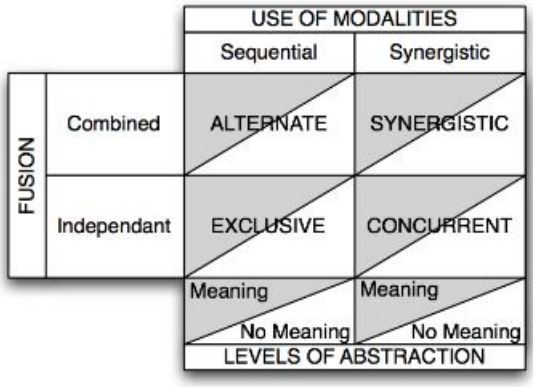


FIGURE 2.3: Modèle CASE [2].

La figure 2.3 montre, dans sa partie supérieure, la temporalité d'utilisation des modalités (Sequential ou Synergistic) et sur sa partie gauche, la possibilité de combinaisons des modalités (Combined ou Independent).
Les créateurs de ce modèle [2] définissent aussi deux niveaux d'abstraction différents pour les combinaisons.
Grâce à ces caractéristiques, les quatre propriétés de ce modèle vont apparaître.

Concurrent : les modalités sont utilisées simultanément mais sans se référencer l'une l'autre. Chaque modalité envoie un message.

Alternate : les modalités sont utilisées séquentiellement et de manière combinée. Les modalités sont reconnues séparément et la composition de ces modalités permet de composer un message.

Synergistic : les modalités sont utilisées de manière parallèle et combinée. Plusieurs modalités sont requises pour un message.

Exclusive : les modalités sont indépendantes et reconnues de manière séquentielle. Une modalité correspond à un message et un message est passé à la fois.

Le modèle CARE présenté par J. Coutaz et L. Nigay en 1995 [8]

Un deuxième modèle parallèle au modèle CASE, est l'architecture CARE.
Il est plus récent et a la particularité de se concentrer principalement sur les possibilités de combinaisons de modalités au niveau de l'utilisateur.
Le nom CARE correspond à Complementarity, Assignement, Redundancy et Equivalent.

Complementarity : les modalités complémentaires sont utilisées pour faire comprendre la signification d'un message. Nous avons besoin de minimum deux modalités pour que la machine comprenne le message.

Assignment : indique qu'une seule modalité suffit pour permettre la compréhension.

Redundancy : est utilisé quand plusieurs modalités différentes sont utilisées pour faire passer le même message.

Equivalence : intervient lorsqu'une modalité substitue plusieurs modalités pour faire passer une signification. Plusieurs modalités envoient le même message mais il n'en suffit que d'une pour l'envoyer.

Ces propriétés sont davantage orientées vers le lien entre l'utilisateur et la machine, en se concentrant sur les possibilités dont l'utilisateur dispose dans le choix de combinaison des modalités.

TYCOON présenté par J-C Martin en 1998 [11]

Le modèle Tycoon est une extension du modèle CARE qui le complète par quelques éléments supplémentaires permettant des interactions plus rapides.
Par rapport à son modèle de référence, TYCOON garde sans les modifier trois caractéristiques : Complementarity, Equivalence et Redundancy.
La caractéristique d'Assignment est modifiée et deux nouvelles caractéristiques sont ajoutées au modèle.

Specialization : est utilisé lorsqu'un type spécifique d'information est toujours traité par la même modalité. Cette caractéristique tient compte du lien entre l'événement et la modalité.

Transfer : permet de compléter des informations manquantes ou mal reconnues venant d'une modalité grâce à sa répétition ou l'envoi d'une autre modalité. Il s'agit d'une coopération de plusieurs modalités.
Comme exemple, nous pouvons citer une phrase prononcée par l'utilisateur mais mal comprise par la machine. Cette phrase pourra être complétée grâce à un clavier ou en reprononçant les parties de phrases mal reconnues mais sans devoir réintroduire la phrase entière.

Concurrency : permet l'utilisation de plusieurs modalités en même temps. Chaque modalité donne simultanément une information différente et ces informations n'ont pas ou peu de lien entre elles.

MMMM [3]

Il s'agit du modèle MMMM pour Multimodal Man Machine Model, appelé anciennement MUSE (Multimodal User Systme Equivalence).

La caractéristique principale de ce modèle est que, contrairement aux autres qui n'envisagent le mécanisme de fusion qu'en un bloc, lui le divise en deux parties distinctes. Il fait la différence entre une partie utilisateur (User-Side) et une partie machine (System-Side).

Cette architecture a été élaborée au sein de l'Université de Namur au cours de l'année 2016.

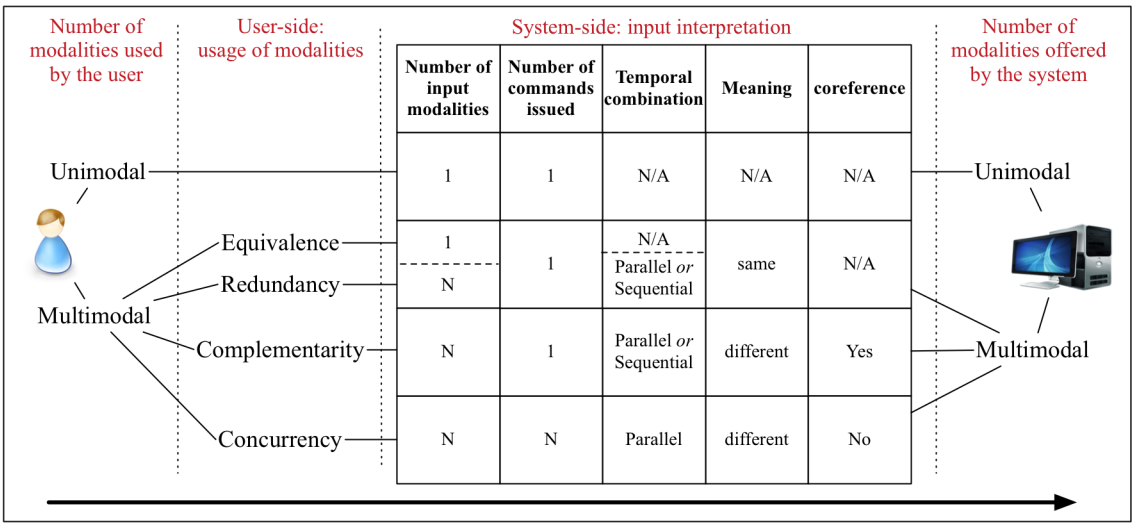


FIGURE 2.4: Structure de l'architecture MMMM [3].

Dans la figure 2.4 [3], il y a une nette séparation entre les deux acteurs de la communication. La partie utilisateur commence par le choix du nombre de modalités utilisées, une seule ou plusieurs.
De ce choix va dépendre la suite de l'interaction. L'utilisateur peut combiner ou non ses canaux de communication. S'il n'utilise qu'une seule modalité, son choix est restreint, mais s'il utilise plusieurs modalités, il est alors confronté à quatre combinaisons possibles.
Nous retrouvons les combinaisons déjà rencontrées lors de la présentation des autres modèles : Equivalence, Redundancy, Complementarity et Concurrency.
Nous entrons ensuite dans la partie système où l'ordinateur va pouvoir, sur base de son tableau de combinaison, interpréter les messages envoyés par l'utilisateur.
Cette architecture se veut plus accessible au développement et à la création d'interaction. Au final, le développeur ne doit se préoccuper que de la gestion de ses modalités, le reste se faisant automatiquement par le côté système. En théorie, l'utilisateur ne devrait donc pas détenir de compétences avancées en programmation pour la création d'interfaces.
Grâce à ce découpage des fonctionnalités, nous obtenons davantage de maniabilité dans la manipulation des différents éléments et de facilité dans leur lecture.

CHAPITRE

3

LANGAGES DE MODÉLISATION POUR L'INTERACTION MULTIMODALE

1 Guidelines

Pour permettre la création des systèmes multimodaux, il existe moult langages de modélisation différents.
Ces langages disposent de caractéristiques différentes et varient quant à leurs niveau de conception et de finition.
Afin de permettre une certaine homogénéité des caractéristiques des langages, Sire et Chatty [12] ont écrit une série de lignes de conduite permettant de juger de la qualité d'un langage de modélisation :

1. **Niveaux d'abstraction**
   Un langage de modélisation doit pouvoir fournir différents niveaux d'abstraction à la modélisation.
   Ceci dans le but de pouvoir séparer différents mécanismes intrinsèques aux interactions multimodales.

2. **Modélisation du dialogue humain-machine**
   Nous pouvons utiliser de nombreux outils afin de modéliser le dialogue humain-machine, comme les machines à états, l'approche impérative, des structures de contrôles,... .

3. **Adaptabilité au contexte et à l'utilisateur**
   L'adaptabilité du langage concerne essentiellement les entrées et les sorties.
   En entrée, le langage doit être capable d'utiliser les informations contextuelles et de l'utilisateur afin de pouvoir appliquer correctement les processus de reconnaissance des entrées et de fusion.
   En sortie, le choix de la modalité de transfert de la réponse et la coordination entre les différents messages dépendent également de l'utilisateur et du contexte.

4. **Contrôle sur le mécanisme de fusion**
   Vu la complexité des algorithmes de fusion, un contrôle de ceux-ci et de leurs paramètres est indispensable afin de garantir un résultat fiable.

5. **Contrôle de la synchronisation des événements**
   Avoir la main mise sur ce système permet de pouvoir gérer au mieux l'afflux des événements et ainsi éviter des collisions dans le traitement de ceux-ci.

6. **Gestion des erreurs**
   Les systèmes multimodaux sont soumis à une forte probabilité d'erreurs, venant de sources multiples.
   Un langage de description doit pouvoir fournir un système de gestion des erreurs afin de d'y pallier.

7. **Gestion des événements**
   La description des événements doit être prise en considération dans le langage et doit permettre à l'utilisateur d'avoir une vue d'ensemble sur le fonctionnement de l'application.

8. **Représentation des entrées et de sorties**
   La représentation des sources et des modalités d'entrées et de sorties permet aux créateurs d'interfaces multimodales d'avoir un certain contrôle sur les modalités utilisées et de pouvoir en gérer les paramètres.

9. **Trouver le bon équilibre entre convivialité et expressivité**
   Les langages doivent trouver une place entre l'humain et la machine, chose peu aisée et d'une importance capitale.

2 Familles de langages

En dehors de ces guidelines, nous retrouvons plusieurs familles de langages. De manière générale, nous allons les regrouper suivant leurs bases de conception définissant la manière la façon dont le langage est conçu :

- les langages par contraintes : ces langages se basent sur de la programmation logique par résolution de contraintes. Ces langages sont peu nombreux (deux à l'écriture de ce mémoire) et leurs fonctionnalités sont limitées. Par exemple, nous pouvons retrouver ce type de système dans la planification de trajet avec le langage Coral [13] : l'utilisateur indique le lieu de son départ, le lieu d'arrivée et peut donner des indication sur le chemin qu'il souhaite suivre. Après avoir reçu les consignes, l'ordinateur trouve le chemin idéal grâce à la résolution des contraintes données, puis le système transmet le parcours à l'utilisateur.

- les langages basés sur du code : ces langages sont écrits grâce à un langage à balises (généralement XML). Celles-ci s'imbriquent de manière récursive et permettent de décrire des structures de données.
La construction de langages par cette méthode permet l'adaptation rapide des différents lexiques contenant les opérations de base de l'interaction. Ils sont également appropriés pour la description des différents liens entre les éléments qui composent l'interaction.

- les langages basés sur l'analyse de flux : ces langages sont basés sur l'évolution de certaines variables au cours du temps par des passages successifs dans une application.
Une des limitations de ces langages est le manque de description de l'état du système au cours de l'évolution de ces variables.

- les langages basés sur des machines à états : comme leur nom l'indique, ces langages utilisent une machine à états finis pour décrire l'interaction.
L'avantage de cette façon de concevoir l'interaction est que nous possédons un outil de description globale de l'interaction et que celui-ci peut facilement se coupler avec une autre notation comme, par exemple, une notation XML.

- les langages basés sur les réseaux de Petri : le comportement de ce type de langage est dicté par un ensemble de variables, par la valeur des différents nœuds et l'ensemble des transitions qui composent ce réseau.
Dans ce système, nous bénéficions d'une grande flexibilité dans la description des différents comportements concurrents que nous retrouvons dans un système multimodal (parallélisme, synchronisation...). Ce système présente un autre avantage : les propriétés mêmes des réseaux de Pétri permettent de modifier dynamiquement la structure du réseau et par ce fait de la garder viable.

3 Synthèse de l'état de l'art

Lors du projet personnel [4] effectué pendant la troisième année de baccalauréat, travail qui est à la base de ce mémoire, nous avons réalisé un état de l'art sur les langages de modélisation. Il effectue un recoupement détaillé de différents tableaux analytiques comparant des langages de modélisation ainsi que leurs frameworks.
Le but du projet était d'y déceler des similitudes ou des dissensions sur les points de comparaison.

L'analyse s'est déroulée sur quatre tableaux différents.
Le premier vient d'un article de B. Dumas, D. Lalane et R. Ingold [5], l'analyse des langages de ce tableau se fait sous le regard des guidelines cités dans ce chapitre. L'article est en rapport direct avec ces lignes de conduite et par la même occasion, il décrit le langage SMUIML.

Le deuxième est lié à une analyse du langage ICO venant de l'article de D. Navarre et al. [14]. Les critères d'analyse dépendent essentiellement des qualités d'expression de chaque langage ou framework, mais également sur des qualités de couverture d'interactions.

