ÉCOLE
POLYTECHNIQUE
DE LOUVAIN

# Mémoire :
## Analyse comparative de formalismes d'interfaces homme-machine


Mémoire présenté par
**Simon Goffin**
en vue de l'obtention du titre de
**Master 120 crédits en ingénieur civil en informatique**
**option applied mathematics**

**Lecteur :**
Sébastien Combéfis

**Promoteurs :**
Charles Pecheur
Jean Vanderdonckt



Année académique 2011-2012

Je tenais à remercier mes promoteurs Charles Pecheur et Jean Vanderdonckt ainsi que mon assistant Sébastien Combéfis pour leur implication et leur aide tout au long de mon travail.
Merci à toute ma famille.


UCL
Université
catholique
de Louvain

# Table des matières

# 1 Introduction

## 1.1 Contexte

Dans le monde moderne qui foisonne d'applications technologiques, l'homme est amené à interagir de plus en plus avec des machines, certaines dans des situations simples (distributeurs bancaires, lecteurs DVD,...) et d'autres dans des situations plus complexes (smartphones, équipement médical, système d'auto-pilotage d'avion, ... ). Ces interactions ont lieu par le biais d'interfaces qui servent de lien entre l'homme et la machine. Lors de la conception d'un système interactif, ces interactions homme-machine requièrent l'utilisation d'outils de conceptualisation spécifiques. C'est ainsi que les interactions homme-machine pourront être modélisées et analysées par des formalismes abstraits capables de décrire la structure des dialogues entre un utilisateur et le système interactif que nous souhaitons conceptualiser. Aujourd'hui de nombreuses familles de modèles formels ont été développées pour modéliser des systèmes interactifs et chacune d'entre elles présente une approche singulière permettant ainsi d'analyser et de décrire le fonctionnement de ces systèmes sous différents angles et dans les situations les plus diverses.

## 1.2 L'objectif du travail

Ce travail s'est intéressé aux formalismes utilisés lors de la modélisation d'un système interactif. Un système interactif est un système qui se compose d'un noyau fonctionnel définissant le fonctionnement interne du système et d'une enveloppe extérieure permettant de refléter l'état interne du système ainsi que de définir les interactions possibles entre l'utilisateur et le noyau fonctionnel. Cette enveloppe extérieure est appelée *Interface Homme Machine* (IHM) et peut être vue comme une encapsulation du noyau fonctionnel.

En génie logiciel, la phase de modélisation est une phase cruciale dans la conception d'un système, car c'est précisément sur base de ce modèle que va s'élaborer toute la conception du système. La particularité de la modélisation d'un système interactif est qu'elle s'intéresse à l'aspect textuel ou graphique, aux comportements (les transitions et les états du système) et aux effets de bord du système interactif (interface monomodale ou multimodale). La modélisation d'un système interactif ne s'intéresse pas au fonctionnement interne du système avec lequel l'interaction est effectuée. Ainsi, le résultat de cette activité est un modèle décrivant précisément les interactions attendues entre l'utilisateur et le système interactif, c'est-à-dire l'IHM ; c'est pourquoi nous parlons de modélisation d'IHM et de formalismes d'IHM pour désigner la modélisation d'un système interactif.

L'objectif de ce travail est de créer une grille de critères permettant de comparer des formalismes d'IHM dans le but de déterminer celui qui est le plus adéquat pour modéliser un système interactif en fonction de son IHM. Le travail se divise en trois grandes parties.

La première partie du travail a consisté à déterminer les besoins de modélisation des différentes classes d'IHMs ainsi que de catégoriser les formalismes en différentes familles. Sur base de cette catégorisation et des besoins des différentes classes d'IHMs, nous avons

établi une grille de critères permettant de comparer les formalismes.

La deuxième partie du travail a consisté à choisir trois formalismes afin de les évaluer sur base des critères établis dans la première partie. On verra que l'évaluation d'un formalisme se divise en deux étapes :
- La première étape consiste à analyser le formalisme de manière théorique, c'est-à-dire que cette analyse se base sur notre grille de critères et sur les ressources disponibles au sujet du formalisme analysé (livres, publications et outils). Cette première étape s'apparente fortement à l'évaluation d'un formalisme quelconque que réalisera un utilisateur lambda sur base de notre grille de critères.
- La deuxième étape consiste à modéliser un système interactif avec le formalisme analysé afin de mettre en évidence l'utilité de nos critères lors de son évaluation théorique .

Et enfin, la troisième partie de ce travail s'est attachée à comparer les trois formalismes analysés sur base de leur évaluation respective afin de déterminer le plus adéquat pour modéliser un système interactif en fonction de son IHM.

L'originalité de ce travail se situe essentiellement à deux niveaux :
- catégoriser les IHMs en différentes classes afin de bien discerner les besoins de modélisation de chaque classe d'IHMs.
- déterminer l'influence d'un formalisme d'IHM sur les phases de conception d'un système interactif en vue de bien comprendre les objectifs remplis par chaque formalisme.

Sur base de ces deux nouvelles approches, nous déterminons les critères qui devraient nous permettre, à l'avenir, de distinguer les formalismes les plus appropriés pour modéliser un système interactif doté d'une interface particulière.

# 2 Les systèmes interactifs

## 2.1 Définition d'un système interactif et d'une interface homme-machine

Un système interactif au sens de D.Harel [1], c'est un système qui doit maintenir une interaction constante avec l'environnement, en produisant des résultats à chaque invocation. Ces résultats dépendent des données fournies par l'environnement et par l'utilisateur lors de l'invocation, ainsi que de l'état interne du système. De manière plus concrète, un système interactif se compose d'un noyau fonctionnel définissant le fonctionnement interne du système et d'une enveloppe extérieure permettant de refléter l'état interne du système et de définir les interactions possibles entre l'utilisateur et le noyau fonctionnel. Cette enveloppe extérieure est appelée Interface Homme Machine (IHM) et peut être vue comme une encapsulation du noyau fonctionnel . Sur la FIGURE 1, nous illustrons les échanges qu'il y a entre l'utilisateur et le système interactif.

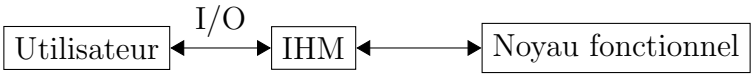



FIGURE 1: Système interactif

Les IHMs sont des médiateurs entre l'utilisateur et le système interactif permettant à l'utilisateur de transmettre des requêtes au noyau fonctionnel du système afin que celui-ci réalise les tâches invoquées. Une IHM est une partie cruciale d'un système interactif dans le sens où elle détermine l'utilisabilité du système car il est évident qu'une application performante disposant d'une interface inutilisable ne sera jamais employée. Une IHM fournit par le biais de ses entrées des moyens pour contrôler le système, et, par le biais de ses sorties, des moyens pour observer l'état du système.

Il y a différentes manières de percevoir les entrées et les sorties d'une IHM, comme par exemple, avec la vision et l'audition. Différents moyens d'interaction peuvent être combinés dans le même système, et pour la même tâche. Par exemple, un utilisateur peut voir les sorties de son système sur l'écran de son ordinateur et peut envoyer des entrées vers le système par des senseurs que sont le clavier, la souris ou un écran tactile.

Une IHM doit, de manière générale, refléter exactement et fidèlement le comportement de l'application (c'est l'aspect informatique). Elle doit faciliter son utilisation (c'est l'aspect ergonomique) et elle doit être accessible au plus grand nombre d'utilisateurs, y compris avec des comportements différents (c'est l'aspect psychologique). C'est pourquoi la conception d'une IHM fait intervenir non seulement des informaticiens pour la conception et le développement mais aussi des ergonomes pour le choix de l'ergonomie et de la disposition des éléments qui composent l'IHM, ainsi que des psychologues pour capturer les comportements des utilisateurs [2].

## 2.2 Classes d'IHMs

Dans ce travail, nous classerons une IHM en fonction de son style d'interaction et de ses modalités d'entrée et de sortie. Cette définition se justifie par le fait qu'une interaction se caractérise notamment par sa forme (type de modalité utilisée) et par son contenu (type d'information véhiculée).

La forme d'une interaction

La forme d'une interaction dépend, d'une part, de la manière dont les entrées sont communiquées au système interactif, c'est-à-dire les modalités d'entrée, et d'autre part, de la manière dont les sorties sont perçues par l'utilisateur, c'est-à-dire les modalités de sortie. Les modalités d'une IHM définissent la forme de l'interaction. Les IHMs se distinguent par la forme de leurs interactions en deux groupes, les interfaces *virtuelles*, favorisant des interactions visuelles, et les interfaces *physiques*, favorisant des interactions matérielles. Chacun de ces deux groupes présentent des avantages et des inconvénients. Par exemple, les interfaces virtuelles ont l'avantage d'être portables et malléables (exemple :clavier tactile sur l'écran d'un smartphone) mais fournissent généralement une moins bonne prise en main que les interfaces physiques (exemple :clavier physique), qui elles ont le désavantage d'être moins portables et malléables, de par leur aspect physique, mais fournissent généralement une meilleure précision. Parmi ces deux groupes d'interfaces, on peut également distinguer les interfaces monomodales et les intefaces multimodales. Comme illustré sur la FIGURE 2, nous considérons qu'une interface monomodale est une interface se composant d'une modalité d'entrée $M_e$ et d'une modalité de sortie $M_s$ et comme illustré sur la FIGURE 3, nous considérons qu'une interface multimodale est une interface se composant de plusieurs modalités d'entrée ($M_{e1}$, $M_{e2}$) et/ou de plusieurs modalités de sortie ($M_{s1}$, $M_{s2}$). Conceptuellement, on peut considérer que toutes les interfaces physiques peuvent être simulées par des interfaces virtuelles, c'est pourquoi dans ce travail nous ne distinguerons pour la forme de l'interaction que deux groupes d'IHMs : les IHMs monomodales virtuelles et les IHMs multimodales virtuelles.


FIGURE 2: Interface monomodale

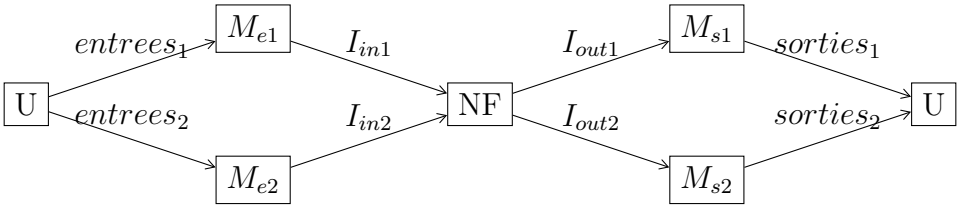



FIGURE 3: Interface multimodale

Le contenu d'une interaction

Comme illustré sur la FIGURE 2 et sur la FIGURE 3, une interaction *entrante* ($I_{in}$) se définit par une séquence d'entrées fournie par l'utilisateur (U), et reconnue par le système, sur laquelle le noyau fonctionnel du système (NF) va réagir et produire une interaction *sortante* ($I_{out}$). Celle-ci est constituée d'une séquence de sorties, et de préférence, reconnue par l'utilisateur. La complexité d'une interaction dépend essentiellement de la taille que nécessite une séquence d'entrées ou de sorties pour être reconnue respectivement comme une interaction entrante par le système ou comme une interaction sortante par l'utilisateur.

Une interaction est dite *simple* si la plus grande séquence d'entrées ou de sorties qui peut être reconnue comme une interaction respectivement par le système ou par l'utilisateur ne dépasse pas un certain seuil de tolérance fixé par l'homme.

De même qu'une interaction est dite *complexe* si la plus grande séquence d'entrées ou de sorties qui peut être reconnue comme une interaction respectivement par le système ou par l'utilisateur dépasse ce seuil de tolérance fixé par l'homme.

Partant de cette idée de complexité d'interaction, deux styles d'interface virtuelle ont vu le jour, les *interfaces graphiques utilisateur* (GUIs) privilégiant les interactions simples et les *interfaces en lignes de commande* (CLIs) privilégiant les interactions complexes. Un exemple pour illustrer ces deux styles d'interaction consiste à déterminer la plus grande séquence d'entrées possible permettant à une GUI et à une CLI de déterminer le chemin permettant d'accéder à un répertoire $R_3$ à partir du répertoire courant $R_1$. Pour une GUI, cela consiste à cliquer successivement sur les répertoires définissant le chemin vers le répertoire $R_3$. La GUI définit dès lors une succession d'interactions simples permettant d'accéder au répertoire $R_3$. Pour une CLI, cela consiste à taper le chemin $R_1/R_2/R_3$ permettant de passer directement du répertoire courant $R_1$ au répertoire $R_3$. La CLI définit dès lors une interaction complexe permettant d'accéder au répertoire $R_3$. Nous avons illustré ces deux situations sur la FIGURE 4 et sur la FIGURE 5.

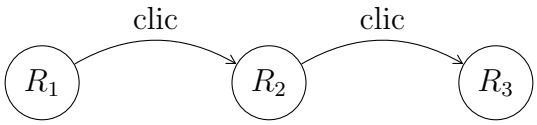



FIGURE 4: Interactions avec une GUI

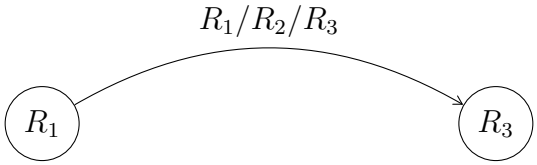



FIGURE 5: Interactions avec une CLI

C'est pourquoi le graphe défini par la relation d'accessibilité d'un répertoire depuis n'importe quel autre répertoire en une interaction est un graphe complet pour les CLIs et un arbre pour les GUIs. Représenter une CLI par une GUI serait complètement inadéquat dans le sens où dans une même fenêtre nous devrions représenter tous les répertoires accessibles en une interaction, c'est-à-dire tous.

Les classes

Sur base de la forme et du contenu d'une interaction, nous distinguerons trois classes d'IHMs, les CLIs monomodales, les GUIs monomodales et les les interfaces multimodales composées de GUIs et de CLIs.

## Les interfaces en ligne de commande (CLI)

Les interfaces utilisateurs en ligne de commande sont des exemples d'interfaces utilisateurs synchronisées et séquencées. Le dialogue entre le système et l'utilisateur est établi comme une séquence de questions et de réponses. A chaque pas d'exécution, le système attend la commande demandée par l'utilisateur, l'exécute, écrit la sortie et se déplace vers le pas d'exécution suivant. Il faut remarquer qu'à chaque pas d'exécution le système est dans un certain état défini par les pas d'exécution précédents et que à chaque ligne de commande invoquée, le système peut passer à un état suivant. Un exemple de ce type d'interfaces est le Unix Shell [3].

## Les interfaces graphiques (GUI)

Les interfaces utilisateur graphiques sont plus riches que les interfaces en ligne de commande dans le sens où elles peuvent supporter d'autres types d'interactions comme la forme "fill-in" avec des cases spécifiques permettant de communiquer des entrées, la forme menu de sélection avec des menus déroulants et la forme manipulation directe avec, par exemple, un panneau de contrôle virtuel [4]. Une GUI peut avoir plusieurs fenêtres sur le même écran avec des objets interactifs [1],comme des menus et des boutons entremêlés avec du texte dans une apparence graphique qui crée un environnement plus plaisant qu'un terminal textuel. Les fenêtres peuvent ainsi changer parmi plusieurs tâches ou plusieurs parties d'une même tâche. Typiquement l'utilisateur peut recourir à sa souris pour pointer et sélectionner une commande depuis le menu plutôt que de taper des lignes de commandes équivalentes dans un certain langage. Les GUIs permettent de cliquer sur des boutons, de sélectionner des objets, ainsi que de prendre et déposer des objets. Ces manipulations évitent de toujours devoir taper du texte à chaque interaction avec le système ce qui rend celle-ci beaucoup plus intuitive. Quand on interagit avec des GUIs, l'ordre dans lequel on réalise des tâches peut ne pas avoir d'importance dans le sens où, par exemple, un utilisateur peut réaliser une tâche $T_1$, l'interrompre, passer à une tâche $T_2$ pour consulter des informations et enfin terminer $T_2$ pour ensuite réactiver $T_1$. Le concept de "multi-threaded dialog" est utilisé dans ce genre d'interactions [3]. Il est important de remarquer que l'état interne d'un système interactif utilisant une GUI est représenté par

1. que nous appellerons *widgets*

la configuration et la disposition des *widgets* (objets interactifs). À chaque manipulation d'un widget, l'état interne du système peut changer. Il existe différents types de GUI tel que "hypertext", "web-based", "direct-manipulation", "rich client", ect.

### Les interfaces multimodales (IHM3)

Les systèmes multimodaux présentent certaines particularités difficiles à modéliser, comme par exemple, la mise en relation des différents médias d'entrée avec les autres composants logiciels. Dans un système conventionnel, les entrées possibles sont simples (utilisation du clavier et de la souris) et souvent standard (clic,double-clic,...). Dans le cas d'un système multimodal, les entrées sont multiples et complexes (sous forme textuelle, auditive, tactile, ect) et il est rapidement nécessaire de représenter la façon dont ces entrées sont connectées au reste de l'application. La concurrence joue également un rôle essentiel (les médias d'entrée et de sortie peuvent être utilisés en même temps, et donc fournir des informations au même moment), et il est aussi indispensable de pouvoir la modéliser. De façon générale, ces problèmes reviennent à savoir modéliser, d'une part, des relations temporelles complexes, comme le parallélisme, l'attente, la synchronisation et d'autre part la fusion d'informations provenant des médias en entrée et la fission de l'information en sortie. Les interfaces multimodales peuvent être vue comme une combinaison d'interfaces monomodales dont la synchronisation a lieu dans le noyau fonctionnel du système interactif [5]. Il est important de remarquer que l'état interne du système est fissionné par le noyau fonctionnel vers toutes les modalités de sortie afin que celles-ci retranscrivent cet état à l'utilisateur. À chaque manipulation d'une ou plusieurs modalités d'entrée de la part de l'utilisateur, les informations sont envoyées au noyau fonctionnel qui fusionne ces informations et décide ou non de passer à l'état suivant.

## 2.3 L'utilité de la modélisation d'un système interactif

L'utilité de la modélisation d'un système interactif est essentiellement de produire une description du système interactif en rapport avec les besoins de l'utilisateur (exigences). Cette description adopte un point de vue externe au système interactif lui-même. En effet, la modélisation s'intéresse à l'aspect [2], aux comportements [3] et aux effets de bord du système interactif [4], et non pas au fonctionnement interne du système avec lequel l'interaction est effectuée. Ainsi, le résultat de cette activité est un modèle décrivant précisément les interactions attendues de l'utilisateur avec le système interactif. Elle ne requiert pas nécessairement la présence de détails relatifs à son implémentation ou à sa réalisation.

En génie logiciel [13], la conception d'un système informatique se déroule en plusieurs phases. La première phase consiste à recueillir les exigences du client. Ensuite, sur base

2. CLI ou GUI
3. les transitions et les états du système
4. monomodale ou multimodale

de ces exigences, vient la phase de modélisation du système qui consiste à décrire les comportements des différentes entités du système : utilisateur, noyau fonctionnel et IHM. La troisième phase consiste à déterminer l'architecture du système sur base de la modélisation. La quatrième phase consiste à implémenter le système sur base de la modélisation des comportements et de l'architecture, et enfin, la dernière phase consiste à valider le système implémenté à l'aide de tests white box et black box. Comme on peut le voir sur la FIGURE 6, l'activité de modélisation est une phase définie en génie logiciel qui permet de concevoir de manière systématique un système.


FIGURE 6: Phases de conception

Notons que l'activité de modélisation d'un système interactif se démarque de l'activité de modélisation en génie logiciel classique, du fait que les interactions décrites dans les modèles réalisés, se concentrent sur les relations entre l'utilisateur et le système interactif, c'est-à-dire sur l'IHM. C'est pourquoi dans la suite de ce travail, nous analysons les formalismes permettant de modéliser des systèmes interactifs en fonction de l'IHM du système que nous voulons conceptualiser, et nous parlons de modélisation d'IHM pour désigner la modélisation d'un système interactif en fonction de son IHM.

Les modèles de conception de systèmes interactifs doivent également assurer l'utilisabilité du système, c'est-à-dire la facilité d'utilisation de l'IHM du système par un utilisateur donné ou une catégorie d'utilisateurs, ou de manière plus générale, son acceptabilité par les utilisateurs [2].

## 2.4 Les besoins de modélisation des IHMs

Après avoir défini l'utilité de la modélisation d'un système interactif, on peut déjà constater que la modélisation d'un système interactif requiert, de manière générale, la possibilité de définir l'état et les transitions d'états d'un système. Maintenant nous allons déterminer les besoins associés à un système interactif pour pouvoir être modélisé. Un système interactif se définira dans dans la suite de ce travail en fonction de son IHM. C'est pourquoi nous parlerons de modélisation d'une CLI, d'une GUI ou d'une IHM3 pour désigner la modélisation d'un système interactif disposant respectivement d'une de ces trois interfaces.

### Les besoins des CLIs

Les interfaces en ligne de commande ne présentent pas un aspect visuel très élaboré et nécessitent un langage textuel dont l'ordre d'apparition des caractères et des mots du langage ont une importance. C'est l'ordre d'apparition des caractères qui va précisément déterminer la ligne de commande invoquée. La modélisation des CLIs requiert la prise en

compte de chacun des caractères utilisés par la CLI ainsi que la possibilité d'établir un ordre sur l'apparition de ceux-ci dans une phrase. Les CLIs ne permettent généralement pas d'exécuter plusieurs lignes de commande simultanément, il faut à chaque fois, attendre que la commande invoquée se termine, avant de pouvoir en invoquer une autre, c'est pourquoi la prise en compte d'aspects temporels tels que la synchronisation, l'attente et la communication n'est pas nécessaire. Nous considérerons dans ce travail que les CLIs sont monomodales car les entrées sont communiquées au système par un clavier, et les sorties sont communiquées à l'utilisateur par un écran. Les entrées et les sorties généralement exprimées par des séquences de caractères sont deux caractéristiques qui doivent impérativement être prises en compte lors de la modélisation d'une CLI.

## Les besoins des GUIs

Les interfaces graphiques définissent l'état interne d'un système en déterminant un *agencement* particulier de ses widgets [5]. On entend par agencement la disposition des widgets sur la GUI mais également l'état respectif de chacun des widgets. L'état d'un widget est généralement défini par son apparence qui est, par exemple, grisée lorsque celui-ci est inutilisable, et, est en sur-impression, lorsque celui-ci est sélectionné. Les GUIs permettent d'interagir avec plusieurs widgets simultanément, c'est-à-dire que nous pouvons, par exemple, consulter plusieurs fichiers en même temps sans devoir à chaque fois refermer le fichier que nous sommes en train de lire pour en ouvrir un autre. Les besoins nécessaires pour la modélisation de GUI sont la possibilité de représenter plusieurs entités concurrentes (les widgets), la prise en compte de la concurrence vu que plusieurs widgets peuvent être sollicités en même temps, la possibilité de synchroniser ces widgets par des moyens de communication et la possibilité de décrire l'apparence de l'interface. Nous considérerons dans ce travail que les GUIs comme les CLIs sont des interfaces monomodales car les entrées sont communiquées au système par le périphérique clavier/souris, et les sorties sont communiquées à l'utilisateur par le biais d'un écran. Les entrées et les sorties faisant partie de la GUI, elles doivent également être prises en compte lors de la modélisation.

## Les besoins des IHM3s

Les interfaces multimodales permettent de communiquer les entrées utilisateur au système par le biais de plusieurs modalités telles que le périphérique clavier/souris, un micro, un écran tactile, un détecteur de mouvement, etc. Ces modalités d'entrée sont toutes reliées au noyau fonctionnel de l'application. Certaines de ces modalités d'entrée peuvent être utilisées simultanément comme c'est le cas dans l'application de Stéphane Chatty [26]. En effet S. Chatty propose à l'utilisateur de cliquer avec deux souris sur un bouton virtuel pour valider une commande critique. Lorsque deux modalités fonctionnent simultanément comme c'est le cas avec les deux souris, celles-ci envoient des informations simultanément au noyau fonctionnel, qui doit se charger ensuite de les fusionner afin de déterminer si les clics relatifs à chacune des souris ont été déclenchés aux mêmes impulsions d'horloge. On remarque dès lors que la synchronisation des modalités d'entrée a lieu dans le noyau fonctionnel du système interactif. Les interfaces multimodales permettent également de

5. Rappel : un widget est un élément visible permettant d'interagir avec la GUI

communiquer les sorties du système par le biais de plusieurs modalités telles que l'écran, les moniteurs audio, des LEDs, etc. Ces modalités de sortie sont également toutes reliées au noyau fonctionnel. Ces modalités de sortie peuvent fonctionner simultanément comme c'est le cas lorsqu'on reçoit un avertissement sonore et un avertissement visuel au même moment. À chaque impulsion d'horloge, l'état interne du système est fissionné par le noyau fonctionnel de l'application vers les différentes modalités de sortie. On peut voir une IHM3 comme une composition d'IHMs monomodales composées de GUIs et de CLIs orchestrées par le noyau fonctionnel, c'est pourquoi les besoins des CLIs et des GUIs sont aussi les besoins des IHM3s. Les besoins particuliers qui sont associés à la modélisation des IHM3s sont la prise en compte du noyau fonctionnel de l'application et la possibilité d'identifier le type de chaque entrée, et de chaque sortie du système.

