ENSMA : Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
LISI : Laboratoire d'Informatique Scientifique et Industrielle

THESE

Pour l'obtention du Grade de

DOCTEUR DE L'UNIVERSITE DE POITIERS

ECOLE NATIONALE SUPERIEURE de MECANIQUE et d'AEROTECHNIQUE
et
FACULTE DES SCIENCES FONDAMENTALES ET APPLIQUEES
(Diplôme National — Arrêté du 30 mars 1992)

Ecole doctorale : **Sciences Pour l'Ingénieur**

Secteur de Recherche : INFORMATIQUE et APPLICATIONS

Présentée et soutenue publiquement
devant la Commission d'Examen le 15 décembre 2003

Mickaël BARON

Vers une approche sûre du développement des Interfaces Homme-Machine

Directeurs de thèse : ***Yamine Aït-Ameur et Patrick Girard***

JURY

Rapporteurs :	Dominique Cansell	MdC – HDR, LORIA, Université de Metz, Lorraine
	Bruno d'Ausbourg	Ingénieur de Recherche - HDR, ONERA-CERT, Toulouse
	Jean Vanderdonckt	Pr., BCHI, Université catholique de Louvain, Louvain
Examinateurs :	Yamine Aït-Ameur	Pr., LISI/ENSMA-UP, Université de Poitiers
	Patrick Girard	Pr., LISI/ENSMA-UP, Université de Poitiers
	Guy Pierra	Pr., LISI/ENSMA-UP, ENSMA, Poitiers
Invités :	Jean-Daniel Fekete	Chargé de recherches, INRIA, Orsay
	Francis Jambon	MdC, CLIPS-IMAG, Université J. Fourier, Grenoble

A toi papa

Merci à

Patrick Girard, mon directeur de thèse, pour son dévouement et la confiance qu'il m'a accordée pour la réalisation de cette thèse. Sa persévérance, son enthousiasme m'ont été d'un grand soutient dans les moments les plus difficiles.

Yamine Aït-Ameur, mon directeur de thèse, pour ses connaissances et l'enrichissement personnel qu'il m'a procuré en apportant une vision différente du monde de l'interaction homme-machine.

Dominique Cansell, **Bruno d'Ausbourg** et **Jean Vanderdonckt**, pour avoir accepté de rapporter mon travail de thèse et également pour leur indulgence quant à la remise un peu tardive du manuscrit.

Jean-Daniel Fekete pour sa présence et également **Francis Jambon** ancien collègue de bureau avec lequel j'ai passé d'agréables moments durant la réalisation de cette première partie de thèse.

Guy Pierra, Directeur du LISI, pour m'avoir accueilli au sein du laboratoire. **Claudine Rault**, secrétaire du LISI, pour sa sympathie et la réalisation des lourdes tâches administratives. Enfin, tous les autres membres du laboratoire et plus particulièrement à **Boulou**, **Christophe**, **Dago**, **David**, **Eric**, **Faby**, **Fred C**, **Fred R**, **Guibguib**, **Hondjack**, **JCP**, **Jéjé**, **Ladjel**, **Lolo**, **Laurent**, **Manu**, **Micky**, **Mourad**, **Pascal** et **Tex** pour tous les bons moments passés en leur compagnie.

Ma famille, ma **mère**, mon **père**, mon frère **Christophe**, mon frère **Vincent**, **Karine** et **Mathilde** pour leur soutient permanent.

Ma belle famille, **Cathy**, **Didier**, **Christelle** et **Elodie** pour leur gentillesse.

Mes amis, et notamment **Jean-Marc**, **Marie-Ange**, **Damien**, **Christelle** et **Erika**.

Enfin, **Manoue**, pour sa compréhension, ses sacrifices et pour avoir partagé avec moi cette longue épreuve.

Résumé

Les interfaces homme-machine (IHM) constituent une part indispensable dans la quasi-totalité des systèmes informatiques. Le recours à des notations de description des IHM, et à des modèles de spécification, de développement, de vérification et de validation devient indispensable pour assurer que le système satisfait les propriétés définissant la notion d'utilisabilté.

Aujourd'hui, on peut considérer que deux approches exploitant les modèles du domaine de l'IHM peuvent être mises en parallèle pour la vérification de propriétés : les approches fondées sur le développement formel et les approches fondées sur la définition d'outils. Malgré des avancées intéressantes, aucune d'elles n'est encore parvenue à s'imposer.

Nous proposons dans cette thèse deux nouvelles approches permettant le développement sûr d'interfaces homme-machine, fondées sur une même méthode formelle (la méthode B).

La première fondée sur le développement formel permet, d'intégrer des notations et des techniques hétérogènes du domaine de l'IHM dans une seule et unique méthode formelle (la méthode B), afin d'exprimer, vérifier et valider des propriétés du système interactif.

La seconde, fondée sur la définition d'outils, (SUIDT), permet de concevoir de manière interactive le dialogue entre un noyau fonctionnel développé formellement en B et une présentation graphique de l'interface, tout en garantissant le respect des propriétés exprimées à la fois dans le noyau fonctionnel et au niveau des tâches de l'utilisateur.

Abstract

Human-Computer Interfaces (HCI) represent an essential part in most computing systems. Resorting to specification, development, checking, validation models and notations from interactive application's description is becoming necessary to ensure that the system perfectly meets the properties that define usability.

Nowadays, we consider that properties can be checked following two approaches : one based on formal developments and the second one is tool based. In spite of great progress, none of them emerged.

In this context, we propose two new approaches allowing a safe Human-Computer interface development, based on a single formal method (B method).

The first approach, based on formal developments, permits the integration of HCI notations and heterogeneous techniques into a single formal method (B method) in order to express, check and validate interactive system properties.

The second one, based on tools definitions (SUIDT) allows creating an interactive dialog between the formally-developed functional core using B and an user interface. Moreover, the latter approach ensures properties what are expressed both into the functional core and by the user's needs.

Table des matières

Remerciements **i**

Résumé **vi**

Abstract **viii**

Table des figures **xiv**

Liste des tableaux **xviii**

Introduction générale **1**

1 État de l'art **7**

1.1 Introduction 7
1.2 Le domaine de l'interaction homme-machine 8
1.2.1 Comprendre la logique de l'utilisateur 9
1.2.2 IHM et Génie Logiciel 11
1.2.3 Modèles de tâches 13
1.2.4 Modèles d'architecture 19
1.2.5 Modèles de description du dialogue 25
1.2.6 Propriétés dans les IHM 29
1.2.7 Bilan sur le domaine de l'interaction homme-machine 33
1.3 Les techniques formelles pour la conception des systèmes interactifs 34
1.3.1 Classification des techniques formelles 35
1.3.2 La nécessité de la formalisation : Insuffisances des notations semi-formelles dans la conception des IHM 37
1.3.3 Les techniques formelles pour la conception des IHM 38
1.3.4 Bilan sur les techniques formelles pour la conception des systèmes interactifs 50
1.4 Les outils de conception 50
1.4.1 Approche ascendante : boîte à outils, squelette d'applications et générateur d'interfaces 51
1.4.2 Approche descendante : Systèmes Basés sur Modèles 56

1.4.3 Bilan sur les outils de conception . . . 71
1.5 Synthèse : justification du travail de thèse . . . 72
1.5.1 Limites des techniques formelles et des outils de conception pour le développement d'interfaces homme-machine . . . 73
1.5.2 Justification du travail de thèse . . . 74
1.5.3 Etude de cas . . . 76

2 Modélisation et validation formelles de descriptions de l'interaction dans les IHM 79
2.1 Introduction . . . 79
2.2 Rappels . . . 81
2.2.1 Systèmes de transitions . . . 81
2.2.2 Logique de Dijkstra . . . 85
2.3 Méthode B . . . 87
2.3.1 Le langage B . . . 88
2.3.2 B événementiel . . . 92
2.3.3 Bilan sur la méthode B . . . 100
2.4 Approche à base de modules . . . 101
2.4.1 Approche du LISI . . . 101
2.4.2 Conception modulaire : application à l'étude de cas . . . 108
2.4.3 Validation de tâches par traces d'opérations : approche explicite . . . 114
2.4.4 Bilan sur l'approche à base de modules . . . 122
2.5 Approche à base d'événements . . . 124
2.5.1 Modélisation du contrôleur de dialogue à base d'événements . . . 124
2.5.2 Validation de tâches par modèle de tâches CTT : approche implicite 133
2.5.3 Bilan sur l'approche à base d'événements . . . 170
2.6 Conclusion . . . 171

3 SUIDT : Une approche expérimentale pour la construction d'interfaces utilisateurs sûres 173
3.1 Introduction . . . 173
3.2 Généralités sur l'approche expérimentale . . . 175
3.2.1 L'approche initiale : GenBUILD . . . 175
3.2.2 Présentation générale de l'approche SUIDT . . . 179
3.2.3 Bilan sur les généralités de l'approche expérimentale . . . 181
3.3 Intégration d'un noyau fonctionnel développé formellement dans une approche de système basés sur modèles . . . 181
3.3.1 Conception du noyau fonctionnel développé formellement . . . 183
3.3.2 Adaptateur du noyau fonctionnel développé formellement . . . 189
3.3.3 Bilan sur l'intégration d'un noyau fonctionnel développé formellement 193
3.4 Validation sur le noyau fonctionnel . . . 194

3.4.1 Description du modèle de tâches abstrait . . . 197
3.4.2 Outil d'édition de la structure du modèle de tâches abstrait . . . 201
3.4.3 Outil de liaison entre les objets du noyau fonctionnel et les tâches du modèle de tâches abstrait . . . 203
3.4.4 Outil de simulation du modèle de tâches abstrait . . . 206
3.4.5 Validation sur le noyau fonctionnel : bilan . . . 209
3.5 Validation sur la présentation . . . 210
3.5.1 Description du modèle de tâches concret . . . 213
3.5.2 Constructeur d'interfaces utilisateur . . . 219
3.5.3 Outils d'édition du modèle de tâches concret et de liaison entre les objets du noyau fonctionnel, de la présentation et des tâches . . . 221
3.5.4 Outil de simulation du modèle de tâches concret . . . 223
3.5.5 Validation sur la présentation : bilan . . . 226
3.6 Conclusion générale sur l'approche SUIDT . . . 227

Conclusion et perspectives **231**

Bibliographie **236**

Publications liées à ce mémoire de thèse **247**

A Présentation de la notation ConcurTaskTrees **249**
A.1 La sémantique de la notation CTT . . . 249
A.1.1 Les catégories des tâches . . . 250
A.1.2 Les objets des tâches . . . 250
A.1.3 Les opérateurs temporels . . . 251
A.1.4 Caractéristiques de tâches . . . 251
A.2 L'outil CTTE . . . 252
A.2.1 L'éditeur de modèle de tâches . . . 252
A.2.2 Le simulateur de modèle de tâches . . . 252

Table des figures

1.1 L'interface, un intermédiaire entre l'homme et la machine 7
1.2 Distances sémantiques et distances articulatoires. 10
1.3 Modèle de développement en V pour les systèmes interactifs. 11
1.4 Modèles intervenant dans le cycle de développement en V pour les systèmes interactifs. 12
1.5 *Faire du café*, un exemple en notation HTA. 14
1.6 *Faire du café*, un exemple en notation MAD. 16
1.7 *Distributeur automatique de billets*, un exemple en notation CTT. 18
1.8 Le modèle SEEHEIM. 21
1.9 Le modèle ARCH. 22
1.10 Le modèle MVC. 23
1.11 Agent PAC (a) et hiérarchie d'agent PAC (b). 24
1.12 *Copier/Coller une zone de texte*, un exemple d'un système de transitions. . 26
1.13 *Copier/Coller une zone de texte*, un exemple d'un réseau de Petri. 27
1.14 Processus de développement formel. 34
1.15 *Boîte blanche* de l'interacteur de CNUCE, adapté de [PF92]. 40
1.16 *Modélisation d'un scrollbar*, un exemple avec les interacteurs en boîte noire de CNUCE, adapté de [PF92]. 41
1.17 Schéma d'un interacteur de York. 42
1.18 Comportement d'un bouton. 43
1.19 Interacteur à *flots de données*. 45
1.20 Approche à la fois expérimentale et formelle de [Roc98]. 45
1.21 Le modèle d'architecture CAV *Control-Abstraction-View*. 48
1.22 Famille des outils de développement. 51
1.23 Abonnement d'un composant graphique Swing à un écouteur. 53
1.24 Exemple de GUI-Builder intégré dans l'environnement JBuilder. 56
1.25 Environnement de développement d'interfaces à base de modèles, adapté de [Sze96]. 58
1.26 Cycle de développement dans MOBI-D. 61
1.27 Architecture générique de l'environnement MOBI-D. 63
1.28 Rôle de la communication CORBA dans l'environnement MASTERMIND. 64

1.29 Architecture générique de l'environnement MASTERMIND. 65
1.30 Architecture générique de l'environnement TERESA. 67
1.31 Formulaire de modification des éléments contenus dans la spécification concrète d'un modèle de TERESA. 68
1.32 Architecture générique de l'environnement PETSHOP. 70
1.33 Présentation des interfaces de l'étude de cas : Convertisseur francs/euros et Compteur. 77

2.1 Exemple d'un automate à deux états. 83
2.2 Interface de l'application Post-It Notes® modélisée au moyen de la méthode B. 102
2.3 Relations entre machines abstraites B de la modélisation du Post-It Notes®.103
2.4 Relations entre machines abstraites B de la modélisation du convertisseur francs/euros (a) et du compteur (b). 109
2.5 Composition modulaire ascendante des sous-systèmes convertisseur francs-/euros compteur. 112
2.6 Description de l'approche explicite. 114
2.7 Décomposition de tâches en sous-tâches. 116
2.8 Décomposition en sous-tâches de la tâche *Trois conversions / Compteur*. . 118
2.9 Différents niveaux de décomposition de sous-systèmes. 125
2.10 Système de transitions étiquetées du convertisseur francs/euros. 126
2.11 Système de transitions étiqueté du compteur. 127
2.12 Transition de l'état *E6* à *E3* de l'automate du convertisseur francs/euros. . 128
2.13 Extrait du système de transitions obtenu par la composition des systèmes de transitions du convertisseur et du compteur. 129
2.14 Décomposition en sous-systèmes convertisseur francs/euros et compteur. . 132
2.15 Description de l'approche implicite. 134
2.16 Modèle de tâches CTT de l'application convertisseur francs/euros et compteur. 161

3.1 Capture d'écran de l'outil GenBUILD. 177
3.2 Architecture générique de l'environnement GenBUILD. 178
3.3 Description générale de l'approche SUIDT. 179
3.4 Démarche pour la prise en compte d'un noyau fonctionnel développé formellement 182
3.5 Conception du module Noyau Fonctionnel de l'architecture ARCH. 183
3.6 Conception du module Adaptateur de Noyau Fonctionnel de l'architecture ARCH. 190
3.7 Capture d'écran de l'outil Animateur de noyau fonctionnel formel. 192
3.8 Démarche pour la modélisation du modèle de tâches abstrait. 195
3.9 Conception « abstraite » du module Dialogue, Présentation et Boîte à outils de l'architecture ARCH. 196

3.10 Exemple de préconditions (gardes) de tâches du modèle de tâches abstrait. 198
3.11 Modèle de tâches abstrait du convertisseur/compteur. 200
3.12 Capture d'écran de l'éditeur de modèle de tâches abstrait. 202
3.13 Capture d'écran de l'outil de liaison entre les objets du noyau fonctionnel et les tâches du modèle de tâches abstrait. 203
3.14 Exemple d'expressions pour la précondition (garde) sur les objets de la tâche *Conversionfranc/euro*. 204
3.15 Exemple d'expressions pour l'action de la tâche interaction *Saisir Valeur*. 205
3.16 Exemple d'expressions pour l'action de la tâche application *Valeur Convertie*. 206
3.17 Capture d'écran de l'outil de simulation du modèle de tâches abstrait : Vue de l'enregistrement d'un scénario. 207
3.18 Capture d'écran de l'outil de simulation du modèle de tâches abstrait : Vue de la lecture d'un scénario. 208
3.19 Démarche pour la modélisation du modèle de tâches concret. 211
3.20 Concrétisation du module Dialogue, Présentation et Boîte à outils de l'architecture ARCH. 212
3.21 Extrait du modèle de tâches concret. 217
3.22 Capture d'écran de l'outil de construction d'interfaces utilisateur. 220
3.23 Capture d'écran de l'éditeur de modèle de tâches concret. 221
3.24 Capture d'écran des attributs de la présentation dans l'outil de liaison entre les objets du noyau fonctionnel et les tâches du modèle de tâches concret. . 222
3.25 Expressions pour l'action de la tâche interaction concrete *Zone Saisie*. . . 223
3.26 Expressions pour l'action de la tâche concrete rendue *ICouleur Euro*. . . . 223
3.27 Capture d'écran de l'outil de simulation du modèle de tâches concret : vue de l'enregistrement d'un scénario. 224
3.28 Architecture générique de l'environnement SUIDT. 228

A.1 Représentation graphiques des catégories de tâches CTT. 250
A.2 Capture d'écran de l'éditeur de modèles de tâches CTT. 253
A.3 Capture d'écran du simulateur de modèles de tâches CTT. 254

Liste des tableaux

1.1 *Ouvrir un fichier dans une application*, un exemple en UAN. 17

2.1 Machine abstraite B générique. 89
2.2 Machine abstraite B avec une opération. 90
2.3 Obligations de preuve d'une machine abstraite B. 90
2.4 Substitutions utilisées pour définir la forme d'une opération. 91
2.5 Substitutions utilisées pour définir le corps d'une opération. 91
2.6 Modèle B événementiel générique. 93
2.7 Obligations de preuve d'un modèle B événementiel. 96
2.8 Refinement B événementiel générique. 97
2.9 Raffinement B événementiel. 98
2.10 Obligation de preuve d'un raffinement en B événementiel. 100
2.11 Extrait des opérations contenues dans la machine abstraite *BControleur*. . 112
2.12 Décomposition en trace de tâches et d'opérations atomiques du contrôleur de dialogue. 116
2.13 Grammaire BNF de la notation de tâches ConcurTaskTrees. 135
2.14 Extrait des événements contenus dans le modèle *BControleurConvertisseurCompteur*. 162

3.1 Noyau fonctionnel générique B. 187

Introduction générale

L'étude des interactions entre l'homme et la machine[1] est éminemment pluridisciplinaire. Elle recouvre des domaines très différents, comme la psychologie ou l'ergonomie d'un côté, et l'informatique ou la physique de l'autre. Même lorsque les rencontres entre les communautés de chercheurs obtiennent un grand succès (la conférence ACM[2]-CHI[3] par exemple), l'éloignement des disciplines rend difficile les échanges directs. Cette diversité se traduit par une complexité naturelle de conception et de mise en œuvre des IHM qui se traduit par une part importante, dans les applications, du code dévolu à la programmation de l'IHM, augmentant ainsi le risque d'erreurs lors des étapes de conception. Selon une étude de [MR92], le volume du code des interfaces représente en moyenne 48% du code des applications et monopolise 50% du temps imparti au processus de développement.

Le recours à des modèles de spécification, de développement, de vérification et de validation, et à des notations de description des IHM devient indispensable pour pouvoir maîtriser la complexité évoquée précédemment et pour augmenter la flexibilité et la facilité d'utilisation des IHM, c'est-à-dire répondre au critère d'utilisabilité des IHM. Ce critère est une caractéristique particulière aux IHM souvent omise dans le développement de logiciels. De nombreux travaux dans le domaine de la psychologie expérimentale ont permis de proposer des modèles pour aider à comprendre le comportement de l'homme face à la machine. À partir de ces modèles, des règles permettant de définir et de mesurer l'utilisabilité des IHM ont été définies. Les modèles et théories tels que le modèle du processeur humain [CMN83] ou la théorie de l'action [Nor86] ont permis d'élaborer des recommandations (conception) et des critères (évaluation et validation). De plus, des notations, le plus souvent semi-formelles, ont permis de décrire des méthodes d'analyse et de description de l'activité (modèle de tâches), facilitant ainsi la conception et surtout la validation des applications interactives.

Cependant, la diversité des modèles et notations, ainsi que la distance entre ces dis-

[1] Nous utiliserons l'acronyme IHM de deux façons différentes : au singulier, il signifiera l'**Interaction Homme-Machine**, alors qu'au pluriel, il devra être interprété comme les **Interfaces Homme-Machine**

[2] Association for Computing Machinery

[3] Computer-human interaction

ciplines (psychologie, ergonomie, etc) et la programmation des systèmes informatiques, rendent extrêmement complexe, l'application de ces résultats au développement de ces mêmes systèmes.

Pourtant, du fait de l'utilisation des outils informatiques dans les systèmes complexes, il devient plus pressant de disposer de méthodes et d'outils permettant de donner des garanties de bout en bout sur la qualité des systèmes informatiques, fussent-ils interactifs. Les méthodes du génie logiciel, et plus particulièrement les méthodes formelles, ont permis d'apporter des solutions dans de nombreux domaines tels que les systèmes critiques (systèmes de sécurité, systèmes embarqués, etc.). Cependant, il faut bien admettre que ces méthodes ne s'appliquent aujourd'hui qu'à une portion de ces logiciels, souvent qualifiée de cœur du système critique. Les IHM sont ainsi souvent exclues, la garantie sur le fonctionnement du système étant généralement apportée au moyen de gardes sur les actions de l'utilisateur, empêchant l'environnement extérieur de perturber le bon déroulement du programme.

Par ailleurs, un point particulièrement important doit être considéré dans l'utilisation des méthodes formelles : les compétences requises pour développer formellement un noyau fonctionnel sont extrêmement éloignées de celles qui sont mises en œuvre pour réaliser la couche de présentation d'un logiciel et il est tout à fait illusoire d'imaginer un concepteur réaliser ces deux tâches. C'est la raison principale de l'absence de percée des méthodes formelles dans le domaine de l'IHM : les nombreux travaux actuels sur l'utilisation de méthodes formelles pour les IHM sont inaccessibles à des non-spécialistes, alors qu'une grande partie du temps passé au développement des IHM ne nécessite pas de telles compétences.

L'objectif général de notre travail consiste à améliorer l'utilisation des méthodes formelles dans le cadre de la conception et de la réalisation et de la validation des applications interactives. Pour cela, nous avons suivi deux axes en parallèle. D'une part, nous avons cherché à réaliser un pont entre les modèles et les notations de modélisation des tâches et ces mêmes méthodes. D'autre part, nous avons étudié la définition d'outils reposant sur des méthodes formelles, et destinés à des non-spécialistes de ces mêmes méthodes. Les deux approches s'appuient sur la même méthode formelle : la méthode B [Abr96a, Abr96b]

Aujourd'hui, on peut considérer que deux approches exploitant les notations et les modèles du domaine de l'IHM, peuvent être mises en parallèle pour la vérification de propriétés : les **approches fondées sur le développement formel** d'une part, et les **approches fondées sur la définition d'outils** d'autre part.

Les approches fondées sur le développement formel consistent à élaborer des modèles sur la base de langages formels durant la phase de spécification. La modélisation formelle

favorise l'expression, la vérification et la preuve *a priori* de propriétés sur les systèmes. L'utilisation des méthodes formelles dans le domaine de l'IHM a donné lieu à de nombreux travaux justifiant ouvrages et conférences dédiées à ce thème. Cependant, les travaux par développement formel n'interviennent qu'à différents niveaux de spécification du système interactif et pour la plupart ne s'intéressent qu'à la couche présentation ou au contrôle de dialogue des applications interactives. Peu d'approches fondées sur le développement formel ont permis d'effectuer réellement la validation de tâches, c'est-à-dire intégrer dans la modélisation formelle les besoins de l'utilisateur. Enfin, les approches fondées sur le développement formel restent trop souvent une histoire d'experts. Peu d'outils abordables par des non-spécialistes sont disponibles. Ceci empêche la participation de tous les acteurs normalement impliqués dans le processus de développement d'une application interactive.

Les approches fondées sur la définition d'outils font appel à des techniques expérimentales (langages, outils ou environnements) inscrites dans un processus de développement itératif. L'absence de formalisation des modèles oblige le concepteur à vérifier d'une part, que le système fonctionne correctement, et d'autre part, qu'il correspond aux attentes de l'utilisateur. Dans cette optique, la phase de validation passe par un ensemble de tests pour vérifier que le comportement et les sorties du système sont bien ceux attendus. Ces approches s'appuient pour la plupart sur des modèles à sémantique pauvre et ne permettent pas un développement incrémental ni l'expression et la vérification des propriétés. Peu d'approches expérimentales couvrent complètement le cycle de développement d'un système tout en gardant les liaisons actives entre les modèles de développements intermédiaire et le code final de l'application à développer. La phase de génération déconnecte généralement ces modèles du code final. En d'autres termes, sans la liaison entre le code final et les modèles intermédiaire l'utilisation de ces derniers pour la validation et la vérification ne garantit pas la préservation des propriétés pour le code final.

Le présent mémoire tente de répondre aux limites évoquées précédemment en favorisant le rapprochement de ces deux approches. En effet, nous pensons que l'utilisation conjointe de deux approches (formelle et outils) dans le cycle de développement permettrait d'améliorer considérablement la conception et le développement des systèmes interactifs, et plus particulièrement leur utilisabilité. D'une part, les approches formelles seraient à même de garantir au plus tôt que les propriétés du système sont respectées. D'autre part, les approches fondées sur la définition d'outils permettraient de représenter des descriptions en se basant sur ces mêmes approches, tout en étant abordables par des non-spécialistes.

Ainsi deux contributions ont été proposées.

1. **Approche de modélisation et validation formelles de description de l'interaction dans les IHM**. En partant d'une notation de description des besoins

utilisateurs et d'un modèle de structuration de l'architecture logicielle existants, nous avons représenté formellement, suivant la méthode B et son extension B événementiel, les différents éléments de ces notations. Dans un premier temps, nous avons réussi à modéliser le dialogue de systèmes interactifs par l'intermédiaire du B événementiel. Dans un second temps nous avons traduit formellement la sémantique d'une notation de description de besoins utilisateurs afin de pouvoir valider formellement des tâches utilisateurs. L'apport principal de cette approche est de pouvoir intégrer des notations et des techniques hétérogènes dans une seule et unique méthode formelle (la méthode B), afin d'exprimer, vérifier et valider des propriétés du système interactif en cours de développement.

2. **Approche expérimentale pour la construction d'interfaces utilisateurs sûres**. La démarche associée à l'outil SUIDT (Safe User Interface Development Tool) consiste à concevoir de manière interactive le dialogue entre un noyau fonctionnel existant et développé formellement, et une présentation graphique de l'interface, tout en garantissant le respect des propriétés exprimées et prouvées lors du développement formel du noyau fonctionnel, tant au niveau du noyau fonctionnel qu'au niveau des tâches de l'utilisateur. Le dialogue de l'application est basé sur une notation de description des tâches de l'utilisateur. Sa construction consiste à raffiner un modèle de tâches depuis un niveau noyau fonctionnel (appelé modèle de tâches abstrait) jusqu'à la spécification des interactions de l'utilisateur et des rendus du système (appelé modèle de tâches concret). Le dialogue homme-machine est ainsi conçu tout en respectant les contraintes de sûreté imposées par le noyau fonctionnel formel et par les besoins de l'utilisateur.

Le point commun entre les deux approches est l'utilisation conjointe de notations prenant en compte l'activité de l'utilisateur et d'une méthode formelle unique, la méthode B, garantissant de bout en bout les propriétés du système. Notons cependant que ce mémoire exclut la description d'une méthodologie permettant de décrire les interconnexions entre les différentes approches et la manière de les utiliser.

La présentation de ces deux approches a conduit à diviser l'organisation de ce mémoire en trois chapitres.

Un premier chapitre présente un « état-de-l'art » sur le développement de logiciels interactifs. Nous présentons tout d'abord les modèles et notations exploités dans le domaine de l'interaction homme-machine. Puis nous dressons un portrait de l'état actuel des travaux de recherche autour de l'utilisation de ces modèles en abordant les techniques formelles pour la conception des systèmes interactifs et les outils de conception. Dans une dernière partie, nous décrivons les motivations pour ce travail et nous présentons l'étude de cas du convertisseur francs/euros avec laquelle nous illustrons nos deux approches.

Un deuxième chapitre présente l'approche fondée sur le développement formel que nous avons mise en œuvre. Nous débutons par un rappel des techniques formelles utilisées dans le cadre de l'IHM, puis nous présentons la méthode B en détaillant brièvement sa syntaxe et sa technique de preuve. Ensuite, nous décrivons par l'exemple l'approche du LISI sur laquelle nous nous sommes basés. La description de nos travaux s'effectue en deux parties. Tout d'abord, nous expliquons notre méthode de validation des tâches par traces de tâches. Ensuite, nous exposons la manière de représenter un système complexe par décomposition en sous-systèmes, en donnant des règles de transformation pour coder une notation semi-formelle de description des besoins utilisateurs dans la technique formelle B.

Un troisième chapitre présente l'approche par outils. Nous commençons par décrire l'approche initiale (GenBUILD) sur laquelle nous avons expérimenté l'intégration du noyau fonctionnel existant dans un outil de développement, dans le respect de ses propriétés. À partir de cette approche, nous avons déterminé les points qui devaient être approfondis et avec lesquels nous avons défini une approche plus globale, SUIDT. Ensuite, nous détaillons chaque partie du modèle qui compose cette approche en donnant à chaque fois la sémantique et les outils associés.

Enfin, nous concluons avec les résultats probants des deux approches et nous annonçons les perspectives de ce travail.

Chapitre 1

État de l'art

1.1 Introduction

Les interfaces homme machine (IHM) se retrouvent dans la quasi-totalité des systèmes informatiques ; ainsi, selon [Mye95] 97% des applications possèdent une interface utilisateur. Dans plusieurs cas, ces IHM constituent le seul point d'entrée de l'utilisateur dans ces systèmes. Elles ont pour but d'établir un lien entre les fonctionnalités brutes du système (la sémantique de l'application, regroupée dans un composant appelé noyau fonctionnel) et l'utilisateur. L'IHM agit donc comme un élément médiateur, voir figure 1.1. Un système interactif est donc vu comme la combinaison d'une IHM et de fonctionnalités brutes de ce système.

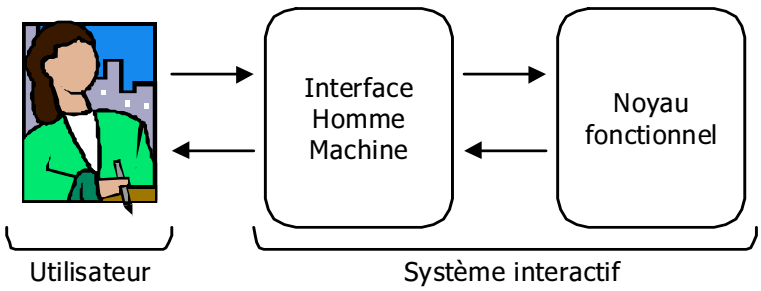

FIG. 1.1 – L'interface, un intermédiaire entre l'homme et la machine

Au cours des vingt dernières années, les interfaces utilisateurs sont devenues de plus en plus complexes. S'il existe toujours des interfaces de type formulaire où l'interaction se limite au déplacement dans un masque de saisie par l'intermédiaire de touches « fonctions », on trouve couramment aujourd'hui des interfaces qui utilisent des interactions

plus évoluées (manipulation directe, utilisation de la voix, du geste...) rendant ces interfaces de plus en plus complexes. La prise en compte de cette complexité se traduit par une part importante du code dévolue à la programmation de l'IHM et donc un effort de développement également important. En conséquence le risque d'erreurs qui pourraient être introduites lors des étapes de conception augmente. Selon [MR92], le développement lié à l'interface représente 48% du code d'une application et monopolise 50% du temps imparti au processus de développement et 37% pour la phase de maintenance.

Afin de prendre en compte cette complexité croissante et par conséquent d'assurer l'utilisabilité des interfaces homme-machine, le recours à des modèles et des notations est rendu nécessaire. Nous proposons d'explorer dans un premier temps, dans ce chapitre, une introduction aux domaines des interfaces homme-machine afin de présenter ces modèles et notations. Puis, dans les deuxième et troisième sections, nous dressons un portrait de l'état actuel des travaux de recherche autour de l'utilisation de ces modèles. Plus précisément nous abordons :

- les techniques formelles pour la conception des systèmes interactifs (section 1.3) ;
- les outils de conception (section 1.4).

Enfin, nous justifions notre contribution et son apport.

1.2 Le domaine de l'interaction homme-machine

Le domaine de l'interaction homme-machine est bien plus qu'une préoccupation d'interface. L'interface homme-machine doit satisfaire à des critères de qualité, par exemple celui d'utilisabilité du système interactif, c'est-à-dire être en adéquation avec les besoins et les capacités de l'utilisateur, afin de permettre à cet utilisateur d'atteindre ses objectifs à travers des trajectoires d'interactions intuitives et sûres. Dans le but d'assurer ces critères de qualité, 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.

Pour ce faire, la conception d'une IHM nécessite la collaboration de spécialistes issus de différents métiers. Elle fait intervenir non seulement des informaticiens pour la concep-

tion et le développement mais aussi des ergonomes pour faciliter son utilisation, et des pyschologues pour capturer les comportements des utilisateurs. Cependant, cette pluridisciplinarité contribue également à l'accroissement de la complexité du développement par l'allongement du processus de développement avec des phases de tests devant prendre en compte ces disciplines.

Le recours à des modèles et à des méthodes s'avère indispensable afin d'assurer l'utilisabilité du système interactif. Ces modèles et méthodes proviennent des différentes communautés citées précédemment (informaticiens, ergonomes, psychologues...) et constituent aujourd'hui un champ de recherche très actif.

Dans le but de mieux les cerner, nous nous intéressons dans une première partie à l'étude du comportement de l'utilisateur par le biais de modèles puis nous découvrons dans une deuxième partie comment l'utilisabilité est prise en compte dans le processus de développement d'une IHM. Ensuite, nous en donnons quelques exemples en présentant dans une troisième partie les modèles de tâches, dans une quatrième partie les modèles d'architecture et enfin les modèles de description du dialogue.

1.2.1 Comprendre la logique de l'utilisateur

Différents modèles sont utilisés pour comprendre la logique de l'utilisateur et prévoir et interpréter les difficultés d'interaction.

Dans le modèle du Processeur Humain, [CMN83] représentent le sujet humain comme un système multiprocesseur (systèmes sensoriel, moteur et cognitif) de traitement de l'information régi par des règles. L'avantage de ce modèle est de pouvoir présenter une image simplifiée de la structure mentale de l'utilisateur, mais en contrepartie il ne fournit pas les informations intéressantes pour la conception d'interfaces homme-machine.

GOMS [CMN83] (Goal, Operator, Method et Selection) est un modèle de description du comportement à différents niveaux d'abstraction, depuis la tâche (qui décrit une activité de haut niveau) jusqu'aux actions physiques (qui composent la tâche). GOMS permet aussi de prédire le temps d'exécution de ces actions. Sa version simplifiée nommée Keystroke [CMN83] concerne les aspects syntaxiques et lexicaux de l'interaction. Outre la facilité de décrire de façon minimale les actions de l'utilisateur sur l'interface, ce type de modèle ne prend pas en compte les erreurs, le contexte et l'apprentissage.

Si GOMS et Keystroke modélisent un comportement observé, la théorie de l'action [Nor86] modélise les processus psychologiques qui conduisent à ce comportement. La théorie de l'action associe la réalisation d'une tâche au parcours d'une distance. Cette dernière

exprime la différence entre la représentation de l'interface et celle maintenue dans l'idée que se fait l'utilisateur [Cou90]. Les distances sémantiques et articulatoires traduisent la difficulté à établir la correspondance entre les buts de l'utilisateur et les commandes disponibles du système. Elle traduit également la difficulté à interpréter l'état du système à partir des grandeurs exprimées par le système (voir figure 1.2). L'objectif du concepteur est d'en diminuer la longueur par l'intermédiaire de l'interface du système interactif (utilisation de métaphores d'interaction, placement des objets d'interaction...). Cette théorie complète les modèles précédents en précisant la notion d'état (effectif et perçu) et souligne certaines difficultés que peut rencontrer l'utilisateur.

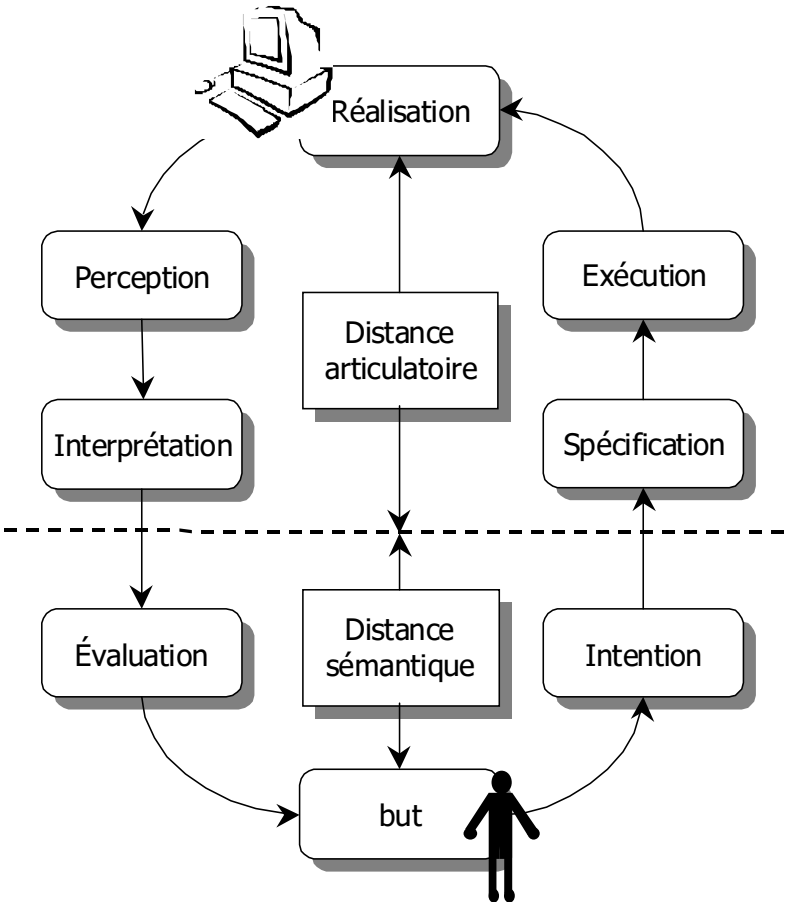

Fig. 1.2 – Distances sémantiques et distances articulatoires.

Ces modèles permettent ainsi d'adapter l'interface du système vis-à-vis de l'analyse du comportement de l'utilisateur que ces modèles fournissent. Les quelques modèles que nous avons présentés sont issu des domaines très riches de l'ergonomie cognitive et psycho-ergonomique, mais nous focaliserons plus spécialement notre étude sur un domaine plus technique : la programmation des interfaces homme-machine.

1.2.2 IHM et Génie Logiciel

Afin d'assurer le critère d'utilisabilité d'un système interactif, des méthodes issues du génie logiciel ont été utilisées et adaptées au domaine de l'IHM. De manière générale, les cycles de développement classiques du génie logiciel s'intéressent principalement à deux critères :

- assurer la **faisabilité** associée à la difficulté et au travail nécessaire pour la réalisation d'un système ;
- assurer la **qualité** liée à la correction, la sûreté et la maintenance du produit.

Ces cycles de développement ne se focalisent que très peu sur les relations entre l'utilisateur et le système interactif.

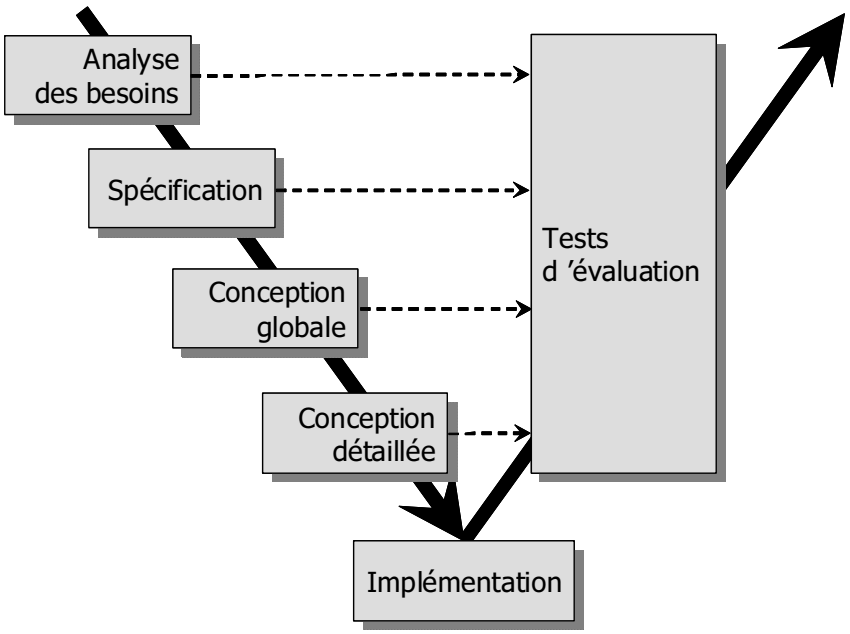

Fig. 1.3 – Modèle de développement en V pour les systèmes interactifs.

Dans le cas du modèle en cascade [Win70] cité dans [Boe81], on peut distinguer globalement quatre phases : l'analyse des besoins, la spécification, la conception du système et les tests du logiciel. Ce modèle définit une réalisation séquentielle des phases avec la possibilité de revenir sur les phases immédiatement antérieures. Ceci a pour conséquence de ne pas favoriser de modifications et tend à stabiliser rapidement le système. La modélisation de l'utilisateur n'est pas préconisée, hormis peut-être lors de la première étape, selon le bon vouloir des concepteurs. L'aspect utilisateur n'intervient réellement que dans la phase de tests du logiciel achevé.

