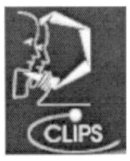
Modélisation et Validation formelles d'IHM

LOT 1
LISI/ENSMA

Auteurs : Y. Aït-Ameur, I. Aït-Sadoune et M. Baron
Adresse : LISI / ENSMA - Téléport 2 - 1, avenue Clément Ader - B.P. 40109 - 86960 Futuroscope Cedex - France
Emails : {yamine, aitsadoune, baron}@ensma.fr



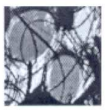


Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Introduction

L'activité de modélisation d'un système interactif a pour but de produire une description du système interactif en rapport avec les besoins de l'utilisateur qu'il s'agisse de conception, de validation, ou de spécification. Cette description adopte un point de vue extérieur au système interactif lui-même. En effet, elle s'intéresse à l'aspect, au comportement et aux effets de bord du système interactif, et non pas au fonctionnement interne du système avec lequel l'interaction est effectuée (appelé le noyau fonctionnel). 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 implantation ou à sa réalisation.

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 d'interaction se concentrent sur les relations entre l'utilisateur et le système interactif. En effet, de manière générale, les modèles de cycle de vie du génie logiciel classique ont principalement pour objectifs : la *faisabilité* associée à la difficulté et au travail nécessaire pour produire un logiciel, et la *qualité* du logiciel qui est souvent liée à la correction, la consistance, la sûreté, la maintenance, etc. En plus des deux objectifs précédents, les modèles de développement des systèmes interactifs doivent assurer également *l'utilisabilité*, c'est-à-dire la facilité d'utilisation d'un logiciel donné par un utilisateur donné ou une catégorie d'utilisateurs [HH93], ou de manière plus générale son *acceptabilité* par les utilisateurs [Nie93].

1.2 Besoins dans le domaine des IHM

La nécessité de réaliser l'interface d'une application avec la même rigueur que celle accordée à l'application elle-même est primordiale dans les systèmes critiques en particulier, pour des raisons de sécurité, et dans les applications informatiques en général pour des raisons de coût et d'utilisabilité.

L'interface doit [Cou90][Shn98] :

- refléter exactement et fidèlement le comportement de l'application. C'est l'aspect informatique,
- faciliter son utilisation. C'est l'aspect ergonomique,
- être accessible au plus grand nombre d'utilisateurs ayant des comportements différents. C'est l'aspect psychologique.

Ainsi, la conception d'une interface fait intervenir non seulement des informaticiens pour la conception et le développement mais aussi des ergonomes pour les choix d'ergonomie et de disposition ou de spécifications externes (modèles de tâches), et des psychologues pour capturer les comportements des utilisateurs (expérimentation).

1.3 Modèles et notations dans les IHM

Avant d'aborder, dans les chapitres suivants, les différentes approches de formalisation de développements et de propriétés de programmes dans les IHM, rappelons que ce domaine de l'informatique a vu, lui aussi, naître de nombreux modèles et notations semi-formels où le graphisme est un élément important. Ces modèles et notations, utilisés aussi bien pour la conception que pour la validation et la vérification, ont été conçus pour être utilisés, en particulier par des non-informaticiens issus des communautés citées précédemment (informaticiens, ergonomes, psychologues etc.). Ils ont suivi les différentes évolutions matérielles et logicielles de ces dernières années. Nous les passons brièvement en revue ci-dessous en utilisant une description qui suit un cycle de développement descendant.

Nous séparons les notations de descriptions (description de tâches, description de l'interaction), souvent issues des non-informaticiens, des notations en conception et en architecture propres aux développements informatiques.

1.3.1 Notations pour la description d'IHM

Afin d'exprimer les besoins des utilisateurs, souvent des non-informaticiens, de nombreuses notations de descriptions d'IHM ont été proposées. Elles sont, pour la plupart, centrées utilisateurs et sont donc loin des implantations informatiques.

MAD [SPG90], pour Méthode Analytique de Description, utilise une notation arborescente qui décrit les différentes tâches répertoriées par l'utilisateur. Chaque niveau dans l'arbre correspond à un niveau de décomposition de tâches. Une représentation par des expressions régulières peut être extraite de cette notation. HTA (Hierarchical Task Analysis) [DFAB93], est une autre notation, similaire à MAD, qui utilise une décomposition des tâches en fonction de leur but, puis des sous-buts etc. Elle est fondée sur la description des objectifs des tâches. D'autre part, une notation telle que UAN [RH93] et son extension XUAN [GEM94] décrit non seulement le comportement de l'interface mais aussi celui de l'utilisateur. Elle utilise une notation à base de triplets comportant l'action de l'utilisateur, la réaction de l'interface et l'état de l'interface. Dans ce cas, la description d'une interface revient à décrire une table regroupant les triplets.

La notation ConcurTaskTrees (CTT[1] [Pat01]) met l'accent sur les activités de l'utilisateur. Elle propose de nombreux opérateurs temporels permettant de décrire différents comportements associés au système interactif (interruption, concurrence, désactivation, ...).

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, ou bien de la validation d'un système interactif déjà conçu et/ou existant.

La figure 1.1, propose l'exemple d'un *Distributeur automatique de billets* inspiré de l'exemple fourni dans l'environnement CTTE[PMG01](ConcurTaskTrees-Environment). Seuls les opérateurs d'activation ($>>$), de choix ($[]$), de désactivation ($[>$) et d'itération ($*$) sont présentés. Nous modélisons le fait que l'utilisateur doit tout d'abord insérer sa carte puis saisir son code PIN avant l'accès à son compte. Cette tâche d'autorisation est suivie par l'accès au distributeur où seule la tâche *Retirer Argent* est représentée. Cette tâche est désactivable à tout moment par la tâche *Terminer Accès*.

Les ergonomes et les psychologues contribuent à la définition de ce type de notations. Les approches centrées utilisateurs de l'IHM fondées sur la description de tâches utilisateurs sont privilégiées.

[1] Cette notation est utilisée dans le cadre du projet VERBATIM. Une description détaillée est présentée dans les chapitres suivants.

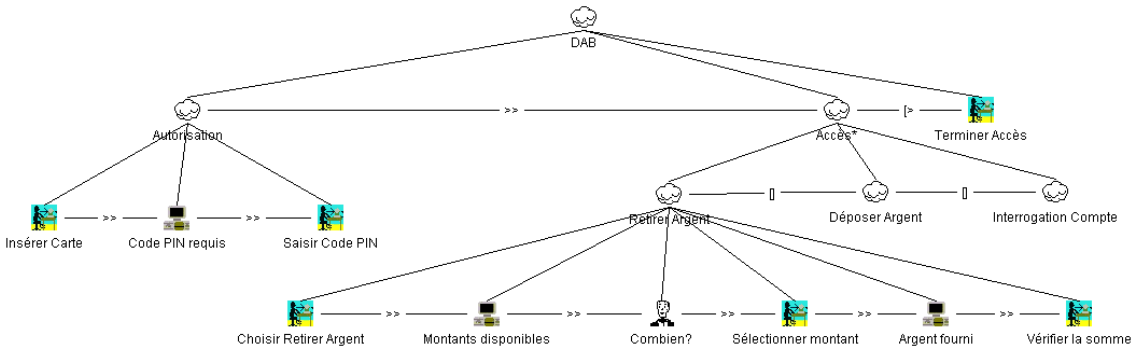


FIG. 1.1 – *Distributeur automatique de billets*, un exemple en notation CTT.

Remarque

Ce type d'approche est en opposition avec le développement de logiciel qui cherche à factoriser et à abstraire les fonctions communes aux différentes tâches. Cela constitue une difficulté à prendre en compte lors du passage de l'analyse de tâches au développement.

1.3.2 Modèles d'architecture et de conception

Tous les modèles définis pour les IHM séparent la conception de l'IHM de la conception de l'application ou noyau fonctionnel. Ils utilisent un mécanisme d'association entre les deux parties. Nous donnons une description sommaire de trois d'entre eux, les plus utilisés. D'autres modèles comme MVC ont été proposés.

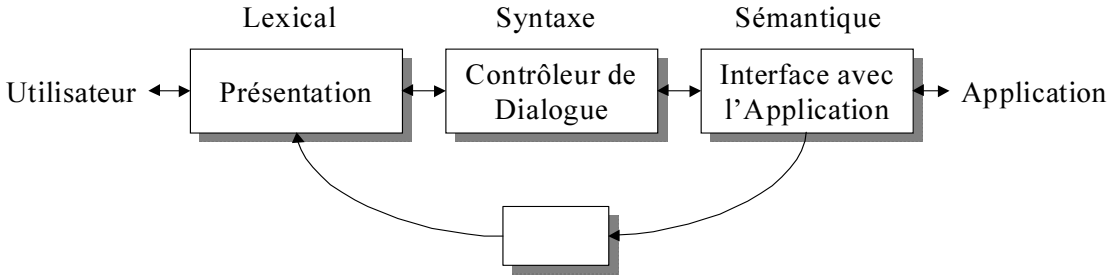


FIG. 1.2 – Le modèle SEEHEIM.

Le modèle de Seeheim [Pfa85] est un modèle global qui comprend trois modules : la présentation, le contrôle du dialogue et l'interface avec l'application (voir figure 1.2).

Le modèle PAC [Cou87] (Présentation, Abstraction et Contrôle), qui inclut le noyau fonctionnel dans l'abstraction, est en quelque sorte une décomposition du modèle de Seeheim. Il décompose l'application en plusieurs agents eux-mêmes séparés en présentation d'un objet de l'interface, application (Abstraction) et contrôle du dialogue[2]. Ce contrôle permet d'établir la liaison entre application et présentation au sens de Seeheim.

Enfin, le modèle ARCH [BPR+91][BFL+92] intègre l'ensemble de l'application depuis le noyau fonctionnel jusqu'à la boîte à outils utilisée pour la présentation en passant par le contrôle de l'application (voir figure 1.3).

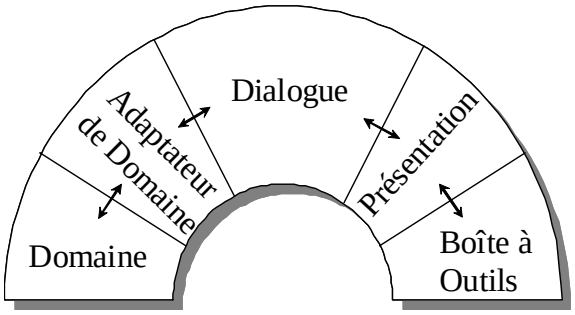


FIG. 1.3 – Le modèle ARCH.

Les modèles précédents 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. D'autres modèles intégrant plusieurs points de vue, qualifiés de modèles hybrides, ont été développés [Gui95a][Fek96].

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 donc être apportées à moindre coût.

Par contre les notations utilisées pour décrire la conception sont à la fois informelles et indépendantes des notations de description.

1.3.3 Les systèmes de développement basés sur des modèles

L'insuffisance de la formalisation des deux types de notations précédentes fait qu'elles ne font pas l'objet d'implantation par des techniques. C'est pourquoi, les recherches se sont orientées vers la définition de techniques plus rigoureuses.

[2] Cette abstraction sera reliée avec une opération d'abstraction pratiquée dans les modèles B développés dans le cadre du projet VERBATIM qui a pour effet de simplifier les obligations de preuve générées ainsi que leur nombre.

Les systèmes de développement basés sur des modèles ou MBS (*Model-Based Systems*) s'appuient sur la définition d'un modèle particulier de l'interface, complètement formalisé (à base de règles ou de systèmes de transitions). L'interface est décrite dans un langage de spécification particulier dont la sémantique est donnée par ce modèle. Il est possible de générer du code implantant la présentation de l'interface à partir de ce modèle. C'est le cas de systèmes tels que FUSE [LS96], MASTERMIND [SSC+95], UIM/x [Fou91] ou H^4 [Gui95a].

Il est possible, moyennant une formalisation dans le modèle du MBS utilisé, de représenter des descriptions en se basant sur les modèles ou notations. Les règles de traduction et de représentation restent empiriques.

Cependant, il faut noter que ce type d'approche ne permet pas de développement incrémental ni l'expression et la vérification de propriétés. Toutes les parties de l'interface doivent être décrites et connues pour que la génération de code, fondée sur ce modèle, puisse être réalisée. Les modifications deviennent très coûteuses et correspondent souvent à un nouveau développement. De plus, pour certains modèles, le lien avec le noyau fonctionnel reste à établir.

1.3.4 Propriétés dans les IHM

Les propriétés dans les IHM sont les propriétés couramment définies pour les systèmes interactifs en général telles que les propriétés de sûreté et d'équité (d'honnêteté) auxquelles sont ajoutées les propriétés particulières aux IHM, c'est-à-dire les propriétés inhérentes au domaine des IHM comme celles liées à la représentation ou les propriétés issues des exigences des ergonomes ou des psychologues.

La grande majorité des travaux s'est intéressée au problème de l'expression et de la vérification de propriétés. [JBAA01] [DFAB93] [HT90] [DH95a] [Roc98a] décrivent une classification des propriétés. Nous les résumons dans ce qui suit.

Deux grandes classes de propriétés ont été définies : les propriétés de validité et les propriétés de robustesse.

- *Propriétés de validité :* ce sont les propriétés qui caractérisent un fonctionnement attendu ou voulu par un utilisateur. Dans cette catégorie de propriétés, on distingue les propriétés de complétude (réalisation d'un objectif donné), de *flexibilité* ou de *souplesse* pour la représentation de l'information et pour le déroulement des tâches (utilisateurs multiples, non-préemption, atteignabilité des états), etc.
- *Propriétés de robustesse :* ce sont les propriétés relatives à la sûreté de fonctionnement du système. Parmi ces propriétés, on trouve les propriétés liées à la visualisation du système telles que l'observabilité, l'insistance et l'équité (l'honnêteté).

Cet ensemble de propriétés permet de qualifier la facilité d'utilisation d'une interface. Enfin, notons que la définition et la description de propriétés liées à l'ergonomie et à la psychologie ne sont pas encore maîtrisées.

1.4 Conclusion

Ce chapitre a dressé un bref panorama des notations et modèles définis et mis en jeu pour la conception de systèmes interactifs. En plus des préoccupations traditionnelles présentes en génie logiciel, les notations et modèles du domaine de l'IHM s'intéressent à d'autres aspects, propres aux domaines, telles que l'ergonomie, la facilité d'utilisation, etc. Tous ces points définissent l'utilisabilité d'une IHM.

Ainsi, en plus de l'informatique, l'utilisabilité regroupe différentes caractéristiques issues de l'ergonomie et de la psychologie. La présence de ces disciplines fait que les systèmes interactifs couvrent plusieurs domaines et disciplines. Les différentes notations pour la conception, l'architecture, la validation et la vérification tentent de prendre en compte ces différents aspects. Leurs définitions ont engendré une hétérogénéité sémantique souvent due à des ambiguïtés d'interprétation du fait de l'absence de formalisation. En effet, le plus souvent ces notations sont semi-formelles associées à des représentations graphiques.

Du point de vue du génie logiciel, les systèmes interactifs soulèvent deux points importants :

- d'une part, la majorité des systèmes interactifs, voire tous, séparent le noyau fonctionnel de l'application de l'IHM. Cette séparation permet de structurer le logiciel en deux parties principales. 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[3] que les programmeurs utilisent massivement pour la conception d'une IHM. Ces boîtes à outils existent sous forme de programme, elles ne sont pas formellement spécifiées, ou tout au moins seules les API sont utilisées comme spécification.

[3] toolkit en anglais

Chapitre 2

Approches formelles en IHM

Les différents modèles présentés dans le chapitre précédents sont décrits de façon semi-formelles. Il n'est pas possible d'en dériver du code, ni d'effectuer des vérifications et/ou des preuves. Ils servent plus de recommandations aux développeurs de codes que de modèles supports de vérification. Afin de permettre des vérifications, l'utilisation de techniques formelles pour les développements d'IHM a été préconisée. Ces techniques ont été appliquées aussi bien pour les modèles d'architectures que pour les modèles de tâches. Notons que l'application des techniques formelles dans les IHM a suivi l'évolution historique de ces techniques et de leur utilisation dans le génie logiciel [Gau95].

2.1 État de l'art

Les premiers modèles ont utilisé des automates et/ou des extensions d'automates tels que les *statecharts* [Har87] ou les ATN [Was81]. [Jac82] fut le premier à utiliser les automates pour la spécification du comportement des IHM à l'aide de systèmes de transitions. Des modèles tels que H^4 [Gui95a] et [Was81] ont utilisé les ATN afin d'éviter le problème de l'explosion du nombre d'états grâce à la décomposition et à la hiérarchisation. D'autres modèles se sont fondés sur des réseaux de Petri [Pal92] comme moyen de représentation du système interactif. Plus récemment, des extensions des automates (BDD) ont été utilisées comme modèle pour des logiques temporelles telles que CTL, CTL* [CES86] ou XTL [Bru97]. Le système SMV [Mil92] a été mis en œuvre par [AWM95] pour vérifier, par *model-checking*, des propriétés des IHM exprimées dans la logique CTL. Le langage synchrone LUSTRE a permis de vérifier et de générer du code dans les IHM à partir de descriptions écrites en UiL/X [Roc98a].

En parallèle, d'autres techniques orientées modèle, basées sur la vérification d'obligations de preuve et sur la description incrémentale, ont été expérimentées sur les IHM comme Z [DH93a, DH95a], VDM [Mar86], B [Abr96b, Abr96a].

La spécification et la vérification d'IHM ont vu aussi l'utilisation des techniques algébriques. [PF92b] furent les premiers à introduire la notion d'interacteur en utilisant LOTOS [Sys84a] (issu de la fusion de ACT ONE pour la partie données et de CCS pour la partie interaction).

Enfin, nous ne saurions être exhaustif si nous ne citions pas les travaux basés sur les systèmes à base de logiques d'ordre supérieur. En effet, le système HOL [GP94] a été utilisé par [BACL95] pour la description d'IHM et pour la vérification de propriétés. Le prouveur de théorèmes de HOL est utilisé dans ce but.