# 3 Les formalismes de modélisation d'IHM

## 3.1 Les méthodes semi-formelles

Avant d'aborder les méthodes formelles qui permettent de modéliser les systèmes interactifs de manière rigoureuse, il est important de rappeler que le domaine des IHMs a vu naître de nombreux modèles et de notations semi-formels où le graphisme est un élément important. Ces modèles et notations, utilisés pour la conception ont été conçus pour être utilisés par des informaticiens mais également par des non-informaticiens tels que les ergonomes et les psychologues. Parmi ces différents modèles et notations, on distingue deux catégories, la première se compose de notations permettant de décrire des tâches (Task Model) et est essentiellement conçue pour les non-informaticiens. La deuxième se compose de modèles d'architecture permettant de faciliter le conception de systèmes interactifs (UIMS) conçus essentiellement pour les informaticiens. Ces modèles ont suivi les différentes évolutions matérielles et logicielles de ces dernières années [2].

### Les notations de tâches

Afin d'exprimer les exigences des utilisateurs, de nombreuses notations de description de systèmes interactifs ont été proposées. Elles sont pour la plupart, centrées sur les utilisateurs et sont donc loins des implantations informatiques. Par exemple, la notation Concurrent Task Trees (CTT) [6] met l'accent sur les activités de l'utilisateur sous la forme d'un arbre représentant les différentes tâches. La notation Concurrent Task Trees propose de nombreux opérateurs temporels permettant de décrire différents comportements associés au système interactif (interruption, concurrence, désactivation,...) mais elle ne permet pas d'effectuer de manière algorithmique la preuve de son exactitude. D'autres notations telles que GOMS ou UAN ont également été développées. Du point de vue du génie logiciel et du cycle de vie, ces notations sont, en général, le point de départ de la réalisation d'un système interactif nouveau [2]. Sur la FIGURE 7, nous avons illustré les phases de conception définies en génie logiciel et encadré la phase correspondant à l'utilisation des notations de tâches.


FIGURE 7: Phases de conception

### Les modèles d'architecture

Pour faciliter la conception de systèmes interactifs, des modèles d'architecture ont vu le jour. Ces différents modèles d'architecture permettent de modulariser le travail et séparent ainsi facilement la conception de l'IHM de la conception du noyau fonctionnel du système interactif. Ces modèles basés sur des architectures conceptuelles sont aussi appelés UIMS (User Interface Management Systems) et leur objectif est d'augmenter la portabilité (degré d'indépendance entre l'apparence et l'application sous-jacente) ainsi

que l'adaptabilité (capacité des systèmes à gérer des changements) des systèmes. Ces architectures peuvent se présenter sous la forme de couches ou de structures orientées objet. Ces modèles d'architecture sont en quelque sorte une première tentative pour rendre la conception de systèmes interactifs plus systématique [3]. Sur la FIGURE 8, nous avons illustré les phases de conception définies en génie logiciel et encadré la phase correspondant à l'utilisation des modèles d'architecture.

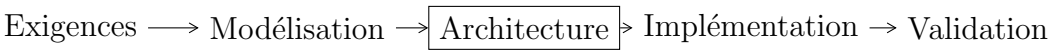



FIGURE 8: Phases de conception

Un exemple d'un de ces modèles d'architecture est le modèle Seeheim qui est inspiré de systèmes linguistiques. Il sépare le système en un aspect lexical, un aspect syntaxique et un aspect sémantique qui correspondent respectivement à la présentation (P), au dialogue (D) et à l'application de l'interface (AI). Ce modèle comporte également un contrôleur (C) qui permet de recevoir des messages de AI et D, et d'envoyer des messages vers P et D. La couche présentation décrit les objets interactifs et les données portées par ceux-ci. La couche de dialogue perçoit des données en entrée et détermine comment elles peuvent être traitées. L'application de l'interface (AI) décrit les services disponibles pour l'utilisateur. Sur la FIGURE 9, nous avons illustré le modèle Seeheim.

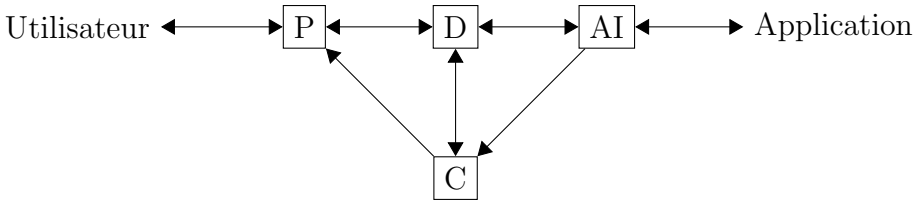



FIGURE 9: Modèle Seeheim

Ces modèles d'architecture utilisent une seule logique de décomposition : le noyau fonctionnel d'une part et l'IHM d'autre part. Ils ont l'avantage de préserver une forme de modularité dès la conception. Ce type d'approche présente les avantages d'une décomposition modulaire dans le sens où les différents composants peuvent être développés séparément et où des modifications peuvent être apportées ensuite à un moindre coût. Par contre les notations utilisées pour décrire la conception sont à la fois informelles et indépendantes des notations de description. De plus, la phase de vérification est réduite à la réalisation de tests manuels sans la moindre systémisation.

Du point de vue génie logiciel, les systèmes interactifs soulèvent deux points importants :
– d'une part, la majorité des systèmes interactifs, voir tous séparent le noyau fonctionnel de l'IHM. Cette séparation permet de structurer le logiciel en deux parties principales et elle est largement exploitée pour la validation et la vérification
– d'autre part, ce domaine s'appuie fortement sur l'existence de boîtes à outils que le programmeurs utilisent massivement pour la conception de systèmes interactifs. Ces

boîtes à outils existent sous forme de programmes, elles ne sont pas formellement spécifiées, ou tout au moins seules les API sont utilisées comme spécification [2].

## 3.2 Les méthodes formelles

Les différentes méthodes semi-formelles présentées plus haut ne nous conviennent pas car elles ne permettent pas de dériver du code, ni d'effectuer des vérifications sur les modèles qu'elles permettent de définir. Elles servent plus de recommandations aux développeurs que de modèles support de vérification. Afin de permettre la vérification des modèles, l'utilisation de méthodes formelles pour la conception de systèmes interactifs est préconisée.

### L'origine des méthodes formelles pour l'élaboration de systèmes interactifs

Pour vérifier que des programmes répondent bien aux exigences de l'utilisateur, on effectue lors de la validation du système des tests manuels (white-box et black-box) qui sont extrêmement coûteux. Ces tests se font à partir des spécifications (abstraites) et de l'implémentation (concrète) afin de détecter des incohérences. Les méthodes formelles ont été introduites dans la conception de systèmes informatiques pour minimiser les tests coûteux que nécessite la validation d'un système. Sur la FIGURE 10, nous avons illustré les phases de conception définies en génie logiciel et encadré la phase correspondant à l'utilisation de tests manuels coûteux.

Exigences ⟶ Modélisation → Architecture ⁃ Implémentation → Validation

FIGURE 10: Phases de conception

### Définition

Un système formel est une modélisation mathématique d'un langage en général spécialisé qui en représente les éléments, termes, formules, dérivations, etc, par des objets finis (entiers, suites, arbres ou graphes finis, etc). Le propre d'un système formel est que l'on peut vérifier algorithmiquement la correction de ses éléments, c'est-à-dire que ceux-ci forment un ensemble récursif. Les méthodes formelles sont des techniques basées sur les mathématiques pour décrire les propriétés d'un système. Elles peuvent être vues comme des mathématiques appliquées du génie logiciel, fournissant des notations, des modèles, des analyses techniques qui peuvent être utilisées pour contrôler et analyser des conceptions informatiques. Les méthodes formelles peuvent aider à augmenter la confiance dans l'exactitude du programme par preuve, raffinement et tests (au niveau des spécifications et de l'implémentation).

Voici les techniques permettant d'augmenter la confiance d'un système formel :

– **La preuve**, parfois appelée vérification formelle implique une démonstration rigoureuse (impliquant habituellement de la logique déductive) qui vérifie qu'une implémentation vérifie bien ses spécifications.
– **Le raffinement** est le développement des implémentations qui sont correctes par construction.
– **Les tests au niveau des spécifications** consistent à exécuter les spécifications formelles pour vérifier (détecter les inconsistances et problèmes internes) et valider (assurer que les exigences du clients sont correctement assurées) les spécifications.
– **Les tests au niveau de l'implémentation** consistent à exécuter une implémentation avec certaines entrées et comparer les résultats qu'elles génèrent avec ceux qu'on attend.

Il est important de remarquer que ces techniques de vérification formelle peuvent très bien s'appliquer sur les phases de modélisation, d'architecture et d'implémentation lors de la conception d'un système. Dans ce travail, nous nous focalisons donc sur les méthodes formelles utilisées lors de la phase de modélisation définie en génie logiciel. C'est pourquoi les techniques de vérification qui nous intéressent dans ce travail sont celles permettant de vérifier les spécifications formelles définies lors de la modélisation d'un système. Il y a deux approches principales pour faire de la vérification de spécifications formelles : le *theorem proving* et le *model checking*. Le theorem proving permet de traiter certains ensembles d'états infinis mais est difficile à automatiser. Le model checking ne permet pas de traiter des ensembles infinis d'états mais est beaucoup plus facile à automatiser.

## Theorem proving

Le theorem proving est une technique de vérification formelle bien établie appliquée pour vérifier si une implémentation donnée ($I$) est conforme à ses spécifications ($S$). Ceci peut être exprimé mathématiquement soit par une implication ($I \rightarrow S$) soit par une relation d'équivalence ($I \equiv S$) entre I et S comme un théorème qui doit être prouvé. Les spécifications et l'implémentation sont exprimées dans le même langage formel. La preuve formelle est rigoureusement construite comme une séquence de pas basée sur un ensemble d'axiomes et de règles d'inférence, comme la simplification, la réécriture et l'induction.

Au contraire du model checking, le theorem proving peut traiter des espaces infinis d'états. Les techniques de preuve par induction permettent de prouver des propriétés sur des domaines infinis. La structure de la preuve est divisée en deux sous preuves : la propriété est d'abord vérifiée par l'état initial (n=1) et ensuite c'est le pas inductif qui vérifie la propriété pour chaque état ultérieur (n+1).

Il y a aussi d'autres techniques de preuve comme la preuve déductive et la preuve par contradiction. Une preuve déductive consiste à construire une conclusion à partir d'une séquence d'étapes basée sur des axiomes et des règles d'inférence afin de prouver une hypothèse ($P$). Dans une preuve par contradiction, le point de départ est la négation de cette hypothèse ($\neg P$) qui doit être prouvée. A partir de cette nouvelle hypothèse ($\neg P$) une preuve déductive est construite et si la conclusion contredit le point de départ alors

l'hypothèse originale ($P$) est prouvée comme étant vraie.

Le theorem proving est une technique de vérification puissante malheureusement, elle ne permet pas d'être automatisée c'est pourquoi cette technique est très peu employée dans le domaine de la vérification d'IHM.

## Model checking

Le model checking est une technique de vérification formelle qui a été appliquée au hardware, aux protocoles de communication et aussi aux systèmes réactifs.

Le système est modélisé comme une machine à états finis et les propriétés que le système doit respecter sont écrites en logique temporelle. Les model checkers [6] sont alors utilisés pour prouver automatiquement par analyse exhaustive de l'entièreté de l'espace des états du système que ces propriétés sont vérifiées dans le modèle du système. Ceci peut être exprimé mathématiquement comme : $S \models P$, signifiant que la propriété $P$ est vérifiée dans le système $S$ (spécifié comme une machine à états finis). Le résultat obtenu par un model checker peut être, soit vrai, soit faux auquel cas un contre-exemple peut être fourni. Le contre-exemple est un chemin, une séquence d'états dans lequel la transition du système qui échoue est montrée. Des exemples d'outils permettant de faire du model checking sont :LTSA, Spin, SMV (Symbolic Model Verifier), HyTech (The Hybrid TECHnology Tool) Kronos et UPAAL.

La logique temporelle est une classe de la logique modale. Elle étend la logique des propositions pour incorporer des opérateurs temporels dans le sens où les formules peuvent évaluer différentes valeurs de vérité en fonction du temps.

L'utilisation de la logique temporelle pour modéliser des systèmes est directe. Chaque état correspond à un état possible du programme et le fait de se déplacer d'un état vers un autre correspond à l'exécution d'un pas du programme .

Il y a différents types de logique temporelle qui correspondent à différentes vues du temps (*branching* ou *linear*, *discrete* ou *continuous*, *past* ou *future*). Avec un modèle en temps linéaire, chaque instant a seulement un successeur. Avec un branching time, chaque instant peut avoir un ou plusieurs instants comme successeurs. Des exemples de langages logiques temporelles formelles sont le Linear Temporal Logic (LTL) et Computational Tree Logic (CTL).

Dans la logique temporelle linéaire (LTL), il est possible d'exprimer des propriétés en fonction d'un état, en fonction d'une séquence d'états, en fonction du passé et en fonction du future.

Dans la branching-time logique, les opérateurs temporelles sont à quantifier en fonction des chemins qui sont possibles depuis un état donné. Cette logique ajoute deux opérateurs

6. une exemple de model checker est Alloy [7]

aux ensembles d'opérateurs linéaires qui sont E (pour un certain chemin) et A (pour tous les chemins).

La logique temporelle peut être un outil puissant pour exprimer des propriétés de *safety*, *liveness* et *fairness* sur un système. Les propriétés de safety vérifient que quelque chose de mauvais n'arrive jamais. Les propriétés de liveness et fairness vérifient que quelque chose de bon arrive éventuellement dans le futur. Fairness peut être vu comme un cas spécial d'une propriété de liveness et peut être utilisé pour exprimer par exemple qu'un ordonanceur n'ignore jamais une procédure.

L'avantage du model checking est qu'il permet d'implémenter des model checkers avec lesquels il est possible d'automatiser la preuve. L'application de l'automatisation de la preuve dans le domaine de la modélisation de système interactif va permettre de systématiser la vérification de celles-ci et permettre dès lors de minimiser les tests manuels (white box et black box) visant à valider leur conception.

## 3.3 Utilité d'une spécification formelle et familles de méthodes formelles

### Utilité et choix du type de méthodes formelles

L'utilisation de méthodes formelles tôt dans le processus de conception informatique permet de vérifier que des conditions d'utilisation sont correctes avant même d'avoir implémenté le système. Ce qui permet de minimiser toute une série de tests coûteux visant à vérifier les comportements du système concptualisé. Une spécification formelle permet de capturer les exigences de l'utilisateur de manière exacte, non ambigue et complète. L'abstraction de haut niveau nous permet de ne pas devoir penser à l'implémentation et nous permet ainsi de focaliser notre attention sur le vrai problème à savoir les comportements. Dans la thèse de [3], il a été montré que l'utilisation de méthodes formelles comportementales permet d'améliorer les processus de test de systèmes interactifs en termes de plus haut degré d'automation et de systématisation. Un plus haut degré d'automation peut être atteint en générant des cas d'essai automatiquement à partir des modèles formels réalisés lors de la modélisation du système interactif.

### Définition et familles de méthodes comportementales

Les méthodes formelles comportementales sont des méthodes formelles qui permettent de décrire les traces d'exécution d'un système en fonction de ses états et de ses transitions. Nous parlerons dans ce travail d'aspects comportementaux pour décrire la manière dont un formalisme décrit ses états et ses transitions d'états.

Après une recherche dans le domaine des méthodes formelles comportementales, nous avons constaté que celles-ci peuvent être catégorisées en cinq grandes familles. Ces fa-

milles se distinguent par la manière dont elles définissent un état et une transition d'états.

Voici les cinq familles que nous distinguons :

- les machines à états
- les réseaux de Petri
- les grammaires
- les théories ensemblistes
- les approches hybrides

## Les besoins des familles de formalismes

Avant de définir les différentes familles de formalismes, nous allons dresser une liste des besoins généraux que nécessitent les formalismes comportementaux pour modéliser les différentes classes d'IHMs. Les besoins de cette liste sont construits à partir des besoins de modélisation des IHMs réalisés au point **2.4** et à partir de l'utilité des méthodes formelles comportementales dans le domaine des IHMs. Ces besoins peuvent être vus comme des critères généraux permettant dans un premier temps de déterminer les avantages et les inconvénients des différentes familles de formalismes.

| Classe d'IHMs | Besoins des familles |
|---|---|
| CLI, GUI, IHM3 | – L'utilisabilité<br>– La possibilité de décrire les états et les transitions du système<br>– La possibilité de décrire les traces d'exécution<br>– La représentation des entrées et des sorties du système<br>– L'automatisation de la preuve |
| CLI | – La prise en compte de tous les caractères du langage<br>– La possibilité de raisonner sur l'agencement des caractères |
| GUI | – La prise en compte de la disposition des widgets<br>– La possibilité de raisonner sur plusieurs tâches concurrentes (synchronisation, attente, communication) |
| IHM3 | – Les besoins des CLIs et des GUIs<br>– La possibilité de déterminer la provenance des entrées et la destination des sorties<br>– La prise en compte du noyau fonctionnel |

On entend par "utilisabilité" la facilité que procure le formalisme pour décrire un système et pour comprendre le fonctionnement d'un système formalisé.

## 3.4 Catégorisation des familles de formalismes

Tout au long de ce paragraphe, nous définissons les différentes familles de formalismes et pour chacune d'elles, nous déterminons ses avantages et ses inconvénients en fonction des besoins des différentes classes d'IHMs.

### Le système Switch

Pour faciliter la compréhension de la description de ces différentes familles de formalismes, nous allons définir les comportements d'un système simple appelé *Switch* [8] permettant de mettre en application un formalisme appartenant à chacune des familles considérées. Le système Switch est une lampe munie d'un interrupteur qui permet de l'allumer ou de l'éteindre lorsqu'on presse dessus et qui ne permet plus aucune interaction lorsque l'ampoule est grillée. Sur la FIGURE 11, nous illustrons les comportements du système Switch.

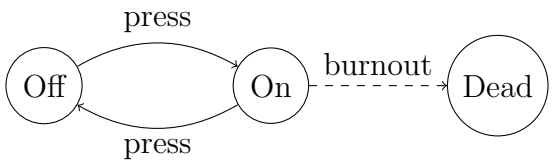



FIGURE 11: Switch

### Les machines à états

Le terme machine à états finis décrit une classe de modèles informatiques qui se compose d'un ensemble fini d'états, d'un état de départ, d'un ensemble d'entrées et d'une fonction de transition qui détermine l'état suivant de la machine à états finis en fonction de l'état courant dans lequel elle se trouve. Les machines à états peuvent être utilisées pour modéliser des systèmes interactifs. Quand un système interactif est modélisé par une machine à états finis de Mealy [7] déterministe, il est exprimé par un sextuplet $< S, X, Y, \sigma, \lambda, s_0 >$ où

- $S$ est un ensemble fini d'états possibles ;
- $X$ est un ensemble fini d'entrées ;
- $Y$ est un ensemble fini de sorties ;
- $\sigma$ est la fonction de transition $S \times X \rightarrow S$ ;
- $\lambda$ est la fonction de sortie $S \times X \rightarrow Y$ ;

7. une machine de Mealy ou automate de Mealy (proposée par Georges H. Mealy) est un automate fini pour lequel les sorties dépendent à la fois de l'état courant et des symboles d'entrée

– $s_0$ appartient à S et est l'état initial.

Chaque transition est actionnée par une entrée d'utilisateur. En réponse à une entrée d'utilisateur, le système réalise une action qui peut changer son état et produire des sorties pour l'utilisateur.

Les notations graphiques
Les notations graphiques des machines à états sont souvent préférées par les développeurs, les analystes et les testeurs plutôt que des informations textuelles car les diagrammes permettent de visualiser plus facilement des relations complexes. Une manière courante de représenter une machine à états finis est d'utiliser des diagrammes de transition d'états où les états sont des noeuds et les transitions sont des arêtes orientées labellisées par les conditions suffisantes de réaction appelées « trigger/action » [9]. Les diagrammes de transition d'états sont adaptés pour représenter des systèmes réactifs de petite envergure mais pour de gros systèmes, ils deviennent rapidement déstructurés, irréalisables et chaotiques. C'est pourquoi David Harel a développé les *statecharts* [1] qui étendent les diagrammes de transition d'états avec les concepts suivants :

– **La profondeur** que l'on décline aussi comme étant une XOR décomposition ou emboîtement d'états. Un état peut contenir exactement une région servant de container pour des sous-états.
– **L'orthogonalité** aussi connue sous le nom de AND décomposition, un état peut se composer de plusieurs régions orthogonales s'exécutant de manière indépendante et concurrente.
– **La communication** utilisée par les entités concurrentes d'un système pour se synchroniser. Une manière de faire pour exprimer la communication consiste à utiliser des variables partagées par plusieurs entités concurrentes à travers lesquelles ces entités concurrentes peuvent échanger de l'information.

Il faut remarquer que les notations UML sont basées sur les statecharts de Harel et utilisent ses extensions. Sur la FIGURE 12, nous avons modélisé les comportements du système Switch avec une notation graphique de machine de Moore [8].

Les notations tabulaires
Nous pouvons également trouver des notations tabulaires de machines à états, sur la FIGURE 13 nous avons modélisé les comportements du système Switch avec une notation tabulaire SCR Mode Transition [10]. Les notations tabulaires sont complémentaires aux notations graphiques et permettent de facilement identifier l'incomplétude des spécifications formelles d'un système interactif en repérant par exemple les cases vides du tableau [9].

Les notations textuelles
Il existe une troisième forme de notation, ce sont les notations textuelles qui peuvent être moins compactes et intuives que les notations graphiques et tabulaires. Il faut néanmoins remarquer que ces notations sont largement utilisées par les outils de vérification de

8. une machine de Moore ou automate de Moore (proposée par Edward F. Moore) est un automate fini pour lequel les sorties ne dépendent que de l'état courant

spécifications formelles car leur utilisation ne nécessite qu'un simple éditeur de texte. Sur la FIGURE 14, nous avons modélisé les comportements du système Switch avec les notations textuelles de l'outil LTSA [16] permettant de vérifier de manière automatique des machines à états.

Les avantages
Grâce aux caractéristiques que fournissent les statecharts de Harel, les machines à états sont des moyens bien adaptés pour décrire les systèmes tels que les GUIs et les IHM3s composés de plusieurs entités concurrentes. De plus, cette famille se décline sous diverses formes : graphique, tabulaire et textuelle, c'est pourquoi elle permet à l'utilisateur de choisir la forme qui lui parait la plus adaptée pour modéliser son système. Les modèles issus de cette famille disposent de nombreux outils pour automatiser leur preuve. Les comportements des IHMs sont explicitement représentés par les machines à états par le biais d'états et de transitions.

Les inconvénients
Les limites que comportent les modèles issus de cette famille sont essentiellement liées aux nombres d'états ($N_s$), de transitions ($N_t$) et d'entités concurrentes ($k$) des systèmes qu'on souhaite modéliser car la complexité des algorithmes de preuve augmente de manière exponentielle en fonction du nombre d'entités concurrentes et vaut :$kN_s^{k-1}N_t$ [11].

Les exemples
Des exemples de formalismes de la famille des machines à états utilisés pour modéliser des IHMs sont *LTSA*, *Process algebras* et *Flownet*.

### Machine à états de Switch

Notation graphique

La modélisation de Switch sur la FIGURE 12 a été réalisée avec l'outil LTSA qui utilise une notation graphique de machine de Moore.

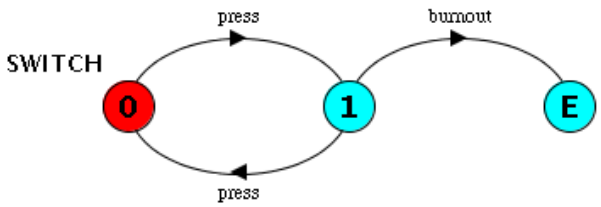



FIGURE 12: Notation graphique du Switch

Notation tabulaire

La modélisation de Switch sur la FIGURE 13 est une notation tabulaire SCR Mode Transition [10] où pour chaque état (Mode), nous énumérons les conditions suffisantes [9] permettant au système de changer d'état (New Mode). À chaque événement permettant de modifier l'état courant nous attribuons @T signifiant que lorsque l'événement survient dans le Mode considéré, le système passe dans le New Mode.

| Old Mode | press | burnout | New Mode |
|---|---|---|---|
| Off | @T | - | On |
| On | @T | - | Off |
| | - | @T | Dead |

FIGURE 13: Notation tabulaire du Switch

Notation textuelle

La modélisation de Switch sur la FIGURE 14 est une notation textuelle définie par LTSA.

```
SWITCH = OFF,
OFF    = (press -> ON),
ON     = (press-> OFF|burnout->END).
```

FIGURE 14: Notation textuelle du Switch

## Les réseaux de Petri

Les réseaux de Petri reposent sur une base mathématique forte qui utilise plusieurs techniques d'analyse qui ont été développées pour supporter leur validation [10]. Les réseaux de Petri se composent de places (des cercles), de transitions (des rectangles noires) et d'arcs dirigés (flèches). À chaque pas d'exécution, les places d'un réseau de Petri peuvent contenir zéro ou plusieurs jetons (points noirs à l'intérieur des cercles). Ce sont précisément les configurations de chacune de ces places qui définissent l'état du système formalisé. Une transition consomme les jetons présents dans ses places d'entrée et en produit dans ses places de sortie. Le nombre de jetons consommés et de jetons renvoyés par une transition dépend du poids de ses arêtes dirigées entrantes et sortantes. Une transition se produit lorsque ses places d'entrée contiennent un nombre requis de jetons. Nous avons modélisé sur la FIGURE 15 les comportements du système Switch avec un réseau de Petri. De la même manière que les statecharts étendent les diagrammes de transition d'états, il existe des réseaux de Petri étendus avec lesquels on peut réduire la taille des modèles.