Le troisième tableau utilisé vient de la thèse de A. Hamon [15]. L'objet de la thèse étant l'application des interactions multimodales dans des cockpits d'avions, l'analyse est plus attentive à des notions de reconnaissance dans le temps et l'espace. Mais des caractéristiques générales liés à la multimodalité s'y retrouve également.

Pour finir le quatrième tableau vient de l'article de N.Elouali et al. [16], l'article traite des interactions multimodales au niveau des applications mobiles.

Ces tableaux sont disponibles dans les annexes.

De cette analyse en est ressorti la figure 3.1.

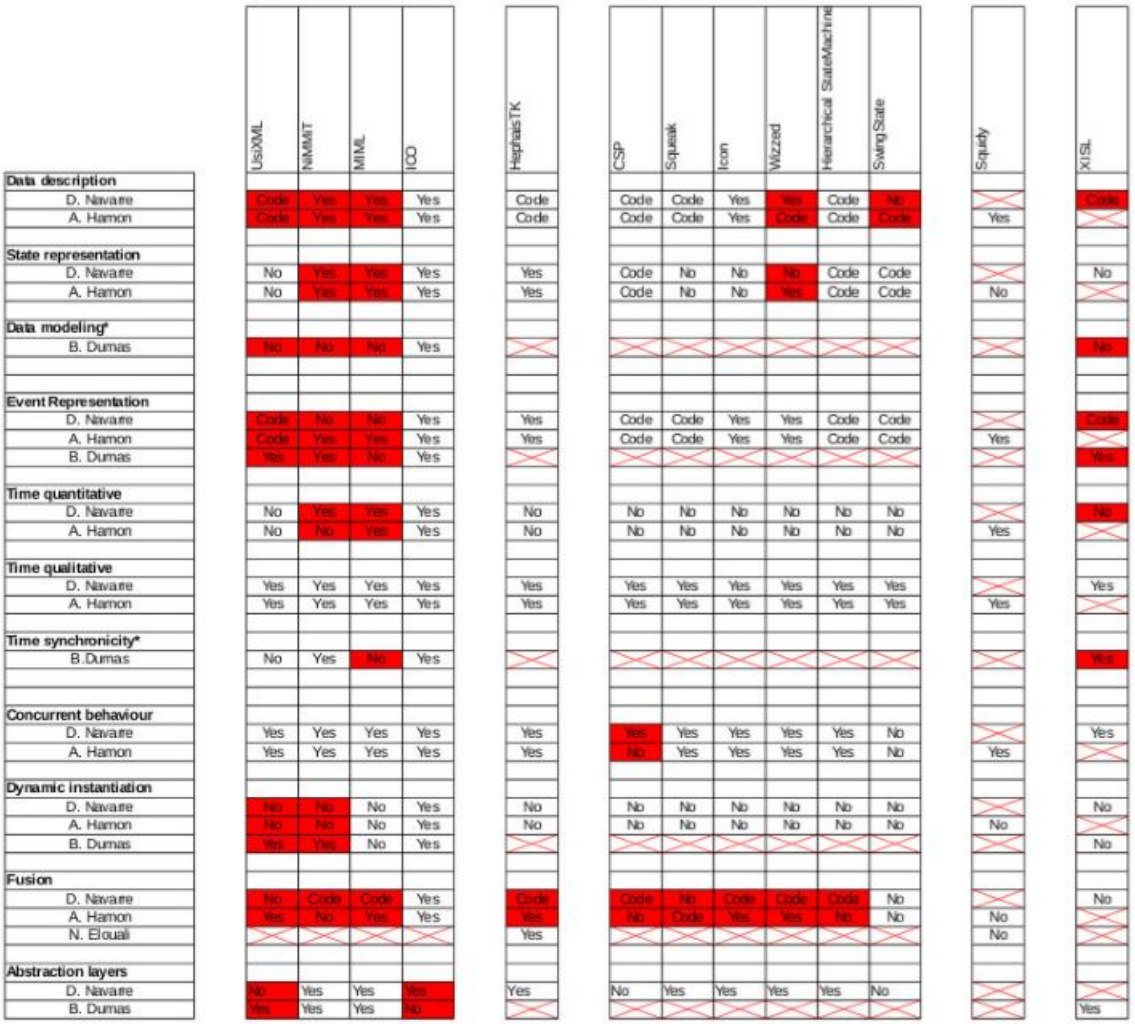


FIGURE 3.1: Cohérence des résultats de l'analyse des différents tableaux comparatifs [4].

Légende :

Yes : le langage gère cette partie ;

No : le langage ne gère pas cette partie ;

Code : le langage gère cette partie par une partie faite en code ;

: le langage n'est pas observé dans ce travail ;

: dissension dans les analyses.

Les points de comparaison de ce tableau [4] proviennent des différents points communs rencontrés dans les tableaux comparatifs étudiés.

- Data description : permet ou non une description des données dans le langage.
- State representation : permet ou non une représentation des états dans le langage.
- Data modelling : caractéristique très proche de la caractéristique de Data Description.
- Event representation : permet ou non une représentation des événements dans le langage.
- Time quantitative : représentation temporelle des évolutions comportementales du système pour un temps donné. Utile pour représenter la fenêtre temporelle d'un moteur de fusion où plusieurs événements sont émis dans la même plage de temps.
- Time qualitative : vise à représenter l'ordre des actions dans l'interaction (succession, simultanéité...).
- Time synchronicity : regroupe les deux caractéristiques de temps.

- Concurrent behaviour : permet de savoir si le langage peut être utilisé de manière séquentielle ou simultanée par plusieurs utilisateurs.
- Dynamic instantiation : permet ou non de créer des éléments manquants au niveau du langage lors de l'interaction.
- Fusion : si le langage dispose ou non d'un moteur de Fusion.
- Abstraction layer : si le langage dispose ou non de plusieurs couches d'abstraction de données.

Dans ce tableau sont présentés les résultats des différents recoupements. Pour faciliter la lecture de l'analyse, les langages ont été triés suivant le nombre d'occurrence dans les différents travaux.

Nous remarquons une cohérence globale des résultats surtout dans les travaux de D. Navarre et A. Hamon. Cette similitude vient peut-être du fait que les deux auteurs viennent du même milieu académique et que A. Hamon se soit inspiré des travaux de son prédécesseur pour réaliser sa thèse. Il faut noter que les critères de comparaison de ces deux travaux sont très semblables. Pour ce qui est des résultats non cohérents, il faut faire attention à la date de réalisation des travaux, un résultat apparaissant « Non » chez D. Navarre, analyse réalisée en 2010, et « Oui » chez A. Hamon, analyse réalisée en 2014, est sans doute dû à un développement du langage ou du framework. Par contre l'inverse ne s'explique pas, un langage ayant des acquis en 2010 peu difficilement les perdre en 2014 (exemple le framework CSP).
Le point de conflit majeur entre ces deux travaux se situe au niveau de la caractéristique de Fusion, neuf résultats sont incohérents.

Nous retrouvons des incohérences entre les travaux de B. Dumas et D. Navarre, pourtant les deux travaux ont été réalisés la même année. Une explication à ces dissensions vient de la divergence de point de vue que les auteurs ont appliqué à leur champ de recherche.
B. Dumas centre son travail sur les langages de modélisation à proprement parler et ne pose un regard critique que sur leurs capacités générales. D. Navarre, quant à lui, offre une analyse plus large se concentrant principalement sur l'expressivité des langages et frameworks.
Pour ce dernier, nous pouvons remarquer que les auteurs présentent leur langage, à savoir ICO, et que l'analyse a dû être orientée de manière à le faire ressortir de l'article. Dans le tableau deux fourni en annexe, seul ce langage donne quasi-entière satisfaction aux caractéristiques de comparaison. Nous sommes peut-être face à un manque d'objectivité des auteurs.

Pour ce qui est du dernier tableau, la comparaison est plus difficile. Les points de comparaison sont situés à une autre niveau d'abstraction que ceux déjà rencontrés.

Après cette analyse, il nous est apparu que SMUIML pouvait bénéficier de nouvelles caractéristiques dans le but de rendre ce langage plus complet au niveau de sa gestion de la multimodalité.

CHAPITRE

4

SMUIML

Le langage SMUIML (Synchronized User Interface Modelling Language) est, comme son nom l'indique, un langage de modélisation dédié aux interactions multimodales. Il fait partie de la catégorie des langages de modélisation par machine à états et s'écrit grâce au langage XML. Son architecture de gestion de la multimodalité est de type CARE.

1 Principes

Afin de permettre la création des interactions, le créateur de ce langage l'a divisé en quatre parties distinctes mais liées. Le langage est divisé en <recognizers>, <triggers>, <actions> et <dialog>. Chaque partie dispose d'un rôle propre et permet d'aboutir au lancement d'une action via une suite de commandes multimodales.

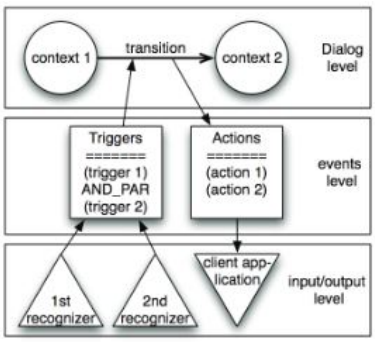


FIGURE 4.1: Les trois niveaux de SMUIML [5].

Grâce au schéma 4.1, nous voyons que SMUIML est divisé en trois niveaux différents. La partie <recognizer> correspond au niveau le plus bas du langage, le plus proche des entrées et sorties. Dans le niveau médian se situent en parallèle les <triggers> et les <actions>, qui ont pour but de gérer les événements.
Au niveau supérieur se situe la phase de <dialog> accueillant la machine à états.
La traduction XML de ce schéma se traduit par un balisage des 4 entités du langage comme illustré dans la figure 4.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<SMUIML>
    <integration_description client="client_app">
        <recognizers>
        <!-- ... -->
        </recognizers>
        <triggers>
        <!-- ... -->
        </triggers>
        <actions>
        <!-- ... -->
        </actions>
        <dialog>
        <!-- ... -->
        </dialog>
    </integration_description>
<SMUIML>
```

FIGURE 4.2: Structure de SMUIML [5].

Recognizers

C'est dans la partie <recognizers> que l'on permet au langage d'associer un « recognizer » à une modalité spécifique pour que celle-ci soit reconnue et puisse être manipulée.
Au niveau du langage, un recognizer va être caractérisé par un nom spécifique, une modalité propre et nous pouvons également lui associer des variables.

```
<recognizers>
    <recognizer name="speech" modality="speech"/>

    <recognizer name="pointing" modality="movement">
        <variable name="posx" value="x" type="int"/>
        <variable name="posy" value="y" type="int"/>
    </recognizer>
</recognizers>
```

FIGURE 4.3: <Recognizers>.

Triggers

La partie <triggers> permet de donner une signification ainsi qu'une valeur aux modalités sélectionnées. Une ou plusieurs significations peuvent être attribuées à un seul recognizer et ceux-ci vont être manipulés par la machine à états.
Un trigger contiendra un nom unique, une valeur ou une signification, et dépendra d'une modalité particulière.

```
<triggers>
    <trigger name="put">
        <source modality="speech" value="put"/>
    </trigger>

    <trigger name="that">
        <source modality="speech" value="that">
            <variable name="posx" value="x" type="int"/>
            <variable name="posy" value="y" type="int"/>
        </source>
    </trigger>

    <trigger name="there">
        <source modality="speech" value="there"/>
            <variable name="posx" value="x" type="int"/>
            <variable name="posy" value="y" type="int"/>
    </trigger>

    <trigger name="point_trigger">
        <source modality="pointing"
            value="x"
            value="y"/>
    </trigger>
</triggers>
```

FIGURE 4.4: <Triggers>.

Action

La partie <actions> indique l'action qu'il faut lancer à la fin de la machine à états. Cette action sera envoyée vers le programme principal.

```
<actions>
    <action name="put_that_there_action">
        <target name="put_that_there_client"
            message=" change that_x=there_x"
            message=" change that_y=there_y">
        </target>
<actions/>
```

FIGURE 4.5: <Actions>.

Dialog

Finalement, nous arrivons dans la partie <dialog> qui reçoit la conception des machines à états. Ces machines sont des combinaisons de triggers qui disposent d'une indication sur la manière dont la multimodalité doit être gérée suivant les principes du modèle CARE. Il existe également un paramètre de temps (LeadTime) indiquant à l'utilisateur le temps dont il dispose pour pouvoir déclencher l'action par le biais des triggers.