Toutes les techniques précédemment décrites permettent l'expression, à un haut niveau d'abstraction, de descriptions, de comportements et de fonctions des interfaces ainsi que la possibilité de les valider. La vérification de propriétés est aussi autorisée dans ce type de système soit automatiquement (par *vérification sur modèles* ou *model-checking*) soit semi-automatiquement (en utilisant un *prouveur de théoèmes* ou *theorem prover* qui permet de décharger les obligations de preuve).

Intéressons nous maintenant à détailler quelques-unes de ces approches les plus significatives afin, d'une part, de les décrire et d'autre part de déterminer leurs limites.

2.2 Les interacteurs (CNUCE, York et CERT)

Une des notions importantes dans la description formelle de systèmes interactifs est la notion d'interacteur. Un interacteur est un composant logiciel fournissant des services particuliers dans une IHM. Il est doté d'un état, il reçoit des événements en entrée et en renvoie en sortie. Cette notion a été introduite afin de pouvoir décrire et/ou concevoir un système interactif par compositions d'interacteurs. Elle permet ainsi le raisonnement compositionnel.

Différentes définitions des interacteurs ont été proposées suivant la nature de l'approche employée [Mar97]. La notion d'interacteur repose sur un principe commun : la description d'un système interactif par composition de processus abstraits indépendants. Ceux-ci ressemblent fortement aux modèles d'architecture multi-agents, par exemple PAC, dans le sens où l'interface est répartie en une multitude d'agents actifs, réagissant à la réception de certains événements. Toutefois, à la différence d'une architecture multi-agents, les approches à base d'interacteurs explicitent formellement la communication externe et le comportement interne de ces interacteurs.

Au fil des approches, les interacteurs sont devenus de plus en plus expressifs pour aboutir à une modélisation plus approfondie du système interactif.

2.2.1 Interacteurs de CNUCE [PF92a, FP90]

Les interacteurs de CNUCE [PF92a, FP90] ont été développés dans le cadre de la formalisation du projet GKS [GKS85]. Ce projet avait pour but de concevoir des composants de base (les interacteurs) avec lesquels un système graphique interactif (à la manière d'une bibliothèque) peut être modélisé, construit et vérifié.

Un interacteur est un composant de l'interface utilisateur. Son comportement est réactif et peut fonctionner en parallèle avec d'autres interacteurs.

Au niveau le plus abstrait, l'interacteur est vu comme une *boîte noire* qui sert d'intermédiaire entre un côté *utilisateur* et un côté *application*. Il peut émettre, recevoir des événements des deux côtés de cet interacteur, et traiter en interne les données qui transitent. A un niveau moins abstrait, l'interacteur est vu comme une *boîte blanche* qui permet alors de comprendre son fonctionnement interne.

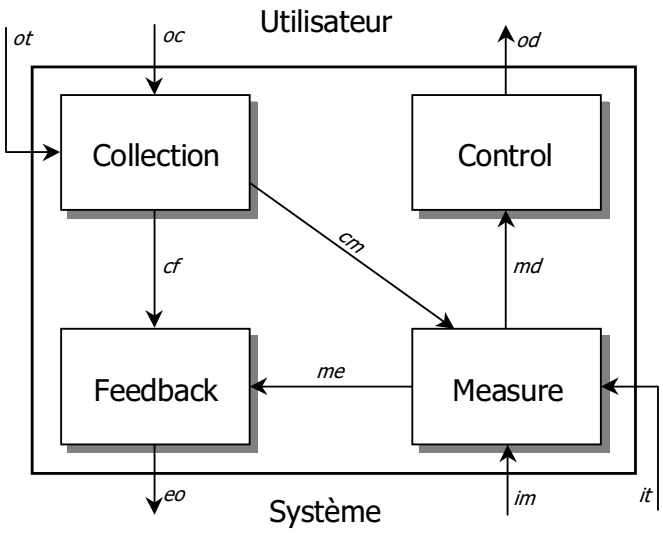


Fig. 2.1 – *Boîte blanche* de l'interacteur de CNUCE, adapté de [PF92a].

La figure 2.1 présente une vue *boîte blanche* d'un interacteur de CNUCE. Suivant cette vue, un interacteur est composé de quatre fonctionnalités qui définissent un intermédiaire entre le côté *utilisateur* et l'*application* :

– la fonction **Collection** maintient une représentation abstraite de l'apparence externe de l'interacteur. Quand la fonction est déclenchée par un événement

ot (**o**utput **t**rigger) de la part d'un autre interacteur, elle transmet les données présentes sur le port *oc* (**o**utput **c**ollection) à la fonction *Feedback*. De même, elle transmet les données présentes sur le port *cm* (**c**ollection **m**easure) à la fonction *Measure* pour lui permettre de transformer la donnée d'entrée *im* (**i**nput **m**easure) et de produire la donnée *md* (**m**easure **d**ata) ;

- la fonction **Measure** reçoit des données issues de l'utilisateur, à l'arrivée d'un événement déclencheur *it* (**i**nput **t**rigger). Elle dirige ces données à *Feedback* et à *Control* ;
- la fonction **Feedback** transmet un echo de l'interacteur à l'utilisateur. Elle reçoit les données *cf* (**c**ollection **f**eedback) et *me* (**m**easure **e**vent) et les transforme en données de sortie *eo* (**e**vent **o**uput) ;
- la fonction **Control** délivre la données *od* (**o**uput **d**ate) aux autres interacteurs.

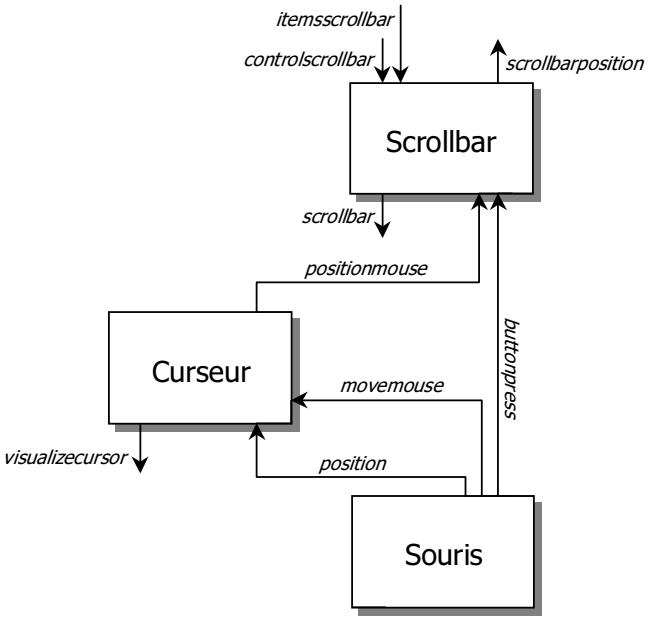


FIG. 2.2 – *Modélisation d'un scrollbar*, un exemple avec les interacteurs en boîte noire de CNUCE, adapté de [PF92a].

La description de la totalité d'une interface utilisateur consiste à composer plusieurs interacteurs. Plus précisément, les sorties des fonctions *Feedback* ou *Control* des uns sont connectées avec les entrées des fonctions *Measure* ou *Collection* des autres (la sortie *Control* avec l'entrée *Measure* et la sortie *Feedback* avec l'entrée *Collection*). De plus, le fonctionnement parallèle des interacteurs composés est ex-

primé par l'intermédiaire d'opérations de composition parallèle des processus.

Nous proposons sur la figure 2.2 l'exemple de la modélisation d'un *scrollbar* suivant une vue *boîte noire*, adapté de [PF92a]. Trois interacteurs sont définis : la souris, le curseur et le scrollbar. L'interacteur souris envoie des flots au curseur pour lui indiquer les nouvelles positions (sorties *movemouse* et *position*). Le curseur quant à lui gère l'affichage du curseur de la souris (sortie *viualizecursor*). Quand la souris transmet un ordre via la sortie *buttonpress* au scrollbar, celui-ci récupère la position du curseur via l'entrée *positionmouse*. L'interacteur scrollbar affiche sa nouvelle position via la sortie *scrollbar*. Les entrées *itemsscrollbar*, *controlscrollbar* et la sortie *scrollbarposition* sont connectées à différents interacteurs (non représentés sur la figure).

L'intérêt de l'approche des interacteurs de CNUCE se situe au niveau de sa modélisation formelle. Plusieurs techniques issues de l'algèbre de processus communiquant LOTOS ont été utilisées pour modéliser les interacteurs : les modèles algébriques Act-One pour décrire la partie données et l'algèbre de processus CCS pour décrire la composition d'interacteurs. Des travaux complémentaires sur la composition et la vérification formelles ont été apportés aux interacteurs de CNUCE par [Mar97].

Cependant, même si cette approche permet l'expression et la vérification de certaines propriétés, plusieurs points sont à considérer :

- **Absence d'état observable** : comme l'a constaté [Gui95b] les interacteurs de CNUCE ne contiennent pas de structure de contrôle du système autre que l'émission et la réception d'événement. La notion d'état est absente et ne permet pas d'exprimer des propriétés relatives à la connaissance de l'état du dialogue comme par exemple les propriétés d'adaptivité ou d'atteignabilité. Par conséquent, les interacteurs de CNUCE sont plutôt adaptés à la modélisation de la couche présentation et ne permettent qu'une validation partielle du système interactif ;
- **Techniques formelles hétérogènes** : les techniques formelles utilisées sont hétérogènes (Act-One et CCS) ; il existe par conséquent deux sémantiques et deux systèmes de preuve différents ;
- **Complexité de la modélisation de l'interface graphique** : la vérification des propriétés est réalisée par un outil de preuve fondé sur la technique de *model-checking*. La validation et la vérification se heurtent au problème d'explosion combinatoire, et cela bien que des techniques de modularisation existent [Mar97].

2.2.2 Interacteurs de York [DH95a, DH93a]

L'idée de modéliser un système interactif au moyen d'interacteurs a été reprise par les travaux réalisés à l'Université de York [DH95a, DH93a]. A la différence de ceux de CNUCE, les interacteurs de York permettent de modéliser plus finement le dialogue du système interactif dans le sens où l'état de l'interacteur est explicitement décrit. Deux évolutions ont été proposées : les objets abstraits d'interaction (« Abstract Interaction Object ») [DH93b] basés sur un modèle à états et une deuxième approche basée sur un modèle événementiel [DH95b].

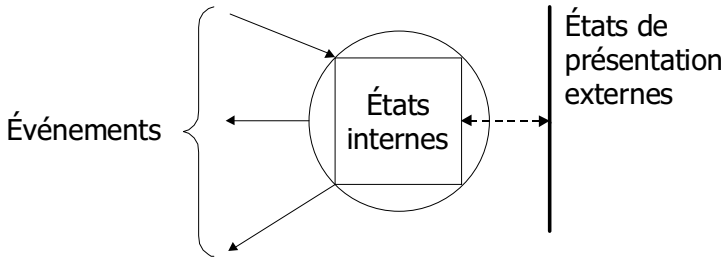


FIG. 2.3 – Schéma d'un interacteur de York.

Dans la version initiale [DH93b], l'interacteur de York est décrit sous la forme d'un système à états de type automate, définissant l'état interne de l'interacteur (voir figure 2.3). Lorsque cet interacteur reçoit des événements ou des actions d'entrée (commandes qui proviennent de l'utilisateur ou de l'application), il modifie son état *interne* et associe à cette modification un changement d'état de présentation *externe*. Il peut aussi transmettre des événements de sortie, encore appelés *actions de sortie*.

Un ensemble d'opérateurs de composition ont été définis. Ils correspondent aux opérateurs de l'algèbre des processus communiquants (synchronisation, renommage par exemple). Ils permettent donc d'effectuer un produit synchronisé à partir des automates issus des interacteurs. Le modèle obtenu décrit alors un réseau de processus communicants.

Dans la version étendue des interacteurs de York, [DH95b] proposent d'approfondir la notion d'événement qui restait abstraite dans la version initiale. Dans ce sens, les auteurs représentent les comportements des interacteurs par des *ensembles partiellement ordonnés d'événements*. Ce formalisme consiste à définir une relation d'ordre partiel sur un ensemble d'événements observés pendant une exécution d'un système. Ces événements sont des occurrences d'actions d'entrée (enfoncer le bouton de la souris) et de sortie (bouton grisé). La relation d'ordre partiel spécifie la dépendance causale entre événements en décrivant l'ordre temporel observé sur ces

événements à l'exécution [Ses02]. Ainsi les ensembles partiellement ordonnés peuvent être assimilés à des traces. Ils peuvent être comparés à cette relation et servir à vérifier des propriétés. Les événements en parallèle ne sont pas reliés par cette relation d'ordre partiel.

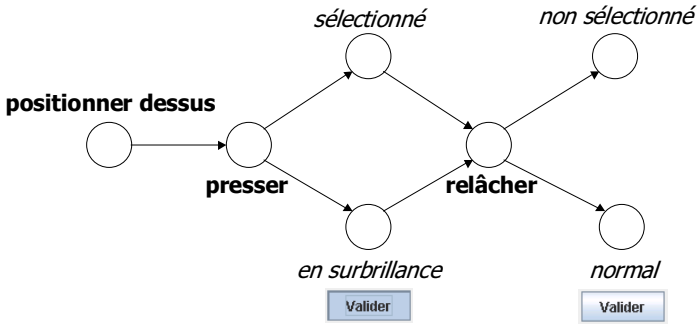


Fig. 2.4 – Comportement d'un bouton.

Nous donnons sur la figure 2.4 la représentation d'une trace associée à l'interacteur Bouton. Elle décrit l'ordre temporel entre les actions d'entrée, par exemple l'occurrence d'action d'entrée *positionner dessus* (le curseur de la souris est positionné sur le bouton) suivie de l'occurrence action d'entrée *presser* (enfoncement du bouton de la souris). Cet ordre temporel implique les occurrences des actions de sortie correspondant aux états internes *sélectionné* et externes *en surbrillance*. À partir d'une description graphique de ce type, la compréhension de la dynamique du comportement d'un interacteur devient lisible.

La version étendue des interacteurs de York permet d'exprimer de nombreuses propriétés comme la propriété d'observabilité. Il suffit d'exprimer qu'à toute trace d'événements correspond un état interne et un état externe. Mais à la différence des interacteurs de CNUCE, ceux de York ont permis de prendre en compte le point de vue de l'utilisateur dans la modélisation du système. A partir d'une trace et d'un état initial de cette trace, il est possible d'atteindre un état désiré afin de vérifier entre autre la propriété d'atteignabilité.

La particularité des interacteurs de York est la définition d'une description formelle dans la technique formelle Z [Spi88]. Cette description constitue une spécification des ensembles partiellement ordonnés d'événements dans la logique du premier ordre. A l'opposé des interacteurs de CNUCE qui emploient une technique fondée sur la vérification sur modèles (*model-checking*), l'approche de York repose sur la technique de preuve par déduction et sur l'utilisation de raffinements successifs des spécifications.

Toutefois, différents points de l'approche des interacteurs de York sont à considérer :

- elle est restée au stade de l'écriture de spécifications et de la vérification de propriétés. La génération ou le contrôle de la génération de code n'y est pas abordé ;
- cette approche intègre succinctement le point de vue de l'utilisateur dans la spécification des interfaces, elle ne permet donc pas de vérifier la totalité des propriétés de validité et de complétude ;
- un dernier point se situe au niveau de l'utilisation du langage Z. Ce langage s'avère difficile d'utilisation et les outils disponibles ne supportent pas toutes les étapes.

2.2.3 Interacteurs du CERT [Roc98b]

L'approche proposée par le CERT [Roc98b] s'appuie sur le concept d'interacteur de York codé avec Lustre [HCRP91]. Cette approche permet la description de systèmes interactifs à l'aide de langages à base de flots de données et permet de vérifier les propriétés exprimées dans ce type de langage.

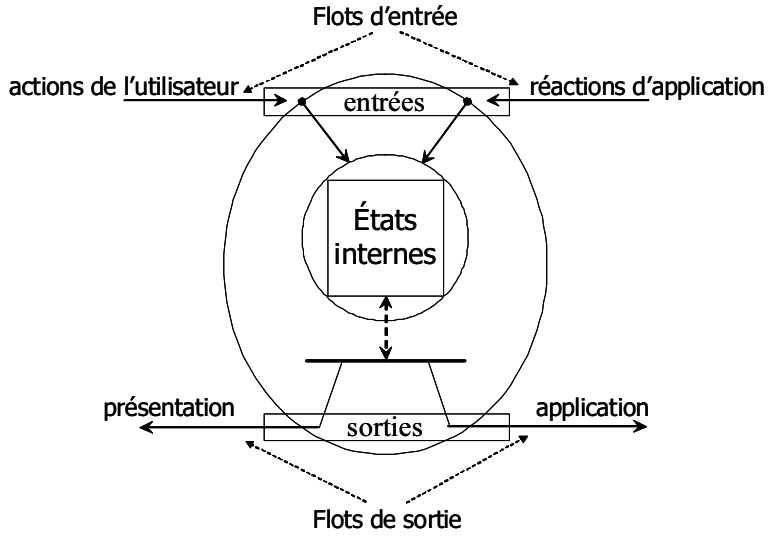


FIG. 2.5 – Interacteur à *flots de données*.

Lustre est le langage à *flots de données synchrone* utilisé. Un système décrit en Lustre est représenté comme un réseau d'opérateurs (ou nœuds) agissant en parallèle. Les opérateurs transforment des flots d'entrée en flots de sortie à chaque

top d'horloge. La communication entre processus est obtenue par composition, c'est-à-dire la connexion des sorties d'opérateurs aux entrées d'autres opérateurs. Les séquences de valeurs en entrée et en sortie des opérateurs sont des *flots de données* représentant des événements.

Les auteurs ont tout d'abord proposé une modélisation d'un interacteur de York dans le langage Lustre conforme à la figure 2.5. Les flots d'entrée qui modifient le comportement de l'interacteur sont issus des actions générées par l'utilisateur, de l'application ou d'un autre interacteur. Les opérateurs contenus dans l'interacteur à flots de données transforment ces flots d'entrée en flots de sortie. Ces derniers sont envoyés à la présentation ou à un autre interacteur. C'est dans ce sens que le comportement d'un interacteur en Lustre s'exprime sous la forme de dépendances entre flots d'entrée et flots de sortie. L'approche par interacteur de Lustre permet également la description de systèmes plus complexes au moyen de la composition d'interacteurs. Cette composition est codée naturellement en Lustre par la composition de nœuds.