En revanche, le modèle en V [MR84], qui est l'un des cycles de développement les plus répandus dans la production de logiciel vise à combler cette lacune (voir figure 1.3). La pente descendante du V reprend les structures du modèle en cascade (spécification, conception et codage). La pente ascendante correspond à un ensemble vérifications et de tests qui permettent de valider une phase dans la pente descendante. Ce processus permet un gain de temps sur les phases de spécification et de conception.

De nombreux modèles décrivent les cycles de développement et sont particulièrement adaptés au processus de développement des IHM : le modèle en spirale qui introduit un processus itératif [Boe88], le modèle en couches issu de [CH94] ou le modèle en étoile de [HH89]. Mais à la lumière des études de [Bal94] et [Nig94], ensuite reprises par [Tab01] sur le développement des IHM, nous nous limiterons au modèle en V comme support pédagogique pour la description des différents modèles, notations et outils qui servent à l'analyse, la spécification et la conception des systèmes interactifs.

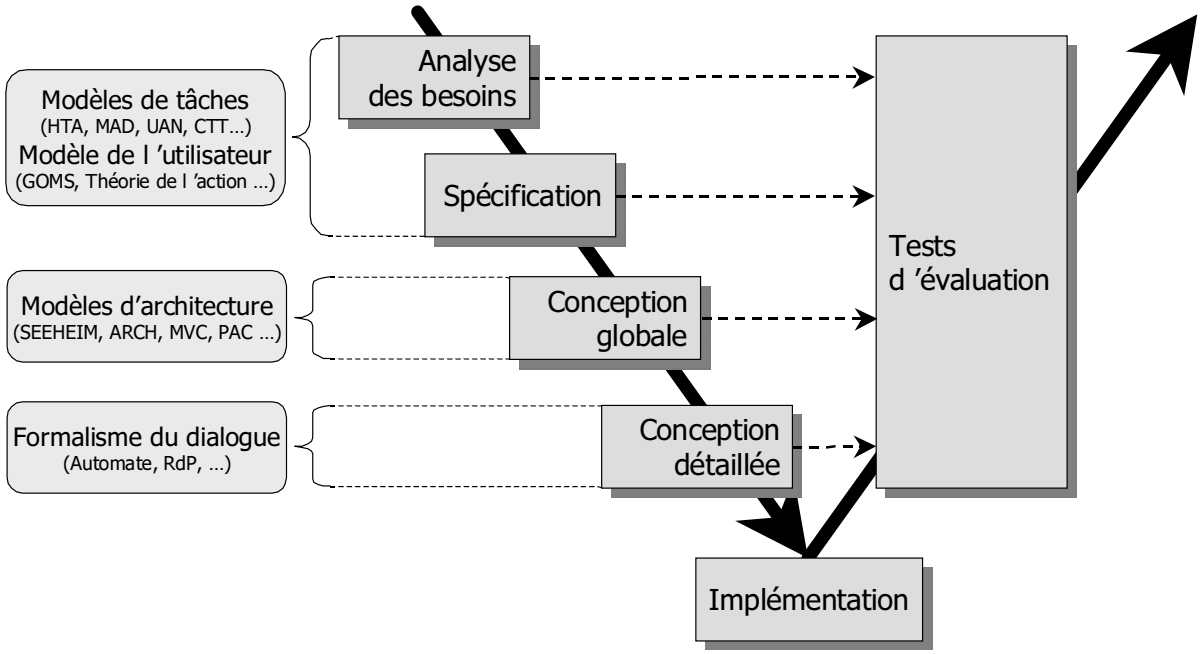

Fig. 1.4 – Modèles intervenant dans le cycle de développement en V pour les systèmes interactifs.

Nous réaliserons dans un premier temps une étude descendante des modèles de base qui interviennent dans le cycle de développement en V d'un système interactif (figure 1.4) :

- les modèles de tâches (analyse de tâches, description de l'interaction) ;
- les modèles d'architecture ;
- les modèles opérationnels de description du dialogue.

Chaque phase de développement de la partie descendante du modèle en V est accompagnée d'une phase de modélisation et d'une phase de validation. De façon générale, le processus de validation doit permettre d'établir les propriétés des interfaces homme-machine. C'est donc dans un deuxième temps que nous étudierons les propriétés des interfaces homme-machine nécessaires pour assurer l'utilisabilité des interfaces.

Dans cette section, nous nous limiterons aux formalismes les plus répandus dans les phases du cycle de développement en V. Notons que la dernière phase du cycle, liée à l'implémentation, sera présentée dans les sections suivantes.

1.2.3 Modèles de tâches

Afin d'exprimer les besoins des utilisateurs, souvent des non-informaticiens, de nombreuses notations de description ont été proposées dans le domaine des IHM. Ces notations appelées couramment *modèle de tâches* sont, pour la plupart, centrées utilisateurs et sont donc loin des implantations informatiques.

Les travaux dans le domaine des modèles de tâches font intervenir des communautés différentes et plus précisément des ergonomes et des psychologues. En outre, ces formalismes possèdent des qualités et des finalités très différentes. Le problème se pose alors pour le concepteur du choix du modèle adapté à ses besoins. Différentes classifications ont été proposées pour résoudre ce problème suivant différents critères. Citons par exemple la classification réalisée par [Bru98, Bal94] et reprise dans [Jam96] (modèles de tâches classés selon le pouvoir d'expression, utilisabilité, origine et finalité), dans [SPG89] et reprise dans [Tab01] (qui différencient les modèles linguistiques, les modèles hiérarchiques orientés tâche et les modèles de la connaissance), dans [Pat01] et reprise dans [Nav01] (modèles de tâches classés selon le langage utilisé pour décrire les modèles, le type d'information que les modèles contiennent et le support que ces notations fournissent), ou encore dernièrement dans [LV03] (but, discipline, les relations conceptuelles et le pouvoir d'expression).

Notre objectif n'est pas d'être exhaustif, c'est pourquoi nous présentons les modèles précurseurs et représentatifs. Nous découvrons aussi que la grande majorité de ces modèles de tâches prennent en considération les principes de Namur [Joh96]. Ces notations peuvent donc être distinguées selon les différents caractéristiques décrites ci-dessous :

- la capacité à abstraire : décomposition hiérarchique de tâches en sous-tâches ;
- la capacité à structurer : pouvoir d'expression des opérateurs et de leur capacité à composer ;

– esthétique : représentation textuelle ou graphique de la notation ;

– outils support : existence d'un outil d'aide à la modélisation et à la vérification ;

– sémantique claire, description précise des opérateurs.

Devant l'éventail important des classifications proposées pour les modèles de tâches, nous choisissons de les regrouper en deux catégories :

– les modèles d'analyse de tâches ;

– les modèles de description de l'interface homme-machine.

1.2.3.1 Modèles d'analyse de tâches : HTA et MAD

L'analyse de tâches consiste à *collecter des informations sur la façon dont les utilisateurs accomplissent une activité*. L'acquisition de ces informations est obtenue par les récits des utilisateurs au moyen de lectures de rapports, d'interviews ou de simulations, [DFAB93]. Mais cette analyse de l'activité doit être indépendante de toute idée d'interface à réaliser et ne doit pas comporter de sous-entendus sur les dispositifs d'interaction. Elle doit donc rester à un haut niveau d'abstraction. Les avantages explicités par [Bal94] sur l'indépendance des détails de l'interaction au niveau de l'analyse de tâches sont la portabilité et la génération automatique d'interfaces.

Ce sont les modèles liés à l'analyse de tâches qui servent de notations supports pour la collecte et l'analyse des informations. A titre d'illustration, nous n'aborderons ici que les modèles HTA et MAD.

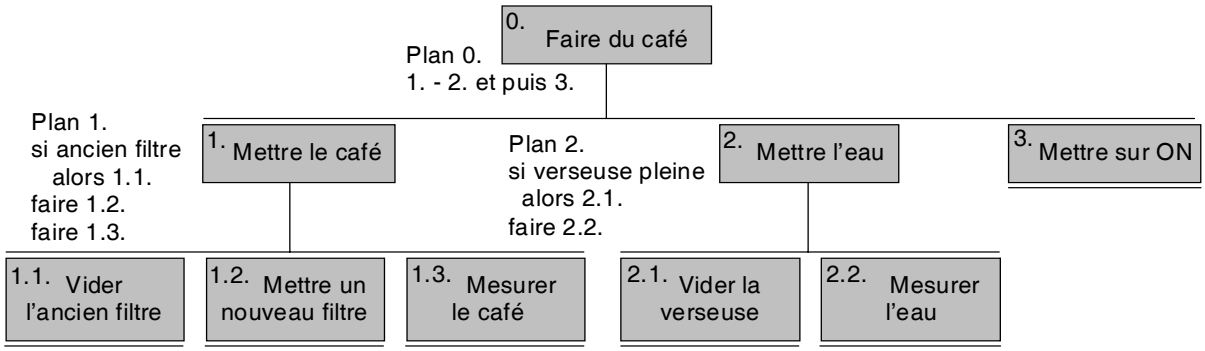

Fig. 1.5 – *Faire du café*, un exemple en notation HTA.

HTA. Le modèle HTA [She89] (Hierarchical Task Analysis) est l'une des premières approches pour la collecte et l'analyse des informations des activités de l'utilisateur. Ce modèle a été largement employé dans l'industrie et a connu de nombreuses variantes.

Sa capacité à abstraire est réalisée par la décomposition hiérarchique des tâches en sous-tâches selon un arbre tâches/sous-tâches, effectuées jusqu'aux tâches élémentaires. La notion de « plan » associée à une tâche révèle la façon dont ces sous-tâches se déroulent. Plus précisément, les plans définissent la structuration de la décomposition afin de décrire les relations temporelles et conditionnelles entre chaque sous-tâche.

HTA est une notation qui admet aussi bien une description graphique que textuelle de la représentation des informations liées aux activités de l'utilisateur. Sur la figure 1.5, nous proposons l'exemple « *Faire du café* » suivant la notation graphique de HTA [She89].

Toutefois, la sémantique de ce modèle est peu claire. En effet, l'absence de véritables opérateurs temporels avec une sémantique précise ne permet pas d'explorer le modèle de tâches afin d'établir les différentes tâches terminales qui entrent en jeu. Ce point peut être gênant pour assurer l'atteignabilité de certaines tâches. Par ailleurs, ce modèle manque d'outils et le coût de mise en œuvre s'accentue au fur et à mesure des décompositions dans l'arbre de tâches.

MAD. (Méthode Analytique de Description) [SPG89] est avant tout une méthode de conception basée sur une approche psycho-ergonomique rigoureuse. Elle se rapproche des travaux de HTA. MAD dispose également d'une notation graphique décrivant hiérarchiquement les tâches, mais contrairement au modèle HTA, MAD introduit différentes caractéristiques. La première est la décomposition hiérarchique des tâches à l'aide de constructeurs qui jouent le rôle de liens temporels. La seconde est la description précise de la tâche à l'aide d'attributs, de pré et de post-conditions. Nous proposons l'exemple de « *Faire du café* » suivant la notation MAD où nous montrons l'utilisation de l'attribut **FAC** (tâche facultative) et le constructeur **SEQ** (tâche séquentielle).

De nombreuses modifications ont ensuite été effectuées pour aboutir à une version appelée MAD* [GS97]. Cette version enrichit MAD par l'introduction d'opérateurs temporels (interruption, désactivation, tâche multi-utilisateur) mais aussi par une sémantique précise des objets manipulés (ces objets sont définis par trois types d'éléments **MADClass**, **MADInstance** et **MADAttribut**). La notation est également outillée pour permettre le recueil et l'édition de modèles avec des outils comme IMAD* [GS97].

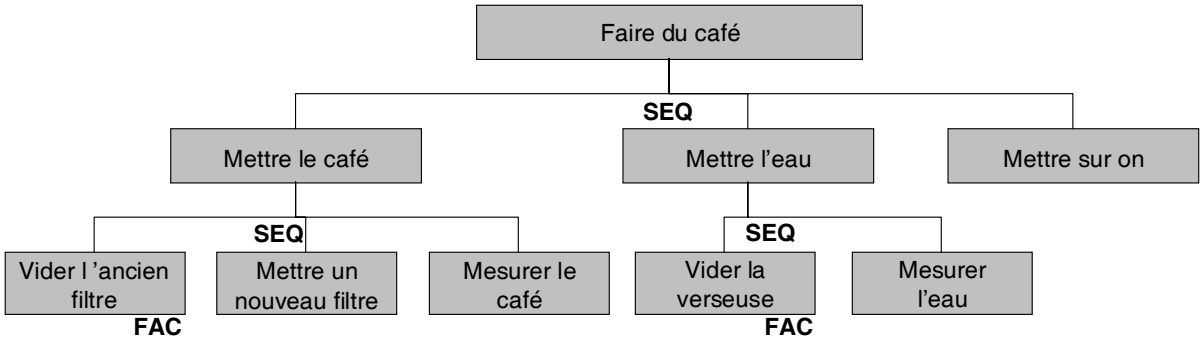

Fig. 1.6 – *Faire du café*, un exemple en notation MAD.

1.2.3.2 Modèles de description de l'interface homme-machine : UAN, XUAN et CTT

Les modèles de description de l'interface homme-machine définissent *la vue que l'utilisateur aura du système interactif*. D'après [Bal94], cette vue doit donner accès aux services définis dans l'analyse des besoins. Ces modèles s'inscrivent dans une logique de continuité par rapport à celle de l'analyse de la tâche, en plus ils doivent obligatoirement intégrer les retours d'information en provenance de l'interface.

Les modèles de description de l'interface homme-machine améliorent la communication entre le concepteur ergonomique et le concepteur de logiciel. A titre d'illustration, notre choix s'est porté sur les modèles UAN (et XUAN) et CTT.

UAN et XUAN. La notation UAN [HH93] (User Action Notation) a été conçue pour servir de support à l'expression des spécifications des interfaces à manipulation directe. C'est une notation qui, à la base, se limite à la description de l'interaction de l'utilisateur sur le système interactif. Elle décrit la tâche de l'utilisateur sous forme textuelle dans un tableau de trois colonnes : les actions physiques exécutées par l'utilisateur quand celui-ci interagit sur des dispositifs d'entrées (enfoncer le bouton de la souris), les retours d'information fournis par le système (objet sélectionné) et finalement l'état de l'interface (objet déplacé). Ainsi, ce niveau lié aux actions de l'utilisateur est particulièrement bien adapté parce qu'il est proche de l'implémentation. Il permet donc au concepteur d'avoir une vue globale sur les actions et leurs impacts sur l'interface. Nous proposons l'exemple *Ouvrir un fichier dans une application* décrit sur le tableau 1.1.

L'utilisateur déplace le pointeur de la souris au-dessus de l'icône du fichier ($\sim [file\_icon]$) puis appuie sur le bouton de la souris (Mv). L'icône du fichier passe alors en inverse vidéo ($file\_icon!$). L'utilisateur déplace la souris vers l'icône de l'application ($\sim$

$[x, y]$) où une image fantôme de l'icône suit le curseur de la souris ($outline(file\_icon) >\sim$). Quand l'icône est au dessus de l'application ($\sim$ [$application\_icon$]), l'icône de celle-ci passe en inverse vidéo ($application\_icon$!). Si l'utilisateur relâche le bouton de la souris $M\hat{}$, le fichier est alors ouvert avec l'application spécifiée $open(selected)$ et l'icône de l'application cesse d'être en inverse vidéo $application\_icon-$!.

Tâche : $Ouvrir fichier$		
Action Utilisateur	**Retour Information**	**Etat Interface**
$\sim [file\_icon]\ Mv$	$file\_icon$!	$selected = file$
$\sim [x, y]$	$outline(file\_icon) >\sim$	
$\sim [application\_icon]$	$application\_icon$!	
$M\hat{}$	$application\_icon-$!	$open(selected)$ avec l'application

Tab. 1.1 – *Ouvrir un fichier dans une application*, un exemple en UAN.

Dans sa forme actuelle, UAN ne propose que la description de tâches élémentaires. Son extension XUAN [GEM94] (Extented User Action Notation) comble cette lacune, en proposant un niveau supplémentaire à celui du niveau des actions, appelé niveau de tâches. Il s'agit d'une capacité à abstraire qui lui permet de décomposer hiérarchiquement des tâches en sous-tâches tout en tenant compte des descriptions du niveau des actions. Cette extension introduit évidement des opérateurs de composition, des relations temporelles et des pré- et post-conditions pour modéliser une structure plus complète de la tâche. Toutefois, à l'opposé de MAD, cette notation est entachée par l'absence des opérateurs d'interruption et de désactivation qui ne lui permettent pas de prendre en compte les erreurs [JBAA01]. De plus, à notre connaissance, il n'existe pas à l'heure actuelle d'outil qui permette de mettre en place une modélisation suivant cette notation.

CTT. La notation ConcurTaskTrees (CTT [Pat01]) met l'accent sur les activités de l'utilisateur et différencie quatre types de tâches : les tâches abstraites, les tâches utilisateurs, les tâches interactions et les tâches applications. C'est une notation à forte capacité à structurer, à l'instar d'UAN par exemple, car elle propose de nombreux opérateurs temporels issus du langage LOTOS (Language Of Temporal Ordering Specification) [Sys84] (interruption, concurrence, désactivation...). Elle permet aussi d'identifier les objets et les actions qui permettent de décrire, par exemple, le comportement de l'interface utilisateur.

CTT possède une représentation graphique et a l'avantage de proposer un outil support appelé CTTE (ConcurTaskTreesEnvironment) [PMG01] qui regroupe des outils d'édition, de simulation et de génération de suite de tâches (scénarii). CTTE offre ainsi la possibilité de simuler des modèles de tâches afin de vérifier la cohérence et le bon déroulement des tâches.

Sur la figure 1.7, nous proposons l'exemple du *Distributeur automatique de billets* qui est inspiré de l'exemple fourni avec CTTE. Ne sont présentés que les opérateurs d'activation (>>), de choix ([]), de désactivation ([>) et la caractéristique d'itération (∗). Nous modélisons le fait que l'utilisateur doive 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 n'est représentée que la tâche *Retirer Argent*. Cette tâche est désactivable par la tâche *Terminer Accès*.

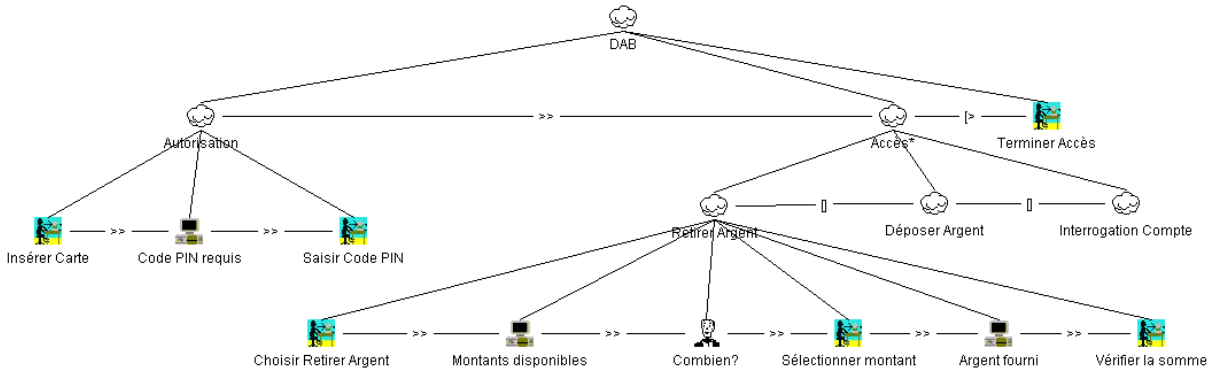

Fig. 1.7 – *Distributeur automatique de billets*, un exemple en notation CTT.

Toutefois, nous pouvons reprocher à cette notation la faiblesse de la description des objets contenue dans les tâches. En conséquence, il n'est pas envisageable, d'une part, de raisonner sur les objets d'une quelconque modélisation CTT, et d'autre part d'utiliser directement ces modélisations sur le système interactif au moment des phases de test.

Une description plus détaillée du formalisme CTT est fournie en annexe A de ce mémoire.

1.2.3.3 Conclusion sur les modèles de tâches

Nous avons décrit un échantillon de notations qui permettent, d'une part, la collecte des informations sur la façon dont les utilisateurs accomplissent une activité, et d'autre part, la description de l'interface homme-machine qui définit la vue que l'utilisateur aura du système interactif. Toutes ces notations constituent, en général, la base de départ de réalisation d'une IHM.

D'après [Bal94], le modèle de tâches issu du domaine IHM est utile pour deux raisons principales. D'une part, il peut servir d'outil dans la conception où il s'inscrit dans une phase du cycle de développement (analyse des besoins) et peut donc aider le concepteur à la conception, à l'évaluation de l'utilisabilité et à la rédaction de documentation (contenu

et structure). D'autre part, le modèle de tâches s'adresse à l'utilisateur pour l'aider dans son travail via l'utilisation d'une aide sous la forme de documentation[1], définie dans la phase de conception.

Les ergonomes et les psychologues, souvent à l'origine de ces notations, privilégient des approches graphiques avec une décomposition hiérarchique de tâches en sous-tâches représentées par des arbres. Cependant ces notations sont informelles et provoquent de nombreuses réserves. Par rapport aux caractéristiques de la section 1.2.3, nous pouvons dégager une première critique sur les opérateurs temporels aux niveaux de la richesse et de la clarté apportée à la sémantique. Dans une seconde critique, nous citons l'absence de description de la sémantique des objets associés au modèle.

En conséquence, des difficultés lors du passage de l'analyse de tâches au développement sont engendrées. En effet, comme nous l'avons vu ci-dessus, la modélisation de la tâche suit un processus purement informel. Elle oblige donc le concepteur à transcrire les besoins utilisateurs dans sa conception suivant un processus informel au risque de définir des représentations conceptuelles erronées et/ou ambiguës.

Dans la contribution que nous apportons dans cette thèse, nous avons employé la notation CTT malgré son caractère pseudo-formel. Ce choix, que nous approfondirons dans la section 1.5, a été motivé par la présence de riches opérateurs qui permettent de couvrir le cadre de description d'une application interactive, et la disponibilité de l'outil CTTE qui permet d'éditer et d'effectuer quelques vérifications sur les modèles CTT.

1.2.4 Modèles d'architecture

Les modèles d'architecture (voir figure 1.4) sont nés du constat que la conception des IHM est un processus complexe, donc itératif, et par conséquent la possibilité d'apporter des modifications au logiciel d'IHM est un élément important du cahier des charges. Dans ce sens, la décomposition de l'application interactive en différents éléments directeurs est rendue nécessaire. Les modèles d'architecture définis pour les IHM reposent sur un principe commun : la séparation du modèle sémantique de l'application (noyau fonctionnel) de l'interface fournie à l'utilisateur pour interagir avec le modèle. Les modèles d'architecture décrivent également la manière dont ces deux composants communiquent entre eux. Ainsi, ils offrent une structure générique destinée au concepteur de l'application.

Les apports d'une telle organisation modulaire sont importants. D'une part, cette organisation permet une conception plus simple dans la mesure où chaque module peut être

[1] On parle souvent dans ce cas d'aide en ligne ou d'aide contextuelle.

réalisé de manière plus ou moins indépendante. D'autre part, les modifications apportées aux modules s'en trouvent amoindris et leur fiabilité accrue.

De nombreux modèles d'architecture ont été élaborés au fur et à mesure des avancées techniques. Nous regroupons ces modèles d'architecture en trois grandes familles :

Les **modèles globaux** sont les plus anciens, comme par exemple le modèle SEEHEIM [Pfa85] ou ARCH [BPR+91] [BHH+88]. Ils s'intéressent à la séparation de la partie interface du reste de l'application, afin de rendre cette dernière indépendante de tout élément spécifique à la présentation. En revanche, ils ne définissent ni la structure interne des composants, ni les interfaces d'échange entre ces composants.

Les **modèles multi-agents** ont été mis en place avec l'avènement des techniques « orientées-objets » pour répondre aux exigences liées à la modularité, l'encapsulation et au parallélisme. L'interface utilisateur est répartie en une multitude d'agents actifs, réagissant à la réception de certains événements. PAC [Cou87] et MVC [Bur92] font partie de cette catégorie.

Les familles précédentes utilisent une seule logique de décomposition, et donc, décrivent pauvrement l'organisation interne des modules. D'autres modèles intégrant ce point de vue, qualifiés de **modèles hybrides**, ont été développés : H^4 [Gui95], le modèle MultiCouche [Fek96b] et PAC-Amodeus [NC91].

Dans cette section, nous présentons, sans souci d'exhaustivité, les modèles les plus cités dans la littérature et qui nous semblent les plus aptes à cerner le rôle de la décomposition modulaire : le modèle SEEHEIM, le modèle ARCH et les modèles MVC et PAC.

1.2.4.1 Le modèle de SEEHEIM

Le modèle de SEEHEIM [Pfa85], définit à l'issue d'une réunion de travail ayant eu lieu dans la ville de SEEHEIM, préconise la décomposition d'une application interactive en trois composants logiques : la présentation, l'interface avec l'application et le contrôleur de dialogue (voir figure 1.8).

- Le composant **présentation** est chargé d'afficher les informations de l'application à l'utilisateur (image du système) et de prendre en compte ses actions par l'intermédiaire des dispositifs d'entrée (souris ou clavier par exemple). C'est aussi dans la présentation que sont intégrées d'éventuelles règles d'ergonomie conduisant à une meilleure organisation de l'écran et une optimisation de l'interaction. Ce composant correspond aux aspects lexicaux du dialogue.

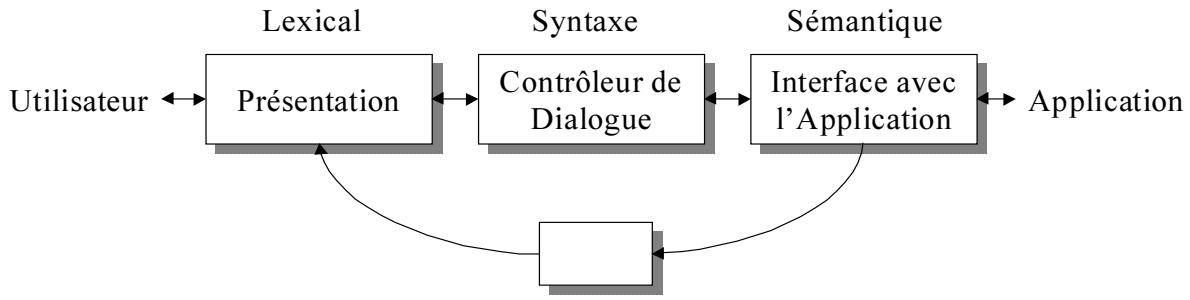

Fig. 1.8 – Le modèle SEEHEIM.

- Le composant **interface avec l'application** est une surcouche de l'application. Il représente l'application du point de vue de l'interface utilisateur. Son rôle est de transformer les données de la couche présentation, transmises par le contrôleur de dialogue, en données de l'application et réciproquement. Ainsi, c'est à ce niveau que les liens entre les concepts de l'application et ceux manipulés par l'utilisateur doivent être déterminés. Ce composant représente la sémantique de l'application.
- Enfin, le composant **contrôleur de dialogue** joue le rôle de médiateur entre la partie présentation et l'interface avec l'application. C'est lui qui est à même d'autoriser et de contrôler l'enchaînement des interactions de l'utilisateur, et qui déclenche les appels des fonctionnalités de l'interface avec application. Il s'occupe de la partie syntaxique de l'interaction.

Notons que le modèle sémantique ou noyau fonctionnel de l'application n'apparaît pas dans le modèle SEEHEIM.

Le modèle SEEHEIM fut le premier modèle d'architecture à proposer une décomposition modulaire d'une application interactive et à préconiser la séparation de l'interface homme-machine avec le reste de l'application. Cependant, plusieurs limites lui ont été attribuées, notamment la faible précision du fonctionnement du contrôleur de dialogue, ou bien la mise en lumière de son caractère monolithique face à l'émergence de l'approche orientée objet.

1.2.4.2 ARCH

Devant les faiblesses du modèle SEEHEIM, plusieurs autres modèles dont le modèle ARCH [BHH+88, BPR+91] ont été mis en œuvre. Ce modèle étend le modèle SEEHEIM, et propose deux composants supplémentaires. ARCH divise donc l'application interactive entière en cinq composants : boîte à outils, présentation, dialogue, adaptateur de domaine et domaine (voir figure 1.9).

- Le **domaine** représente les objets et les fonctions propres à l'application qui demeurent indépendantes de l'interface utilisateur.
- L'**adaptateur de domaine** joue le rôle d'intermédiaire entre le domaine et le dialogue en garantissant une totale indépendance entre eux. Il permet d'implémenter notamment les tâches utilisateurs relatives au domaine.
- Le **dialogue** est, comme dans SEEHEIM, chargé du contrôle du dialogue. Mais à la différence de celui-ci, son rôle est précisé, il est ainsi chargé de maintenir la cohérence entre les différentes vues d'un même objet.
- La **présentation** agit en intermédiaire entre le dialogue et la boîte à outils. Elle est composée de représentations abstraites d'objets d'interactions.
- Enfin, le composant **boite à outils** implémente l'interaction physique avec l'utilisateur.

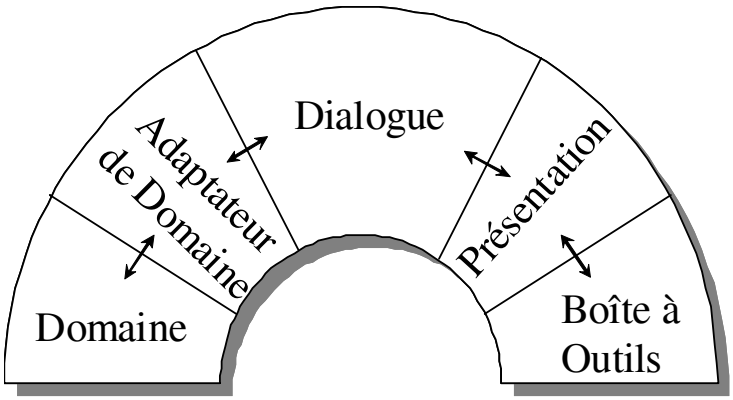

Fig. 1.9 – Le modèle ARCH.

La décomposition plus fine de ARCH surnommée « SEEHEIM étendu » supporte d'avantage les modifications dues aux évolutions de l'IHM. Cependant, comme le modèle SEEHEIM, ARCH ne précise pas la structure interne de ses composants.

1.2.4.3 MVC

MVC pour (Model View Controller) est issu du langage orienté-objet Smalltalk-80 [Gol84]. Ce modèle est largement utilisé dans l'industrie et particulièrement dans la boîte à outils Swing du langage JAVA. Ce modèle propose de décomposer l'interface utilisateur en une hiérarchie d'agents autonomes. Chaque agent est découpé en triplet (Modèle, Vue, Contrôleur) et communique avec les autres agents par envois de messages au moyen de son entité Modèle, figure 1.10. À la différence des modèles explorés précédemment, MVC est le premier modèle où le contrôle de l'interface est réparti et non pas centralisé. De

plus, ce modèle n'est pas statique et permet la création dynamique de modèles MVC supplémentaires.

- Le **modèle** réunit l'accès à l'ensemble des fonctionnalités du système et notifie les modifications à sa vue et aux autres objets MVC.
- La **vue** est la représentation externe des données du modèle. Elle interroge le modèle pour afficher les modifications survenues. Enfin, la vue prévient le contrôleur des modifications affectant les entrées.
- Le **contrôleur** gère et interprète les actions de l'utilisateur. Il prévient le modèle des actions et gère la vue.

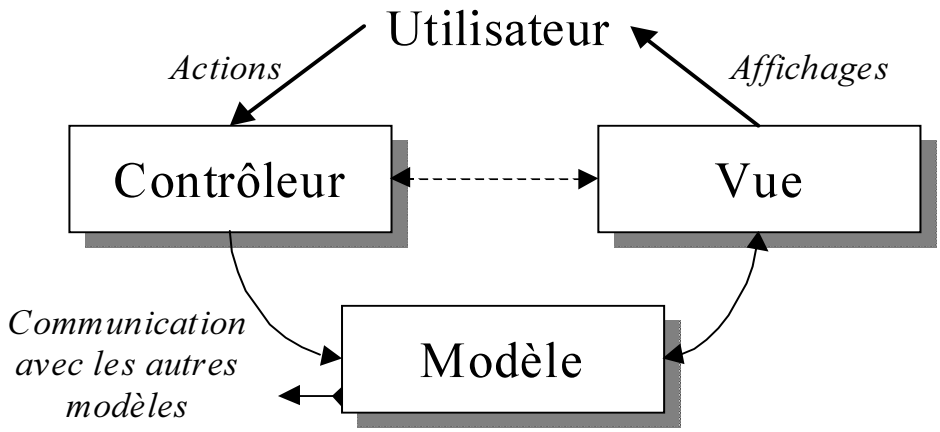

Fig. 1.10 – Le modèle MVC.

1.2.4.4 PAC

Le modèle d'architecture PAC (Présentation Abstraction Contrôle) a été développé en 1987 par [Cou87]. Cette architecture structure une application sous la forme d'une hiérarchie d'agents interactifs. PAC peut être vu, tout comme MVC, comme une architecture SEEHEIM répartie, à la différence que PAC peut modéliser l'intégralité d'une application, là où MVC ne s'occupe que de la décomposition de l'interface. De plus, PAC est indépendant du langage d'implémentation. La décomposition modulaire d'un système interactif est constituée de trois points : la présentation, l'abstraction et le contrôle comme le montre la figure 1.11.

- La **présentation** définit le comportement en entrée comme en sortie vis-à-vis de l'utilisateur.

- L'**abstraction** définit indépendamment des aspects graphiques les aspects conceptuels de l'agent. Au sens de SEEHEIM, elle constitue le noyau fonctionnel.
- Le **contrôle** sert de pont entre les aspects présentation et abstraction. Il prend en charge également les échanges avec les autres modèles.

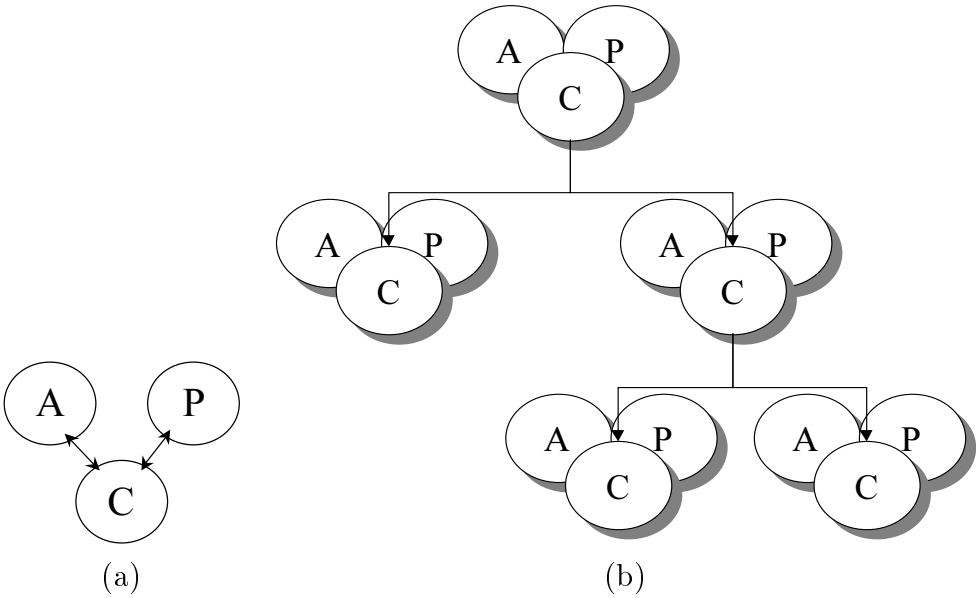

Fig. 1.11 – Agent PAC (a) et hiérarchie d'agent PAC (b).

1.2.4.5 Conclusion sur les modèles d'architecture

Les modèles d'architecture sont nombreux, ils sont plus ou moins bien adaptés à chaque style d'interface. Nous avons remarqué que l'objectif commun des modèles d'architecture est de définir des composants susceptibles d'être développés avec une relative indépendance, afin d'assurer par exemple des critères de génie logiciel, comme la modularité et la réutilisabilité.

Toutefois, même si la place et le rôle de l'utilisateur dans les architectures sont précisés (lecture des informations issues de la présentation ou les actions sur la présentation), il en va différemment en ce qui concerne l'intégration des besoins utilisateurs. En effet, dans tous les modèles évoqués précédemment, une lacune importante est l'absence de description du fonctionnement du composant dialogue qui doit exprimer la dynamique du dialogue homme-machine. Or, l'absence d'une description précise du dialogue ne permet pas de vérifier au plus tôt sa conformité vis-à-vis des modèles de tâches de l'utilisateur de la section 1.2.3.

En-soi, les modèles d'architecture permettent de définir la structure générale d'une

application sans décrire précisément comment elle va fonctionner. Les aspects liés à la vérification et à la validation sont donc absents.

C'est précisément l'objectif des formalismes de description du dialogue que de réaliser la représentation explicite du dialogue entre l'utilisateur et le système.

1.2.5 Modèles de description du dialogue

Les formalismes de description du dialogue, appelés également « modèles de dialogue », servent à spécifier la dynamique du dialogue homme-machine. Le rôle de ce dernier est défini dans [DFAB93] comme une double liaison. D'une part, il est lié à la sémantique du système interactif pour savoir ce qu'il doit faire, et d'autre part, le dialogue est lié à la présentation pour en donner sa visualisation.

De nombreux formalismes permettent la description du dialogue. Nous les retrouvons suivant des classifications comme [Bru98]. Nous pouvons citer des modèles issus de : langages à événements, grammaire, langages orientés objets, théorie des graphes, systèmes de transitions, réseaux de Petri, algèbre de processus et langages à flots de données.

De par le nombre important de ces formalismes, nous limitons notre exploration à ceux qui sont le plus utilisés dans la spécification du dialogue homme-machine. Ainsi, nous nous attardons uniquement sur les formalismes à base d'états comme les *automates* et les *réseaux de Petri* puis sur les formalismes à base d'*événements*.

1.2.5.1 Modèles à base de Systèmes de transitions

Historiquement, les travaux sur les automates à états, appelés systèmes de transition étiquetés (STE), sont les premiers à avoir représenté le dialogue d'une application interactive.

Les STE sont basés sur un modèle mathématique qui permet le raisonnement. Ils sont exprimables graphiquement. Un système interactif est décrit par un ensemble d'états, un ensemble d'actions, un état initial et une relation entre les états, appelée relation de transition. Le comportement du dialogue est établi par le déclenchement d'événements (étiquettes de la transition) qui permet le passage d'un état à un autre au moyen des transitions.

[Jac82] fut le premier à utiliser les automates pour la spécification du comportement des interfaces à l'aide de systèmes de transitions. Dans cette description, les arcs sont

étiquetés avec des éléments lexicaux d'entrée (actions sur la souris par exemple) et les noeuds donnent une représentation des états possibles de l'interaction. L'exemple donné sur la figure 1.12 décrit le système de transitions de « *Copier/Coller* » une zone de texte. Nous modélisons dans cet exemple les actions de l'utilisateur de choisir, dans un premier temps une zone de texte, d'effectuer la copie puis de terminer par le collage du texte soit à la suite du texte sélectionné ou soit dans une nouvelle position.

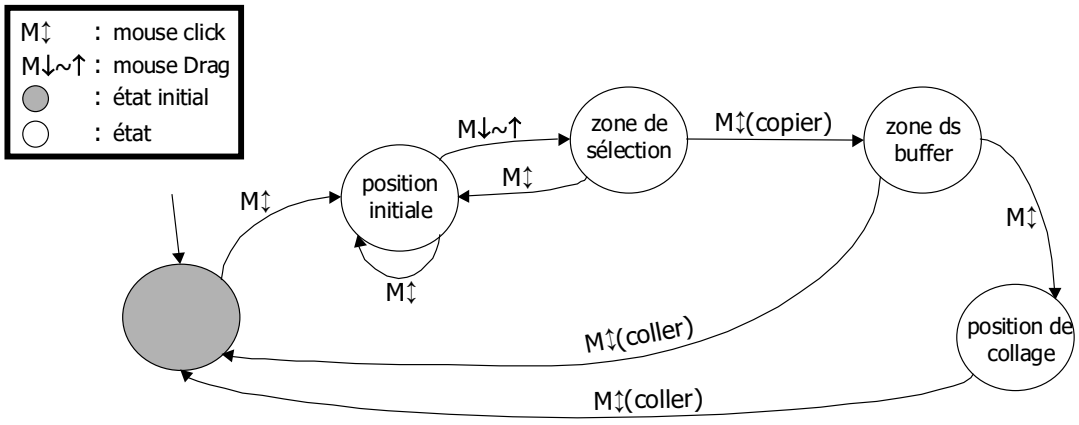

Fig. 1.12 – *Copier/Coller une zone de texte*, un exemple d'un système de transitions.

Pour modéliser un système complexe, celui-ci peut être obtenu par composition de sous-systèmes, chacun étant représenté par un automate. La composition est obtenue par une opération de produits cartésiens et de produits synchronisés. Ces opérateurs ont pour effet de produire des automates de plus en plus complexes au fur et à mesure que les produits sont effectués. Elles engendrent des modèles complexes à construire et à valider. Une solution aux problèmes d'explosion combinatoire dus aux produits composition/décomposition des automates est présentée dans le chapitre 2.

C'est ainsi que des extensions des automates ont été apportées. [Woo70] définit alors les diagrammes de transition récursifs (RTN : Recursive Transition Networks) qui permettent une structuration hiérarchique d'un automate à états finis. De la même façon, [Was81], définit les réseaux de transition augmentés (ATN : Augmented Transition Networks) qui introduisent la notion de registres (variables d'état de l'automate). Enfin, citons, les statecharts définis par [Har87] qui autorisent par exemple l'expression du parallélisme et de la synchronisation.