9. trigger conditions
10. Ces techniques sont des techniques utilisées en model checking

Les réseaux de Petri de haut niveau comme les réseaux de Petri colorés et les réseaux de Petri annotés en sont des exemples.

Les avantages
Les réseaux de Petri permettent d'exprimer de manière très naturelle toutes les interactions simples [11] qui caractérisent les GUIs. Ils permettent d'exprimer de manière adaptée la synchronisation, la communication et l'attente. Les modèles issus de cette famille disposent d'outils permettant d'automatiser leur preuve notamment en utilisant la matrice d'incidence du réseau. Leur notation graphique les rend facilement utilisable.

Les inconvénients
Les réseaux de Petri représentent l'état interne du système formalisé de manière moins explicite que les machines à états car c'est la combinaison de la configuration de toutes les places du réseau qui définit l'état du système. Les réseaux de Petri ne sont pas du tout adaptés pour modéliser les CLIs car si pour chaque caractère du langage d'une CLI, il faut dessiner une place destinée à représenter la présence ou l'absence de celui-ci alors le réseau devient rapidement chaotique et illisible. De plus l'expression de l'ordre attribué à chacun de ces caractères pour définir un mot serait complètement irréalisable pour des dialogues complexes.

Les exemples
Un exemple de formalisme de la famille des réseaux de Petri utilisé pour modéliser des IHMs est ICO (Interactive Cooperative Object) [22].

### Réseau de Petri du Switch

Les actions sur Switch {press, burnout} seront représentées par des transitions. Nous définirons $T_1$ pour représenter `press` lorsque Switch est sur l'état `Off`, $T_2$ pour représenter `press` lorsque Switch est sur l'état `On` et $T_3$ pour représenter `burnout` lorsque Switch est sur l'état `On`. Sur la FIGURE 15, nous avons modélisé le réseau de Petri du système Switch avec l'outil PIPE2 [16] et sur la FIGURE 16, nous avons représenté la matrice d'incidence relative ce réseau de Petri. L'état du système est défini par le vecteur suivant $(P_1, P_2, P_3)$ où $P_i$ définit le nombre de jetons présent dans la place i. L'état `Off` est représenté par $(1, 0, 0)$, l'état `On` par $(0, 1, 0)$ et l'état `Dead` par $(0, 0, 1)$. Sur la FIGURE 17, on peut voir un exemple d'utilisation de la matrice d'incidence $M_t$ où $V_t$ est le vecteur servant à déterminer les transitions actionnées.

## Les grammaires

Une grammaire formelle peut définir précisément un langage formel par un ensemble de règles qui peuvent être utilisées pour générer tous les mots du langage en faisant de la réécriture à partir d'un symbole de départ (grammaire générative). Une grammaire peut également être utilisée pour analyser si un mot d'entrée est un membre du langage (gram-

11. comme défini au point **2.2**

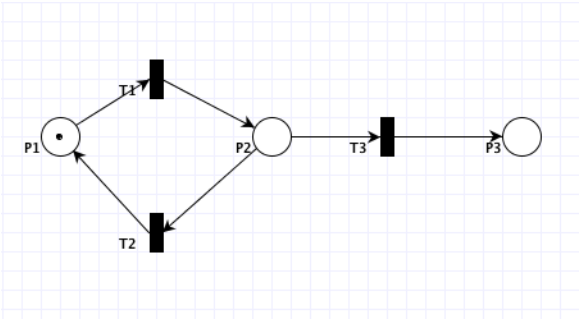



FIGURE 15: Réseau de Petri du Switch

| | $P_1$ | $P_2$ | $P_3$ |
|---|---|---|---|
| $T_1$ | $-1$ | 1 | 0 |
| $T_2$ | 1 | $-1$ | 0 |
| $T_3$ | 0 | $-1$ | 1 |

FIGURE 16: Matrice d'incidence $M_t$ du Switch

$$\overbrace{(1,0,0)}^{Off} + \overbrace{(1,0,0)}^{V_t} * \overbrace{\begin{pmatrix} -1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & -1 & 1 \end{pmatrix}}^{M_t} = \overbrace{(0,1,0)}^{On}$$

FIGURE 17: Utilisation de la matrice d'incidence

maire analytique).

Une grammaire générative peut être définie formellement par un quadruplet $(N, \Sigma, P, S)$ [14] où,

– $N$ est un ensemble fini de non terminaux ;
– $\Sigma$ est un ensemble fini de symboles terminaux, disjoint de $N$ ;
– $P$ est un ensemble fini de règles de production ;
– $S$ est un symbole de départ (un non-terminal de $N$).

De manière générique, une règle de production est de la forme $v \to w$, où v, w sont des mots de terminaux et de non-terminaux où formellement v, w appartiennent à l'expression régulière $(\Sigma \cup N)^*$. Les non-terminaux d'une grammaire sont des symboles permettant de dériver les chaînes de terminaux acceptées par un langage. Si le côté gauche de toutes les règles de productions d'une grammaire n'est formé que d'un symbole non-terminal [12], alors cette grammaire est appelée Grammaire Non-Contextuelle.

Backus-Naur Form (BNF) est un exemple d'une notation utilisée pour décrire des Grammaires Non-Contextuelles. Chaque règle est composée d'un non-terminal abstrait du côté gauche qui est défini par (:=) comme un terme plus spécifique du côté droit. Ce terme

12. qui peut être différent pour chaque règle de production

plus spécifique est composé d'alternatives, de successions et d'options qui sont indiquées par un "or" (|), un "and" (+), et des crochets fermés ([...]) respectivement.

Les avantages
Les Grammaires Non-Contextuelles sont relativement communes pour modéliser les interfaces en ligne de commande. Elles permettent de spécifier les commandes textuelles ou les expressions qu'un programme doit contenir. Les terminaux dans la grammaire sont des mots d'entrée générés par l'utilisateur via les modalités d'entrée. Ces mots représentent les actions utilisateurs. Les terminaux sont combinés par des règles productions dans la grammaire pour former des structures de plus haut niveau appelés non-terminaux et c'est précisément ces symboles non terminaux qui définissent l'état du système lorsqu'on utilise des grammaires non-contextuelles. La collection des chaînes de productions d'une grammaire définit le langage employé par l'utilisateur dans son interaction avec l'ordinateur. La description formelle d'un système avec des grammaires peut être vérifiée de manière complète et consistante notamment en traduisant cette description dans une logique modale permettant de faire du model checking .

Les inconvénients
Les techniques basées sur les grammaires sont difficiles à utiliser pour décrire des interfaces telles que les GUIs dirigées par une souris et composées de widgets où par exemple la modélisation de manipulations directes de widgets par des séquences rigides d'actions requises sont presque toujours indésirables. Les grammaires ne s'adaptent pas bien aux changements, ne sont pas adéquates pour représenter la concurrence et ne supportent une représentation explicite des états. De plus, un autre problème qui survient avec les grammaires est que l'ordre dans lequel les règles de production sont utilisées dépendent du type d'algorithme utilisé par le parser. Dans le cas d'un algorithme d'analyse bottom-up, une production est utilisée quand tous les symboles du côté droit ont été reconnus. Dans le cas d'une analyse top-down, une production est utilisée quand le premier symbole terminal qui pourrait être généré du côté droit est rencontré. Une dernière remarque qu'on peut notifier sur l'utilisabilité des grammaires est que celles-ci sont difficiles à lire et à écrire pour quelqu'un n'ayant pas un certain bagage mathématique. C'est pourquoi actuellement l'utilisation de grammaires pour modéliser des interfaces utilisateur tend à être plutôt rare.

Les exemples
Des exemples de grammaires utilisées dans la formalisation d'IHM sont TAG (Task Action Grammar), CMG (Constraint Multiset Grammars) et VEG (Visual Event Grammar) [24].

## Représentation du Switch avec une grammaire non contextuelle

Pour chaque action sur Switch {press,burnout} nous allons associer un symbole terminal, c'est-à-dire que pour l'action "press" lorsque Switch est sur l'état "On" nous associerons le terminal "$a$" et lorsqu'il est sur l'état "Off" nous associerons à "press" le terminal "$b$". Pour l'action aléatoire "burnout" qui ne survient que lorsque Switch est sur l'état "On" nous associerons le terminal "$\lambda$". C'est pourquoi les traces d'exécution de Switch

seront représentées par l'expression régulière $\rho$ suivante $((ba)^* + b)$. On peut voir sur la FIGURE 18, l'automate $\alpha$ représentant l'expression régulière $\rho$. Cette automate $\alpha$ peut être défini par la grammaire $\Gamma$ suivante :

- $N = \{A, B\}$
- $\Sigma = \{a, b\}$
- $P = \{B \to bA, A \to aB|\lambda\}$
- $S = B$

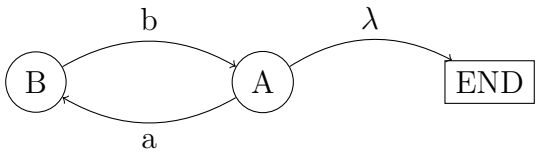



FIGURE 18: Automate $\alpha$

## Les théories ensemblistes

Dans les théories ensemblistes, l'état du système est modélisé explicitement par des constructions mathématiques comme des ensembles, des appartenances, des fonctions et des relations. Les opérations du système sont spécifiées en définissant comment elles affectent l'état du système. La théorie axiomatique des ensembles, le lambda calcul et la logique du premier ordre sont des langages fortement utilisés dans ce genre de formalisme.

Typiquement, les langages des théories ensemblistes ont des états et des opérations qui changent l'état du modèle. Les invariants sont des expressions booléennes qui restreignent l'ensemble des états accessibles du système formalisé. Les opérations peuvent avoir des pré-conditions et des post-conditions associées. Les pré-conditions déterminent l'ensemble des états où l'opération peut se produire et les post-conditions l'état atteint après l'exécution de l'opération(ainsi que la valeur retournée par l'opération) ou restreint juste l'ensemble des états dans lequel le système se terminera après l'exécution de l'opération.

Il y a différents types et styles de spécification dans les théories ensemblistes. Elles peuvent être exécutables ou non-executables, et explicites ou implicites. Une spécification exécutable facilite la validation contre les exigences informelles du client vu qu'elle propose des tests qui peuvent être rapidement vérifiés. Une spécification implicite décrit la fonctionnalité par des moyens d'opérations/méthodes avec des pré-conditions (implicites) et des post-conditions. Une post-condition implicite permet de vérifier la validité d'un résultat obtenu par la méthode définie par une spécification mais ne permet pas de le calculer. Une spécification explicite décrit la fonctionnalité par des moyens de post-conditions explicites ou des corps de méthode algorithmique avec lesquels il est possible de calculer le résultat attendu.

Les notations les plus utilisées pour le développement de spécifications relatives aux théories ensemblistes sont VDM-SL (Vienna Development Method Specification Language), Z [12], et leur exetension orientée objet VDM++ et Object-Z.

Les avantages
Une méthode utilisée dans la modélisation d'IHM avec les théories ensemblistes consiste à modéliser un ensemble d'interacteurs en Z et à les combiner entre eux pour former le modèle du système interactif. Chaque interacteur définit une entité concurrente d'une GUI et permet d'exprimer la synchronisation, l'attente et la communication. Cette technique permet aussi la preuve de propriétés du système formalisé et explicite clairement le lien entre les actions de l'utilisateur et la réaction du système (schémas d'opérations). Cette famille dispose de techniques puissantes pour faire la preuve de ses modèles.

Les inconvénients
On constate que cette famille de formalismes n'est pas fort utilisée pour modéliser des IHMs car elle est très lourde à mettre en oeuvre, peu réutilisable, illisible pour quelqu'un n'ayant pas de solides bases mathématiques. Les théories ensemblistes ne permettent de représenter le temps que de façon simpliste (le temps est vu comme un nombre, représentant le temps écoulé depuis un instant de référence) [5]. Les techniques de preuve utilisées par les théories ensemblistes sont généralement des techniques de theorem proving qui sont difficiles à automatiser.

Les exemples
Un exemple de ce type de formalisme utilisé dans la modélisation d'IHM est IVY [17].

## Modèle Z du système Switch

Types primitifs

$MACHINE : \{objet\}$
$ETAT : \{on, off, dead\}$

Schema type

*Lampe*

$etat : MACHINE \rightarrow ETAT$

Schema d'operation de consultation

*ConsultationLampe*

$[-]Lampe$
$o? : MACHINE$
$e! : ETAT$

---

$e! = etat(o?)$

Schemas d'operation de modification

*Press*

$\triangle Lampe$
$o? : MACHINE$

---

$etat(o?) = on \lor etat(o?) = off$
$etat' = etat \oplus \{o? \mapsto \{on, off\} \setminus \{etat(o?)\}\}$

*Burnout*

$\triangle Lampe$
$o? : MACHINE$

---

$etat(o?) = on$
$etat' = etat \oplus \{o? \mapsto dead\}$

Operation robuste

$OperationLampeRobuste = Press \lor Burnout$

### Les approches hybrides

L'objectif des formalismes hybrides est de combiner des caractéristiques d'au moins deux langages de spécification pour construire un langage final plus riche qui combine le meilleur des langages utilisés. Une approche largement utilisée est de combiner les spécifications des théories ensemblistes avec celles des machines à états [3].

## 3.5 Caractéristiques comportementales des familles de formalismes

Ces différents tableaux vont permettre de comparer la manière dont chacune de ces familles définissent un état, une transition d'états et une trace d'exécution.

### Un état

| Familles | Description |
|---|---|
| Machine à états | un état est représenté de manière explicite par une noeud labellisé appartenant à l'ensemble S de tous les états possibles du système |
| Réseau de Petri | un état est représenté de manière implicite par la combinaison des configurations de toutes les places du réseau, celui-ci est noté par un vecteur $V = (p_1, p_2, ..., p_n)$ où $p_i$ représente le nombre de jetons présents dans la place $i$ |
| Grammaire | un état est représenté de manière implicite par la partie gauche d'une règle de production appelé un non terminal |
| Théorie ensembliste | un état est défini par des attributs symbolisés par des variables ou des ensembles |

### Une transition

| Familles | Description |
|---|---|
| Machine à états | une transition est représentée de manière explicite par un arc orienté labellisé et est définie par $\sigma$ qui est la fonction de transition $S \times X \rightarrow S$ où X est l'ensemble des entrées du système |
| Réseau de Petri | une transition est représentée de manière explicite par un rectangle labellisé relié à des arcs orientés entrants et à des arcs orientés sortants. Ces arcs sont définis par des poids qui déterminent la consommation ou la production d'un certain nombre de jetons dans les places qu'ils relient aux transitions lorsque les transitions sont enclenchées. On peut représenter les transitions du réseau par une matrice d'incidence |
| Grammaire | une transition est définie par $v \rightarrow e_1e_2w \vert e_3e_4x$ où $v$ est un non terminal représentant l'état du système, $w$ et $x$ sont des mots composés de terminaux et de non terminaux et $e_1e_2$, $e_3e_4$ sont des entrées du système représentées par des terminaux |
| Théorie ensembliste | une transition est définie par des fonctions, des relations ou des opérations respectant des pré-conditions, des post-conditions et des invariants. Ces invariants permettent de définir tous les états accessibles du système |

### Une trace d'exécution

| Familles | Les réseaux de Petri |
|---|---|
| Machine à états | une trace est définie par un état initial et par une succession de transitions notée $S_0$,$T_1$,$T_2$,$T_3$,... |
| Réseau de Petri | une trace est définie par la somme d'un vecteur initial $V$ définissant la configuration des places avec $(t_1, t_2, ..., t_m) \times A$ où A est la matrice d'incidence et $(t_1, t_2, ..., t_m)$ est le vecteur des transitions enclenchées |
| Grammaire | une trace est définie par une chaîne de caractères $\zeta$ composées de terminaux et de non terminaux et $\zeta \in (\Sigma \cup N)^*$ |
| Théorie ensembliste | une trace est définie par une configuration initiale $C_0$ des attributs du système et par une succession d'opérations $O_1, O_2, ...O_m$ |

## 3.6 Conclusion

En se basant sur les besoins des CLIs et sur les caractéristiques comportementales des différentes familles de formalismes, on remarque que la famille de formalismes la plus adéquate pour modéliser les CLIs est la famille des grammaires. Cette constatation se justifie par le fait que les entrées complexes d'une CLI définissent les transitions d'états de la CLI et que pour être modélisées, ces transitions d'états d'une CLI nécessitent une

prise en compte de l'ensemble des caractères du langage et d'un algorithme permettant de décider de l'appartenance d'une séquence de caractères dans ce langage. Cette constatation ne signifie pas que les autres familles de formalismes ne permettent pas de modéliser les CLIs mais témoigne bien du fait que les caractéristiques inhérentes aux grammaires répondent de manière adéquate aux besoins des CLIs.

Au contraire des CLIs, les GUIs et les IHM3s forment un ensemble hétérogène d'IHMs dont il n'est pas possible de déterminer la famille la plus adéquate tant les GUIs se déclinent sous des formes variées [13]. Cependant en analysant les besoins des GUIs et des IHM3s, on remarque que leur modélisation nécessite impérativement la possibilité de décrire des entités concurrentes avec toutes les implications qui leurs sont associées [14].

13. menu déroulant, forme "fill-in", manipulation directe,...
14. synchronisation, communication et opérateurs temporels

# 4 Les critères

Maintenant que nous avons une vue d'ensemble sur ce que sont les méthodes formelles comportementales et leur application respective dans le domaine des IHMs, nous allons pouvoir dresser une liste de critères permettant d'évaluer la capacité qu'ont ces formalismes pour modéliser des IHMs. Cette liste se base évidemment sur les besoins de modélisation des IHMs développés au point **2.4**.

Notre objectif principal dans ce travail consiste à déterminer les méthodes formelles les plus adéquates qui permettent de modéliser des systèmes interactifs en fonction de leur IHM. Dans un but de généralisation nous nous baserons sur les besoins que nécessitent les IHMs multimodales pour déterminer les besoins des formalismes car il est important de constater que les méthodes formelles qui sont adéquates pour les IHMs multimodales alors le sont nécessairement pour les CLIs et les GUIs. Cette constatation est due au fait que les IHMs multimodales peuvent être vues comme une combinaison d'IHMs monomodales comprenant des GUIs et des CLIs.

Les principales caractéristiques que nécessitent les formalismes pour modéliser les IHMs multimodales sont :

- la possibilité de représenter les aspects comportementaux de l'IHM
- la possibilité de représenter plusieurs entités concurrentes
- la communication et la synchronisation de ces entités concurrentes
- une représentation du temps
- une représentation explicite des entrées et des sorties
- une capacité à s'adapter aux modifications
- des outils de vérification automatisés
- la prise en compte du noyau fonctionnel
- une notation utilisable avec un pouvoir expressif suffisant

Dans ce chapitre, nous allons déterminer sur base de ces besoins les critères de modélisation qui nous permettront de déterminer les formalismes les plus adéquats pour conceptualiser, analyser et maintenir des modèles d'IHM.
Ces critères seront divisés en trois familles :

- les critères de conception
- les critères d'analyse
- les critères de maintenance

## 4.1 Définition des concepts utilisés

Modélisation d'une IHM
Nous parlons de la modélisation d'une IHM pour désigner l'activité de modélisation d'un système interactif en fonction de son IHM.

Formalisme d'IHM
Nous considérons que les formalismes d'IHM sont des méthodes formelles comportementales utilisées lors de la phase de modélisation d'un système interactif. Ces formalismes décrivent les interactions entre l'utilisateur et le système interactif et sont utilisés dans le but de systématiser la conception d'un système interactif.

Catégorisation
La catégorisation des formalismes permet de classer les formalismes en fonction de leurs aspects comportementaux. La catégorisation des formalismes a été réalisée au point **3.4**.

Critère
Un critère est une des caractéristiques que nécessite un formalise pour permettre à l'utilisateur du formalisme de tirer profit de son utilisation en fonction des objectifs qu'il se donne.
Les principaux objectifs sont :

- conceptualiser un modèle du système
- analyser un modèle du système
- maintenir un modèle du système

Nous classerons les critères parmi trois familles de critères, chacune représentant un de ces trois objectifs. Les critères permettent de comparer les formalismes entre eux sur base d'objectifs définis afin de déterminer le formalisme le plus adéquat. Cette comparaison peut se faire en évaluant les critères, c'est-à-dire en assignant une valeur significative à chacun des critères.

Méthodologie
La méthodologie est le procédé qui permet d'objectiver les critères, c'est-à-dire de les évaluer. Pour chaque critère, une méthodologie sera mise en place afin que chaque critère puisse être évalué.

Famille de critères
Les familles de critères permettent d'identifier l'utilité que l'on peut tirer des critères appartenant à une même famille. Nous distinguerons trois familles de critères :

- **les critères de conception** permettant d'évaluer la conception du modèle d'un système
- **les critères d'analyse** permettant d'évaluer l'analyse du système formalisé
- **les critères de maintenance** permettant d'évaluer la maintenance du système formalisé

Par exemple, un critère de conception permettra à l'utilisateur du formalisme d'évaluer la capacité du formalisme à concevoir le modèle du système qu'il désire réaliser. Evidemment chaque critère d'une même famille permet d'évaluer un aspect différent de l'objectif principal qu'il dessert. Nous définirons en détails les différents aspects couverts par chaque critère dans le paragraphe suivant.

Aspects statiques
Ce sont les structures du formalisme qui permettent de décrire les aspects d'un système qui ne varient pas au cours du temps lorsqu'on utilise ce système, c'est-à-dire l'architecture du système.

Aspects comportementaux
Ce sont les structures du formalisme qui permettent de décrire les états et les transitions d'états d'une IHM.

Encapsulation
En programmation orientée objet, l'encapsulation est l'idée de protéger l'information contenue dans un objet et de ne proposer que des méthodes de manipulation de cet objet. Ainsi, les propriétés et axiomes associés aux informations contenues dans l'objet seront assurés et validés par les méthodes de l'objet et ne seront plus de la responsabilité de l'utilisateur extérieur. L'utilisateur extérieur ne pourra pas modifier directement l'information et risquer de mettre en péril les axiomes et les propriétés comportementales de l'objet. L'objet est ainsi vu de l'extérieur comme une boîte noire ayant certaines propriétés et ayant un comportement spécifié. La manière dont ces propriétés ont été paramétrées est alors cachée aux utilisateurs de la classe. On peut modifier ce paramétrage sans changer le comportement extérieur de l'objet. Cela permet donc de séparer la spécification du comportement d'un objet, du paramétrage pratique de ces spécifications.

Héritage
L'héritage est un concept puissant de la programmation orientée objet, permettant entre autres la réutilisabilité (décomposition du système en composants) et l'adaptabilité des objets grâce au polymorphisme. Elle se nomme ainsi car le principe est en quelque sorte le même que celui d'un arbre généalogique. Ce principe est fondé sur des classes dont les « filles » héritent des caractéristiques de leur(s) « mère(s) ». Chaque classe possède des caractéristiques (attributs et méthodes) qui lui sont propres. Lorsqu'une classe fille hérite d'une classe mère, elle peut alors utiliser ses caractéristiques.

Polymorphisme
En informatique, le polymorphisme est l'idée d'autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.

Hiérarchisation
La hiérarchisation permet de décrire les schémas d'organisation des structures de données comme le fait le langage UML.

Récursion
La récursivité est une démarche qui fait référence à l'objet de la démarche qui permet de décrire un processus dépendant de données en faisant appel à ce même processus sur d'autres données plus «simples».

Emboîtement

L'emboîtement est la capacité d'un formalise à pouvoir décrire des supers états se comportant comme des containers de sous-états.

## 4.2 Les familles de critères

Une manière de comparer différents formalismes est de définir des critères d'évaluation qui permettent de quantifier sur base d'une méthodologie les raisons qui orientent notre choix vers un formalisme bien déterminé. Encore une fois, les critères d'évaluation ne se déclinent pas de manière unidimensionnelle, il existe bien entendu différentes familles de critères :

- les critères de conception
- les critères d'analyse
- le critère de maintenance

Ce paragraphe va servir à définir les différents aspects couverts par chaque famille de critères.