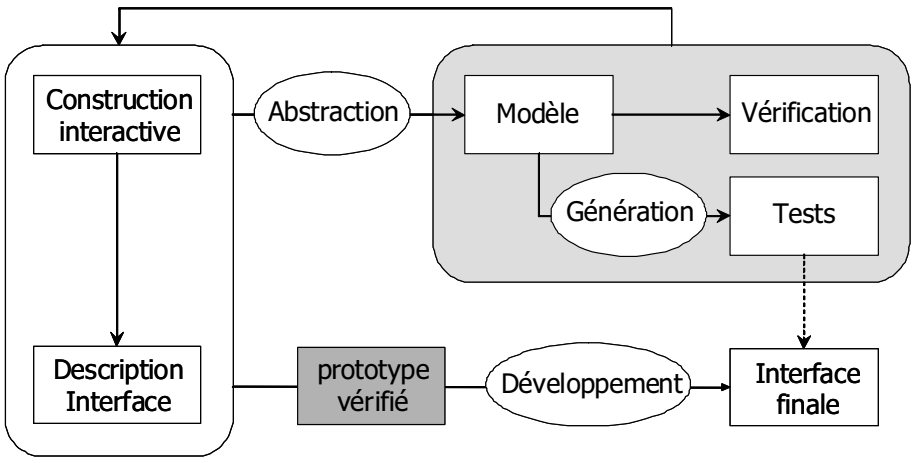


Fig. 2.6 – Approche à la fois expérimentale et formelle de [Roc98b].

[Roc98b] propose d'utiliser la modélisation d'un interacteur à base du langage Lustre dans une approche à la fois expérimentale et formelle, illustrée sur la figure 2.6. A l'opposé des approches de CNUCE et de York, l'approche du CERT couvre l'intégralité du cycle de développement d'un système interactif : spécification, vérification, validation, conception et implémentation. Elle se base sur l'utilisation du générateur de présentation Uim/x [Hal91] pour le développement de la partie graphique et d'une production à partir de ces outils d'une description de l'interface dans le langage Uil [Hal91]. A partir de cette description un modèle formel Lustre basé sur la composition d'interacteurs est extrait automatiquement. Le modèle formel

permet alors une vérification des propriétés et une génération de jeux de tests.

Cette approche a l'avantage d'être facile d'emploi parce que toute la modélisation formelle est gérée automatiquement et reste transparente pour le concepteur.

On notera cependant que :

- les propriétés concernant l'interaction sont vérifiées par l'outil Lesar, qui implante des techniques de vérification sur modèles. Malgré des techniques d'optimisation, cet outil ne permet pas d'éviter le problème d'explosion combinatoire ;
- la modélisation de la partie applicative du système interactif est absente. Il s'agit donc d'une approche qui modélise principalement la couche graphique de l'IHM.

Enfin nous ne saurions être exhaustifs si nous ne citions pas les travaux de [Ses02]. Ces travaux reprennent l'approche Lustre pour représenter les modèles d'interacteurs de York et du CERT pour répondre à deux besoins. Le premier concerne la compréhension des modèles Lustre par les utilisateurs et la seconde s'intéresse au problème de l'explosion combinatoire inhérente au model checking.

2.3 ICO : Interactive Cooperative Object [PBS95]

Des travaux utilisant des techniques formelles basées sur les réseaux de Petri afin de vérifier des propriétés sur les systèmes interactifs ont été également réalisés. [BP90] se sont tout d'abord intéressés aux techniques de spécification dans le but de décrire le dialogue d'une application interactive. Basés sur l'architecture ARCH (décrite en section 1.3.2), ces travaux ont permis de mettre en évidence l'utilité d'employer une technique de spécification formelle exécutable. Toutefois, la description des liens avec les différents modules de l'architecture n'était pas précisée. Par conséquent, cette approche ne permettait pas de vérifier un ensemble exhaustif de propriétés du système interactif.

C'est dans ce sens que [PBS95] ont présenté un nouveau formalisme, les Objets Coopératifs Interactifs ou ICO (Interactive Cooperative Object) qui est une extension au formalisme des Objets Coopératifs ou OC proposé par [Bas92]. Ce formalisme couvre une grande partie du modèle d'architecture ARCH où les Objets Coopératifs permettent la description de l'adaptateur du noyau fonctionnel et d'une partie de ce noyau fonctionnel. Le dialogue est quant à lui décrit par l'augmentation des objets coopératifs par la gestion d'événements (événements utilisateur et événements du noyau fonctionnel). Les modules présentation et boîte à outils sont détaillés au moyen d'un ensemble d'objets décrivant l'interaction. Une fonction d'ac-

tivation définit le lien entre l'ensemble des objets de la présentation et le réseau de Petri du dialogue.

Dernièrement une extension des ICO a été proposée par [Nav01] pour résoudre principalement le problème de modélisation de systèmes interactifs complexes. Cette extension apporte une structuration des modèles ICO à la manière des approches interacteurs mais en se basant sur l'architecture multi-agents MVC. L'auteur s'intéresse aussi à la validation croisée de modèles de tâches CTT et des modèles du système.

Les propriétés classiques issues des réseaux de Petri et de leurs graphes de marquage peuvent être vérifiées par ce type d'approche. Les propriétés, comme l'atteignabilité d'un état par exemple, des systèmes interactifs peuvent être exprimées et vérifiées.

On notera cependant que :

- même si cette approche montre la volonté de décrire un système interactif complet, elle ne permet cependant pas de fournir une sémantique formelle homogène pour les langages de modélisation. En effet, la modélisation de la partie présentation n'est pas clairement définie et l'utilisation du modèle de tâches CTT sous une forme semi-formelle n'autorise pas la validation formelle des tâches ;
- la technique de vérification utilisée se base sur une approche de *model-checking* qui a l'avantage de fournir une spécification exécutable mais en contrepartie se heurte au problème de l'explosion combinatoire ;
- à la différence des interacteurs, l'approche des ICO ne définit pas formellement les opérateurs de composition et de décomposition. Elle engendre donc des réseaux de Petri complexes et difficiles à lire. L'absence d'opérateurs de décomposition/composition clairement définis ne permet pas de construire le système de façon incrémentale.

2.4 Modélisation modulaire du LISI

Dans un premier temps [AAGJ98a] et [AAGJ98c] se sont intéressés à la formalisation des spécifications d'un système interactif opérationnel tout en assurant certaines propriétés ergonomiques. Cette approche se fonde sur une technique orientée modèle : la méthode B [Abr96a]. Cette dernière a été appliquée aux systèmes interactifs de type WIMP (Windows, Icons, Menus et Pointers). À partir d'études de cas, comme le Post-It Notes®[1] le *range-slider* et le convertisseur francs/euros,

[1] « Post-It Notes® » est une appellation déposée par 3M

ces travaux ont permis de formaliser de nombreux types d'interaction (par exemple la manipulation directe via le glisser/déplacer) et d'exprimer et de vérifier des propriétés comme l'observabilité, l'insistance ou la robustesse. De plus, les travaux de [AA00] ont permis de représenter et vérifier avec B toutes les étapes de raffinement pour aboutir à une représentation dans un langage de programmation.

Plusieurs avantages ont été mis en avant :

- l'intégration des notations et techniques déjà définies et souvent utilisées par les concepteurs d'IHM ;
- l'utilisation de la méthode formelle B comme technique pivot et de l'outil associé AtelierB [Cle97]. Ce dernier génère automatiquement toutes les obligations de preuves et est doté d'un prouveur automatique et d'un prouveur interactif. Ces obligations de preuve sont déchargées par le prouveur ;
- l'utilisation de la technique B a permis de représenter la description, la spécification formelle et la conception de chaque module suivant l'architecture ARCH utilisée afin de structurer le système interactif.

En comparaison avec les approches formelles de la famille des vérificateurs sur modèles (*model-checkers*), cette contribution qualifiée de descendante ne souffre pas du problème d'explosion combinatoire. En effet, l'utilisation de techniques de preuve permet de décharger les obligations de preuve. De plus ces approches sont restées au niveau de l'écriture de spécifications et ne permettent pas, hormis l'approche de [Roc98b] et en partie les ICO, la génération de code.

Une description plus étoffée de cette première approche se trouve dans le chapitre 3.

Différentes limites des approches du LISI autour de la technique de B sont à considérer :

1. les propriétés liées à la représentation des tâches n'ont pas été exprimées et il n'est donc pas possible de vérifier des propriétés de validité, de complétude et de déroulement des tâches ;
2. tout est à la charge du concepteur. La modélisation de la boîte à outils, la présentation, le dialogue, le noyau fonctionnel et la formalisation des propriétés. Contrairement à l'approche complémentaire des interacteurs du CERT [Roc98b], l'approche modulaire du LISI ne s'appuie ni sur des outils pour faciliter la tâche du concepteur ni sur des schémas globaux représentant les propriétés ;
3. le dialogue n'est pas décrit. La description de processus concurrents n'est pas possible ;
4. les études de cas sont limitées aux systèmes WIMP.

2.5 Conclusion

Les contributions passées en revue dans ce chapitre ont montré la possibilité de mise en œuvre d'une démarche raisonnée d'ingénierie des systèmes interactifs basée sur des techniques formelles.

Dans le cas des interacteurs, les approches s'intéressent principalement à la couche présentation et à la composition d'interacteurs pour modéliser des interfaces complexes (systèmes concurrents et études de cas). Au contraire, les approches du LISI ou des ICO s'intéressent beaucoup plus à la modélisation complète d'un système interactif (description du noyau fonctionnel, le dialogue et la couche présentation). L'approche du LISI permet également la réutilisation par rétro-conception et l'établissement de propriétés sur les différentes parties d'un système interactif.

Ces approches sont encore trop partielles et manquent donc d'intégration à un processus complet d'ingénierie des systèmes interactifs. En effet, les propriétés vérifiées ne couvrent qu'une partie des besoins et interviennent à des étapes distinctes d'un développement complet. D'autre part, la place de l'utilisateur dans la modélisation (tâches utilisateur par exemple) n'est prise en compte que partiellement.

Chapitre 3

Approche fondée sur la preuve

L'approche fondée sur la preuve est issue des travaux menés au LISI [Bar03, AAB04, AAB05, AABG03, AABK03]. Cette approche permet de supporter différentes étapes du cycle de vie : spécification, conception, validation, vérification et raffinement. Elle a été mise en œuvre avec la technique formelle B fondée sur la preuve dans ses deux définitions : B classique qualifiée d'approche à base de modules et B événementiel qualifiée d'approche à base d'événements et permettant de supporter le contrôle de dialogue.

Toutes les modélisations B ont été implantées dans l'Atelier B, support des phases de développement, spécification, raffinement, validation et vérification. Il nous permet d'automatiser les tâches de conception en vérifiant la syntaxe des composants, en générant automatiquement des obligations de preuve et en traduisant des implantations B vers des langages de programmation. Il nous aide aussi à la preuve en déchargeant tout ou partie des obligations de preuve grâce à un prouveur automatique et interactif.

3.1 Approche à base de modules fondée sur l'utilisation du B classique

Cette première approche est fondée sur une approche de conception ascendante où le B classique est utilisé dans toute la phase de conception. Cette contribution appelée approche à base de modules. Elle permet de supporter la phase de conception et apporte une solution à la validation de tâches utilisateur. Seule la version B classique est utilisée pour construire et valider des tâches utilisateurs. L'approche de validation de tâches appelée **approche explicite** s'appuie sur une démarche

comparable aux traces explicites ou aux diagrammes de séquence d'UML dans le sens où des traces d'opérations sont explicitement construites et validées.

Nous présentons dans la suite un rappel de B, l'approche de conception et de spécification d'un système interactif en employant l'étude de cas du Post-It Notes®. Enfin nous terminons par la validation de tâches utilisateur en présentant l'approche explicite.

3.1.1 Méthode B

La méthode B a été conçue par [Abr96a], à partir des travaux de [Hoa69] sur les préconditions et postconditions et des travaux de [Dij76a] sur la plus faible précondition. C'est une méthode formelle qui s'appuie sur des bases mathématiques (logique de premier ordre et théorie des ensembles). Elle emploie une notation utilisable tout au long du développement offrant un cadre formel uniformisé allant de la spécification et la conception jusqu'à la réalisation des composants logiciels exécutables.

Un développement B débute par la construction d'un modèle reprenant les descriptions du cahier des charges, en décrivant les principales variables d'état du système, les propriétés que ces variables devront satisfaire à tout moment et les transformations de ces variables. Le modèle B obtenu constitue une spécification de ce que devra réaliser le système.

Le modèle B est ensuite raffiné, c'est-à-dire spécialisé, jusqu'à obtenir une implantation complète du système logiciel. Mais le raffinement peut également être utilisé comme technique de spécification. Il permet d'inclure *petit à petit* les détails du cahier des charges dans le développement formel. La spécification est alors réalisée progressivement et non pas directement.

La cohérence du modèle, puis la conformité des raffinements par rapport à ce modèle sont garanties par le biais des obligations de preuve. Ces dernières sont générées automatiquement grâce au calcul des substitutions généralisées inspiré des transformations de prédicat de Dijkstra. A la différence des autres techniques formelles comme Z [Spi88], la méthode B possède des outils de preuves automatiques commercialisés. Les outils les plus répandus sont Atelier-B[1] et B-Toolkit[2].

[1] ClearSy : http ://www.clearsy.com
[2] B-Core (UK) Limited : http ://www.b-core.com

MACHINE Nom
SETS
Noms de types et noms d'ensembles
CONSTANTS
Déclaration du nom des constantes
PROPERTIES
Définition des propriétés logiques des constantes
VARIABLES
Déclaration du nom des variables
INVARIANT
Définition des propriétés statiques par des formules logiques
ASSERTIONS
Définition de propriétés sur les variables et les constantes
INITIALISATION
Description de l'état initial
OPERATIONS
Énumération des opérations associées à la machine
END

Tab. 3.1 – Machine abstraite B générique

Le langage B

La *machine abstraite* est le mécanisme de structuration de base de la méthode B. Elle spécifie un élément de modélisation du système. Elle peut être comparée aux notions connues en programmation comme les classes, les modules ou les types abstraits de données. La machine abstraite encapsule les données. Elle contient des *données* (cachées) et propose à ses utilisateurs des *opérations* (visibles). Ces dernières permettent d'accéder à ces données et de les manipuler.

La structure d'une machine abstraite est définie dans le tableau 3.1. Chaque mot clé en gras définit une clause. Nous n'avons représenté que les clauses principales.

Les données d'une machine abstraite peuvent être des constantes définies par la clause **PROPERTIES** et/ou des variables typées par la clause **INVARIANT**. Ces données sont décrites par des *expressions* qui utilisent des concepts mathématiques comme les ensembles, les fonctions, etc.

Un invariant permet d'exprimer sous la forme d'un *prédicat* les exigences de fonctionnement de la machine. En d'autres termes, l'invariant définit des propriétés sur les variables qui doivent toujours être vérifiées. Une machine abstraite est dite cohérente si :

- son invariant est maintenu par l'initialisation ;
- les opérations préservent l'invariant ;
- l'assertion est respectée.

Cohérence de machine abstraite : les obligations de preuve. La cohérence d'une machine abstraite est assurée par la validation d'obligations de preuve (OP). Une OP est un théorème à démontrer, indiqué par la théorie de B et généré à partir de la description du système en question.

La modélisation des modifications des données des machines abstraites s'effectue au sein des opérations à l'aide d'un pseudo-code nommé *substitutions*. Les substitutions sont des notations mathématiques qui jouent le rôle de transformateurs de prédicats. Ces substitutions se basent sur le calcul de la plus faible précondition [Dij76b].

Les substitutions permettent également d'établir systématiquement des obligations de preuve à partir des composants B (machines abstraites, raffinements ou implantations). Quand une substitution est utilisée, le système de génération d'obligations de preuve associé au système de preuve automatiques s'assure que cette substitution est valide compte tenu des invariants de la machine et des préconditions de l'opération concernée.

Nous présentons sur le tableau 3.2 une machine abstraite générique simplifiée en B classique (sans constantes ni propriétés logiques sur ces constantes) correspondant à celles que nous employons. Il est à noter que la forme de l'opération dépend de la présence ou pas de paramètres de sortie et/ou de paramètres d'entrée.

MACHINE M
VARIABLES
x
INVARIANT
$I(x)$
ASSERTIONS
$A(x)$
INITIALISATION
$Init(x)$
OPERATIONS
$u \longleftarrow O(w) =$
S
END

TAB. 3.2 – Machine abstraite B avec une opération.

Les obligations de preuve associées à cette machine sont décrites sur le tableau 3.3.

La première INV_1 concerne l'initialisation (une opération particulière) qui établit l'invariant après l'appel de l'opération d'initialisation (il n'y a pas de valeur avant l'initialisation). La deuxième obligation de preuve INV_2 concerne une opération O qui préserve l'invariant c'est-à-dire que, sous couvert de l'invariant, cette opération

	Obligation de preuve
INV_1	$[Init(x)]I(x)$
INV_2	$I(x) \Rightarrow [S]I(x)$
INV_3	$I(x) \Rightarrow A(x)$

Tab. 3.3 – Obligations de preuve d'une machine abstraite B.

établit l'invariant. Enfin la troisième INV_3 concerne la clause **ASSERTIONS**. $A(x)$ est un prédicat que chaque état des variables x vérifiera. Ce n'est pas un invariant, c'est une propriété que nous pouvons déduire de l'invariant $I(x)$.

Une opération a différentes formes possibles selon le type de substitution utilisé. Par conséquent, selon la substitution employée l'obligation de preuve INV_2 n'est plus la même. Nous présentons sur le tableau 3.4, les principales substitutions que nous utilisons dans la suite de ce chapitre.

Nom de la substitution	Syntaxe de l'opération
bloc	**BEGIN** $T(x)$ **END**
précondition	**PRE** $P(x)$ **THEN** $T(x)$ **END**
sélection	**SELECT** $G(x)$ **THEN** $T(x)$ **END**
choix non borné	**ANY** l **WHERE** $G(x,l)$ **THEN** $T(x)$ **END**

Tab. 3.4 – Substitutions utilisées pour définir la forme d'une opération.