Enfin notons que les STE sont associés à des systèmes de preuve permettant la validation et la vérification de nombreuses propriétés du dialogue dans les IHM (atteignabilité, sûreté, vivacité, etc).

1.2.5.2 Réseaux de Petri

Les réseaux de Petri (RdP) dérivés des automates sont également définis mathématiquement et offrent une représentation graphique. Ils permettent la modélisation de systèmes concurrents.

Un réseau de Petri est composé de deux types d'objets, les **places** et les **transitions**. Les places désignent les états possibles du système, et les transitions désignent les opérateurs de changement d'état. L'état du réseau est représenté par un ensemble de jetons répartis dans les places du réseau, appelé marquage du réseau. Les places et les transitions sont reliées par des arcs orientés. La figure 1.13 illustre le réseau de Petri de l'exemple « *Copier/Coller* » une zone de texte. Sur la zone de droite nous décrivons les places et les transitions du réseau de Petri.

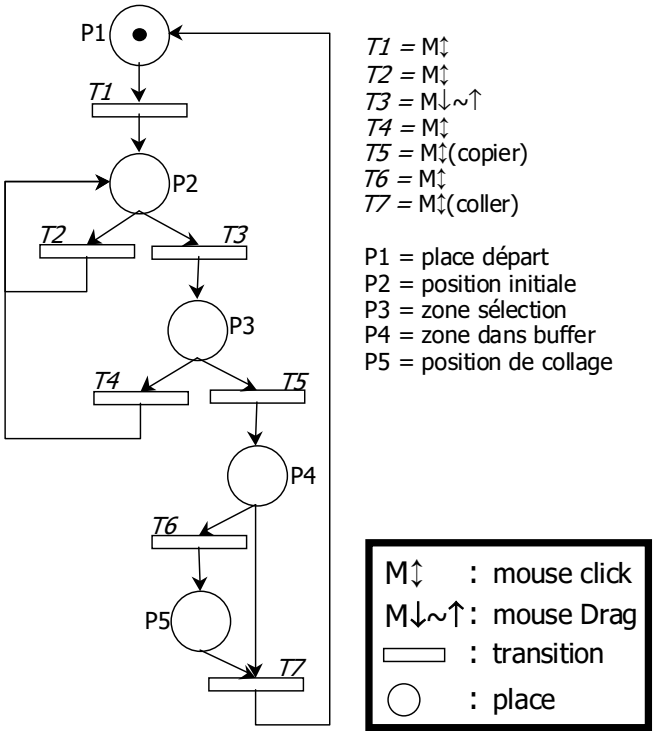

Fig. 1.13 – *Copier/Coller une zone de texte*, un exemple d'un réseau de Petri.

Toutefois, les réseaux de Petri présentent les mêmes inconvénients que les automates à états finis relativement à la modularité, puisqu'un système est modélisé par un seul réseau.

C'est dans ce sens que des travaux de recherche se sont intéressés à l'extension de ce formalisme. Citons par exemple les réseaux de Petri à objets (RPO) [SB85] qui utilisent

le concept de structuration lié aux langages orientés objets. Mais, ce n'est qu'à partir des Objets Coopératifs [Bas92] que la modularité est intervenue dans la modélisation. En effet, le système complet est alors constitué d'un certain nombre d'Objets Coopératifs qui communiquent entre eux. Les Objets Coopératifs ne sont cependant pas spécialement destinés aux domaines des IHM. C'est pourquoi [Pal92, PBS95] adjoint la notion de composant de présentation aux Objets Coopératifs pour définir le formalisme des Objets Coopératifs Interactifs (ICO). Nous reviendrons plus précisément sur l'apport des ICO dans le domaine des IHM dans les sections 1.3 et 1.4.

Finalement, à l'image des STE, les réseaux de Petri présentent une explosion combinatoire du nombre d'états et de transitions, qui s'accentue à mesure que le système à modéliser devient complexe.

1.2.5.3 Modèles à base d'événements

Les modèles à base d'événements sont à l'opposé des modèles à base d'états dans le sens où les premiers décrivent la manière de parvenir aux états des secondes. Ils reposent sur le concept d'événements, de gestionnaires d'événements et de procédures de traitement [Pal92]. Lorsqu'un événement est émis, il est envoyé au gestionnaire d'événements pour une phase de traitement et il est dirigé vers la procédure de traitement susceptible de le prendre en compte. Enfin la procédure de traitement exécute un traitement particulier suivant le type et les paramètres de l'événement.

L'intérêt des modèles à base d'événements est l'exécutabilité et leur important pouvoir d'expression (concurrence par exemple). Les modèles à base d'événements sont largement utilisés dans la plupart des langages de programmation (le langage Java/Swing[2]) et dans plusieurs langages de modélisation (LUSTRE [HCRP91]).

Toutefois, à l'inverse des formalismes à états, les modèles à base d'événements ne permettent pas de représenter explicitement l'état du dialogue du système interactif [Gui95]. Des informations plus précises concernant les formalismes à événements peuvent être trouvés dans [Tar93].

1.2.5.4 Conclusion sur les modèles de description du dialogue

Nous venons de voir que les formalismes de description du dialogue permettaient de décrire la dynamique du dialogue homme-machine suivant un principe commun. Le système

[2]Sun Microsystems : http ://www.sun.com

interactif peut être vu comme un ensemble d'états liés par des transitions et le passage d'un état à un autre établit alors la dynamique du système ou bien comme ensemble d'événements identifiant des états.

Toutefois, la modélisation du dialogue doit naturellement intégrer l'utilisateur pour garantir l'utilisabilité du futur système. Toute trace ou scénario issu du modèle de tâches qui décrit les besoins de cet utilisateur doit alors se retrouver dans la dynamique du dialogue. Malheureusement, nous avons montré dans la section 1.2.3 que les notations de description des besoins utilisateurs privilégiaient l'aspect graphique au détriment de la clarté de la sémantique. Dans ce sens, seuls des tests exhaustifs permettent de valider que les traces issues du modèle de tâches sont *incluses* dans la modélisation du dialogue.

Nous proposons une double contribution dans les chapitres 2 et 3 pour répondre à ce problème de validation de tâches. D'une part, l'approche qualifiée de formelle consiste à formaliser suivant une technique formelle unique un modèle de tâches CTT et un dialogue décrit par des STE. D'autre part, l'approche qualifiée d'expérimentale utilise des outils pour construire un modèle de tâches CTT et un dialogue défini par une approche événementielle.

1.2.6 Propriétés dans les IHM

Les propriétés dans les interfaces homme-machine sont celles couramment définies pour les systèmes informatiques en général, telles que les propriétés de sûreté et d'équité auxquelles s'ajoutent les propriétés particulières aux IHM, comme celles liées à la représentation ou celles issues des exigences des ergonomes ou des psychologues.

Les propriétés définissent la notion d'utilisabilité d'un système interactif. Elles qualifient la capacité d'un système à permettre à l'utilisateur d'atteindre ses objectifs avec efficacité, confort et sécurité.

De nombreuses formulations et classifications des propriétés inhérentes aux IHM ont été proposées dans [DFAB93] [GC96] [Roc98] [DH95]. Nous pouvons les résumer suivant deux grandes classes de propriétés :

- les propriétés de **validité** qui caractérisent un fonctionnement attendu ou voulu par un utilisateur ;
- les propriétés de **robustesse** qui sont relatives à la sûreté de fonctionnement du système.

Dans cette section, nous détaillons ces deux classes de propriétés.

1.2.6.1 Propriétés de validité

Dans cette catégorie de propriétés, nous distinguons les propriétés de complétude pour la réalisation d'un objectif donné et les propriétés de flexibilité pour la représentation de l'information, pour le déroulement des tâches et pour l'adaptation du dialogue face aux réactions de l'utilisateur.

Propriétés de complétude. [GC96] s'intéressent à différencier la complétude des traitements qui est une propriété du système (*Task Completeness*) de la complétude des objectifs qui est une propriété impliquant l'utilisateur et le système (*Goal Completeness*). Dans le premier cas, on parle de complétude des traitements si le système est à même d'autoriser l'exécution correcte des traitements qui le composent. On parle cependant de complétude des objectifs si l'utilisateur peut atteindre un objectif donné au moyen du système.

Propriétés de flexibilité. La flexibilité se rapporte aux différentes façons avec lesquelles l'utilisateur et le système échangent des informations au moment de l'exécution d'une tâche. Selon [GC96], les propriétés de flexibilité *obligent de la part des concepteurs à reconnaître que les gens réagissent différemment, et que ces mêmes concepteurs doivent répondre aux différences et aux préférences de l'utilisateur en fournissant une variété de techniques d'interaction.* Toujours selon [GC96], les propriétés de flexibilité peuvent être listées suivant trois sous catégories : la représentation de l'information, l'adaptation du dialogue et le déroulement des tâches.

Représentation de l'information. Cette sous-catégorie englobe les propriétés liées à la multiplicité des périphériques, à la multiplicité de la représentation et la réutilisabilité des données d'entrée et de sortie.

- La **multiplicité des périphériques** est la capacité du système à offrir plusieurs périphériques d'entrées (souris, clavier, caméra vidéo, etc) et de sortie (écran, sons, etc) pour la communication.
- La **multiplicité de la représentation** est la capacité du système à fournir plusieurs représentations d'un même concept. Dans le cas d'une horloge, il existe plusieurs formes différentes de représentation : numérique/analogique.
- La **réutilisabilité** des données d'entrée et de sortie est la capacité du système à autoriser l'usage des entrées et des sorties précédentes comme entrées futures. Dans le cas des sorties du système, la technique du *couper-coller* est un concept

utilisable comme données d'entrée. A l'inverse les *valeurs par défaut* comme entrées de l'utilisateur sont réutilisables par le système en sortie.

Adaptation du dialogue. Cette catégorie englobe principalement les propriétés liées à l'adaptativité et à l'adaptabilité.

- L'**adaptativité** est la capacité du système à s'adapter à l'utilisateur sans intervention explicite de sa part (par exemple, menus adaptatifs de la dernière série de la suite Office de Microsoft).
- L'**adaptabilité** ou la **reconfigurabilité** est la capacité du système à supporter sa personnalisation de la part de l'utilisateur (par exemple, personnalisation des barres d'outils).

Déroulement des tâches. Enfin, cette catégorie de propriétés englobe principalement les propriétés liées à l'atteignabilité, la non-préemption et l'interaction de plusieurs fils (multithreading).

- L'**atteignabilité** est la capacité du système à permettre à l'utilisateur d'atteindre un état désiré du système depuis l'état actuel.
- La **non-préemption** est la capacité du système à fournir directement le prochain but à l'utilisateur sans lui imposer un cadre strict.
- L'**interaction à plusieurs fils** est la capacité du système à fournir un système de gestion de processus entrelacés. Il permet à l'utilisateur de lancer plusieurs traitement à la fois.

1.2.6.2 Propriétés de robustesse

Selon [GC96], *un système interactif est dit robuste s'il assiste l'utilisateur dans l'accomplissement de ses tâches, sans qu'il se produise des erreurs irréversibles, et s'il donne aux utilisateurs une représentation correcte et complète de la progression de la tâche.* Parmi les propriétés de robustesse, nous trouvons les propriétés liées à la visualisation du système pour l'observabilité, l'insistance et l'honnêteté, et les propriétés liées à la gestion des erreurs principalement pour la prédictabilité et la tolérance aux écarts.

Visualisation du système. Les propriétés liées à la visualisation du système permettent d'assurer une représentation correcte et complète de l'état du système interactif via l'interface.

- L'**observabilité** est la capacité pour l'utilisateur à évaluer l'état interne du système. Le système fait en sorte que toutes les informations disponibles soient visualisées à l'écran.
- L'**insistance** est la capacité du système à forcer la perception de l'état du système. Le système fait en sorte que les informations nécessaires à l'utilisateur soient affichées à l'écran.
- L'**honnêteté** est la capacité à rendre conforme l'état interne du système aux yeux de l'utilisateur (WYSIWYG)[3].

Gestion des erreurs. Les propriétés liées caractérisent la capacité du système à recouvrer ou prévenir les erreurs des utilisateurs.

- La **prédictabilité** est la capacité pour l'utilisateur de prévoir les états accessibles du système à partir d'un état courant observable.
- La **tolérance aux écarts** est la capacité du système à aider l'utilisateur lorsqu'une ou plusieurs erreurs se produisent sans conséquences catastrophiques.

1.2.6.3 Conclusion sur les propriétés dans les IHM

Nous avons vu dans cette sous-section un ensemble de propriétés permettant de qualifier la facilité d'utilisation d'une interface interactive. Toutefois cette liste n'est pas exhaustive, la définition et la description de propriétés liées à l'ergonomie pour la représentation graphique des objets et à la psychologie ne font pas l'objet du champ de notre étude.

La vérification des propriétés permet de garantir l'utilisabilité de l'interface interactive. Dans ce but, il existe différentes techniques de vérification selon l'approche utilisée au moment de la phase de spécification. Rappelons que cette dernière s'intéresse principalement à l'établissement des modèles vus précédemment (modèles de tâches, modèles d'architecture et modèles de description du dialogue). On distingue principalement deux types d'approches de modélisation :

- *Les approches formelles* : cette première approche consiste à élaborer des modèles sur la base de langages formels durant la phase de spécification. En effet, la modélisation formelle reste nécessaire afin de raisonner sur les modèles de la spécification. Par raisonnement, nous entendons la possibilité d'exprimer, de vérifier et de prouver à priori des propriétés sur les systèmes interactifs. La section 1.3 s'intéresse à explorer

[3] **W**hat **Y**ou **S**ee is **W**hat **Y**ou **G**et

ces approches formelles appliquées au domaine de l'interaction homme-machine qui permettent d'exprimer formellement le système interactif, les propriétés et de les vérifier.

– *Les approches expérimentales* : cette seconde approche fait appel à des techniques expérimentales (langages, outils ou environnement) inscrites dans un processus de développement itératif. Elle intègre à chaque étape du processus différentes personnes concernées par la réalisation du système interactif. L'absence de formalisation des modèles oblige le concepteur à vérifier d'une part que le système fonctionne correctement et d'autre part qu'il correspond aux attentes de l'utilisateur. Dans cette optique, cette phase de validation passe obligatoirement par un ensemble de tests pour vérifier que le comportement et les sorties du système sont bien ceux attendus. Cette phase peut être éventuellement assistée par des outils de validation. La section 1.4 explore les approches dites *outils* qui permettent de vérifier les propriétés du système interactif à développer.

Enfin, quelle que soit la nature de l'approche utilisée, la *relecture des spécifications* permet d'évaluer la qualité du système interactif. C'est une activité qui nécessite la connaissance précise de l'ensemble des formalismes et notations employés dans ces spécifications.

Des informations complémentaires au sujet de la vérification des propriétés peuvent être trouvées dans [JBAA01].

Au niveau de la vérification des propriétés, nous proposons une contribution complémentaire (formelle et expérimentale) dans les chapitres 2 et 3.

1.2.7 Bilan sur le domaine de l'interaction homme-machine

Dans cette section, nous avons montré que pour concevoir un système interactif tout en assurant des critères d'utilisabilité, il fallait principalement intégrer les besoins de l'utilisateur dans la conception.

L'utilisation de techniques issues du monde du génie logiciel (cycle de développement, modélisation, vérification et validation, etc), l'exploitation de notations et de modèles (modèles de tâches, modèles d'architecture et modèles de description du dialogue) provenant de différentes communautés (informaticiens, ergonomes, psychologues etc.) permettent d'assurer ce critère de qualité.

Nous avons aussi listé les propriétés inhérentes aux IHM qui doivent être vérifiées pour assurer la notion d'utilisabilité du système interactif.

Il reste à savoir comment ces propriétés vont être vérifiées par les approches formelles et expérimentales. Tel est l'objet des prochaines sections.

1.3 Les techniques formelles pour la conception des systèmes interactifs

Une technique est dite formelle si le (ou les) langage(s) de spécification servant à décrire un système est exprimé dans un langage à syntaxe et à sémantique formelle, à base mathématique. Une syntaxe formelle permet alors de décrire les différentes constructions autorisées tandis qu'une sémantique formelle permet d'exprimer sans ambiguïté le sens des constructions syntaxiques. Un système de preuve associé permet d'établir des propriétés sur les objets manipulés par cette technique.

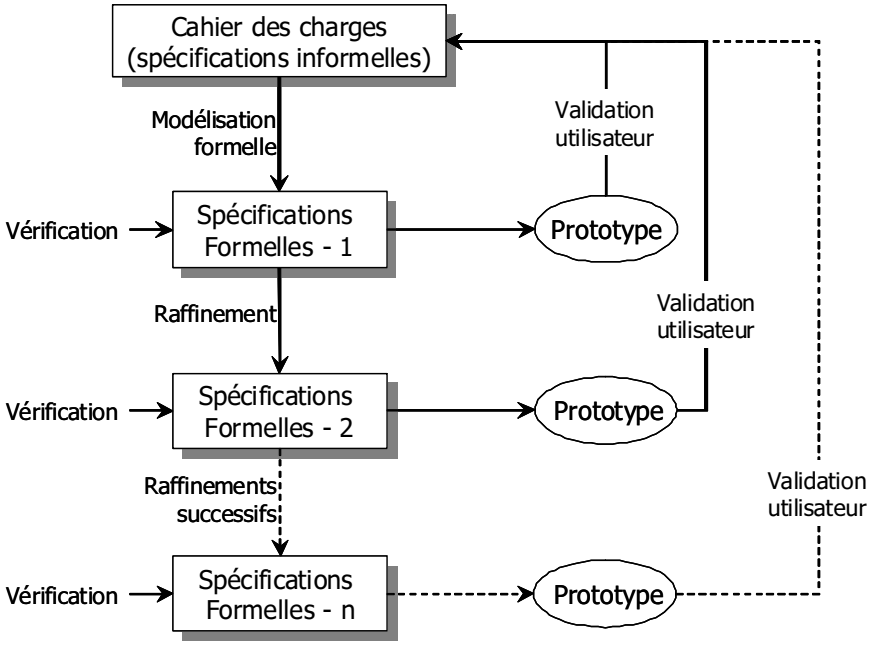

Fig. 1.14 – Processus de développement formel.

Des outils de preuve sont aussi utilisés pour traiter les spécifications formelles de façon automatique ou semi-automatique. Ils permettent notamment de vérifier des propriétés du système par *model-checking* ou par la preuve en utilisant un *theorem prover* qui permet de décharger les obligations de preuve.

L'émergence des techniques formelles a été motivée par le besoin de développer des logiciels sûrs. L'utilisation accrue de systèmes informatiques dans le fonctionnement de

systèmes critiques (avionique, contrôle de traffic aérien, gestion de centrale nucléaire, systèmes embarqués, etc.) et la mise en évidence d'anomalies (passage à l'an 2000, vol 501 d'Ariane) ont imposé l'évolution du processus de développement des logiciels de façon à en améliorer la fiabilité.

C'est dans ce but que l'intégration des techniques formelles dans les différentes phases du cycle de développement d'un système est rendue indispensable (spécification, conception, vérification et validation) telle que présentée dans la figure 1.14. Dans la phase de spécification, le système est décrit au moyen d'un modèle formel qui en donne une abstraction. Le modèle spécifie les exigences exprimées dans le cahier des charges et permet alors d'y détecter des erreurs d'incohérences et donc de satisfaire ces exigences. Enfin la possibilité de vérifier automatiquement les propriétés du système, réduit considérablement les phases de tests exhaustifs.

Depuis peu, avec l'emergence de la complexité des interfaces homme-machine et donc du besoin d'en assurer une meilleure utilisabilité, des travaux se sont intéressés à mettre en œuvre des techniques formelles dans ce domaine.

En conséquence, une première partie de cette section présente les différentes familles de techniques formelles exposant leurs avantages et leurs inconvénients. Une deuxième partie répond à la question de la nécessité de formalisation des interfaces homme-machine. Nous étudierons en quoi les modèles et les notations du domaine des IHM se doivent d'être formalisés. Enfin la dernière partie est un état de l'art des approches formelles dans ce domaine.

1.3.1 Classification des techniques formelles

Une technique formelle se caractérise par sa sémantique formelle ainsi que son système de preuve. C'est dans cette optique que nous nous intéressons à distinguer d'une part, les différentes approches formelles suivant la sémantique employée, et d'autre part, les systèmes de preuve. Cette classification proposée hors domaine d'étude met dans un dernier point l'accent sur les différents avantages et inconvénients des techniques formelles. Elle est basée sur les études de [AA00c], [JBAA01] et de [Bru98].

1.3.1.1 Approches formelles

Deux approches sont à distinguer : **l'approche basée sur les modèles** et **l'approche algébrique**. Ces approches intègrent une sémantique particulière permettant la description du système ainsi que les propriétés de ce système.

- **Approche basée sur modèles** : différentes dénominations sont proposées (approche *constructive*, *opérationnelle* ou *basée sur état explicite*). Cette approche se fonde sur la définition d'un état du système qui évolue en fonction des opérations qui sont appliquées [AA00c, JBAA01]. Ainsi, l'état du système est modifié par des opérations qui définissent l'évolution du modèle du système. Une opération est spécifiée à l'aide de pré- et post-conditions et permet d'interroger ou de modifier des variables de l'état. Les propriétés sur les systèmes modélisés peuvent être exprimées par l'intermédiaire d'expressions logiques sur les variables de l'état. Nous retrouvons à titre d'exemple dans la famille d'approches basées sur modèles, les STE, les réseaux de Petri, cités précédemment dans la section 1.2.5.2, VDM [Bjo87], B [Abr96a] ou Z [Spi88] ;
- **Approche algébrique** : aussi appelée approche *fonctionnelle*, elle concerne les langages où la sémantique est donnée par une algèbre. Les variables et les opérations sont décrites suivant cette algèbre. En d'autres termes, cette approche décrit le système par un ensemble d'équations pour en saisir le comportement. De nombreux langages de spécification algébriques ont été définis comme LOTOS [Sys84] ou LUSTRE [HCRP91].

1.3.1.2 Systèmes de preuves formelles

Les systèmes de preuves formelles permettent de prouver les propriétés sur ces systèmes exprimés dans la sémantique formelle d'une technique formelle particulière. Il existe deux principaux types de systèmes de preuve :

- **Vérification sur modèle ou « model checking »** : Ce type de systèmes de preuve est principalement adapté aux formalismes de type *automate*, par exemple les réseaux de Petri, les automates à états finis, LOTOS ou LUSTRE. Son principe est de parcourir chacun des états et de vérifier que chaque propriété est vraie dans tous les états possibles du modèle.
- **Démonstration de théorèmes ou « theorem proving »** : Ce type de système de preuve est basé sur un ensemble d'axiomes ainsi que sur un ensemble de règles de déduction. Les propriétés sont démontrées comme des théorèmes. Ces derniers sont prouvés à partir de l'application de règle de déduction. B, Z ou VDM utilisent la démonstration de théorèmes comme système de preuves.

1.3.1.3 Conception descendante / ascendante

Suivant la sémantique et le système de preuve employés, nous distinguons deux types de modélisation.

La modélisation dite **descendante** ou par décomposition : cette modélisation consiste à construire, à partir d'une spécification abstraite, une spécification concrète en passant par une ou plusieurs étapes de raffinements. Le raffinement est un mécanisme qui permet d'introduire de nouvelles descriptions à la spécification. La totalité du système est ainsi bâtie de proche en proche. Par exemple dans le langage B, l'étape de raffinement s'arrête à l'obtention d'un modèle appelé implémentation. Il s'agit du niveau le plus concret et du niveau langage de programmation. L'avantage de cette modélisation par raffinement est de permettre la révision ainsi que la précision des spécifications au fur et à mesure que le développement est réalisé. De plus, elle permet de définir les propriétés à prouver à différents niveaux dés que suffisamment d'éléments permettant leur expression apparaissent. Enfin, un autre avantage est la conservation, par le raffinement, des propriétés déjà établies.

La modélisation **ascendante** ou par composition est à l'opposé de la modélisation *descendante* : elle se base sur une composition de sous-systèmes pour décrire un système plus complexe. Cette modélisation est à comparer avec les langages de programmation qui exploitent la modularité pour structurer la conception. Il est souvent nécessaire de prouver des propriétés sur le système dans sa totalité et cela peut aboutir à des preuves ou à des vérifications complexes.

Le travail que nous présentons dans le chapitre 2 propose une utilisation de ces deux types de modélisation. Nous approfondirons notamment l'intérêt d'une modélisation descendante.

1.3.2 La nécessité de la formalisation : Insuffisances des notations semi-formelles dans la conception des IHM

La modélisation d'une application interactive au moyen de techniques semi-formelles présente des insuffisances.

Tout d'abord la description semi-formelle du contenu des modules de l'architecture logicielle s'appuie sur une sémantique pauvre et ambiguë, c'est-à-dire qu'aucun raisonnement ni preuve sur les méthodes ne peut-être effectué. La spécification semi-formelle ne peut garantir que le typage et les propriétés pertinentes du système peuvent être validées.

En ce qui concerne la notation de description de tâches utilisateurs, leur sémantique n'autorise au plus que la vérification de l'ordonnancement des tâches. Cette vérification assure qu'une tâche du modèle est atteignable. En revanche, elle ne permet pas la vérification des contraintes qui permet de garantir par exemple que la postcondition d'une tâche mère sera conforme à la postcondition de ses tâches filles. Actuellement la phase de vérification de l'ordonnancement de tâches s'effectue par simulation et réside dans la nécessité de passer par de lourdes phases de tests. Des outils tels que CTTE permettent de supporter ces tests. Mais tous les contrôles sont reportés à l'étape de test et n'utilisent que les techniques classiques de vérification et de tests de programmes [JGB99]. Dans tous les cas, la validation du modèle de tâches ne peut se faire que lorsque l'IHM est dans une phase de construction avancée. Il serait profitable d'effectuer cette validation au plus tôt dans le développement au niveau de la spécification ou de la conception, en fait dès qu'elle est possible.

La nécessité de la formalisation devient donc indispensable puisque les modèles et les notations semi-formelles ne donnent qu'un synoptique et une structuration de l'IHM qui aident pour le développement. L'absence de sémantique formelle rend empirique l'application de ces notations. Il n'est donc pas possible d'effectuer des vérifications formelles sur ces modèles en amont à l'aide d'un outil support car ces modèles sont directement codés dans des langages n'autorisant pas la preuve [AA00c].

1.3.3 Les techniques formelles pour la conception des IHM

Les insuffisances évoquées précédemment ont conduit les chercheurs de la communauté IHM à proposer l'utilisation de techniques formelles pour résoudre les problèmes de vérification et de validation. L'application de ces techniques formelles dans les IHM a suivi l'évolution historique de ces techniques et de leur utilisation dans le génie logiciel [Gau95]. De nombreuses techniques ont alors été employées à différents niveaux de spécification du système interactif.

Nous présentons ici un résumé des travaux effectués dans le domaine de l'IHM suivant une classification à trois niveaux : dialogue, composant et système complet. Toutefois certaines des approches que nous présentons peuvent correspondre à plusieurs niveaux.

Les premiers travaux se sont intéressés à décrire uniquement le niveau du dialogue par l'intermédiaire de techniques issues de la famille des automates. Il s'agit des formalismes étudiés dans la section 1.2.5, les automates à états finis [Woo70] ou encore les travaux sur les réseaux de Petri comme les ICO [Pal92, PBS95].

D'autres approches sont utilisées pour la description du niveau composant logiciel.

Citons les approches à base d'*interacteurs* (*interactor* en anglais) qui, sans être des approches strictement du niveau composant, parce que les interacteurs modélisent aussi une partie du système, peuvent s'en rapprocher. Les interacteurs utilisent le concept d'agent qui a été mis en place avec l'avènement des techniques « orientées-objets » afin de répondre aux exigences liées à la modularité, l'encapsulation et au parallélisme. Citons par exemple les interacteurs de CNUCE [FP90, PF92] qui utilisent LOTOS, les interacteurs de York [DH93, DH95] décrit en langage Z, et enfin les interacteurs du CERT [Roc98] modélisés par le langage synchrone LUSTRE. Dans ce niveau, nous pouvons aussi citer les travaux récents de [Jam03] en terme de vérification et de rétro-conception de composants graphiques.

Cependant, les approches précédemment citées ne permettent pas (même si elles s'en rapprochent) de modéliser complètement le système. C'est donc dans une dernière approche que nous avons classé les travaux qui permettent de couvrir la totalité du cycle de développement d'un logiciel interactif (spécification, validation, conception, vérification et implémentation). La méthode B a été largement exploitée dans cette optique [AAGJ98b] et [Jam02]. Cependant, ces travaux n'ont pas abordé l'aspect validation de tâches. C'est ce que nous proposons en complément dans le chapitre 2.

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.

1.3.3.1 Niveau composant : Interacteurs (CNUCE, York et CERT)

Différentes définitions des interacteurs ont été proposées suivant la nature de l'approche employée [Mar97]. La notion d'interacteur repose sur un principe commum : 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.

Interacteurs de CNUCE. Les interacteurs de CNUCE ont été développés dans le cadre de la formalisation du projet GKS [GKS85]. Ce projet a 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 fonctionne de façon 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.

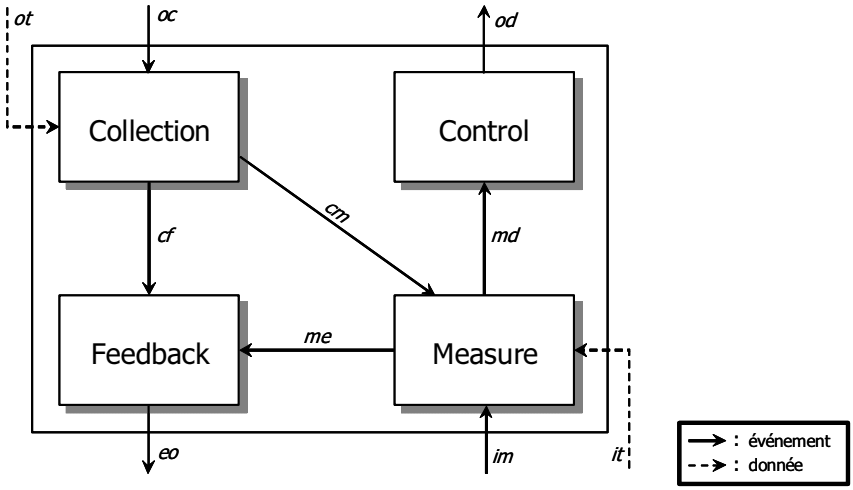

Fig. 1.15 – *Boîte blanche* de l'interacteur de CNUCE, adapté de [PF92].

La figure 1.15 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 *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 *oc* (**o**utput **c**ollection) à la fonction *Feedback*. De même, elle transmet la donnée *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.

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 en parallèle des interacteurs composés est exprimé par l'intermédiaire d'opérations de composition parallèle des processus.

Nous proposons sur la figure 1.16 l'exemple de la modélisation d'un *scrollbar* suivant une vue *boîte noire*, adapté de [PF92].

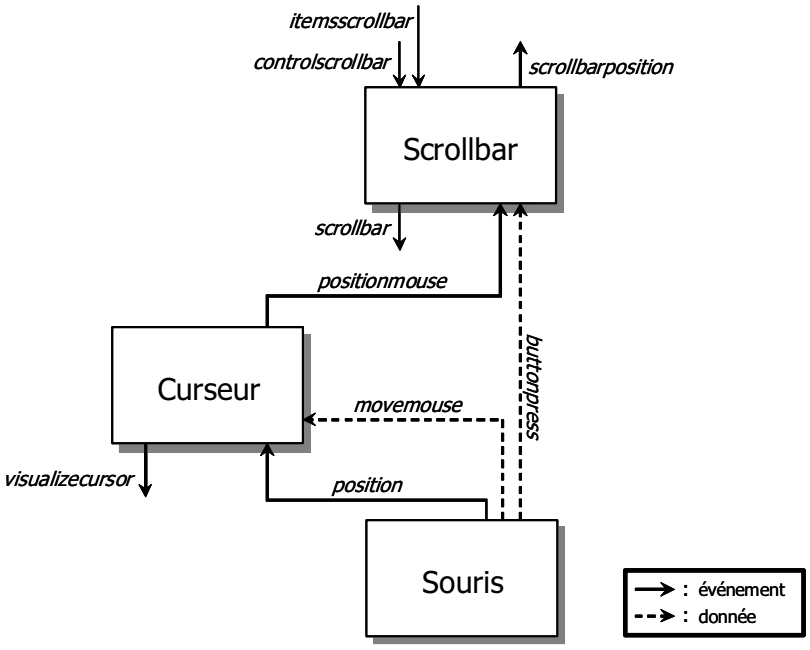

Fig. 1.16 – *Modélisation d'un scrollbar*, un exemple avec les interacteurs en boîte noire de CNUCE, adapté de [PF92].

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 négatifs sont à considérer :

- **Absence d'état observable** : comme l'a constaté [Gui95] 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 preuves 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]

Interacteurs de York. L'idée de modéliser un système interactif au moyen d'interacteurs a été reprise par les travaux de York. 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 implicitement décrit. Deux évolutions ont été proposées : les objets abstraits d'interaction (« Abstract Interaction Object ») [DH93] basés sur un modèle à états et une deuxième approche basée sur un modèle événementiel [DH95].

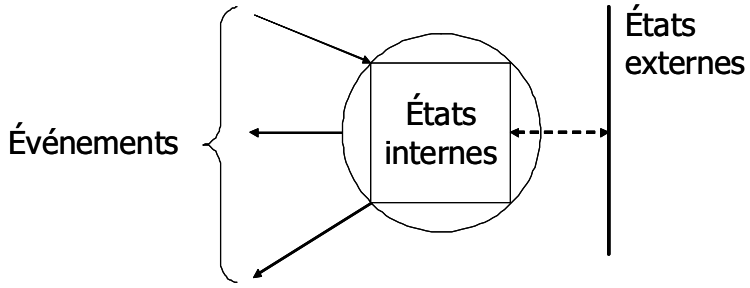

Fig. 1.17 – Schéma d'un interacteur de York.

Dans la version initiale [DH93], 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 1.17). 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 *externe* . Il peut aussi transmettre des événements de sortie, encore appelés des actions de sortie.

Un ensemble d'opérateurs de composition ont été définis. Ils correspondent aux opérateurs de l'algèbre des processus communiquants (synchronisation, renommage...). 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, [DH95] 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.

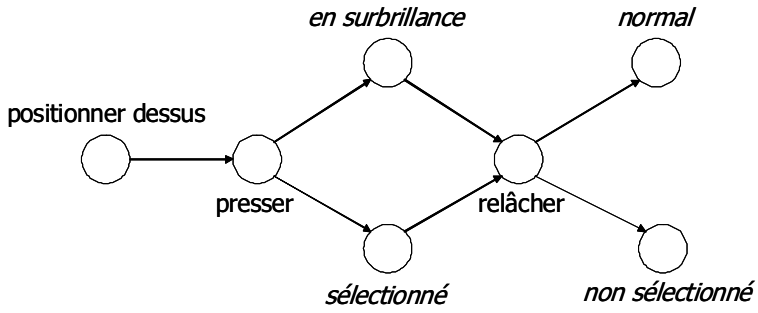

Fig. 1.18 – Comportement d'un bouton.

Nous donnons sur la figure 1.18 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 sa description formelle dans une 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. Ce dernier réduit le problème d'explosion combinatoire des états au moment de la composition d'interacteur de York.

Toutefois, différents points négatifs 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 décrite en section 1.2.6 ;
- un dernier point se situe au niveau de l'utilisation du langage Z. Ce langage s'avère difficile d'utilisation. Il est en effet plutôt cantonné au domaine de la recherche et son outil associé ne facilite pas l'aide au développement et à la construction de preuves.

Interacteurs du CERT. L'approche proposée par le CERT [Roc98] s'appuie sur le concept d'interacteur de York codé avec Lustre [HCRP91]. Cette nouvelle approche permet la description de systèmes interactifs à l'aide d'une approche à flots de données et permet de vérifier les propriétés de ces techniques par vérification sur modèles.

Lustre est un langage à *flots de données synchrones* (une horloge). Un système décrit en Lustre est représenté comme un réseau d'opérateurs (ou nœuds) agissant en parallèle. Les opérateurs transforment des événements d'entrée en événements 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*.

L'auteur a tout d'abord proposé une modélisation d'un interacteur de York dans le langage Lustre conforme à la figure 1.19. 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

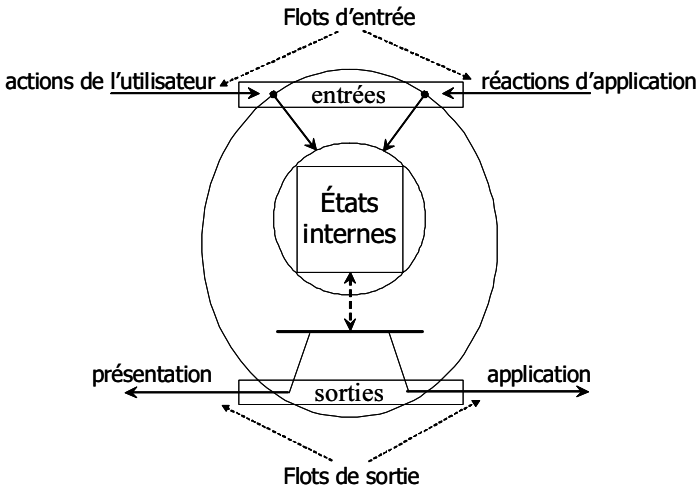

Fig. 1.19 – Interacteur à *flots de données*.

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.

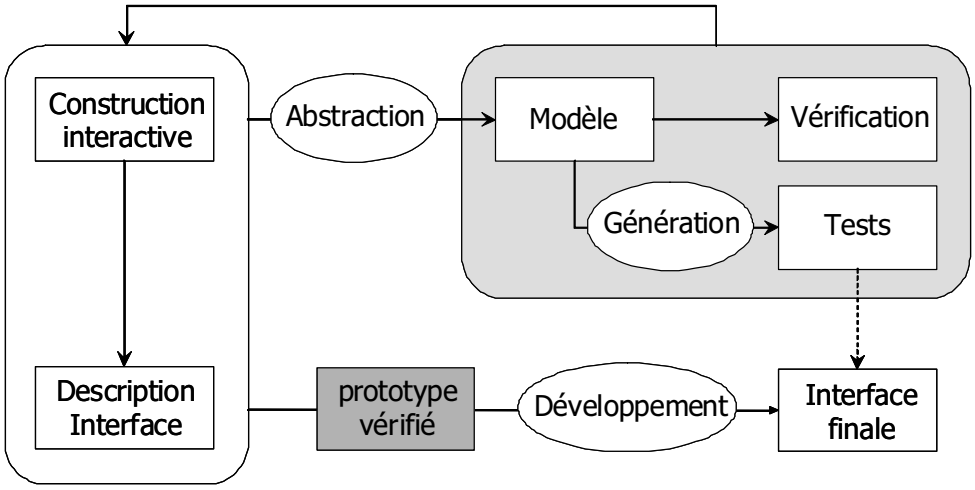

Fig. 1.20 – Approche à la fois expérimentale et formelle de [Roc98].

[Roc98] propose d'utiliser la modélisation d'un interacteur de Lustre dans une approche à la fois expérimentale et formelle, illustrée sur la figure 1.20. 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 (voir section 1.4.1.3)

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 est extrait automatiquement un modèle formel Lustre basé sur la composition d'interacteurs. 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 de preuves, 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'interacteur de York et du CERT pour répondre à deux besoins. Le premier concerne la compréhension des modèles Lustre par les utilisateurs et la deuxième s'intéresse au problème de l'explosion combinatoire inhérente au model checking.

1.3.3.2 Approche multi-niveaux : ICO

Des travaux ont été réalisés quant à l'utilisation de techniques formelles basées sur les réseaux de Petri afin de vérifier des propriétés sur les systèmes interactifs. [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 1.2.4.2), 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 par 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 fonction-

nel. Le dialogue est quant à lui décrit par l'ajout aux objets coopératifs de 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'activation 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.

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 preuve employée 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.

1.3.3.3 Modélisation modulaire du LISI

Dans un premier temps [AAGJ98a] et [AAGJ98b] 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 à un système interactif de type WIMP (Windows, Icons, Menus et Pointers), le Post-It Notes®[4]. À partir d'études de cas, 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 [AA00a] ont permis de représenter et vérifier avec B toutes les étapes de raffinement pour aboutir à une représentation dans un langage de programmation.

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

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 ascendantes de la famille des model checking, 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 preuves. De plus ces approches sont restées au niveau de l'écriture de spécifications et ne permettent pas, hormis l'approche de [Roc98], la génération de code.

Une description plus étoffée de cette première approche peut être trouvée dans le chapitre 2.4.1.

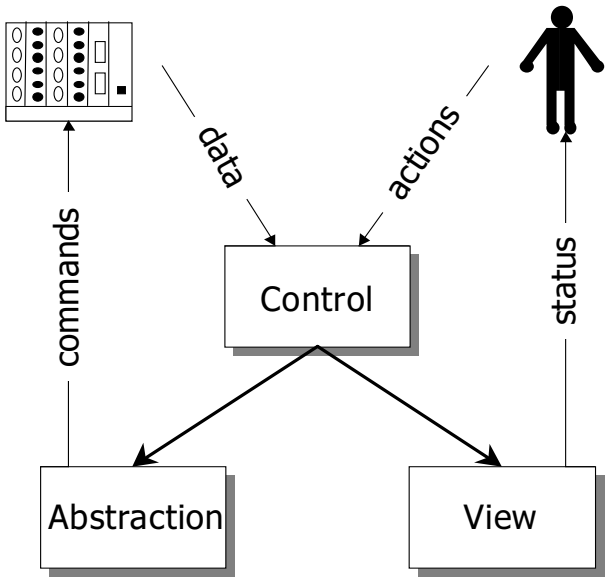

Fig. 1.21 – Le modèle d'architecture CAV *Control-Abstraction-View*.