### 1 Les critères de conception

Cette première famille de critères que nous distinguons concerne tous les critères permettant de favoriser la conception du modèle de l'IHM. C'est-à-dire tous les aspects liés à l'utilisateur et aux structures fournies à l'utilisateur pour favoriser la modélisation d'IHM. Voici les différents critères que nous définirons dans cette famille :

- **1.1** la structuration de l'information véhiculée par le formalisme
- **1.2** le point de vue externe
- **1.3** le parallélisme
- **1.4** les ressources mises à disposition

### 2 Les critères d'analyse

Cette deuxième famille de critères que nous distinguons concerne tous les critères liés à l'analyse du système formalisé, c'est-à-dire tous les aspects liés à l'algorithmique mise en place pour automatiser l'analyse de la cohérence du système formalisé et les aspects liés à l'expressivité du formalisme. Il est important de remarquer que l'analyse du système se réalise à partir de la modélisation du système. C'est précisément cette famille de critères qui va permettre d'évaluer la capacité du formalisme à minimiser les tests coûteux que nécessite la validation d'un système interactif.
Voici les différents critères que nous évaluerons dans cette famille :

- **2.1** le bagage théorique nécessaire
- **2.2** l'expressivité du formalisme
- **2.3** l'interrogation du système formalisé

### 3 Les critères de maintenance

Cette troisième famille de critères que nous distinguons concerne tous les critères liés à la maintenance du système formalisé, c'est-à-dire tous les aspects liés à la modularité et à la réutilisabilité du système formalisé ainsi que tout ce qui permet à l'utilisateur de réaliser des modifications sur son système formalisé. C'est précisément cette famille de critères qui va permettre de faciliter les modifications du système formalisé dues aux exigences variables que peuvent susciter des utilisateurs à l'égard d'une IHM.
Voici le critère que nous évaluerons dans cette famille :

– **3.1** la maintenabilité du système formalisé

### 4 Un critère de complémentarité

Pour mettre en évidence l'utilité que génère des formalismes hybrides composés de plusieurs formalismes appartenant à des familles de formalismes différentes, nous définissons un critère permettant de mettre en évidence la complémentarité que chacun d'entre eux apporte pour la mise en oeuvre des différents objectifs.
Voici le critère que nous évaluerons dans cette famille :

– **4.1** la complémentarité des hybrides

## 4.3 Développement des critères

Dans ce paragraphe, nous allons définir tous les critères évoqués et déterminer pour chacun d'eux une méthodologie permettant de les objectiver.

### 1 Les critères de conception

#### 1.1 Structuration de l'information véhiculée par le formalisme

Ce critère prend en compte les structures utilisées par le formalisme pour représenter des caractéristiques inhérentes aux IHMs . Ce critère va déterminer les structures employées pour décrire les aspects statiques et comportementaux d'un système interactif. C'est précisément la prise en compte des aspects statiques du système interactif qui va permettre de faciliter le choix de l'architecture de ce système car il est généralement très difficile d'établir le lien qu'il y a entre les interactions survenant à travers l'IHM et la manière dont celles-ci sont prises en compte par le noyau fonctionnel. La aspects comportementaux du formalisme vont permettre de définir la famille de formalismes à laquelle appartient le formalisme. Ce critère évalue également si il est possible de manière explicite de représenter les entrées et les sorties du système interactif et quels sont les moyens mis en place pour faire le lien entre les entrées, les sorties et les aspects statiques du noyau fonctionnel. Par définition, un système interactif comporte toujours des entrées et des sorties que l'utilisateur doit impérativement voir, c'est pourquoi la prise en compte

de leur visibilité par le formalisme est un aspect primordial dans l'interaction homme-machine. Nous parlerons de la prise en compte de la représentation de l'IHM pour parler de la visibilité des entrées et des sorties.

La méthodologie consiste à déterminer si le formalisme :

- **1.1.1** prend en compte les aspects statiques
- **1.1.2** prend en compte les aspects comportementaux
- **1.1.3** fait le lien entre I/O et les aspects statiques du noyau fonctionnel
- **1.1.4** représente explicitement les entrées et les sorties
- **1.1.5** prend en compte la représentation de l'IHM

### 1.2 Point de vue externe

La qualité d'un formalisme peut être évaluée de deux manières différentes [3] :
- soit d'un point de vue utilisateur, tout ce qui concerne les aspects utilisateur (point de vue externe)
- soit d'un point de vue software engineering, tout ce qui concerne l'exactitude, la vérification, l'architecture, le code, l'expressivité, etc (point de vue interne)

Ce critère ne tiendra compte que du point de point de vue externe car le point de vue interne relatif à la modélisation d'une IHM est pris en compte par d'autres critères que nous détaillerons dans la suite de ce travail.

Voici les aspects pris en compte par le point de vue externe :

- **1.2.1** Satisfaction : ce critère est lié au point de vue subjectif que l'utilisateur se fait sur l'aspect du formalisme, c'est-à-dire son côté confortable, intuitif,...
- **1.2.2** Fiabilité : ce critère tient compte du fait que le formalisme permet de mettre en évidence les erreurs que l'utilisateur peut commettre. Ce critère est étroitement lié à la flexibilité du système. Un système flexible donne une plus grande liberté à l'utilisateur et par la même occasion une plus grande opportunité de se tromper. Alors qu'un formalisme rigide, lui, donne moins de liberté à l'utilisateur mais aussi moins d'opportunité de se tromper.
- **1.2.3** Apprentissage : ce critère se réfère au temps que l'utilisateur met pour apprendre comment utiliser le formalisme.
- **1.2.4** L'efficacité d'utilisation : ce critère tient compte de l'efficacité que peut avoir l'utilisateur lorsqu'il réalise une tâche avec le formalisme. Ce critère peut être mesuré en temps et/ou en nombre d'actions nécessaire pour réaliser une tâche. Un formalisme inefficace peut être inutile.

La méthodologie de ce critère va consister à donner cinq points de balance à chacun des aspects définis et plus la valeur attribuée à chacun des aspects est grande pour le formalise considéré plus celui-ci tend à satisfaire l'aspect. Nous sommes conscient que l'évaluation de ce critère est partielle car elle n'a pas été réalisée par un échantillon de gens suffisant pour permettre d'inférer une quelconque conclusion à son égard.

### 1.3 Parallélisme

Ce critère prend en compte les aspects temporels et concurrentiels supportés par le formalisme ainsi que la communication nécessaire à la synchronisation des différentes entités concurrentes d'une IHM. Sachant qu'une IHM multimodale se compose de plusieurs entités concurrentes dont l'ordre des exécutions appliquées sur chacune d'entre elles a une importance, la considération du temps (opérateurs temporelles) par le formalisme est indispensable. La communication entre les différentes entités concurrentes est également une propriété indispensable que doit fournir le formalisme pour permettre la synchronisation entre celles-ci. Généralement la communication peut être prise en compte par le formalisme par le biais de variables partagées. Il est important de remarquer que la représentation du noyau fonctionnel peut se faire par le biais d'une entité concurrente sur laquelle se synchronisent toutes les modalités d'entrée et de sortie.

La méthodologie consiste à déterminer si le formalisme :

- **1.3.1** permet de modéliser les entités concurrentes d'un système
- **1.3.2** permet la communication/synchronisation entre les entités concurrentes
- **1.3.3** permet d'exprimer le temps

### 1.4 Ressources mises à disposition

Ce critère évalue le degré d'utilisation du formalisme en fonction de la présence d'un outil performant développé pour le mettre en pratique, de la présence d'un tutoriel pour apprendre à utiliser le formalisme et en fonction de la date de la dernière publication à son sujet. Ce critère va permettre à l'utilisateur du formalisme de quantifier les ressources qui sont mises à sa disposition pour assimiler le formalisme mais également pour utiliser le formalisme.

L'évaluation des documents se fera en fonction :

- **1.4.1** de la présence d'un tutoriel
- **1.4.2** de la date de la dernière publication

Pour les outils, l'évaluation se fera en fonction de l'outil le plus performant.
La performance d'un outil dépendra :

- **1.4.3** de la présence d'un support visuel (éditeur)
- **1.4.4** de la présence d'un vérificateur permettant d'analyser et de vérifier certaines propriétés
- **1.4.5** de la présence d'un générateur de code
- **1.4.6** de la présence d'un simulateur
- **1.4.7** de la mise à jour

## 2 Les critères d'analyse

### 2.1 Bagage théorique nécessaire

Ce critère va déterminer le bagage théorique nécessaire pour utiliser le formalisme. Ce critère va permettre à l'utilisateur de se faire une idée sur le type d'algorithmique qu'il pourra mettre en oeuvre pour réaliser ses objectifs.

– **2.1.1** L'objectif de ce critère est de déterminer les théories dominantes qui sont requises pour l'utilisation du formalisme. L'évaluation se fera en fonction du cursus standard d'un ingénieur en informatique et les valeurs attribuées dépendront du fait que toutes les théories requises sont obligatoires dans le programme d'étude d'un ingénieur (**valeur=3**) ou qu'elles sont au moins toutes proposées en option (**valeur=2**) ou qu'au moins une n'est pas proposée (**valeur=1**) dans son programme.

### 2.2 Expressivité du formalisme

Notion de base

Avant de commencer à développer ce critère, certaines notions de calculabilité sont nécessaires pour pouvoir définir l'expressivité d'un formalisme ainsi que le lien entre la puissance utilisatrice et expressive de celui-ci [18] ou encore pour évoquer le compromis qu'il y a entre la puissance expressive d'un langage et la complétude de la preuve dont ce langage bénéficie (théorème de Gödel) [15].

La première notion que nous allons définir est la notion d'expressivité d'un formalisme. On dit qu'un formalisme A est plus expressif qu'un formalisme B si il existe un algorithme qui permet de traduire tout système formalisé de B vers A et si il existe également un algorithme qui permet de traduire tout système formalisé de A vers B alors on dit que les deux formalismes ont une expressivité équivalente. Une manière de déterminer l'expressivité d'un formalisme est de vérifier si il est possible de coder les machines de Turing dans ce formalisme (et si il est possible de coder ce formalisme avec les machines de Turing alors on dit que le formalisme est équivalent aux machines de Turing) car par la thèse de Church-Turing on sait que tout problème de calcul fondé sur une procédure algorithmique peut être résolu par une machine de Turing [19].
En informatique et en logique, on utilise l'adjectif Turing-complet pour décrire un système formel ou formalisable ayant la force de calcul des machines de Turing. Une machine de Turing est un modèle abstrait du fonctionnement des appareils mécaniques de calcul, tel un ordinateur et sa mémoire, créé par Alan Turing en vue de donner une définition précise au concept d'algorithmique.

La seconde notion à développer est la puissance descriptive et la puissance utilisable énoncées par Mark Green. La puissance descriptive d'une notation est l'ensemble des classes d'interfaces utilisateur qui peuvent être décrites par la notation. Plus l'ensemble est grand plus la notation est puissante. Déterminer la puissance descriptive d'un type de notation peut être converti en un problème de la théorie des langages formels et dans la majeure partie des cas une réponse définitive peut être donnée. La puissance utilisable

d'une notation est l'ensemble des classes d'interfaces utilisateur qui peuvent être facilement décrites par ce type de notation. La puissance utilisable d'une notation est toujours un sous-ensemble de la puissance descriptive de cette notation.

Et enfin la troisième notion à développer est celle concernant le compromis qu'il y a entre l'expressivité d'un formalisme et la complétude que fournit le système de preuve du formalisme. Cette notion peut se résumer par le théorème de Gödel qui dit que la complétude absolue n'existe pas car il n'existe pas de système de preuve complet pour toute logique qui contient de l'arithmétique.

Décidabilité algorithmique et décidabilité logique

Il est important de remarquer que la décidabilité algorithmique et la décidabilité logique sont deux concepts différents. Un problème de décision est dit algorithmiquement décidable s'il existe un algorithme, une procédure mécanique qui termine en un nombre fini d'étapes et qui décide du problème, c'est-à-dire qui réponde par oui ou par non à la question posée par le problème. S'il n'existe pas de tels algorithmes, le problème est dit algorithmiquement indécidable. Par contre, en logique classique, d'après le théorème de complétude, une proposition est dite indécidable dans une théorie s'il existe des modèles de la théorie où la proposition est fausse et des modèles où elle est vraie.

Description du critère

Ce critère va déterminer si le formalisme est Turing complet ou non. Ce critère va également déterminer le type de logique sous-jacente employée par l'outil du formalisme pour effectuer la preuve automatisée du système formalisé et va décrire la technique de preuve employée. Ce critère va permettre d'évaluer la puissance descriptive et la puissance de vérification automatisée dont bénéficie le formalisme.

L'évaluation de l'expressivité dépendra :

- **2.2.1** du fait que la notation du formalisme est Turing complet ou pas
- **2.2.2** du type de logique sous-jacente employée par l'outil du formalisme (décidabilité logique)
- **2.2.3** de la technique de preuve utilisée (décidabilité algorithmique)

## 2.3 Interrogation du système formalisé

Ce critère évalue l'information utile que l'on peut extraire automatiquement à partir du système formalisé lorsqu'on analyse le système. Ce critère va déterminer si il est possible de simuler des traces d'exécution, si il est possible de repérer des deadlocks, si il est possible d'évaluer la liveness et si il est possible d'évaluer la reachability avec l'outil associé au formalisme. Le but de ce critère est d'évaluer la capacité de l'outil associé au formalisme à minimiser les tests coûteux qui accompagnent la phase de validation d'un

système interactif.

L'évaluation de ce critère tiendra compte de :

- **2.3.1** la capacité de l'outil à simuler des traces d'exécution du système
- **2.3.2** la capacité de l'outil à repérer des deadlocks
- **2.3.3** la capacité de l'outil à évaluer la liveness
- **2.3.4** la capacité de l'outil à évaluer la reachability

# 3 Le critère de maintenance

## 3.1 Maintenabilité du système formalisé

Ce critère évalue la capacité du formalisme à permettre des modifications du système formalisé. Certaines propriétés telles que l'héritage, le polymorphisme ou encore la récursion permettent de simplifier la tâche de l'utilisateur lorsqu'il s'agit de changer ou de réutiliser certaines caractéristiques du système formalisé. Voyant aujourd'hui la vitesse à laquelle les applications de systèmes interactifs changent (telles que celles des smartphones et autres systèmes disposant d'une GUI), un formalisme adapté aux changements répondrait complètement aux besoins de ce genre de système.

L'évaluation de ce critère tiendra compte des propriétés suivantes :

- **3.1.1** l'héritage
- **3.1.2** le polymorphisme
- **3.1.3** l'encapsulation
- **3.1.4** la récursion
- **3.1.5** la hiérarchisation
- **3.1.6** l'emboîtement

# 4 Un critère de complémentarité

## 4.1 Complémentarité des hybrides

Ce critère va évaluer la complémentarité des formalismes hybrides combinant plusieurs formalismes en se basant sur l'évaluation individuelle de chacun de ses formalismes. L'évaluation individuelle signifie une évaluation par rapport aux autres critères définis. Cette complémentarité pourra s'évaluer en fonction des objectifs déterminés par l'utilisateur du formalisme. Nous remarquons qu'aujourd'hui de plus en plus de formalismes sont hybrides, ceci est essentiellement dû au fait que l'expression d'un même système avec plusieurs formalismes va permettre d'analyser le système sous différents angles afin d'établir une traçabilité entre ceux-ci. La traçabilité est à prendre au sens de Axel van Lamsweerde [13], celle-ci permet d'augmenter la cohérence et la robustesse d'un système à conceptualiser.

Le but est d'identifier les différents formalismes concurrents et d'évaluer la complémentarité qu'ils ont entre eux . Pour identifier les différents formalismes concurrents utilisés, il suffit de se référer aux définitions des différentes familles de formalismes. Pour évaluer la complémentarité que ces différents formalismes ont entre eux, il faut dans un premier temps appliquer tous les autres critères individuellement sur chaque formalisme identifié et dans un second temps, il faut appliquer l'algorithme de complémentarité. On considère qu'un formalisme A apporte un point de complémentarité à un autre formalisme B si A a une évaluation plus favorable par rapport à un critère de B pour l'objectif visé (chaque critère est trié par objectif). L'algorithme de complémentarité est un algorithme incrémental commençant par le formalisme dont la somme des évaluations des différents critères est la plus élevée et l'analyse de la complémentarité se fait à chaque étape par rapport au formalisme restant dont la somme des évaluations des différents critères est la plus élevée. Les points de complémentarité peuvent être déterminés en fonction de différents objectifs car les critères sont classés par rapport aux objectif qu'ils tentent de mettre en évidence.

Exemple

Un formalisme F est composé de trois formalismes concurrents A, B et C. Les critères analysés sont $C_1$, $C_2$ et $C_3$ et les valeurs respectives pour A, B et C sont (1,4,5), (2,4,1) et (0,0,7). On prend d'abord A car il possède la somme de critères la plus élevée, $C_1 + C_2 + C_3 = 10$. Ensuite on prend B car il possède la somme la plus élevée parmi les formalismes restants (B,C) et on comptabilise les points de complémentarité apportés par B. B apporte 1 point de complémentarité à A par rapport au critère $C_1$. L'algorithme étant incrémental, on fusionne A et B en ne gardant que les meilleurs critères, c'est-à-dire $A + B = (2, 4, 5)$.
Ensuite on prend C qui, lui, apporte 2 points de complémentarité à $A \cup B$ par rapport à $C_3$. Maintenant si on sait que $C_1$, $C_2$ et $C_3$ correspondent respectivement aux objectifs $O_1$, $O_2$ et $O_3$, alors si notre objectif est $O_1$, le critère de complémentarité vaut 1.

## 4.4 Tableaux de critères

| Critères de conception | Détails |
|---|---|
| Structuration | – Aspects statiques<br>– Aspects comportementaux<br>– Description de la présentation de l'IHM<br>– Description des I/O<br>– Lien entre I/O et les aspects statiques du noyau fonctionnel |
| Point de vue externe | – Satisfaction<br>– Fiabilité<br>– Apprentissage<br>– Efficacité |
| Parallélisme | – Modélisation des entités concurrentes<br>– Communication/synchronisation entre les entités concurrentes<br>– Expression du temps |
| Ressources | – Documents (tutoriel, date de la dernière publication)<br>– Outils (éditeur, vérificateur, générateur de code, simulateur, mise à jour) |

| Critères d'analyse | Détails |
|---|---|
| Bagage théorique nécessaire | – Valeur déterminée en fonction du cursus d'un ingénieur |
| Expressivité du formalisme | – Turing-complet<br>– Type de logique<br>– Type de preuve |
| Interrogation du système formalisé (outil) | – Simulation de traces<br>– Détection de deadlocks<br>– Evaluation de la liveness<br>– Evaluation de la reachability |

| Critère de maintenance | Détails |
|---|---|
| Maintenabilité | – Héritage<br>– Polymorphisme<br>– Encapsulation<br>– Récursion<br>– Hiérarchisation<br>– Emboîtement |

| Critère de complémentarité | $Formalisme_1$ | $Formalisme_2$ |
|---|---|---|
| Complémentarité des hybrides | – conception : $C_1$<br>– analyse : $A_1$<br>– maintenance : $M_1$ | – conception : $C_2$<br>– analyse : $A_2$<br>– maintenance : $M_2$ |
| | $Formalisme_{1\cup2}$ | |
| | – comp. conception : $C_{1\cup2} = \|C_1 - C_2\|$<br>– comp. analyse : $A_{1\cup2} = \|A_1 - A_2\|$<br>– comp. maintenance : $M_{1\cup2} = \|M_1 - M_2\|$ | |

# 5 Présentation des formalismes choisis

Dans ce chapitre, nous allons présenter les différents formalismes choisis et introduire leur notation en modélisant pour chacun d'eux le système Switch présenté au point **3.4**. Nous terminerons ce chapitre par la présentation d'un système plus complexe nommé *Microwave* avec lequel nous mettrons en application dans les chapitres suivants tous les formalismes choisis. Les chapitres suivants consisteront à évaluer chacun des formalismes choisis sur base des critères développés au chapitre **4**. L'évaluation de chaque formalisme se divisera en deux parties :

- la première partie consistera à analyser, de manière théorique, le formalisme sur base des critères. Cette partie s'apparente à la phase d'évaluation que réalisera un utilisateur lambda sur base de notre grille de critères vis à vis d'un formalisme qu'il ne connaît pas.
- la deuxième partie consistera à analyser de manière pratique le formalisme en l'utilisant pour modéliser le système Microwave. Cette partie permettra de mettre en évidence l'utilité des critères.

Les formalismes que nous avons choisis ne sont pas des formalismes hybrides, c'est pourquoi nous n'évaluerons pas le critère de complémentarité lors de l'analyse théorique des formalismes choisis.

## 5.1 Visual Event Grammar (VEG)

### Présentation

Le premier formalisme sur lequel nous allons appliquer nos critères est VEG (Visual Event Grammar) qui est un formalisme de la famille des grammaires. Ce formalisme ainsi que son outil supportent la spécification formelle, vérification, conception et implémentation d'interfaces graphiques utilisateur. VEG ne tient pas compte des aspects de présentation de la GUI. Il est seulement concerné par la description des dialogues de contrôle de la GUI par des moyens modulaires, grammaires communicantes, avec une notation visuelle supportée par un éditeur appelé Dialog Control Editor (DCE) [20].

Nous avons choisi ce premier formalisme car celui-ci apporte une approche singulière de modélisation de GUI qui consiste à décrire les transitions internes du système par le biais de règles de production [23].

### Modélisation de Switch avec VEG

Dans VEG, une règle de grammaire prend la forme :

```
currentstate::=<user_input> \visual_action nextstate
```

Dès lors, les non-terminaux sont représentés par {`On,Off,Dead`} qui sont les états de Switch et les terminaux sont représentés par les entrées d'utilisateur {`<press>,<burnout>`}

et par les sorties visuelles du système {\Lampe_allumee, \Lampe_eteinte, \Lampe_cassee}. La ligne `Axioms Off` signifie que l'état initial du modèle Switch est `Off`. Nous avons illustré sur la FIGURE 19 la grammaire définissant les comportements de Switch qu'on appelle un `Model` en VEG et sur la FIGURE 20, nous avons réalisé la même modélisation avec l'outil DCE.

```
Model Switch
        Axioms Off
        Off::=<press> \Lampe_allumee On
        On::=<press> \Lampe_eteinte Off
             | <burnout> \Lampe_cassee Dead
End Switch
```

FIGURE 19: Modélisation de Switch en VEG

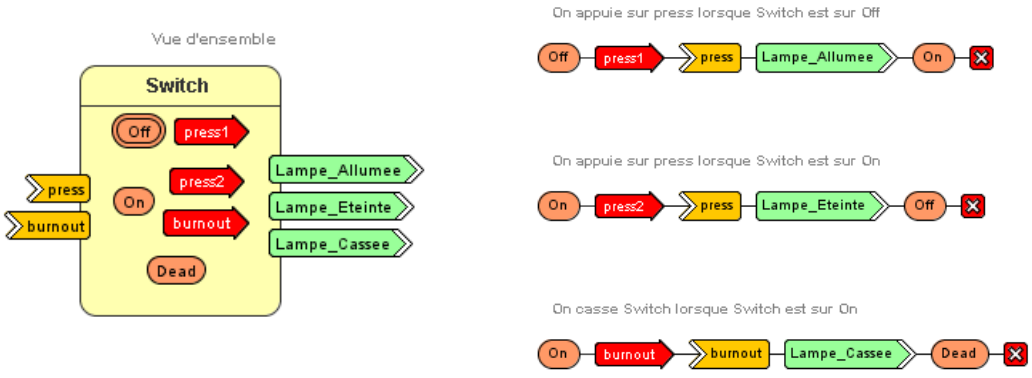



FIGURE 20: Switch avec l'outil DCE

## 5.2 IVY

### Présentation

Le second formalisme sur lequel nous allons appliquer nos critères est IVY qui est un formalisme de la famille des théories ensemblistes. Ce formalisme applique des techniques de model checking pour vérifier les propriétés d'un système interactif. Il modélise les systèmes interactifs comme une composition d'entités indépendantes appelées interacteurs. Les interacteurs peuvent être vus comme des abstractions d'architecture informatiques similaires à des objets dans la programmation orientée objet. Chaque interacteur se compose d'un état interne qui est décrit par ses attributs ainsi que d'actions permettant de décrire leurs transitions d'états. La spécification des interacteurs se fait par de la logique modale (deonic) qui est ensuite traduite en NuSMV. Les propriétés des interacteurs sont alors décrites par des formules CTL ou LTL et vérifiées par le model checker NuSMV.

Nous avons choisi ce deuxième formalisme car il développe une approche très intéressante dans la modélisation d'IHM à savoir les interacteurs [21] développés à York tout en utilisant des techniques de model checking.