Nous reviendrons sur les substitutions *bloc*, *sélection* et *choix non borné* dans la section (3.2.1) traitant du B événementiel. Dans le cas où l'opération utilise une substitution *précondition*, l'obligation de preuve INV_2 serait :

$$(INV_2) \quad I(x) \wedge P(x) \Rightarrow [T(x)]I(x)$$

Enfin, nous définissons dans le tableau 3.5, les différentes formes de substitutions que nous employons pour $T(x)$. Nous en donnons pour chacune la transformation de prédicat suivant la logique de Dijkstra.

La substitution *appel d'opération* permet d'appliquer la substitution d'une opération, en remplaçant les paramètres formels par des paramètres effectifs. L'appel d'opération se définit sous quatre formes différentes, selon la présence de paramètres d'entrée et/ou de sortie. Si *op*, définie par $Y \longleftarrow op(X) = S$, la signification d'un appel de $R \longleftarrow op(E)$ (où X est une liste d'expressions représentant les paramètres

Nom	Transformation de prédicat
devient égal	$[x := E]P(x) \equiv [x := E]P(x)$
appel d'opération	$[R \longleftarrow op(E)]P(x) \equiv [X := E; T(x); R := Y]P(x)$
identité	$[skip]P(x) \equiv P(x)$
simultanée	$T(x) \parallel U(y)$ (x et y distinctes)
séquencement	$[T(x); U(x)]P(x) \equiv [T(x)][U(x)]P(x)$

TAB. 3.5 – Substitutions utilisées pour définir le corps d'une opération.

d'entrée de *op* et E une liste d'expressions représentant les paramètres d'entrée effectifs de *op*).

Quant à la substitution *simultanée*, elle permet la composition en parallèle de deux substitutions.

3.1.2 Conception à base de modules

La décomposition modulaire des applications étudiées a été représentée en B en suivant le modèle d'architecture imposé par ARCH. Différentes machines abstraites ont été définies. Conformément à ARCH, elles correspondent au noyau fonctionnel, à l'adaptateur du noyau fonctionnel (qui dans notre cas correspond à l'application principale), au contrôle du dialogue, à la présentation et à la boîte à outils.

Une étude de cas

L'étude de cas Post-It Notes® est issue du groupe de travail Flashi du GDR-PRC CHM. Elle décrit une application, à manipulation directe (c'est-à-dire les objets sont manipulés par l'utilisateur directement via une souris par exemple), représentant des Post-It Notes® (voir figure 3.1). Celle-ci est résumée dans ce qui suit.

Un bloc, duquel un utilisateur peut détacher des Post-It, sur lesquels il peut écrire des messages et les envoyer à un autre utilisateur, représente un ensemble de Post-It. Chaque Post It est lui-même représenté par une fenêtre que l'on peut agrandir, déplacer, iconifier et détruire. Les utilisateurs sont représentés par des icônes. Le passage d'un Post It sur cette icône provoque l'émission du message que contient ce Post It et son affichage sur l'écran de l'utilisateur correspondant. Par ailleurs, il n'est pas possible de détacher un Post It si le bloc n'est pas actif et seul les utilisateurs connectés sont représentés par des icônes etc. Plusieurs propriétés devant être satisfaites par ce système sont aussi définies.

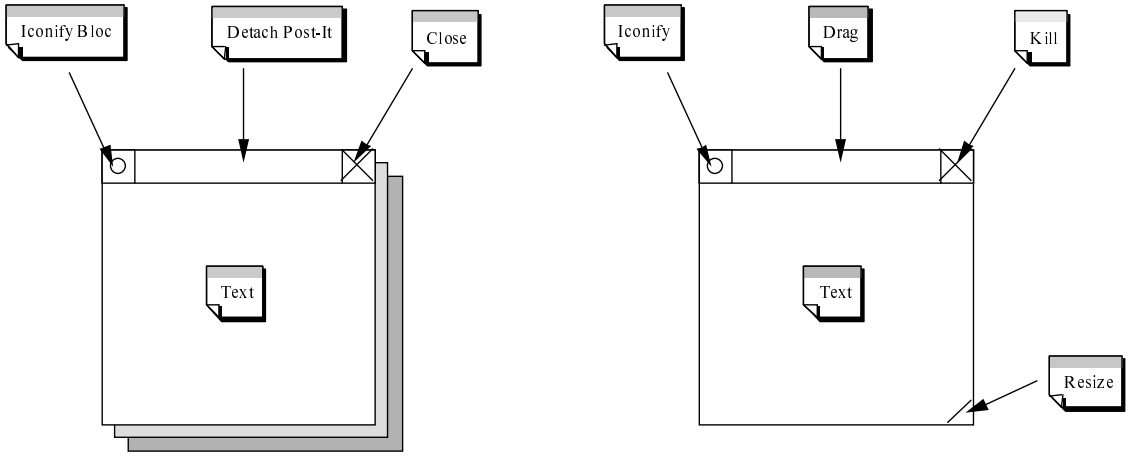


Fig. 3.1 – Interface de l'application Post-It Notes® modélisée au moyen de la méthode B.

Par rapport au modèle d'architecture ARCH, nous avons défini le découpage suivant :

- le noyau fonctionnel comprend les machines abstraites de gestion des messages, des utilisateurs et des émissions de messages ;
- l'application principale établit un lien entre les machines du noyau fonctionnel et celles de l'interface ;
- le contrôle du dialogue regroupe les machines de gestion de l'interaction (avec la souris) des différents utilisateurs, Post Its et de la souris ;
- la présentation comprend les machines abstraites nécessaires à la représentation de fenêtres (pour les Post Its et la gestion de la souris). Cette partie est en fait une rétro-conception des quelques éléments de la boîte à outils utilisée ;
- enfin, la boîte à outils, est basée sur Tcl/Tk. Une machine représentant les principales fonctions utilisées par la partie représentation a été réalisée. Cette machine est importée par les machines de représentation. Elle permet ainsi une spécification indépendante de la boîte à outils utilisée. Cette machine a été obtenue par rétro-conception de façon empirique. Nous y avons abstrait les propriétés que nous avons jugées pertinentes pour cette application. Il faut d'ailleurs approfondir ce travail afin de proposer des rétro-conceptions de boîtes à outils usuelles.

Cette approche nous a permis de formaliser les spécifications de l'application [AAGJ98b] ainsi que son développement. Nous avons représenté et vérifié avec B toutes les étapes de raffinement [AAG01]. Nous avons atteint, dans la dernière étape

une représentation dans un langage de programmation où les structures de données ne sont pas implantées.

Par exemple, les opérations de création de souris *(create_ mouse)* et de glissement de la souris *(move_ mouse_ with_ drag)* consistent à modifier l'état du système décrit dans la clause *variables* de B tout en respectant les invariants, exprimés sur ces variables, dans la clause *invariant*. L'opération de création initialise l'état à des valeurs par défaut et ajoute l'élément créé dans l'ensemble des éléments déjà créés.

$mm \longleftarrow create\_mouse =$
PRE
$mouse \neq MOUSE$
THEN
ANY
tt
WHERE
$tt \in MOUSE - mouse$
THEN
$mm := tt \;||$
$mouse := mouse \cup \{tt\} \;||$
$x\_pos\_mouse(mm) := x\_mouse\_default \;||$
$y\_pos\_mouse(mm) := y\_mouse\_default \;||$
$mouse\_state(tt) := up \;||$
$mouse\_creation(tt) := TRUE$
END
END

L'opération de déplacement de la souris sur l'écran donne les nouvelles coordonnées de la souris sur l'écran. Elle exige que les nouvelles coordonnées de la souris restent dans l'écran et nécessite l'enfoncement préalable de la souris. Cette opération est donnée par :

$move\_mouse\_with\_drag(mm, aa, bb) =$
PRE
$mm \in mouse \wedge$
$xx \in NAT \wedge xx \in 1..max\_mouse\_wide \wedge$
$yy \in NAT \wedge yy \in 1..max\_mouse\_high \wedge$
$mouse\_state(mm) = down \wedge$
$mouse\_creation(mm) = TRUE$
THEN
$x\_pos\_mouse(mm) := xx \;||$
$y\_pos\_mouse(mm) := yy$
END

Après plusieurs étapes de raffinement consistant à introduire du contrôle dans les opérations, on obtient les raffinements d'opérations suivants qui préservent toutes les propriétés décrites au départ :

```
mm ⟵ create_mouse =
BEGIN
  VAR
    tt IN tt :∈ MOUSE − mouse
  IF
    (mouse ≠ MOUSE)
  THEN
    mouse := mouse ∪ {tt};
    x_pos_mouse(mm) := x_mouse_default ;
    y_pos_mouse(mm) := y_mouse_default ;
    mouse_state(tt) := up ;
    mouse_creation(tt) := TRUE ;
    mm := tt
  END
END
```

et en :

```
move_mouse_with_drag(mm, aa, bb) =
BEGIN
  IF
    mm ∈ mouse∧
    xx ∈ NAT ∧ xx ∈ 1..max_mouse_wide ∧
    yy ∈ NAT ∧ yy ∈ 1..max_mouse_high ∧
    mouse_state(mm) = down ∧
    mouse_creation(mm) = TRUE
  THEN
    x_pos_mouse(mm) := xx;
    y_pos_mouse(mm) := yy
  END
END
```

Le dernier niveau de raffinement obtenu est décrit dans un langage impératif où les structures de données abstraites n'ont pas été implantées par des structures de données concrètes.

Nous avons tenté d'utiliser la dernière étape d'implantation de la méthode B. Pour les deux raisons suivantes, nous avons abandonné cette voie :

- la première est liée à la complexité et à la lisibilité des obligations de preuve obtenues lors de l'implantation. Ces obligations de preuve sont liées aux choix des représentations de données abstraites par des données concrètes qui est simplifiée par exemple dans le cas de spécifications algébriques. Nous nous sommes heurtés à des obligations de preuve complexes dans le cas de l'implantation des fonctions d'accès du niveau machine par un enregistrement, ou bien dans le cas de l'implantation des ensembles. Ces représentations sont connues et il existe de nombreux raffinements proposés et basés sur d'autres techniques. Nous avons privilégié la réutilisation des développements d'implantations existants et la simplification du développement ;

– la seconde raison est liée à la nature du code généré. En général, ce code (qui n'est pas fait pour être lu) n'est pas en adéquation avec les codes présents dans les boîtes à outils. La réutilisation de boîtes à outils dans le cadre d'un développement formel en B est alors compromise et il faut procéder au re-développement de la boîte à outils.

Vérification de propriétés avec la méthode B

De nombreuses propriétés ont été exprimées et vérifiées suivant notre approche de conception et d'autres sont exclues du champ de la vérification, telles que les propriétés issues des travaux des ergonomes et des psychologues qui n'ont pu être exprimées.

Les propriétés de sûreté sont exprimées dans les invariants des modèles décrits en B. Sur l'étude de cas du Post-Its Notes®, un invariant exprimé par :

$$\begin{array}{c} \forall mm \in mouse, \\ (x\_pos\_mouse(mm) \in 1..max\_mouse\_wide \wedge \\ y\_pos\_mouse(mm) \in 1..max\_mouse\_high) \end{array}$$

Cet invariant indique que tout élément de l'ensemble des souris ne peut pas dépasser les dimensions de l'écran. C'est cette propriété qui contribuera à représenter l'atteignabilité des fenêtres.

D'autres propriétés ont été formalisées :

– *Visibilité :* elle est assurée par la définition des opérations de gestion des fenêtres. Par exemple, il est possible de prouver que toute fenêtre déplacée peut rendre une fenêtre visible ou bien en cacher une autre. Cette propriété a été exprimée par l'expression de conditions d'intersection de deux fenêtres.

– *Atteignabilité :* l'intégration des machines abstraites de gestion de souris et de gestion de fenêtres permet d'assurer que la souris peut manipuler et donc atteindre toutes les fenêtres présentes sur l'écran. Ce type de propriété est vérifié grâce aux invariants de collage qui assurent que les coordonnées maximales de l'écran sont égales aux coordonnées maximales de la souris.

– *Représentation de l'information et Wysiwyg :* nous pouvons, par exemple, sur les spécifications, prouver qu'il n'y a qu'une seule fenêtre active à un instant donné, que les fenêtres ont une couleur donnée dans un état donné. Cette dernière propriété touche à une partie de l'ergonomie.

Contrairement à l'approche développée par [Roc98a], nous n'avons pas décrit de schémas globaux représentant ces propriétés. Dans notre cas, la représentation et la formalisation de ces propriétés est encore à la charge du développeur.

3.1.3 Validation de tâches par traces d'opérations : approche explicite

Nous présentons dans cette section l'approche explicite qui vise à apporter une solution à la validation de tâches dans le cadre de l'approche de conception à base de modules. De façon générale, rappelons que la phase de validation de tâches consiste à vérifier que l'ensemble des tâches utilisateurs est supporté par l'IHM.

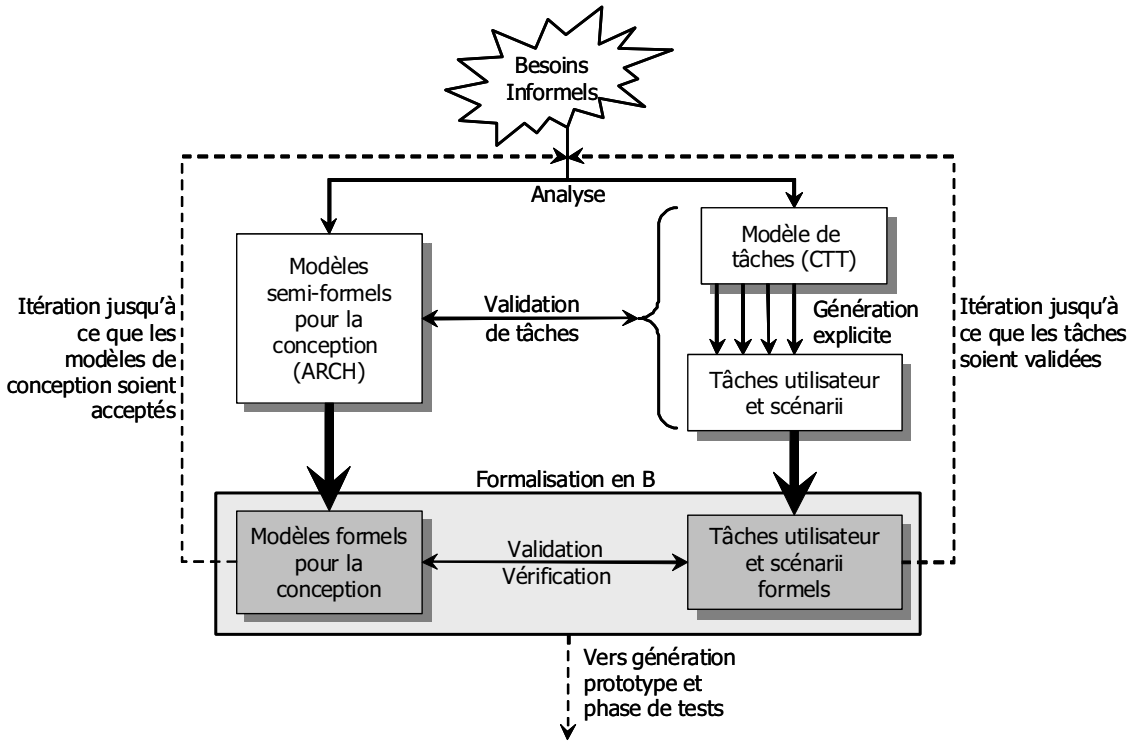


Fig. 3.2 – Description de l'approche explicite.

Sur la figure 3.2 une description de notre approche, où nous distinguons deux aspects de modélisation, est proposée.

Un premier aspect concerne la modélisation semi-formelle (partie haute de la figure) qui représente les modèles liés à la conception (*modèles semi-formels pour la conception*) et les notations liées aux tâches utilisateurs (*modèle de tâches* et *tâches utilisateur et scénarii*). Le modèle de tâches (par exemple CTT[Pat01]) permet la prise en compte des besoins de l'utilisateur dans la conception. La validation de tâches consiste à vérifier que les tâches satisfont la conception (double flèche *validation de tâches*). Toutefois, il faut pouvoir extraire du modèle de tâches (*génération*

explicite), les scénarii ou traces de tâches qui devront être validés sur la conception.

Un second aspect concerne l'approche que nous proposons qui est cadrée sur la partie basse de la figure 3.2. Elle se base sur les modèles et les notations du premier aspect (*formalisation en B*). Notre approche consiste à partir de la modélisation formelle de la conception, déjà traitée dans la section 3.1.2, puis de formaliser en B les traces de tâches construites explicitement par le concepteur à partir d'un modèle de tâches (rectangle *Tâches utilisateur et scénarii formels*) et de les valider sur la conception formelle (rectangle *Modèles formels pour la conception*). Nous précisons par les flèches en pointillées deux phases de validation. Celle de gauche indique la validation de la conception abordée en section 3.1.2 tandis que celle de droite précise la validation de tâches abordée dans cette section.

Le scénario est donc composé d'un *ensemble* de tâches, ou *traces* de tâches, ordonnées en séquence. La validation de tâches consiste alors à vérifier qu'une trace de tâches d'un scénario est supportée par la modélisation formelle de l'architecture.

Une première sous section présente la construction explicite d'un scénario générique puis enfin nous enchaînons sur sa modélisation dans la méthode B.

Description de l'approche explicite

La construction d'un scénario est vue comme la décomposition en différents niveaux d'abstraction. Au plus haut niveau de cette décomposition se trouve la tâche correspondant au but du scénario décrite par un état initial et un état final.

Cette tâche est décomposée par une séquence de plusieurs sous-tâches qui sont elles-mêmes décomposées en d'autres séquences de sous-tâches et ainsi de suite. Ce processus est appliqué jusqu'à ce que les tâches ne puissent plus être décomposées, c'est-à-dire que le niveau des opérations de la modélisation de l'IHM est atteint. Ce niveau correspond aux actions effectuées par le système ou par l'utilisateur sur l'interface homme-machine. Plus concrètement, les tâches terminales correspondent aux opérations du contrôleur de dialogue que nous appelons *opérations atomiques*.