Dans un second temps, des travaux se sont intéressés à la structuration des développements écrits en B selon une approche à objets [JGAA01], telle que MVC ou PAC. Toutefois, les contraintes du langage B ne permettaient pas d'utiliser directement les

modèles à agents. C'est pour cette raison qu'une nouvelle architecture hybride appelée CAV [Jam02] (Control-Abstraction-View) a été développée figure 1.21. À la différence des autres approches citées précédemment, ces travaux ne s'intéressent qu'à la partie du code dit *sécuritaire* (noyau fonctionnel) et non à la modélisation des IHM.

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 (section 1.4.1.1), la présentation, le dialogue, le noyau fonctionnel et la formalisation des propriétés. Contrairement à l'approche complémentaire des interacteurs du CERT [Roc98], 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. Il s'appuie sur une logique séquentielle décrite par un ensemble d'opérations. La description de processus concurrents n'est donc pas possible ;
4. les études de cas sont limitées aux systèmes WIMP.

Nous nous attacherons dans une partie de cette thèse aux points 1 et 3 de l'approche du LISI que nous présenterons dans le chapitre 2.

1.3.3.4 Conclusion sur les techniques formelles pour la conception des IHM

Les contributions passées en revue dans cette section 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 au pro-

cessus complet d'ingénierie des systèmes interactifs. En effet, les propriétés vérifiées ne couvrent qu'une partie des besoins et interviennent à des étapes distinctes du développement complet d'ingénierie. D'autre part, la place de l'utilisateur dans la modélisation (tâches utilisateur par exemple) n'est prise en compte que partiellement.

1.3.4 Bilan sur les techniques formelles pour la conception des systèmes interactifs

Dans cette section, nous avons exploré une classification des différentes techniques formelles et montré que les techniques basées sur la vérification de modèles permettaient de rendre exécutable des spécifications formelles, mais en contrepartie souffraient du problème d'explosion combinatoire des états.

A l'opposé, les techniques basées sur la preuve autorisent le découpage de la preuve au moyen du raffinement et la préservation de propriétés d'un raffinement à un autre, mais sont pénalisées par l'aspect semi-automatique de la preuve. Cependant, le processus de raffinement rend cette tâche moins fastidieuse.

De façon générale, les approches formelles pour la conception des interfaces homme-machine ne traitent pas la totalité du cahier des charges, sont hétérogènes et souffrent des inconvénients des systèmes de preuves employés (explosion combinatoire des états, spécifications non exécutables, etc).

1.4 Les outils de conception

L'augmentation de la complexité des interfaces homme-machine a conduit à accroître la quantité de code nécessaire à son développement. Rappelons d'après Myers [MR92] que le développement lié à l'interface représente 48% du code de l'application et monopolise 50% du temps imparti au processus de développement. Pour des raisons d'homogénéité de l'interface, de complexité (rendre accessible la programmation à tous) et d'économie (temps de réalisation), un grand nombre d'outils d'aide au développement ont été créés depuis une vingtaine d'années.

Ces outils se sont développés successivement, chaque étape s'appuyant sur les produits de l'étape précédente. [Pat99] a utilisé la métaphore de la *boîte gigogne* où chaque nouvelle boîte est conçue au dessus de la précédente formant plusieurs couches, figure 1.22. De ce fait, plus l'outil est de haut niveau, plus il offre des mécanismes évolués (vérification, aide au développement...) et plus la conception du système interactif est facilitée.

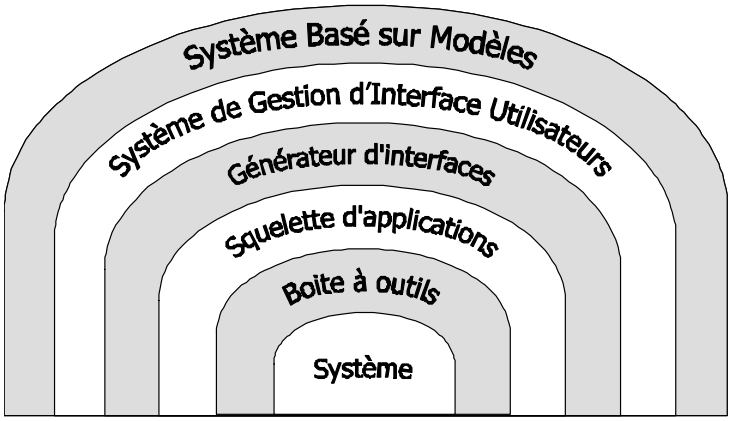

Fig. 1.22 – Famille des outils de développement.

Des études se sont intéressées à présenter les catégories d'outils [Cou90], [Pat99] et [FG01]. À l'image de [Tex00] nous nous sommes basés sur une description des outils suivant deux grandes approches :

- l'**approche ascendante** s'intéresse aux outils dont le noyau est la programmation de la couche présentation jusqu'à la description du dialogue de l'application ;
- l'**approche descendante** s'intéresse aux outils de description pour générer tout ou partie de la présentation et du contrôleur dialogue.

L'objectif de cette section est de décrire et comparer ces deux approches. Nous montrons aussi que l'approche descendante est plus appropriée pour répondre précisément aux problèmes de la vérification des propriétés d'un système interactif.

En conséquence, nous décrivons dans une première partie les outils de l'approche ascendante concernant les boîtes à outils, les squelettes d'applications et les générateurs d'interfaces. Puis, dans une seconde partie, nous présentons les approches descendantes et plus précisément les systèmes basés sur modèles.

1.4.1 Approche ascendante : boîte à outils, squelette d'applications et générateur d'interfaces

Le principe de l'approche ascendante est de fournir des outils pour construire la couche présentation au moyen d'éléments de présentation tels que des boutons, des fenêtres, des menus, etc. Ensuite, le concepteur relie les éléments d'entrée (réaction aux événements) et de sortie (attribut d'un élément) de la présentation avec des fonctions du noyau fonctionnel.

Ainsi, quand l'utilisateur interagit avec la couche de la présentation, des éléments du noyau fonctionnel sont modifiés par des fonctions *modifieurs* et l'état du noyau fonctionnel est retourné à la présentation par des fonctions *accesseurs*. Les éléments de la présentation appelés communément *widget* ou *composants graphiques* sont regroupés dans les boîtes à outils.

D'autres outils se sont basés sur les boîtes à outils pour aider à concevoir des interfaces. Les squelettes d'applications réalisent les fonctions usuelles de l'interface sous la forme d'un logiciel réutilisable et extensible [Cou90]. Il incombe au concepteur d'ajouter sur le squelette les éléments spécifiques à l'application finale. Le squelette est construit au dessus d'une boîte à outils et d'une architecture logicielle. Enfin, les générateurs d'interfaces permettent de construire l'interface de l'application de manière graphique tout en se basant sur un squelette d'application.

1.4.1.1 Boîtes à outils

Une boîte à outils est une bibliothèque d'objets d'interaction appelés widgets ou composants graphiques. Les boîtes à outils fournissent une interface de programmation (API[5]) orientée objet qui encapsule les données. Elles sont relativement nombreuses, pour ne citer que les principales Java/Swing[6], MFC (Microsoft Fundation Class)[7], QT[8], Tk [Ous94] ou AMULET [MMM+97]. L'interface de l'application est créée en combinant un ensemble de composants graphiques, tels que les fenêtres, les boutons, les menus, etc.

Comme nous l'avons vu dans la section précédente, les composants graphiques réagissent aux interactions de l'utilisateur. Dans ce but, les boîtes à outils utilisent le mécanisme d'événements (section 1.2.5.3) pour gérer le dialogue entre l'utilisateur et la présentation. Quand l'utilisateur interagit avec un composant graphique de la présentation, son action est transformée par la boîte à outils en événement(s). A la réception de(s) l'événement(s), le composant graphique réagit au travers de deux comportements :

- le comportement **interne** représente une réaction propre d'un composant graphique à un événement. Il n'est généralement pas modifiable et son rôle essentiellement esthétique n'a d'importance que dans la présentation. Par exemple, l'état d'un bouton *enfoncé* ou *non enfoncé* est modifié quand l'utilisateur appuie sur le bouton de la souris et que le pointeur se trouve dessus ;

[5]Application Program Interface : Interface de Programmation d'Applications
[6]Sun Microsystems : http ://www.sun.com
[7]Microsoft Corporation : http ://www.microsoft.fr
[8]TrollTech : http ://www.trolltech.com

– le comportement **externe** est associé au résultat que le concepteur souhaite obtenir quand le composant graphique subit une action de la part de l'utilisateur. Ce comportement doit être programmé par le concepteur. Pour cela, le concepteur utilise un mécanisme de traitement des événements qui diffère selon la boîte à outils employée : *boucle d'événement*, *fonctions de rappels* ou *abonnements*. Par exemple, dans le cas du traitement des événements par abonnement, utilisé dans la boîte à outils Java/Swing, la conception consiste à associer à chaque composant graphique émetteur d'événements un ou plusieurs objets récepteurs d'événements qui se chargeront d'appeler des éléments du noyau fonctionnel. Ainsi, les boîtes à outils permettent la description de la présentation mais aussi assurent le dialogue de l'application.

```
JButton mon_button = new JButton("OK")
mon_button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
            ref_nf.incrementeCompteur();
      }
});
```

Fig. 1.23 – Abonnement d'un composant graphique Swing à un écouteur.

L'exemple de la figure 1.23 décrit l'utilisation du traitement des événements par la technique d'abonnement. L'objet *mon_button* est créé sur la première ligne puis il est associé à un objet écouteur *ActionListener* sur la deuxième ligne. Suivant la nature de l'événement émis, la méthode *actionPerformed* appelle la méthode *incrementeCompteur* du noyau fonctionnel (*ref_nf*).

Une boîte à outils présente de nombreux avantages. Tout d'abord, elle permet de donner un style à l'interface et permet d'obtenir une homogénéité visuelle entre différentes applications. Ensuite, elle permet d'intégrer des critères ergonomiques dans les composants graphiques même si le concepteur ne possède pas de compétences en psychologie cognitive.

En contrepartie, les boîtes à outils souffrent de nombreux inconvénients :

– **difficulté d'utilisation** : il est nécessaire d'être informaticien et de connaître des langages de programmation pour utiliser les boîtes à outils. Cet inconvénient exclut de leur utilisation les personnes issues des domaines comme la psychologie, l'ergonomie et bien sûr l'utilisateur final. Le temps d'apprentissage est souvent long. [Cou90] compare les boîte à outils à *un grand sac de fonctions en vrac.*

– **absence de structuration explicite du code** : elle aboutit logiquement à des architectures logicielles floues. Il n'y a pas de réelle séparation entre le noyau fonctionnel et la présentation de l'application. La maintenance de l'application est donc rendue difficile.

– **difficulté de vérification** : afin de juger la notion d'utilisabilité, il est nécessaire de passer par de longues phases de tests exhaustifs. La relecture éventuelle du code de l'application pour vérifier les propriétés du système est difficile puisque le code contient à la fois des éléments de la présentation, du dialogue et du noyau fonctionnel.

1.4.1.2 Squelettes d'applications

L'utilisation des boîtes à outils amène le concepteur à programmer certaines séquences de code un grand nombre de fois, dans des applications différentes.

Il parait intéressant de coder une fois pour toutes les séquences sous forme d'une structure d'application interactive adaptable : c'est la notion de squelette d'application (*application frameworks* en anglais). Le rôle du concepteur est alors d'adapter les squelettes d'application à son développement en supprimant des portions de code inutiles et modifiant celles inadaptées en ajoutant des nouvelles fonctionnalités.

Les squelettes d'application permettent ainsi la factorisation de parties de code récurrentes et la diminution de l'effort de développement d'un facteur quatre à cinq par rapport aux seules boîtes à outils [Shu86]. Ils facilitent le développement et permettent au concepteur de se concentrer sur des fonctionnalités annexes telle que le dialogue de l'application ou les fonctionnalités. Ils garantissent aussi une homogénéité élevée des systèmes interactifs créés à partir des mêmes squelettes d'applications.

JAVA/Swing propose dans ce sens plusieurs squelettes d'application tant au niveau de la partie interface graphique que sur des aspects purement fonctionnels [FG01]. Par exemple, cette boîte à outils propose des squelettes d'application pour gérer le mécanisme *Défaire/Refaire*, de sauvegarde au moyen du mécanisme de la *serialization*, ou pour se connecter à d'autres sous-systèmes (base de données, serveurs HTTP ...). JAVA/Swing propose aussi des squelettes d'applications liés à des éléments de l'interface graphique comme des *boîtes de dialogue* prédéfinies.

En contrepartie, l'utilisation des squelettes d'applications amène les remarques suivantes :

– **compréhension du fonctionnement** : souvent très abstrait, les squelettes d'applications nécessitent de la part du concepteur une bonne connaissance de leur fonctionnement. Ils ne suppriment pas non plus la nécessité de connaître la boîte à outils sous-jacente, ce qui réserve leur utilisation aux informaticiens ;

– **limitation du domaine d'application** : les squelettes d'applications sont par nature stéréotypés, ce qui limite leur domaine d'application.

1.4.1.3 Générateurs de présentations

Les générateurs de présentation ou GUI-Builder (*Graphic User Interface Builders*) sont des outils qui permettent de créer facilement la couche présentation d'une application de façon graphique. Pour cela, le concepteur dessine son interface en utilisant des primitives graphiques représentant les widgets de la boîte à outils sous-jacente. Le concepteur visualise instantanément le résultat et, il lui est possible sous certaines limites de tester son interface construite comme par l'exemple dans l'outil GILT [MMM+97].

Employant généralement des techniques de programmation visuelle (manipulation directe des objets graphiques au moyen de la souris, visualisation des informations...), les générateurs de présentations sont faciles à utiliser et permettent d'obtenir rapidement des maquettes. Il n'est pas nécessaire d'être un spécialiste en programmation pour construire la couche présentation.

Les générateurs de présentations produisent un squelette d'application à partir de l'interface construite graphiquement par le concepteur. Il suffit d'ajouter la dynamique du dialogue et les appels aux fonctions du noyau fonctionnel pour construire l'application. Toutefois, cette étape doit être réalisée par un concepteur ayant des connaissances en programmation.

Les générateurs de présentation sont bien au point, et maintenant intégrés dans des produits commerciaux. Par exemple, Uim/x [Hal91] présenté dans la section précédente, Visual Studio[9], Delphi et JBuilder de Borland[10] figure 1.24, disposent de constructeurs d'interface directement intégrés à leurs environnements de programmation. Ces environnements intègrent de nombreux squelettes d'applications qui facilitent le développement de l'application interactive, par exemple l'utilisation d'assistants (*Wizzard* en anglais).

Cependant, les générateurs de présentation souffrent des inconvénients issus de la boîte à outils sous-jacente. Ils ne se limitent qu'à la couche présentation en permettant la représentation graphique des différents écrans et boîtes de dialogue d'une application, mais sans définir la succession de ces objets à l'écran (dynamique). Dans ce sens, l'outil BeanBuilder[11] associé à la technologie JAVA/Beans apporte une contribution dans ce domaine. En effet, il autorise une construction de la couche présentation et une ébauche d'un *pseudo* dialogue au travers des objets Beans.

[9]Microsoft Corporation : http ://www.microsoft.fr
[10]Borland Software Corporation : http ://www.borland.com
[11]Sun Microsystems : http ://www.sun.com

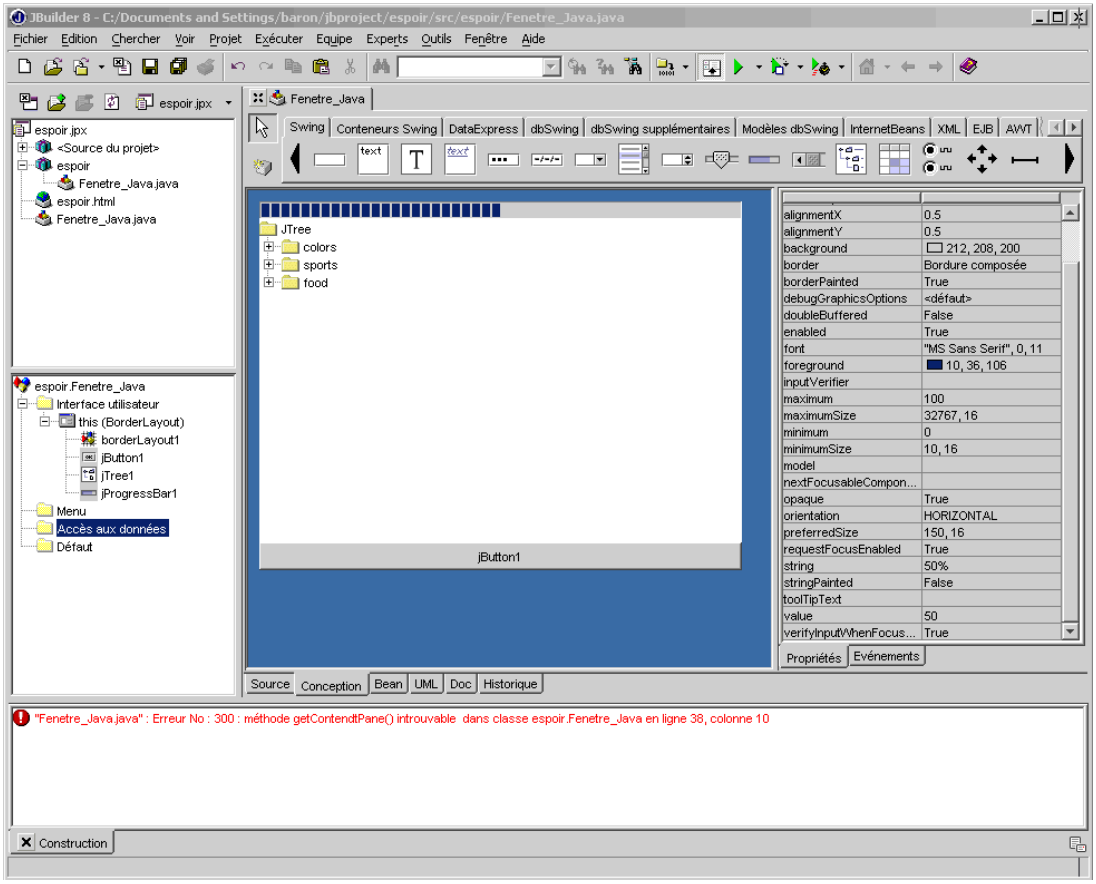

Fig. 1.24 – Exemple de GUI-Builder intégré dans l'environnement JBuilder.

1.4.1.4 Conclusion sur les approches ascendantes

Les outils de l'approche ascendante, à savoir les boîtes à outils, les squelettes d'applications et les générateurs d'interfaces, privilégient la couche présentation. Ils permettent de réaliser le dialogue et la liaison avec les éléments du noyau fonctionnel au fur et à mesure de la construction de l'interface.

La prise en compte des besoins utilisateurs et d'une architecture logicielle sont laissées au bon vouloir du concepteur avec l'inconvénient majeur de rendre plus difficile la vérification des propriétés du système interactif.

1.4.2 Approche descendante : Systèmes Basés sur Modèles

Le principe de l'approche descendante requiert des concepteurs, non plus une programmation informatique (codage de la présentation au moyen d'une boîte à outils), mais la

rédaction de spécifications dans un langage de haut niveau. Ces spécifications sont alors traduites ou interprétées pour générer totalement ou partiellement la couche présentation.

Dans cette section, nous définissons plus précisément les approches descendantes et en particulier, les outils qualifiés de systèmes basés sur modèles. De la même façon que dans l'étude réalisée sur les approches formelles dans les IHM (section 1.3.3), nous décrivons un échantillon de ces outils en donnant leurs avantages et leurs inconvénients.

1.4.2.1 Principes de base

Les systèmes fondés sur l'approche descendante se sont développés au cours des années 80 et 90 parallèlement aux approches ascendantes. Les outils utilisant cette approche se basent sur des langages spécialisés de haut-niveau d'abstraction et sont appelés Système de Gestion d'Interface Utilisateur (SGIU ou UIMS : User Interface Management Systems). Au début des années 90, ils ont évolué, prenant en compte de plus en plus d'éléments, issus aussi bien du dialogue que de la présentation, et permettant de générer des interfaces de plus en plus riches et de plus en plus complexes. Les systèmes actuels manipulent différentes descriptions : celle des tâches utilisateur, celle de la structure et des relations des informations manipulées par l'application, celle de la couche de présentation, etc.

L'expression « Systèmes Basés sur Modèles » ou MBS (Model Based Systems) représente l'évolution des UIMS et fait référence à un ensemble d'outils qui permettent de construire l'interface de l'application au moyen d'un ou plusieurs modèle(s) (modèle de tâches, modèle de l'utilisateur, etc). Les systèmes basés sur modèles sont de véritables environnements de développement disposant d'outils multiples pour fournir une assistance dans le processus de développement d'une application interactive.

Même si les systèmes basés sur modèles sont beaucoup plus sophistiqués que les outils de l'approche ascendante, ils demeurent toutefois aujourd'hui confinés aux études de recherche. Ils ne s'imposent pas face aux outils commerciaux de l'approche ascendante.

D'après [Sze96], la figure 1.25 montre les composants classiques qui se trouvent dans les systèmes basés sur modèles. Les rectangles arrondis représentent les différents outils, tandis que les autres rectangles représentent les informations produites ou consommées par les outils.

Les principaux composants d'un environnement MBS sont le modèle, les outils de modélisation, les outils de conception automatisée, les outils d'implémentation et les outils de validation.

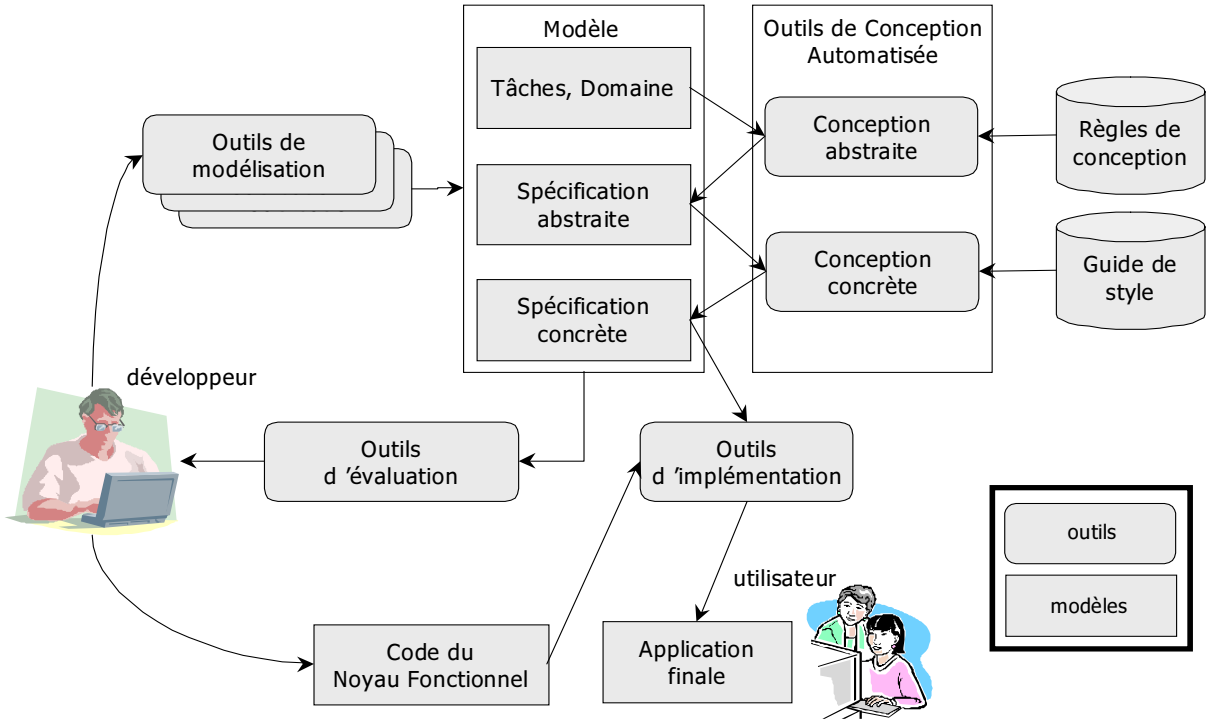

Fig. 1.25 – Environnement de développement d'interfaces à base de modèles, adapté de [Sze96].

Le concepteur utilise les outils de modélisation pour créer le modèle. Comme indiqué plus haut, ce modèle peut être constitué d'un ensemble de modèles particuliers (données, tâches...).

Les outils de conception automatisée sont utilisés pour effectuer des transformations sur le modèle. Les outils d'implémentation gênèrent le code de l'interface qui, une fois liée avec le code du noyau fonctionnel, devient exécutable puis exploitable par les utilisateurs finaux.

Modèle. Le modèle est le principal composant d'un système basé sur modèles. Il représente une description des caractéristiques de l'interface d'une application interactive. Il peut donc contenir des notions de dialogue, de domaine et de présentation. Le modèle peut globalement être décomposé en trois niveaux d'abstraction [Sze96] :

1. Au plus haut niveau, nous trouvons le **modèle de tâches** et le **modèle du domaine**. Le modèle du domaine représente les données et les fonctions (accesseurs et modifieurs) supportées par l'application. Le modèle de tâches représente les tâches que l'utilisateur doit pouvoir effectuer sur l'application. Il est décomposé en une

hiérarchie de tâches et sous-tâches pour correspondre à un arbre de tâches à la manière des notations HTA, MAD ou CTT vues dans la section 1.2.3. Les feuilles de cet arbre correspondent à des opérations directement réalisables avec l'application.

2. Le second niveau du modèle est appelé **spécification abstraite** de l'interface utilisateur et représente la structure générale de l'interface de l'application. Elle est décrite en termes d'objets d'interaction de bas-niveau (sélection d'un élément dans un ensemble), d'éléments d'information (valeurs, ou ensemble de valeurs du domaine, constantes...) et d'unités de présentation (abstractions des fenêtres). La spécification abstraite décrit la manière d'afficher les informations dans chaque fenêtre et la façon d'interagir avec ces informations.

3. Le troisième niveau du modèle est la **spécification concrète** de l'interface utilisateur. Elle concrétise les éléments du deuxième niveau d'abstraction (objets d'interaction, éléments d'information et unités de présentation) au moyen d'éléments issus d'une boîte à outils (boutons, boîtes à cocher,...).

Outils de modélisation. Les outils de modélisation servent à assister le concepteur dans la construction des modèles. Le but de ces outils est, d'une part, de cacher aux concepteurs la syntaxe des langages de modélisation, et d'autre part, de fournir aux développeurs une interface conviviale pour faciliter la description des spécifications contenues dans les modèles. Les outils sont généralement dédiés à un modèle particulier. Ces outils peuvent être des éditeurs de textes pour construire les spécifications textuelles du modèle comme dans ITS [WBB+90] ou MASTERMIND [SSC+95], ou encore des formulaires de création d'éléments du modèle MECANO [Pue96].

Outils de conception automatisée. Les outils de conception automatique permettent au concepteur de spécifier certains aspects de l'interface seulement. Ils sont capables de déduire des spécifications fournies et non finalisées les aspects manquants et de générer de nouvelles spécifications. Par exemple, avec Janus [BHKN96], le concepteur ne décrit que le modèle du domaine, les autres niveaux du modèle, la spécification abstraite et la spécification concrète de l'interface sont générés automatiquement par les outils de conception automatique. Toutefois, tous les systèmes basés sur modèles ne possèdent pas d'outils de conception automatisée (ITS et MASTERMIND par exemple). Ils nécessitent de la part du concepteur de spécifier explicitement tous les niveaux du modèle.

Ces outils permettent de diminuer le travail des concepteurs, du fait qu'ils ont moins d'éléments à écrire. Ces outils utilisent souvent des règles de conception et/ou des guides de styles, détaillés dans [Van99]. Cependant, l'utilisation de tels outils réduit l'éventail des

domaines d'application qu'il est possible de créer. Pour reprendre l'exemple de JANUS, l'outil peut uniquement générer des interfaces pour les bases de données.

Outils d'implémentation Les outils d'implémentation traduisent la spécification concrète de l'interface en une représentation qui peut être directement utilisée par une boîte à outils ou un générateur d'interfaces. Il existe trois solutions possibles d'implémentation :

- l'outil d'implémentation produit du code utilisant une boîte à outils spécifique, MASTERMIND ou JANUS par exemple ;
- les générateurs de langages de description d'interface UIMS produisent une description de la présentation pouvant être lue par un générateur d'interface, FUSE [LS96] par exemple ;
- un système basé sur modèles peut mettre en application un interprète, qui est responsable de l'exécution des modèles, comme ITS. L'interprétation des modèles offre l'avantage de pouvoir tester rapidement l'interface construite sans avoir besoin de passer par de longues phases de compilation.

Outils de validation A la différence des approches ascendantes, les systèmes basés sur modèles disposent d'outils de validation. Ces derniers s'intéressent à l'analyse et à l'évaluation des modèles disposant de riches descriptions. Ensuite, c'est au concepteur de prendre en compte les résultats de validation pour modifier au moyen des outils de modélisation les modèles incohérents. Ainsi, ces outils de validation permettent de vérifier a posteriori des propriétés de l'application interactive. Par exemple, il est possible de vérifier la possibilité d'atteindre toutes les fonctionnalités du système interactif (propriétés d'atteignabilité).

Cependant, la majorité de ces outils de validation n'intervient qu'au stade final de la modélisation. En effet, ces outils ne s'intéressent pour la plupart qu'au niveau de la spécification concrète du modèle pour en établir des critiques et des évaluations.

Nous proposons d'approfondir dans la suite de cette section les problèmes liés à cette validation du modèle au moyen de l'étude d'approches MBS. Puisque la recherche dans le domaine des systèmes basés sur modèles est très importante, nous limitons notre étude à quelques systèmes que nous pensons les plus représentatifs de ce problème de validation (MOBI-D [PE98], MASTERMIND [SSC+95], TERESA et PETSHOP [Nav01]). Cette étude est basée à la fois sur le positionnement des systèmes basés sur modèles face à l'architecture générique de la figure 1.25, puis sur les avantages et/ou inconvénients des outils de validation.

Il est à noter que l'expertise de ces systèmes a été réalisée en globalité d'après la lecture d'articles puisque nous ne disposions pas directement des prototypes pour les évaluer. Seules les prototypes de Teresa et de Petshop ont pu être réellement testés.

Les lecteurs intéressés par une étude plus exhaustives sur l'approche des MBS peuvent consulter les références suivantes [Tab01], [PdS00] et [Cal98].

1.4.2.2 MOBI-D

MOBI-D [PE98] s'inscrit dans la continuité du projet MECANO [Pue96]. Il définit un processus de conception qui permet de construire des IHM plus expressives (applications militaires, médicales, etc) que les simples interfaces de types formulaires obtenues par MECANO.

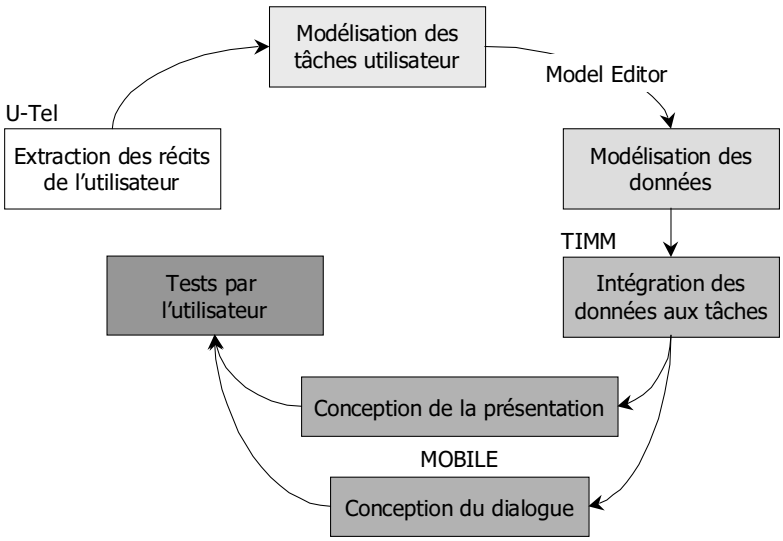

Fig. 1.26 – Cycle de développement dans MOBI-D.

La figure 1.26 montre le cycle itératif de développement d'une application interactive au moyen de l'environnement MOBI-D. Dans un premier temps, le concepteur effectue une collecte des informations sur la façon dont les utilisateurs accomplissent une activité. L'acquisition de ces informations est obtenue par les récits des utilisateurs et rédigée au moyen de l'outil *U-Tel*.

Dans un deuxième temps, les points essentiels de l'analyse précédente sont utilisés pour construire le modèle de tâches et le modèle du domaine. Cette construction est effectuée au moyen de l'outil d'édition *Model editor*. Le concepteur définit successivement le modèle de tâches (création de tâches élémentaires, de sous-taches...), le modèle du

domaine (création des types...) puis définit l'intégration des éléments du modèle de tâches aux éléments du modèle du domaine. A chaque intégration, l'outil crée une relation de conception. L'ensemble de toutes les relations de conception constitue la spécification abstraite de l'interface du modèle.

Dans un troisième temps, l'outil *TIMM* (The Interface Model Mapper) propose d'automatiser la création de l'interface par des mécanismes de décisions basés sur un ensemble de règles pré-établies ou éditées par le concepteur. Cet outil génère automatiquement pour chaque relation de conception un choix de composants graphiques. La visualisation de ces composants désigne le modèle de présentation tandis que le comportement décrit le modèle du dialogue.

Un dernier outil appelé *MOBILE* permet au concepteur de construire l'interface concrète de l'application au moyen des composants graphiques proposés pour chaque relation de conception. C'est au concepteur de choisir parmi ces composants graphiques ceux qu'il souhaite utiliser. Par exemple, si une tâche de l'utilisateur est *saisir un nombre*, MOBILE lui propose : un champs de saisie ou un dispositif de variation (rangeslider en anglais). Le concepteur intègre, s'il le souhaite, ces composants graphiques au moyen du générateur de présentation intégré à *MOBILE*. Il peut alors modifier les attributs du composant et l'agencement global de l'interface de l'application. L'outil *TIMM* et *MOBILE* permettent donc de construire le niveau de la spécification concrète (présentation et dialogue) de l'interface du modèle.

Nous proposons sur la figure 1.27, l'architecture de l'environnement de développement MOBI-D suivant celle de la figure 1.25. On notera l'absence d'outils pour la conception automatique de la spécification abstraite et pour la validation des modèles (rectangle en pointillé). Enfin, MOBI-D permet de générer l'application finale pouvant être utilisée directement par les utilisateurs finaux.

De façon plus générale, le projet MOBI-D offre un atelier logiciel utilisable à différents niveau de corps de métiers (ergonomie, psychologie, création d'interface...). Il facilite la conception puisqu'il s'affranchit des langages de programmation classiques. L'utilisation du méta-langage *MIMIC* pour décrire les composants, les structures et l'organisation des modèles d'interface permet de lier rapidement et sans transformation différentes interfaces. Enfin, cette approche offre l'avantage de pouvoir personnaliser l'interface par rapport aux besoins des utilisateurs.

Toutefois, malgré ces nombreux avantages, nous dénombrons certains inconvénients de l'approche MOBI-D :

– la validation de l'interface est reléguée au dernier niveau du modèle puisque cette

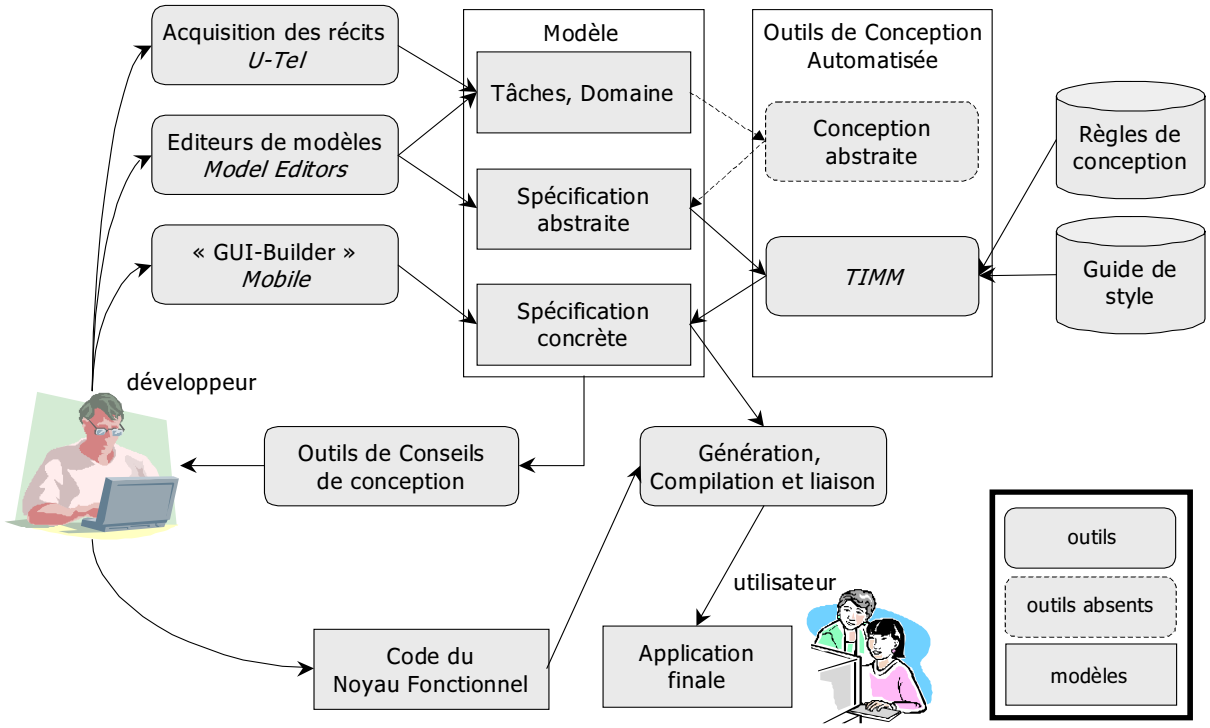

Fig. 1.27 – Architecture générique de l'environnement MOBI-D.

approche ne permet pas de tester la modélisation en cours de conception. De plus, MOBI-D n'offre pas d'outils de validation qui permettraient de vérifier a priori l'atteignabilité d'une tâche. Il s'agit donc d'un outil de conception ;

– il est nécessaire de passer par des phases de compilation. Le noyau fonctionnel de l'application doit être compilé avec l'interface générée, ce qui ralentit le processus de développement. Cette approche ne permet pas non plus au concepteur d'alterner phase de développement et phase de test sans perdre le contexte d'exécution.

1.4.2.3 MASTERMIND

MASTERMIND [SSC+95] (Models Allowing Shared Tools and Explicit Representations to Make Interfaces Natural to Develop) se base sur les travaux des systèmes basés sur modèles HUMANOID [SLN93] et UIDE [FS94]. Alors que le premier met l'accent sur le modèle de présentation et d'outils de modélisation, UIDE s'intéresse plutôt au modèle du dialogue, aux outils de conseils et de critiques de conception et de génération d'aide.

La figure 1.28 montre l'architecture de conception de l'environnement MASTERMIND. Les modèles sont décrits au moyen du modèle objet de CORBA (Common Object Request

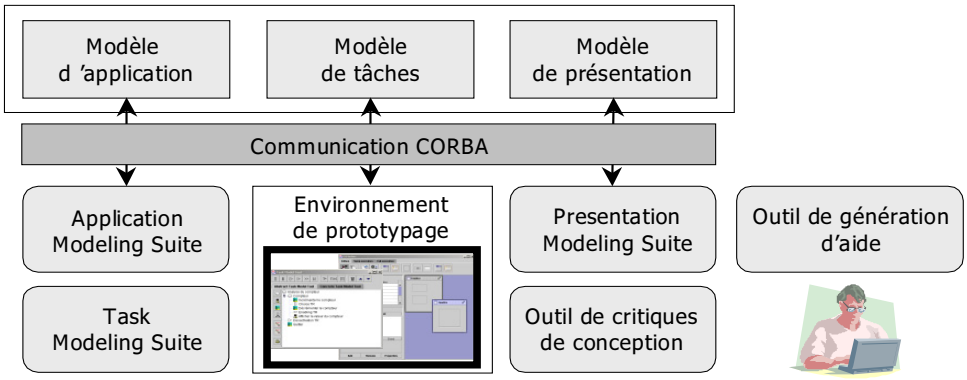

Fig. 1.28 – Rôle de la communication CORBA dans l'environnement MASTERMIND.

Broker Architecture) et fonctionnent dans un processus séparé via une couche de communication appelée serveur de modèle. Les outils de modélisation et l'environnement de prototypage sont donc vus comme des clients qui peuvent accéder aux modèles en lecture et en écriture. Les évolutions des modèles (suite à une modification de la part d'un outil par exemple) sont systématiquement propagées vers l'ensemble des autres outils. En particulier, l'environnement de prototypage qui génère une interface exécutable à partir des modèles est aussi mis à jour. Cette approche permet au concepteur de visualiser instantanément le résultat des modifications. Enfin, même si les modèles ne sont pas finalisés, l'environnement est capable de combler les spécifications manquantes afin de produire une interface exécutable.

Cette approche est différente à celle de MOBI-D. Ce dernier obligeait le concepteur à finaliser la modélisation pour en générer une interface exécutable.