```
<dialog>
    <context name="put_that_there">
        <transition_name="modification">
            <seq_and>
                <trigger name="put"/>
                <transition>
                    <par_and>
                    <trigger name="point_trigger">
                    <trigger name="that">
                </transition>
                <transition>
                    <par_and>
                    <trigger name="point_trigger">
                    <trigger name="there">
                </transition>
            </seq_and>
            <result action="put_that_there_action">
            <result context="put_that_there">
        </transition>
    </context>
<dialog/>
```

FIGURE 4.6: <Dialog>.

Un outil supplémentaire intégré dans ce langage permet de gérer la multimodalité.
Il est possible de réaliser des combinaisons de triggers qui vont alors être utilisés de manière séquentielle ou parallèle.
Pour indiquer cela, nous utilisons une série de mots-clés :

- par : désigne les triggers utilisés de manière parallèle ;
- seq : désigne les triggers utilisés de manières séquentielle.

De plus, nous pouvons utiliser plusieurs triggers de manière combinée ou indépendamment les uns des autres ;

- and : désigne les triggers utilisés conjointemen ;
- or : désigne les triggers utilisés de manière distincte ;

Ces mots-clés sont combinés entre eux afin d'obtenir les quatre caractéristiques de fusion du modèle CARE :

- `par_and` (Complementarity) : est utilisé quand plusieurs triggers doivent être validés ensemble.
- `par_or` (Redundancy) : est utilisé quand nous avons affaire à plusieurs triggers redondants dont la signification est similaire. Chacun permet d'obtenir la signification du message envoyé mais le fait de pouvoir utiliser plusieurs triggers pour un message augmente la robustesse du message envoyé et le niveau de compréhension de la machine.
- `seq_and` (Assignation) : est utilisé quand un ou plusieurs triggers individuels sont nécessaires pour la séquence de transition de la machine à états.
- `seq_or` (Equivalence) : est utilisé quand plusieurs triggers peuvent arriver au même résultat mais qu'un seul est pris en compte.

```
<transition>
    <par_and>
        <!--complementarity-->
    </par_and>

    <seq_and>
        <!--assignation-->
    </seq_and>

    <par_or>
        <!--redundancy-->
    </par_or>

    <seq_or>
        <!--equivalence-->
    </seq_or>
</transition>
```

FIGURE 4.7: Différentes combinaisons de triggers [5].

2 HephaisTK

HephaisTK [1] [6] est la boite à outils du langage SMUIML.
Il donne une interface graphique au développeur et sert d'intermédiaire entre le logiciel principal utilisé par l'utilisateur et le langage SMUIML.
Pour l'instant, ce « toolkit » n'a pas encore de déploiement officiel.

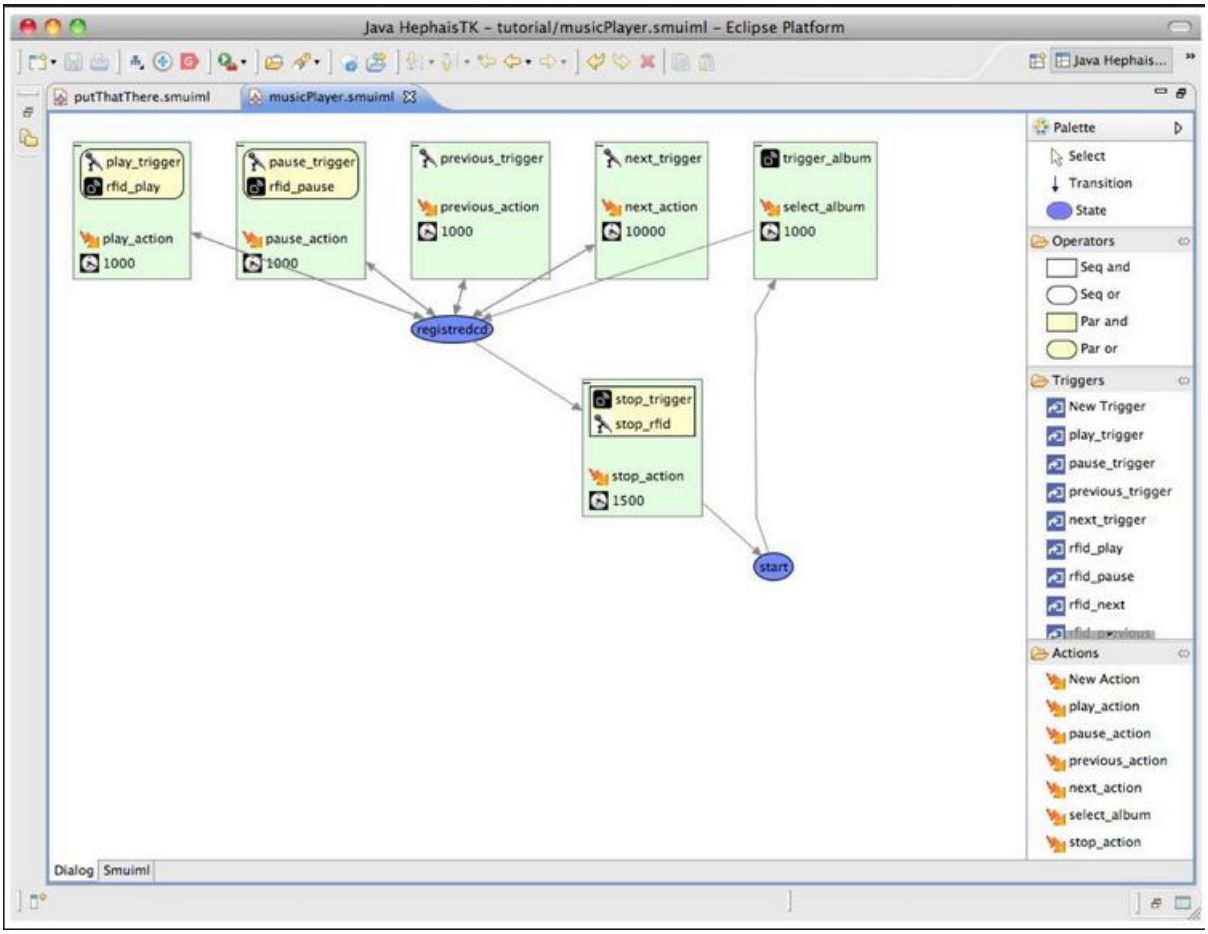


FIGURE 4.8: Exemple « Music player » HephaisTK [6].

Les interactions créées par l'utilisateur se composent de plusieurs éléments.
Le début de l'interaction commence au point « start ». Chaque possibilité d'interaction est définie par une boite dans laquelle se trouve le nom du ou des triggers qui vont être utilisés par l'utilisateur. Les comportements concurrents sont indiqués par des boîtes « operators » qui englobent un certain nombre de triggers et définissent la manière dont ils sont utilisés lors de la communication.
Dans la zone inférieure de la boite se situe l'action qui devra résulter de la suite de triggers et sera de manière générale un message envoyé vers le programme principal.
De plus, le « leadtime » indique le temps dont dispose l'utilisateur pour valider les triggers.
Les boites peuvent être reliées entre elles ou à un nœud concentrateur qui peut servir de lien entre plusieurs boites mais également de boucle pour pouvoir recommencer une nouvelle interaction.

1. http ://diuf.unifr.ch/main/diva/research/research-projects/hephaistk

3 Recueil de données

Grâce à l'analyse de l'état de l'art [4], nous avons pu découvrir et lister une série de critiques qui ont été émises à l'encontre du fonctionnement de SMUIML et de HephaisTK et qui nous permettront de lister les améliorations à développer sur le langage.
Par souci de lisibilité, les liens de références contenus dans ces articles ont été modifiés pour correspondre à la bibliographie de ce travail.

La première remarque nous vient de F. Cuenca [17],

> ≪ HephaisTK is a rapid prototypying tool that offers visual notations for describing multimodal interactions.
> In HephaisTK models, there is a clear separation between the specifications of events and the dialog model, which, in our opinion enhances their readability. In HepahisTK, each arc of a FSM is annotated with a user-defined event pattern and an event-handling callback.
> The callbacks are to be launched when the event patterns occur, thus causing the system switch to a new state.
> The straightforward way of a binding event patterns to event-handling callbacks is the key idea we used to create Hasselt.
> Such type of binding if something that programmers have been used to for decades ; to define the human-machine interaction in WIMP system, one has to bind events such as mouse clicks and keystrokes to event handlers. To define an event pattern, HephaisTK users have to specify the relation among its constituent events.
> In HephaisTK, the relations among events can be Complementarity, Assigned, Redundant or Equivalent, which are all the ways in which modalities can interact among each other, according to a well-known theoretical framework, called CARE [8]. One limitation of this tool is its inability to provide partial feedback.
> Event-handling callbacks are launched after some pattern of events has been fully detected, but cannot be launched in the middle of this process.
> Although, in theory, this can be fixed by splitting the event pattern and rewiring the visual model, this unnecessarily increases the size of the model and the task time.
> Unlike HephaisTK, Hasselt allows binding multiple event-handling callbacks to one single event pattern ;
> this is possible thanks to its notations for specifying the moment when the callbacks have to be launched. ≫

L'auteur de cet article signale que le modèle CARE ne permet pas d'obtenir de feedback partiel sur l'avancement d'un process mais uniquement à la fin de celui-ci.
F. Cuenca a conçu un langage de modélisation basé sur des machines à état, HasseltUIMS [7]. Son langage permet à l'utilisateur de visualiser l'évolution au cours du temps par une représentation graphique de la machine à états.

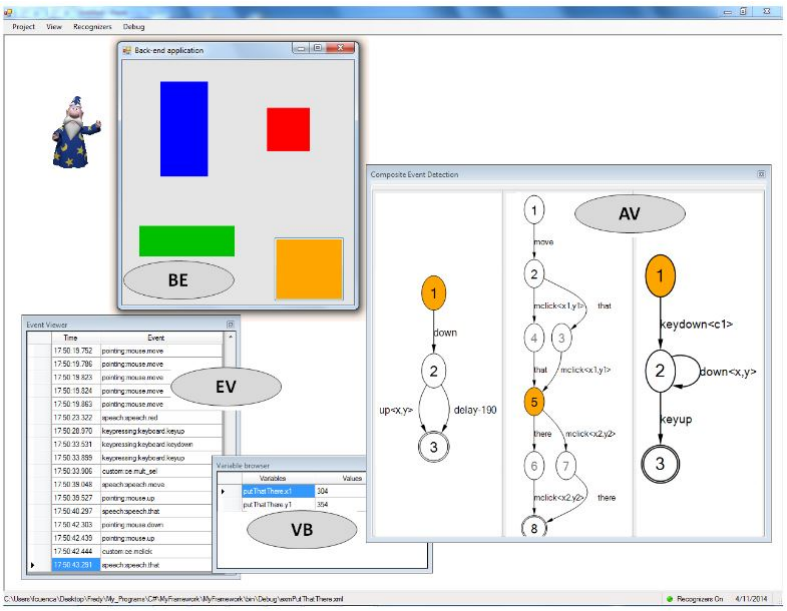


FIGURE 4.9: HasseltUIMS [7].

La figure 4.9 montre HasselUIMS durant son fonctionnement, la fenêtre EV (Event Viewer) affiche les événements détectés par les modalités d'entrée. La fenêtre VB (Variable Browser) montre les valeurs des différents événements. La fenêtre AV (Automata View) donne l'évolution de la machine à états suivant la détection des événements. Et pour finir la fenêtre BE (Back-End application) représente l'application avec laquelle l'utilisateur final doit interagir.
Effectivement, au niveau conceptuel, HephaisTK, et par conséquent SMUIML, ne donne aucun retour sur l'état d'un processus en cours.
Cela peut s'avérer contre-productif dans le cas d'une information mal réceptionnée ou mal comprise par l'ordinateur.

D'autres critiques nous viennent de l'article écrit par N. Elouali, J.C. Tarby, X. Le Pallec et J. Rouillard [16].

> « Le tableau montre que les projets Dynamo et HephaisTK/SMUIML ont apporté des réponses concrètes au problème de création des interfaces multimodales. Toutefois, une faiblesse de ces projets concerne l'écosystème associé à leur plate-forme : les configurations matérielles sont inhabituelles et peu répandues ou dans le cas contraire (c'est-à-dire ordinateurs de bureau classiques) le nombre de modalités est réduit. »

Dans cet article consacré à un langage dérivé de SMUIML et ayant pour cible les appareils mobiles, les auteurs mentionnent le fait que le nombre de modalités supportées par SMUIML est assez réduit.
SMUIML dispose déjà d'un développement assez évolué, mais il est insuffisant pour être appliqué à des technologies comme des smartphones ou tout autre appareillage embarqué.

> « Notre méta-modèle s'inspire de SMUIML [18]. Une application est représentée au travers d'états avec des événements en entrée et en sortie (non traités dans SMUIML) associés à des modalités et possédant des effets. »

SMUIML est à la base un langage centré sur les modalités d'entrée et n'a pas encore été développé de manière approfondie pour gérer les modalités de sortie.

Dans l'article de C. Branton et al. [19], les auteurs se concentrent plus sur HephaisTK que sur le langage SMUIML en tant que tel.

> « HephaisTK [20] is also used for multimodal system development, designed primarily to serve as a test bed for multimodal fusion algorithms [21]. HephaisTK uses SMUIML to specify and integrate multiple levels of program behavior, but does not support distributed applications directly. »

Le point qui est mis en avant dans cette article est que SMUIML ne supporte pas les applications distribuées. En réponse à cela, nous pouvons dire que le langage n'est, à la base, pas prévu pour ce genre d'applications et n'est donc pas capable de les gérer. Son créateur ne cherchait pas à gérer les applications embarquées, mais de fournir un langage fonctionnel, stable et polyvalent. Il est possible qu'au cours d'un développement ultérieur, ce type d'application soit pris en charge.

J.F. Ladry mentionne, dans son article [22], que SMUIML ne dispose pas de modélisation bas-niveau. Ceci entre dans le contexte d'une comparaison entre SMUIML et un langage orienté objet, à savoir ICO.

> « HephaisTK permet grâce au système multi-agents de pouvoir modéliser indépendamment chaque périphérique et ensuite de relier les événements de ces périphériques vers un agent de plus haut niveau permettant de réaliser des interactions plus complexes.
> Par contre, la notation n'est pas prévue pour permettre de modéliser les interactions de bas-niveau. »

Dans le cadre de son article sur le langage de modélisation ICO, dans lequel il le compare à d'autres langages, J.F. Ladry signale que SMUIML ne dispose pas de modélisation d'interaction bas-niveau.
Encore une fois SMUIML n'a pas été pensé pour ce genre d'actions et son objectif n'est pas de créer des interactions bas-niveau.

Dans un article analysant différents « toolkit » et englobant HephaisTK [23], les auteurs mentionnent le fait que celui-ci ne peut gérer qu'un seul output par événement.

> « They can not handle multiple parallel outputs and therefore can only have one kind of output per event. »

Nous pouvons répondre à cette critique de la même manière qu'à celle traitant du défaut du langage à ne pas gérer les modalités de sortie.

D. Navarre [24] remarque dans son article, via un tableau comparatif, qu'il n'existe pas encore d'application « taille réelle » pour SMUIML et que la description des données fournies par le langage n'est disponible qu'au niveau du code fourni et non au niveau de l'UIDL.

Selon lui, il manque également une représentation quantitative du temps.

En plus de toutes ces remarques, B. Dumas, le créateur du langage, a émis certaines remarques afin d'améliorer son langage.
Il aimerait que l'architecture MMMM soit intégrée au langage pour pouvoir donner plus de puissance à ce dernier et l'orientée plus vers l'utilisateur.
Le fait de modifier l'architecture globale du système permettra de pallier certains de ces manquements relevés par les critiques, notamment le manque de retour d'information vers l'utilisateur lors d'un processus.

CHAPITRE

5

MODIFICATIONS APPORTÉES

Après la phase d'analyse élaborée dans le chapitre précédent, nous voici en possession d'un cahier de charge. Nous allons pouvoir aborder une phase d'avantage orientée sur le design.
Étant donné que le but principal de ce travail est d'apporter des modifications au langage SMUIML en vue de l'améliorer, nous passerons en revue les différents changements qui ont été effectués sur le langage dans ce chapitre.
En plus du langage, la refonte graphique de HephaisTK, plus adaptée à la nouvelle structure du langage, a été pensée également.

Cahier des charges

Suite à la lecture des différentes critiques et des besoins de B. Dumas, revoyons les points sur lesquels vont s'orienter le travail :

- Feedback partiel d'un processus en cours.
- Implémentation de l'architecture MMMM.
- Symboles peu compréhensibles.
- Problème de temporalité.
- Changements de labels pour certains éléments et attributs.
- Nouvelle interface graphique.

1 Modifications apportées à la structure du langage

1.1 Définition d'une modalité

Avant de s'attaquer aux modifications de la structure du langage, nous avons dans un premier temps défini ce qu'était une modalité.
Effectivement, tout au long des analyses faites pour les travaux de cette année et de l'année précédente, aucun des auteurs lus ne fait mention de la modalité en tant que telle.

Sur la base du cours de méthodes d'interaction avancées [25], nous avons réalisé une modélisation sous forme XML de ce que peut être une modalité reconnue. Cette définition pose les limites d'une modalité ce qui améliore sa reconnaissance.

Voici, en figure 5.1, la structure d'une définition de modalité pour le mouvement.