Sur la figure 3.3 nous montrons une décomposition de $Tâche_1$ dans un ensemble ordonné de tâches par $Tâche_2$ et $Tâche_3$ qui sont elles-mêmes décomposées en sous-tâches. Les tâches feuilles (par exemple $Tâche_4$ ou $Tâche_7$) décrivent le niveau des modifications sur l'IHM. Ces tâches terminales sont associées aux opérations atomiques du contrôleur de dialogue notées op_i.

La représentation de la décomposition est comparable aux modèles de tâches hiérarchiques tels que ceux issus de notations de la section 1.3.1. Cependant, la sémantique employée pour le décrire est pauvre. Il n'existe pas de précondition de tâches et ce modèle de tâches ne possède que l'opérateur de séquence pour structurer

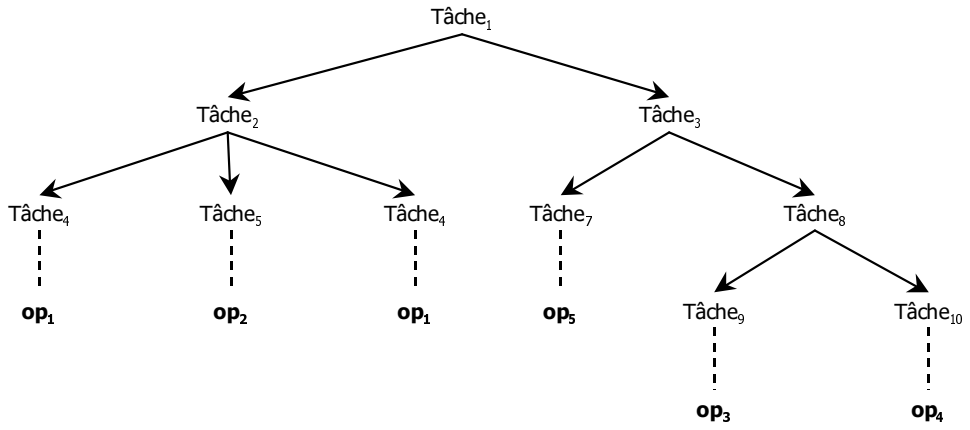


Fig. 3.3 – Décomposition de tâches en sous-tâches.

la décomposition. Celle-ci permet d'obtenir une trace d'opérations atomiques qui devra être supportée par la modélisation de l'architecture de l'application, un peu à la manière des diagrammes de séquence de UML. La décomposition complète de l'exemple présenté sur la figure 3.3 est donnée sur le tableau 3.6 où $Tâche_i$ désignent les tâches et où Op_i sont les opérations atomiques du contrôleur de dialogue.

$\text{Tâche}_1 = \text{Tâche}_2 \ ; \ \text{Tâche}_3$
$\text{Tâche}_2 = \text{Tâche}_4 \ ; \ \text{Tâche}_5 \ ; \ \text{Tâche}_4$
$\text{Tâche}_3 = \text{Tâche}_7 \ ; \ \text{Tâche}_8$
$\text{Tâche}_3 = \text{Tâche}_9 \ ; \ \text{Tâche}_{10}$
$\text{Tâche}_1 = \text{Tâche}_4 \ ; \ \text{Tâche}_5 \ ; \ \text{Tâche}_4 \ ; \ \text{Tâche}_7 \ ; \ \text{Tâche}_9 \ ; \ \text{Tâche}_{10}$
$\mathbf{Tâche}_1 = \mathbf{Op}_1 \ ; \ \mathbf{Op}_2 \ ; \ \mathbf{Op}_1 \ ; \ \mathbf{Op}_5 \ ; \ \mathbf{Op}_3 \ ; \ \mathbf{Op}_4$

Tab. 3.6 – Décomposition en trace de tâches et d'opérations atomiques du contrôleur de dialogue.

Enfin, nous aurions pu donner directement la trace à tester sans passer par des étapes de décomposition de tâches en sous-tâches. Seulement, nous partons de l'hypothèse que l'utilisateur en charge de la validation ne connaît pas *à priori* la conception. Il doit alors décomposer par niveaux hiérarchiques les tâches (comportement abstrait de la tâche) jusqu'à aboutir au niveau des feuilles de l'arbre, c'est-à-dire les opérations atomiques du contrôleur de dialogue. Seulement, la décomposition hiérarchique que nous proposons ne comporte qu'un opérateur : la séquence.

Approche explicite et B

La tâche de plus haut niveau est décrite en B classique dans une machine abstraite. Nous utilisons la technique de raffinement pour construire la décomposition de cette tâche dans un ensemble ordonné de sous-tâches. Le raffinement permet alors d'introduire l'opérateur de *séquence* et les sous-tâches ou opérations. Dés lors que toutes les opérations atomiques du contrôleur de dialogue sont atteintes, le processus de raffinement se termine. Les obligations de preuve associées à la technique de raffinement assurent ainsi que la décomposition est correcte.

Plus concrètement, la machine abstraite qui décrit les tâches exploite la modélisation de la conception en étendant le module du contrôleur de dialogue qui représente ainsi le point d'entrée de la conception. Cette machine abstraite définit aussi une opération unique qui décrit le comportement de la tâche de plus haut niveau au moyen de la substitution bloc *BEGIN S END*. Par ailleurs, les sous-tâches ($Tâche_2$, $Tâche_3$, ...) sont définies chacune par une opération décrite dans la modélisation de la conception. A l'opposé de la décomposition hiérarchique de la section 3.1.3 et grâce à B, ces opérations permettent la description de la postcondition d'une tâche en effectuant des effets de bords sur les variables de la conception.

L'étape de raffinement consiste à enrichir l'unique opération (tâche de haut niveau) de la machine abstraite afin d'introduire le déclenchement en séquence de sous-tâches (appel en séquence des opérations qui servent à décrire les postconditions des sous-tâches). Quand la trace d'opérations atomiques du contrôleur de dialogue est atteinte, le processus de raffinement peut être arrêté (tout dépend de l'usage de la modélisation qui peut être fait). La modélisation de la validation de tâches est alors suffisante.

Les propriétés à vérifier sont celles de la modélisation de l'IHM. Trois aspects de la validation sont concernés :

- en premier, la trace d'opérations atomiques du contrôleur de dialogue montre qu'il existe une séquence d'opérations qui implémente la tâche de plus haut niveau. Cet aspect concerne la validation de tâches ;
- en deuxième, si une des opérations n'est pas présente et/ou des obligations de preuve concernant ces opérations atomiques ne peuvent être déchargées, dans les machines abstraites B de la conception, alors, nous pouvons affirmer que des opérations atomiques sont manquantes et/ou la spécification est défectueuse. La conception devra être améliorée et/ou complétée. Cet aspect concerne la validation de la conception ;
- en troisième, nous pouvons ajouter de nouvelles propriétés dans la clause **INVARIANT** portant sur les variables de la conception. Cet aspect permet de vérifier de nouvelles propriétés en évitant le test.

Enfin, notons que chaque tâche (ou scénario) à valider conduit à la mise en place d'une machine abstraite construite suivant le principe de construction de traces d'opérations. Toutes ces machines abstraites obtenues, indépendantes les unes des autres, étendent la modélisation du contrôleur de dialogue.

3.1.4 Bilan sur l'approche à base de modules

La phase de conception se fait de façon ascendante. Elle est ainsi entièrement basée sur une structuration modulaire des machines abstraites pour composer des sous-systèmes afin d'aboutir au système complexe.

L'approche explicite repose sur la validation de tâches par raffinements successifs. L'objectif est d'obtenir une trace d'opérations atomiques (du contrôleur de dialogue) par décomposition en sous-tâches où le seul opérateur de contrôle est la séquence. Si ce développement est valide (toutes les obligations de preuve générées ont été déchargées), alors la tâche est valide. Elle permet donc la validation a priori de tâches (faisabilité), c'est-à-dire qu'il existe une séquence d'opérations qui implémente la tâche de plus haut niveau. Elle permet aussi la validation a priori de la conception (complétude), c'est-à-dire que toutes les obligations de preuve de la modélisation de la conception sont déchargées.

Cette approche ne nécessite pas de modification de la partie conception. Par ailleurs, la trace à valider est construite au moyen de la méthode B dans sa version classique. Il y a donc homogénéité des langages de modélisation entre la partie conception et validation. L'intérêt principal est d'éviter au concepteur l'apprentissage d'une nouvelle technique formelle à chaque phase de développement.

Toutefois, deux critiques de l'approche à base de modules peuvent être effectuées :

1. la première concerne la phase de conception. L'approche à base de modules ne permet pas de décrire l'état du dialogue d'une application interactive. En effet, elle ne définit qu'un ensemble d'opérations mais pas les événements qui permettent de les déclencher ni leur ordre. La description de systèmes concurrents n'est donc pas possible. Cette lacune est due principalement à la sémantique de B classique qui ne permet pas de décrire des systèmes réactifs comme les IHM. Par ailleurs, la conception ascendante oblige à reprouver à chaque composition des obligations de preuve plus complexes ;
2. la seconde concerne la phase de validation de tâches. Nous avons montré que l'établissement de scénarii reste fastidieux à mettre en place. Seul l'opérateur de séquence permet de concevoir la trace d'opérations. En comparaison avec les formalismes possédant une forte capacité à structurer, comme CTT, l'ap-

proche explicite représente hiérarchiquement les différents opérateurs (interruption, désactivation, entrelacement et itérations) directement dans la trace par l'opérateur de séquence à la manière du diagramme de séquence du formalisme UML. Finalement, l'approche explicite peut être comparée aux approches formelles basées sur la vérification sur modèles (*model-checking*). D'une part, il est nécessaire de décrire l'intégralité des séquences de tâches. D'autre part, à chaque validation de scénario toutes les propriétés de la modélisation doivent être une nouvelle fois vérifiées.

L'approche des systèmes interactifs que nous proposons dans la suite propose une réponse aux deux critiques formulées ci-dessus au moyen de B événementiel qui offre les caractéristiques suivantes :

1. **comportement dynamique du contrôleur de dialogue** : avant tout rappelons qu'un système modélisé en B événementiel décrit un système réactif à base d'événements. Il est donc possible de définir le comportement d'un système interactif. Dans ce sens, nous étudierons la modélisation d'un contrôleur de dialogue en définissant son comportement en B événementiel. Nous exploitons une forme de conception dite descendante au moyen de la technique de raffinement de la méthode B qui permettra d'introduire au fur et à mesure les détails de l'interaction ;

2. **notation de description de l'utilisateur** : du point de vue de la phase de validation de tâches, B événementiel permettra le codage d'opérateurs de composition de tâches autres que la simple séquence. Nous pourrons employer un langage de modélisation de tâches utilisateurs en l'occurrence ConcurTaskTrees pour valider les tâches utilisateurs. L'apport de nos travaux dans cette optique repose donc sur la description formelle de la sémantique de la notation CTT au moyen de B événementiel. La validation formelle de tâches exprimée en CTT sera rendue possible.

L'approche à base de modules fondée sur l'utilisation du B classique a été appliquée sur l'étude de cas du convertisseur francs/euros et compteur. Nous ne discuterons pas des résultats obtenus de cette étude de cas dans ce rapport mais ils peuvent être trouvés dans [Bar03]. Au contraire nous nous attarderons plus longuement sur la présentation des propriétés vérifiées de cette étude de cas dans la prochaine section traitant de l'approche à base d'événements choisie dans le cadre du projet VERBATIM.

3.2 Approche à base d'événements fondée sur l'utilisation du B événementiel

Cette seconde contribution que nous appelons **approche à base d'événements** n'a pu être mise en place que par l'utilisation du B événementiel et de la technique de raffinement associée. En effet, dans l'approche modulaire toute la modélisation est basée sur la sémantique du B classique. Il n'était pas possible de définir le comportement dynamique de systèmes interactifs. Cette approche exploite également les travaux sur l'approche modulaire développés dans 3.1. Elle enrichit ces travaux pour permettre de décrire le comportement dynamique du contrôleur de dialogue. Ici, seul le contrôleur de dialogue est modélisé en B événementiel. Il fait appel aux opérations des autres machines exprimées en B classique. Les travaux sur la validation de tâches, appelés **approche implicite** exploitent une notation de tâches, en l'occurrence ConcurTaskTrees, issue du domaine des IHM.

3.2.1 B événementiel

La méthode B dans sa version classique ne permet pas de tenir compte de toutes les propriétés d'un système. Par exemple, il n'est pas envisageable d'exprimer des systèmes concurrents et interactifs et leurs propriétés. C'est dans ce but que [Abr96b] a étendu les possibilités d'application de la méthode B, sans changer la théorie sous jacente. Cette nouvelle extension, appelée B événementiel, introduit la notion de raffinement du comportement indispensable aux systèmes à base d'événements. Le B événementiel permet d'envisager l'application plus en amont dans les phases de modélisation, et respecte l'approche descendante : les propriétés attendues sont prises en compte dés le début de la spécification et sont préservées tout au long des raffinements successifs. Cette extension permet donc de s'intéresser par exemple aux systèmes distribués, réactifs, interactifs et multi-modaux.

De façon schématique, l'utilisation du B classique permet la description d'un système où la machine abstraite peut-être vue comme un automate. Les variables d'états sont définies par un invariant et les transitions étiquetées sont représentées par les opérations. Seulement, la sémantique du B classique ne permet pas de définir un comportement réactif. Au contraire avec l'extension B événementiel, le système passe d'un comportement passif à un comportement réactif.

Dans le B événementiel, la notion de machine abstraite n'existe plus et a été remplacée par celle de *modèle*. Nous ne parlons plus d'opérations, mais d'*événements*. La syntaxe de la méthode B est rendue plus simple. Il n'y a plus de notion de

MODEL Nom
  SETS
    Noms de types et noms d'ensembles
  CONSTANTS
    Déclaration du nom des constantes
  PROPERTIES
    Définition des propriétés logiques des constantes
  VARIABLES
    Déclaration du nom des variables
  INVARIANT
    Définition des propriétés statiques par des formules logiques
  ASSERTIONS
    Définition de propriétés sur les variables et les constantes
  INITIALISATION
    Description de l'état initial
  EVENTS
    Énumération des événements associés au modèle
END

TAB. 3.7 – Modèle B événementiel générique.

précondition car le système modélisé est clos. Le séquencement a disparu, puisque les nouveaux événements réalisent l'enchaînement séquentiel des instructions. Il n'y a plus de boucles car là aussi les événements pourront être déclenchés tant que leur garde est vraie. Cette évolution apporte un allégement de la complexité des preuves mais aussi une meilleure compréhension des modélisations. Nous donnons sur le tableau 3.7 la structure d'un modèle B événementiel générique.

Nous allons focaliser notre présentation sur la notion d'événement, sur la préservation de l'invariant, sur le raffinement puis sur l'utilisation du B classique pour traduire le B événementiel. Cette présentation du B événementiel n'est pas complète. Elle est basée sur les travaux réalisés par [Can03] et des informations complémentaires peuvent y être trouvées.

Les événements

Un événement correspond à un changement d'état et modélise donc une transition discrète du système à modéliser. Un événement possède un nom, une garde (condition nécessaire au déclenchement de l'événement) et une action ou encore appelée corps de l'événement. L'action est définie par une substitution qui explique comment l'événement modifie les variables.

Le langage B est un langage asynchrone. En effet, si deux événements d'un modèle ont leurs gardes vraies au même instant, ils ne sont pas déclenchés en même temps : il y a entrelacement des événements dans un ordre non déterminé. Toutefois, la durée

d'exécution d'un événement est considérée comme nulle et cet instant correspond au changement d'état.

Le B événementiel n'admet que trois formes possibles d'événements. Nous les présentons ci-dessous. Dans ce qui va suivre, nous choisissons la variable x comme variable d'état, définie dans la clause **VARIABLES**.

Événement simple. L'événement suivant est le plus simple. Sa garde est toujours vraie et $S(x)$ est une substitution qui modifie l'état de la variable x :

nomEvt =
BEGIN
  $S(x)$
END

Événement gardé. Un événement gardé est une substitution $S(x)$ gardée par l'expression logique $G(x)$. L'événement se déclenche lorsque la garde $G(x)$ est vraie.

nomEvt =
SELECT
  $G(x)$
THEN
  $S(x)$
END

Événement indéterministe. L'événement suivant est un événement indéterministe gardée par $\exists l.G(l,x)$. Cet événement ne peut se déclencher que s'il existe des valeurs pour les variables locales l qui satisferont la condition $G(x,l)$.

nomEvt =
ANY l **WHERE**
  $G(x,l)$
THEN
  $S(x,l)$
END

Préservation de l'invariant

Une fois que le système est construit, il faut montrer qu'il est cohérent. Suivant la même démarche que pour le B classique, il faut montrer que l'événement d'initialisation préserve l'invariant et que chaque événement du système préserve également l'invariant. Plus précisément, pour qu'un événement soit faisable, il faut que sous couvert de la garde de l'événement et l'invariant une transition de cet événement soit toujours possible.

Initialisation. L'événement d'initialisation est une substitution de la forme $Init(x)$. Nous avons donc une obligation de preuve identique à celle du B classique :

$$(INV_1) \quad [Init(x)]I(x)$$

Événement simple. Pour une action de la forme $S(x)$ l'obligation de preuve est la suivante :

$$(INV_2) \quad I(x) \Rightarrow [S(x)]I(x)$$

Cet événement est faisable sous couvert de l'invariant $I(x)$ et si le prédicat obtenu après transformation de l'invariant par la substitution $S(x)$ établit cet invariant.

Événement gardé. Pour l'événement gardé par $G(x)$ et de substitution $S(x)$ comme action, l'obligation de preuve est la suivante :

$$(INV_3) \quad I(x) \Rightarrow (G(x) \Rightarrow [S(x)]I(x)) \text{ ou alors } I(x) \wedge G(x) \Rightarrow [S(x)]I(x)$$

Événement indéterministe. Pour l'événement gardé par $\exists l.G(l,x)$ et de substitution $S(x)$ comme action, l'obligation de preuve de l'événement indéterministe est la suivante :

$$(INV_4) \quad I(x) \Rightarrow (\forall l.(G(l,x) \Rightarrow [S(x,l)]I(x)))$$

Récapitulatif des obligations de preuve pour l'invariant. Le tableau 3.8 récapitule les obligations de preuve de préservation de l'invariant.