MASTERMIND est basé sur trois modèles :

- le modèle de l'application est une extension du modèle objet de CORBA. Il complète ce dernier par des pré-conditions et des notifications. Ce modèle est décrit au moyen de l'outil *Application Modeling Suite* ;
- le modèle de tâches décrit les tâches utilisateur. Le concepteur modélise au moyen de l'outil *Task Modeling Suite* la structure hiérarchique du modèle de tâches afin de traduire les intentions de l'utilisateur sur l'application et le comportement de l'interface face aux actions de cet utilisateur. Une tâche est définie à partir de trois catégories (tâche utilisateur, tâche présentation et tâche application). Les tâches utilisateurs de plus bas niveau dans l'arbre de tâches sont des techniques d'interaction (clicker sur un bouton) tandis qu'une tâche présentation désigne une modification d'un attribut de l'interface. Ainsi, le modèle de tâches définit le modèle du dialogue. Pour finir, la sémantique du modèle de tâches de MASTERMIND s'inspire fortement de celle

de GOMS. D'après les auteurs, il est possible d'évaluer les performances du système en termes d'entrée utilisateur. Mais cette caractéristique n'a pour l'instant pas été implémentée ;

– le modèle de présentation définit l'apparence visuelle de l'interface. Il s'exprime en terme d'éléments de présentation où chaque terme est défini par des attributs spécifiques. A l'inverse de MOBI-D, il n'y a pas de réelle distinction entre les spécifications abstraite et concrète. Le modèle de la présentation est construit à l'aide de l'outil *Presentation Modeling Suite* qui reprend les techniques employées par les générateurs de présentation.

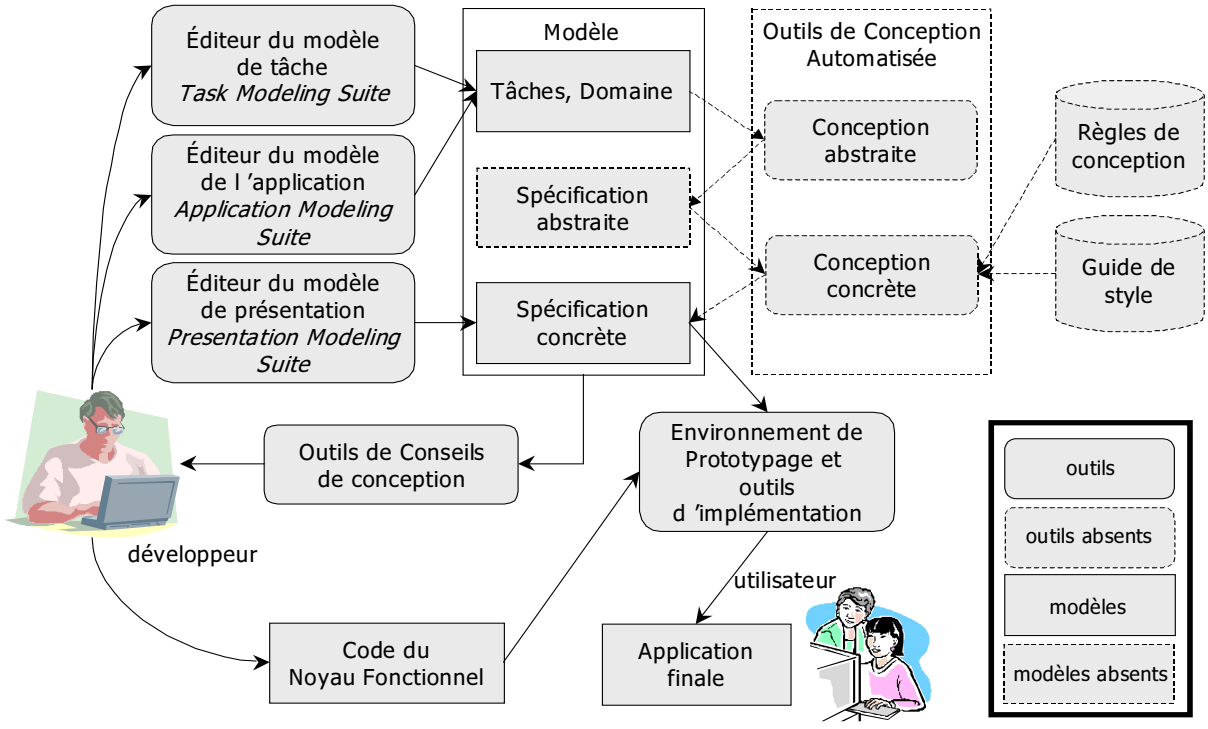

Fig. 1.29 – Architecture générique de l'environnement MASTERMIND.

Nous proposons à partir de cette étude, l'architecture de développement de MASTERMIND, figure 1.29 positionnée par rapport à celle de la figure 1.25. A la différence de MOBI-D, il n'y a pas dans MASTERMIND d'outils de conception automatisée, ni de règles de conception, ni de guide de style, ni de niveau abstrait pour le modèle (rectangle en pointillé), tout est à la charge du concepteur.

Différentes réserves de l'approche MASTERMIND sont à considérer :

– l'absence de spécification abstraite ne permet pas de vérifier au plus tôt la cohérence entre les objets du domaine et les tâches du modèle de tâches ;

- MASTERMIND utilise des formalismes propriétaires développés pour cette approche. Il redéfinit un modèle de tâche qui pourrait ressembler aux modèles étudiés dans la section 1.2.3. Ce point est particulièrement gênant puisqu'il impose l'apprentissage de nouveaux formalismes ;
- la pauvreté de la sémantique utilisée dans le modèle (utilisations de spécifications informelles) ne permet pas d'effectuer des raisonnements et ainsi de valider des propriétés comme la robustesse.

1.4.2.4 TERESA

L'approche TERESA [MPS03, PS02] (Transformation Environment for inteRactivE Systems representAtions) s'intéresse au problème de la génération d'interfaces pour des applications multi-supports (téléphone portable, assistant personnel ou ordinateur de bureau). En effet, suivant le support considéré, différentes considérations sont à prendre en compte pour assurer l'utilisabilité d'une application interactive : taille de l'écran, absence de multi-fenêtrage, système d'exploitation et boîte à outils.

Cette approche se base alors sur un ensemble de modèles et d'outils pour construire une interface adaptée au support (figure 1.30). Il est à noter dans l'approche TERESA qu'il s'agit avant tout d'un outil qui ne s'occupe que de la couche graphique. C'est à la charge au concepteur d'intervenir pour effectuer la liaison entre les objets du noyau fonctionnel et ceux de l'interface qui ont été générés (croix sur l'utilisateur).

La modélisation débute par l'établissement d'un modèle de tâches qui s'appuie sur le formalisme CTT [Pat01] décrit dans la section 1.2.3. A partir de ce modèle de tâches et des descriptions faites sur les objets des tâches (description textuelle du rôle de la tâche), un modèle du domaine est aussi établi. Ces modèles sont construits par l'intermédiaire de l'outil CTTE [PMG01]. L'utilisation de cet outil, rappelons le, permet aussi la simulation de modèle de tâches afin de vérifier la cohérence et le bon déroulement des tâches pour éventuellement proposer des optimisations. Enfin, c'est par l'intermédiaire du modèle de tâches CTT (opérateurs temporels) que le dialogue du système interactif est décrit.

Une spécification abstraite de l'interface utilisateur appelée *AUI* (Abstract User Interface) est obtenue en analysant le modèle de tâches et le modèle du domaine au moyen de l'outil TERESA. Cette phase est automatique ; l'outil identifie différentes structures du modèle de tâches liées aux opérateurs temporels et aux objets contenus dans les tâches pour définir les objets d'interaction abstraits. Par exemple, deux tâches interactions T_1 et T_2 dont le rôle est de saisir un nombre (objet spécifique associé à cette description), activables de façon indépendante (opérateur d'ordre indépendant) et ayant la même tâche mère, leur représentation est donnée par deux champs de textes groupés.

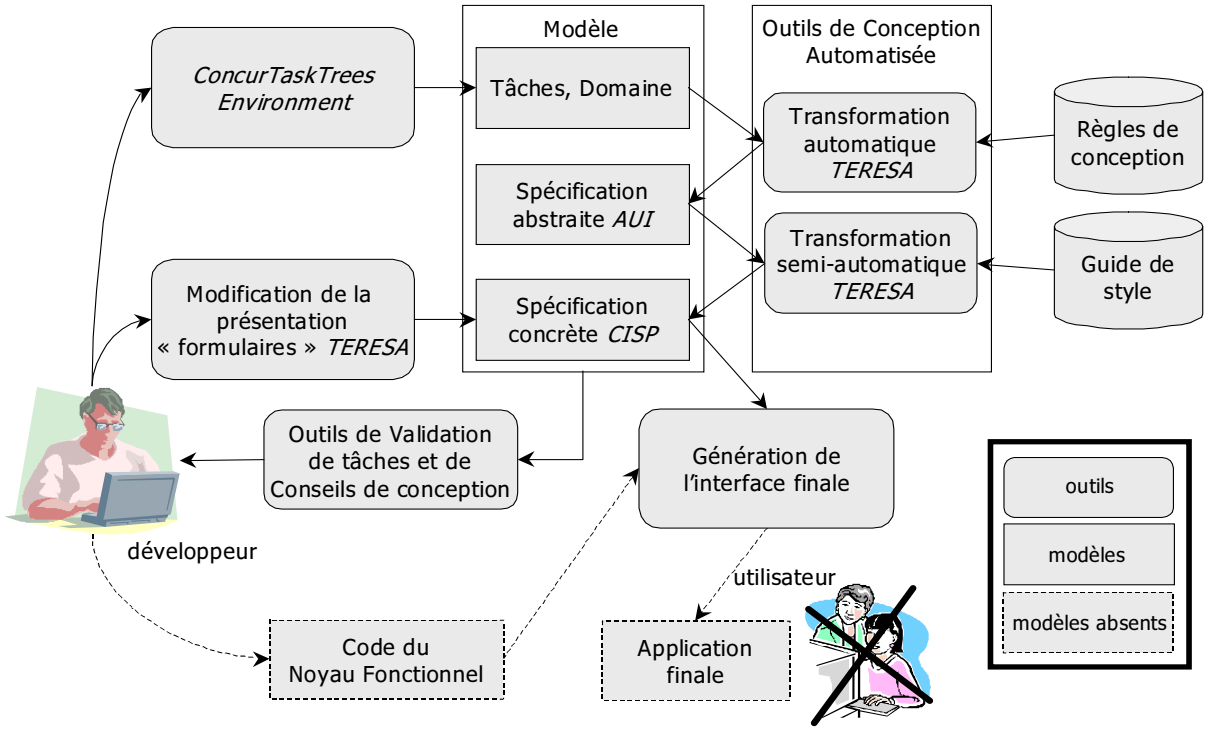

Fig. 1.30 – Architecture générique de l'environnement TERESA.

Enfin, le dernier niveau définit le modèle de spécification concrète appelé CISP (Concrete Interface for the Specific Platform) est dépendante de la plate-forme et doit prendre en compte les propriétés spécifiques du système cible. En effet, les éléments de l'interface concrète doivent satisfaire les exigences de l'affichage, des composants graphiques, du système d'exploitation et de la boîte à outils. Cette phase de modélisation de l'interface concrète n'est pas entièrement automatique. TERESA génère une première interface concrète qui reste modifiable par le concepteur. L'outil propose un ensemble de composants graphiques adaptés à chaque tâche interaction. Au concepteur de choisir la représentation qu'il désire. TERESA, se comporte donc comme un outil de conseils un peu à la manière de MOBI-D.

La liberté laissée à l'utilisateur est limitée. L'outil ne propose pas de générateur de présentation. Les modifications sont apportées à certains attributs des composants graphiques (couleur du texte, image, etc) et se font dans un formulaire, figure (1.31).

L'étape de génération de l'interface consiste en la transformation de la spécification concrète en un prototype conforme à la plate-forme utilisée. L'interface ainsi générée correspond aux attentes définies dans le modèle de tâches abstrait.

Toutefois, certaines réserves de l'approche TERESA sont à considérer :

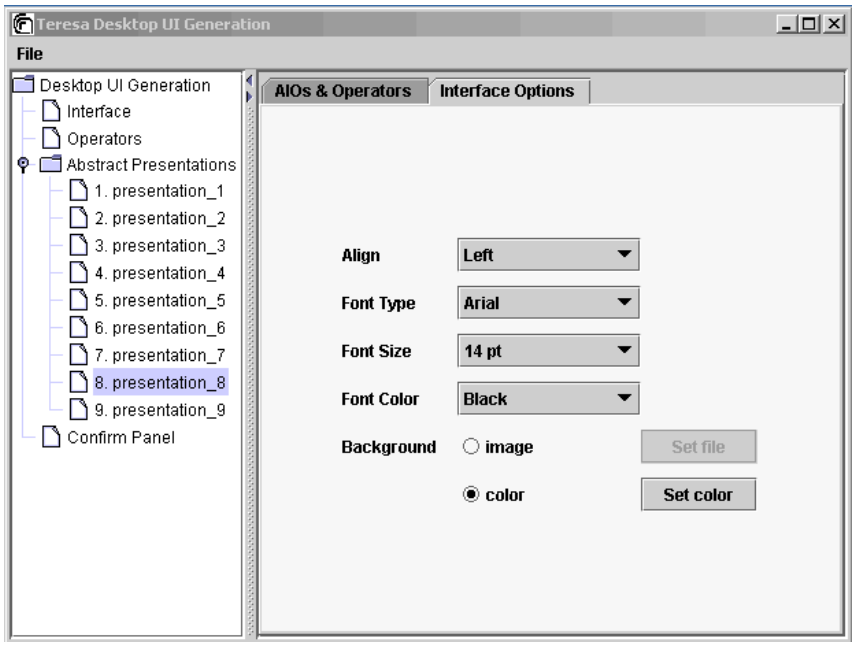

Fig. 1.31 – Formulaire de modification des éléments contenus dans la spécification concrète d'un modèle de TERESA.

- limitation à la couche présentation. L'outil TERESA ne génère que l'interface et le dialogue associé aux composants de cette interface. La liaison avec le noyau fonctionnel doit être faite manuellement, un peu à la manière des outils de l'approche ascendante, section 1.4.1 ;
- la validation du modèle de tâches CTT. Ce point s'adresse directement à l'outil CTTE. Ce dernier ne permet que de vérifier la cohérence et le bon déroulement des tâches. Aucun raisonnement n'est donc possible puisque les objets des tâches ne décrivent que textuellement (pas de sémantique) les objets du noyau fonctionnel ;
- la modélisation du dialogue. L'outil TERESA se base sur la modélisation de tâches et plus concrètement sur les relations temporelles et structurelles du modèle de tâches comme modèle du dialogue. Cette approche est donc principalement adaptée aux interfaces de type formulaires.

1.4.2.5 PETSHOP

Nous avons vu pour l'instant que les approches basées sur modèles employaient des modèles à sémantique non formalisée. Le point de départ de toute modélisation était le modèle de tâches et du domaine, puis chaque modèle était construit suivant le précédent. PETSHOP [Nav01, Ous01] propose une approche différente, dans le sens où le système

interactif (domaine, dialogue et présentation) et le modèle de tâches sont décrits de façon indépendantes. C'est à la fin de la modélisation des modèles que la collaboration entre la description de la tâche et celle du système peut être faite.

Dans l'étape d'analyse, le modèle de tâches est construit dans la notation CTT à l'aide de l'éditeur CTTE. Une première vérification est effectuée au moyen de l'outil de simulation.

La modélisation du système est réalisée au moyen de l'outil PETSHOP. C'est un éditeur-interpréteur d'ICO, formalisme que nous avons étudié dans la section 1.3.3. Il permet de spécifier le modèle du domaine au travers d'un ensemble de classes d'objets coopératifs appelé les *CO-classes*.

Le comportement d'un objet des CO-classes est décrit au moyen d'un réseau de Petri appelé *ObCS* (Object Control Structure). L'ensemble des ObCS décrit le système. Le concepteur peut évaluer chaque instance d'une CO-classes appelée *CO-instance*, par l'intermédiaire de l'outil PETSHOP. Il peut éditer, exécuter et analyser son réseau de Petri sans perte de contexte. Par exemple, de nouvelles places et transitions peuvent être ajoutées, modifiées ou supprimées à tout moment pendant l'exécution des CO-instances.

La liaison entre le modèle de tâches et le modèle du système est assurée par un éditeur de correspondance. Ce dernier extrait, dans un premier temps, l'ensemble des tâches interactives qui doivent être exécutées sur le système, puis l'ensemble des services utilisateur de la spécification du système. A partir de cette extraction, le concepteur effectue la mise en relation d'une tâche interactive avec un service utilisateur. Cette étape peut être désignée comme la spécification abstraite de l'interface du modèle.

Le niveau spécification concrète, dans l'approche PETSHOP est représenté par le prototype du système. Il est obtenu en définissant le lien (fonction d'activation) entre l'ensemble des objets de la présentation et des ObCS. Ces éléments de la présentation sont obtenus par l'intermédiaire du générateur de présentation JBuilder. Toute modification apportée aux ObCS (modification d'une transition par exemple) est directement perceptible sur l'interface à la manière de MASTERMIND.

L'avantage de la liaison avec un modèle de tâches CTT est de pouvoir rejouer des scénarii (trace de tâches à réaliser) afin de vérifier si le prototype est conforme aux attentes de l'utilisateur.

Nous résumons sur la figure 1.32, l'architecture de l'environnement de développement PETSHOP suivant celle de la figure 1.25. PETSHOP est avant tout un outil de validation et ne propose pas d'outils pour la conception automatique de la spécification concrète, ni de règles de conception et ni de guide de style. Il permet de tester l'application en cours de

développement mais ne propose pas la génération qui permettrait aux utilisateurs finaux d'utiliser directement le système (croix sur les utilisateurs).

Toutefois, certains points négatifs sont à considérer dans l'approche PETSHOP :

- la déconnexion entre les objets de l'application et de la présentation avec les tâches. La liaison entre les objets de l'application et des tâches est réalisée manuellement à la fin de chacune des deux modélisations avec le risque d'une liaison erronée. De plus, le test ne peut être réalisé que si cette liaison est terminée ;
- un processus de validation long. Ce point est une conséquence du précédent. Si la validation des tâches met en lumière une erreur dans la conception du système, elle nécessite d'itérer de nouveau le processus que nous avons décrit ci-dessus.

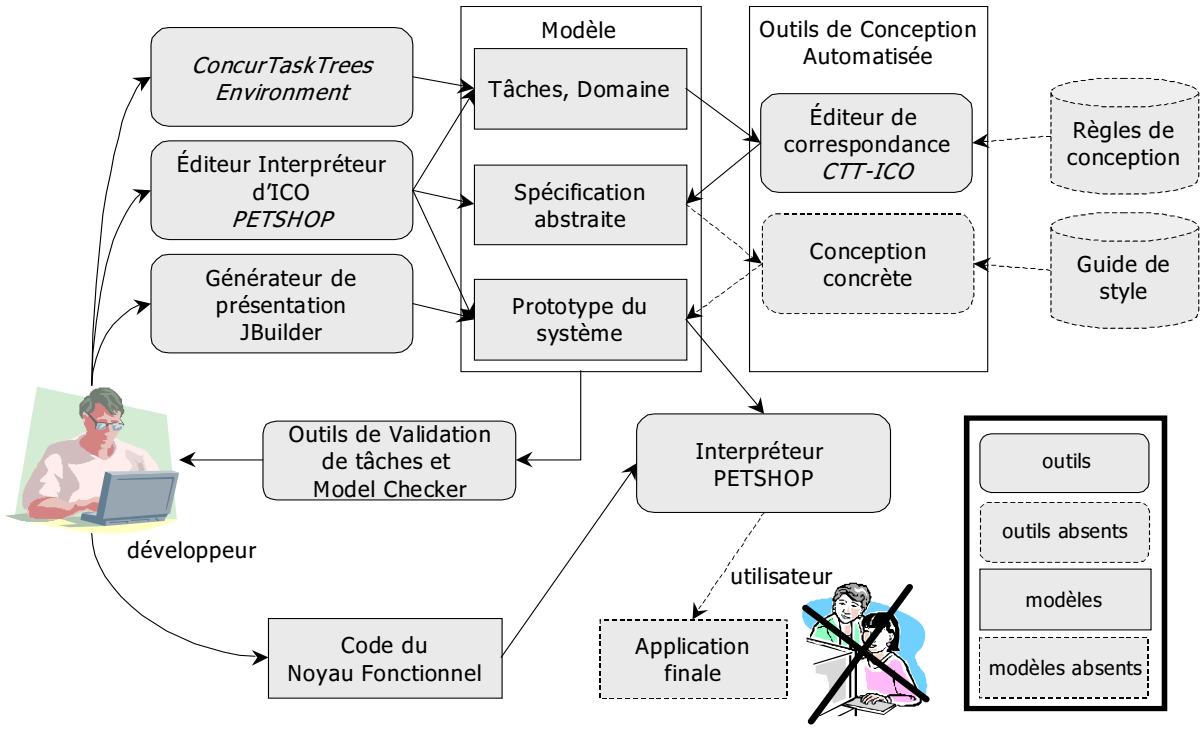

Fig. 1.32 – Architecture générique de l'environnement PETSHOP.

1.4.2.6 Conclusion sur les approches descendantes

Différents points de l'approche descendante n'ont pas été évoqués et méritent d'être signalés. Nous ne nous sommes pas attardés par exemple sur le point de l'architecture « embarquée » mais nous l'avons entrevu au moment des descriptions des différentes approches. La plupart de celles-ci séparent clairement la modélisation du système interactif

en plusieurs modules décrivant les modèles (domaine, tâches, dialogue, spécification abstraite et concrète) et le passage d'un niveau du modèle à un autre définissant les liaisons entre les modules d'une architecture. Nous ne saurions être exhaustifs si nous ne parlions pas des approches qui apportent des contributions complètes dans le sens où elles proposent des méthodes de conception TRIDENT [BHL+95] ou TOOD [Tab01].

Cependant, les approches basées sur modèle se sont essentiellement développées dans le monde de la recherche et n'ont pas encore réellement trouvé leur place dans des outils commerciaux comme Power Designer[12], Oracle[13] et Windev[14]. Par exemple, ce dernier propose des outils d'analyse de la qualité de l'interface, se plaçant par là dans les outils de validation. Cependant, il ne s'intéresse qu'à la couche présentation, avec des propriétés simple comme l'harmonisation des couleurs ou l'alignement des composants graphiques.

1.4.3 Bilan sur les outils de conception

Dans cette section, nous avons étudié les outils de conception qui se basent sur deux grandes approches : l'approche ascendante ou l'approche descendante.

L'approche ascendante consiste à créer l'interface à l'aide d'outils de description de la couche présentation. Le concepteur lie la couche présentation au reste de l'application (noyau fonctionnel et dialogue) de façon textuelle ou par programmation. Cette approche permet de créer la partie graphique de l'application et laisse une très grande liberté au concepteur dans le choix de l'aspect externe de l'application. Cependant, ces outils sont trop proches d'un langage de programmation et aucun raisonnement ne peut donc être fait sur la modélisation. La vérification et la validation sont donc effectuées au moyen de longues phases de tests exhaustifs et/ou de relecture de lignes de code.

L'approche descendante consiste à créer l'application au moyen de la description de modèles définis dans un ou plusieurs langage(s). En général, la modélisation débute par la description du modèle de tâches (besoins de l'utilisateur), du domaine (données de l'application) puis celle du modèle de dialogue et du modèle de la présentation. Le code de l'interface est généré (présentation et dialogue), puis lié au noyau fonctionnel par le concepteur pour fournir une application interactive respectant les contraintes énoncées dans les modèles. L'approche descendante, à l'inverse de l'approche ascendante, autorise des raisonnements sur les modèles.

Cependant, comme nous l'avons remarqué dans notre étude, le raisonnement ne s'ef-

[12]Sybase Inc : http ://www.sybase.com
[13]Oracle Coorporation : http ://www.oracle.com
[14]PC SOFT : http ://www.pcsoft.fr

fectue que sur des éléments partiels de la modélisation. Les travaux dans ce domaine se sont principalement intéressés à l'aspect de la génération et aux guides de style et ne mettent pas en avant la vérification et la validation à tous les niveaux de la modélisation.

Une des raisons de cette lacune est l'absence de sémantique formelle dans le(s) langage(s) qui décrit(vent) les modèles. Peu d'approches se sont intéressées à proposer des modèles à sémantique formelle. Toutefois, il est à noter que le fossé qui existe entre les systèmes basés sur modèles et les approches formelles étudiées dans la section 1.3 tend à se diminuer : pour preuve, l'approche des ICO accompagnée de l'outil PETSHOP ou encore l'approche proposée par les interacteurs du CERT section 1.3.3.1.

1.5 Synthèse : justification du travail de thèse

Le développement de logiciels de qualité nécessite l'utilisation de techniques très rigoureuses. Ces techniques doivent assurer que l'interface homme-machine en développement satisfait les propriétés qui traduisent les exigences définies dans le cahier des charges. Pour développer des **applications interactives sûres**, nous avons montré dans les sections précédentes que deux approches peuvent aujourd'hui être mises en parallèles : les techniques formelles d'une part permettent d'introduire une démarche formelle dans le développement de ces systèmes, et les Systèmes Basés sur Modèles d'autre part qui tirent partis des spécifications de modèles (modèle du domaine, modèle de tâches, etc) pour construire et valider des systèmes interactifs.

L'objet de notre travail de thèse est l'étude du développement d'applications sûres suivant une combinaison de ces deux approches. Nous présentons dans cette section un résumé des sections précédentes et une justification du travail de cette thèse. Dans une première partie, nous discutons les aspects des travaux des approches formelles et des approches MBS dans le but de cerner leurs limites. Tout naturellement, une seconde partie s'intéresse à présenter les grandes lignes de nos travaux en s'appuyant sur ces limites. Nous répondrons notamment aux questions suivantes :

- pourquoi exploiter deux approches de conception ?
- quelles sont les finalités de ces approches ?
- quels sont les besoins (techniques et modélisations) envisagés ?
- quelles sont les limites de nos travaux ?

Enfin, une dernière section présente notre étude de cas (le convertisseur/compteur) avec laquelle nous illustrerons nos approches.

1.5.1 Limites des techniques formelles et des outils de conception pour le développement d'interfaces homme-machine

D'un point de vue général, nous rappelons que le développement d'une interface homme-machine se distingue par deux phases importantes :

1. une *phase de conception* qui permet de produire le code implémentant une IHM de qualité et ses liaisons avec le noyau fonctionnel de l'application. Dans cette phase, des modèles d'architectures, vérification, validation, spécification, raffinement et techniques de programmation sont utilisés par les concepteurs de l'IHM ;

2. une *phase de validation de tâches* qui consiste à la validation des besoins utilisateurs. Cette phase n'est pas maîtrisée par les concepteurs d'IHM puisque la plupart de ces validations sont issues de non spécialistes en informatique comme les psychologues et les ergonomes. Un ensemble de tâches est décrit au moment de la phase d'analyse et devra être supporté par l'IHM finale.

Si, aujourd'hui, le premier point commence à être maîtrisé au moyen des deux approches de conception (techniques formelles et systèmes basés sur modèles), il en n'est pas de même en ce qui concerne la phase de validation de tâches utilisateur. Pourtant, ce point est tout aussi important puisqu'il introduit le comportement de l'utilisateur dans la conception et permet donc d'assurer le critère d'utilisabilité de l'IHM produite.

1.5.1.1 Techniques formelles

Les techniques formelles interviennent à différents niveaux de spécification du système interactif et pour la plupart ne s'intéressent qu'à la couche présentation (les approches à base d'interacteurs par exemple). Au contraire, et nous l'avons présenté, l'approche du LISI permet de couvrir complètement la phase de conception en intégrant des modèles et des notations déjà employés dans le domaine des IHM (modèles d'architecture et boîtes à outils).

De plus, aucune des approches étudiées précédemment n'a permis d'effectuer réellement la validation de tâches. Les interacteurs de York, par exemple, intègrent implicitement la validation de tâches, mais n'ont permis de traiter entièrement cet aspect puisque le développement n'était pas complet (validation de tâches partielle). Les ICO ont permis d'intégrer la validation de tâches mais n'ont pas pu fournir une sémantique formelle homogène pour décrire chacun des modèles (ICO et CTT).

Finalement, la majorité des approches employées à l'exception de l'approche du LISI et des interacteurs de York emploie une technique de vérification sur modèles qui nécessite de parcourir l'ensemble des états de la modélisation au risque d'une explosion combinatoire si l'application modélisée devient importante.

1.5.1.2 Systèmes Basés sur Modèles

Les systèmes basés sur modèles à l'exception de Petshop s'appuient sur des modèles à sémantique pauvre et ne permettent pas un développement incrémental ni l'expression et la vérification de propriétés. Nous avons aussi vu dans la section 1.4.2 que les approches MBS ne fournissaient pas suffisamment d'outils de validation.

Une nouvelle tendance est d'incorporer des approches formelles dans les MBS afin que les modèles soient plus expressifs pour la validation. Petshop intègre une sémantique formelle pour les RdP et des outils de validation. Toutefois, rappelons-le, la prise en compte des besoins utilisateurs dans cet outil ne s'effectue qu'au niveau le plus bas de la modélisation et ne permet pas d'intégrer de modèle de tâches au début de la conception.

Cependant, peu d'environnements MBS (MASTERMIND ou Petshop) couvrent le cycle de développement complet d'une application tout en gardant des liaisons actives entre les modèles, et avec le code final. La phase de génération (soit automatique, semi-automatique ou manuelle) déconnecte généralement les modèles du code généré, et casse la liaison sémantique à l'intérieur du projet. En d'autres termes, sans cet aspect, il devient impossible de travailler directement avec les modèles pendant l'exécution de l'application.

Enfin, un dernier point concerne la facilité d'utilisation. On reproche souvent aux approches MBS une difficulté d'utilisation. Ces systèmes demandent une maîtrise des concepts qu'ils manipulent, ce qui requiert un important apprentissage. Nous avons vu aussi que la frontière des compétences des concepteurs n'est pas clairement explicitée.

1.5.2 Justification du travail de thèse

Notre travail de thèse propose de définir une combinaison des techniques formelles et des systèmes basés sur modèles. Nous pensons que l'utilisation conjointe de ces deux approches dans le cycle de développement permettrait d'améliorer considérablement l'utilisabilité des interfaces homme-machine. En effet :

– d'une part, les techniques formelles pour la conception des IHM permettent l'expression, à un haut niveau d'abstraction, de descriptions, de comportements et de

fonctions des interfaces homme-machine ainsi que la possibilité de les valider. Leurs rôles seraient à même de garantir au plus tôt que les propriétés du système sont respectés ;

– d'autre part, les systèmes basés sur modèles permettent de représenter des descriptions en se basant sur les modèles ou notations déjà employés dans le domaine des IHM.

Dans ce sens, nous nous intéressons plus particulièrement dans notre travail de thèse à la mise en place de deux nouvelles approches qui essayent de répondre aux limites présentées précédemment. Les aspects liés à la méthodologie notamment, qui définissent les liaisons précises de ces nouvelles approches les unes par rapport aux autres seront discutés dans les perspectives de ce travail.

Présentons brièvement les caractéristiques des approches que nous allons développer successivement dans ce mémoire.

1.5.2.1 Approche 1 : Modélisation et validation formelles de descriptions de l'interaction dans les IHM

Nous avons choisi de partir de l'approche fondée sur la modélisation modulaire du LISI que nous avions présentée dans la section 1.3.3.3. Elle offre l'avantage de reposer sur l'utilisation de techniques déjà utilisées dans les IHM (modèle d'architecture et boîte à outils). Cette approche permet déjà aux concepteurs du domaine des IHM de pouvoir utiliser des concepts qu'ils connaissent déjà. Cette approche a montré la possibilité de modéliser des systèmes possédant des interactions complexes (manipulation directe par exemple), puis d'exprimer et vérifier des propriétés du domaine des IHM. La présence de l'outil Atelier B et de son prouveur interactif limite la difficulté d'utilisation liée à la vérification de ces propriétés.

Cependant, la totalité des propriétés décrites en section 1.2.6 et notamment pour la représentation de tâches afin de pouvoir vérifier des propriétés de validité, de complétude et de déroulement de tâches n'ont pas été exprimées. Nous pensons, et c'est l'objet de cette contribution, que des spécifications décrites avec B dédiées aux tâches peuvent être bâties à partir de la spécification de la phase de conception pour valider des modèles de tâches. L'intérêt est de pouvoir utiliser une technique formelle homogène dans toutes les phases de développement d'une IHM et de pouvoir vérifier des propriétés *à priori*, assez tôt dans le processus de développement.

1.5.2.2 Approche 2 : Une approche expérimentale pour la construction d'interfaces utilisateurs sûres (SUIDT)

SUIDT (Safe User Interface Development Tool[15]) est un système basé sur modèles qui permet de construire interactivement une interface homme-machine avec le respect de la sémantique formelle d'un noyau fonctionnel. Il réalise une coopération complète entre le modèle de tâches et les modèles du domaine et de la présentation, tout en respectant les propriétés des modèles. Enfin, il maintient tout au long du cycle de développement des liaisons entre chaque partie du système ainsi que les objets du noyau fonctionnel (code existant), ce que les approches existantes ne permettent pas.

Nous montrons que cette approche est originale parce que toute la construction de l'IHM se base sur un modèle de tâches. Elle permet donc de prendre en compte au plus tôt les besoins de l'utilisateur. L'aspect de la vérification de propriétés est renforcé par des outils de validation qui permettent de tester l'IHM en construction tout en assurant les liaisons entre chaque modèle.

1.5.3 Etude de cas

Nos travaux ont pour domaine d'application les IHM dans les systèmes critiques (avionique par exemple). Cependant, nous avons choisi une étude de cas simple de type WIMP permettant d'illustrer les différents concepts de nos contributions. Elle est composée de deux applications : une application de conversion francs/euros et un compteur, figure 1.33.

1.5.3.1 Convertisseur francs/euros et Compteur

Le convertisseur, repère 1, est un logiciel utilitaire permettant de réaliser des conversions d'une somme en francs en son équivalent en euros et vice versa. L'utilisateur saisit la valeur qu'il désire convertir, choisit le sens de la conversion, déclenche la conversion, puis lit le résultat dans la monnaie souhaitée. Après chaque conversion l'application compteur, repère 2, 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.

[15] Outil de développement d'interface utilisateur sûre

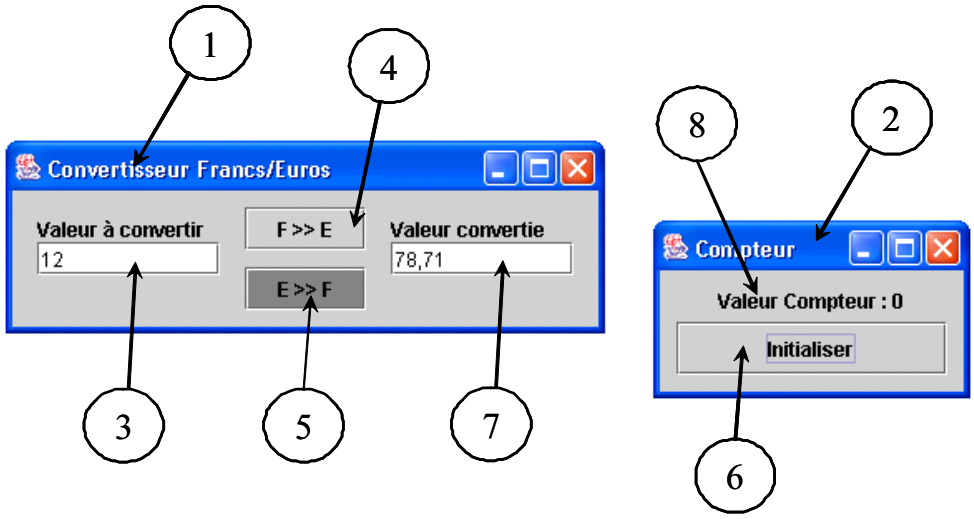

Fig. 1.33 – Présentation des interfaces de l'étude de cas : Convertisseur francs/euros et Compteur.

1.5.3.2 Aspects interactifs de l'étude de cas

L'utilisateur interagit avec les applications convertisseur et compteur via des interfaces graphiques construites au moyen de composants graphiques comme des fenêtres, des boutons, des textes et des zones de saisie.

Les interactions de l'utilisateur se limitent à la saisie de la valeur à convertir, repère 3, à l'appui sur un bouton *E»F* ou *F»E*, repère 4 et 5, et à l'initialisation du compteur par l'appui du bouton *Initialiser*, repère 6.

La couleur des boutons de conversion change en fonction du sens de conversion. Cette caractéristique est illustrée sur l'interface de la figure 1.33 où l'état de conversion est euros vers francs.

Quand trois conversions ont été effectuées, les interactions de l'utilisateur se limitent à l'application compteur. Inversement, tant que ces trois conversions n'ont pas été effectuées, l'utilisateur ne peut interagir que sur le convertisseur. La valeur compteur est incrémentée à chaque nouvelle conversion.

La visualisation des données est obtenue au moyen d'une zone de texte qui retourne la valeur convertie, repère 7, et d'un texte qui affiche la valeur du compteur, repère 8.

Chaque nouvelle interaction effectuée sur un composant provoque une modification de la focalisation. Cette donnée permet de connaître le composant actif d'une interface. Cet

aspect est intéressant puisqu'il nous permet de contraindre l'interaction de la saisie de chiffre. En effet, l'utilisateur ne peut pas saisir un chiffre si le composant correspondant à la zone de saisie n'est pas actif.

Enfin, quand l'utilisateur clique sur la zone de lecture avec le curseur de la souris le convertisseur est initialisé (zone de saisie et zone de lecture sont vides et les boutons n'ont plus de couleur de conversion).

Chapitre 2

Modélisation et validation formelles de descriptions de l'interaction dans les IHM

2.1 Introduction

De nos jours, le fait que les méthodes formelles aident à augmenter la qualité du logiciel a été largement accepté. Elles consistent à élaborer des modèles sur la base de langages formels durant la phase de spécification. La modélisation formelle reste nécessaire afin d'exprimer, de vérifier et de prouver *a priori* des propriétés sur les systèmes. Nous avons vu dans le chapitre précédent que ces méthodes formelles sont exploitées dans le domaine de l'interaction homme-machine. Cependant, ce domaine se distingue par rapport aux autres par la prise en compte d'un aspect particulier dans le développement : le comportement de l'utilisateur et les tâches qu'il souhaite effectuer à l'aide de l'interface. Ce problème est important et particulier au domaine des interactions homme-machine.

Nous avons remarqué dans le chapitre précédent que les travaux concernant l'utilisation de techniques formelles dans les IHM se limitaient principalement à la phase de conception et ne s'intéressaient pas à la phase de validation de tâches.

Dans ce chapitre, nous proposons une démarche fondée sur l'utilisation conjointe d'une technique formelle pour la mise en place de la phase de conception et de la phase de validation de tâches. Basée sur les travaux de l'approche développée au LISI [AAGJ98a, AAGJ98b, AA00a] et de la méthode B en ce qui concerne la phase de conception, notre contribution porte principalement sur la phase de validation de tâches. Elle traite le sujet

important de la validation de tâches utilisateur dans le contexte des développements formels. Nous pensons qu'il est possible de valider des tâches utilisateurs pour la phase de spécification et de conception évitant ainsi des remises en cause des développements. La validation au plus tôt a des répercutions positives sur le coût de développement mais aussi sur l'amélioration de cette validation.

En outre, nous montrons qu'il est possible d'intégrer dans une même technique formelle (la méthode B) à la fois les étapes de conception et de validation de tâches utilisateur. Par conséquent, des propriétés relatives à l'utilisateur pourront être exprimées. Notons aussi que tous les développements décrits dans ce chapitre sont supportés par l'outil Atelier B. De plus, nous ne suggérons pas de nouvelles techniques ni de nouvelles notations. Notre préoccupation a été de fournir des outils et des techniques formelles afin d'aider les concepteurs à appliquer des notations et des modèles déjà définis et utilisés par des spécialistes du domaine de l'interaction homme-machine. Enfin, nous avons volontairement omis de présenter les différentes étapes de raffinement qui conduisent à une représentation dans un langage de programmation dans la mesure où cette étape n'intervient pas directement dans notre contribution. Des informations complémentaires à ce sujet peuvent être trouvées dans [AA00b].

Notre démarche repose sur deux approches de validation de tâches développées en fonction de la version de la méthode B. Dans un premier temps, nous nous sommes basés sur la version de la méthode B exploitée par l'approche du LISI. La validation de tâches est comparable à un diagramme de séquence UML dans le sens où des traces d'opérations sont explicitement construites et validées. Dans un second temps, nous exploitons la sémantique de l'extension de la méthode B appelée B événementiel pour décrire formellement la sémantique de la notation ConcurTaskTrees [Pat01] et valider des tâches exprimées en CTT.

Ce chapitre se décompose en cinq sections. La première rappelle les notions d'automates et de logique de Dijkstra que nous employons dans la suite du chapitre. La deuxième section décrit la méthode B en présentant sa version initiale appelée par la suite B « classique » et son extension appelée B événementiel. Dans la troisième section nous abordons notre première approche, appelée **approche à base de modules**, concernant l'utilisation du B classique pour la phase de conception et de validation de tâches. Puis, dans une quatrième section, nous présentons notre seconde approche, appelée **approche à base d'événements**, qui utilise le B événementiel. Nous discutons évidemment des avantages que cette deuxième approche apporte par rapport à la première. Finalement, nous concluons en comparant nos travaux aux approches présentées dans le chapitre précédent.

2.2 Rappels

Pour les besoins de la modélisation du comportement de systèmes interactifs et de la compréhension de la logique de preuve employée dans la technique formelle B, nous effectuons dans cette section un rapide rappel des notions :

1. d'automates ou systèmes de transitions ;
2. de logique de Dijkstra.

2.2.1 Systèmes de transitions

La représentation formelle de systèmes de processus est nécessaire pour raisonner sur ces systèmes. Les systèmes de transitions (les automates) constituent un moyen puissant permettant de représenter des systèmes de processus. Ce que nous désignons par le terme d'automate emprunte des caractéristiques issues aussi bien de la notion d'automate fini en théorie des langages, que des notions de structure de Kripke [Kri59, Kri63] ou de systèmes de transitions dans d'autres domaines.

Dans le chapitre précédent, nous avions vu que les automates constituaient la base des modèles développés pour décrire le dialogue des systèmes interactifs dans le domaine de l'IHM. Cette section décrit les concepts de base des systèmes de transitions que nous employons dans la suite de ce chapitre.