### Modélisation de Switch en IVY

Sur la FIGURE 21, nous avons modélisé les comportements du système Switch en IVY. Cette modélisation ne nécessite qu'un seul interacteur nommé `main` car il est constitué juste d'une entité. Le type `NewType` sert à définir un ensemble d'atomes {`On, Off, Dead`}. Les attributs servent à définir l'état du système, dans le cas de Switch, on a juste besoin de l'attribut `etat` qui est une variable de type `NewType`. Les actions {`press, burnout`} servent à définir les interactions possibles avec Switch et le préfixe `[vis]` sert à déterminer si les attributs ou les actions sont visibles par l'utilisateur de telle sorte qu'on puisse par exemple définir les actions internes du système telle que `burnout`. `[]etat=Off` sert à définir l'état initial. `per(press)`→ `etat'=On|etat'=Off` sert à déterminer tous les états accessibles en réalisant l'action `press`, c'est-à-dire `On` ou `Off`.
La règle `etat=On` →`[press] etat'=Off` signifie que lorsque Switch est dans l'état `On` et qu'on réalise l'action `press` alors Switch se retrouve dans l'état `Off`. `keep(etat)` sert signaler que l'attribut `etat` conserve la même assignation. Sur la FIGURE 22, nous avons illustré le diagramme IVY représentant l'interacteur `main` du Switch.

```
types
 NewType = {On, Off, Dead}

interactor main
 attributes
  [vis] etat: NewType
 actions
  [vis] press
  burnout
 axioms
  [] etat=Off #initial state
  per(press) -> etat'=On | etat'=Off
  per(burnout) -> etat'=Dead
  etat=On ->[press] etat'=Off
  etat=Off ->[press] etat'=On
  etat=On ->[burnout] etat'=Dead
  etat=Dead ->[press] keep(etat)
```

FIGURE 21: Modélisation de Switch en IVY

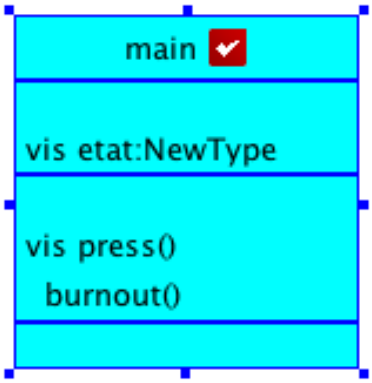



FIGURE 22: Diagramme du Switch avec l'outil IVY

## 5.3 Interactive Cooperative Object(ICO)

### Présentation

Et enfin le troisième formalisme que nous avons choisi d'analyser est ICO (Interactive Cooperative Object) qui est un formalisme de la famille des réseaux de Petri. ICO est un formalisme orienté objet spécialement conçu pour la modélisation d'interfaces dirigées par des événements. Chaque objet ICO est défini par quatre composants : un objet coopératif avec des services utilisateur, une partie présentation, et deux fonctions (la fonction d'activation et la fonction de rendu) qui font le lien entre l'objet coopératif et la partie présentation. L'objet coopératif (CO) décrit comment un composant du système interactif réagit aux stimuli externes d'après son état interne. Ce comportement, appelé la Structure du Contrôle de l'Objet (ObCS) est décrit au moyen d'un réseau de Petri de haut niveau. Les transitions de ces réseaux de Petri sont labellisées par les actions d' un objet CO. Chaque classe d'un objet CO spécifie une ou plusieurs interfaces d'un composant du système interactif et fait correspondre à une de ses actions une signature d'une des interfaces qu'il spécifie. Une transition peut se produire quand les places d'entrée sont peuplées d'un nombre de tokens suffisant. Chaque place du réseau de Petri est typée, c'est-à-dire que les jetons à l'intérieur des places doivent être du même type. À chaque fois qu'une transition se produit, l'action liée à la transition est exécutée sur les objets identifiées par les tokens consommés par la transition . Les transitions peuvent générer de nouveaux objets, supprimer des objets et mettre à jour des objets. Les objets modifiés et les nouveaux objets sont envoyés dans les places de sortie.

Nous avons choisi ce troisième formalisme car il permet de prendre en compte les aspects statiques d'une IHM (les interfaces du noyau fonctionnel) tout en permettant d'utiliser des techniques de model checking sur les aspects comportementaux [5].

### Modélisation de Switch avec ICO

La modélisation de Switch avec ICO consiste dans un premier temps à créer la classe de l'objet coopératif appelé `Switch-CO` (Figure 24) qui spécifie l'interface SwitchI (Figure 23) grâce à la commande `specifies`. L'interface SwitchI contient toutes les méthodes nécessaires pour implémenter le système Switch. À partir de la classe `Switch-CO`, nous pouvons construire le réseau de Petri relatif aux comportements du système Switch (Figure 26). À chaque transition du réseau est reliée un action exécutée sur l'objet CO impliqué. La fonction permettant de faire le lien entre la place [15], le widget [16], l'événement [17], et le service rendu par le système [18], est appelée fonction d'activation (Figure 27), et la fonction permettant de faire le lien entre une modification sur le réseau de Petri et le rendu du système est la fonction de rendu (Figure 28).

15. On, Off ou Dead
16. l'interrupteur du Switch
17. une pression sur l'interrupteur
18. appliquer la méthode `press`

```
public interface SwitchI {

        void press();

}
```

FIGURE 23: Interface du Switch

```
1  class Switch-CO
2  specifies ISwitch{
3          place Off <Switch>={<new Switch()>};
4          place On <Switch>;
5          place Dead <Switch>;
6
7          transition t1{
8                  action{
9                          x.press();
10                 }
11         }
12
13         transition t2{
14                 action{
15                         x.press();
16                 }
17         }
18
19         transition t3{
20                 action{
21                         //aucune action disponible
22                 }
23         }
24
25         Rendering methods{
26                 void display(String){
27                         //Montre l'etat du Switch
28                 }
29         }
30 }
```

FIGURE 24: Classe CO du Switch

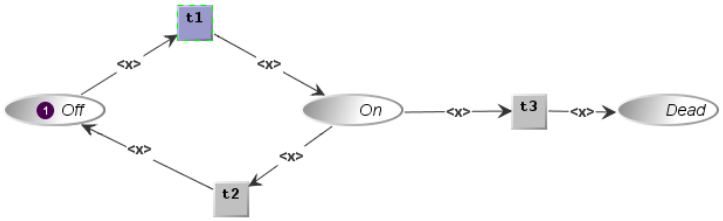



FIGURE 25: Modèle ObCS du Switch

| | $Off$ | $On$ | $Dead$ |
|---|---|---|---|
| $T_1$ | $-1$ | 1 | 0 |
| $T_2$ | 1 | $-1$ | 0 |
| $T_3$ | 0 | $-1$ | 1 |

FIGURE 26: Matrice d'incidence du Switch

| Place | Widget | Event | Service |
|---|---|---|---|
| Off | interrupteur | pression | press |
| On | interrupteur | pression | press |
| Dead | aucun | burnout | aucun |

FIGURE 27: Fonction d'activation du Switch

| ObCS element | | Méthode de rendu |
|---|---|---|
| Nom | caractéristique | |
| t1 | enclenchée | display("Lampe_allumee") |
| t2 | enclenchée | display("Lampe_eteinte") |
| t3 | enclenchée | display("Lampe_cassee") |

FIGURE 28: Fonction de rendu du Switch

## 5.4 Le système Microwave

### L'IHM de Microwave

Le système Microwave est un système interactif dont l'interface se compose d'une modalité d'entrée, le clavier (Keyboard) et de deux modalités de sortie, la lumière (Light) et l'écran (Display). C'est pourquoi selon la définition d'une IHM multimodale, on peut considérer que l'IHM de Microwave est une interface multimodale. Sur la FIGURE 29, on remarque que le noyau fonctionnel (Mode) fissionne son état courant vers les deux modalités de sortie que sont la lumière (Light) et l'écran (Display).

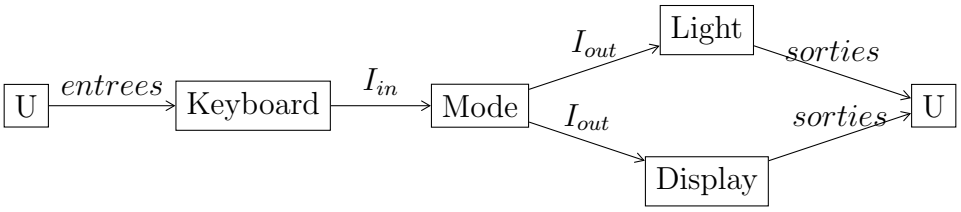



FIGURE 29: IHM de Microwave

### Les comportements de Microwave

Il est important de remarquer que les entrées et les sorties du système Microwave sont représentées par des transitions et que les états respectifs des différentes entités concurrentes sont représentées par des rectangles munis d'un label et emboîtés dans un super état [19]. L'IHM de Microwave est multimodale, c'est pourquoi il est nécessaire de modéliser son noyau fonctionnel (Mode) comme une entité concurrente permettant de synchroniser les deux autres entités concurrentes que sont la lumière (Light) et l'écran (Display) en fonction des entrées fournies par l'utilisateur en interagissant avec le clavier (Keyboard). On peut d'ailleurs voir sur la FIGURE 30 que l'état du noyau fonctionnel est illustré par le super état Mode et que la lumière et l'écran vers lesquels le noyau fonctionnel fissionne son état sont illustrés respectivement par Light et par Display. Le H à l'intérieur d'un super état $S$ signifie que celui-ci est doté d'une mémoire, c'est-à-dire que lorsqu'une entité concurrente quitte $S$ pour être dans un autre état quelconque alors il mémorise le sous-état $\sigma$ dans lequel il était dans $S$ afin de retourner dans $\sigma$ lorsqu'il sera dans $S$. Sur la FIGURE 31, nous avons illustré les statecharts de bas niveau du statechart illustré sur la FIGURE 30. La simplicité des interactions proposées par le système Microwave, nous permet de constater que les modalités de ce système peuvent être simulées par des GUIs.

19. qu'on appelle état hiérarchique

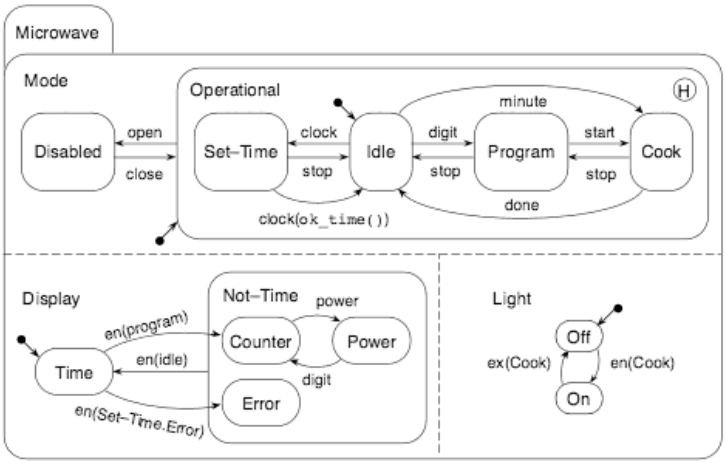



FIGURE 30: Statechart de haut niveau

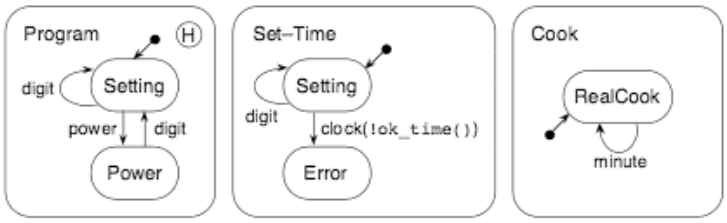



FIGURE 31: Statecharts de bas niveau

# 6 Analyse de VEG

## 6.1 Analyse théorique

### 1 Les critères de conception

#### 1.1 Structuration de l'information

**1.1.1** Aspects statiques : **non**

Les aspects statiques ne sont pas pris en compte par VEG.

**1.1.2** Aspects comportementaux : **oui**

L'idée de base sous-jacente à VEG est de considérer les séquences d'événements d'entrée comme des phrases dans un langage formel. Ces phrases obéissent aux règles syntaxiques décrites par les grammaires communicantes du modèle. Chaque grammaire communicante représente une entité concurrente de l'IHM modélisée . Ces grammaires ne décrivent que les séquences d'actions d'entrée autorisées. Les notations VEG sont basées sur la description d'un ensemble de règles [20] dont chacune d'entre elles définit le comportement d'un composant dans un état particulier. La structure typique de chaque règle est :

<currentstate>::=<user_input><communication><visual_action><goto nextstate>

où le <currentstate> est l'état logique d'un composant de l'IHM, <user_input> est l'occurrence d'un ou plusieurs événements (ex :un utilisateur pousse sur un bouton), <communication> envoie des événements à d'autres composants, <visual_action> est la spécification d'un changement visuel (basé sur la présentation de l'IHM). Il est important de remarquer que <currentstate> et <goto nextstate> sont à considérer comme des symboles non terminaux et que <user_input>, <communication> et <visual_action> sont à considérer comme des symboles terminaux.

**1.1.3** Représentation explicite des entrées et des sorties : **oui**

Les entrées sont explicitement représentées par des `user_input` et les sorties par des `visual_action`. Il est important de remarquer que les entrées et les sorties représentées par VEG sont des symboles terminaux qui permettent d'être combinés avec des symboles non-termminaux [21] pour former des mots reconnaissables par les grammaires communicantes du système que nous avons modélisé.

**1.1.4** Lien entre les I/O et les aspects statiques du noyau fonctionnel : **non**

20. une règle peut être composée d'une ou plusieurs alternatives séparées par "|"
21. représentant l'état du système

VEG ne tient pas compte des aspects statiques, c'est pourquoi VEG ne permet pas de faire le lien entre les I/O et les aspects statiques du noyau fonctionnel.

#### 1.1.5 Présentation de l'IHM : **non**

VEG ne tient pas compte de la présentation de l'IHM.

### 1.2 Point de vue externe

#### 1.2.1 Satisfaction : **4**

#### 1.2.2 Fiabilité : **3**

#### 1.2.3 Apprentissage : **4**

#### 1.2.4 Efficacité : **3**

### 1.3 Parallélisme

#### 1.3.1 Description d'entités concurrentes : **oui**

VEG permet de modéliser plusieurs entités concurrentes par le biais de grammaires communicantes [22] au sein d'un même système. Un objet VEG [23] peut avoir des enfants en utilisant la commande `fork` ou la commande `launch`. En utilisant la commande `fork`, l'objet VEG peut créer un ou plusieurs fils ayant les mêmes fonctions que les siennes et en utilisant la commande `launch`, l'objet VEG peut lancer un ou plusieurs objets concurrents dans un état spécifique ayant des fonctions différentes des siennes. Lorsqu'un objet VEG utilise la commande `fork`, il doit attendre que tous ses fils se terminent avant de pouvoir reprendre la main. Ce qui n'est pas le cas avec la commande `launch` qui elle, permet à l'objet créant les entités concurrentes de fonctionner avec les objets créés. Sur la FIGURE 32, on peut voir le modèle `Controller` lançant un objet du modèle `Switch` par le biais de la commande `launch(switch=Switch.Off)` où `switch` est le nom de l'objet et `Off` est l'état initial de l'objet lancé. Dans le modèle `Switch`, `?press` est à considérer comme un symbole terminal représentant une communication entrante et dans le modèle `Controller`, `!switch.press` est à considérer comme un symbole terminal représentant une communication sortante vers l'objet `switch`.

#### 1.3.2 Moyens de communication/synchronisation : **oui**

Les objets concurrents communiquent entre eux en utilisant les commandes suivantes :
– `? source.event` pour attendre un message provenant de `source`

22. une grammaire comminucante est appelée un `Model` en VEG
23. un objet VEG ayant des enfants ou créant des processus concurrents s'appelle un `container`

```
Model Switch
        Axioms Off
        Off::=?press \Lampe_allumee On
        On::=?press \Lampe_eteinte Off
             | ?burnout \Lampe_cassee
End Switch

Model Controller
        Axioms Start
        Start::= launch(switch=Switch.Off) Actif
        Actif::=<press> !switch.press Actif
                | <burnout> !switch.burnout Inactif
        Inactif::=<quit> \Termine_le_processus
End Controller
```

FIGURE 32: Entités concurrentes en VEG

– `! destination.event` pour envoyer un message vers `destination`

**1.3.3** Expression du temps : **partiellement**

VEG permet de représenter implicitement le temps en définissant un modèle exécutant des actions à intervalles réguliers sur lequel les autres modules concurrents se synchronisent.

### 1.4 Ressources mises à disposition

Documents

#### 1.4.1 Tutoriel :partiellement

Il n'existe pas de tutoriel pour VEG mais le document [23] permet d'étudier le formalisme. L'outil de VEG ne dispose d'aucun tutoriel mais avec l'exemple du `CALCULATOR` fourni avec l'outil, il est possible d'appréhender son utilisation.

#### 1.4.2 Date de la dernière publication :2004

La date de la dernière publication au sujet de VEG remonte à 2004 via le document [24].

Outils

#### 1.4.3 Editeur : partiellement

Les dialogues en VEG sont spécifiés par des grammaires communicantes qui ont des notations visuelles intuitives supportées par un éditeur visuel appelé Dialog Control Editor(DCE). Il faut remarquer cependant que certaines fonctions de VEG ne sont pas ex-

primables avec DCE, c'est par exemple le cas avec la fonction `launch` dans laquelle on ne peut pas spécifier le modèle existant lancé ni son état initial. DCE permet de visualiser les notations visuelles de manière textuelle dans le langage VEG et en XML. La visualisation textuelle des notations graphiques permet à ceux qui connaissent la notation de VEG de vérifier leurs spécifications graphiques.

#### 1.4.4 Vérificateur : partiellement

Les spécifications en VEG peuvent être vérifiées avec les model checker Spin afin de tester la consistance et l'exactitude, pour détecter les deadlocks et les états inaccessibles, mais aussi pour générer les cas de test pour la validation de certaines propriétés. La vérification en VEG est complètement automatisée et est basée sur une abstraction sécurisée des spécifications VEG, qui élimine de manière basique la manipulation de données.

#### 1.4.5 Générateur de code : oui

Pour produire l'application réel, la description XML de l'interface est utilisée par le Code Template Generator (basé sur le ANTLR parser generator) pour construire les modèles des classes controller. Ces classes se composent d'un ensemble de parsers communicants et d'évaluateurs sémantiques avec des parties qui doivent encore être remplies avec du code. La Visual Platform Editor (VPE), le composant chargé de lier le controller à la présentation, construit la version finale du code automatiquement en générant toutes les séquences activant et désactivant les composants nécessaires pour rendre comforme l'interface graphique avec les spécifications VEG. VPE prend en entrée la description XML de la spécification, les modèles Java du Code Template Generator et les fichiers Java de la présentation. Sa sortie est l'application complète composée des mêmes fichiers de présentation enrichis par le code qui permet au controller de guider l'interface visuelle et le code du controller.

#### 1.4.6 Simulateur : non

VEG ne dispose pas de simulateur.

#### 1.4.7 Mise à jour :2001

Les dernières mises à jour datent de 2001.

## 2 Les critères d'analyse

### 2.1 Bagage théorique nécessaire

#### 2.1.1 Valeur : 3

Comme la notation est simple et visuelle, les utilisateurs ne se rendent pas compte qu'ils manipulent des grammaires étant donné que chaque règle de grammaire décrit simplement des séquences d'interactions . C'est pourquoi même si les notations sont complètement formelles, elles peuvent aussi être appréciées par des gens sans bagage spécial en mathématique. Cependant, il est préférable d'avoir certaines bases en langage des traducteurs, en model checking (SPIN) et en JAVA pour maîtriser correctement cet outil.

## 2.2 Expressivité du formalisme

### 2.2.1 Turing complet : **non**

L'idée de base sous-jacente à VEG est de considérer les séquences d'entrées d'utilisateur comme des phrases dans un langage formel. Ces phrases obéissent aux règles syntaxiques décrites par les grammaires (automates). Par exemple, dans certaines circonstances, ouvrir un document qui est déjà ouvert devrait être interdit. C'est pourquoi les grammaires ne peuvent décrire que des séquences d'actions d'entrée autorisées. Un modèle est une grammaire locale décrivant les comportements d'un automate. Une grammaire est composée d'un ensemble de règles grammaticales. Chaque règle peut avoir un côté gauche décrivant un nom de variable (aussi appelé un état ou un symbole non terminal). Chaque règle a toujours un côté droit décrivant le comportement d'un modèle quand il est dans cet état (décrit par le côté droit). Quand une règle a un côté gauche vide, elle est appelée règle omniprésente et peut être appliquée à tous les états. Le côté droit d'une règle est une production dans les grammaires BNF traditionnelles contenant des éléments non terminaux et des éléments terminaux qui peuvent être, soit des événements d'entrée, soit des actions visuelles. Les actions visuelles sont les réponses de la GUI aux interactions de l'utilisateur. En général le côté droit d'une règle peut être dans un format BNF étendu, c'est-à-dire avec des opérateurs alternatifs "|", les opérateurs d'option"[]" et les opérateur d'itération"{ }". VEG utilise un format simplifié pour les règles de production appelé *generalized-right-linear* qui est très efficace pour la vérification automatique :

-le côté droit de chaque règle est composé d'une ou plusieurs alternatives.

-pour chaque alternative il y a au plus un non terminal appelé l'état suivant qui est à la position la plus à droite de l'alternative.

Comme toutes les grammaires non contextuelles, VEG n'est pas Turing complet.

### 2.2.2 Type de logique : **LTL**

L'outil DCE permet théoriquement de traduire ses modèles en SPIN afin de raisonner avec de la logique linéaire temporelle sur les modèles exprimés en VEG.

### 2.2.3 Type de preuve : **model checking**

La technique de preuve utilisée par SPIN est le model checking.

## 2.3 Interrogation du système formalisé

**Interrogation partielle**
Le formalisme VEG ne permet pas d'interroger de manière explicite les modèles qu'il permet de définir. Cependant l'outil de VEG permet théoriquement de traduire le système formalisé en SPIN et l'outil SPIN lui permet de faire du model checking.

## 3 Le critère de maintenance

### 3.1 Maintenabilité du système formalisé

**3.1.1** héritage : **oui**

Généralement une GUI est divisée en plusieurs packages. Un package est un collection de modèles avec des règles de visibilité standard similaires à celle de Java. Un package peut importer d'autres packages.

**3.1.2** polymorphisme : **non**

VEG ne permet pas de faire du polymorphisme.

**3.1.3** encapsulation : **oui**

La commande `launch` permet de créer des objets sans `user_input` qui ne peuvent être modifier que par communication avec l'entité qui les a créés . C'est pourquoi le fait de définir un objet $O_u$ permettant de lancer un ensemble d'objets concurrents $E_m$ est une forme d'encapsulation dans le sens où l'utilisateur sera obligé de passer par l'objet $O_u$ pour changer l'état courant d'un quelconque objet de $E_m$.

**3.1.4** récursion : **non**

VEG ne permet pas d'exprimer la récursion.

**3.1.5** hiérarchisation : **partiellement**

VEG permet de visualiser un diagramme partiel du système interactif représentant les différents modules du système.

**3.1.6** emboîtement : **oui**

L'emboîtement et l'emboîtement doté d'une mémoire sont représentables en VEG grâce à la commande `launch` qui permet de lancer les sous-modèles d'un super modèle.

# 6.2 Analyse pratique

## La modélisation de Microwave

La modélisation de Microwave en VEG se trouve en Annexe (**12.1**).

### L'analyse de la modélisation de Microwave

#### La conception du modèle

Pour modéliser Microwave avec VEG, nous avons défini `Mode`, `Display` et `Light` avec des grammaires communicantes. La particularité de VEG pour définir plusieurs entités concurrentes consiste à définir un modèle nommé `Controller` permettant de lancer les grammaires communicantes afin que celles-ci puissent lors de leur création recevoir en arguments les noms des grammaires (modèles) avec lesquelles elles communiquent. Toutes les entrées d'utilisateur sont directement prises en charge par le noyau fonctionnel nommé `Mode` par les terminaux d'entrée `user_input` et sont ensuite retransmis par les terminaux `!communication` aux entités concurrentes dont le noyau fonctionnel connaît les noms. Grâce à ces communications les deux entités concurrentes que sont `Light` et `Display` se synchronisent sur `Mode` par la commande `?communication`. Les sorties du système représentant l'état interne de celui-ci sont prises en charge par `Light` et `Display` grâce aux terminaux \`visual_action`

#### L'analyse du modèle

L'outil DCE ne permettant pas d'utiliser pleinement la commande `launch`, nous n'avons pas pu modéliser Microwave avec l'outil de VEG. C'est pourquoi nous n'avons pas non plus pu réaliser de vérifications automatisées du système formalisé. Le traducteur permettant de traduire les modèles VEG en SPIN n'est pas non plus disponible avec l'outil DCE.

#### La maintenance du modèle

La modélisation de Microwave a nécessité l'utilisation de l'emboîtement basique et de l'emboîtement mémorisé. L'emboîtement mémorisé s'exprime avec VEG en ne détruisant pas un processus lancé [24] lorsqu'on ne le sollicite plus et l'emboîtement basique consiste à détruire le processus lancé dès qu'on termine sa tâche grâce au terminal `?quit` réceptionné par l'entité détruite. Il faut remarquer que l'encapsulation de Microwave aurait pu être modélisée par la prise en charge de toutes les entrées d'utilisateur par le modèle `Controller` qui se serait ensuite chargé de les communiquer au modèle `Mode`.

## 6.3 Conclusion

Le formalisme VEG permet de modéliser toutes les formes d'IHM que nous considérons avec une certaine facilité pour modifier les modèles qu'il permet de réaliser. Le problème majeur de VEG est que son outil ne permet pas de vérifier de manière automatisée des propriétés du système formalisé. Ce désagrément rend la modélisation en VEG peu rentable car tous les efforts fournis pour modéliser les comportements du système de manière rigoureuse ne permettent pas de tirer la moindre conclusion quant à la vérification du

24. avec la commande `launch`

système.