```
<Modality_Definition>

    <modality>
        <device>        <!-- Nom du Device utilisé pour la reconnaissance-->
            <default>   <!-- Définiton par défaut-->
                <representation>    <!-- Représentation de la modalité-->

                    <signal>        <!-- Signal qui doit être perçu par le device-->
                    <!-- Signal reconnu-->
                    </signal>

                    <context>       <!-- Environnement contextuel capté pouvant modifier l'interaction-->
                        <!-- Contexte reconnu-->
                    </context>

                </representation>
            </default>


            <expert>        <!-- Mode expert-->
                <movement_recognition>
                <!-- reconnaissance de mouvements particuliers de l'utilisateur-->
                </movement_recognition>
            </expert>

        </device>
    </modality>
```

FIGURE 5.1: Structure d'une définition de modalité de type mouvement.

Dans un premier temps, la modalité porte un nom. Dans notre modélisation, ce nom référencie la modalité à son type principal (le geste, les sons...).
Chaque modalité possède deux caractéristiques différentes, à savoir le « device » et la « representation » [8].
Le « device » détermine par quel moyen physique l'utilisateur va faire passer son message : par la caméra, le micro, un outil connecté... .
La « representation » est un type particulier du type principal de la modalité.
Par exemple, pour une modalité concernant les gestes (type principal), nous retrouvons comme « representation » toute une série de mouvements comme l'abduction, l'adduction, la rotation, le balayage, le pointage... .
Ces deux éléments permettent déjà de définir un grand nombre de modalités pouvant être prises en charge lors d'une communication homme-machine.
Afin d'améliorer la précision de la définition, nous pouvons augmenter sa portée sur des éléments anatomiques ou physiques, par exemple, en ajoutant à la définition quelle partie du corps est concernée par le mouvement.

Dans cette définition, nous signalons un niveau par défaut et un niveau expert.
Le niveau par défaut est une définition en toute généralité de ce que l'on peut retrouver comme définition générale et commune au plus grand nombre.
Le niveau expert permet à l'utilisateur de créer des définitions personnalisées à ses caractéristiques propres (reconnaissance de sa voix propre, de sa morphologie,...).
Un dernier élément, mais non des moindres dans cette modélisation, nous faisons la différence entre ce que l'on peut appeler le signal et le contexte.
Le signal regroupe tout ce qui doit être reconnu dans le cadre de l'interaction entre l'homme et la machine, tout ce que l'ordinateur capte et reconnait comme source d'information.
Le contexte va regrouper des événements ponctuels qui peuvent modifier l'interaction. Il dépend principalement de l'environnement extérieur qui peut avoir un impact sur la réponse de la machine. Par exemple, l'utilisateur se trouve dans un endroit fort bruyant et aimerait écouter de

la musique via son smartphone. Il lance son application musicale grâce à une interaction mais le volume sera malheureusement pour lui trop faible pour être audible à cause du bruit extérieur. Grâce à la prise en compte du contexte environnemental de l'utilisateur, le smartphone peut, de lui-même, modifier le volume pour qu'il corresponde à un niveau audible pour son utilisateur. En annexe, se trouvent des exemples de définitions complètes afin d'illustrer ces propos.

1.2 Modifications de la structure du langage

MMMM

La première modification majeure que l'on peut répertorier est le changement d'architecture du langage. Celle-ci a été modifiée de manière à correspondre au modèle MMMM examiné au chapitre 5.

Pour valider cette architecture, nous avons effectué une séparation au niveau de la section <trigger>. Nous avons créé deux nouveaux triggers, les <user_ trigger> et les <system_ trigger>. Ils modélisent la séparation, présente dans MMMM, entre l'utilisateur et le système. Les <user_ trigger> gardent la même fonction que les <triggers> du SMUIML d'origine, ils donnent une signification aux modalités.
Par contre, nous avons ajouté une ligne de validation du trigger. Celle-ci a pour rôle d'envoyer un booléen au système quand l'utilisateur valide un trigger se trouvant dans la machine à états. Le but de cette validation est de pouvoir créer un système de gestion des erreurs qui sera explicité dans une autre section de ce chapitre.

```
<utrigger name="there">
    <source modality="speech" value="there"/>
        <variable name="posx" value="x" type="int"/>
        <variable name="posy" value="y" type="int"/>
    </source>
    <validation name="there_ok" value="true"/>
</utrigger>

<utrigger name="point_trigger">
    <source modality="pointing">
        <value="x"/>
        <value="y"/>
    </source>
    <validation name="pointing_ok" value="true"/>
</utrigger>
```

FIGURE 5.2: Utrigger « There » et utrigger « Pointing ».

Dans la figure 5.2, nous avons, dans un premier temps, un trigger appelé « There » utilisant la modalité « speech » et ayant deux variables x, y attachées à elle.
Ensuite, nous trouvons un trigger nommé « pointing » qui permet, grâce à une reconnaissance par caméra, d'indiquer une position dans l'axe x, y.

Le rôle des <system_ trigger> est de recevoir les différentes combinaisons de <user_ trigger>. La validation de ces nouveaux éléments se fait par l'envoi et la réception correcte des différents <user_ trigger> dont ils sont dépositaires.

```
<strigger name="there">
    <complement>
    <TW value='1500'/>
    <req utrigger 'there'/>
    <req utrigger 'pointing_trigger'/>
    <complement/>
</strigger>
```

FIGURE 5.3: Structure d'une définition de modalité de type mouvement.

Dans cette partie du code, les deux éléments utilisateurs sont combinés dans ce trigger système. Pour pouvoir le valider, l'utilisateur devra effectuer en complémentarité l'action de dire « there » et le geste de pointer vers une position x, y. Nous avons aussi une variable TW (Time Within) liée au temps disponible pour la validation des triggers complémentaires.

Ceci permet une simplification de la phase de dialogue mais aussi une meilleure capacité de construction d'un système.
Les triggers systèmes sont des entités plus facilement manipulables pour une machine à états car elle regroupe déjà une partie des combinaisons nécessaires pour le système.

Combinaison

Pour mettre en place la nouvelle architecture, il a fallu modifier les combinaisons de modalités disponibles.
L'ancien système nous fournissait les combinaisons présentées dans le chapitre 7.
Or, dans le nouveau système, nous retrouvons ceci :

```
<transition>
    <unimod>
        <!--1-1-1-->
        <!--userUnimodal -- systemUnimodal-->
    </>
    <equival>
        <!--N-1-1-->
        <!--userUnimodal -- systemEquivalence without redundance-->
    </>
    <redundancy>
        <!--N-N-1-->
        <!--userRedundancy -- systemEquivalence with redundance-->
    </>
    <complement>
        <!--N-N-1-->
        <!--userComplementarity -- systemComplementarity-->
    </>
    <concur>
        <!--N-N-N-->
        <!--userConcurrency -- systemConcurrency-->
    </>
</transition>
```

FIGURE 5.4: Les combinaisons de MMMM.

Dorénavant, les différentes combinaisons possibles portent des noms explicites pour une meilleure compréhension et correspondent aux cinq combinaisons disponibles dans le modèle MMMM expliqué au chapitre 2.
Dans les commentaires associés aux combinaisons, une indication de multiplicité est présente <!–x-y-z–>, où x indiquant le nombre de modalités en entrée devant être utilisées, y, le nombre de modalités reconnues par le système, et z le nombre de commandes envoyées en réaction.

Dialog

```
<dialog>
    <context name="put_that_there">
        <transition_name="modification">
                <strigger name="put">
                    <TTN value='1000'/>
                </strigger>
                <strigger name="that"/>
                    <TTN value='1000'/>
                </strigger>
                <strigger name="there"/>
            <result command="put_that_there_action"/>
            <result context="put_that_there"/>
        </transition>
    </context>

    <context name="put_error">
    <validation putOk value='false'/>
        <result error="putError"/>
    <context/>

    <context name="that_error">
    <validation that_ok value='false'/>
        <result error="that_error"/>
    <context/>

    <context name="there_error">
    <validation there_ok value='false'/>
        <result error="there_error"/>
    <context/>

    <context name="pointing_error">
    <validation pointing_ok value='false'/>
        <result error="pointing_error"/>
    <context/>
<dialog/>
```

FIGURE 5.5: <Dialog>.

La phase de dialogue est maintenant simplifiée, elle reçoit une série de triggers système qui doivent être validés pour aboutir à l'envoi de l'action vers le programme principal.
L'utilisateur peut déterminer un temps limite de validation pour un trigger à partir de la validation du trigger précédent, le Time To Next (TTN).

De nouveaux éléments apparaissent dans cette partie <dialog>. Nous avons intégré une gestion des erreurs dans cette phase. Elle n'apparaitra pas explicitement dans la machine à états mais se déclenchera à partir du moment où un trigger ne sera pas validé correctement par l'utilisateur. L'utilisateur obtiendra alors en retour un message lui indiquant son erreur et sera invité à réenvoyer l'information manquante.

Ce changement d'architecture, qui est une modification profonde du langage SMUIML, se justifie par la volonté de le rendre plus orienté vers l'utilisateur.
Cette accessibilité est réalisée par la séparation des processus utilisateur et processus système, ce qui facilite la création et la gestion d'interactions multimodales, que ce soit la phase de combinaison ou la phase de création de machine à états.
Nous retrouvons en plus une explicitation des symboles combinatoires dans ce langage. Dans la première version, ces symboles étaient peu instinctifs (par_and, par_or, seq_and, seq_or), tandis que dans cette version nous avons une nomenclature plus distincte et compréhensible.
Le fait d'avoir la possibilité de gérer des erreurs de validation de triggers permet à l'utilisateur une meilleure gestion de son interaction et l'obtention d'un retour sur l'évolution du processus en cours.

Triggers conditionnels

Une autre modification au niveau des triggers est l'ajout de triggers conditionnels.
De manière générale, ce sont des triggers déjà existants auxquels s'ajoutent une ou plusieurs conditions supplémentaires pour leur validation.
Ceci permet de déclencher des actions particulières suivant l'environnement interne du programme principal.