	Nature de l'événement	Obligation de preuve
INV_1	Initialisation	$[Init(x)]I(x)$
INV_2	Simple	$I(x) \Rightarrow [S(x)]I(x)$
INV_3	Gardé	$I(x) \Rightarrow (G(x) \Rightarrow [S(x)]I(x))$
INV_4	Indéterministe	$I(x) \Rightarrow (\forall l.(G(l,x) \Rightarrow [S(x,l)]I(x)))$

TAB. 3.8 – Obligations de preuve d'un modèle B événementiel.

Raffinement

Le mécanisme du raffinement consiste à reformuler, par étapes successives un modèle abstrait en une suite de modèles plus précis (modèle concret) où l'on pourra avoir plus de détails en ajoutant des variables et où l'on pourra observer plus finement en ajoutant de nouveaux événements [Can03]. Nous donnons sur le tableau 3.9 la forme d'un modèle raffiné.

REFINEMENT Nom
REFINES
Noms du modèle abstrait
SETS
Noms de types et noms d'ensembles
CONSTANTS
Déclaration du nom des constantes
PROPERTIES
Définition des propriétés logiques des constantes
VARIABLES
Déclaration du nom des variables
INVARIANT
Définition des propriétés statiques par des formules logiques
VARIANT
Définition du variant décrémenté par les événements
ASSERTIONS
Définition de propriétés sur les variables et les constantes
INITIALISATION
Description de l'état initial
EVENTS
Énumération des événements associés au modèle
END

Tab. 3.9 – Raffinement de modèles B événementiel générique.

Un raffinement est un modèle dont les comportements sont des comportements de l'abstraction. Il satisfait donc l'invariant abstrait. Il faut aussi faire correspondre les variables de l'abstraction à celles du raffinement par un invariant (défini dans la clause **INVARIANT** du raffinement) que nous appelons invariant de collage $J(x, y)$ car une partie de cet invariant explicite comment sont liés les ensembles de variables. L'autre partie de cet invariant est utilisée pour représenter des propriétés sur les nouvelles variables introduites dans le raffinement.

Les raffinements conservent les événements abstraits (même nom) par contre ces événements peuvent très bien être reformulés (garde et action) afin de réduire le non-déterminisme et pour les rendre plus précis.

Nous proposons sur le tableau 3.10 le modèle concret R qui raffine le modèle abstrait M et nous donnons maintenant les obligations de preuve associées.

Le raffinement de l'initialisation. L'obligation de preuve pour l'initialisation consiste à montrer que l'initialisation concrète $Init(y)$ doit établir qu'il est impossible que l'initialisation abstraite $Init(x)$ établisse la négation de l'invariant de collage.

REFINEMENT R
REFINES
M
VARIABLES
y
INVARIANT
$J(x, y)$
VARIANT
$V(y)$
INITIALISATION
$Init(y)$
EVENTS
$<$ *liste d'événement* $>$
END

TAB. 3.10 – Raffinement B événementiel.

L'obligation de preuve associée à l'initialisation est la suivante :

$$(REF_1) \quad [Init(y)]\neg[Init(x)]\neg J(x, y)$$

Le raffinement d'événement. De même, des obligations de preuve qui permettent de démontrer l'établissement par conservation d'un événement lors de son raffinement sont engendrées. Nous donnons de façon générale l'obligation de preuve de raffinement d'une forme quelconque en forme quelconque où les gardes des événements abstrait et concret sont $G(x)$ et $H(y)$ et où les substitutions abstraite et concrète sont $S(x)$ et $T(y)$.

Cette obligation de preuve est la suivante :

$$(REF_2) \quad I(x) \wedge J(x, y) \wedge G(x) \Rightarrow H(y) \wedge [T(y)]\neg[S(x)]\neg J(x, y)$$

Remarquons que l'on ne reprouve pas l'invariant abstrait car la preuve de cet invariant a déjà été faite dans l'abstraction. A titre d'exemple nous donnons l'obligation de preuve d'un événement de forme simple en forme indéterministe $(\exists l.H(y, l))$. Leurs descriptions sont données ci-dessous :

BEGIN	**ANY** l **WHERE**
$S(x)$	$H(y, l)$
END	**THEN**
	$T(y, l)$
	END

L'obligation de preuve de raffinement est la suivante :

$$(REF_2) \quad I(x) \wedge J(x,y) \Rightarrow \exists l.H(y,l) \wedge [T(y,l)]\neg[S(x)]\neg J(x,y)$$

Le raffinement des nouveaux événements. De nouveaux événements peuvent apparaître dans le modèle concret. Ils servent à préciser le modèle abstrait par l'observation de nouveaux événements. Ces nouveaux événements raffinent tous *skip* ($x := x$). Le nouveau modèle doit être aussi vivant que son abstraction, il ne doit pas se bloquer plus que son abstraction.

Nous aurons l'obligation de preuve suivante :

$$(REF_3) \quad I(x) \wedge J(x,y) \Rightarrow H(y) \wedge [T(y)]J(x,y)$$

Le raffinement de modèles. Il faut s'assurer que le modèle raffiné ne se bloque pas plus que le modèle abstrait. Dans ce but, nous ajoutons une obligation de preuve qui nous assure cette propriété.

Cette obligation de preuve est présentée ci-dessous :

$$(REF_4) \quad I(x) \wedge J(x,y) \wedge (G_1(x) \vee \ldots \vee G_n(x)) \Rightarrow (H_1(y) \vee \ldots \vee H_m(y))$$

Cette obligation de preuve énonce que sous couvert des invariants (de l'abstraction et du raffinement) et de la disjonction des gardes abstraites, nous pouvons en déduire la disjonction des gardes concrètes (c'est-à-dire un événement peut se déclencher à tout moment).

Par ailleurs, lorsque les nouveaux événements raffinent **skip** (qui peut se déclencher à tout instant), il faut s'assurer que les événements ne prennent pas la main indéfiniment. Pour cela nous définissons un variant $V(y)$ (un entier naturel), défini dans la clause **VARIANT** du raffinement, que chaque nouvel événement fait décroître pour assurer le non blocage. Les obligations de preuve sont donc les suivantes :

$$(REF_5) \quad I(x) \wedge J(x,y) \Rightarrow V(y) \in N$$

$$(REF_6) \quad I(x) \wedge J(x,y) \wedge (V(y) = \lambda) \Rightarrow [S(y)](V(y) < \lambda)$$

Récapitulatif des obligations de preuve pour l'invariant. Le tableau 3.11 récapitule l'ensemble des obligations de preuve d'un raffinement.

	Obligation de preuve
REF_1	$[Init(y)]\neg[Init(x)]\neg J(x,y)$
REF_2	$I(x) \wedge J(x,y) \wedge G(x) \Rightarrow H(y) \wedge [T(y)]\neg[S(x)]\neg J(x,y)$
REF_3	$I(x) \wedge J(x,y) \Rightarrow H(y) \wedge [T(y)]J(x,y)$
REF_4	$I(x) \wedge J(x,y) \wedge (G_1(x) \vee \ldots \vee G_n(x)) \Rightarrow (H_1(y) \vee \ldots \vee H_m(y))$
REF_5	$I(x) \wedge J(x,y) \Rightarrow V(y) \in N$
REF_6	$I(x) \wedge J(x,y) \wedge (V(y) = \lambda) \Rightarrow [S(y)](V(y) < \lambda)$

TAB. 3.11 – Obligations de preuve d'un raffinement en B événementiel.

3.2.2 Modélisation du contrôleur de dialogue à base d'événements

L'apport principal de l'utilisation de B événementiel pour la modélisation de systèmes interactifs est la possibilité de décrire les comportements, en particulier le contrôleur de dialogue. En effet, celui-ci est un composant logiciel essentiel dans un système interactif. Il reçoit les événements en entrée et déclenche des événements qui a leur tour déclenchent des événements du noyau fonctionnel et/ou de la présentation et des boîtes à outils.

Dans cette approche, le contrôleur de dialogue est représenté en B événementiel par un ensemble d'événements gardés, qui sont activés lorsque la garde est vraie. Cet ensemble d'événements définit un système de transitions étiquetés (par les événements). Chaque transition fait passer le système d'un état (identifiés par les variables du système) à un autre. Le système ainsi décrit est asynchrone avec entrelacement d'événements.

La conception avec B événementiel d'un tel contrôleur se fait par décomposition. Un ensemble d'événements décrivant une spécification de départ est raffiné jusqu'à obtention des événements du niveau implémentation. Des propriétés invariantes ainsi que le non blocage sont préservées par ces raffinements.

A l'heure actuelle, des développements ad'hoc ont été menés. L'étude de cas complète du convertisseur francs/euros a été développée en suivant ce type de décomposition. A titre d'illustration nous donnons une partie de la spécification de ce convertisseur francs/euros. Notons que les travaux en cours dans le projet VERBATIM visent à définir une méthodologie de conception de tels systèmes.

Application à une étude de cas : le convertisseur francs/euros et le compteur.

L'étude de cas est composée de deux applications fonctionnant en symbiose : une application de conversion francs/euros et un compteur. Après chaque conversion l'application compteur incrémente d'une unité une valeur compteur et l'affiche. Au bout de trois conversions, l'application compteur interdit la conversion tant que l'utilisateur n'a pas ré-initialisé la valeur compteur.

Par rapport à la conception de l'approche par modules, l'approche à base d'événements nous a permis d'exprimer des propriétés supplémentaires et notamment les propriétés de non-blocage grâce à la disjonction des gardes.

L'extrait ci-dessous présente la disjonction des gardes d'un raffinement dans la clause **ASSERTIONS**. Elle établit des propriétés sur les variables d'état de l'application :

ASSERTIONS
/* Garde de l'événement *evtClickInitialiser* */
$(compteur = 3 \wedge$
$b\_mouse\_y \in b\_frame\_y + b\_button\_EF\_y..b\_button\_EF\_high + \ldots \wedge$
$b\_mouse\_state = clicked \wedge b\_buttonEF\_state = unpressed \wedge \ldots) \vee$
/* Garde des autres événements */
...

L'événement *evtClickInitialiser* est l'événement déclenché quand l'utilisateur clique sur le bouton **intilialiser** de l'application compteur. Nous vérifions par l'intermédiaire de cet événement que la position du curseur de la souris se trouve sur le bouton **initialiser** (c'est une propriété d'affichage) et la valeur du compteur qui doit être à trois (c'est une propriété fonctionnelle).

3.2.3 Validation de tâches par modèle de tâches CTT : approche implicite

Les insuffisances de l'approche explicite évoquées en section 3.1.3 nous ont conduit à proposer une extension à l'approche explicite où nous utilisons un langage de modélisation de tâches utilisateurs défini par la communauté IHM, en l'occurrence ConcurTaskTrees, qui permettra :

- de décrire des tâches complexes par des expressions combinées à des opérateurs temporels ;
- d'éviter la définition de traces pour chaque tâche grâce à l'introduction d'autres opérateurs de contrôle. Il sera inutile d'énumérer toutes les séquences possibles

des tâches à valider. Elles seront simplement caractérisées par le modèle de tâches CTT.

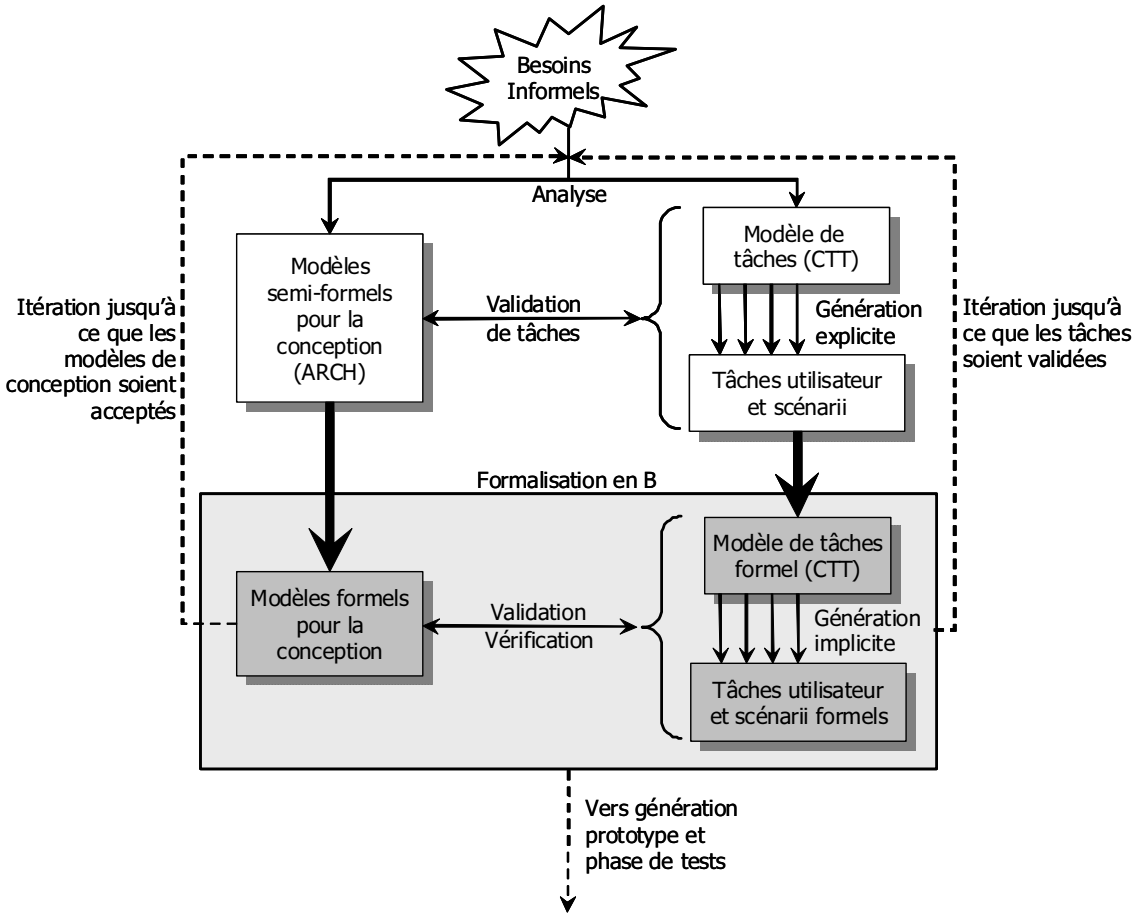


FIG. 3.4 – Description de l'approche implicite.

L'approche implicite que nous présentons est résumée sur la figure 3.4. Nous distinguons sur cette figure les mêmes aspects que sur la figure 3.2, c'est-à-dire une partie haute qui concerne la modélisation par des modèles et notations semi-formels et une partie basse qui concerne le développement formel en utilisant les modèles et notations de la partie haute. Alors que dans l'approche explicite (section 3.1.3) la formalisation des besoins utilisateur s'effectuait à partir des scénarii et des traces de tâches, construits explicitement par l'expertise du concepteur, l'approche implicite formalise directement le modèle de tâches.

L'avantage de cette approche est de pouvoir caractériser implicitement les scénarii et les traces de tâches à partir du langage de description de tâches CTT. Dans ce cas il y a génération implicite de tâches utilisateur et scénarii formels.

Cependant, la sémantique utilisée par CTT est une sémantique semi-formelle. C'est pourquoi, nous montrons dans cette section comment la sémantique de ConcurTaskTrees peut être formellement décrite dans la méthode B nous donnant ainsi la possibilité d'exprimer à la fois la conception de l'IHM et l'expression des tâches utilisateurs dans une même technique formelle, la méthode B dans sa version événementielle.

Nous décrivons formellement la sémantique de CTT dans la méthode B suivant les règles de la grammaire BNF de CTT proposée dans le tableau 3.12. Elle présente aussi les opérateurs temporels et les caractéristiques de tâches (itération, tâche optionnelle, etc). Cette représentation permet de mettre en avant le fait que les tâches sont décomposables en sous-tâches et qu'une tâche feuille est atomique.

Task : :=	$Task >> Task$	- - Activation
|	$Task[]Task$	- - Choix
|	$Task_{At}$	- - Tâche atomique
|	$Task \models\mid Task$	- - Ordre indépendant
|	$[Task]$	- - Tâche optionnelle
|	$Task[> Task$	- - Désactivation
|	$Task^*[> Task$	- - Désactivation d'une tâche itérative
|	$Task\mid > Task$	- - Interruption
|	$Task\mid\mid Task$	- - Concurrence
|	$Task^N$	- - Tâche itérative finie

Tab. 3.12 – Grammaire BNF de la notation de tâches ConcurTaskTrees.

Ces règles de grammaire ne présentent que l'aspect temporel du déroulement entre tâches. Cet aspect est défini par la sémantique de Lotos [Sys84b]. A cela s'ajoutent aussi les informations relatives à la manipulation des objets par les tâches, il s'agit plus particulièrement de la formalisation des catégories (tâche abstraite, interactive, etc), des préconditions et des postconditions.

L'approche de formalisation que nous avons adoptée consiste à représenter l'algèbre de processus sous jacente à CTT dans B événementiel. Cette formalisation permettra le codage, par raffinement des arbres de tâches issus de CTT. Chaque opérateur de CTT est représenté par un raffinement. Ainsi une règle BNF de la forme $T_0 ::= T_1\ op\ T_2$ est décomposée en deux machines : la première contient l'événement T_0 et la seconde raffinant la première contient les événements T_0, T_1 et T_2 et effectue la décomposition de T_0 en $T_1\ op\ T_2$.

Afin d'illustrer ce mécanisme, nous donnons la décomposition sur les opérateurs de base de CTT à partir d'un exemple simple : l'addition de deux nombres.

Un exemple simple comme illustration : l'addition. La somme Sum de deux nombres naturels aa et bb sera utilisée pour illustrer le fonctionnement des règles de traduction du langage CTT en B événementiel. Nous donnerons plusieurs raffinements différents permettant de calculer la somme des deux nombres naturels. Le premier modèle B événementiel $Task\_Sum_{T0}$ représente une instance de la tâche T_0.