2.2.1.1 Définition générale

On suppose donné un ensemble $P = \{P_1, \ldots\}$ de propositions (propriétés) élémentaires. Un système de transitions étiquetées (noté STE dans la suite) est un quadruplet $A = (Q, E, T, q_0, l)$ où :

- Q est un ensemble fini d'états ;
- E est l'ensemble fini des étiquettes associées aux transitions ;
- T avec $T \subseteq Q \times E \times Q$ est l'ensemble des transitions. Une transition définit une relation entre deux états de Q et est étiquetée par un élément de E ;
- q_0 est l'état initial du STE ;

– l est l'application qui associe à tout état de Q un ensemble fini inclus dans P de propriétés élémentaires vérifiées dans cet état.

Une transition t est donc un triplet $t = (p, e, q)$ que l'on note $p \longrightarrow^{e} q$.

Un automate est *déterministe* si pour tout état q et pour toute étiquette e il existe au plus une transition ayant pour origine p et étiquette e.

Pour la spécification de systèmes de processus, l'ensemble des étiquettes E associé aux transitions de T comportera les événements ou actions qui peuvent être activées lors du passage d'un état à un autre.

L'application l permet d'obtenir les différentes propriétés vérifiées dans un état donné. Elle permet entre autre de décrire les variables d'états qui caractérisent un automate et de les observer lors d'un changement d'état.

2.2.1.2 Définitions complémentaires

Variables d'états. Les variables d'états caractérisent une propriété de l'état du système représenté. Une variable d'état prend des valeurs dans un ensemble fini ou infini. Les techniques de vérification et de preuve dépendent de la nature de l'ensemble des valeurs des variables d'états (fini ou infini).

Les liens entre automates et variables d'états peuvent être de deux types :

– **affectations** : une transition peut modifier la valeur d'une (ou de plusieurs) variables(s) ;
– **gardes** : une transition peut être gardée par une condition sur les variables d'états. Cela signifie que le franchissement de la transition n'est possible que si la condition sur les variables est vérifiée.

Représentation graphique. Il est possible de représenter un automate en utilisant une représentation graphique. Les états sont représentés par des ronds. L'état initial est distingué par une flèche arrivant sur cet état et sans origine. L'état terminal, s'il en existe est représenté par deux cercles concentriques. Les transitions sont des arcs orientés dans le sens état de départ vers état d'arrivée. L'arc désignant la transition est annoté par une étiquette. Les gardes et les actions sont indiquées sur les transitions suivant une convention graphique bien établie : les actions sont précédées d'une flèche qui indique que les gardes doivent être respectées pour que les actions soient exécutées.

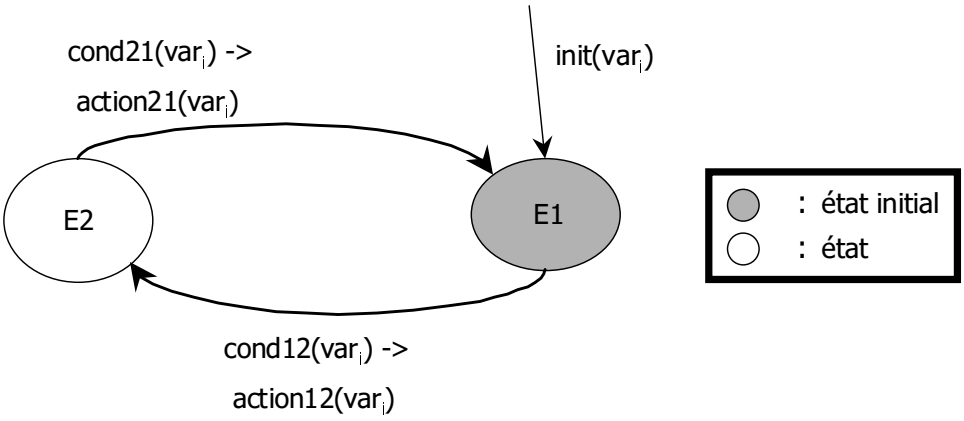

Fig. 2.1 – Exemple d'un automate à deux états.

Nous donnons sur la figure 2.1 une représentation graphique d'un automate à deux états. Cet automate est représenté par un état initial $E1$ et un deuxième état appelé $E2$. $cond21(var_i)$ et $cond12(var_i)$ désignent respectivement les gardes des transitions des états $E1$ à $E2$ et $E2$ à $E1$. De même, les actions $action21(var_i)$ et $action12(var_i)$ modifient la valeur de la variable var_i si les gardes associées à ces actions sont vraies (rôle de la flèche $\rightarrow$).

Notion de trace. Une *trace* ou encore appelée *chemin* dans un automate A est une suite σ, finie ou infinie, de transitions (q_i, e_i, q'_i) de A qui s'enchaînent, c'est-à-dire $q'_i = q_{i+1}$ pour tout i. On note souvent une telle suite sous la forme $q_1 \longrightarrow^{e1} q_2 \longrightarrow^{e2} q_3 \longrightarrow^{e3} q_3 \ldots$.

La longueur du chemin σ noté $|\sigma|$ est le nombre de transitions qu'il contient, éventuellement infinie. Le i-ème état de σ, noté $\sigma(i)$, est l'état q_i atteint après i transitions. Il n'est défini que si $i < |\sigma|$.

Une *exécution partielle* est un chemin partant de l'état initial. Une *exécution complète* est une exécution (partielle) maximale, c'est-à-dire une exécution qui ne peut être prolongée.

2.2.1.3 Produit cartésien

Lors de la description d'un système de processus, on procède souvent par la description des différents processus individuellement. Chaque processus peut être décrit par un automate. Le système complexe est lui décrit par l'ensemble des processus et donc par l'ensemble d'automates.

Le produit cartésien (produit libre) et produit synchronisé d'automates permettent de décrire des systèmes de processus par assemblage (opération de produit) d'automates de base. Ces opérations de produit permettent d'obtenir un automate qui décrit globalement le système de processus. Généralement, l'automate obtenu possède un très grand nombre d'états, si bien qu'il est difficile voir impossible de le construire.

Produit cartésien ou produit libre. Considérons une famille de n automates $A_i = (Q_i, E_i, T_i, q_{0,i}, l_i), i = 1 \ldots n$.

Soit '-' une nouvelle étiquette permettant d'exprimer l'action fictive. Cette action fictive servirait à décrire qu'un des sous-automates n'effectue aucune transition dans l'automate global.

Le produit cartésien $A_1 \times A_2 \times \ldots \times A_n$ de ces automates, est l'automate $A = (Q, E, T, q_0, l)$ tel que :

- $Q = Q_1 \times Q_2 \times \ldots \times Q_n$
- $E = \prod_{i=1}^{n}(E_i \cup \{-\})$
- $T = \{((q_1, \ldots, q_n)(e_1, \ldots, e_n)(q'_1, \ldots, q'_n))|\forall i \in 1..n, (e_i =' -' \text{ et } q_i = q'_i)$ ou bien $(e_i \neq' -' \text{ et } (q_i, e_i, q'_i) \in T_i)\}$
- $q_0 = (q_{0,1}, \ldots, q_{0,n})$
- $l = \bigcup_{i=1}^{n}(l_i(q_i))$

Dans un produit cartésien, chaque automate A_i peut, lors d'une transition, soit effectuer une transition locale, soit ne rien faire (action fictive). Il n'y a aucune obligation de synchronisation entre les différents automates. De plus, le produit cartésien permet des transitions où tous les automates ne font rien.

Produit synchronisé. Pour synchroniser les différents automates d'un produit cartésien, il est nécessaire de restreindre les transitions possibles dans l'automate résultant du produit cartésien. Seules les transitions correspondant à des transitions de synchronisations acceptées seront conservées.

Considérons un ensemble de synchronisations *Sync*, ou encore appelé vecteur de synchronisation) défini par :

$$Sync \subseteq \prod_{i=1}^{n}(E_i \cup \{-\})$$

Le produit synchronisé $(A_1 \parallel A_2 \parallel \ldots \parallel A_n)_{Sync}$ des automates A_i, est l'automate $A = (Q, E, T, q_0, l)$ tel que :

- $Q = Q_1 \times Q_2 \times \ldots \times Q_n$
- $E = \prod_{i=1}^{n}(E_i \cup \{-\})$
- $T = \{((q_1, \ldots, q_n)(e_1, \ldots, e_n)(q'_1, \ldots, q'_n))|(e_1, \ldots, e_n) \in Sync$ et $\forall i \in 1..n, (e_i =' -'$ et $q_i = q'_i)$ ou bien $(e_i \neq' -'$ et $(q_i, e_i, q'_i) \in T_i)\}$
- $q_0 = (q_{0,1}, \ldots, q_{0,n})$
- $l = \bigcup_{i=1}^{n}(l_i(q_i))$

Dans la contribution que nous apportons dans ce chapitre, nous allons construire le produit synchronisé de manière ascendante et descendante.

2.2.2 Logique de Dijkstra

La logique de Dijkstra [Dij76] a été développée après les travaux de Hoare [Hoa69]. Dans cette approche, Dijkstra propose un calcul sur les assertions qui est ascendant. Il propose en partant d'une postcondition, de trouver la plus faible précondition qui vérifie le triplet de Hoare.

Ce rappel débute par une définition générale de cette logique puis nous présentons des exemples de transformations de prédicats relatives à des opérations de base d'un langage de programmation. Nous verrons pendant la présentation de la méthode B que ces opérations de base seront utilisées pour décrire les obligations de preuves.

2.2.2.1 Définition générale

Pour chaque élément de programme ou de spécification S, nous définissons, à partir d'une postcondition Q, la plus faible précondition P telle que $\{P\}S\{Q\}$.

Cette plus faible précondition est notée $\{[S]Q\}S\{Q\}$.

De plus si $P \Rightarrow S[Q]$ alors on a $\{P\}S\{Q\}$

Où $\{P\}S\{Q\}$ désigne le triplet de Hoare.

2.2.2.2 Exemples de transformations de prédicats

Toutes les constructions de base du langage de programmation sont représentées à l'aide de la logique de Dijkstra. Des règles associées à chacune des constructions d'un type indiquent les transformations de prédicats. Les expressions du langage sont représentées en gras. Toute construction de langage S sera définie par $[S]Q$.

Interprétation de skip. L'instruction *skip* est l'instruction qui « ne fait rien ». Elle joue le rôle d'instruction neutre.

$$[\mathbf{skip}]Q \equiv Q$$

Elle est notamment utilisée pour décrire que certaines branches d'une construction de type *conditionnelle* par exemple ne modifient pas les variables.

Interprétation de l'affectation. La règle associée à l'affectation $x := E$ consiste à vérifier la précondition en substituant l'occurrence de la variable affectée x par l'expression E.

$$[\mathbf{x} := \mathbf{E}]Q \equiv [x := E]Q$$

Si $x := x + 1$ est l'affectation et Q le prédicat $x \neq 2$, alors la plus faible précondition pour assurer $x \neq 2$ après $x + 1$ est $x \neq 1$ et doit être vérifiée.

Interprétation de la séquence. La séquence est l'opération de base qui permet de composer des programmes.

$$[\mathbf{S}\ ;\mathbf{T}]Q \equiv [\mathbf{S}][\mathbf{T}]Q$$

Ce qui signifie que le prédicat obtenu par application de l'opération séquence $S;T$ sur le prédicat Q est le prédicat obtenu par application de S sur le résultat de l'application de T sur Q.

2.3 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 [Dij76] 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 quoi*.

Le modèle B est ensuite raffiné, c'est-à-dire spécialisé, jusqu'à obtenir une implantation complète du système logiciel : *le comment*. 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. Bien entendu, plusieurs raffinements peuvent satisfaire une spécification.

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 est utilisable industriellement car il existe des outils de preuves automatiques commercialisés. Les outils les plus répandus sont Atelier-B[1] et B-Tool[2].

Toutefois, la méthode B 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 qui est indispensable dans les 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.

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

Cette section étudie la méthode B. Nous nous intéressons dans une première partie à la présenter sous sa forme « classique » puis sous la forme B « événementiel ». Nous focalisons notre présentation sur les aspects liés à la technique de preuve tout en donnant suffisamment d'éléments de syntaxe pour que les lecteurs prennent déjà connaissance du vocabulaire quand des exemples lui seront présentés. Enfin, nous terminons en discutant sur les points avantageux de la méthode B face à nos besoins en termes de modélisation de systèmes interactifs.

2.3.1 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 donnée. 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 sur le tableau 2.1. Chaque mot clé en gras est appelé une clause. Il est à noter que nous n'avons représenté que les clauses principales, mais pour ne pas alourdir cette présentation nous ne les présentons pas explicitement.

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.

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. 2.1 – Machine abstraite B générique.

2.3.1.1 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 [Dij76] que nous avons présenté précédemment.

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 2.2 une machine abstraite générique simplifiée en B

classique (sans constantes ni propriétés logiques sur ces constantes) correspondant à celles que nous employons dans une partie de notre contribution. Il est à noter que la forme de l'opération dépend de la présence ou pas de paramètres de sortie et 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. 2.2 – Machine abstraite B avec une opération.

Les obligations de preuve de cette machine à vérifier sont décrites sur le tableau 2.3.

	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. 2.3 – Obligations de preuve d'une machine abstraite B.

La première INV_1 concerne l'initialisation (une opération particulière) qui établie 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 l'opération O qui préserve l'invariant c'est-à-dire que, sous couvert de l'invariant, l'opération 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 2.4, 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. 2.4 – Substitutions utilisées pour définir la forme d'une opération.

Nous reviendrons sur les substitutions *bloc*, *sélection* et *choix non borné* dans la section 2.3.2. 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 2.5, les différentes formes de substitutions que nous employons pour $T(x)$. Il est à noter que nous les avons en partie présentées dans la section 2.2.2. Nous en donnons pour chacune la transformation de prédicat suivant la logique de Dijkstra.

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

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 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 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.

2.3.1.2 Structuration d'une modélisation B

La méthode B propose des techniques de structuration qui permettent le découpage d'un projet B. En effet, il n'est pas envisageable de spécifier complètement un système au moyen d'une unique et seule machine abstraite car le nombre d'obligations de preuve serait trop important. Le passage par des raffinements successifs, que nous étudions plus en détail dans la partie liée au B événementiel, est déjà un moyen mais dés que la complexité des raffinements devient trop élevée, la décomposition en plusieurs parties plus simples est possible.

L'avantage des techniques de composition réalisées au moyen de clauses spécifiques (**EXTENDS**, **INCLUDES**, **IMPORTS**, etc) à B permet de regrouper dans une machine abstraite ou un raffinement, les constituants (ensemble, constantes et variables) de machines ainsi que leurs propriétés (clause **PROPERTIES** et **INVARIANT**), afin de créer un composant enrichi à l'aide d'autres machines abstraites. Ce lien permet de construire de manière modulaire des machines abstraites ou des raffinements.

Par exemple, l'Atelier B offre des bibliothèques qui gèrent des tableaux et des opérations qui effectuent des sorties sur la console du terminal.

Cette technique est à rapprocher à la conception dite ascendante et permet une conception du système par composition modulaire. Cependant à chaque composition, il faut de nouveau re-prouver certaines des obligations de preuve des machines composées ce qui peut compliquer la preuve des obligations de preuve.

L'approche du LISI s'appuie sur cette forme de conception par composition ascendante pour modéliser les systèmes interactifs. Nous l'utilisons dans la première partie de notre contribution.

2.3.2 B événementiel

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, l'automate devient temporisé et 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

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. 2.6 – Modèle B événementiel générique.

de la méthode B est rendue plus simple. Il n'y a plus de notion de 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 2.6 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.

2.3.2.1 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, défini 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 :

$$\begin{array}{|l} nomEvt\ = \\ \textbf{BEGIN} \\ \ S(x) \\ \textbf{END} \end{array}$$

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.

$$\begin{array}{|l} nomEvt\ = \\ \textbf{SELECT} \\ \ G(x) \\ \textbf{THEN} \\ \ S(x) \\ \textbf{END} \end{array}$$

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

2.3.2.2 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 2.7 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. 2.7 – Obligations de preuve d'un modèle B événementiel.

2.3.2.3 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 2.8 la forme d'un modèle raffiné.

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 2.9 le modèle concret R qui raffine le modèle abstrait M et nous donnons maintenant les obligations de preuve associées.

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. 2.8 – Refinement B événementiel générique.

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.

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 nous engendrons des obligations de preuve qui permettent de démontrer l'établissement par conservation d'un événement lors de son raffinement. 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 $T(y)$.

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

Tab. 2.9 – Raffinement B événementiel.

Cette obligation de preuve est la suivante :

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

Il est à remarquer 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 qu'ils 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 2.10 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. 2.10 – Obligation de preuve d'un raffinement en B événementiel.

2.3.2.4 Du B classique au B événementiel

Il n'existe pas encore d'outil qui implante le B événementiel. Les modèles B événementiel sont représentés en langage B classique afin d'utiliser l'outil Atelier B pour générer automatiquement les obligations de preuve et pour décharger ces preuves grâce au prouveur automatique et interactif.

Lors de la traduction, il faut ajouter dans la clause **ASSERTIONS**, la propriété liée à l'obligation de preuve (REF_5) qui permet d'assurer que le système est réactif (propriété de non blocage). Cette propriété est exprimée au moyen de la disjonction des gardes et montre qu'il y a toujours un événement qui a sa garde de vérifiée. Il faut également décrire la décroissante des variants par des événements [Can03].

2.3.3 Bilan sur la méthode B

Notre contribution consiste à définir une approche de validation de tâches lorsque les développements sont réalisés avec B. Nous avons apporté deux contributions.

La première est fondée sur l'approche du LISI. Elle se base complètement sur le mode de conception dit ascendant où le B classique est utilisé dans toute la phase de conception. Cette première contribution appelée **approche à base de modules** vise à exploiter les résultats de l'approche du LISI pour la phase de conception afin d'apporter une solution à la validation de tâches utilisateur. Nous utilisons uniquement la version B classique pour construire et valider des tâches utilisateurs. L'approche de validation de tâches que nous appelons **approche explicite** s'appuie sur une démarche comparable aux diagrammes de séquence d'UML dans le sens où des traces d'opérations sont construites et validées.

Une 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 du raffinement.

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 du LISI. Elle l'enrichit pour permettre de décrire le comportement dynamique du contrôleur de dialogue. Ici, seule le contrôleur de dialogue est modélisé en B événementiel. Il fait appel aux opérations des autres machines exprimées en B classique. Nous employons pour cela, les systèmes de transition, le produit synchronisé, la sémantique du B événementiel et la technique de raffinement. Les travaux sur la validation de tâches, appelés **approche implicite** exploitent une notation de tâches, en l'occurrence ConcurTaskTrees [Pat01], issue du domaine des IHM.

Toutes les modélisations B ont été réalisées avec l'Atelier B, support des phases de développement depuis la spécification jusqu'au raffinement. 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 démontrant les obligations de preuve grâce à un prouveur automatique et interactif.

2.4 Approche à base de modules

Dans une première partie, nous proposons d'étudier l'approche développée au sein du LISI au travers de son étude de cas : Post-It Notes®. Puis, nous l'appliquons dans une deuxième partie à notre étude de cas du convertisseur francs/euros et compteur en insistant sur la modélisation modulaire et sur les propriétés qui ont été exprimées et vérifiées. Une troisième partie s'intéresse à la description de la validation de tâches utilisateur par l'approche explicite tout en l'appliquant à notre étude de cas. Enfin, une dernière section établit un bilan de l'approche à base de modules et montre ses limites.

2.4.1 Approche du LISI

L'approche développée au sein du LISI [AAGJ98a, AAGJ98b] s'est intéressée à exploiter la méthode B, et plus particulièrement dans sa version classique, pour modéliser la phase de conception des systèmes interactifs de type WIMP.

A notre connaissance, l'application de la méthode B n'avait pas été réalisée dans le domaine de l'interaction homme-machine. Les auteurs ont montré dans cette première expérience que l'utilisation de cette méthode pouvait s'appliquer à la modélisation de

systèmes interactifs tout en réutilisant des techniques et notations déjà utilisées dans les IHM.

Une étude de cas concrète a permis de mettre en œuvre cette approche : Post-It Notes®.

Il s'agit d'une application à manipulation directe (c'est-à-dire que les objets graphiques sont manipulés par l'utilisateur directement via une souris par exemple) qui se compose d'un bloc dont l'utilisateur peut détacher des notes et sur lesquelles il peut écrire de messages et les envoyer à un autre utilisateur. Chaque note est elle-même représentée par une fenêtre que l'on peut agrandir, déplacer, iconifier et détruire par manipulation directe. Une vue de cette application est donnée sur la figure 2.2.

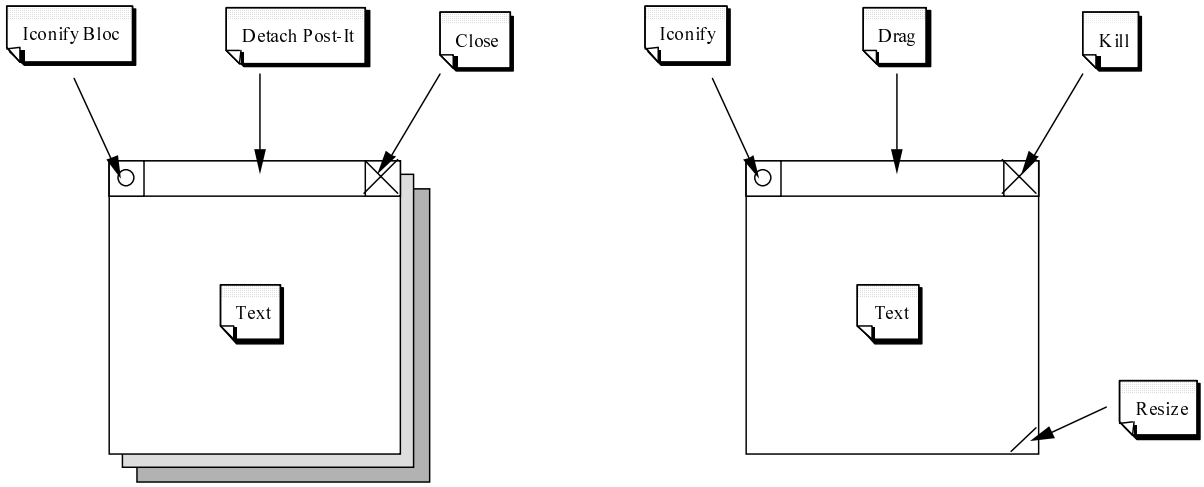

Fig. 2.2 – Interface de l'application Post-It Notes® modélisée au moyen de la méthode B.

Nous ne présentons pas ici l'intégralité du développement mais nous donnons quelques exemples des points importants de l'approche : modularité, rétro-conception, contrôleur de dialogue et expression des propriétés. L'intégralité du développement est disponible dans [AAGJ98b]. Enfin, il est à noter que cette approche ne définit pas de méthode : toute la modélisation se fait de façon empirique.

2.4.1.1 B et l'architecture logicielle

Le concept simplifié d'encapsulation a été utilisé pour construire les machines abstraites B. La décomposition modulaire de l'application suit le schéma général de l'architecture ARCH. Différentes machines abstraites ont été définies pour correspondre au **domaine**, **adaptateur du domaine**, **contrôleur de dialogue**, **présentation** et **boîte**

à outils. Sur la figure 2.3, les formes ovales représentent les machines abstraites, alors que les flèches désignent les relations de visibilité entre machines. La structuration des machines abstraites a été obtenue par la clause **EXTENDS**. Elle permet à une machine d'intégrer les données (variables, opérations et propriétés) de la machine incluse. Ainsi, la liaison entre les groupes de machines qui définissent des modules est réalisée au travers de cette clause.

L'approche du LISI est une approche ascendante qui modélise le système complet par composition de machines abstraites. Nous ne le montrons pas, mais la particularité de l'approche du LISI est de pouvoir atteindre directement le code en raffinement chaque machine abstraite jusqu'à la phase d'implémentation. Des informations complémentaires peuvent être trouvées dans [AA00a].

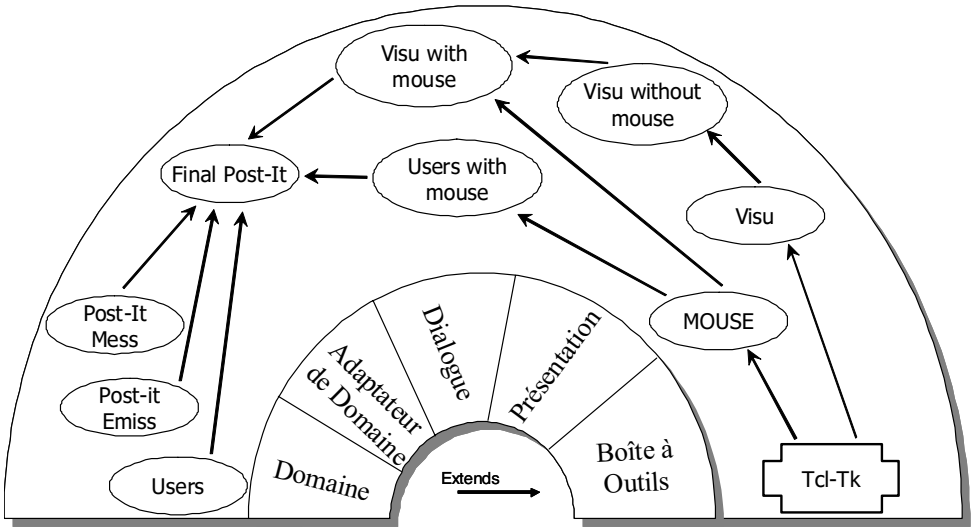

Fig. 2.3 – Relations entre machines abstraites B de la modélisation du Post-It Notes®.

La machine $Final\ Post-It$ de plus haut niveau décrit le module d'adaptateur de domaine et joue le rôle de contrôleur de dialogue. Les machines du module boîte à outils, utilisées par le module présentation, sont obtenues par rétro-conception de la boite à outils Tcl-Tk.

2.4.1.2 Rétro-conception

Nous avons vu dans la section 1.4.1.1 qu'il existait de nombreuses boîtes à outils qui facilitaient la conception de l'interface. Le développement de l'IHM consiste alors à réutiliser des objets d'une boîte à outils et à les appeler. Dans le but de spécification de l'interface par la méthode B, il convient de modéliser ces objets. La modélisation

est obtenue par rétro-conception où chaque machine décrit le comportement d'un de ces objets. Ainsi, le module boîte à outils de l'architecture ARCH contient-il un ensemble de machines abstraites liées par la clause EXTENDS.

Nous proposons l'exemple de la spécification de la machine gérant la souris qui malgré sa simplicité illustre convenablement les propriétés qui peuvent être gérées par B.

L'ensemble ($MOUSE\_STATE$) décrit les différents états de la souris (haut, bas et cliqué). Il est à noter qu'il s'agit d'une souris à un bouton. Les constantes définissent la zone de couverture du curseur de la souris ($max\_mouse\_pos\_width, max\_mouse\_pos\_high$) et les positions par défaut du curseur ($x\_mouse\_pos\_default, y\_mouse\_pos\_default$).

MACHINE $POSTIT\_MOUSE$
SETS
$MOUSE\_STATE = \{up,\ down,\ clicked\}$
CONSTANTS
$x\_mouse\_pos\_default,\ y\_mouse\_pos\_default,$
$max\_mouse\_pos\_width,\ max\_mouse\_pos\_high$
PROPERTIES
$x\_mouse\_pos\_default = 20 \wedge$
$y\_mouse\_pos\_default = 20 \wedge$
$max\_mouse\_pos\_width = 1024 \wedge$
$max\_mouse\_pos\_high = 768$

Les variables ($x\_post\_it\_mouse\_pos$, $y\_post\_it\_mouse\_pos$) définissent la position du curseur. L'état de la souris est stocké dans ($post\_it\_mouse\_state$). L'invariant sert à typer les variables et à définir une propriété sur le curseur de la souris. Cette propriété exprime que le curseur ne doit jamais dépasser les limites de l'écran[3]. Quelle que soit l'opération appelée, cette propriété doit être toujours vraie.

VARIABLES
$x\_post\_it\_mouse\_pos,$
$y\_post\_it\_mouse\_pos,$
$post\_it\_mouse\_state,$
INVARIANT
$x\_post\_it\_mouse\_pos \in NAT \wedge$
$y\_post\_it\_mouse\_pos \in NAT \wedge$
$post\_it\_mouse\_state \in MOUSE\_STATE \wedge$
$x\_post\_it\_mouse\_pos \in 1..max\_mouse\_pos\_width \wedge \ldots$

[3]Dans la suite du mémoire, nous utiliserons .. pour représenter un intervalle et ... pour désigner une suspension de code.

Dans cette spécification où sont modélisées des données (incomplète, il manque la clause **INITIALISATION**), plusieurs opérations sont définies. Une opération qui se charge de modifier la position de la souris ($move\_mouse$), deux opérations qui modifient les états ($mouse\_down$, $mouse\_clicked$) puis une opération qui correspond au déplacement de la souris, bouton enfoncé ($Drag$).

Nous n'illustrons ici que cette dernière. La précondition de cette opération vérifie que les nouvelles coordonnées de la souris ne violent pas l'invariant, et que le bouton est bien toujours enfoncé pendant le déplacement. Si toutes ces conditions sont bien remplies, l'état de la souris est modifiable.

$move\_mouse\_with\_drag(px, py)\ =$
PRE
$px \in NAT\ \wedge\ px \in 1..max\_mouse\_pos\_width\ \wedge$
$py \in NAT\ \wedge\ py \in 1..max\_mouse\_pos\_high\ \wedge$
$post\_it\_mouse\_state = down$
THEN
$x\_post\_it\_mouse\_pos := px\ \ \|$
$y\_post\_it\_mouse\_pos := py$
END
$move\_mouse(px,\ py) = \ldots\ ;$
$mouse\_down() = \ldots\ ;$
$mouse\_clicked() = \ldots$
END

2.4.1.3 Liaison formelle des modules de l'ARCH

L'utilisation de la clause EXTENDS permet de définir les dépendances entre les machines abstraites et par conséquent les liaisons entre les modules de l'architecture ARCH. Dans un premier temps nous présentons quelques éléments de la machine $POSTIT\_VISU\_WITHOUT\_INT$ qui gère la visualisation des notes indépendamment de tout aspect d'interaction.

MACHINE $POSTIT\_VISU\_WITHOUT\_INT$
EXTENDS $POSTIT\_VISUALIZATION$
...
PROPERTIES
$max\_post\_it\_width = 300 \wedge$
$max\_post\_it\_high = 250$
VARIABLES
...
INVARIANT
$\forall\, xx.(xx \in the\_post\_it\_visu \Rightarrow$
$(x\_post\_it\_position \in 1..max\_post\_it\_width \wedge$
$y\_post\_it\_position \in 1..max\_post\_it\_high \wedge)$
...

Dans cette machine, nous définissons, les dimensions maximales d'une note, puis l'invariant qui exprime le fait que toutes les notes contenues dans l'ensemble *the_post_it_visu* doivent avoir leur coin supérieur gauche dans l'écran. Il s'agit d'une propriété qui doit être toujours vraie (invariant).

MACHINE $POSTIT\_VISU\_WITH\_INT\_MOUSE$
EXTENDS $POSTIT\_MOUSE,\ POSTIT\_VISU\_SANS\_INT$
...
OPERATIONS
$move\_window\_with\_mouse(p\_int,\ px,\ py)\ =$
PRE
$px \in NAT\ \wedge\ px \in 1..max\_post\_it\_width \wedge$
$py \in NAT\ \wedge\ py \in 1..max\_post\_it\_high \wedge$
...
$x\_post\_it\_mouse\_pos \in$
$x\_post\_it\_position(p\_int) + 5\ ..$
$x\_post\_it\_position(p\_int) +$
$x\_post\_it\_window(p\int) - 5 \wedge$
...
THEN
$move\_window\_pos(p\_int,\ px,\ py) \parallel$
$move\_mouse\_with\_drag(\ px,\ py)$
END ;

Ici l'opération *move_window_with_mouse* n'est exécutable que si le curseur de la souris se trouve dans une zone rectangulaire définie par la taille de la note (la valeur entière

5 représente une tolérance due à la bordure de la fenêtre). Cette composition permet de réutiliser des opérations définies dans d'autres machines, puis de conserver et respecter les propriétés de ces machines.

2.4.1.4 Contrôleur de dialogue

La machine *Final_Post_It* regroupe toutes les machines afin de permettre la gestion globale du projet de modélisation. Elle joue le rôle du programme principal ou encore de contrôleur de dialogue en établissant un lien entre les machines du noyau fonctionnel et celles de l'interfaces. Plus précisément cette machine est composée d'un ensemble d'opérations qui effectuent dans certains cas (parce qu'ici l'adaptateur du domaine est simple) des appels aux opérations du module présentation et adaptateur du domaine.

Prenons l'exemple de l'opération *Détruire un Post It*, décrite ci-dessous, qui permet de détruire une note. Elle appelle à la fois une opération de l'adaptateur du domaine *destroy_message* pour supprimer les messages de la note puis l'opération *eliminate_post_it_with_mouse* qui supprime la note. Sa précondition est obtenue par la conjonction des préconditions des deux opérations.

MACHINE *Final_Post_It*
EXTENDS *POSTIT_EMISS*, *POSTIT_VISU_WITH_INT_MOUSE*, ...
...
OPERATIONS
destroy_post_it(pp) =
PRE
...
THEN
destroy_message(pp) ||
eliminate_post_it_with_mouse(pp)
END ;

L'utilisation de B classique pour représenter le contrôleur de dialogue ne nous permet pas de représenter l'interactivité et la réactivité. En effet, dans cette approche nous avons supposé qu'une boucle (sur le temps) englobe ces différentes opérations qui sont déclenchées par l'arrivée d'appels (événements). Ces derniers n'ont pu être modélisés par la suite qu'avec l'apparition du B événementiel.

2.4.1.5 Vérification des propriétés

La méthode B permet de prouver au moyen des invariants que des propriétés d'une IHM sont valides. Ainsi, dans l'exemple du Post-It Notes®, des propriétés ont été exprimées et vérifiées à l'aide de l'Atelier B :

- propriétés de sûreté : assurées par la préservation des invariants de la boîte à outils. Ni la souris, ni les notes ne peuvent se trouver à un moment donné hors de l'écran ;
- propriétés de visibilité : définies par les opérations de gestion des fenêtres. Il a été vérifié que toute note déplacée peut rendre une note visible ou bien en cacher une autre. Cela a été exprimé par l'expression de conditions d'interaction de deux notes ;
- propriétés d'atteignabilité : la combinaison des machines abstraites de gestion de souris et de gestion de notes permet d'assurer que la souris peut manipuler et donc atteindre toutes les fenêtres présentes sur l'écran.

Signalons enfin que la composition de machines abstraites favorise le conception et la réutilisation. En effet chaque machine définit des propriétés qui sont indépendantes les unes des autres. La composition permet alors la création de nouvelles propriétés et assure la cohérence entre les différentes contraintes des machines incluses. Le prouveur de l'Atelier B se chargera de vérifier que toutes les propriétés exprimées sont compatibles.

2.4.2 Conception modulaire : application à l'étude de cas

En comparaison avec l'application du Post-It Notes® et malgré sa simplicité, l'étude de cas que nous proposons, le convertisseur francs/euro et le compteur (présentée dans la section 1.5.3), est beaucoup plus complexe en termes d'IHM (prise en compte du clavier, complexité de la présentation, etc). Des propriétés supplémentaires pourront être exprimées comme par exemple la propriété d'insistance : état de conversion retourné par la couleur des boutons.

Une première phase consiste à modéliser le convertisseur francs/euros puis le compteur de telle façon que ces deux sous-systèmes ne communiquent pas ensemble. Nous avons procédé par une modélisation indépendante des deux applications. C'est une approche qui peut-être rapprochée avec la pratique de conception des systèmes interactifs.

Une seconde phase de modélisation consiste à composer ces deux sous-systèmes pour aboutir au système terminal. De nouvelles propriétés sur ce système pourrront être exprimées et vérifiées par l'intermédiaire de l'outil Atelier B.

2.4.2.1 Description de l'architecture

Comme pour l'étude de cas du Post-It Notes®, la décomposition modulaire du convertisseur francs/euros et du compteur a été représentée en B suivant le modèle d'architecture ARCH. Plusieurs machines abstraites ont été définies et nous les avons représentées partiellement sur la figure 2.4a en ce qui concerne le convertisseur et sur la figure 2.4b pour l'application du compteur. Conformément à l'architecture ARCH, elles correspondent au domaine, à l'adaptateur du domaine, à la présentation et finalement à la boîte à outils.

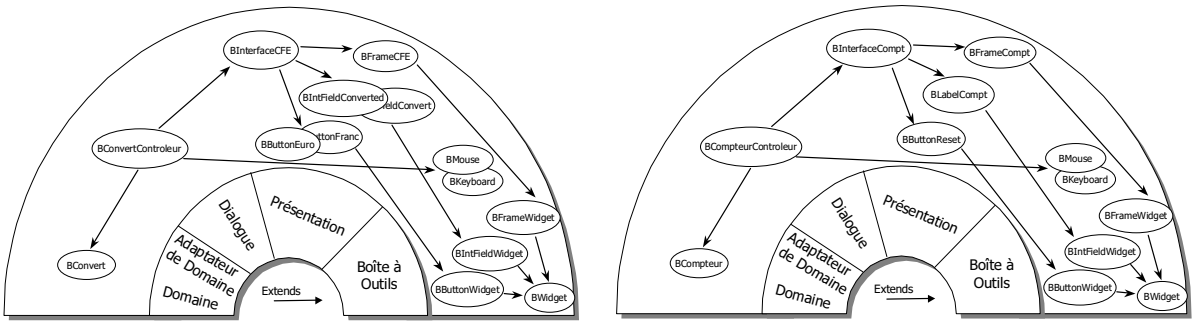

Fig. 2.4 – Relations entre machines abstraites B de la modélisation du convertisseur francs/euros (a) et du compteur (b).

Nous avons respecté la décomposition modulaire suivant le modèle d'architecture ARCH et en utilisant le mécanisme de structuration par la clause **EXTENDS** pour assembler les différentes machines. De plus certaines machines abstraites ont été ré-utilisées comme *BMouse* (qui modélise le périphérique de la souris) ou enrichies comme *BScreenWidget* (qui modélise l'espace de travail des fenêtres).

Boîte à outils. Toutes les machines de la boîte à outils sont communes aux deux modélisations (convertisseur et compteur). Ce module est basé sur la rétro-conception de la boîte à outils JAVA/Swing. Cette rétro-conception est similaire à celle présentée dans la section 2.4.1.2. La machine abstraite *BWidget* est la machine de plus bas niveau. Son rôle est de décrire les ensembles et les valeurs par défaut utilisés par toutes les autres machines. Les machines abstraites *BIntFieldWidget*, *BFrameWidget*, *BButtonWidget*, *BLabelWidget* et *BScreenWidget* s'appuient sur la machine *BWidget*. Elles sont associées à un composant graphique particulier (zone de texte, bouton, etc) et décrivent aussi des ensembles de constantes.

Par ailleurs, la machine *BKeyboard* et *BMouse* décrivent les périphériques d'entrées. La gestion du clavier a été prise en compte pour saisir la valeur à convertir, cependant

pour les besoins de l'étude de cas nous n'avons modélisé que la gestion de saisie de chiffres numériques. Ainsi dans la machine *BKeyboard*, les ensembles qui caractérisent l'état d'une touche du clavier et les différentes touches sur lesquelles l'utilisateur peut interagir sont définis.

Présentation. Le module présentation définit les deux interfaces graphiques. La machine *BInterfaceCFE* spécifie l'interface du convertisseur francs/euros et la machine *BInterfaceCompt* celle du compteur. La modularité, par l'intermédiaire de la clause **EXTENDS** offre l'avantage de pouvoir fournir des bibliothèques réutilisables. Par comparaison avec la pratique de conception des systèmes interactifs, cette modularité est parfaitement adaptée dans le sens où les composants graphiques des boîte à outils sont composés pour aboutir aux interfaces utilisateur.

Dans le cas de l'interface du convertisseur, la machine abstraite *BInterfaceCFE* étend les machines abstraites *BButtonEuro* et *BButtonFranc* pour spécifier les boutons de conversion, *BIntFieldConvert* et *BIntFieldConverted* pour saisir et afficher les valeurs, enfin *BFrameCFE* pour définir la taille et la position de la fenêtre du convertisseur.

Enfin dans le cas de l'interface du compteur, la machine *BInterfaceCompt* étend les machines abstraites *BLabelCompt* pour afficher la valeur du compteur, *BButtonCompt* pour initialiser la valeur du compteur et *BFrameCompt* pour décrire la fenêtre de l'interface.

Noyau fonctionnel et adaptateur de présentation. Les machines *BConvert* et *BCompteur* gèrent à la fois le domaine et l'adaptateur de l'application. Cependant, la modélisation de ces machines n'apporte pas de réelle contribution aux domaines des IHM. Nous avons donc simplifié cette modélisation en ne traitant que des valeurs entières pour traiter la conversion (simplification de la preuve par rapport à des valeurs réelles).

Ainsi, la machine *BConvert* possède des opérations et des variables qui permettent de calculer des valeurs entières en francs ou en euros. Par ailleurs, la machine *BCompteur* décrit des opérations qui permettent d'incrémenter et d'initialiser un compteur.

Contrôleur de dialogue. Le module du contrôleur de dialogue établit le lien entre le noyau fonctionnel et la présentation des deux applications. Dans l'application de notre étude de cas, *BConvertControleur* et *BCompteurControleur* sont des machines de plus haut niveau qui étendent chacune les machines liées aux modules du noyau fonctionnel et de la présentation. Ces machines définissent un ensemble d'opérations sans décrire comment ces opérations sont déclenchées.