```
<ConditionnalTrigger name="put_test">
    <condition variable 'speechContent' test= '==put'/>

    <condition variable 'pointingContent' element= 'OK_cursor' test= '==pointing_OK' />

</ConditionnalTrigger>
```

FIGURE 5.6: <Conditional trigger>.

Dans la figure 5.6, nous avons un trigger qui met une condition sur le trigger « Put ». L'objectif de base de ce trigger est d'être un initiateur à l'interaction « Put-That-There »où l'utilisateur ne fait que prononcer le mot « Put ». Grâce au trigger conditionnel, nous pouvons ajouter une nouvelle action au trigger d'origine si, dans ce cas-ci, le curseur de souris se situe sur un bouton « OK ». La position de ce bouton est fournie par l'application principale.
Grâce à ces triggers, nous offrons la possibilité à l'utilisateur d'enrichir le nombre d'actions possibles pour un seul trigger.
Cela est possible grâce à l'architecture utilisée par Android et IOS, le design pattern« Modèle-Vue-Contrôleur » [26].
Pour rappel, ce pattern est formé de trois entités distinctes pour obtenir une gestion optimale des interfaces graphiques.
Nous avons dans un premier temps, le « modèle » qui contient et gère les données, il garantit également l'intégrité de celles-ci.
Puis nous avons la « vue » qui est l'interface avec l'utilisateur et lui affiche les données contenues dans le modèle. Cette entité reçoit également toutes les informations et actions envoyées par l'utilisateur qui seront ensuite transmises au contrôleur
Pour finir, nous retrouvons le « contrôleur », qui gère la partie logique de l'architecture. Il est en charge de synchroniser et mettre à jour le modèle et la vue suivant les actions de l'utilisateur.
Cette séparation garantit l'indépendance des entités, et le développement d'une des entités n'affecte pas le contenu des deux autres.

Dans la figure 5.7, nous pouvons observer les interactions entre les trois parties de l'architecture.

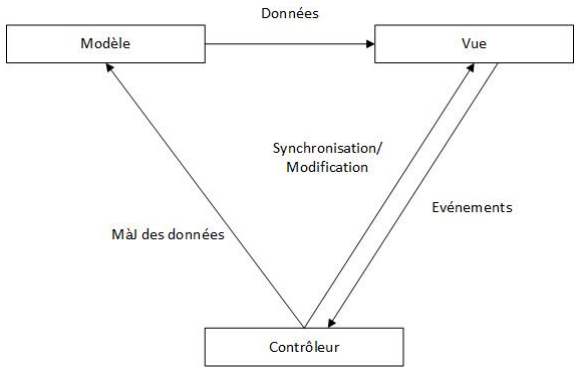


FIGURE 5.7: Interactions entre le modèle, la vue et le contrôleur [27].

Dans un premier temps, nous pouvons envisager les premiers contacts entre SMUIML et cette architecture via la résolution des triggers conditionnels. Dans les OS embarqués, l'entité « modèle » est représentée par des fichiers XML, ce qui est tout à fait compatible avec notre langage de modélisation.
Cette partie « modèle » regroupe toutes les informations sur les éléments de l'interface graphique dont SMUIML a l'utilité, comme par exemple la taille des éléments ou leur position sur l'écran. Le langage n'aurait qu'à aller chercher l'information, via un contrôleur intermédiaire ou HephaisTK, dont il a besoin pour valider le trigger conditionnel.

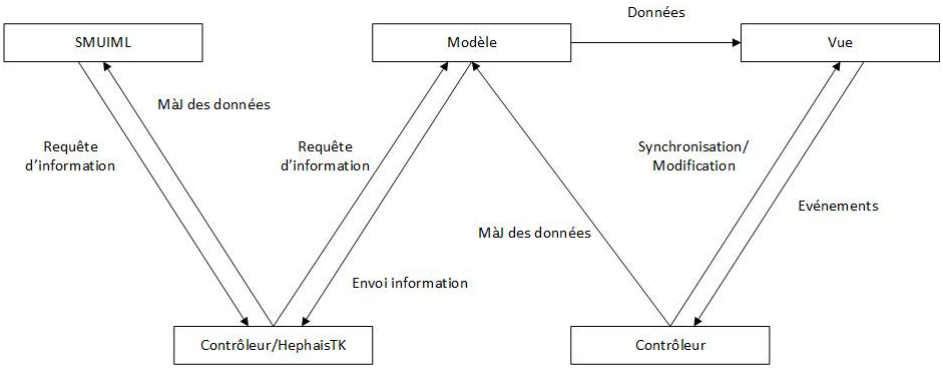


FIGURE 5.8: Interactions SMUIML et architecture MVC.

En seconde intention, nous pourrions envisager HephaisTK comme une application de création d'interactions directement liée à l'OS.
Le développeur d'interactions créerait ses machines à état directement dans l'application et le code XML qui en ressortirait serait intégré aux fichiers ressources d'Android ou d'IOS.
Malheureusement, faute de temps, nous n'avons pas pu aller plus loin dans la réflexion et dans l'implémentation de ces solutions.

Temporalité

Le « lead time » de la partie dialog a été modifié. Nous retrouvons maintenant une notion de TTN (Time To Next) et de TW (Time Within).
Time To Next indique le temps dont l'utilisateur dispose entre l'enregistrement de deux triggers.
Time Within indique le temps dont l'utilisateur dispose pour introduire une série de triggers.
Avec ces modifications, l'utilisateur dispose d'un meilleur contrôle sur la durée d'interaction.

En annexe se trouvent plusieurs implémentations avec l'architecture MMMM. Ces exemples [3] ont servis de tests pour évaluer le modèle.
La figure 5.9 énonce ces exemples.

System name	Use case
Bolt's "put-that-there"	1. The user points a form and utters "Put that" then points a location and utters "there" 2. "Move this to the right of the purple square" + pointing the blue 3. "Delete this purple square"
Notebook	1. "Clear all" 2. "Insert a note here, between the second and the third" 3. "Next note" + clicking on the next note button 4. Clicking on the insert a note button, then uttering "between the second and the third"
MATIS	1. "Show me the USAir flights from Boston to this city" along with the selection of "Pittsburgh" with the mouse on the screen 2. clicking on the Denver airport to see the flights, then typing in the dedicated text window "to Boston" 3. "Show me the flights from Boston to Pittsburgh" and clicking on the Denver airport to see the flights

FIGURE 5.9: Enoncé des exemples.

Après toutes ces modifications, nous pouvons mettre à jour la figure 4.1.

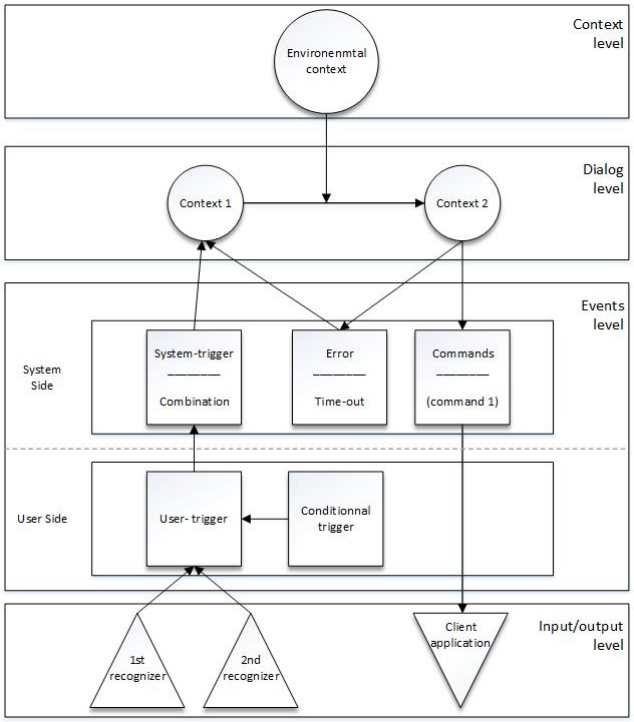


FIGURE 5.10: Les quatre niveaux de SMUIML.

Nous voyons dans cette structure, en figure 5.10, un nouveau niveau de données apparaitre, le niveau contextuel qui influence la phase de dialogue.
Le niveau trigger s'est vu séparé en deux parties distinctes, un « user side » et un « system side ».
Les triggers conditionnels se situent au niveau de l'utilisateur et vont influer les triggers utilisateurs.
Le côté système est plus riche : il regroupe ses propres triggers, et c'est cette partie qui va gérer

le retour des erreurs et l'envoi de la commande à l'application cliente.
La phase de dialog ne change pas.

2 Modifications apportées à l'éditeur graphique

Dans cette phase de design, nous avons modifié, en plus du langage, l'éditeur graphique HephaisTK.
Cette refonte a pour but de parvenir à obtenir une meilleure correspondance avec la nouvelle architecture de SMUIML mais aussi de pouvoir appréhender toutes les phases de conception d'un système.
En raison de contraintes temporelles, nous n'avons pas pu coder une réelle implémentation graphique. Ces changements se trouvent donc sous la forme d'un wireframe cliquable créé grâce au logiciel Balsamiq [1].

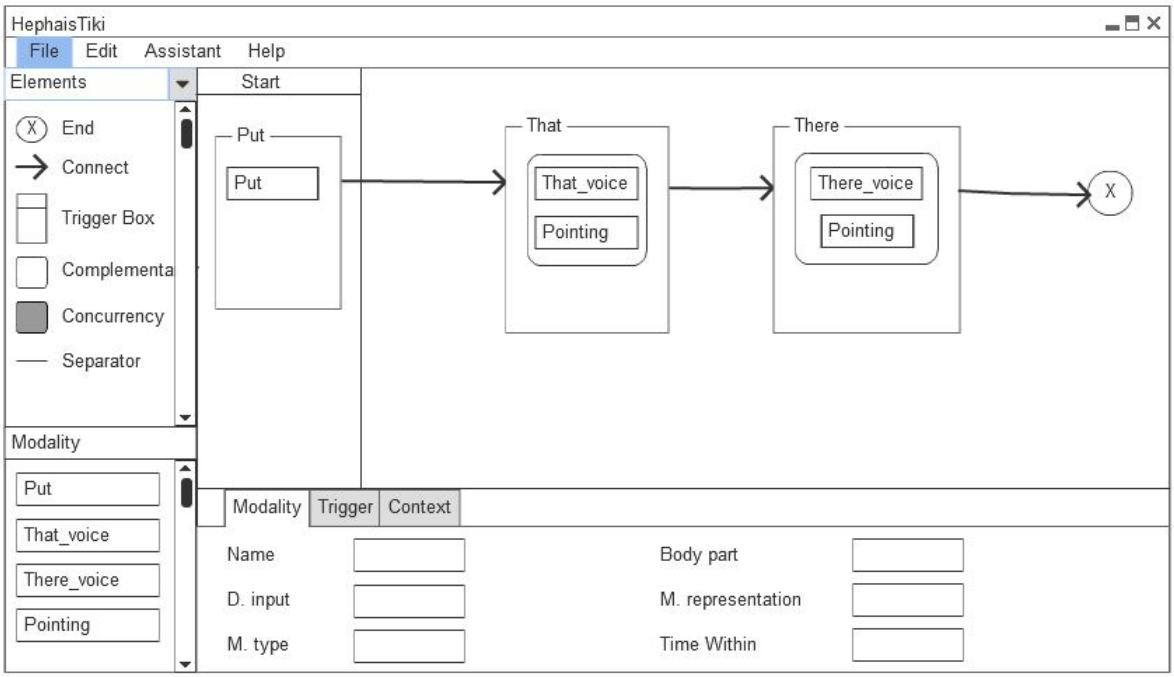


FIGURE 5.11: HephaisTiKi.

Dans cette interface illustrée en figure 5.11, renommée HephaisTiki pour éviter toute confusion avec son logiciel parent, nous retrouvons une fenêtre principale dans laquelle sera créée la machine à états pour aboutir à la réalisation de l'interaction.
Cette fenêtre principale est composée d'une zone « Start »où la trigger box placée à cet endroit est utilisée comme état initial.
À gauche de la fenêtre principale se situent deux fenêtres regroupant des éléments différents.
Celle du dessus contient une série d'éléments utiles à la création d'interactions : un état final, un élément connecteur et un élément « trigger box ». Ces dernières ont pour but de recevoir les modalités dont l'utilisateur se servira. Pour faire un parallèle avec SMUIML, ce sont les <system_trigger>.
Ensuite, nous retrouvons les éléments propres aux propriétés combinatoires du langage. Une boîte pour la combinaison complémentaire, une pour la combinaison concurrentielle et un

1. https ://balsamiq.com/

élément séparateur.
L'utilisateur doit englober les modalités dans les boites complémentaires ou concurrentielles afin d'indiquer au système avec quelle combinaison l'association de modalité doit être traitée. L'élément de séparation est utile pour différencier l'équivalence et la redondance. La première combinaison va utiliser cette séparation dans la trigger box tandis que la deuxième ne les utilisera pas. L'unimodalité n'est pas concernée par ces éléments car elle se retrouvera seule dans la trigger box.

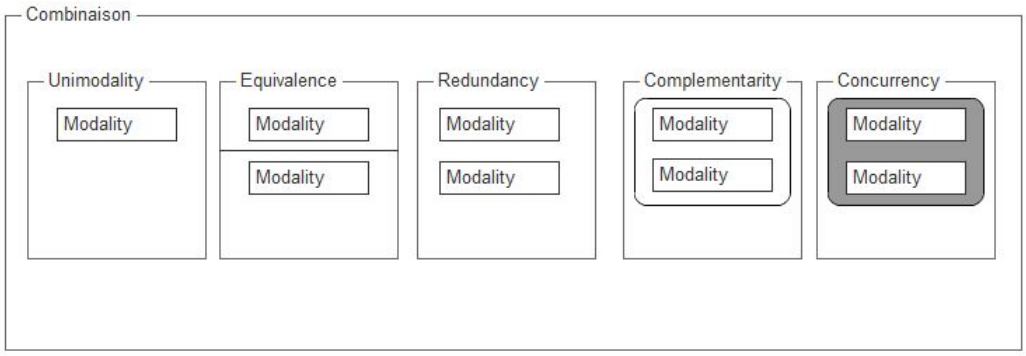


FIGURE 5.12: Représentation graphique des combinaisons.

Ensuite dans le coin inférieur gauche se trouve la fenêtre contenant les modalités créées par l'utilisateur.
Pour les concevoir, il dispose d'un assistant de création.

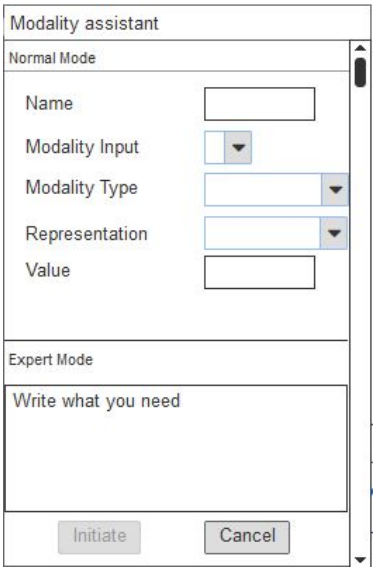


FIGURE 5.13: Assistant de créateur de modalités.

Celui-ci demande différentes informations utiles à la création de la modalité comme son nom, son périphérique de reconnaissance, de quel type cette modalité sera (un mouvement, un son...), quelle sera sa représentation et pour finir la valeur qui sera associée à cette modalité.
Pour faire un parallèle avec SMUIML, nous nous retrouvons dans la partie <user_ trigger>.

Après la création de ses différentes modalités, l'utilisateur peut élaborer une machine à états. Elle doit se composer d'un départ représenté par une trigger box dans la zone de départ (≪ Start

Zone ≫), d'un corps qui doit se composer au minimum d'un symbole connecteur et pour finir d'au minimum un état final. Une machine peut avoir plusieurs états finaux mais doit alors disposer de plusieurs chemins possibles.

Afin d'aider l'utilisateur a accomplir son but, une aide lui est proposée lui expliquant les notions de modalités, trigger et context et lui définissant les types de combinaisons possibles.

Une option est aussi disponible pour l'importation et l'exportation de code XML.
Si le code introduit par l'utilisateur correspond à SMUIML, il serait possible de recréer une machine à états complète. Et inversement, à partir de la machine à états, il y aurait possibilité de générer automatiquement du code.

La modélisation par machine à états a été choisie pour sa grande maniabilité et sa facilité à la compréhension pour un utilisateur lambda.
On obtient alors une structure facilement transformable et permettant une grande liberté dans la création.

Le but de cette implémentation est de permettre à l'utilisateur de créer un système d'interactions sans passer par du code qui peut s'avérer rébarbatif pour les non-initiés.

3 Validation des modifications

Afin de vérifier et de valider les modifications effectuées, nous avons procédé à une phase d'évaluation du système.
Une méthodologie d'évaluation est fournie en annexe de ce document.
Un échantillon de trois testeurs, chacun ayant un niveau en informatique différent (bas, moyen, professionnel), s'est prêté au jeu de l'évaluation.
Voici le profil des testeurs :

- Testeur 1 : femme, 33 ans, pharmacienne, compétences en informatique basses.
- Testeur 2 : homme, 37 ans, technologue en imagerie médicale, compétences en informatique moyennes.
- Testeur 3 : homme, 32 ans, informaticien, compétences en informatique professionnelles.

Cette évaluation s'est réalisée en deux parties, la première sur le wireframe créé comme exemple d'implémentation graphique. Les testeurs doivent l'explorer tout en donnant leurs impressions et leurs avis sur son aspect graphique et sur sa praticité. Toutes les options n'étant pas encore implémentées, l'évaluateur doit leur fournir les informations manquantes.
La deuxième partie du test concerne la partie compréhension et construction d'une interaction multimodale par le biais de plusieurs exercices de modélisation graphique.
Nous n'avons délibérément pas demandé aux testeurs de faire une modélisation XML de l'interaction, car leurs niveaux de compétences varient fortement et il était évidemment exclu de demander aux testeurs d'apprendre la programmation XML. Préalablement au test, nous partons du postulat que le code est normalement réussi, car généré automatiquement, si les testeurs réussissent la modélisation.
Cette partie du test s'est réalisée sur papier, le wireframe ne proposant pas d'interaction fonctionnelle de création.

Plusieurs constats sont ressorties de ces évaluations :
- **Graphique** :

Pros :
- Visuel bien différencié entre complémentarité et concurrence.
- Côté instinctif.

Cons :
- Aspect graphique un peu trop austère, manque de couleurs.
- Manque de reconnaissance visuelle sur le type de modalités (voix, geste...).
- Manque de visualisation du contexte (non implémenté dans cette version).
- Système de séparation pour l'équivalence et la redondance pouvant entrainer des confusions.
- La différence entre le normal mode et l'expert mode dans la création de modalité. n'est pas assez marquée.
- Le nom de « trigger box » est peu explicite.
- Série d'onglets de bas de fenêtre devrait être séparés, confusion entre « modality », « trigger »et « context ».

- **Fonctionnel** :

Pros :
- Assistant utile à la création de modalité.
- Possibilité de créer plusieurs chemins.
- Menu Help utile dans l'explication des différentes combinaisons.
- Création des modalités logique et instinctive.

Cons :
- Manque d'un accès au code après validation de la machine à états.
- Pas de visualisation de la partie <commands>.
- Pas de possibilité de modifier la partie expert.
- Manque d'éléments dans les menus.
- Obligation de passer par l'assistant pour créer une interaction, manque de visibilité de cette information.
- Principe des « separator » mal compris.

- **Compréhension et construction** :

Pros :
- Compréhension générale de la notion de multimodalité satisfaisante.

Cons :