# 7 Analyse de IVY

## 7.1 Analyse théorique

### 1 Les critères de conception

#### 1.1 Structuration de l'information

**1.1.1** Aspects statiques : **non**

IVY ne prend pas en compte les aspects statiques.

**1.1.2** Aspects comportementaux : **oui**

La représentation des comportements se fait par le biais d'attributs et d'actions. Les attributs permettent de caractériser les états du système et les actions permettent de décrire les transitions du système. Les différentes entités du système sont représentées par des interacteurs pouvant partager des variables par le biais d'extensions ou d'héritages. Les actions se définissent par des axiomes qui déterminent les pré-conditions sur les attributs de l'état courant et les post-conditions de l'état suivant.

**1.1.3** Représentation explicite des entrées et des sorties : **oui**

Les entrées utilisateurs sont représentées par des actions visibles et les sorites sont représentées par des attributs visibles. C'est en ajoutant le préfixe `[vis]`(pour visibility) devant certaines actions et certains attributs qu'on détermine les entrées et les sorties du système formalisé.

**1.1.4** Lien entre les I/O et les aspects statiques du noyau fonctionnel : **non**

IVY ne tient pas compte des aspects statiques, c'est pourquoi IVY ne permet pas de faire le lien entre les I/O et les aspects statiques du noyau fonctionnel.

**1.1.5** Présentation de l'IHM : **non**

IVY ne tient pas compte de la présentation de l'IHM.

#### 1.2 Point de vue externe

**1.2.1** Satisfaction : **3**

**1.2.2** Fiabilité : **3**

**1.2.3** Apprentissage : **2**

**1.2.4** Efficacité : **4**

### 1.3 Parallélisme

**1.3.1** Description d'entités concurrentes : **oui**

Les interacteurs permettent de décrire plusieurs entités concurrentes d'un système interactif.

**1.3.2** Moyens de communication/synchronisation : **oui**

Les interacteurs peuvent lire les attributs d'autres interacteurs grâce à la commande `aggregates`. Comme on peut le voir sur la FIGURE 33, l'interacteur `main` peut lire tous les attributs de l'interacteur `entity1` et `entity2` par le biais des variables respectives `I1` et `I2`.

```
interactor main
 aggregates
  entity1 via I1
  entity2 via I2
 axioms
  [](I1.etat1=Off & I2.etat2=Off)
  effect(I1.press1) <-> effect(I2.press2)
```

FIGURE 33: La commande `aggregates`

L'interacteur `main` peut synchroniser les interacteurs `entity1` et `entity2` grâce à la commande suivante `effect(I1.press1)` ↔ `effect(I2.press2)` qui permet de synchroniser `press1` de l'interacteur `I1` avec `press2` de l'interacteur `I2`. On peut voir sur la FIGURE 34 la hiérachisation des interacteurs (`main`, `entity1`, `entity2`).

**1.3.3** Expression du temps : **partiellement**

IVY permet de représenter implicitement le temps en définissant un interacteur exécutant des actions à intervalles réguliers sur lequel les autres interacteurs concurrents se synchronisent.

### 1.4 Ressources mises à disposition

Documents

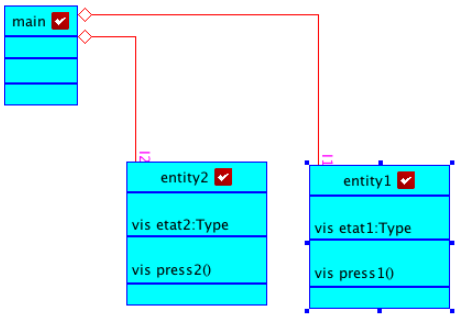



FIGURE 34: Hiérarchisation en IVY

### 1.4.1 Tutoriel :partiellement

Pour étudier le formalisme et pour utiliser l'outil du formalisme, un début de tutoriel a été réalisé mais celui n'englobe pas encore toutes les caractéristiques nécessaires à l'utilisation et à la compréhension de IVY. Le tutriel complet est encore en cours de préparation. Le document [17] permet d'appréhender le formalisme et l'outil.

### 1.4.2 Date de la dernière publication :2009

La date de la dernière publication au sujet de IVY remonte à 2009 avec [17].

Outils

### 1.4.3 Editeur :oui

-Le Model Editor permet de définir le squelette des interacteurs de manière graphique en utilisant le même genre de notation que celle utilisée en UML et permet de définir les attributs, les actions et les axiomes en MAL.

### 1.4.4 Vérificateur :oui

– Le Property Editor fournit des patterns de propriété à vérifier sur le système formalisé. Les propriétés de vérification sont écrites en CTL ou LTL. L'utilisateur IVY peut choisir à partir des patterns de propriété, le pattern qui convient le mieux pour l'analyse de son système formalisé et il l'instancie avec des actions et des attributs du modèle.

– C'est NuSMV qui va permettre de vérifier les propriétés du système formalisé en utilisant des techniques de model checking à savoir CTL et LTL. Sur la FIGURE 35, nous avons illustré deux propriétés à vérifier en CTL par NuSMV sur le système Switch formalisé en IVY. La première propriété exprime que pour toutes les traces partant de l'état `On`, l'état `Dead` est accessible dans les états suivants. Cette propriété est vérifiée par le système formalisé. La seconde propriété exprime que pour toutes les traces partant de l'état `Dead`, l'état `On` est accessible dans les états suivants. Cette propriété n'est pas vérifiée par le système formalisé.

– Le Traces Analyzer permet de mettre en évidence les traces ne vérifiant pas certaines propriétés. La représentation des traces se déclinent sous plusieurs formes. Voici les représentations d'une trace ne vérifiant pas la seconde propriété présentée sur la Figure 35 :
  – sous la forme d'une trace textuelle avec les actions responsables (Figure 36)
  – sous forme d'un arbre (Figure 37)
  – sous forme tabulaire (Figure 38)
  – sous forme de diagramme d'états (Figure 39)
  – sous forme de diagramme d'états montrant les conditions logiques (Figure 40)
  – sous forme d'un diagramme basé sur les actions (Figure 41)

```
test
        AG( etat=On -> EF( etat=Dead ))
test
        AG( etat=Dead -> EF( etat=On ))
```

Figure 35: Vérification en CTL

```
-- specification AG (etat = Dead -> EF etat = On)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  etat = Off
  action = nil
-> Input: 1.2 <-
-> State: 1.2 <-
  etat = Dead
  action = burnout
system diameter: 3
reachable states: 7 (2^2.80735) out of 9 (2^3.16993)
```

Figure 36: Trace textuelle

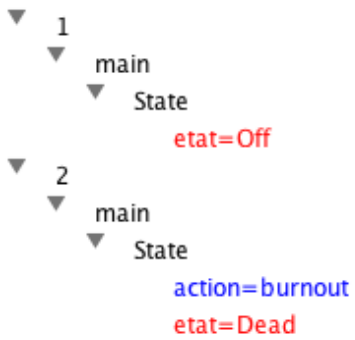



Figure 37: Arbre

| | 1 | 2 |
|---|---|---|
| main.action | | burnout |
| etat | Off | Dead |

FIGURE 38: Forme tabulaire

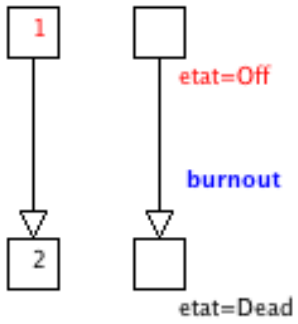



FIGURE 39: Diagramme d'états

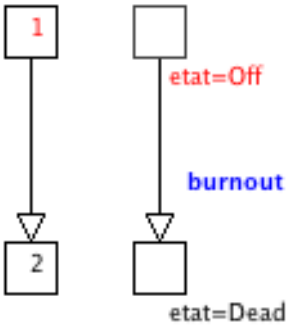



FIGURE 40: Diagramme d'états montrant les contions logiques

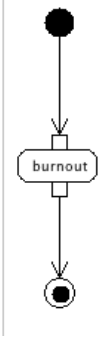



FIGURE 41: Diagramme d'actions

#### 1.4.5 Générateur de code :non
Il n'y a pas de générateur de code.

#### 1.4.6 Simulateur :non
Pour le moment IVY ne dispose pas de simulateur mais dans les mois qui viennent un simulateur du nom de aniMAL sera disponible.

#### 1.4.7 Mise à jour :2012
La dernière mise à jour de l'outil de IVY date de mai 2012. Il faut remarquer que l'outil de IVY est toujours en cours de développement.

## 2 Les critères d'analyse

### 2.1 Bagage théorique nécessaire

#### 2.1.1 Valeur :2
Le bagage théorique nécessaire pour utiliser IVY de manière efficace comprend la maîtrise du langage MAL [25], des notions d'UML et une compréhension des méthodes de vérification CTL et LTL utilisée en model checking.

### 2.2 Expressivité du formalisme

#### 2.2.1 Turing complet : **non**

Le formalisme IVY n'est pas Turing complet.

#### 2.2.2 Type de logique : **LTL/CTL**

L'outil IVY traduit ses modèles en NuSMV. NuSMV utilise les opérateurs modaux de la logique temporelle arborescente (A [26] et E [27]) afin d'exprimer le fait qu'à chaque instant il peut y avoir plusieurs branches possibles et pour chaque branche NuSMV utilise les opérateurs de logique temporelle linéaire (X [28], G [29] et F [30]).

#### 2.2.3 Type de preuve : **model checking**

La technique de preuve utilisée par NuSMV est le model checking.

25. Modal Action Logic
26. A : modalité universelle désignant tous les chemins possibles à partir de l'instant courant
27. E : modalité existentielle désignant un chemin possible et existant à partir de l'instant courant
28. X : symbolisant l'état suivant d'une branche
29. G : symbolisant tous les états suivants d'une branche
30. F : symbolisant la feuille d'une branche

### 2.3 Interrogation du système formalisé

**2.3.1** Trace d'exécution : **non**

Pour le moment avec l'outil IVY, il n'est pas possible de simuler des traces d'exécution car l'outil aniMAL n'est pas encore disponible.

**2.3.2** Détection de deadlocks : **oui**

L'outil NuSMV permet la détection automatisée de deadlocks.

**2.3.3** Evaluation de la liveness : **oui**

L'outil NuSMV permet l'évaluation automatisée de la liveness.

**2.3.4** Evaluation de la reachability : **oui**

L'outil NuSMV permet l'évaluation automatisée de la reachability

## 3 Le critère de maintenance

### 3.1 Maintenabilité du système formalisé

**3.1.1** héritage : **oui**

Les interacteurs peuvent hériter des attributs, des actions et des axiomes d'autres interacteurs.

**3.1.2** polymorphisme : **non**

IVY ne permet pas d'exprimer le polymorphisme.

**3.1.3** encapsulation : **oui**

Un manière d'utiliser de l'encapsulation avec IVY serait de créer un unique interacteur $I_u$ communiquant avec un ensemble $E_m$ d'interacteurs. Les actions et les attributs des interacteurs de $E_m$ devraient être invisibles aux utilisateurs et les utilisateurs seraient dès lors obligés d'interagir avec $I_u$ pour modifier les attributs d'un quelconque interacteur de $E_m$.

**3.1.4** récursion : **non**

IVY ne permet pas de définir la récursion.

**3.1.5** hiérarchisation : **oui**

IVY utilisent les notations UML pour décrire les liens qu'il y a entre les différents interacteurs.

**3.1.6** emboîtement : **oui**

L'emboîtement basique et l'emboîtement mémorisé se formulent en IVY par des attributs supplémentaires permettant de définir les sous-états des supers attributs [31].

## 7.2 Analyse pratique

### La modélisation de Microwave

La modélisation de Microwave en IVY se trouve en Annexe (**12.2**).

### L'analyse de la modélisation de Microwave

#### La conception du modèle

Pour modéliser Microwave avec IVY, nous avons défini trois interacteurs appelé `mode`, `display` et `light`. Ces trois interacteurs représentent les trois entités concurrentes du système Microwave. La particularité de IVY est qu'il faut impérativement définir un interacteur appelé `main` permettant de synchroniser les trois autres interacteurs. C'est pourquoi par le biais de l'interacteur `main` nous synchronisons `light` et `display` sur le noyau fonctionnel (`mode`). Toutes les entrées d'utilisateur sont représentées par des actions précédées du préfixe `[vis]` signifiant que celles-ci sont visibles. Les sorties du système sont quant à elles définies par les attributs précédés également par le préfix `[vis]` signifiant qu'ils sont visibles. Les entrées du système sont les actions perçues par l'interacteur Mode et les sorties du système sont les attributs des interacteurs `display` et `light`.

#### L'analyse du modèle

L'outil de IVY traduit automatiquement le système formalisé en NuSMV ce qui nous permet d'utiliser des techniques de vérification formelle de type model checking (LTL et CTL). Les vérifications de spécifications formelles prises en charge par NuSMV se font de manière automatisées.

#### La maintenance du modèle

La modélisation de Microwave a nécessité l'utilisation d'emboîtement basique et d'emboîtement mémorisé. L'emboîtement basique et l'emboîtement mémorisé se formulent en IVY par des attributs supplémentaires permettant de définir les sous-états des supers attributs. Il faut remarquer que l'encapsulation de Microwave aurait pus être modélisée par la définition d'un interacteur `Controller` se chargeant de recevoir directement toutes les

31. les supers états sont représentés par des supers attributs et les sous-états par sous-attributs

actions visibles d'utilisateur pour les communiquer à l'interacteur `Mode` qui lui se chargerait de synchroniser les deux autres entités concurrentes.

## 7.3 Conclusion

Le formalisme IVY permet de formaliser toutes les IHMs que nous considérons dans ce travail avec une certaine facilité pour maintenir le système formalisé. Il faut remarquer qu'IVY dispose d'un traducteur automatique permettant de traduire un système formalisé avec IVY en NuSMV afin d'appliquer sur les spécifications formelles de ce système des vérifications automatisées en LTL et en CTL. C'est précisément NuSMV qui va permettre d'éviter un bon nombre de tests coûteux lors de la validation du système conceptualisé du fait qu'il a permis de déceler des traces incorrectes du système dés la conception du modèle.

# 8 Analyse de ICO

## 8.1 Analyse théorique

### 1 Les critères de conception

#### 1.1 Structuration de l'information

**1.1.1** Aspects statiques : **oui**

ICO utilise l'approche objet pour décrire les aspects statiques ou structurels du système. Un objet CO [32] implémenté en CORBA-IDL [33] permet de faire le lien entre les interfaces [34] nécessaires pour implémenter le système interactif et les comportements du système interactif (ObSC) [25].

**1.1.2** Aspects comportementaux **oui**

Les comportements de chaque objet CO sont représentés par un ObSC qui est un réseau de Petri d'objets (OPN). Les transitions de ces réseaux de Petri sont labellisées par les actions d' un objet CO. Chaque classe d'un objet CO spécifie une ou plusieurs interfaces d'un composant du système et fait correspondre à une de ses actions une signature d'une des interfaces qu'il spécifie. Une transition peut se produire quand les places d'entrée sont peuplées d'un nombre de tokens suffisant. Chaque place du réseau de Petri est typée, c'est-à-dire que les jetons à l'intérieur des places doivent être du même type. À chaque fois qu'une transition se produit, l'action liée à la transition est exécutée sur les objets identifiées par les tokens consommés par la transition . Les transitions peuvent générer de nouveaux objets, supprimer des objets et mettre à jour des objets. Les objets modifiés et les nouveaux objets sont envoyés dans les places de sortie.

**1.1.3** Représentation explicite des entrées et des sorties : **oui**

-Pour les entrées, l'utilisateur ne peut interagir que par le biais de widgets et à chaque widget est associé une fonction d'activation qui permet de déclencher la transition associée lorsque celle-ci est disponible. Les interactions que permet de représenter ICO sont très simples et inadéquates pour modéliser des interactions complexes telles que celles que l'on rencontre avec les CLIs. ICO permet de déterminer un type particulier de données à chacune des places comprises dans un réseau de Petri relatif à un objet ICO. Le fait de pouvoir identifier le type de jeton présent dans chacune des places permet d'identifier la provenance des entrées fournies par l'utilisateur. Cette caractéristique est très intéressante pour modéliser les IHM3s car elle permet d'identifier la modalité entrante/-sortante concernée par chacune des interactions gérées par le noyau fonctionnel.

32. Cooperative Object
33. Interface Definition Language pour IDL et Common Object Request Broker Architecture pour CORBA
34. contenant les signatures des méthodes

-Pour les sorties, la fonction de rendu vise à présenter à l'utilisateur les changements d'état du système interactif. La fonction de rendu maintient la consistance entre l'état interne du système et son apparence externe en reflétant les changements d'états du système. La fonction de rendu est représentée par la configuration de toutes les places relatives à chaque objet ICO du système.

**1.1.4** Lien entre les I/O et les aspects statiques du noyau fonctionnel : **oui**

La fonction permettant de faire le lien entre la place [35], le widget, l'événement et le service rendu par le noyau fonctionnel est appelée fonction d'activation.

**1.1.5** Présentation de l'IHM : **non**

La visibilité des widgets n'est pas prise en compte par ICO.

### 1.2 Point de vue externe

**1.2.1** Satisfaction : **3**

**1.2.2** Fiabilité : **2**

**1.2.3** Apprentissage : **3**

**1.2.4** Efficacité : **3**

### 1.3 Parallélisme

**1.3.1** Description d'entités concurrentes : **oui**

En ICO, une entité concurrente peut soit être représentée par un objet ICO qui ne spécifie qu'une seule interface [36], soit par plusieurs objets différents pris en charge par un même objet coopératif [37]

**1.3.2** Moyens de communication : **oui**

La communication entre deux objets coopératifs se fait par le biais d'un protocole client/-server permettant de transmettre des jetons entre leur ObSC respectif. Sur la FIGURE 42, nous avons représenté une synchronisation entre deux entités concurrentes appartenant à un même réseau de Petri (ObSC) par une transition $t1$ prenant en entrées deux places contenant chacune des jetons de type différents ($< x >$ et $< y >$). Dés que cette transition est enclenchée, une action est réalisée sur les deux jetons consommés, c'est-à-dire

35. déterminant le type de données
36. signatures de méthodes
37. spécifiant l'interface relative à chacun de ces objets

qu'une méthode $m1$ est opérée sur le jeton de type $< x >$ et une méthode $m2$ est opérée sur l'autre jeton de type $< y >$. Le fait de pouvoir séparer les entités concurrentes en plusieurs objets coopératifs permet de ne pas surcharger la représentation du réseau de Petri englobant tout le système.

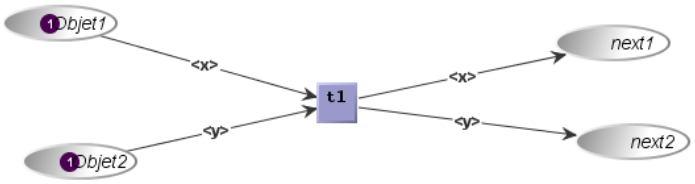



FIGURE 42: Synchronisation en ICO

### 1.3.3 Expression du temps : **partiellement**

ICO permet de représenter le temps en utilisant une place contenant des jetons pour représenter une sorte de sablier où les jetons sont des grains de sable consommés par une transition enclenchée à intervalles réguliers et sur laquelle toutes les transitions du réseau de Petri doivent se synchroniser.

## 1.4 Ressources mises à disposition

Documents

### 1.4.1 Tutoriel : oui

Le document [22] est un tutoriel complet du formalisme et de l'outil de ICO.

### 1.4.2 Date de la dernière publication :2008

Le document [27] est la dernière publication traitant de ICO.

Outils

### 1.4.3 Editeur : partiellement

L'éditeur de Petshop est un outil graphique qui permet, d'une part, de dessiner les réseaux de Petri et, d'autre part, il permet de spécifier le type des jetons attendus dans chaque place et pour chaque transition. Petshop dispose également d'une fenêtre interface avec laquelle on peut lire la liste des services fournis par une classe ICO. Il faut remarquer cependant que l'importation d'interfaces n'est pas encore totalement opérationnelle avec l'outil Petshop.

### 1.4.4 Vérificateur : partiellement

L'outil Petshop dispose d'un vérificateur qui détecte les invariants et les places à capacité

infinie.

#### 1.4.5 Générateur de code : non
L'outil Petshop ne dispose pas de générateur de code.

#### 1.4.6 Simulateur : oui
L'outil Petshop dispose d'un simulateur qui permet à partir d'un état initial défini par une configuration de jetons dans chaque place d'actionner les transitions disponibles.

#### 1.4.7 Mise à jour : 2008
La dernière mise à jour date de 2008. Il faut remarquer que l'outil ICO est encore en cours de développement.

## 2 Les critères d'analyse

### 2.1 Bagage théorique nécessaire

#### 2.1.1 Valeur : 2
Le bagage théorique requis pour être capable d'utiliser ICO comprend la compréhension de la programmation orientée objet pour l'expression des aspects statiques du système et la connaissance des réseaux de Petri étendu pour la description des comportements du système.

### 2.2 Expressivité du formalisme

#### 2.2.1 Turing complet :**oui**

La programmation orientée objet et les réseaux de Petri étendu sont Turing-complets. Il faut remarquer que les simple réseaux de Petri ne sont pas Turing complet mais les réseaux de Petri étendu permettant d'exprimer les arcs inhibiteurs eux sont Turing complet, ce qui est le cas pour ICO. Un arc inhibiteur impose qu'une transition soit réalisée seulement si la place d'entrée de la transition est vide, ce genre de transitions permet d'exprimer des conditions sur le nombre de jetons des places.

#### 2.2.2 Type de logique : **Réseaux de Petri**

ICO utilise comme logique sous-jacente pour raisonner sur les comportements d'un système interactif les réseaux de Petri.

#### 2.2.3 Type de preuve : **model checking**

Les réseaux de Petri utilisent le model checking comme technique de preuve.

### 2.3 Interrogation du système formalisé

**2.3.1** Trace d'exécution : **oui**

À partir du système formalisé, l'outil ICO permet de simuler les traces d'exécution d'objets CO. La représentation visuelle des réseaux de Petri nous permet d'observer l'état du système après chaque transition.

**2.3.2** Détection de deadlocks : **non**

L'outil ICO ne permet pas de détecter de manière automatisée des deadlocks du système formalisé.

**2.3.3** Evaluation de la liveness : **non**

L'outil ICO ne permet pas d'évaluer de manière automatisée la liveness du système formalisé.

**2.3.4** Evaluation de la reachability : **non**

L'outil ICO ne permet pas d'évaluer de manière automatisée la reachability du système formalisé.

## 3 Le critère de maintenance

### 3.1 Maintenabilité du système formalisé

**3.1.1** héritage : **oui**

L'héritage est assuré par le langage orienté objet.

**3.1.2** polymorphisme : **oui**

Le polymorphisme est assuré par le langage orienté objet.

**3.1.3** encapsulation : **oui**

L'encapsulation est assurée par le langage orienté objet.

**3.1.4** récursion : **oui**

La récursion est assurée par le langage orienté objet.

**3.1.5** hiérarchisation : **oui**

La hiérarchisation est assurée par le langage orienté objet.

**3.1.6** emboîtement : **oui**

Les emboîtements dotés d'une mémoire sont faciles à exprimer grâce aux arcs de test permettant de tester la présence de jetons sans les consommer. Sur la FIGURE 43, nous avons illustré un exemple d'emboîtement doté d'une mémoire. Les emboîtements basiques sont beaucoup plus compliqués à exprimer du fait qu'il faut supprimer tous les jetons des places présentes dans le super état. Sur la FIGURE 44, nous avons représenté de l'emboîtement basique avec ICO. On remarque sur cette illustration que tous les sous-états de *Super*1 doivent posséder une transition vers *Super*2. Nous considérons également que *Super*1 est un sous-état de lui-même, cette considération est correcte car la place initiale du jeton lorsqu'il accède à *Super*1 est en fait son premier sous-état. Sur la FIGURE 44, les deux places *Super*1 représentent la même place mais avec des positions différentes.