A ce niveau de modélisation, nous avons raisonné de la même manière que pour l'application du Post-It Notes® (décomposition modulaire de l'architecture ARCH, retro-conception, etc) pour modéliser la phase de conception des deux sous-systèmes (convertisseur frans/euros et compteur). Par ailleurs, nous avons montré que certains éléments du Post-It Notes® pouvaient être réutilisés (machine *BMouse* par exemple).

Nous allons également montrer dans ce travail deux manières pour coder la composition de sous-systèmes. D'une part, nous réalisons la composition de manière ascendante de deux sous-systèmes afin d'obtenir un système complet ce qui constitue une insuffisance de l'approche à base de modules. D'autre part, nous résolvons cette insuffisance, dans la section 2.5.1 en proposant une approche descendante à base d'événements.

2.4.2.2 Opération de composition

L'opération de composition permet de construire un système complexe à partir de sous-systèmes existants. C'est une conception dite ascendante. Le système est construit par compositions successives.

Du point de vue de l'approche du LISI et de la méthode B, l'opération de composition est obtenue par modularité par l'intermédiaire de la clause **EXTENDS**. Cette clause est utilisée pour remonter l'ensemble des données, des opérations et des propriétés des machines étendues dans une nouvelle machine abstraite afin de décrire le comportement du système complet.

En ce qui concerne notre étude de cas, nous définissons une nouvelle machine abstraite appelée *BControleur* qui va étendre les deux machines de plus haut niveau des modélisations des applications convertisseur et compteur, c'est-à-dire *BConvertControleur* et *BCompteurControleur*. Nous présentons sur la figure 2.5, l'ensemble des machines abstraites utilisées pour modéliser notre étude de cas. Nous avons volontairement simplifié le schéma afin de représenter uniquement le lien de modularité entre la machine *BControleur* et les autres.

Ainsi, la nouvelle machine abstraite a accès à toutes les données et opérations de l'ensemble des machines des deux sous-systèmes et peut alors définir des nouvelles opérations qui servent à établir les transitions entre les variables du convertisseur et du compteur. Nous présentons sur le tableau la liste de ces opérations et leurs descriptifs.

Cette démarche de conception de manière ascendante a été expérimentée sur deux études de cas (le Post-It Notes® d'une part et le convertisseur francs/euros et compteur d'autre part). Par ailleurs, nous ne l'avons pas montré mais toutes les machines sont

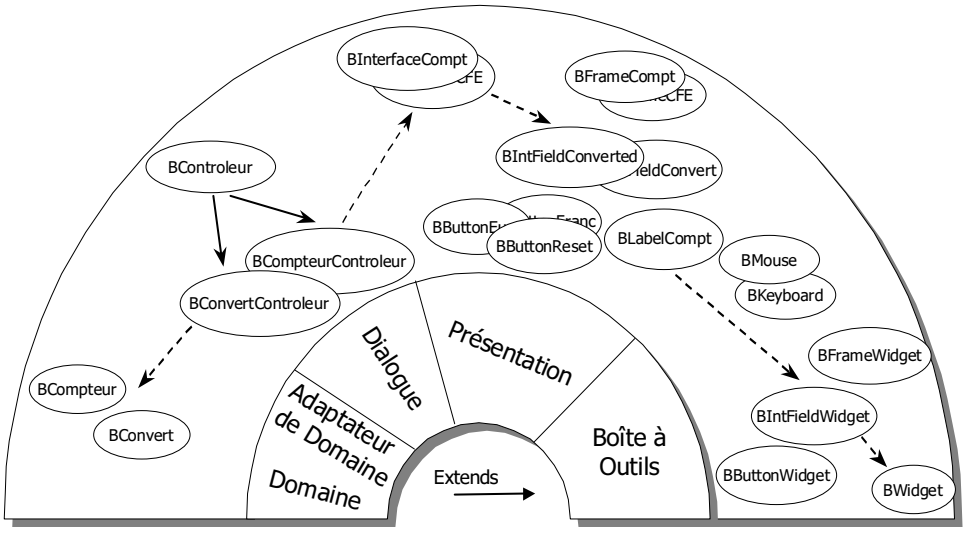

Fig. 2.5 – Composition modulaire ascendante des sous-systèmes convertisseur francs-/euros compteur.

Nomination	Description
opClickQuitButton	Cliquer sur le bouton quitter
opClickQuitApplication	Fermeture de l'application
opInputValue	Saisir un chiffre
opDisplayValue	Met à jour les vues après une conversion
opClickInitialiser	Cliquer sur le bouton initialiser du compteur
opInitialiser	Met à jour les vues après l'initialisation du compteur
opClickEuro	Cliquer sur le bouton $F >> E$
opClickFranc	Cliquer sur le bouton $E >> F$

Tab. 2.11 – Extrait des opérations contenues dans la machine abstraite *BControleur*.

ensuite raffinées jusqu'au code. Nous rappelons que des informations complémentaires à ce sujet sont disponibles dans [AA00a].

2.4.2.3 Vérification de propriétés de robustesse

De la retro-conception de la boîte à outils JAVA/Swing nous avons exprimé des propriétés similaires à l'application Post-It Notes®. Par exemple la propriété sur la souris qui exprime le fait que le pointeur ne peut jamais sortir de l'écran.

En ce qui concerne le module présentation, nous avons obtenu des résultats originaux

comme par exemple des propriétés liées à l'observabilité, à l'insistance ou à la représentation de l'information pour ne citer que les principales.

- la propriété d'**observabilité** qui énonce la capacité pour l'utilisateur à évaluer l'état interne du système a été exprimée en vérifiant que la couleur du texte des boutons de conversion est différente de celle du fond des boutons.

$$\boxed{b\_button\_EF\_textcolor \neq b\_button\_EF\_backgroundcolor}$$

- la propriété d'**insistance** qui énonce la capacité du système à forcer la perception de l'état du système a été traduit par un retour d'information. Concrètement, la présentation retourne le sens de conversion (francs vers euros ou euros vers francs) en modifiant la couleur de fond des boutons de conversion. Pour cela, il faut assurer deux contraintes. La première exprime que si la zone à convertir est vide alors la couleur de fond des deux boutons de conversion doit être celle par défaut : $b\_button\_widget\_color\_no\_selection$. La deuxième exprime qu'il ne peut y avoir qu'un seul et unique bouton dont la couleur de fond est $b\_button\_widget\_color\_selection$ dans l'état de conversion.

$$\boxed{\begin{array}{l}(b\_intfield\_convert\_content = nothing \Rightarrow \\ \quad b\_button\_EF\_backgroundcolor = b\_button\_widget\_color\_no\_selection \wedge \\ \quad b\_button\_FE\_backgroundcolor = b\_button\_widget\_color\_no\_selection) \wedge \\ (b\_intfield\_convert\_content = with \Rightarrow \\ \quad (b\_button\_EF\_backgroundcolor = b\_button\_widget\_color\_selection \wedge \\ \quad b\_button\_FE\_backgroundcolor = b\_button\_widget\_color\_no\_selection) \vee \\ \quad (b\_button\_EF\_backgroundcolor = b\_button\_widget\_color\_no\_selection \wedge \\ \quad b\_button\_EF\_backgroundcolor = b\_button\_widget\_color\_selection)) \\ \quad ((EtatConversion = 3 \ \vee \ EtatConversion = 4) \ \wedge \ EtatCompteur = 1) \ \vee \ \ldots\end{array}}$$

- la **représentation de l'information** et **WYSIWYG** qui ont été exprimées par la focalisation d'un composant graphique de l'interface ou bien par le fait qu'il n'y ait qu'une seule fenêtre active à un instant (interface du convertisseur ou celle du compteur).

Nous avons aussi exprimé des propriétés sur les modules du domaine et de l'adaptateur de domaine. Il s'agit de propriétés de sûreté liées à la manipulation des données de l'application. Par exemple, il s'agit de vérifier que le compteur ne dépasse jamais la valeur 3.

Finalement, nous avons exprimé des propriétés dans le module du contrôleur de dialogue. Il s'agit de propriétés relative au fonctionnement du système, par exemple assurer que l'utilisateur ne peut convertir une somme si $compteur = 3$.

2.4.3 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 à base de modules de conception. 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.

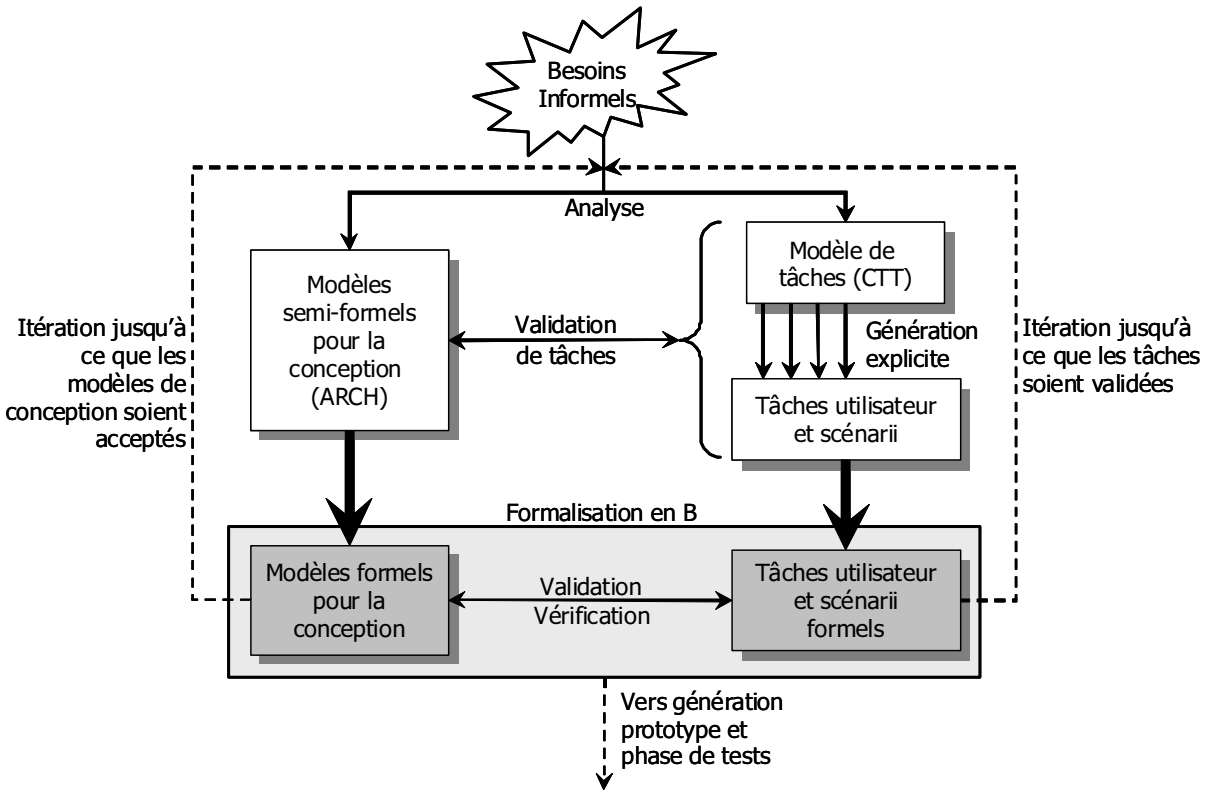

Fig. 2.6 – Description de l'approche explicite.

Sur la figure 2.6 une description de notre approche est proposée où nous distinguons deux aspects de modélisation.

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) 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 testés sur la conception.

Un second aspect concerne l'approche que nous proposons qui est cadrée sur la partie basse de la figure 2.6. 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 2.4.2, puis de modéliser formellement 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 tandis que celle de droite précise la validation de tâches.

Des exemples de scénarii de notre étude de cas pourraient être : *convertir trois euros en francs* ou *convertir six francs en euros*. 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 nous enchaînons sur sa modélisation dans la méthode B. Enfin, nous l'appliquons à l'étude de cas en discutant sur les propriétés vérifiées.

2.4.3.1 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. C'est par exemple la tâche *convertir six francs en euros* où son état initial est *pas de conversion* et son état final est *valeur convertie en euros*.

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 tâches 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 par la suite *opérations atomiques*.

Sur la figure 2.7 nous montrons une décomposition de $Tâche_1$ dans un ensemble de tâches ordonné par $Tâche_2$ et $Tâche_3$ qui sont elles-mêmes ordonnées 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.

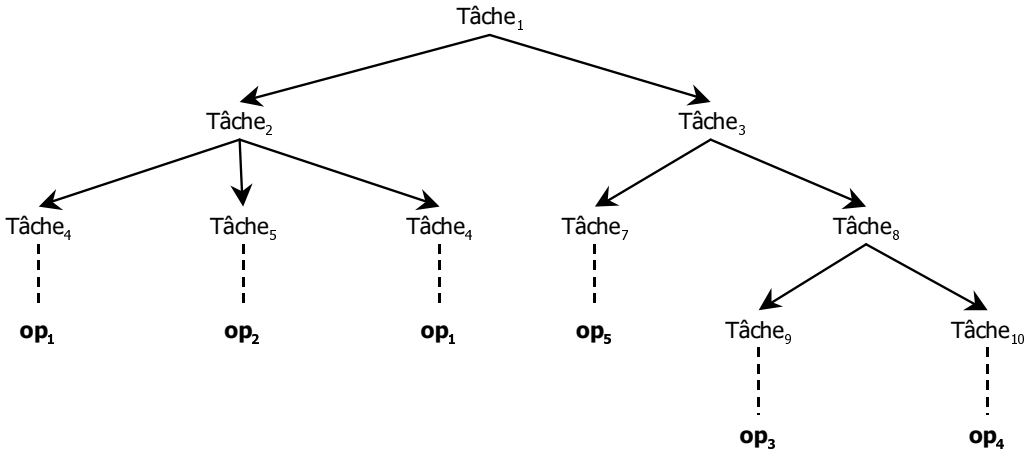

Fig. 2.7 – Décomposition de tâches en sous-tâches.

La représentation de la décomposition est comparable aux modèles de tâches hiérarchiques que nous avons présentés dans la section 1.2.3. Cependant, la sémantique employée pour le décrire est pauvre. Il n'existe pas de précondition de tâches et il ne possède que l'opérateur de séquence pour structurer 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 du diagramme de séquence du formalisme UML[4]. La décomposition complète de l'exemple présenté sur la figure 2.7 est donnée sur le tableau 2.12 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. 2.12 – 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

[4]UML : Unified Modeling Language

l'utilisateur qui s'occupe de la validation ne connaît pas *à priori* la conception. Ce *valideur* se doit alors de 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. Cette représentation hiérarchique des tâches est à rapprocher aux modèles de tâches présentés dans la section 1.2.3. Seulement, la décomposition hiérarchique que nous proposons ne comporte qu'un opérateur : la séquence.

2.4.3.2 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 preuves associées à la technique de raffinement assure 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 ($T\hat{a}che_2$, $T\hat{a}che_3$, ...) sont définies chacune par une opération décrite dans la modélisation de la conception. A l'opposé de la décomposition hiérarchique de la section 2.4.3.1 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 séquencement 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 qui peut être fait de la modélisation). 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 construit 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.

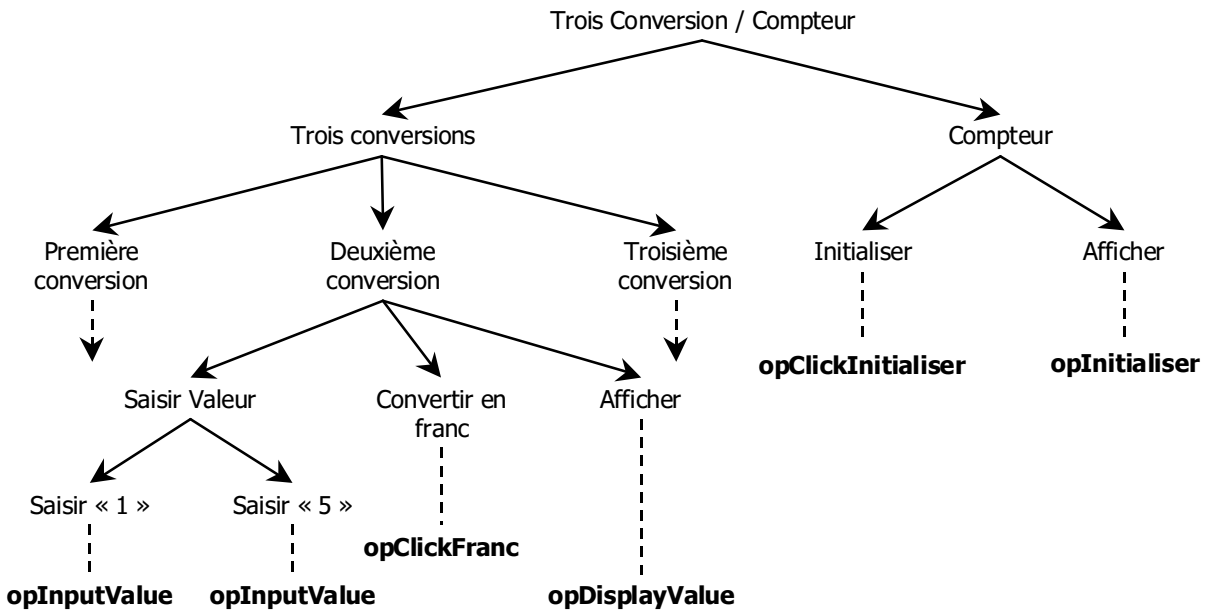

Fig. 2.8 – Décomposition en sous-tâches de la tâche *Trois conversions / Compteur*.

2.4.3.3 Application à l'étude de cas

Afin d'illustrer l'approche explicite, nous proposons de l'appliquer à notre étude de cas. Rappelons que les tâches terminales s'appuient sur des opérations atomiques issues de la machine abstraite *BControleur* où une partie de ces opérations sont décrites dans le tableau 2.11.

Nous avons validé une tâche complexe dont le but est de convertir trois fois une somme

puis d'initialiser le compteur. De façon à simplifier l'écriture, la conversion ne s'effectuera que sur une somme identique (15 euros) et dans le sens euros vers francs. Nous présentons sur la figure 2.8 l'arbre de décomposition de cette tâche.

Nous précisons aussi que les tâches *Première conversion* et *Troisième conversion* se décomposent de la même façon que la tâche *Deuxième conversion*. La modélisation dans la méthode B est obtenue par cinq raffinements successifs correspondants aux cinq niveaux de décomposition de l'arbre de la figure 2.8.

Première machine B : nous décrivons dans la machine abstraite *Tâches Explicites* ci-dessous la tâche *Trois Conversion / Compteur*. Cette machine ne contient qu'une opération unique $Trois\ Conversion\ /\ Compteur$ dont son traitement est d'appeler l'opération $opPostTroisConversionsCompteur$ (non détaillée dans ce mémoire) de la modélisation de la conception pour déterminer la postcondition de cette tâche.

MACHINE *Tâches Explicites*
EXTENDS *BControleur*, ...
INVARIANT
$I(var_i)$
OPERATIONS
Trois Conversion / Compteur =
BEGIN
opPostTroisConversionsCompteur
END ;

La liaison avec le module du contrôleur de dialogue est obtenue par la clause **EXTENDS**. Par ailleurs, l'invariant $I(var_i)$ permet ici d'exprimer de nouvelles propriétés portant sur les variables de la conception.

Premier raffinement : ce premier raffinement permet de décomposer la tâche précédente par les sous-tâches *Trois conversions* et *Compteur*.

REFINEMENT *Tâches Explicites Ref1*
EXTENDS *BControleur, ...*
REFINES *Tâches Explicites*
INVARIANT
$J(var_i, var_j, \ldots)$
OPERATIONS
Trois Conversion / Compteur =
BEGIN
opPostTroisConversions ; *opPostCompteur*
END ;

L'invariant de collage $J(var_i, var_j, \ldots)$ permet de faire correspondre les propriétés du raffinement *Tâches Explicites Ref1* à l'abstraction *Tâches Explicites*. Les opérations *opPostTroisConversions* et *opPostCompteur* (non détaillées dans ce mémoire) décrivent respectivement les postconditions des tâches *Trois conversions* et *Compteur*.

Deuxième raffinement : ce deuxième raffinement introduit les opérations atomiques liées au compteur *opClickInitialiser* et *opInitialiser*.

REFINEMENT *Tâches Explicites Ref2*
EXTENDS *BControleur, ...*
REFINES *Tâches Explicites Ref1*
INVARIANT
$K(var_i, var_j, var_k, \ldots)$
OPERATIONS
Trois Conversion / Compteur =
BEGIN
opPostPremiereConversion ; *opPostDeuxiemeConversion* ;
opPostTroisiemeConversion ;
opClickInitialiser ; *opInitialiser*
END ;

Nous introduisons aussi la décomposition de la tâche *Trois Conversions* par trois opérations qui déterminent les postconditions de chacune des tâches associées. Les opérations *opPostPremiereConversion*, *opPostDeuxiemeConversion* et *opPostTroisiemeConversion* (non détaillées dans ce mémoire) décrivent respectivement les postconditions des tâches *Première conversion*, *Deuxième conversion* et *Troisième conversion*.

Troisième raffinement : le niveau feuille de la tâche *Compteur* est désormais atteint. Ce raffinement ne s'intéresse donc qu'à la décomposition des trois tâches de conversion tout en conservant la décomposition de la tâche *Compteur*.

REFINEMENT *Tâches Explicites Ref3*
EXTENDS *BControleur*, ...
REFINES *Tâches Explicites Ref2*
INVARIANT
$L(var_i, var_j, var_k, var_l, \ldots)$
OPERATIONS
Trois Conversion / Compteur =
BEGIN
/* Décomposition de la tâche Première Conversion */
opPostSaisirValeur ; *opClickFranc* ; *opDisplayValue* ;
/* Décomposition de la tâche Deuxième Conversion */
opPostSaisirValeur ; *opClickFranc* ; *opDisplayValue* ;
/* Décomposition de la tâche Troisième Conversion */
opPostSaisirValeur ; *opClickFranc* ; *opDisplayValue* ;
/* Décomposition de la tâche Compteur */
opClickInitialiser ; *opInitialiser*
END ;

L'invariant de collage $L(var_i, var_j, var_k, var_l, \ldots)$ permet de faire correspondre les propriétés du raffinement à l'abstraction *Tâches Explicites Ref2*. L'opération *opPostSaisirValeur* décrit la postcondition de la tâche *Saisir Valeur*.

Quatrième raffinement : toutes les opérations atomiques du contrôleur de dialogue sont atteintes. Il s'agit du dernier raffinement qui permet d'obtenir une trace d'opérations qui devra être supportée par la modélisation de l'architecture de l'application. Nous donnons ci-dessous cette trace :

REFINEMENT *Tâches Explicites Ref4*
EXTENDS *BControleur, ...*
REFINES *Tâches Explicites Ref3*
INVARIANT
$M(var_i, var_j, var_k, var_l, var_m, \ldots)$
OPERATIONS
Trois Conversion / Compteur =
BEGIN
/* Décomposition de la tâche *Première Conversion* */
opInputValue ; *opInputValue* ; *opClickFranc* ; *opDisplayValue* ;
/* Décomposition de la tâche *Deuxième Conversion* */
opInputValue ; *opInputValue* ; *opClickFranc* ; *opDisplayValue* ;
/* Décomposition de la tâche *Troisième Conversion* */
opInputValue ; *opInputValue* ; *opClickFranc* ; *opDisplayValue* ;
/* Décomposition de la tâche *Compteur* */
opClickInitialiser ; *opInitialiser*
END ;

La validation de la tâche *Trois Conversion / Compteur* va permettre d'exprimer de nouvelles propriétés dans la modélisation et notamment la propriété d'atteignabilité. Par exemple, nous montrons que si la tâche *Trois Conversion / Compteur* est valide alors l'objectif de l'utilisateur de convertir trois sommes puis initialiser est réalisable. C'est aussi de montrer que tant que l'utilisateur n'a pas converti trois sommes, il ne peut initialiser le compteur.

2.4.4 Bilan sur l'approche à base de modules

Nous avons présenté dans cette section l'**approche à base de modules** basée sur l'**approche du LISI** pour la phase de conception que nous avons enrichie par l'**approche explicite** pour la validation de tâches.

La phase de conception se fait de façon ascendante. Elle est 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 validation de 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 sur l'approche à base de modules peuvent être effectuées :

1. la première concerne la phase de conception. L'approche du LISI 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 étudiés dans la section 1.2.3 qui possèdent une forte capacité à structurer, l'approche 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[5]. 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. En effet :

1. **comportement dynamique du contrôleur de dialogue** : avant tout rappelons qu'un système modélisé en B événementiel décrit un système réactif à base d'événe-

[5]UML : Unified Modeling Language

ments. 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 va permettre 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 notre contribution 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.

2.5 Approche à base d'événements

Nous nous proposons d'étudier dans une première partie la modélisation du contrôleur de dialogue par une approche à base d'événements. Nous étudions plus particulièrement la mise en oeuvre du codage du produit synchronisé et nous l'appliquons à l'étude de cas du convertisseur et du compteur.

Dans une deuxième partie, nous décrivons l'approche implicite qui permet la modélisation formelle d'un modèle de tâches CTT et nous l'appliquons directement à notre étude de cas. Enfin une troisième partie dresse le bilan de l'approche à base d'événements.

2.5.1 Modélisation du contrôleur de dialogue à base d'événements

La construction du contrôleur de dialogue se base sur une approche à base d'événements dans le sens où les actions émises par l'utilisateur correspondent à des événements instantanés que ce contrôleur de dialogue doit gérer. Il est donc composé d'un ensemble d'événements gardés définis chacun par un prédicat de garde. Un événement est déclenché lorsque sa garde est vraie. Si deux gardes sont vraies, les événements sont entrelacés. L'ensemble de ces événements définit un système de transitions étiquetées global qui décrit le comportement du système.

2.5.1.1 Produit synchronisé de sous-systèmes.

Notre contribution utilise une conception descendante pour coder la décomposition de systèmes interactifs abstraits en systèmes interactifs concrets. Le raffinement autorise l'apparition de nouveaux événements provenant du codage d'autres systèmes de transitions. Chaque système est alors décrit au fur et à mesure des raffinements de manière incrémentale.

Nous montrons sur la figure 2.9, la décomposition de deux sous-systèmes par une conception descendante. Le modèle abstrait *Model* décrit le premier sous-système STE_1. Le raffinement $Raffinement\ 1$ raffine le modèle *Model* en l'enrichissant de nouveaux événements et de nouvelles variables d'états. Le système STE'_1 obtenu par ce raffinement peut être considéré comme la composition de deux systèmes de transitions STE_{11} et STE_{12} qui ne sont pas explicités. Cette décomposition par raffinement permet donc le codage de l'opération de produit synchronisé sans expliciter les systèmes de transitions qui compose cette opération.

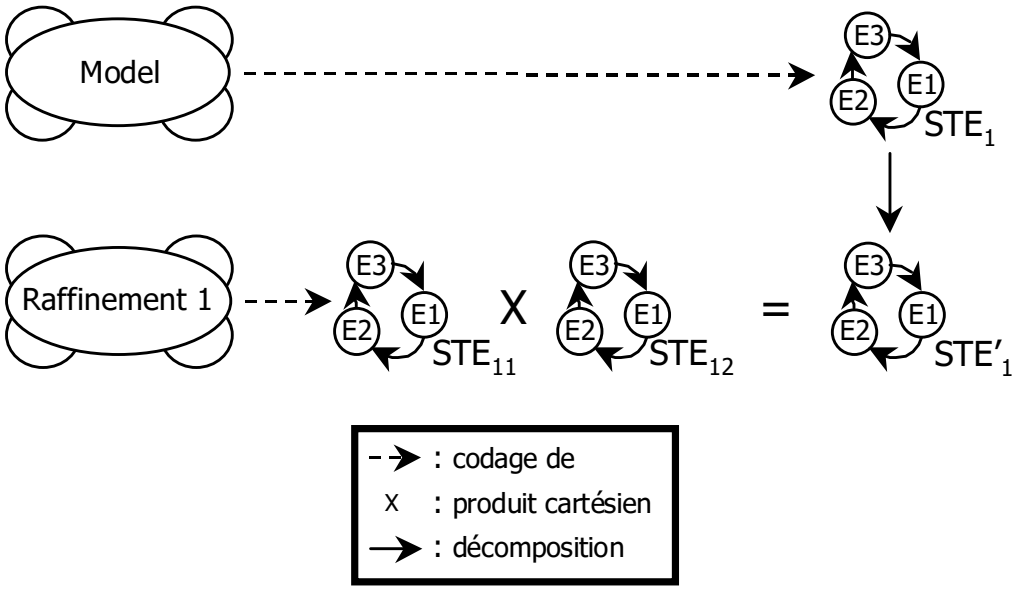

Fig. 2.9 – Différents niveaux de décomposition de sous-systèmes.

Du point de vue de la construction des raffinements, plusieurs points sont pris en compte :

- le raffinement d'un modèle B est un composant qui conserve la même interface que le modèle abstrait ce qui implique que les événements correspondant à l'ensemble des transitions du système à composer doivent tous raffiner la substitution *identité skip* ;

– de nouvelles variables décrivant l'état du système sont introduites pendant le raffinement. En ce qui concerne l'approche que nous proposons, l'ajout de variables est obtenu par extension de machines abstraites qui apportent des variables et des opérations issues de la modélisation de l'architecture ;

– l'opération de composition amène une description de ce qui a été spécifiée dans l'abstraction. Les gardes des événements s'enrichissent de contraintes supplémentaires dues aux variables des systèmes composés. Par conséquent, la disjonction des gardes décrite dans la clause **ASSERTIONS** est reformulée tout en conservant son comportement de l'abstraction.

2.5.1.2 Application à notre étude de cas

La figure 2.10 présente une partie du système de transitions étiquetées de l'application convertisseur francs/euros. Nous avons volontairement réduit le nombre d'états et de transitions afin d'alléger la figure.

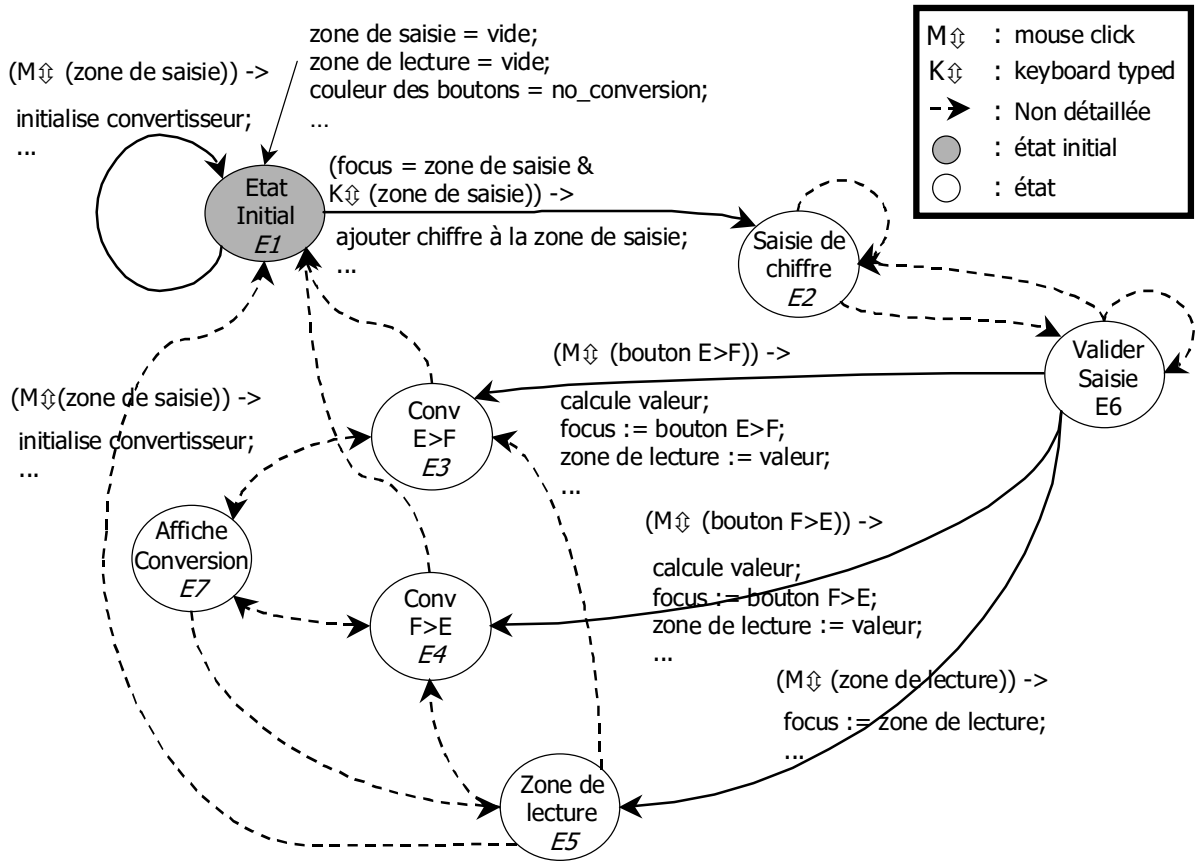

Fig. 2.10 – Système de transitions étiquetées du convertisseur francs/euros.

Nous avons décrit dans cet automate tous les aspects interactifs de la section 1.5.3.2. Par exemple, nous avons modélisé le fait que le passage de l'état initial $E1$ à $E2$ ne peut se faire que si la zone de saisie a la focalisation de l'interface *focus = zone de saisie* et si l'utilisateur appuie sur une touche du clavier $K\updownarrow$*(zone de saisie)*. Si cette garde est respectée l'événement *ajouter chiffre à la zone de saisie* peut être déclenché. Nous avons aussi modélisé le choix de conversion qui correspond aux états *E3* et *E4*. Après avoir saisi un ou plusieurs chiffres (état *E2*) et validé la saisie (état *E6*), l'utilisateur peut atteindre les états *E3* et *E4* en effectuant un click sur le bouton $E > F$ ou sur le bouton $F > E$. Enfin, la valeur de conversion est affichée dans la zone de lecture par le passage de l'état *E3* à *E7* ou *E4* à *E7*. Les événements associés à ces transitions (non détaillés sur la figure) modifient à la fois la valeur de la zone de saisie et la couleur des boutons afin que l'IHM retourne l'état de conversion.

Le système de transitions de l'application compteur est présenté quant à lui sur la figure 2.11. Nous avons identifié deux états $E1$ et $E2$ correspondant en fait à l'état de la variable *compteur*. La transition de l'état $E1$ à l'état $E1$ ne peut se faire que si *compteur* $<$ 3. Son action incrémente d'une unité la valeur du compteur. La transition de l'état $E1$ à $E2$ modélise quant à elle le fait que si *compteur* $= 3$ le bouton d'initialisation est activé. Enfin, quand l'utilisateur appuie sur le bouton reset, l'action de cette transition initialise le compteur (*compteur* $= 0$) et désactive le bouton.

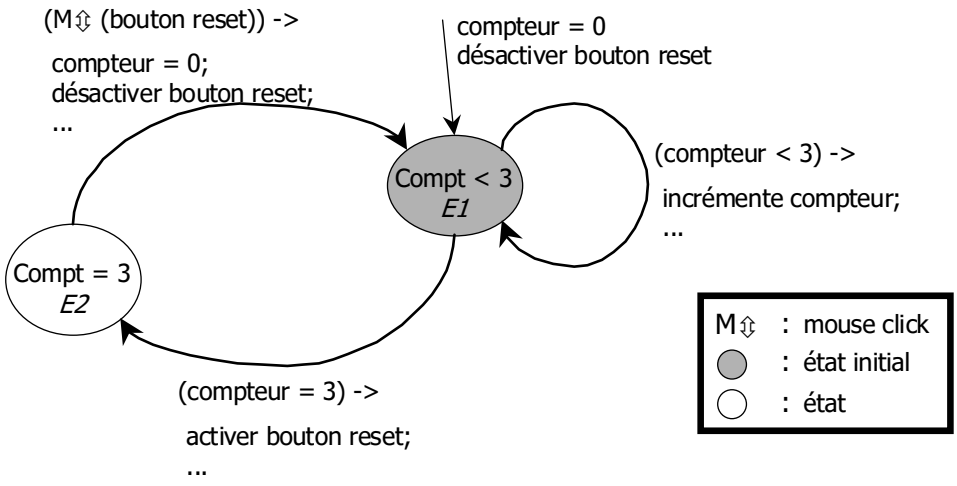

Fig. 2.11 – Système de transitions étiqueté du compteur.

Nous venons de décrire explicitement les systèmes de transitions des deux applications. Nous allons à présent nous intéresser au codage en B événementiel de ces systèmes par conception descendante qui ne permet pas la description explicite de ces automates.

La spécification du modèle *BControleurConvertisseur* ci-dessous décrit le système de transitions du convertisseur. Les machines abstraites issues de la description de l'architec-

ture ARCH (section 2.4.2.1) concernant le convertisseur sont étendues : $BInterfaceCFE$, $BCompteur$, $BMouse$ et $BKeyboard$. Cette extension permet d'utiliser les éléments de l'architecture. Le contenu de la clause **ASSERTIONS** sera détaillé dans la section 2.5.1.3.

MODEL *BControleurConvertisseur*
EXTENDS
$BInterfaceCFE$, $BCompteur$, $BMouse$, $BKeyboard$
...

Nous nous attardons à décrire l'événement *evtClickFranc* qui traduit la transition de l'état *E6* à *E3* (figure 2.12). Sa garde définit un prédicat qui conditionne l'état de l'automate par les variables d'état de l'application. Ces dernières expriment le fait que le curseur de la souris doive être positionné sur le bouton, que ce bouton ne soit pas déjà pressé et que l'état de la souris soit cliqué.

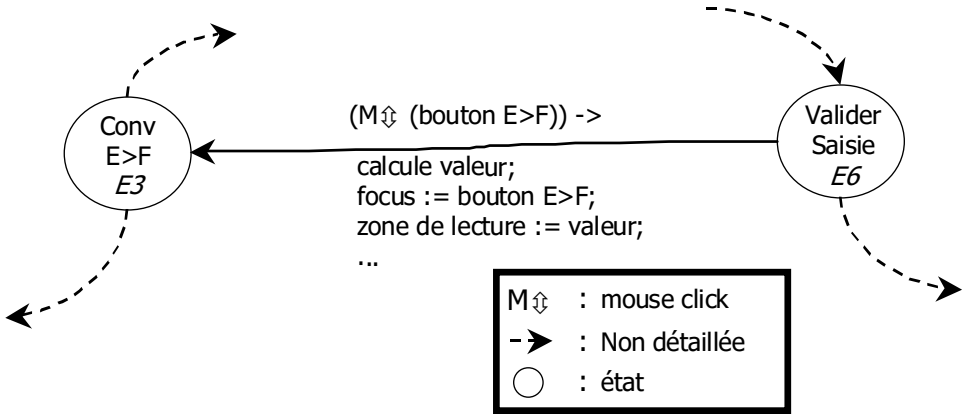

Fig. 2.12 – Transition de l'état *E6* à *E3* de l'automate du convertisseur francs/euros.

EVENTS
$evtClickFranc$ =
SELECT
$b\_mouse\_state = clicked \wedge$
$b\_mouse\_x \in b\_frame\_x + b\_button\_EF\_x..b\_button\_EF\_wide + \ldots \wedge$
$b\_mouse\_y \in b\_frame\_y + b\_button\_EF\_y..b\_button\_EF\_high + \ldots \wedge$
$b\_buttonEF\_state = unpressed$

L'action de cet événement explique comment sont modifiées les variables d'état de l'automate c'est-à-dire les variables de l'application. Nous utilisons pour cela des opérations de la conception. La première ($convertir\_euro\_franc$) est une opération du

module domaine et permet de convertir la somme saisie en franc. La seconde opération ($setInterfaceCFEFocusOnButtonFranc$) change la focalisation de l'interface par l'objet bouton et la dernière ($setBackgroundColorFrancSelection$) modifie la couleur des boutons de conversion pour retourner l'état de conversion. Notons que dans la figure 2.12 de l'automate, ces opérations sont appelées en séquence. Cet ordre d'appel n'a pas d'influence sur le résultat de l'action de la transition. Nous pouvons par conséquent utiliser l'opérateur parallèle (||) de la méthode B qui décrit un appel simultané aux opérations. Au contraire, si l'ordre d'appel aux opérations de la conception avait une influence sur le résultat de l'action de la transition, nous aurions dû utiliser des événements supplémentaires pour traduire le séquencement d'appel aux opérations.

THEN
$convertir\_euro\_franc$ ||
$setInterfaceCFEFocusOnButtonFranc$ ||
$setBackgroundColorFrancSelection$ ||
...
END ;
...

La figure 2.13 présente un extrait du système de transitions obtenu par produit synchronisé entre les automates du convertisseur et du compteur. Nous allons maintenant montrer comment est effectué le codage du produit synchronisé en B événementiel.

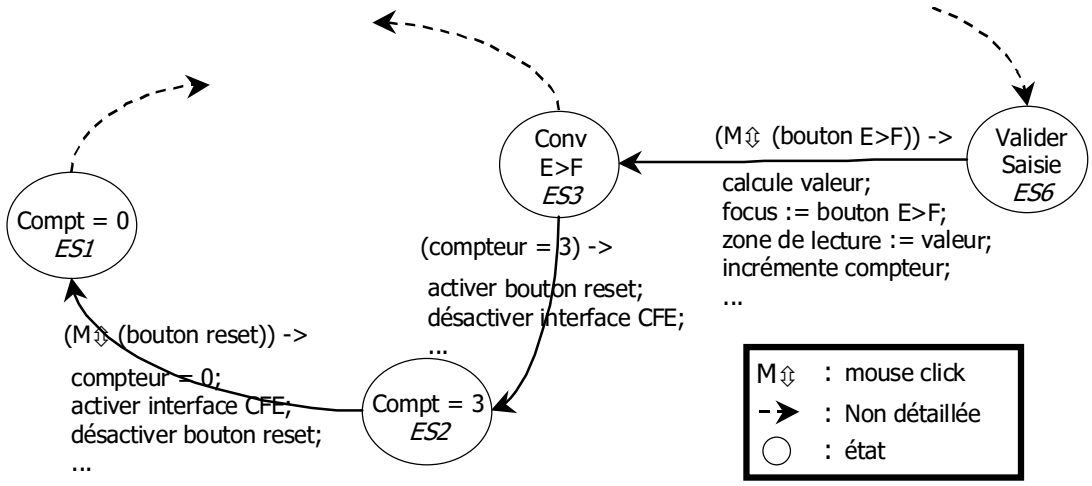

Fig. 2.13 – Extrait du système de transitions obtenu par la composition des systèmes de transitions du convertisseur et du compteur.