- La nomination des différents éléments peut embrouiller les sujets.
- Difficulté de différenciation entre équivalence et redondance.

Pour la deuxième partie de l'évaluation, les testeurs ont dû appliquer la construction d'interaction sur trois exemples choisis.
Nous prenons comme mise en situation un utilisateur voulant interagir avec son smartphone.
Exercice 1 :
- Allumer son téléphone en disant « Salut R2 » ou en claquant des doigts ou en appuyant sur le bouton de démarrage ;
Exercice 2 :
- Ouvrir une application en appuyant sur le bouton démarrage et en disant « ouvre app » ;
Exercice 3 :
- Lancer une application musicale en appuyant sur l'icône de l'application et en disant le nom de la musique voulue et si le volume est trop bas, le mettre en position moyenne.

Ces interactions simples mettent en place des notions et concepts liés au modèle MMMM.
La première utilise la logique d'équivalence, la deuxième de complémentarité et enfin la troisième fait appel à la complémentarité, mais également à la notion de contexte. Cette interaction contextuelle doit s'écrire sous la forme d'un deuxième état initial dans la zone de départ.

	Testeur 1	Testeur 2	Testeur 3
Exercice 1	V	X	V
Exercice 2	V	V	V
Exercice 3	X	X	V

FIGURE 5.14: Résultats des tests.

Nous obtenons après la découverte de l'interface et l'explication nécessaire à la construction d'une interface multimodale une majorité de réussite dans les tests.
La partie la plus difficile ressentie par les testeurs est la gestion du contexte et son intégration dans la machine à états.
Le testeur 2 a eu une difficulté lors de la création de la première interaction car les explications données par l'évaluateur n'ont pas été assez claires sur la manière dont se finalisait l'interaction. Mais après réexplication du principe, le testeur n'a pas éprouvé d'obstacles lors de la création du deuxième système.

CHAPITRE

6

CONCLUSIONS ET TRAVAUX FUTURS

Au fil de ce travail, nous avons fait avancer et évoluer le langage SMUIML.
Nous avons comblé certains manques relevés par ses détracteurs afin de rendre le langage plus fonctionnel et plus facile d'utilisation.
Du modèle CARE, nous sommes passés à une toute nouvelle architecture MMMM. Cette évolution technique non négligeable et s'attaquant aux fondements même de la structure de SMUIML augmente sa flexibilité et permet une meilleure accessibilité du langage pour le développement d'interaction multimodales.
Nous avons créé une gestion des erreurs, permettant à l'utilisateur final de recevoir un feedback utile sur la réussite ou non de son interaction.
Nous avons donné une définition aux modalités. Grâce à cela, le système de reconnaissance se fait plus aisément et de plus nous avons intégré une gestion du contexte dans le langage. Tout ceci a été réalisé dans le but d'enrichir les interactions et d'obtenir une réponse plus précise et contextuelle de la part de la machine.
Les triggers conditionnels peuvent également être associés à cet enrichissement. Grâce à eux l'utilisateur peut disposer d'un panel de triggers différents.
La nouvelle implémentation graphique ouvre la porte à un renouveau d'HephaisTK pour plus tard aboutir à une véritable interface graphique tout à fait fonctionnelle.
Même s'il reste encore des pistes à explorer pour améliorer SMUIML, comme par exemple, la gestion des modalités en sortie ou la compatibilité de SMUIML et d'HephaisTK avec des OS embarqués, le travail effectué a permis de rendre SMUIML plus accessible aux développeurs et utilisateurs, ils ont désormais une panoplie d'outils à leur disposition pour pouvoir créer des systèmes complets.
SMUIML est également plus efficace dans la prise en charge des modalités d'entrées grâce à une augmentation de la précision dans la reconnaissance des informations envoyés par l'utilisateur. La contextualisation de l'interaction augmente encore cette précision en ciblant précisément le type de réponse que l'utilisateur attend de l'ordinateur.
SMUIML a franchi un pas en s'adaptant à une manière différente de penser les interactions multimodales et ces modifications n'ont fait qu'augmenter son efficacité à répondre aux besoins utilisateurs.

BIBLIOGRAPHIE

[1] Bruno Dumas. Ihdc030 interaction homme/machine. In *Cours donné en Faculté d'Informatique à l'Université de Namur*, 2015.

[2] Laurence Nigay and Joëlle Coutaz. A design space for multimodal systems : concurrent processing and data fusion. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 172–178. ACM, 1993.

[3] Bruno Dumas, Jonathan Pirau, and Denis Lalanne. Modelling fusion of modalities in multimodal interactive systems with mmmm. PReCISE Research Center, université de Namur, Belgium, 2017. Article en cours de soumission.

[4] Max Tallier. Modélisation de l'interaction multimodale : les langages de description. 2016. Projet personnel dans le cadre de la troisième année baccalauréat de la Faculté d'Informatique à l'Université de Namur.

[5] Bruno Dumas, Denis Lalanne, and Rolf Ingold. Description languages for multimodal interaction : a set of guidelines and its illustration with smuiml. *Journal on multimodal user interfaces*, 3(3) :237–247, 2010.

[6] Bruno Dumas, Beat Signer, and Denis Lalanne. A graphical uidl editor for multimodal interaction design based on smuiml. In *Proceedings of the workshop on software support for user interface description language*, 2011.

[7] Fredy Cuenca, Jan Van den Bergh, Kris Luyten, and Karin Coninx. Hasselt uims : a tool for describing multimodal interactions with composite events. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 226–229. ACM, 2015.

[8] 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 *Human—Computer Interaction*, pages 115–120. Springer, 1995.

[9] Richard A Bolt. *"Put-that-there" : Voice and gesture at the graphics interface*, volume 14. ACM, 1980.

[10] Bruno Dumas, Denis Lalanne, and Sharon Oviatt. Multimodal interfaces : A survey of principles, models and frameworks. *Human machine interaction*, pages 3–26, 2009.

[11] Jean-Claude Martin. Tycoon : Theoretical framework and software tools for multimodal interfaces. *Intelligence and Multimodality in Multimedia interfaces*, pages 1–25, 1998.

[12] Stéphane Sire and C Chatty. The markup way to multimodal toolkits. In *W3C multimodal interaction workshop*, 2002.

[13] Robert Dale, Sabine Geldof, and Jean-Philippe Prost. Using natural language generation in automatic route. *Journal of Research and practice in Information Technology*, 37(1) :89, 2005.

[14] Eric Barboni, Célia Martinie, David Navarre, Philippe Palanque, and Marco Winckler. Bridging the gap between a behavioural formal description technique and a user interface description language : Enhancing ico with a graphical user interface markup language. *Science of Computer Programming*, 86 :3–29, 2014.

[15] Arnaud Hamon-Keromen. *Définition d'un langage et d'une méthode pour la description et la spécification d'IHM post-WIMP pour les cockpits interactifs.* PhD thesis, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2014.

[16] Nadia Elouali, Jean-Claude Tarby, Xavier Le Pallec, and José Rouillard. Approche idm pour le développement d'applications mobiles multimodales. In *9ème édition de la conférence MAnifestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication-MajecSTIC 2012 (2012)*, 2012.

[17] Fredy Cuenca Lucero. *Towards a composite event-based language for describing multimodal interactions.* PhD thesis, 2016.

[18] Bruno Dumas. *Frameworks, description languages and fusion engines for multimodal interactive systems.* PhD thesis, Faculty of Science, University of Fribourg (Switzerland, 2010.

[19] Chris Branton, Brygg Ullmer, Andre Wiggins, Landon Rogge, Narendra Setty, Stephen David Beck, and Alex Reeser. Toward rapid and iterative development of tangible, collaborative, distributed user interfaces. In *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 239–248. ACM, 2013.

[20] Bruno Dumas, Denis Lalanne, and Rolf Ingold. Hephaistk : a toolkit for rapid prototyping of multimodal interfaces. In *Proceedings of the 2009 international conference on Multimodal interfaces*, pages 231–232. ACM, 2009.

[21] Bruno Dumas, Beat Signer, and Denis Lalanne. Fusion in multimodal interactive systems : an hmm-based algorithm for user-induced adaptation. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 15–24. ACM, 2012.

[22] Jean-François Ladry. *Une notion et un processus outillé pour le développement de systèmes interactifs multimodaux critiques.* PhD thesis, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2010.

[23] Philipp Neumeier. Prototyping toolkits. *Prototyping for the digital city*, 2014.

[24] David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. Icos : A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(4) :18, 2009.

[25] Bruno Dumas. Infom435 méthodes d'interaction avancées. In *Cours donné en Faculté d'Informatique à l'Université de Namur*, 2016.

[26] F Buschmann, R Meunier, H Rohnert, and P Sommerlad. Stal, and m.(1996) :"pattern-oriented software architecture : A system of patterns".

[27] Olivier Carton. Interfaces graphiques en licence d'informatique-architecture modèle/vue/contrôleur, 2003-204.

CHAPITRE

7

APPENDICES

1 Tableau B. Dumas

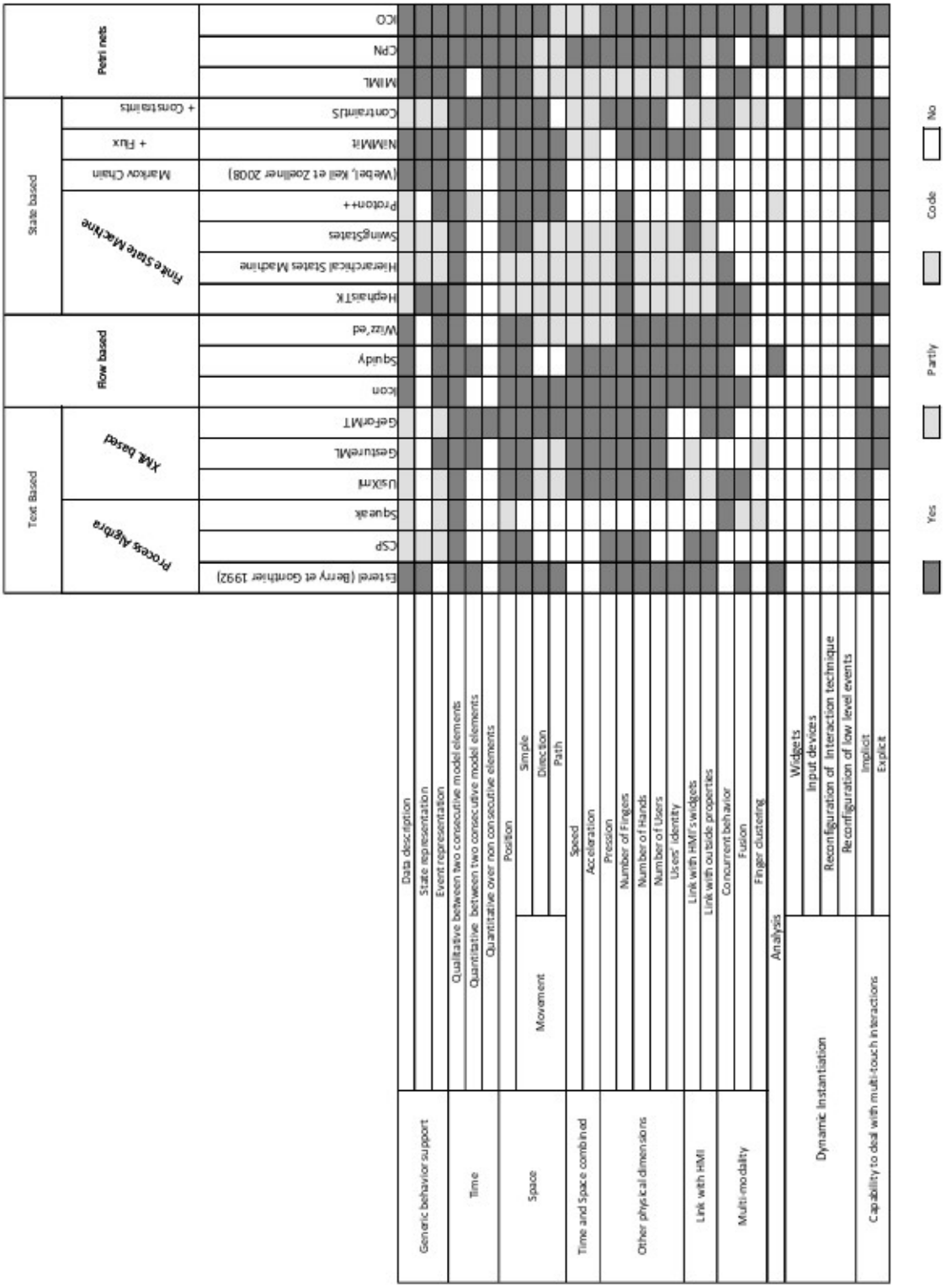
	Layers	Events	Time	Plasticity	Web-oriented	Error handling	Data Modeling
EMMA							X
XISL		X	X		X		
ICO		X	X			X	X
UsiXML	X	X		X	X		
TeresaXML	X	X					
MIML	X				X		
NiMMiT	X	X	X				

Description languages for multimodal interaction : a set of guidelines and its illustration with SMUIML- B. Dumas

Table 1
UIDLs overview.

				Interaction Coverage			Tool support	Expressiveness						
				Other										
Underlying notation	Language	Reference to the first paper	Post-WIMP specific (Low Level, Multi-modality, Tangible, Fusion)	Widget	Rendering	Dialogue		Data Description	State Representation	Event Representation	Time quantitative	Time Qualitative	Concurrent Behaviour	Dynamic Instantiation
Constraints	Coral	[58] : Szekely et al. 1988	(N, Y, N, N)	Y	Code	N	N	Code	N	N	N	Y	N	N
	Apogee	[30] : Hudson 1989	(N, N, N, N)	Y	Y	N	Y	Code	N	N	N	N	N	N
Code-based	Squeak	[17] : Cardelli et al. 1985	(Y, Y, N, N)	Y	Code	Code	N	Code	N	Code	N	Y	Y	N
	CSP	[56] : Smith et al. 1999	(N, Y, N, Code)	N	Code	N	N	Code	Code	Code	N	Y	Y	N
	DMI	[2] : Anson 1982	(Y, Y, N, N)	N	N	N	Y	Code	Code	Code	N	Y	N	N
	Subarctic	[31] : Hudson et al. 2005	(N, N, N, N)	N	Code	N	Y	N	Code	Code	N	Y	N	N
	XSL	[35] : Katsurada et al. 2003	(N, Y, N, N)	N	Code	Code	Y	Code	N	Code	N	Y	Y	N
	UsiXML	[40] : Limbourg et al. 2004	(N, Y, Y, N)	Y	Code	Code	Y	Code	N	Code	N	Y	Y	N
Flow-based	GAVUIMS	[53] : Silbert et al. 1986	(N, Y, N, N)	Y	Code	N	Y	Y	N	Code	N	Y	Y	N
	Tatsukawa	[59] : Tatsukawa 1991	(Y, N, N, N)	N	N	N	N	Y	N	Y	N	Y	Y	N
	Marigold	[63] : Willans et al. 2001	(Y, N, N, N)	N	N	N	Y	Y	Y	Y	N	Y	Y	N
	Wizzed	[24] : Esteban et al. 1995	(Y, N, N, Code)	Y	Code	N	Y	Y	N	Y	N	Y	Y	N
	ICON	[22] : Dragicevic et al. 2004	(Y, Y, N, Code)	N	Code	N	Y	Y	N	Y	N	Y	Y	N
State-based	Swingstates	[3] : Appert et al. 2006	(N, N, N, N)	Y	Code	N	Y	N	Code	Code	N	Y	N	N
	Hierarchical	[13] : Blanch et al. 2006	(Y, Y, N, Code)	Y	N	N	Y	Code	Code	Code	N	Y	Y	N
	3-State Model	[15] : Buxton 1990	(N, N, N, N)	N	N	N	N	N	Y	Y	N	Y	N	N
	GIN	[37] : Kienas et al. 1983	(N, N, N, N)	N	N	N	N	N	Y	Y	N	Y	N	N
	Hephais TK	[23] : Dumas et al. 2008	(Y, Y, Y, Code)	N	Code	Y	Y	Code	Y	Y	N	Y	Y	N
	IDG	[18] : Carr 1994	(Y, N, N, N)	Y	N	N	Y	Y	Y	Y	N	Y	N	N
	Shadow	[33] : Jacob 1986	(Y, Y, Y, N)	Y	Code	Y	Y	Y	Y	Y	N	Y	Y	N
	NiMMiT	[19] : Conins et al. 2007	(Y, Y, Y, Code)	N	Code	Y	Y	Y	Y	Y	N	Y	Y	N
Petri nets	Hinckley	[29] : Hinckley et al. 1998	(Y, Y, N, Y)	N	N	N	N	N	Y	Y	N	Y	Y	N
	OSU	[36] : Keh et al. 1991	(Y, N, N, N)	N	Code	Y	Y	Y	Y	Y	N	Y	Y	N
	MIML	[39] : Latoschik 2002	(N, Y, Y, Code)	N	Code	Y	Y	Y	Y	Y	N	Y	Y	N
	ICO	This Paper	(Y, Y, N, Y)	Y	Code	Y	Y	Y	Y	Y	Y	Y	Y	Y

3 Tableau A. Hamon



Définition d'un langage et d'une méthode pour la description et la spécification d'IHM post-WIMP pour les cockpits interactifs- A. Hamon

4 Tableau N. Elouali et al.

	DynaMo	HephaisTK	Squidy	i*Chameleon	CrossWeaver
Puissance d'expression (propriétés CARE/ niveau de détail)	C, A, R, E / élevé (bas niveau)	C, A, R, E /moyen	A / en fonction du zoom	A / très faible	C, A, E / très faible
Audience	Informaticien	Informaticien	Informaticien	Utilisateur final	Utilisateur final
Logique de combinaison	Moteur de fusion (algo. à base de frames [9])	Moteur de fusion (algo. à base de frames + algorithme à base de HMM (Hidden Markov models) [7])	Pas de fusion	Pas de fusion	Moteur de fusion (algo. à base de frames)
Support logiciel	Mono-plateforme	Mono-plateforme	Multi-plateforme	Multi-plateforme	Multi-plateforme

Approche IDM pour le développement d'applications mobiles multimodales- N. Elouali

5 Exemples de définition de modalités