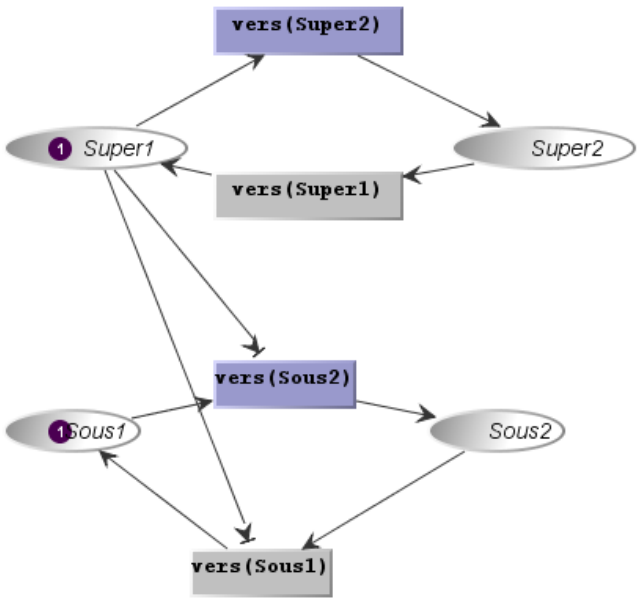



FIGURE 43: Emboîtement mémorisé en ICO

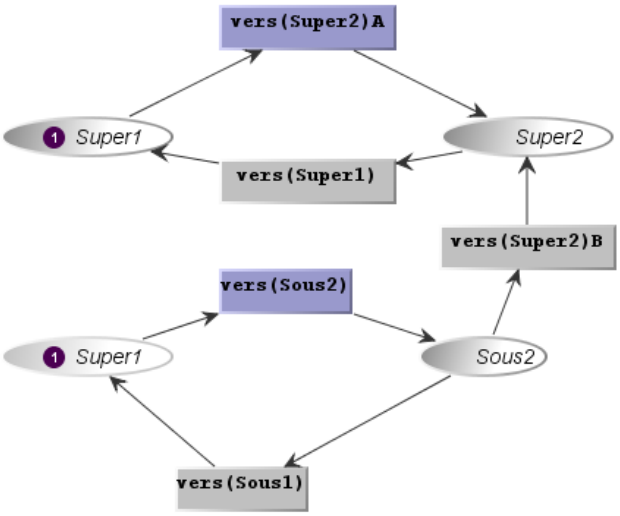



FIGURE 44: Emboîtement basique en ICO

## 8.2 Analyse pratique

### La modélisation de Microwave

La modélisation de Microwave en ICO se trouve en Annexe (**12.3**).

### L'analyse de la modélisation

#### La conception du modèle

Pour modéliser Microwave avec ICO, nous avons défini les trois interfaces ModeI, DisplayI et LightI contenant chacune les signatures des méthodes nécessaires à l'implémentation de ces trois entités concurrentes. Sur base de ces trois interfaces, nous avons pu construire l'objet coopératif Micvrowave-CO permettant de spécifier par la commande `specifies` les trois interfaces qu'elle prend en charge. L'objet coopératif Microwave se compose de quatre éléménts : une classe Microwave-CO, un ObCS représentant les comportements de l'objet coopératif par un réseau de Petri, une fonction d'activation faisant le lien entre les entrées et les méthodes du noyau fonctionnel et la fonction de rendu faisant le lien entre les transitions du système et les sorties. Nous avons utilisé un seul objet coopératif pour décrire les trois entités concurrentes, c'est pourquoi nous avons représenté leur compotretment respectif sur le même réseau de Petri (ObSC).

#### L'analyse du modèle

L'outil Petshop ne permet pas de réaliser une analyse très poussée du système formalisé, seuls les invariants du réseau de Petri sont calculés automatiquement par Petshop. Il faut remarquer qu'un simulateur permet de tester les traces du système.

#### La maintenance du modèle

La modélisation de Microwave avec ICO a nécessité de l'emboîtement basique et mémorisé qui se formulent en ICO par le biais d'arc de test [38] et d'arcs normaux. L'utilisation du polymorphisme en utilisant les mêmes noms de méthodes pour les trois types d'objets présents sur le réseau (<`Mode`>, <`Display`>, <`Light`>) nous a permis de rendre plus lisible la classe Microwave-ICO.

## 8.3 Conclusion

Le formalisme ICO permet de modéliser toutes les IHMs considérées dans ce travail avec une grande facilité pour maintenir ses modèles du fait qu'il possède toutes les caractéristiques du langage orienté objet pour définir un objet coopératif. ICO dispose d'une caractéristique peu fréquente dans la modélisation d'IHM à savoir la prise en compte des aspects statiques du noyau fonctionnel. Cette caractéristique permet de faire facilement

38. un arc de test permet d'activer une transition en fonction de la présence d'un nombre spécifié de jetons sans les consommer

le lien entre les interactions gérées par le système interactif et les services rendus par le noyau fonctionnel. Ceci, d'un point de vue génie logiciel, permet de faciliter la tâche qui consiste à choisir une architecture adéquate pour le système après la modélisation. Le problème majeur de ICO est que son outil ne permet de vérifier, de manière automatisée, qu'un nombre très limité des propriétés du système formalisé, à savoir les invariants des réseaux du système. Ce désagrément rend la modélisation en ICO peu rentable car tous les efforts fournis pour modéliser les comportements du système de manière rigoureuse ne permettent pas de tirer la moindre conclusion quant à la vérification du système.

# 9 Comparaison des analyses

Dans ce chapitre, nous allons construire, pour chaque famille de critères, un tableau récapitulant l'évaluation théorique des différents formalismes considérés, et sur base des besoins des IHMs nous distinguerons les caractéristiques [39] relatives aux CLIs, aux GUIs et aux IHM3s. Les caractéristiques relatives aux CLIs, aux GUIs et aux IHM3s seront notées en noir. Les caractéristiques relatives aux GUIs et aux IHM3s seront notées en bleu et les caractéristiques uniquement relatives aux IHM3s seront notées en vert.

## 9.1 Comparaison de l'évaluation des critères de conception

### Tableau des critères de conception

| Les critères de conception | | | | |
|---|---|---|---|---|
| **Critères** | **Caractéristiques** | **VEG** | **IVY** | **ICO** |
| **Structuration** | Aspects statiques | non | non | oui |
| | Aspects comportementaux | oui | oui | oui |
| | Aspects visuels | non | non | oui |
| | I/O | oui | oui | oui |
| | Lien entre I/O et noyau fonctionnel | non | non | oui |
| **Point de vue externe** | Satisfaction | 4 | 3 | 3 |
| | Fiabilité | 3 | 3 | 2 |
| | Apprentissage | 4 | 2 | 3 |
| | Efficacité | 3 | 4 | 3 |
| **Parallélisme** | Entités concurrentes | oui | oui | oui |
| | Moyens de communication | oui | oui | oui |
| | Expression du temps | partiel | partiel | partiel |
| **Documents** | Tutoriel | partiel | partiel | oui |
| | Date | 2004 | 2009 | 2008 |
| **Outils** | Editeur | partiel | oui | oui |
| | Vérificateur | partiel | oui | partiel |
| | Générateur | oui | non | non |
| | Simulateur | non | non | oui |
| | Date | 2001 | 2012 | 2008 |

### Interprétations de l'évaluation des critères de conception

Pour les CLIs :

39. les caractéristiques sont les différents aspects pris en compte par chaque critère

En observant le tableau, on remarque que le formalisme le plus adéquat pour conceptualiser des modèles de CLI est VEG par le fait qu'il permet de représenter explicitement les entrées et les sorties complexes [40] d'une CLI. De plus le fait d'utiliser des règles de production d'une grammaire pour définir les transitions d'états d'un système permet de raisonner sur l'agencement des entrées et des sorties du système. Théoriquement VEG permet de traduire ses modèles en SPIN afin d'assurer une vérification automatisée de ceux-ci. Il faut cependant remarquer que IVY et ICO permettent également de modéliser des CLIs mais de manière moins adéquate car ils permettent difficilement de décrire des interactions complexes.

Pour les GUIs :

En observant le tableau, on remarque que les trois formalismes sont adaptés pour conceptualiser des modèles de GUI, mais chacun d'entre eux dispose de ses propres spécificités. ICO dispose d'un simulateur permettant de générer les traces d'exécution du système en cours de modélisation tandis que VEG et IVY disposent d'un vérificateur permettant d'automatiser la preuve [41] des modèles qu'ils permettent de définir. On remarque cependant que aucun de ces trois formalismes ne tient compte de la présentation d'une IHM. Ce qui peut être un réel écueil de la part de VEG, de IVY et de ICO car lorsqu'on utilise une GUI, on remarque très rapidement que la visibilité des widgets est indispensable pour permettre à l'utilisateur de comprendre l'état du système. Le fait de ne pas tenir compte de la présentation revient à ne pas tenir compte des modalités de sortie du système qui sont pourtant fondamentales dans l'interaction Homme-Machine.

Pour les IHM3s :

En observant le tableau, on remarque que les trois formalismes sont adaptés pour conceptualiser des modèles d'IHM3 mais chacun dispose de ses propres spécificités dépendant du fait que l'IHM3 se compose essentiellement de GUIs ou de CLIs. Il est cependant intéressant de remarquer que le seul formalisme permettant de prendre en compte les aspects statiques du noyau fonctionnel est ICO car il permet de faire le lien entre les interfaces du noyau fonctionnel d'un système interactif et l'IHM de ce système interactif. Cette caractéristique dont dispose ICO va permettre de faciliter la conception du système interactif car sa modélisation fait directement le lien entre les interactions IHM-utilisateur et les méthodes invoquées par le noyau fonctionnel.

40. succession de caractères formant des mots
41. en faisant du model checking

## 9.2 Comparaison de l'évaluation des critères d'analyse

### Tableau des critères d'analyse

| Les critères d'analyse | | | | |
|---|---|---|---|---|
| **Critères** | **Caractéristiques** | **VEG** | **IVY** | **ICO** |
| **Bagage théor.** | Valeur | 3 | 2 | 2 |
| **Expressivité du formalisme** | Turing-complet | non | non | oui |
| | Type de logique | logique modale | | |
| | Type de preuve | model checking | | |
| **Interrogation du système formalisé** | Simulation de traces | non | non | oui |
| | Deadlocks | partiel | oui | non |
| | Liveness | partiel | oui | non |
| | Reachability | partiel | oui | non |

### Interprétation de l'évaluation des critères d'analyse

Il faut remarquer que l'analyse d'un système formalisé dépend de la capacité du formalisme à concevoir le modèle du système car c'est précisément à partir de ce modèle qu'on effectue l'analyse du système. C'est pourquoi la détermination du formalisme le plus adéquat pour analyser les différentes classes d'IHMs modélisées va dépendre des résultats récoltés lors de l'évaluation des critères de conception.

Pour les CLIs :

En théorie, VEG dispose de SPIN qui est un outil puissant pour assurer une vérification automatisée. Cependant l'outil de VEG ne permet pas en pratique de traduire ses modèles en SPIN.

Pour les GUIs et les IHM3s :

En observant le tableau et en nous basant sur les résultats récoltés lors de l'évaluation des critères de conception, on remarque que le formalisme le plus adéquat pour automatiser la vérification de la safety, de la liveness et de la reachability est IVY car il dispose d'un outil puissant de vérification qui est NuSMV. En théorie, VEG dispose de SPIN qui est également un outil puissant pour assurer une vérification automatisée. Cependant l'outil de VEG ne permet pas en pratique de traduire ses modèles en SPIN. ICO est un formalisme très expressif permettant de formaliser tous les langages calculables mais son outil ne dispose d'aucune technique automatisée permettant de vérifier ses modèles. Ceci rend la modélisation formelle de ICO peu rentable.

## 9.3 Comparaison de l'évaluation des critères de maintenance

### Tableau des critères de maintenance

| Les critères de maintenance | | | | |
|---|---|---|---|---|
| **Critères** | **Caractéristiques** | **VEG** | **IVY** | **ICO** |
| **Maintenabilité** | Héritage | oui | oui | oui |
| | Polymorphisme | non | non | oui |
| | Encapsulation | oui | oui | oui |
| | Récursion | non | non | oui |
| | Hiérarchisation | partiel | oui | oui |
| | Emboîtement | oui | oui | oui |

### Interprétation de l'évaluation des critères de maintenance

Il faut remarquer que la maintenance d'un système formalisé dépend également de la capacité du formalisme à concevoir le modèle du système car c'est précisément à partir de ce modèle qu'on effectue la maintenance du système formalisé. C'est pourquoi la détermination du formalisme le plus adéquat pour maintenir les différentes classes d'IHMs modélisées va dépendre des résultats récoltés lors de l'évaluation des critères de conception.

Pour les CLIs

En observant le tableau et en nous basant sur les résultats récoltés lors de l'évaluation des critères de conception, on remarque que IVY par rapport à VEG dispose d'une meilleure vue d'ensemble sur les systèmes qu'il formalise. Cette constatation est liée au fait qu'IVY utilise des notations UML pour définir les liens qu'il y a entre les différents interacteurs ainsi que pour détailler leurs actions et leurs attributs relatifs à chaque interacteur. Cependant vu que VEG permet de modéliser les CLIs de manière plus adéquate qu'IVY, nous privilégierons VEG pour maintenir les modèles qu'il permet de formaliser.

Pour les GUIs et les IHM3s :

En observant le tableau et en nous basant sur les résultats récoltés lors de l'évaluation des critères de conception, on remarque qu'ICO est le plus adéquat pour maintenir le système formalisé car il dispose de toutes les caractéristiques dont bénéficie un langage orienté objet pour faire de la maintenance.

# 10 Conclusions et contributions

## 10.1 Contribution

Ce travail catégorise les différentes classes d'IHMs en fonction de leur aspect (CLI ou GUI) et en fonction de leur forme (monomodale ou multimodale). Cette classification des IHMs a permis de déterminer une grille de critères pour comparer des formalismes d'IHM dans le but de déterminer le plus adéquat pour systématiser la conception d'un système interactif.

## 10.2 La conception du modèle

On remarque que le choix du formalisme va dépendre essentiellement des critères de conception et de l'IHM du système interactif que nous souhaitons modéliser car c'est précisément à partir de la conception du modèle que va pouvoir se réaliser l'analyse et la maintenance du système. C'est pourquoi un formalisme répondant aux critères de conception permettra de faciliter la modélisation d'un système interactif. Sur la FIGURE 45, nous illustrons l'influence qu'a un formalisme répondant aux critères de conception lors de la modélisation du système interactif sur les phases de conception définies en génie logiciel.


FIGURE 45: Phases de conception

### Les CLIs

En se basant sur les besoins des CLIs et sur les caractéristiques comportementales des différentes familles de formalismes, on remarque que la famille de formalismes la plus adéquate pour modéliser les CLIs est la famille des grammaires. Cette constatation se justifie par le fait que les entrées complexes d'une CLI définissent les transitions d'états de la CLI et que pour être modélisées, ces transitions d'états de la CLI nécessitent une prise en compte de l'ensemble des caractères du langage et d'un algorithme permettant de décider de l'appartenance d'une séquence de caractères dans ce langage. Cette constatation ne signifie pas que les autres familles de formalismes ne permettent pas de modéliser les CLIs mais témoigne bien du fait que les caractéristiques inhérentes aux grammaires répondent de manière adéquate aux besoins des CLIs.

### Les GUIs

Au contraire des CLIs, les GUIs forment un ensemble hétérogène d'IHMs dont il n'est pas possible de déterminer la famille de formalismes la plus adéquate tant les GUIs se

déclinent sous des formes variées [42]. Cependant en analysant les besoins des GUIs, on remarque que leur modélisation nécessite impérativement la possibilité de décrire des entités concurrentes avec toutes les implications qui leurs sont associées [43]. Après avoir analysé plusieurs méthodes formelles comportementales, nous remarquons que généralement celles-ci ont tendance à ne pas tenir compte de l'aspect visuel de l'IHM. Ce qui peut être un réel écueil car lorsqu'on utilise une GUI, on remarque très rapidement que la visibilité des widgets est indispensable pour permettre à l'utilisateur de comprendre l'état du système. Le fait de ne pas tenir compte de la présentation revient à ne pas tenir compte des modalités de sortie du système qui sont pourtant fondamentales dans l'interaction entre le système et l'homme.

### Les IHM3s

De même que pour les GUIs, les IHM3s forment un ensemble hétérogène d'IHMs dont il n'est pas possible de déterminer la famille de formalismes la plus adéquate. De manière générale, on remarque en génie logiciel que de nombreuses méthodes formelles et semi-formelles séparent complètement la phase de modélisation et d'architecture. Cette séparation peut engendrer un gouffre entre la phase de modélisation du système interactif et les phases ultérieures qui reposent sur cette modélisation. Ce gouffre se crée parce qu'après la modélisation formelle du système interactif, il est très difficile d'établir le lien qu'il y a entre les interactions survenant à travers l'IHM et la manière dont celles-ci sont prises en compte par le noyau fonctionnel. Une manière d'éviter ce gouffre est d'intégrer l'architecture du noyau fonctionnel dans la modélisation de l'IHM comme le fait ICO en associant à chaque interaction utilisateur-IHM une méthode invoquée par le noyau fonctionnel du système interactif. Sur la FIGURE 46, nous illustrons l'influence de la prise en compte des aspects statiques du noyau fonctionnel lors de la modélisation du système interactif sur les phases de conception définies en génie logiciel.

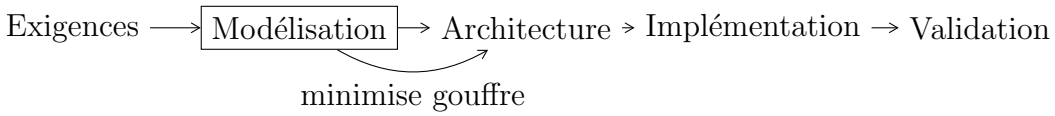



FIGURE 46: Phases de conception

## 10.3 L'analyse

Comme expliqué précédemment, l'analyse d'un système formalisé dépend de la modélisation du système, c'est pourquoi l'expressivité du formalisme joue un rôle considérable dans le choix du formalisme car si le formalisme est trop peu expressif il est possible qu'il ne permette pas de décrire certains systèmes. De manière générale, un formalisme expressif

42. menu déroulant, forme fill-in, manipulation directe,...
43. synchronisation, communication et représentation du temps

sera souvent combiné avec une logique sous-jacente dans laquelle les modèles du formalisme seront traduits. Cette logique sous-jacente permet d'assurer la décidabilité logique des propriétés que celle-ci permet de vérifier. Il est important de remarquer qu'ici intervient le théorème de Gödel qui exprime le fait que plus un formalisme est expressif plus la logique dans laquelle il peut être traduit algorithmiquement tend à être indécidable. Un exemple de ce genre de compromis entre expressivité et décidabilité logique est la logique modale qui, elle, bénéficie d'une expressivité supérieure à la logique des prédicats grâce à ces deux opérateurs modaux [44] tout en étant logiquement décidable. La logique modale est une logique fréquemment utilisée pour faire du model checking. Le model checking semble être la technique de vérification de spécifications formelles la plus adaptée pour vérifier des modèles de système interactif car elle permet d'implémenter des model checkers avec lesquels il est possible d'automatiser la preuve. L'application de l'automatisation de la preuve dés la modélisation formelle des systèmes interactifs va permettre de systématiser leur vérification avant même de les avoir implémenter et permettre dès lors de minimiser les tests manuels coûteux (white box et black box) visant à les valider. Sur la FIGURE 47, nous illustrons l'influence de l'automatisation de la vérification des spécifications formelles lors de la modélisation du système interactif sur les phases de conception définies en génie logiciel.

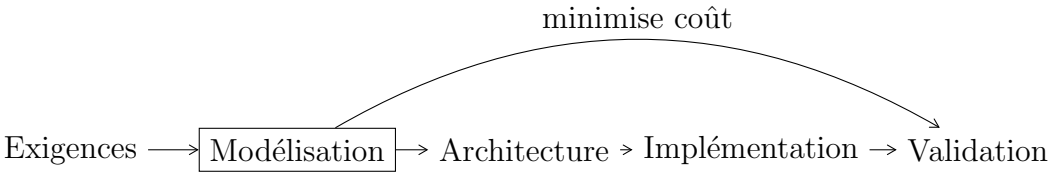



FIGURE 47: Phases de conception

## 10.4 La maintenance

Une IHM est par définition un point de convergence entre l'homme et la machine, plus l'interface est adaptée aux exigences de l'homme plus celle-ci rend le système utilisable. Aujourd'hui les possibilités d'interaction entre un homme et une machine ne cessent de s'améliorer en vue de répondre aux exigences de tout un chacun. On constate que les exigences peuvent fortement varier d'une personne à l'autre ce qui se traduit évidemment par de nombreuses modifications du système interactif, et par conséquent de son modèle. C'est pourquoi un formalisme adapté pour supporter de multiples modifications, permet de mieux prendre en compte les exigences variables du client. Sur la FIGURE 48, nous illustrons l'influence de l'utilisation d'un formalisme adapté à supporter des modifications lors de la modélisation du système interactif sur les phases de conception d'un système.

44. nécessaire et possible


FIGURE 48: Phases de conception

# 11 Conclusion générale

La grille de critères que nous avons élaborée apporte aux concepteurs de systèmes interactifs un outil analytique pour réaliser la phase cruciale que constitue la modélisation d'un système. Cette grille, par ses familles de critères, détermine l'influence d'un formalisme sur les phases de conception d'un système, et par les classes d'IHMs qu'elle distingue, discerne les critères nécessaires que doit remplir un formalisme pour modéliser un système interactif doté d'une IHM particulière. C'est pourquoi, à partir de cette grille, un concepteur pourra, en fonction de l'IHM du système interactif qu'il veut modéliser et des phases de conception qu'il désire privilégier, comparer objectivement les formalismes mis à sa disposition.

Dans ce travail, nous nous sommes intéressés aux méthodes formelles utilisées lors de la modélisation de systèmes interactifs. Il serait maintenant intéressant d'établir une grille de critères permettant de comparer les méthodes formelles utilisées pour les autres phases de la conception d'un système interactif afin de permettre au concepteur d'orienter ses choix tout au long de la conception d'un système.

# 12 Annexes

Dans ces annexes, nous fournissons les modélisations de Microwave en VEG, IVY et ICO.

## 12.1 Modélisation de Microwave en VEG

### Notation textuelle

```
Model Light
        Axioms off
        off::=?en(cook) \lampe_allumee on
        on::=?ex(cook) \lampe_eteinte off
End Light

Model Display
        Axioms time
        time::=?en(program) \affiche_compteur counter
                |? en(set_time_error) \affiche_erreur error
        counter::=?power \affiche_puissance power
                |?en(idle) \affiche_temps time
        power::=?digit \affiche_compteur counter
                |? en(idle) \affiche_temps time
        error::=?en(idle) \affiche_temps time
End Display

Model Mode(light,display)
        Axioms begin
        begin::=launch(program_mode=Program.setting) idle
        idle::=<digit> !program_mode.digit !light.en(program)
                        program
                |<minute> !light.en(cook)
                        launch(cook_mode=Cook.realcook) cook
                |<clock> launch(set_time_mode=SetTime.setting)
                        set_time
                |<open> open_idle
        open_idle::=<close> idle
        program::=<start> !light.en(cook)
                        launch(cook_mode=Cook.realcook) cook
                |<stop> !display.en(idle) idle
                |<digit> !program_mode.digit program
                |<power> !program_mode.power program
                |<open> open_program
        open_program::=<close> program
        set_time::=<clock(ok_time)> !set_time_mode.quit idle
                |<stop> !set_time_mode.quit idle
                |<digit> !set_time_mode.digit set_time
                |<clock(!ok_time)>
                        !set_time_mode.clock(!ok_time) set_time
```

```
                |<open> open_set_time
        open_set_time::=<close> set_time
        cook::=<stop> !light.ex(cook) !cook_mode.quit program
                |<done> !light.ex(cook)
                        !display.en(idle) !cook_mode.quit idle
                |<minute> !cook_mode.minute cook
                |<open> open_cook
        open_cook::=<close> cook
End Mode

Model Program
        Axioms setting
        setting::=?digit setting
                |?power power
        power::=?digit setting
End Program

Model SetTime
        Axioms setting
        setting::=?digit setting
                |?clock(!ok_time) error
                |?quit
        error::=?quit
End SetTime

Model Cook
        Axioms realcook
        realcook::=?minute realcook
        |?quit
End Cook

Model Controller
        Axioms begin
        begin::=launch(light=Light.off
                        display=Display.time mode=
                        Mode(light, display).idle) actif
        actif::=<quit>
End Controller
```

## 12.2 Modélisation de Microwave en IVY

### Notation textuelle

```
types

        typedis = {Disabled, Operational}
        typeoper = {SetTime, Idle, Program, Cook}
        typeprog = {Not, Setting, Power}
        typesettime = {Not, Setting, Tilt}
        typelight = {On, Off}
        typedisp = {Time, NotTime}
        typenottime = {Not, Counter, Tilt, Power}

interactor mode

        attributes

                modedis: typedis
                modeoper: typeoper
                modeprog: typeprog
                modesettime: typesettime

        actions

                [vis] open
                [vis] close
                [vis] clock
                [vis] stopSetTime
                [vis] stopCook
                [vis] stopProgram
                [vis] digitSetTime
                [vis] digitProgram
                [vis] digitIdle
                [vis] power
                [vis] start
                [vis] minuteIdle
                [vis] minuteCook
                [vis] done
                [vis] clockok
                [vis] clocknotok

        axioms

                #etat initial