MODEL $Task\_Sum_{T0}$
INVARIANT
$Sum \in NAT \wedge aa \in NAT \wedge bb \in NAT1$
INITIALISATION
$Sum :\in NAT \parallel aa :\in NAT \parallel bb :\in NAT$
EVENTS
$Evt_0 =$
BEGIN
$Sum := aa + bb$
END ;

La clause **INITIALISATION** permet d'initialiser l'ensemble des variables de telle façon que Sum, aa et bb choisissent des valeurs parmis des nombres naturels. Ce modèle contient l'événement Evt_0 où sa garde (G_0 du modèle générique) est vraie et où son corps permet le calcul de la somme.

Pour les différents raffinements de cette illustration, nous utiliserons $RSum$, AA et BB comme nouvelles variables de raffinement. Elles correspondent respectivement aux variables raffinées de celles abstraites Sum, aa et bb. Ces variables sont liées entre elles par l'invariant de collage ($RSum + AA + BB = aa + bb$) qui assure la correspondance entre les deux niveaux de modélisation et par conséquent au codage des opérateurs CTT. Enfin, dans les différents raffinement nous utiliserons une expression d'initialisation $aa, AA :\in (aa \in NAT \wedge AA = aa)$ qui assurera d'une part que les variables aa et AA sont choisies arbitrairement parmis des nombres naturels et d'autre part qu'elles sont égales. La même expression s'appliquera également aux variables bb et BB.

Dans la suite, l'entier Naturel $AA + BB$ déterminera le variant. Il décroîtra jusqu'à 0 pour déclencher l'événement raffiné de l'abstraction.

Activation : $T_0 ::= T_1 >> T_2$. Nous considérons ici que la somme de aa et bb soit réalisée de manière séquentielle. Tout d'abord la variable aa est affectée à $RSum$ par l'événement Evt_1 et ensuite la variable bb est ajoutée à $RSum$ par l'événement Evt_2. Ces deux événements du raffinement travaillent pour l'événement Evt_0 dont le rôle est de modifier Sum et de raffiner Evt_0 de l'abstraction.

REFINEMENT $RefEnablingTask_{T0}$
REFINE $Task\_Sum_{T0}$
INVARIANT
$(RSum + AA + BB) = (aa + bb) \wedge \ldots$
VARIANT
$(AA + BB)$
ASSERTIONS
$(AA = 0 \wedge BB = 0 \wedge aa + bb \in NAT) \vee \ldots$
INITIALISATION
$RSum := 0 \parallel Sum :\in NAT \parallel aa, AA :\in (aa \in NAT \wedge AA = aa) \parallel \ldots$

EVENTS

$Evt_1 =$	$Evt_2 =$	$Evt_0 =$
SELECT	**SELECT**	**SELECT**
$AA \neq 0 \wedge BB \neq 0 \wedge \ldots$	$AA = 0 \wedge BB \neq 0 \wedge \ldots$	$AA = 0 \wedge$
THEN	**THEN**	$BB = 0 \wedge \ldots$
$RSum := RSum + AA$	$RSum := RSum + BB$	**THEN**
$\parallel AA := 0$	$\parallel BB := 0$	$Sum := RSum$
END ;	**END ;**	**END ;**

La clause **ASSERTIONS** assure que les nouveaux événements sont déclenchés et que le variant (modélisé par les variables de l'addition) garantit le déclenchement en séquence.

Choix : $T_0 ::= T_1[]T_2$**.** Considérons que la somme de aa et bb est réalisée en utilisant un choix non déterministe. Cette solution est obtenue par la sémantique du B événementiel qui permet de déclencher des événements de façon non déterministe.

Deux événements sont définis. Le premier Evt_1 calcule le résultat $RSum = AA + BB$ et le second Evt_2 calcule le résultat $RSum = BB + AA$. Ces deux événements du raffinement travaillent pour l'événement Evt_0 qui collecte les résultats et raffine l'événement Evt_0 de l'abstraction. Ici le variant est codé par une expression logique sur AA et BB (l'une est $AA \neq 0 \wedge BB \neq 0$ et l'autre est $AA = 0 \wedge BB = 0$). Dans le raffinement ces deux expressions caractérisent deux états.

Remarquons que le choix non déterministe est obtenu grâce à la présence de l'expression $AA \neq 0 \wedge BB \neq 0$ dans les deux événements Evt_1 et Evt_2.

REFINEMENT $RefChoiceTask_{T0}$
REFINE $Task\_Sum_{T0}$
INVARIANT
$(RSum + AA + BB) = (aa + bb) \wedge \ldots$
ASSERTIONS
$(AA = 0 \wedge (BB = 0 \vee BB \wedge aa + bb \in NAT) \vee \ldots$
VARIANT
$(AA + BB)$
INITIALISATION
$RSum := 0 \parallel Sum :\in NAT \parallel aa, AA :\in (aa \in NAT \wedge AA = aa) \parallel \ldots$

EVENTS

$Evt_0 =$	$Evt_1 =$	$Evt_2 =$
SELECT	**SELECT**	**SELECT**
$AA = 0 \wedge$	$AA \neq 0 \ \wedge BB \neq 0 \wedge \ldots$	$AA \neq 0 \wedge BB \neq 0 \wedge \ldots$
$BB = 0 \wedge \ldots$	**THEN**	**THEN**
THEN	$RSum := AA + BB \ \parallel$	$RSum := BB + AA \ \parallel$
$Sum := RSum$	$AA := 0 \parallel BB := 0$	$BB := 0 \parallel AA := 0$
END ;	**END ;**	**END ;**
	END ;	**END ;**

Tâche itérative : T^*. Considérons la somme de deux nombres aa et bb obtenue au moyen d'une boucle. Le principe est d'affecter en premier $RSum := AA$ et d'ajouter, BB fois, la valeur 1 à $RSum$.

REFINEMENT $RefIterativeTask_{T0}$
REFINE $Task\_Sum_{T0}$
INVARIANT
$(RSum + AA + BB) = (aa + bb) \wedge \ldots$
ASSERTIONS
$(AA = 0 \wedge BB = 0 \wedge aa + bb \in NAT) \vee \ldots$
VARIANT
$(AA + BB)$
INITIALISATION
$RSum := 0 \parallel Sum :\in NAT \parallel aa, AA :\in (aa \in NAT \wedge AA = aa) \parallel \ldots$

EVENTS

$Evt_0 =$	$Evt_{InitLoop} =$
SELECT	**SELECT**
$AA = 0 \wedge BB = 0 \wedge \ldots$	$AA \neq 0 \wedge BB \neq 0 \wedge \ldots$
THEN	**THEN**
$Sum := RSum$	$RSum := RSum + AA \parallel AA := 0$
END ;	**END ;**

$Evt_{Loop1} =$
SELECT
$AA = 0 \wedge BB \neq 0 \wedge \ldots$
THEN
$RSum := RSum + 1 \parallel BB := BB - 1$
END ;

Cette solution utilise trois événements. L'événement Evt_{Loop1} décrémente la variable BB jusqu'à ce que sa valeur soit 0. Quand le variant est égal à 0, Evt_0 est déclenché de façon à donner la main à l'abstraction.

Concurrence : $T_0 ::= T_1||T_2$**.** Pour le même exemple, nous avons imaginé un raffinement avec un opérateur de concurrence. Deux événements en concurrence sont définis. Evt_1 ajoute la valeur de AA à $RSum$ alors que l'événement Evt_2 ajoute BB à $RSum$. Quand ces deux événements sont déclenchés, le dernier événement Evt_0 est déclenché pour respecter l'abstraction ($Sum := RSum$). L'abstraction ne reprend la main que si le traitement en parallèle (par entrelacement) est terminé.

REFINEMENT $RefConcurrencyTask_{T0}$
REFINE $Task\_Sum_{T0}$
INVARIANT
$(RSum + AA + BB) = (aa + bb) \wedge \ldots$
ASSERTIONS
$(AA = 0 \wedge BB = 0 \wedge aa + bb \in NAT) \vee \ldots$
VARIANT
$(AA + BB)$
INITIALISATION
$RSum := 0 \parallel Sum :\in NAT \parallel aa, AA :\in (aa \in NAT \wedge AA = aa) \parallel \ldots$

EVENTS

$Evt_0 =$	$Evt_1 =$
SELECT	**SELECT**
$AA = 0 \wedge BB = 0 \wedge \ldots$	$AA \neq 0 \wedge \ldots$
THEN	**THEN**
$Sum := RSum$	$RSum := RSum + AA \parallel AA := 0$
END ;	**END ;**

$Evt_2 =$
SELECT
$BB \neq 0 \wedge \ldots$
THEN
$RSum := RSum + BB \parallel BB := 0$
END ;

Traduction des autres opérateurs CTT. Les opérateurs basiques tels que ($>>$, $||$, [ ] and $*$) ont été complètement représentés en B événementiel. Nous complétons la grammaire CTT (*ordre indépendant*, *tâche optionnelle*, *désactivation*, *interruption*) en se basant sur les résultats précédents.

Ordre indépendant. Considérons $T_0 ::= T_1 \models| T_2$. Dans une sémantique à base de traces, l'ordre indépendant est interprété par :

$$T_0 ::= T_1 \models| T_2 \text{ est traduite en } T_0 ::= (T_1 >> T_2)[](T_2 >> T_1)$$

Les opérateurs basiques *activation* et *choix* sont utilisés pour cette traduction. Ils définissent des traces entrelacées permettant le codage de l'opérateur *ordre indépendant*.

Tâche optionnelle. Considérons $T_0 ::= [T_1]$. La tâche optionnelle indique que la tâche T_1 peut être accomplie. Dans ce cas, nous introduisons la tâche atomique vide, nommée T_{Skip}. Ainsi, la caractéristique optionnelle est traduite par le choix entre la tâche T_1 ou la tâche vide. Nous obtenons :

$$T_0 ::= [T_1] \text{ est traduite en } T_0 ::= T_1 [] T_{Skip}$$

Désactivation. Considérons $T_0 ::= T_1 [> T_2$ où T_1 est désactivée par T_2. La désactivation considère deux cas : soit T_1 est atomique (i.e. ne peut pas être raffinée ou décomposée) soit elle ne l'est pas.

En effet, si T_1 est une tâche atomique, cela signifie que soit T_1 est accomplie ou soit T_2 est accomplie. Nous abtenons une traduction par un choix :

$$T_0 ::= T_1 [> T_2 \text{ est traduite en } T_0 ::= T_1 [] T_2$$

Quand T_1 n'est pas une tâche atomique (i.e. cela implique que les opérateurs CTT définissent des états observables), la désactivation peut se produire dans la trace définie par T_1. Si T_1 est décomposée en $T_{1,1}\ op_1\ T_{1,2}\ op_2 \cdots\ op_n\ T_{1,n+1}$, alors la traduction de l'opérateur de désactivation est définie par le choix de toutes les traces possibles qui découlent de la désactivation i.e. la désactivation peut se produire à chaque état observable de la décomposition T_1. Nous obtenons la traduction suivante :

$$T_0 ::= T_{1,1}\ op_1\ T_{1,2}\ op_2 \cdots\ op_n\ T_{1,n+1} [> T_2 \text{ est traduite en}$$

$$\begin{array}{l} T_0 ::= T_2\ [] \\ T_{1,1}\ >>\ T_2\ [] \\ T_{1,1}\ op_1\ T_{1,2}\ >>\ T_2\ [] \\ T_{1,1}\ op_1\ T_{1,2} \cdots op_i\ T_{1,i+1}\ >>\ T_2\ [] \\ \cdots\ >>\ T_2\ [] \\ T_{1,1}\ op_1\ T_{1,2} \cdots op_n\ T_{1,n+1} \end{array}$$

Désactivation d'une tâche infinie. Considérons $T_0 ::= T_1^* [> T_2$. Dans ce cas, la traduction utilise le même raisonnement que celle proposée précédemment concernant la désactivation. Nous obtenons :

$$T_0 ::= T_1^* [> T_2 \text{ est traduite en}$$

$$T_0 ::= (T_1)^N\ [>\ T_2, \text{ où } N \text{ est un nombre naturel arbitraire.}$$

Enfin, si la tâche T_1 n'est pas atomique, alors la désactivation peut se produire à chaque état observable de la décomposition T_1 comme définie dans la section 3.2.3.

Interruption. Considérons $T_0 ::= T_1 |> T_2$. La tâche T_1 est interrompue par la tâche T_2. Comme pour l'opérateur de désactivation, la traduction de l'interruption considère deux cas.

Si T_1 est une tâche atomique (ne comportant pas d'opérateur CTT), cela signifie que T_2 peut être déclenchée un nombre arbitraire de fois (peut être 0) et ensuite T_1 est activée. Dans ce cas, nous obtenons :

$$T_0 ::= T_1 |> T_2 \text{ est traduite en } T_0 ::= T_2^N >> T_1$$

Quand T_1 n'est pas une tâche atomique (comportant des opérateurs CTT) l'interruption peut se produire un nombre de fois à chaque état observable de la trace définie par T_1. Si T_1 est écrite comme $T_{1,1}\ op_1\ T_{1,2}\ op_2 \cdots\ op_n\ T_{1,n+1}$, alors la traduction de l'interruption est définie par le choix de toutes les traces possibles de l'interruption. Nous obtenons :

$$T_0 ::= T_{1,1}\ op_1\ T_{1,2}\ op_2 \cdots\ op_n\ T_{1,n+1} |> T_2 \text{ est traduite en}$$
$$T_0 ::= T_2^{N_1} >> T_{1,1}\ >>\ T_2^{N_2}\ op_1\ T_{1,2} \cdots >> T_2^{N_{n+1}}\ op_n\ T_{1,n+1}$$

Ici N_i sont des nombres naturels arbitraires montrant que la tâche T_2 associée à l'interruption peut se produire zéro ou plusieurs fois. Il est à noter que nous avons choisi d'interpréter l'opérateur d'interruption par cette approche. Nous aurions pu choisir d'activer l'interruption une seule fois ($N_i = 1$).

Application à une étude de cas : le convertisseur francs/euros et le compteur

Sur la figure 3.5 nous représentons le modèle de tâches CTT que nous formalisons. *Conversion franc/euro* modélise la tâche liée à l'application du convertisseur. Elle décrit le fait que l'utilisateur peut saisir une valeur en continu (tâche itérative) jusqu'à ce qu'il choisisse le sens de conversion (en euro ou en franc).

Une fois le choix de conversion effectué, la valeur convertie est affichée (tâche *Valeur Convertie*). Après trois conversions (trois itérations sur la tâche *Conversion franc/euro*), la tâche liée à l'application compteur est rendue activable. Cette tâche se décompose en deux sous-tâches. La première décrit le fait que l'utilisateur doit agir sur le bouton d'initialisation. La seconde (*Mise à jour*) décrit la mise à jour de l'affichage des présentations après la demande d'initialisation.

Une fois la mise à jour effectuée, l'utilisateur peut de nouveau convertir une somme en franc ou en euro. A tout moment du déroulement de la tâche *Application*, l'utilisateur peut la désactiver et quitter l'étude de cas.

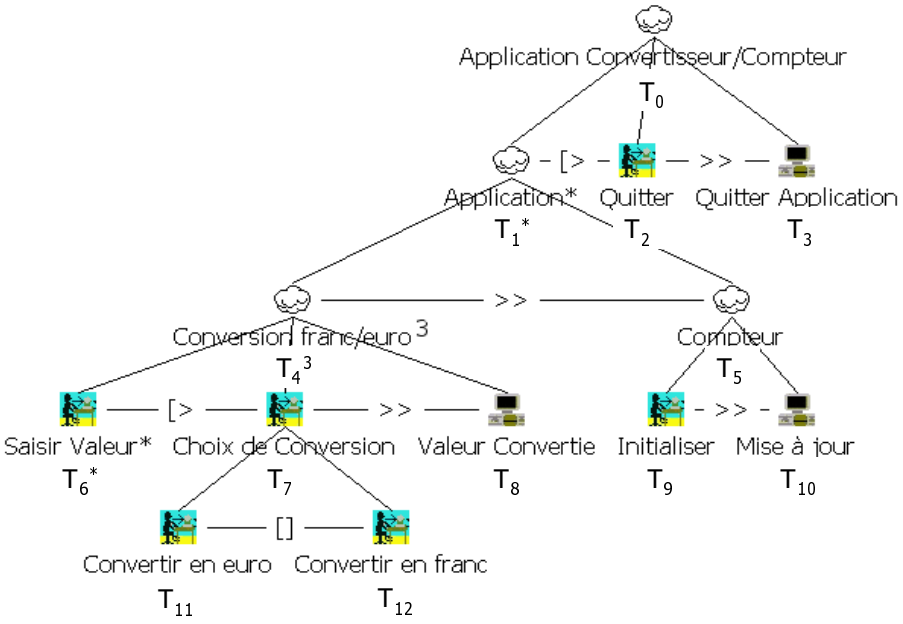


FIG. 3.5 – Modèle de tâches CTT de l'application convertisseur francs/euros et compteur.

Nous suivons une modélisation par niveaux de tâches. La décomposition hiérarchique de la tâche T_0 peut être décrite comme :

$$T_0 = T_1^*[> T_2 >> T_3$$
$$T_1 = T_4^3 >> T_5$$
$$T_4 = T_6^*[> T_7 >> T_8$$
$$T_5 = T_9 >> T_{10}$$
$$T_7 = T_{11}[]T_{12}$$

Quatre niveaux de raffinements qui correspondent chacun à des niveaux de l'arbre du modèle de tâches sont nécessaires. Le premier modèle décrit la tâche T_0. Le premier raffinement décompose cette tâche en $T_1^*[> T_2 >> T_3$, le deuxième raffinement décrit la décomposition de T_1^* par $T_4^3 >> T_5$ puis le troisième raffinement décrit la décomposition de T_4^3 et de T_5. Finalement, le dernier raffinement décrit la décomposition de T_7 par $T_{11}[]T_{12}$.

Même si cet exemple est simple, il a permis de montrer la faisabilité des règles de traduction pour valider des propriétés de validité :

– l'utilisateur ne peut choisir le sens de conversion avant de saisir une valeur ;
– l'utilisateur peut saisir plusieurs valeurs avant d'effectuer sa conversion ;

- au bout de trois conversions, l'utilisateur ne peut plus convertir et doit absolument initialiser le compteur ;
- à la suite d'une conversion, le champ de lecture est modifié et la valeur du compteur mise à jour.