```
<!--La portée de définition d'une modalité va dépendre du type de modalité que l'on utilise.
 De manière générale, nous déterminerons une définition par défaut avec un minimum de paramètres
 afin que la modalité puisse être reconnue.
 Pour une application plus précise, il faudra fournir un niveau de définition des modalités
 suffisant pour remplir les conditions demandées.
 Pour les caractéristiques expert, données par texte libre créés par l'utilisateur
 Intégrer la notion de contexte à la définition de la modalité
 aller voir le cours master-->

<Modality_Definition>

    <modality/>
        <device/>
        <representation/>

    <modality name='mouse'>


    <modality name='gesture'>
        <device name='video_input'>
        <default>
            <representation>
            <signal>
                <type_movement>
                    <body_part name='head'>
                        <movement name='shake'/>
                        <movement name='tilted'/>
                        <movement name='rotate'/>
                    <body_part/>

                    <body_part name='right_hand'>
                        <movement name='left_sweep'/>
                        <movement name='right_sweep'/>
                        <movement name='shake'/>
                        <movement name='pointing'/>     <!-- vs digit-->
                    <body_part/>

                    <body_part name='left_hand'>
                        <movement name='left_sweep'/>
                        <movement name='right_sweep'/>
                        <movement name='shake'/>
                        <movement name='pointing'/>     <!-- vs digit-->
                    <body_part/>
                </type_movement>

                <type_position>
                    <body_part name='head'>
                        <position name='upward'/>
                        <position name='downward'/>
                        <position name='leftward'/>
                        <position name='rightward'/>
                    <body_part/>
                </type_position>

            </signal>

            <context>
                <type_movement>
                    <body_part name='eye'>
                        <movement name='look_left'/>
                        <movement name='look_right'/>
                        <movement name='look_up'/>
                        <movement name='look_down'/>
                    </body_part>
                </type_movement>

            </context>
            </representation>
        </default>
```

```
70          <expert>
71              <movement_recognition>
72              <!-- reconnaissance de mouvements particuliers de l'utilisateur-->
73              </movement_recognition>
74          </expert>
75          </device>
76      </modality>
77
78      <modality name='sound'>
79          <device name='micro_input'/>
80          <default>
81              <representation>
82              <signal>
83                  <type_sound>
84                  <sound name='speech'>
85                      <value>
86                      <!--Ordre vocal-->
87                      </value>
88                  </sound>
89                  <sound name='whistle'>
90                      <value>
91                      <!--reconnaissance des différents sifflements-->
92                      </value>
93                  </sound>
94                  <sound name='yell'>
95                      <value>
96                      <!--Hurlement-->
97                      </value>
98                  </sound>
99                  </type_sound>
100
101                 <type_other>
102                 <other name='blowing'>
103                     <value>
104                     <!--Soufflement dans le micro-->
105                     </value>
106                 </other>
107                 </type_other>
108             </signal>
109
110             <context>
111                 <type_sound>
112                 <sound name='vehicule_noise'>
113                     <value>
114                     <!--Bruit de véhicule-->
115                     </value>
116                 </sound>
117                 <sound name='urban_noise'>
118                     <value>
119                     <!--Bruit venant de la ville-->
120                     </value>
121                 </sound>
122                 <sound name='farmer_noise'>
123                     <value>
124                     <!--Meuglement-->
125                     </value>
126                 </sound>
127                 </type_sound>
128             </context>
129             </representation>
130         </default>
131         <expert>
132             <voice_recognition>
133             <!-- reconnaissance de la voix de l'utilisateur-->
134             </voice_recognition>
135         </expert>
136         </device>
137     </modality>
138
139     <modality name='stylus'> <!--plus large que seulement écriture (dessin,
        pointing,..)-->
140     <device name='video_stylus_input'/>
```