```

```
43                  [] modeoper=Idle & modedis=Operational
44                          & modeprog=Not & modesettime=Not
45
46                  #conditions de transition
47
48                  per(open)-> modedis=Operational
49                  per(close)-> modedis=Disabled
50                  per(clock)-> modedis=Operational & modeoper=Idle
51                  per(stopSetTime)->
52                          modedis=Operational & modeoper=SetTime
53                  per(stopCook)->
54                          modedis=Operational & modeoper=Cook
55                  per(stopProgram)->
56                          modedis=Operational & modeoper=Program
57                  per(digitSetTime)->
58                          modedis=Operational & modeoper=SetTime
59                          & modesettime=Setting
60                  per(digitProgram)->
61                          modedis=Operational & modeoper=Program
62                          & (modeprog=Setting | modeprog=Power)
63                  per(digitIdle)->
64                          modedis=Operational & modeoper=Idle
65                  per(start)->
66                          modedis=Operational & modeoper=Program
67                  per(minuteIdle)->
68                          modedis=Operational & modeoper=Idle
69                  per(minuteCook)->
70                          modedis=Operational & modeoper=Cook
71                  per(done)->
72                          modedis=Operational & modeoper=Cook
73                  per(clockok)->
74                          modedis=Operational & modeoper=SetTime
75                  per(power)->
76                          modedis=Operational & modeoper=Program
77                          & modeprog=Setting
78                  per(clocknotok)->
79                          modedis=Operational
80                          & modeoper=SetTime & modesettime=Setting
81
82                  #transitions
83
84                  modedis=Operational->[open]
85                          keep(modeoper,modeprog,modesettime)
86                          & modedis'=Disabled
87                  modedis=Disabled->[close]
88                          keep(modeoper,modeprog,modesettime)
```

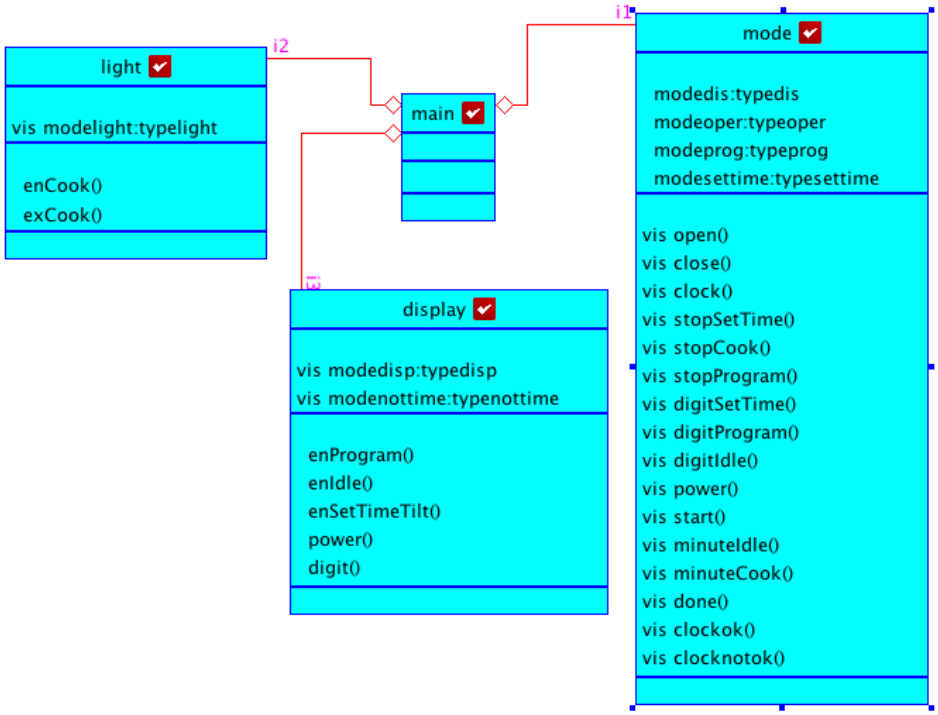
```
89                  & modedis'=Operational
90
91          modeoper=Idle->[digitIdle]
92                  keep(modedis,modesettime)
93                  & modeoper'=Program & modeprog'=Setting
94          modeoper=Idle->[clock]
95                  keep(modedis,modeprog) & modeoper'=SetTime
96                  & modesettime'=Setting
97          modeoper=Idle->[minuteIdle]
98                  keep(modedis,modeprog) & modeoper=Cook
99                  & modesettime=Not
100
101         modeoper=SetTime & modesettime=Setting ->[digitSetTime]
102                 keep(modeoper,modedis,modeprog,modesettime)
103         modeoper=SetTime & modesettime=Setting ->[clocknotok]
104                 keep(modeoper,modedis,modeprog)
105                 & modesettime'=Tilt
106         modeoper=SetTime->[stopSetTime]
107                 keep(modedis,modeprog,modesettime)
108                 & modeoper'=Idle
109
110         modeoper=Program->[stopProgram]
111                 keep(modedis,modeprog,modesettime)
112                 & modeoper'=Idle
113         modeoper=Program->[start]
114                 keep(modedis,modeprog,modesettime)
115                 & modeoper'=Cook
116         modeoper=Program & modeprog=Setting->[digitProgram]
117                 keep(modeoper,modedis,modeprog,modesettime)
118         modeoper=Program & modeprog=Power->[digitProgram]
119                 keep(modeoper,modedis,modesettime)
120                 & modeprog'=Setting
121         modeoper=Program & modeprog=Setting->[power]
122                 keep(modeoper,modedis,modesettime)
123                 & modeprog'=Power
124
125         modeoper=Cook->[stopCook]
126                 keep(modedis,modeprog,modesettime)
127                 & modeoper'=Program
128         modeoper=Cook->[done]
129                 keep(modedis,modeprog,modesettime)
130                 & modeoper'=Idle
131         modeoper=Cook->[minuteCook]
132                 keep(modeoper,modedis,modeprog,
133                         modesettime)
134
```

```
135 interactor light
136
137         attributes
138
139                 [vis] modelight: typelight
140
141         actions
142
143                 enCook
144                 exCook
145
146         axioms
147
148                 #etat initial
149
150                 [] modelight=Off
151
152                 #conditions de transition
153
154                 per(enCook)-> modelight=Off | modelight=On
155                 per(exCook)-> modelight=On
156
157                 #transitions
158
159                 modelight=Off->[enCook] modelight'=On
160                 modelight=On->[enCook] modelight'=On
161                 modelight=On->[exCook] modelight'=Off
162
163 interactor display
164
165         attributes
166
167                 [vis] modedisp: typedisp
168                 [vis] modenottime: typenottime
169
170         actions
171
172                 enProgram
173                 enIdle
174                 enSetTimeTilt
175                 power
176                 digit
177
178         axioms
179
180                 #etat initial
```

```
181
182                 [] modedisp=Time & modenottime=Not
183
184                 #conditions de transition
185
186                 per(enProgram)->modedisp=Time
187                 per(enIdle)->modedisp=NotTime
188                 per(enSetTimeTilt)->modedisp=Time
189                 per(power)->modedisp=NotTime & modenottime=Counter
190                 per(digit)->modedisp=NotTime & modenottime=Power
191
192                 #transitions
193
194                 modedisp=Time->[enProgram]
195                         modedisp'=NotTime & modenottime'=Counter
196                 modedisp=Time->[enSetTimeTilt]
197                         modedisp'=NotTime & modenottime'=Tilt
198                 modedisp=NotTime->[enIdle]
199                         keep(modenottime) & modedisp'=Time
200                 modedisp=NotTime & modenottime=Counter ->[power]
201                         keep(modedisp) & modenottime'=Power
202                 modedisp=NotTime & modenottime=Power ->[digit]
203                         keep(modedisp) & modenottime'=Counter
204
205 interactor main
206
207         aggregates
208
209                 mode via i1
210                 light via i2
211                 display via i3
212
213         axioms
214
215                 #synchronisation operational et light
216
217                 effect(i1.start)<->effect(i2.enCook)
218                 effect(i1.stopCook)<->effect(i2.exCook)
219                 effect(i1.minuteIdle)<->effect(i2.enCook)
220                 effect(i1.done)<->effect(i2.exCook)
221
222                 #synchronisation operational et display
223
224                 effect(i1.digitIdle)<->(i3.enProgram)
225                 effect(i1.stopCook)<->(i3.enProgram)
226                 effect(i1.done)<->(i3.enIdle)
```

```
effect(i1.stopProgram)<->(i3.enIdle)
effect(i1.digitProgram)<->(i3.digit)
effect(i1.power)<->(i3.power)
```

FIGURE 49: Hiérarchisation de Microwave



## 12.3 Modélisation de Microwave avec ICO

### Les interfaces de Microwave

```
1  public interface ModeI {
2
3          void open();
4          void close();
5          void clock():
6          void error_clock();
7          void valid_clock();
8          void power();
9          void digit();
10         void stop_SetTime();
11         void stop_Program();
12         void stop_Cook();
13         void start();
14         void minute();
15         void done();
16
17 }
```

FIGURE 50: Interface de Mode

```
1  public interface DisplayI {
2
3          void power();
4          void digit();
5          void stop_Error();
6
7  }
```

FIGURE 51: Interface de Display

```
1 public interface LightI {
2
3         void start();
4         void stop_Cook();
5         void minute();
6         void done();
7
8 }
```

FIGURE 52: Interface de Light

## L'objet coopératif Microwave-CO

Classe CO de Microwave

```
1  class Microwave-CO
2  specifies IMode, IDisplay, ILight{
3          place Idle <Mode>={<new Mode()>};
4          place Open_Idle <Mode>;
5          place SetTime <Mode>;
6          place Open_SetTime <Mode>;
7          place Program <Mode>;
8          place Open_Program <Mode>;
9          place Setting_P <Mode>;
10         place Power_P <Mode>;
11         place Real_Cook <Mode>;
12         place Open_Cook <Mode>;
13
14         place Time <Display>={<new Display()>};
15         place Not_Time <Display>;
16         place Counter <Display>;
17         place Power_D <Display>;
18         place Error <Display>;
19
20         place Light_On <Light>;
21         place Light_Off <Light>={<new Light()>};
22
23         transition open_I{
24                 action{
25                         x.open();
26                 }
27         }
28
29         transition open_S{
30                 action{
31                         x.open();
32                 }
33         }
34
35         transition open_P{
36                 action{
37                         x.open();
38                 }
39         }
40
41         transition open_C{
42                 action{
43                         x.open();
44                 }
45         }
46
```

```
47          transition close_I{
48                  action{
49                          x.close();
50                  }
51          }
52
53          transition close_S{
54                  action{
55                          x.close();
56                  }
57          }
58
59          transition close_P{
60                  action{
61                          x.close();
62                  }
63          }
64
65          transition close_C{
66                  action{
67                          x.close();
68                  }
69          }
70
71          transition clock{
72                  action{
73                          x.clock();
74                  }
75          }
76
77          transition error_clock{
78                  action{
79                          x.error_clock();
80                  }
81          }
82
83          transition valid_clock{
84                  action{
85                          x.valid_clock();
86                  }
87          }
88
89          transition power{
90                  action{
91                          x.power();
92                  }
93          }
94
95          transition digit_I{
```

```
96                  action{
97                          x.digit();
98                  }
99          }
100
101         transition digit_P{
102                 action{
103                         x.digit();
104                 }
105         }
106
107         transition digit_S{
108                 action{
109                         x.digit();
110                 }
111         }
112
113         transition stop_S{
114                 action{
115                         x.stop_SetTime();
116                 }
117         }
118
119         transition stop_P{
120                 action{
121                         x.stop_Program();
122                 }
123         }
124
125         transition stop_C{
126                 action{
127                         x.stop_Cook();
128                 }
129         }
130
131         transition stop_E{
132                 action{
133                         x.stop_Error();
134                 }
135         }
136
137         transition start{
138                 action{
139                         x.start();
140                 }
141         }
142
143         transition minute{
144                 action{
```

```
145                         x.minute();
146                 }
147         }
148
149         transition minute_C{
150                 action{
151                         x.minute();
152                 }
153         }
154
155         transition done{
156                 action{
157                         x.done();
158                 }
159         }
160
161
162         Rendering methods{
163                 void display(String){
164                         //Montre l'etat du Mode
165                 }
166         }
167 }
```

FIGURE 53: ObCS de Microwave-CO

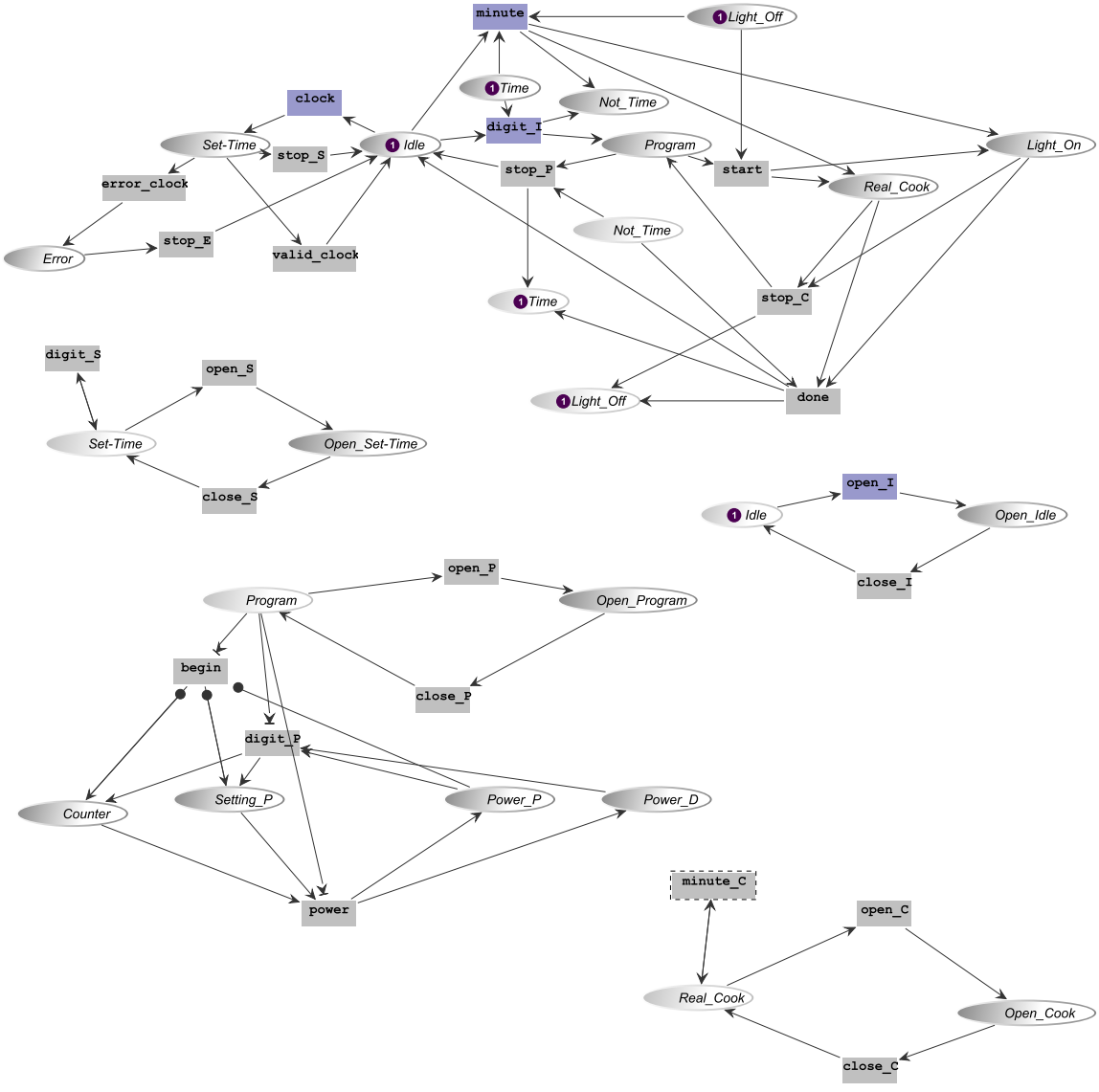


FIGURE 54: Matrice d'incidence de l'ObCS de Microwave-CO

| Places... | clock | stop_S | valid_c... | digit_I | stop_P | start | stop_C | minute | done | open_I | close_I | open_P | close_P | open_S | close_S | open_C | close_C | begin | digit_P | power | minute... | digit_S | error_c... | stop_E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Idle | -1 | 1 | 1 | -1 | 1 | 0 | 0 | -1 | 1 | -1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Set-Ti... | 1 | -1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 0 |
| Program | 0 | 0 | 0 | 1 | -1 | -1 | 1 | 0 | 0 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Real_... | 0 | 0 | 0 | 0 | 0 | 1 | -1 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Open_... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Open_... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Open_... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Open_... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Time | 0 | 0 | 0 | -1 | 1 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Not_Ti... | 0 | 0 | 0 | 1 | -1 | 0 | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Light_... | 0 | 0 | 0 | 0 | 0 | 1 | -1 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Setting... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | -1 | 0 | 0 | 0 | 0 |
| Power... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | 0 |
| Error | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | -1 | 0 | 0 | 1 | -1 |
| Counter | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | -1 | 0 | 0 | 0 | 0 |
| Power... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | 0 |
| Light_... | 0 | 0 | 0 | 0 | 0 | -1 | 1 | -1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FIGURE 55: Fonction d'activation de Microwave-CO

| **Place** | **Widget** | **Event** | **Service** |
|---|---|---|---|
| Idle | Clavier | clock | clock() |
| Idle | Clavier | digit | digit() |
| Idle | Clavier | minute | minute() |
| Idle | Clavier | open | open() |
| Set-Time | Clavier | stop | stop_SetTime() |
| Set-Time | Clavier | error | error_clock() |
| Set-Time | Clavier | digit | digit() |
| Set-Time | Clavier | open | open() |
| Program | Clavier | stop | stop_Program() |
| Program | Clavier | start | start() |
| Program | Clavier | open | open() |
| Real_Cook | Clavier | stop | stop_Cook() |
| Real_Cook | Clavier | done | done() |
| Open_Idle | Clavier | close | close() |
| Open_Program | Clavier | close | close() |
| Open_Set-Time | Clavier | close | close() |
| Open_Cook | Clavier | close | close() |
| Time | Clavier | minute | minute() |
| Time | Clavier | digit | digit() |
| Not_Time | Clavier | stop | stop_Program() |
| Light_On | Clavier | stop | stop_C() |
| Light_On | Clavier | done | done() |
| Setting_P | Clavier | power | power() |
| Power_P | Clavier | digit | digit() |
| Error | Clavier | stop | stop_Error() |
| Counter | Clavier | power | power() |
| Power_D | Clavier | digit | digit() |
| Light_Off | Clavier | minute | minute() |
| Light_Off | Clavier | start | start() |

FIGURE 56: Fonction de rendu de Microwace-CO

| **ObCS element** | | **Méthode de rendu** |
|---|---|---|
| **Nom** | **caractéristique** | |
| clock | enclenchée sur jeton < *Mode* > | display("Set-Time") |
| stop_S | enclenchée sur jeton < *Mode* > | display("Idle") |
| valid_clock | enclenchée sur jeton < *Mode* > | display("Idle") |
| digit_I | enclenchée sur jeton < *Mode* > | display("Program") |
| stop_P | enclenchée sur jeton < *Mode* > | display("Idle") |
| stop_P | enclenchée sur jeton < *Display* > | display("Time") |
| start | enclenchée sur jeton < *Mode* > | display("Cook") |
| start | enclenchée sur jeton < *Light* > | display("On") |
| stop_C | enclenchée sur jeton < *Mode* > | display("Program") |
| stop_C | enclenchée sur jeton < *Light* > | display("Off") |
| minute | enclenchée sur jeton < *Mode* > | display("Cook") |
| minute | enclenchée sur jeton < *Display* > | display("Not_Time") |
| minute | enclenchée sur jeton < *Light* > | display("On") |
| done | enclenchée sur jeton < *Mode* > | display("Idle") |
| done | enclenchée sur jeton < *Display* > | display("Time") |
| done | enclenchée sur jeton < *Light* > | display("Off") |
| open_I | enclenchée sur jeton < *Mode* > | display("Disabled") |
| close_I | enclenchée sur jeton < *Mode* > | display("Operational") |
| open_P | enclenchée sur jeton < *Mode* > | display("Disabled") |
| close_P | enclenchée sur jeton < *Mode* > | display("Operational") |
| open_S | enclenchée sur jeton < *Mode* > | display("Disabled") |
| close_S | enclenchée sur jeton < *Mode* > | display("Operational") |
| open_C | enclenchée sur jeton < *Mode* > | display("Disabled") |
| close_C | enclenchée sur jeton < *Mode* > | display("Operational") |
| begin | enclenchée | display("Setting et Counter") |
| digit_P | enclenchée sur jeton < *Mode* > | display("Setting Program") |
| digit_P | enclenchée sur jeton < *Display* > | display("Counter") |
| power | enclenchée sur jeton < *Mode* > | display("Power") |
| power | enclenchée sur jeton < *Display* > | display("Power") |
| minute_C | enclenchée sur jeton < *Light* > | display("On") |
| digit_S | enclenchée sur jeton < *Mode* > | display("Setting") |
| error_clock | enclenchée sur jeton < *Display* > | display("Error") |
| stop_E | enclenchée sur jeton < *Display* > | display("Time") |

# Bibliographie


[1] D. Harel, Statecharts : a visual formalism for complex systems, Science Computer Programming Volume 8 Issue 3, June 1987

[2] Y. Aït-Ameur, I. Aït-Sadoune, M. Baron, Modélisation et Validation formelles d'IHM : LOT 1 (LISI/ENSMA), pp. 73., 2005

[3] A.C.R.P. Pimenta, Automated Specification Based Testing of Graphical User Interfaces, PhD, Engineering Faculty of Porto University, Department of Electrical and Computer Engineering, 2006

[4] S. Miller, A. Tribble, A methodology for improving mode awareness in flight guidance design, In Digital Avionics Systems Conference, 2002

[5] A. Schyn, D. Navarre, P. Palanque, L. P. Nedel, Description formelle dune technique dinteraction multimodale dans une application de réalité virtuelle immersive. In Proceeding of the 15th French Speaking conference on human-computer interaction (IHM 2003), p. 25–28, Caen, France

[6] J. Vanderdonckt, Computer-Supported Collaborative Work, Université catholique de Louvain, 2012

[7] G. Dennis, R. Seater, Alloy Analyzer 4 Tutorial, Software Design Group, MIT

[8] S. Combéfis, D. Giannakopoulou, C. Pecheur, M. S. Feary, Learning system abstractions for human-machine interactions, in Proceedings of the 2011 International Workshop on Machine Learning Technologies in Software Engineering

[9] M. Herrmannsdoerfer, S. Konrad, B. Berenbach, Tabular notations for state machine-based specifications, Cross Talk, The Joiurnal of defense Software Engineering, March 2008

[10] D. Heitmeyer, Tool for Constructing Requirements Specifications : The SCR Toolset at the Age of Ten, International Journal of Computer Systems Science and Engineering 20.1 (2005)

[11] J. Magee, J. Kramer, Concurrency, State Models & JavaProgramming, second edition, 2006

[12] A. van Lamsweerde, Software Engineering, Université catholique de Louvain, 2011

[13] A. van Lamsweerde, Requirements Engineering From System Goals to UML Models to Software Specifications, John Wiley

[14] B. Le Charlier, Langages et Traducteurs, Université catholique de Louvain, 2011

[15] C. Pecheur, Concurrent systems : models and analysis, Université catholique de Louvain, 2010

[16] C. Pecheur, Concurrent systems : models and analysis, Université catholique de Louvain, 2010

[17] J.C. Campos, M.D. Harrison, Interaction Engineering Using the IVY Tool. In : EICS'09. Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, 2009, Pittsburgh, PA, USA : ACM

[18] M. Green, A survey of three dialogue models, ACM Transactions on Graphics (TOG), v.5 n.3, p.244-275, July 1986

[19] Y. Deville, Calculabilité, Université catholique de Louvain, 2011

[20] A. Campi, E. Martinez, P. S. Pietro, Experiences with a Formal Method for Design and Automatic Checking of User Interfaces, in Proceedings of the Position paper in IUI/CADUI 2004 Workshop on Making Model-Based UI Design Practical : usable and open methods and tools, 13th January, 2004

[21] J. Campos, M. D. Harrison, Model Checking Interactor Specifications, in Automated Software Engineering, vol. 8, 2001

[22] O. Sy, D. Navarre, Le Duc Hoa, P. Palanque, R. Bastide, The MEFISTO Project, L.I.H.S., University of Toulouse, 1999

[23] J. Berstel, S. Crespi Reghizzi, G. Roussel, P. San Pietro, A scalable formal method for design and automatic checking of user interfaces, Proceedings of the 23rd International Conference on Software Engineering, p.453-462, May 12-19, 2001, Toronto, Ontario, Canada

[24] A. Campi, E. Martinez, P. San Pietro, Experiences with Formal Method for Design and Automatic Checking of User Interfaces, Dipartimento di Elettronica E Informazione, Politecnico di Milano, 2004

[25] P. Palanque, R. Bastide, Synergistic modelling of tasks, users and systems using formal specification techniques, Interacting with Computers Volume 9 Issue 2, November 1997

[26] S. Chatty, Extending a graphical toolkit for twohanded interaction. In Proceedings of the ACM symposium on User Interface Software and Technology, UIST'94 (1994, Marina del Rey, California), ACM Press, 1994, pp. 195-204

[27] P. Palanque, D. Navarre, S. Basnyat, Usability Service Continuation through Reconfiguration of Input and Output Devices in Safety Critical Interactive Systems. The 27th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2008), 22-25 September 2008, Newcastle upon Tyne, UK