La vérification a porté, d'une part, sur l'ordonnancement des tâches (atteignabilité, tâche non blocante, etc) et d'autre part, sur les contraintes qui garantissent par exemple qu'une postcondition d'une tâche mère est conforme à la postcondition de ses tâches filles.

3.2.4 Bilan sur l'approche à base d'événements

Nous avons présenté dans cette section l'approche à base d'événements qui se fonde sur l'approche à base de modules et qui permet la modélisation des modules boîtes à outils, présentation, adaptateur du domaine et domaine de l'architecture ARCH. Ces travaux se sont tout d'abord intéressés à la description du comportement du contrôleur de dialogue au moyen de systèmes de transitions. B événementiel a été utilisé pour coder les systèmes de transitions et l'opération de produit synchronisé grâce au raffinement pour représenter la décomposition, la méthode B a permis l'introduction de nouveaux événements dus à la décomposition d'automate. La manipulation des systèmes de transitions n'est pas explicite. Le concepteur ne décrit que les événements en donnant la condition d'apparition de l'événement (garde) et l'action résultante lorsque la garde est vraie.

Nous avons vu que la contribution apportée dans cette section a permis d'exprimer de nouvelles propriétés (propriétés de non blocage ou d'équité) par rapport à l'approche à base de modules et permettait ainsi la validation de la conception. Assurer l'utilisabilité d'une IHM nécessite aussi la vérification de propriétés de validité qui caractérisent un fonctionnement voulu par un utilisateur. L'approche implicite permet de représenter et de prendre en compte en amont des notations centrées utilisateurs (en l'occurrence CTT) dans la conception des IHM. Elle reprend aussi les mêmes avantages que la validation de tâches par traces d'opérations, c'est-à-dire la validation à priori des tâches et de la conception. S'ajoute à cela, l'homogénéité de l'approche qui permet de décrire à la fois la description du modèles de tâches et de la conception à l'aide de la même technique formelle B. De plus, elle évite l'énumération de traces de tâches qui alourdirait la mise en place de la phase de validation. Les scénarii et la validation se font de façon implicite sans que le concepteur ait une quelconque expertise sur les spécifications (à l'opposé de l'approche explicite expérimentée en premier).

3.3 Conclusion

Les travaux présentés dans ce chapitre ont été réalisés au LISI de l'ENSMA. Tous ces travaux s'appuient sur la méthode B et la preuve. Dans l'état actuel, ces travaux :

- n'ont abordé que les systèmes interactifs de type WIMP avec manipulation directe ;
- ne présentent pas de méthodologie permettant de mettre en œuvre des développements formels fondés sur B. Tous les développements présentés sont ad-hoc ;
- ne permettent pas l'expérimentation au sens des psychologues et des ergonomes. En effet, les test in-situ ne sont pas supportés ;

Cependant, ces travaux couvrent une grande partie du cycle de développement d'un système interactif y compris en intégrant la validation de tâches utilisateurs.

Chapitre 4

Conclusion : Applications aux IHM3

4.1 Résultats obtenus

L'application des approches formelles fondées sur la preuve aux développements de systèmes interactifs a été abordée avec deux techniques formelles : Z[DH93a, DH95a] B [Abr96b, Abr96a].

Les travaux avec Z ont été les précurseurs. Ils ont formalisé la notion d'interacteur, mais ne se sont intéressés qu'aux descriptions d'IHM. La vérification de propriétés, la validation de tâches ainsi que le raffinement n'ont été abordées que partiellement. Les travaux avec B, essentiellement menés au LISI, se sont interessés aux différentes phases de développement des systèmes interactifs : spécification, conception, validation de tâches utilisateurs, vérifications de certaines propriétés ergonomiques, animation de l'IHM, ... De plus, la version événementielle a permis de modéliser et de vérifier le contrôleur de dialogue et son comportement. C'est un composant essentiel de tout système interactif. Les développements fondés sur le raffinement et la décomposition ont été mis en œuvre. Enfin l'approche avec B a également permis la prise en compte de notations permettant d'évaluer l'utilisabilité d'une IHM. L'intégration de ces notations a été réalisée en donnant à ces notations une sémantique formelle avec la méthode B.

4.2 Insuffisances

Malgré les résultats obtenus et recensés précédemment, on peut noter les insuffisances suivantes :

– **limitation aux systèmes interactifs de type WIMP**. Tous les travaux abordés précédemment utilisant des approches fondées sur la preuve ont étu-

dié une catégorie particulière, mais très répandue, de systèmes interactifs : les systèmes interactifs de type WIMP (Windows Icons Menus Pointers). Ces travaux ne se sont pas intéressés aux autres types de systèmes interactifs, comme les systèmes interactifs Multi-Modaux, les systèmes interactifs ubiquitaires ou bien les systèmes interactifs mobiles ;

- **absence d'expérimentation**. Bien que des approches de tests et de génération de tests aient été proposées pour des techniques formelles comme B, ces approches ne permettent pas l'expérimentation in-situ indispensable à la validation des propriétés d'utilisabilité et d'ergonomie. A l'heure actuelle, la présentation n'est pas associée aux modèles formels décrits ;
- **absence de méthodologie**. Nous ne disposons pas à l'heure actuelle, d'un modèle de cycle de vie éprouvé, qui pourrait être utilisé pour le développement de systèmes interactifs en général. Plusieurs études de cas ont été réalisées avec B. Elles nous ont fourni des éléments méthodologiques qu'il faudrait intégrer dans une méthodologie à venir.

Bibliographie

[AA00] Yamine Aït Ameur. *Cooperation of formal methods in an engineering based software development process*, pages 136–155. 2000.

[AAB04] Yamine Aït-Ameur and Mickaël Baron. Bridging the gap between formal and experimental validation approaches in hci systems design : use of the event b proof based technique. In Department of Computer Science University of Cyprus, editor, *ISOLA 2004 - 1st International Symposium on Leveraging Applications of Formal Methods*, pages 74–81, Paphos, Cyprus, 2004.

[AAB05] Yamine Aït-Ameur and Mickaël Baron. Bridging the gap between formal and experimental validation approaches in hci systems design : use of the event b proof based technique (revue). *To appear in International Journal on Software Tools for Technology Transfer Special Section*, 2005.

[AABG03] Yamine Aït-Ameur, Mickaël Baron, and Patrick Girard. Formal validation of hci user tasks. In Al-Ani Ban, Arabnia H.R, and Mum Youngsong, editors, *The 2003 International Conference on Software Engineering Research and Practice - SERP 2003*, volume 2, pages 732–738, Las Vegas, Nevada USA, 2003. CSREA Press.

[AABK03] Yamine Aït-Ameur, Mickaël Baron, and Nadjet Kamel. Utilisation de techniques formelles dans la modélisation d'interfaces homme-machine. une expérience comparative entre b et promela/spin. In *6th International Symposium on Programming and Systems ISPS 2003*, pages 57–66, Algérie, 2003.

[AAG01] Y. Aït-Ameur and P. Girard. Specification, Design, Refinement and Implementation of Interactive Systems : the B Method. Technical report, LISI/ENSMA, March 2001.

[AAGJ98a] Yamine Aït-Ameur, Patrick Girard, and Francis Jambon. A uniform approach for the specification and design of interactive systems : the b

method. In Panos Markopoulos and Peter Johnson, editors, *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'98)*, volume Proceedings, pages 333–352, Abingdon, UK, 1998.

[AAGJ98b] Yamine Aït-Ameur, Patrick Girard, and Francis Jambon. A uniform approach for the specification and design of interactive systems : the b method. In Panos Markopoulos and Peter Johnson, editors, *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'98)*, volume Proceedings, pages 333–352, Abingdon, UK, 1998.

[AAGJ98c] Yamine Aït-Ameur, Patrick Girard, and Francis Jambon. Using the b formal approach for incremental specification design of interactive systems. In Stéphane Chatty and Prasun Dewan, editors, *Engineering for Human-Computer Interaction*, volume 22, pages 91–108. Kluwer Academic Publishers, 1998.

[Abr96a] J-R Abrial. *The B Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr96b] J-R Abrial. Extending b without changing it (for developing distributed systems). In H Habrias, editor, *First B Conference, Putting Into Pratice Methods and Tools for Information System Design*, page 21, Nantes, France, 1996.

[AWM95] G.D. Abowd, H-M. Wang, and A.F. Monk. A Formal Technique for Automated Dialogue Development. In G.M. Olsan and S. Schuon, editors, *Proceedings of DIS'95*, pages 219–226, 1995.

[BACL95] P. Bumbulis, P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena. Combining Formal Techniques and Prototyping in User Interface Construction and Verification. In P. Palanque and R. Bastide, editors, *2nd Workshop on Design, Specification and Verification of Interactive Systems DSVIS*, pages 174–192. Springer Verlag, 1995.

[Bar03] Mickaël Baron. *Vers une approche sûre du développement des Interfaces Homme-Machine (Thesis)*. Thèse de doctorat, Université de Poitiers, 2003.

[Bas92] Rémi Bastide. *Objets coopératifs : Un formalisme pour la modélisation des systèmes concurrents*. PhD thesis, Université de Toulouse 1, 1992.

[BFL+92] L. Bass, R. Faneuf, R. Little, N. Mayer, R. Pellegrino, S. Reed, R. Seacord, S. Sheppard, and M. Szczur. A metamodel for the runtime architecture of an interactive system. *SIGCHI Bulletin*, 24(1) :32–37, 1992.

[BP90] Rémi Bastide and Philippe Palanque. Petri net objects for the design, validation and prototyping of user-driven interfaces. In *3rd IFIP conference Interact'90*, pages 625–631, North-Holland, 1990.

[BPR+91] L. Bass, R. Pellegrino, S. Reed, S. Sheppard, and M. Szczur. The Arch Model : Seeheim Revisited. In *CHI 91 User Interface Developper's Workshop*, 1991.

[Bru97] P. Brun. XTL : a Temporal Logic for the Formal Development of Interactive Systems. In P. Palanque and F. Paterno, editors, *Formal Methods for Human-Computer Interaction*, pages 121–139. Springer-Verlag, 1997.

[Can03] Dominique Cansell. *Assistance au développement incrémental et à sa preuve*. Habilitation à diriger les recherches, Université Henri Poincaré, 2003.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, 1986.

[Cle97] ClearSy. Atelier b - version 3.5, 1997.

[Cou87] J. Coutaz. PAC an Implementation Model for Dialogue Design. In *Proceedings of INTERACT*, pages 431–437. North Holland, 1987.

[Cou90] J. Coutaz. *Interface Homme-Ordinateur, Conception et Réalisation*. Dunod Informatique- Paris, 1990.

[DFAB93] A.J. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice Hall, 1993.

[DH93a] D. Duke and M. D. Harrison. Abstract Interaction Objects. In *Proceedings of Eurographics conference and computer graphics forum*, volume 12, pages 25–36, 1993.

[DH93b] David J. Duke and Michael D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3) :25–36, 1993.

[DH95a] D. Duke and M. D. Harrison. Event Model of Human-System Interaction. *IEEE Software Engineering Journal*, 10(1) :3–10, 1995.

[DH95b] D Duke and M. D. Harrison. Event model of human-system interaction. *IEEE Software*, 1(10) :3–10, 1995.

[Dij76a] E.W. Dijkstra. In *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[Dij76b] E.W. Dijkstra. In *A Discipline of Programming*. Prentice-Hall Englewood Cliffs, 1976.

[Fek96] J.D. Fekete. *Un Modèle Multicouche pour la Construction d'Applications Graphiques Interactives*. PhD thesis, LRI-Orsay, 1996.

[Fou91] Open Software Foundation. *OSF/MOTIF Programmer's Guide*. Prentice Hall, 1991.

[FP90] Giorgio P. Faconti and Fabio Paternò. An approach to the formal specification of the components of an interaction. In Carlo E. Vandoni and David A. Duce, editors, *European Computer Graphics Conference and Exhibition*, pages 481–494, Montreux, Switzerland, 1990. Elsevier Science.

[Gau95] M.C Gaudel. Formal Specification Techniques for Interactive Systems. In P. Palanque and R. Bastide, editors, *2nd Workshop on Design, Specification and Verification of Interactive Systems DSVIS*, pages 21–26. Springer Verlag, 1995.

[GEM94] P. Gray, D. England, and S. McGowan. XUAN : Enhancing the UAN to Capture Temporal Relation Among Actions. Technical report, Department of Computing Science, University of Glasgow, 2 1994.

[GKS85] GKS. Graphical kernel system - functional description. Technical Report IS 7942, ISO, 1985.

[GP94] M. Gordon and A. Pitts. The HOL Logic and System. *Towards Verified Systems, Elsevier, Real-Time Safety Critical Systems*, 2, 1994. Condensed from Introduction to HOL by Gordon and Melham.

[Gui95a] L. Guittet. *Contribution à l'Ingénierie des Interfaces Homme-Machine. Théorie des Interacteurs et Architecture H4 dans le Système NODAOO*. PhD thesis, LISI/ENSMA, Poitiers, July 1995.

[Gui95b] Laurent Guittet. *Contribution à l'Ingénierie des Interfaces Homme-Machine - Théorie des Interacteurs et Architecture H4 dans le système NODAOO*. Doctorat d'université (phd thesis), Université de Poitiers, 1995.

[Hal91] Prentice Hall. *OSF/Motif Programmer's Guide*. 1991.

[Har87] D. Harel. Statecharts : a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3) :231–274, 1987.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[HH93] D. Hix and H.R. Hartson. *Developping User Interfaces : Ensuring Usability Through Product and Process*. John Wiley & Sons, inc., Newyork, USA, 1993.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, 12(10) :576–583, 1969.

[HT90] M. Harrison and H. Thimbleby. *Formal Methods in Human-Computer Interaction*. Series on Human computer interaction. Cambridge University Press, 1990.

[Jac82] R.J.K. Jacob. Using Formal Specification in the Design of Human-Computer Interface. In *Human factors in computing systems*, pages 315–321, 1982.

[JBAA01] Francis Jambon, Philippe Brun, and Yamine Aït-Ameur. *Spécifications des systèmes interactifs (chapitre 6)*, volume 1 of *Interaction homme-machine pour les S.I.*, pages 175–206. Hermès Science, Paris, France, 2001.

[LS96] F. Lonczewski and S. Schreiber. The FUSE-System : an Integrated User Interface Design Environment. In Jean Vanderdonck, editor, *CADUI'96*, pages 37–56, 1996.

[Mar86] L. S. Marshall. *A Formal Description Method for User Interfaces*. PhD thesis, University of Manchester, 1986.

[Mar97] Panagiotis Markopoulos. *A compositional model for the formal specification of user interface software*. Phd thesis, University of London, 1997.

[Mil92] K. Mc Millan. The SMV System. Technical report, Carnegie Mellon University, 1992.

[Nav01] David Navarre. *Contribution à l'ingénierie en Interaction Homme-Machine*. Doctorat d'université (phd thesis), Université Toulouse 3, 2001.

[Nie93] R. Nielsen. *Usability Engineering*. Academic Press, 1993.

[Pal92] P. Palanque. *Modélisation par Objets Coopératifs Interactifs d'Interfaces Homme-Machine Dirigées par l'Utilisateur*. PhD thesis, LIS, Université de Toulouse I, 1992.

[Pat01] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2001.

[PBS95] Philippe Palanque, Rémi Bastide, and Valérie Sengès. Validating interactive system design through the verification of formal task and system models. In Leonard J Bass and Claus Unger, editors, *IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI'95)*, pages 189–212, Grand Targhee Resort (Yellowstone Park), USA, 1995. Chapman & Hall.

[PF92a] Fabio Paternò and Giorgio P. Faconti. *On the LOTOS use to describe graphical interaction*, pages 155–173. Cambridge University Press, 1992.

[PF92b] F. Paterno and G. Faconti. On the LOTOS Use to Describe Graphical Interaction. In *Proceedings of HCI, People and Computer*, pages 155–173. Cambridge University Press, 1992.

[Pfa85] G.E. Pfaff. *User Interface Managment Systems*. Eurographics seminars. Springer Verlag - Berlin, 1985.

[PMG01] F Paternò, G Mori, and R Galimberti. Ctte : An environment for analysis and development of task models of cooperative applications. In *ACM CHI 2001*, volume 2, Seattle, 2001. ACM/SIGCHI.

[RH93] D. Rix and H.R. Hartson. *Developping User Interfaces : Ensuring Usability Through Product & Process*. Wiley professional computing. John Wiley & Sons, inc. NY, USA, 1993.

[Roc98a] P. Roche. *Modélisation et Validation d'Interfaces Homme-Machine*. PhD thesis, ENSAE, March 1998.

[Roc98b] Pierre Roche. *Modélisation et validation d'interface homme-machine*. Doctorat d'université (phd thesis), École Nationale Supérieure de l'Aéronautique et de l'Espace, 1998.

[Ses02] Irina Sessitskaia. *Apport des Techniques d'Abstraction pour la Vérification des Interfaces Homme-Machine*. Doctorat d'université (phd thesis), Ecole Nationale Supérieure de l'Aéronautique et de l'Espace (ENSAE), 2002.

[Shn98] B. Shneiderman. *Designing the User Interface*. Addison Wesley, 1998.

[SPG90] D. L. Scapin and C. Pierret-Golbreich. Towards a Method for Task Description : MAD. In *Work with display units*. Elsevier Science Publishers, North-Holland, 1990.

[Spi88] J M. Spivey. *The Z notation : A Reference Manual*. Prentice–Hall Int., 1988.

[SSC+95] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumara samy, and E. Salcher. Declarative Interface Models for User Interface Construction Tools : the MASTERMIND Approach. In *ECHCI'95*, pages 120–150. Chapmann and Hall, 1995.

[Sys84a] ISO Information Processing Systems. *Definition of the Temporal Ordering Specification Language lOTOS*. TC 97/16 N1987. ISO Genève, 1984.

[Sys84b] ISO Information Processing Systems. Definition of the temporal ordering specification language lotos. Technical Report TC 97/16 N1987, ISO, 1984.

[Was81] A. Waserman. User Software Engineering and the Design of Interactive Systems. In *5th IEEE International Conference on Software Engineering*, pages 387–393. IEEE Society Press, 1981.