```
        <default>
            <representation>
            <signal>
                <type_written>
                <written name='writing'>
                    <value>
                    <!--écriture-->
                    </value>
                </written>
                <written name='draw'>
                    <value>
                    <!--dessin-->
                    </value>
                </written>
                <written name='graphic'>
                    <value>
                    <!--graphique-->
                    </value>
                </written>
                </type_written>
            </signal>

            <context>
                <type_written>
                <writen name='trembling'>
                    <value>
                    <!--Tremblement utilisateur-->
                    </value>
                </written>
                </type_written>
            </context>

    </expert>
</Modality_Definition>
```

6 Exemple de Put-That-There

```
<?XML version= "1.0" encoding="UTF-8"?>
<smuiml>
    <integration_description client="Put_that_there">
        <recognizers>
            <recognizer name="speech" modality="speech"/>

            <recognizer name="pointing" modality="movement">
                <variable name="posx" value="x" type="int"/>
                <variable name="posy" value="y" type="int"/>
            </recognizer>
        </recognizers>


        <triggers>
            <userTrigger>
            <DefaultTrigger>
                <utrigger name="put">
                    <source modality="speech" value="put"/>
                    <validation name="put_ok" value="true"/>
                </utrigger>

                <utrigger name="that">
                    <source modality="speech" value="that">
                        <variable name="posx" value="x" type="int"/>
                        <variable name="posy" value="y" type="int"/>
                    </source>
                    <validation name="that_ok" value="true"/>
                </utrigger>

                <utrigger name="there">
                    <source modality="speech" value="there"/>
                        <variable name="posx" value="x" type="int"/>
                        <variable name="posy" value="y" type="int"/>
                    </source>
                    <validation name="there_ok" value="true"/>
                </utrigger>

                <utrigger name="point_trigger">
                    <source modality="pointing">
                        <value="x"/>
                        <value="y"/>
                    </source>
                    <validation name="pointing_ok" value="true"/>
                </utrigger>
            </DefaultTrigger>


            <ConditionnalTrigger name="put_test">
                <condition variable 'speechContent' test= '==put'/>

                <condition variable 'pointingContent' element= 'OK_cursor' test=
                '==pointing_OK' />

            </ConditionnalTrigger>

            </userTrigger>


            <systemTrigger>

                <strigger name="put">
                    <req utrigger 'put'/>
                </strigger>

                <strigger name="that">
                    <complement>
                    <TW value='1500'/>
                    <req utrigger 'that'/>
                    <req utrigger 'pointing_trigger'/>
                    <complement/>
                </strigger>

```

```
                <strigger name="there">
                    <complement>
                    <TW value='1500'/>
                    <req utrigger 'there'/>
                    <req utrigger 'pointing_trigger'/>
                    <complement/>
                </strigger>

            </systemTrigger>
        </triggers>


        <commands>
            <command name="put_that_there_action">
                <target name="put_that_there_client"
                    message=" change that_x=there_x"
                    message=" change that_y=there_y">
                </target>
        <commands/>


        <errors>
            <error name="put_error">
                <target name="put_that_there_client"
                    message=" Say Put">
                </target>
            </error>

            <error name="that_error">
                <target name="put_that_there_client"
                    message=" Say That">
                </target>
            </error>

            <error name="there_error">
                <target name="put_that_there_client"
                    message=" Say there">
                </target>
            </error>

            <error name="pointing_error">
                <target name="put_that_there_client"
                    message=" Bad pointing">
                </target>
            </error>
        </errors>


        <dialog>
            <context name="put_that_there">
                <transition_name="modification">
                        <strigger name="put">
                            <TTN value='1000'/>
                        </strigger>
                        <strigger name="that"/>
                            <TTN value='1000'/>
                        </strigger>
                        <strigger name="there"/>
                    <result command="put_that_there_action"/>
                    <result context="put_that_there"/>
                </transition>
            </context>

            <context name="put_error">
            <validation putOk value='false'/>
                <result error="putError"/>
            <context/>

            <context name="that_error">
            <validation that_ok value='false'/>
                <result error="that_error"/>
            <context/>
```

```
144
145             <context name="there_error">
146             <validation there_ok value='false'/>
147                 <result error="there_error"/>
148             <context/>
149
150             <context name="pointing_error">
151             <validation pointing_ok value='false'/>
152                 <result error="pointing_error"/>
153             <context/>
154         <dialog/>
155
156     </integration description client="Put_that_there">
157 </smuiml>
```

7 Exemple de Notebook

```
<?XML version= "1.0" encoding="UTF-8"?>
<NoteBook>
<integration_description client="NoteBook">

    <recognizers>
        <recognizer name="speech" modality="speech"/>

        <recognizer name="mouse" modality="mouse"/>

        <recognizer name="pointing" modality="movement">
                <variable name="posx" value="x" type="int"/>
                <variable name="posy" value="y" type="int"/>
        </recognizer>

    </recognizers>

    <triggers>

        <user_trigger>
        <utrigger name='clear'>
            <source modality='speech' value='clear all'/>
        </utrigger>

        <utrigger name='insert_note'>
            <source modality='speech' value='insert a note'/>
        </utrigger>

        <utrigger name="here">
            <source modality="speech" value="here">
                <variable name="posx" value="x" type="int"/>
                <variable name="posy" value="y" type="int"/>
            </source>
        </utrigger>

        <utrigger name="point_trigger">
            <source modality="pointing">
                <value="x"/>
                <value="y"/>
            </source>
        </utrigger>

        <utrigger name='note_position_voice'>
            <source modality='speech' value='before | between | after'/>
        </utrigger>

        <utrigger name='number'>
            <source modality='speech' value='first | first and | second | second and
            | third'/>
        </utrigger>

        <utrigger name='next'>
            <source modality='speech' value='next | next note'/>
        </utrigger>

        <utrigger name='click_insert'>
            <source modality='mouse' value='left button mouse click'/>
        </utrigger>

        <utrigger name='click_next'>
            <source modality='mouse' value='left button mouse click'/>
        </utrigger>

        </user_trigger>


        <sysTrigger>

            <strigger name='clear'>
            <unimod>
                <req utrigger 'clear'/>
            </unimod>
            </strigger>
```

```
            <target name="Matis" message="show_flight" />
        </command>
    </commands>

    <dialog>

        <context name="show_click">
            <complementary>
            <strigger name='show'/>
            <strigger name='click_airport'/>
            </complementary>
            <result action name='show_flight'/>
        </context>

        <context name="show_keyboard">
            <complementary>
            <strigger name='show'/>
            <strigger name='enter_keyboard'/>
            </complementary>
            <result action name='show_flight'/>
        </context>


    </dialog>

</integration description client="NoteBook">
</Notebook>

```

8 Exemple Matis

```
<?XML version= "1.0" encoding="UTF-8"?>
<Matis>
<integration_description client="Matis">

    <recognizers>
        <recognizer name="speech" modality="speech"/>

        <recognizer name="mouse" modality="mouse"/>

        <recognizer name="keyboard" modality="keyboard">


    </recognizers>

    <triggers>

        <user_trigger>
        <utrigger name='show'>
            <source modality='speech' value='show me'/>
        </utrigger>


        <utrigger name='flight'>
            <source modality='speech' value='flight | USAflight'/>
        </utrigger>

        <utrigger name='from'>
            <source modality='speech' value='from'/>
        </utrigger>

        <utrigger name='to'>
            <source modality='speech' value='to'/>
        </utrigger>

        <utrigger name='city'>
            <source modality='speech' value='Boston | Pittsburgh'/>
        </utrigger>

        <utrigger name='click_airport'>
            <source modality='mouse' value='left button mouse click'/>
        </utrigger>

        <utrigger name='enter_keyboard'>
            <source modality='keyboard' value='to ***'/>
        </utrigger>

        </user_trigger>


        <sysTrigger>

            <strigger name='show flight'>
            <complementary>
                <req utrigger 'show'/>
                <req utrigger 'flight'/>
            </complementary>
            </strigger>

            <strigger name='from *** to ***'>
            <complementary>
                <req utrigger 'from'/>
                <req utrriger 'city'/>
                <req utrriger 'to'/>
                <req utrriger 'city'/>
            </ucomplementary>
            </strigger>

        </sysTrigger>
    </triggers>

    <commands>
        <command name='show_me'>
```

```
            <target name="Matis" message="show_flight" />
        </command>
    </commands>

    <dialog>

        <context name="show_click">
            <complementary>
            <strigger name='show'/>
            <strigger name='click_airport'/>
            </complementary>
            <result action name='show_flight'/>
        </context>

        <context name="show_keyboard">
            <complementary>
            <strigger name='show'/>
            <strigger name='enter_keyboard'/>
            </complementary>
            <result action name='show_flight'/>
        </context>


    </dialog>

</integration description client="NoteBook">
</Notebook>

```

9 Méthodologie d'évaluation

Dans ce document, nous décrivons la manière de procéder à l'évaluation d'un système. Celui-ci est le résultat du travail effectué au cours de cette année dans le cadre du mémoire.

Méthodologie

Afin d'évaluer le travail effectué, nous allons utiliser la méthode quasi empirique d'évaluation rapide vue lors du cours d'interaction Homme-Machine.
Nous choisissons cette méthode car elle est plus aisée à mettre en oeuvre, au vu du peu de temps disponible pour cette évaluation.
Elle consiste à mettre 2 ou 3 utilisateurs devant un prototype du système, de les laisser se familiariser avec l'environnement, de les guider sur l'élaboration d'une tâche basique puis finalement les laisser effectuer seuls plusieurs autres tâches.
L'observateur doit alors prendre note des remarques et observations émises par les utilisateurs lors de l'utilisation du système.

Méthodologie appliquée

Dans le cadre de l'évaluation de ce travail, nous choisissons 3 personnes ayant des connaissances en informatique de niveaux différents (bas, intermédiaire et professionnel).
L'évaluation devra se passer dans un lieu calme sans source de distraction.
L'évaluation aura comme base le wireframe HepahisTiki (le nom peut être modifié), mais celui-ci n'étant utile que pour une présentation graphique de l'interface et de ses fonctionnalités, les testeurs devront réaliser l'exercice de création sur format papier. Des canevas de l'implémentation graphique leur seront fournis.
L'évaluateur devra fournir toutes les informations nécessaires liées aux fonctionnalités non implémentées du système.
Etapes de l'évaluation :

1. Mise en situation ;
   Explication des objectifs de l'évaluation aux testeurs.
2. Explication du concept de multimodalité ;
3. Présentation du wireframe HephaisTiki ;
   Approche de l'interface graphique et des différents éléments qui la composent.
4. Mise en situation guidée ;
   Après explication du cas Put-That-There, les testeurs devront créer à l'aide du wireframe cette interaction simple. L'évaluateur aura pour rôle de guider les testeurs dans la réussite de cette exemple.
5. Mise en situation non guidée ;

   Les testeurs vont devoir créer plusieurs interactions sans aide de la part de l'évaluateur.

Comme mise en situation, nous choisissons le développement d'interaction

entre l'utilisateur et son smartphone.

Smartphone :
- Allumer son téléphone en disant "Salut R2" ou en claquant des doigts ou en appuyant sur le bouton de démarrage ;
- Ouvrir une application en appuyant sur le bouton démarrage et en disant "ouvre app" ;
- Lancer une application musicale en appuyant sur l'icône de l'application et en disant le nom de la musique voulue et si le volume est trop bas, le mettre en position moyenne.

Le testeur devra indiquer la marche à suivre pour la création de ses différents triggers puis devra créer la machine à états de l'interaction.
Un des exemples mais en place la notion de contexte et la création d'un deuxième chemin pour la réalisation de l'interaction.

L'évaluation se fera par l'observation des sujets et leur niveau de compréhension du système.

Points d'attention de l'évaluateur

1.Aspect graphique
- Appréciation de l'aspect général de l'interface
- Visibilité et manipulation des différentes éléments
- Compréhension de la signification des éléments
- Utilisation de l'aide
- Utilisation de l'assistant

2. Aspect pratique dans la construction de la multimodalité
- Gestion des différents éléments
- Construction modalité
- Construction trigger
- Construction contexte
- Construction d'une interaction totale menant à la commande

3. Aspect compréhension du système multimodal
- Place de l'utilisateur au sein de l'architecture
- Accessibilité et facilité d'utilisation
- Compréhension de la notion de temporalité dans la combinaison des modalités
- Compréhension du concept de contexte

4. Scénario
- Compréhension et construction d'un scénario avec aide (Put-that-there)
- Compréhension et construction d'un scénario sans aide (Smartphone)

5. Remarques diverses faites par testeurs