Le raffinement $BControleurConvertisseurCompteur$, présenté ci-dessous, compose les systèmes de transitions étiquetées relatifs aux systèmes du convertisseur et du compteur. Il

complète la spécification précédente par la prise en compte des machines de l'architecture de l'application du compteur.

REFINEMENT *BControleurConvertisseurCompteur*
REFINE *BControleurConvertisseur*
EXTENDS
BInterfaceCFE, *BCompteur*, *BMouse*, *BKeyboard*, *BInterfaceCompt*, *BConvert*
...

BControleurConvertisseurCompteur enrichit le modèle *BControleurConvertisseur* en introduisant des variables issues de la conception du compteur. Nous présentons ci-dessous l'enrichissement de l'événement *evtClickFranc* décrivant la transition de l'état $ES6$ à $ES3$. Le corps de cet événement modifie les variables d'état des applications par appels d'opérations. L'opération *incrementeCompteur* modifie par exemple le noyau fonctionnel de l'application du compteur.

$evtClickFranc$ =
SELECT
$b\_mouse\_state = clicked \wedge$
$b\_mouse\_x \in b\_frame\_x + b\_button\_EF\_x..b\_button\_EF\_wide + \ldots \wedge$
$b\_mouse\_y \in b\_frame\_y + b\_button\_EF\_y..b\_button\_EF\_high + \ldots \wedge$
$b\_buttonEF\_state = unpressed$
THEN
$convertir\_euro\_franc \parallel$
$setInterfaceCFEFocusOnButtonFranc \parallel$
$setBackgroundColorFrancSelection \parallel$
$incrementeCompteur \parallel$
...
END;

Par ailleurs, le raffinement enrichit le modèle *BControleurConvertisseur* par l'introduction de nouveaux événements qui, rappelons-le raffinent tous *skip*. Nous présentons ci-dessous l'événement *evtClickInitialiser* qui décrit l'action de l'utilisateur sur le bouton Initialiser (transition de l'état $ES2$ à $ES1$). La garde de cet événement vérifie que l'interface du convertisseur est désactivée, que le curseur de la souris est positionné sur le bouton d'initialisation, que le bouton est activé ($b\_button\_init\_state = unpressed$) et que la valeur du compteur est égale à trois.

EVENTS
evtClickInitialiser =
SELECT
$b\_frame\_cfe\_state = desactivated \ldots$
$b\_mouse\_x \in b\_frame\_x + b\_button\_init\_x..b\_button\_init\_wide + \ldots \wedge$
$b\_mouse\_y \in b\_frame\_y + b\_button\_init\_y..b\_button\_init\_high + \ldots \wedge$
$b\_button\_init\_state = unpressed \wedge$
$b\_mouse\_state = clicked \wedge$
$compteur = 3 \wedge$
$\ldots$

La substitution de l'événement *evtClickInitialiser* modifie la variable du compteur (opération *initialiseCounter* de la machine Compteur), active l'interface du convertisseur et désactive le bouton d'initialisation de l'interface du compteur.

THEN
$initialiseCounter \parallel$
$setActivateInterfaceCFE \parallel$
$setDesactivateBoutonReset \parallel$
$\ldots$
END ;

La figure 2.14 présente la composition descendante des sous-systèmes convertisseur francs/euros et compteur obtenue par raffinement. Cette figure ne présente pas entièrement les machines qui décrivent les modules de l'architecture ARCH ni les liens entre ces machines (flèches en pointillées).

Le modèle *BControleurConvertisseur* nous a permis de coder le système de transitions du convertisseur francs/euros en exploitant (par la clause **EXTENDS**) uniquement les éléments de l'architecture de cette application (flèches vers les machines de la présentation et du noyau fonctionnel du convertisseur).

Pour composer l'application du compteur avec celle du convertisseur nous avons utilisé la technique de raffinement qui réalise la composition par une conception descendante. Les variables d'état de l'application ont été introduites par la clause **EXTENDS** (flèches vers les machines de la présentation et du noyau fonctionnel du convertisseur et du compteur). Le raffinement nous a permis de coder le produit synchronisé entre les deux applications sans expliciter les systèmes de transitions.

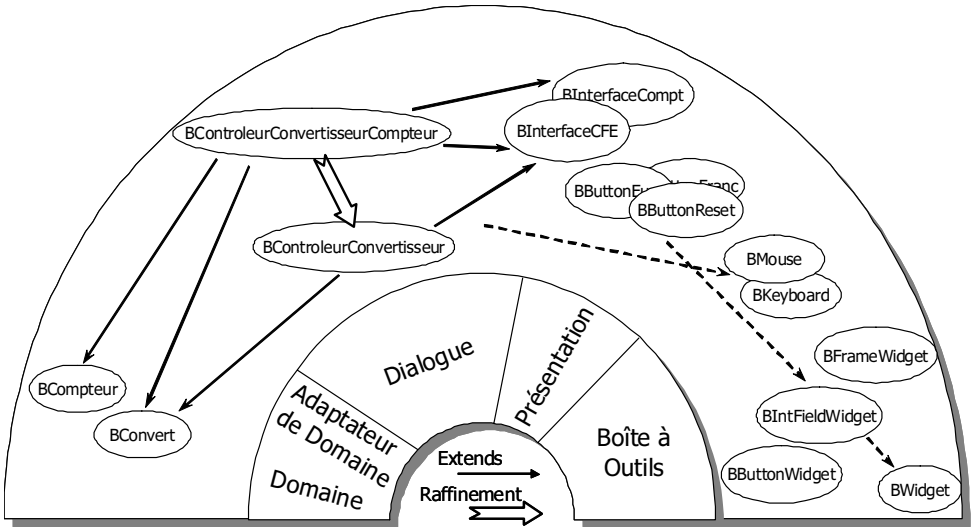

Fig. 2.14 – Décomposition en sous-systèmes convertisseur francs/euros et compteur.

2.5.1.3 Propriétés du contrôleur de dialogue

La contribution que nous venons de présenter permet l'expression de propriétés supplémentaires qui n'étaient pas modélisées avec l'approche du LISI. La propriété de non-blocage par exemple a été prouvée grâce à l'utilisation de l'extension B événementiel qui permet à l'aide de la disjonction des gardes d'exprimer cette famille de propriétés.

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

ASSERTIONS
/* Garde de l'événement *evtClickFranc* */
$(b\_frame\_cfe\_state = desactivated \wedge compteur = 3 \wedge$
$b\_mouse\_y \in b\_frame\_y + b\_button\_init\_y..b\_button\_init\_high + \ldots \wedge$
$b\_button\_init\_state = unpressed \wedge b\_mouse\_state = clicked \wedge \ldots) \vee$
/* Garde de l'événement *evtClickFranc* */
$(compteur = 3 \wedge$
$b\_mouse\_y \in b\_frame\_y + b\_button\_EF\_y..b\_button\_EF\_high + \ldots \wedge$
$b\_mouse\_state = clicked \wedge b\_buttonEF\_state = unpressed \wedge \ldots) \vee$
/* Garde de l'événement *evtInitialiser* */
$((compteur = 3 \vee Eb\_interface\_cfe\_focus = f\_button\_franc \vee \ldots) \vee$
/* Garde des autres événements */
...

Notons aussi que la spécification du contrôleur de dialogue étend toutes les machines des autres modules. Par conséquent les propriétés liées à la présentation, à la boîte à outils ou au domaine se retrouvent toutes dans la modélisation de ce module. Les propriétés sont alors établies une seule fois au niveau où elles sont introduites dans la modélisation, l'invariant de collage permet de les préserver lors du raffinement.

Avec l'aide de l'outil Atelier B, les obligations de preuve correspondant aux propriétés sont générées automatiquement et sont déchargées par le prouveur automatique et interactif. Les obligations de preuve non déchargées permettent de décider si la modélisation de l'application doit être modifiée et le processus formel de développement réitéré (comme le montre la figure 1.14), il s'agit de l'aspect validation de la conception.

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

Les insuffisances de l'approche explicite évoquées en section 2.4.3 nous ont conduit à proposer une extension à l'approche explicite où nous employons un langage de modélisation de tâches utilisateurs, 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 puisque d'autres opérateurs de contrôle sont introduits. Il sera inutile d'énumérer toutes les tâches à valider. Elles seront simplement caractérisées par le modèle de tâches CTT.

L'approche implicite que nous présentons est résumée sur la figure 2.15. Nous distinguons sur cette figure les mêmes aspects que sur la figure 2.4.3, c'est-à-dire la partie haute qui concerne la modélisation par des modèles et notations semi-formelles 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 2.4.3) 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. Nous parlons dans ce cas de génération implicite de tâches utilisateur et scénarii formels.

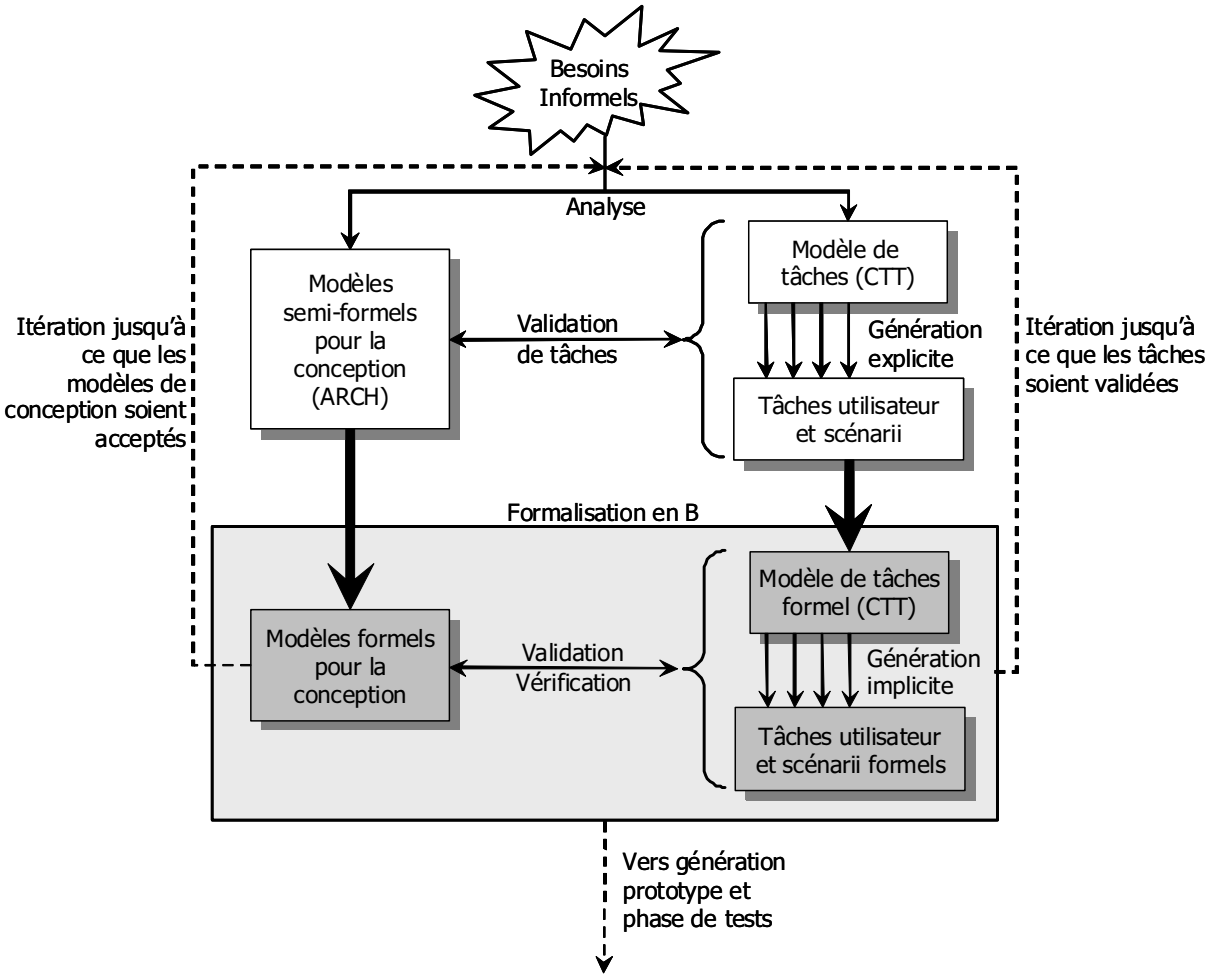

Fig. 2.15 – Description de l'approche implicite.

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.

Nous décrivons formellement la sémantique de CTT dans la méthode B suivant les règles de la grammaire BNF[6] de CTT proposée dans le tableau 2.13. Elle présente aussi d'une manière différente que la section 1.2.3.2 et l'annexe A, 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.

[6]Backus Naur Form

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. 2.13 – Grammaire BNF de la notation de tâches ConcurTaskTrees.

En examinant cette grammaire, deux opérateurs de la notation originelle sont absents (concurrence et activation avec passage d'information). Nous reviendrons sur cette absence lors de la formalisation des opérateurs.

Ces règles de grammaire ne présentent que l'aspect temporel du déroulement entre tâches. Cet aspect est défini par la sémantique de Lotos [Sys84]. 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 est la suivante : nous présentons dans un premier temps la traduction du langage CTT en B événementiel puis la formalisation des informations d'une tâche CTT dans un second temps. Nous regroupons ces résultats afin de montrer la construction complète d'un arbre de tâches au moyen d'une étude de de cas. Enfin nous discutons les apports des techniques de preuve face aux techniques de vérification sur modèle dans ce contexte de formalisation.

2.5.2.1 Formalisation de la grammaire CTT

Deux étapes ont été nécessaires pour traduire formellement la grammaire CTT dans la méthode B. La première n'utilise que la méthode B sous sa forme classique et a montré ses faiblesses quant à la traduction de plusieurs constructions du langage CTT comme l'itération et la concurrence. La seconde étape exploite les caractéristiques de B événementiel. Elle a permis de traduire l'intégralité des opérateurs et caractéristiques de CTT.

Nous montrons dans la section concernant la modélisation des opérateurs en B événementiel que les opérateurs basiques *activation*, *choix*, *itération*, *concurrence* et *tâche*

atomique sont suffisants pour coder l'intégralité des opérateurs CTT sous réserve que la sémantique utilisée soit une sémantique d'entrelacement à base de traces. C'est uniquement à partir de cette hypothèse que nous pourrons donner des règles de transformation des constructions CTT (opérateurs temporels et caractéristiques de tâche) dans la méthode B.

Transformation des constructions CTT en opérateurs basiques. Nous avons isolé deux catégories d'opérateurs : les *opérateurs basiques* et les opérateurs restants. Les premiers permettent de coder les seconds. Les opérateurs basiques que nous avons définis sont les suivants : *activation*, *choix*, *tâche itérative*, *concurrence* et *tâche atomique*.

Ordre indépendant. Si nous considérons la tâche $T_0 ::= T_1 \models\mid T_2$, qui correspond à l'activation dans n'importe quel ordre de T_1 et T_2, sa transformation en opérateurs basiques est donnée par toutes les combinaisons des tâches soit :

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

Tâche optionnelle. Nous considérons la caractéristique optionnelle de la tâche notée $T_0 ::= [T_1]$ qui traduit le fait que la tâche T_1 peut avoir lieu ou pas :

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

Où T_{Skip} est une tâche vide qui ne fait rien.

Désactivation. Considérons la tâche $T_0 ::= T_1[> T_2$ où T_1 est désactivable par T_2. La désactivation admet deux cas selon que T_1 soit atomique ou non.

Dans le cas où T_1 est une tâche atomique, la désactivation indique que soit T_1 est exécutée complètement ou bien elle ne l'est pas et T_2 est exécutée. Nous obtenons une traduction par l'opérateur choix :

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

Par contre si T_1 n'est pas une tâche atomique la désactivation peut intervenir dans un état de la décomposition de la tâche T_1 (états observables). Ainsi, 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 en opérateurs de base est obtenue par le choix de toutes les traces possibles terminées par T_2 :

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

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

Désactivation d'une tâche itérative. Considérons la tâche $T_0 ::= T_1^*[> T_2$. Dans ce cas, la traduction par des opérateurs de base utilise le même raisonnement que pour l'opérateur de désactivation précédent.

Si T_1 est une tâche atomique, alors T_1 est exécutée un nombre arbitraire de fois (éventuellement nul) et ensuite la tâche T_2 est activée. Nous obtenons l'écriture suivante :

$T_0 ::= T_1^*[> T_2$ est traduite en $T_0 ::= (T_1)^N\ >>\ T_2$ où N est un entier naturel.

Par ailleurs, si T_1 n'est pas une tâche atomique et son écriture est de la forme $T_{1,1}$ - $op_1\ T_{1,2}\ op_2 \cdots\ op_n\ T_{1,n+1}$ alors sa traduction par des opérateurs de base est obtenue par :

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

$$\begin{aligned} T_0 ::= \quad & T_2\ [] \\ & (T_1)^{N_1}\ >>\ T_{1,1}\ >>\ T_2\ [] \\ & (T_1)^{N_2}\ >>\ T_{1,1}\ op_1\ T_{1,2}\ >>\ T_2\ [] \\ & (T_1)^{N_3}\ >>\ T_{1,1}\ op_1\ T_{1,2} \cdots op_i\ T_{1,i+1}\ >>\ T_2\ [] \\ & \cdots [] \\ & (T_1)^{N_i}\ >>\ T_2 \end{aligned}$$

où N_i est un entier naturel arbitraire (éventuellement nul).

Le même raisonnement s'applique au tâche $T_{1,i}$ si elles ne sont pas atomiques.

Interruption. Considérons la tâche $T_0 ::= T_1 |> T_2$. La tâche T_2 peut suspendre l'exécution de la tâche T_1 et lorsque T_2 se termine, l'exécution de T_1 reprend. Suivant le même raisonnement que pour l'opérateur désactivation, l'opérateur interruption admet différents cas suivant la nature de la tâche T_1 (atomique ou non).

Si T_1 est une tâche atomique, alors T_2 peut être exécutée un nombre arbitraire de fois, ensuite T_1 est exécutée. Dans ce cas, nous obtenons la translation en opérateurs basiques :

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

Par ailleurs, si T_1 n'est pas une tâche atomique, l'interruption peut se produire autant de fois dans chaque état observable de T_1 définie comme $T_{1,1}\ op_1\ T_{1,2}\ op_2 \cdots\ op_n\ T_{1,n+1}$. La traduction par des opérateurs de base est définie par le choix de toutes les traces possibles :

$$\begin{array}{c} T_0 ::= T_{1,1}\ op_1\ T_{1,2}\ op_2 \cdots\ op_n\ T_{1,n+1} |> T_2 \text{ est traduite en} \\ T_0 ::= T_2^{N_1}\ >>\ T_{1,1}\ op_1\ T_2^{N_2}\ >>\ T1,2 \cdots op_n\ T_2^{N_{n+1}}\ >>\ T_{1,n+1} \end{array}$$

Ici les N_i sont des nombres naturels arbitraires qui montrent que l'interruption peut se produire zero, une ou plusieurs fois.

Le même raisonnement s'applique au tâche $T_{1,i}$ si elles ne sont pas atomiques.

Représentation des constructions CTT en B classique. Seuls les opérateurs de *choix* et *activation* ont une correspondance directe avec des substitutions de la méthode B. L'opérateur d'itération ne peut trouver une correspondance directe. Quant à la traduction de l'opérateur de concurrence, cela revient à procéder de la même manière que l'approche explicite, c'est-à-dire décrire tous les entrelacements en décrivant les différentes combinaisons.

Notons aussi que nous avons volontairement décrit la représentation des constructions CTT dans la méthode B classique afin de justifier le choix de l'extension B événementiel pour traduire ces constructions.

Pour toutes les règles de traduction présentées ci-dessous, nous noterons T_i pour les tâches, S_i pour des substitutions qui sont associées respectivement aux tâches T_i. Une substitution S_i peut être une affectation ou le déclenchement d'événements atomiques issus du module du contrôleur de dialogue.

Activation. Si nous considérons la tâche $T_1 >> T_2$ signifiant que l'exécution de T_1 est suivie par celle de T_2. La traduction en B est obtenue par la substitution généralisée *séquencement* qui correspond à l'application en séquence de deux substitutions. Elle est traduite de la façon suivante :

$$S_1 \ ; S_2$$

Choix. La tâche $T_1[]T_2$ décrit une situation où il existe un nombre fini de comportements possibles sans préciser lequel sera choisi. Elle définit donc un comportement non déterministe. L'opérateur choix est donc traduit par la substitution *choix borné*. Ainsi cet opérateur peut être traduit en B par :

CHOICE
S_1
OR
S_2
END

Tâche itérative. Dans la sémantique de la notation CTT, deux types d'itération sont définis : la tâche itérative qui se répète tant qu'une autre tâche ne la désactive pas et la tâche itérative finie qui se répète n fois à moins qu'une autre tâche ne la désactive. Toutefois, la méthode B classique ne facilite pas la description d'éléments itératifs même s'il existe une substitution *boucle tant que*. Cependant, cette substitution est difficile à modéliser et ne peut être utilisée qu'au niveau implémentation. Ce point peut être partiellement résolu en ce qui concerne l'itération finie où, dans ce cas, la boucle est traduisible au moyen d'une trace. Elle consiste alors à énumérer toutes les occurrences de la boucle un peu à la manière de l'approche explicite. Dans l'exemple de la tâche T_1^N, la traduction en B se fait par :

$$S_1 \ ; S_1 \ ; S_1 \ ; S_1 \ ; S_1 \ ; S_1 \ ; S_1 \ ; S_1 \ ; S_1 \ ; S_1 \ ; S_1 \ ; ... \ ; S_1$$

S_1 est ici répétée N fois. Cette solution n'est donc pas valable en ce qui concerne la construction de l'itération où N est un nombre naturel arbitraire. Il faudrait alors énumérer explicitement toutes les occurences, pour chaque valeur de N dans l'ensemble des entiers naturels. Cette démarche aboutirait à un nombre élevé de tâches à décrire (et donc à prouver). L'opération de concurrence reprend ce même problème.

Représentation en B événementiel des constructions de base de CTT. Les traductions en B classique des constructions de base ont montré des insuffisances que nous proposons de contourner. C'est pour cette raison que nous nous sommes intéressés dans une seconde étape à employer le B événementiel.

Nous sommes partis de la description d'une tâche atomique T_i au moyen d'un événement Evt_i. Un événement est défini par sa garde P_i et sa substitution généralisée S_i (peut être un événement atomique du contrôleur de dialogue). S_i correspond aux actions exécutées par la tâche T_i et P_i la condition de déclenchement de cette tâche.

En outre, nous avons choisi d'introduire la notion de variable entière positive appelée par la suite *variant* qui décroît vers 0. Les variants identifient l'état du modèle de tâches et plus précisément la tâche à exécuter. Le variant décrit les contraintes de précédences entre les tâches. La notion de variant est utilisée dans les raffinements et dans les nouveaux événements qui doivent le faire décroître pour assurer que les événements de l'abstraction sont déclenchés. Quand le variant est nul, les événements du *raffinement* rendent le contrôle aux événements de l'abstraction. Les événements de l'abstraction reprennent la main et sont déclenchables. La représentation des variants avec l'Atelier B a nécessité l'utilisation des variables des machines pour coder les différents cas correspondants aux différentes valeurs du variant.

Les prédicats P_i sont décrits au moyen d'expressions combinant les variants et les variables d'état de la conception. Les substitutions S_i, quant à elles, modifient les variables d'état et les variants. Enfin, les clauses **ASSERTIONS** et **INVARIANT** sont utilisées pour modéliser les conditions nécessaires (sur les variants et sur les variables d'état de conception) pour contrôler la dynamique de l'exécution des tâches.

Dans la clause **INVARIANT** nous exprimons l'invariant de collage entre les variables du niveau abstrait et celles du niveau concret. Plus précisément, cet invariant permet de garantir que la postcondition d'une tâche mère est établie lorsque les tâches filles sont déclenchées selon l'opérateur utilisé. Nous exprimons aussi un invariant qui type les variants. Finalement, nous ajoutons aussi un invariant qui permet d'exprimer des propriétés supplémentaires sur les variables de la conception.

Par ailleurs, nous assurons dans la clause **ASSERTIONS** qu'il n'y a pas de blocage en écrivant que la disjonction des gardes abstraites implique ($\Rightarrow$) la disjonction des gardes concrètes.

Nous donnons dans la suite le codage des opérateurs temporels en insistant sur les entrelacements possibles, nous n'insistons pas sur la correction fonctionnelle. Cette dernière est garantie par l'invariant de collage.

MODEL $Tache_{T0}$
INVARIANT $I(var_i)$

EVENTS
$Evt_0 =$
SELECT
G_0
THEN
S_0
END ;

Par ailleurs, nous exploiterons la modélisation B événementiel ($Tache_{T0}$) de la tâche de plus haut niveau T_0 qui nous servira tout au long de la description B événementiel des opérateurs de base. Elle est représentée par l'événement Evt_0. Notons aussi la forme de l'invariant $I(var_i)$ qui permet entre autre la description de propriétés sur les variables d'état (var_i) du modèle.

Activation. Considérons $T_0 ::= T_1 >> T_2$ sa traduction en B événementiel est donnée par le raffinement $RefActivationTache_{T0}$ (raffine le modèle $Tache_{T0}$ décrivant la tâche T_0) avec deux événements (Evt_0, Evt_1 et Evt_2) et un variant $EtatAct$ qui décroît.

RAFFINEMENT $RefActivationTache_{T0}$
REFINE $Tache_{T0}$
INVARIANT
$J(var_i, var_j) \wedge EtatAct \in \{0,1,2\} \wedge J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((EtatAct = 2 \wedge G_1) \vee (EtatAct = 1 \wedge G_2) \vee (EtatAct = 0 \wedge G_0))$
INITIALISATION
$EtatAct := 2$

EVENTS

$Evt_1 =$	$Evt_2 =$	$Evt_0 =$
SELECT	**SELECT**	**SELECT**
$EtatAct = 2 \wedge G_1$	$EtatAct = 1 \wedge G_2$	$EtatAct = 0 \wedge G'_0$
THEN	**THEN**	**THEN**
$EtatAct := 1 \parallel$	$EtatAct := 0 \parallel$	S'_0
S_1	S_2	**END ;**
END ;	**END ;**	

Le rôle de l'initialisation (clause **INITIALISATION**) est de déterminer l'événement qui sera déclenché en premier, dans ce cas il s'agit de l'événement Evt_1. Le déclenchement de cet événement modifie le variant *EtatAct* de tel sorte que l'événement Evt_2 puisse être déclenché en séquence (garde respectée). Il est à noter que le déclenchement de l'événement Evt_2 modifie le variant *EtatAct* de telle sorte qu'il rende la *main* à l'événement Evt_0 de l'abstraction. Cet événement donne le raffinement de l'événement correspondant à la tâche T_0.

La clause **ASSERTIONS** qui décrit la disjonction des gardes permet d'assurer que les événements peuvent être déclenchés. Pour montrer qu'il n'y a pas de blocage nous ajoutons que la disjonction des gardes abstraites implique ($\Rightarrow$) la disjonction des gardes concrètes.

Par ailleurs, l'invariant de collage $J(var_i, var_j)$ permet d'assurer la correspondance entre les deux niveaux de modélisation (correspondance entre les variables var_i du modèle abstrait et les variables var_j du modèle concret). Cet invariant de collage permet de garantir que la postcondition de T_0 décrite par S_0 est établie lorsque T_1 et T_2 sont déclenchées en séquence. Les tâches T_1 et T_2 « travaillent » ou se déclenchent pour que T_0 de l'abstraction se réalise correctement. L'invariant $EtatAct \in \{0, 1, 2\}$ permet de typer le variant *EtatAct*. Finalement, l'invariant $J'(var_j)$ permet d'exprimer des propriétés supplémentaires sur les variables de la conception.

Choix. Considérons la tâche $T_0 ::= T_1[]T_2$ qui définit un choix non déterministe entre la tâche T_1 et T_2. La traduction en B événementiel est donnée par le raffinement $RefActivationTache_{T0}$ qui modélise trois événements Evt_1, Evt_2 associés aux tâches T_1 et T_2, tandis que l'événement $Evt_{InitChoix}$ donne une valeur au variant au moyen de la substitution **ANY WHERE THEN**. La valeur du variant détermine si Evt_1 ou Evt_2 est déclenché.

RAFFINEMENT $RefChoixTache_{T0}$
REFINE $Tache_{T0}$
INVARIANT
$J(var_i, var_j) \wedge EtatChoix \in \{0, 1, 2, 3\} \wedge J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((\exists(p).(p \in \{1, 2\} \wedge EtatChoix = 3)) \vee (G_1 \wedge EtatChoix = 1) \vee$
$(G_2 \wedge EtatChoix = 2) \vee (EtatChoix = 0 \wedge G_0))$
INITIALISATION
$EtatChoix := 3$

EVENTS			
$Evt_{InitChoix} =$	$Evt_1 =$	$Evt_2 =$	$Evt_0 =$
ANY p	**SELECT**	**SELECT**	**SELECT**
WHERE	$EtatChoix = 1 \wedge G_1$	$EtatChoix = 2 \wedge G_2$	$EtatChoix = 0$
$p \in \{1, 2\} \wedge$	**THEN**	**THEN**	$\wedge\ G'_0$
$EtatChoix = 3$	$EtatChoix := 0 \ \|\|$	$EtatChoix := 0 \parallel S_2$	**THEN**
THEN	S_1	**END ;**	S'_0
$EtatChoix := p$	**END ;**		**END ;**
END ;			

Notons aussi la présence de l'événement Evt_0 qui raffine la tâche T_0. Par ailleurs, l'invariant de collage $J(var_i, var_j)$ décrit la correspondance entre les variables du modèle concret et celles du modèle abstrait. Il permet de garantir que la postcondition de T_0 décrite par S_0 est établie lorsque T_1 ou T_2 sont déclenchée. Les tâches T_1 ou T_2 se déclenche pour que T_0 de l'abstraction se réalise correctement.

Tâche itérative. Considérons la tâche itérative finie $T_0 ::= T_1^N$. Si N est un entier naturel fixé, cette forme est utilisée pour décrire une tâche itérative finie (par exemple T_1^5 qui indique que T_1 est réalisée cinq fois).

Dans le cas où il s'agit d'une itération finie où N est un entier fixé, la traduction s'effectue uniquement par un événement $EvtLoop_1$ qui correspond à la tâche à exécuter plusieurs fois.

La traduction en B événementiel est donnée par le raffinement $RefIterationFinie-Tache_{T0}$ ci-dessous :

RAFFINEMENT $RefIterationFinieTache_{T0}$
REFINE $Tache_{T0}$
INVARIANT
$J(var_i, var_j) \wedge EtatLoop \in NAT \wedge J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((EtatLoop > 0 \wedge G_1) \vee (EtatLoop = 0 \wedge G'_0))$
INITIALISATION
$EtatLoop := N$

EVENTS

$Evt_{Loop1} =$	$Evt_0 =$
SELECT	**SELECT**
$EtatLoop > 0 \wedge G_1$	$EtatLoop = 0 \wedge G'_0$
THEN	**THEN**
$EtatLoop := EtatLoop - 1 \parallel S_{Loop}$	S'_0
END ;	**END ;**

Le variant est initialisé dans la clause **INITIALISATION**. L'événement Evt_{Loop1} est déclenché en premier pendant N fois puis redonne la main à l'abstraction (Evt_0 déclenché). Nous retrouvons l'invariant de collage $J(var_i, var_j)$, l'invariant pour le typage du variant $EtatLoop$ et l'invariant $J'(var_j)$ qui exprime des propriétés supplémentaires sur la conception.

Dans le cas où il s'agit d'une itération avec un nombre arbitraire N, la spécification donnée ci-dessus change et un nouvel événement $Evt_{InitLoop}$ est introduit. Le raffinement $RefIterationTache_{T0}$ ci-dessous traduit cette modification :

RAFFINEMENT $RefIterationTache_{T0}$
REFINE $Tache_{T0}$
INVARIANT
$J(var_i, var_j) \wedge EtatLoop \in NAT \wedge Depart \in \{0, 1\} \wedge J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((\exists(N).(N \in NAT \wedge Depart = 0)) \vee$
$(EtatLoop > 0 \wedge G_1 \wedge Depart = 1) \vee (EtatLoop = 0 \wedge G'_0 \wedge Depart = 1)$
INITIALISATION
$EtatLoop :: NAT \parallel Depart := 0$

EVENTS

$Evt_{InitLoop} =$	$Evt_{Loop1} =$	$Evt_0 =$
ANY N	**SELECT**	**SELECT**
WHERE	$EtatLoop > 0 \wedge G_1 \wedge Depart = 1$	$EtatLoop = 0 \wedge$
$N \in NAT \wedge Depart = 0$	**THEN**	$G'_0 \wedge Depart = 1$
THEN	$EtatLoop := EtatLoop - 1 \parallel$	**THEN**
$EtatLoop := N \parallel$	S_{Loop}	S'_0
$Depart := 1$	**END ;**	**END ;**
END ;		

Il existe toujours un invariant pour le collage $J(var_i, var_j)$ des variables du niveau

abstrait et celles du niveau concret, un invariant pour les variants et un invariant pour les propriétés.

Trois événements sont utilisés pour modéliser la boucle avec un nombre arbitraire. Le premier événement $Evt_{InitLoop}$ déclenché si $Depart = 0$ initialise le variant de boucle *EtatLoop*. Le deuxième événement Evt_{Loop1} modélise le corps de la boucle et décrémente le variant *EtatLoop*. Finalement le troisième événement correspond à la tâche de fin et termine la boucle et redonne la main à l'abstraction.

Concurrence. Considérons la tâche $T_0 ::= T_1 \parallel T_2$. La sémantique de la concurrence dans une sémantique par entrelacement oblige à décrire tous les états possibles un peu à la manière de l'approche explicite (section 2.4.3). En ce qui concerne la tâche T_0, il faut alors décrire l'ensemble de tous les états observables.

La sémantique d'entrelacement à base de traces du B événementiel est utilisée pour traduire cette opération. Nous obtenons la traduction dans le raffinement ($RefParallele-Tache_{T0}$) suivant :

RAFFINEMENT $RefParalleleTache_{T0}$
REFINE $Tache_{T0}$
INVARIANT
$J(var_i, var_j) \wedge EtatConc_1 \in \{0,1\} \wedge EtatConc_2 \in \{0,1\} \wedge J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((G_1 \wedge EtatConc_1 = 1) \vee (G_2 \wedge EtatConc_2 = 1) \vee$
$(EtatConc_1 = 0 \wedge EtatConc_2 = 0 \wedge G'_0))$
INITIALISATION
$EtatConc_1 := 1 \parallel EtatConc_2 := 1$

EVENTS

$Evt_1 =$	$Evt_2 =$	$Evt_0 =$
SELECT	**SELECT**	**SELECT**
$G_1 \wedge EtatConc_1 = 1$	$G_2 \wedge EtatConc_2 = 1$	$EtatConc_1 = 0 \wedge$
THEN	**THEN**	$EtatConc_2 = 0 \wedge G'_0$
$EtatConc_1 = 0 \parallel$	$EtatConc_2 = 0 \parallel$	**THEN**
S_1	S_2	S'_0
END ;	**END ;**	**END ;**

L'invariant de collage $J(var_i, var_j)$ permet de garantir que la postcondition de T_0 décrite par S_0 est établie lorsque T_1 et T_2 sont déclenchées en parallèle. Les tâches T_1 et

T_2 se déclenchent pour que T_0 de l'abstraction se réalise correctement. Nous retrouvons par ailleurs, les invariants liés au typage des variant $EtatConc_1$ et $EtatConc_2$ puis l'invariant $J'(var_j)$ qui exprime des propriétés sur les variables de la conception.

Les événements Evt_1 et Evt_2 peuvent être déclenchés dans n'importe quel ordre et à n'importe quel moment par l'intermédiaire de la clause **INITIALISATION** et des variants $EtatConc_1$ et $EtatConc_2$. Une fois que ces événements ont été déclenchés, ils ne sont plus déclenchables puisque leur garde ne l'autorise plus. L'originalité de cette traduction vient de l'événement décrivant la tâche mère T_0 qui n'est déclenchable que si les deux événements Evt_1 et Evt_2 ont terminé leur traitement. L'abstraction ne reprend la main que si le traitement en parallèle (par entrelacement) est terminé.

Représentation en B événementiel des autres constructions de CTT. Les opérateurs basiques tels que (>>, $\|$, [], $*$, ^{N}) ont été complètement représentés en B événementiel. Nous complétons la grammaire CTT en se basant sur ces résultats.

Par ailleurs, afin de faciliter la lecture du code B, nous simplifierons les écritures des traductions et notamment celles de la clause **ASSERTIONS**. Nous nous servirons également de l'écriture de la modélisation de la tâche T_0 que nous rappelons ci-dessous.

MODEL $Tache_{T0}$
INVARIANT
$I(var_i)$

EVENTS
$Evt_0 =$
SELECT
G_0
THEN
S_0
END ;

Ordre Indépendant. Soit $T_0 ::= T_1 \models| T_2$, sa représentation en B événementiel est donnée par le raffinement $RefIndependantTache_{T0}$ ci-dessous :

RAFFINEMENT $RefIndependantTache_{T0}$
REFINE $Tache_{T0}$
INVARIANT
$J(var_i, var_j) \wedge (EtatCh \in \{0,1,2,3\}) \wedge (EtatAct \in \{0,1,2\}) \wedge J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((EtatCh = 1 \wedge EtatAct = 2 \wedge G_1) \vee$
$(EtatCh = 1 \wedge EtatAct = 1 \wedge G_2) \vee (EtatCh = 2 \wedge EtatAct = 2 \wedge G_2) \vee \ldots)$
INITIALISATION
$EtatCh := 3 \parallel EtatAct := 2$
EVENTS

$Evt_{11} =$	$Evt_{12} =$	$Evt_{21} =$	$Evt_{22} =$
SELECT	**SELECT**	**SELECT**	**SELECT**
$EtatCh = 1 \wedge$	$EtatCh = 1 \wedge$	$EtatCh = 2 \wedge$	$EtatCh = 2 \wedge$
$EtatAct = 2 \wedge G_1$	$EtatAct = 1 \wedge G_2$	$EtatAct = 2 \wedge G_2$	$EtatAct = 1 \wedge G_1$
THEN	**THEN**	**THEN**	**THEN**
$EtatAct := 1 \parallel$	$EtatAct := 0 \parallel$	$EtatAct := 1 \parallel$	$EtatAct := 0 \parallel$
S_1	$EtatCh := 0 \parallel$	S_2	$EtatCh := 0 \parallel$
END ;	S_2	**END ;**	S_1
	END ;		**END ;**

L'invariant de collage permet de garantir que la postcondition de T_0 décrite pas S_0 est établie lorsque T_1 et T_2 sont déclenchées de façon indépendante. L'invariant $(EtatCh \in \{0,1,2,3\}) \wedge (EtatAct \in \{0,1,2\})$ permet d'exprimer le typage des variants $EtatCh$ et $EtatAct$. Dans la clause **ASSERTIONS**, nous exprimons le non blocage en ajoutant la propriétés « la disjonction des gardes abstraites implique la disjonction des gardes concrètes ».

$Evt_{Init} =$	$Evt_0 =$
ANY p	**SELECT**
WHERE	$EtatAct = 0 \ \wedge \ EtatCh = 0 \wedge G'_0$
$p \in \{1,2\} \ \wedge \ EtatCh = 3$	**THEN**
THEN	S'_0
$EtatCh := p$	**END ;**
END ;	

Sont introduits dans cette modélisation six événements et deux variants $EtatCh$ et $EtatAct$ correspondant respectivement aux opérateurs *activation* et *choix*.

La clause **INITIALISATION** permet de déclencher au tout démarrage l'événement Evt_{Init}. Cet événement choisit la tâche à activer. Par exemple, dans le cas où $EtatCh = 1$, Evt_{12} puis Evt_{22} sont déclenchés.

A la fin du traitement de l'ordre indépendant, l'événement Evt_0 est déclenché et redonne la main à l'abstraction.

Tâche Optionnelle. Une tâche optionnelle $T_0 ::= [T_1]$ est traduite par trois événements. Nous employons l'événement Evt_{InitOp} pour déterminer si Evt_1 ou Evt_{skip} est déclenché au moyen du variant *EtatOp*. L'événement Evt_{skip} ne fait rien d'autre que modifier l'état du variant *EtatOp*, c'est-à-dire qu'il n'affecte pas l'état du système puisqu'il n'y a pas d'événement ou d'opération appelés de la conception :

RAFFINEMENT $RefOptionnelleTache_{T0}$
REFINE $Tache_{T0}$
INVARIANT
$J(var_i, var_j) \wedge EtatOp \in \{0,1,2,3\} \wedge J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((\exists(p).(p \in \{1,2\} \wedge EtatOp = 3)) \vee (G_1 \wedge EtatOp = 1) \vee$
$(EtatOp = 2) \vee (EtatOp = 0 \wedge G'_0))$
INITIALISATION
$EtatOp := 3$
EVENTS

$Evt_{InitOp} =$	$Evt_1 =$	$Evt_{skip} =$	$Evt_0 =$
ANY p	**SELECT**	**SELECT**	**SELECT**
WHERE	$EtatOp = 1 \wedge G_1$	$EtatOp = 2$	$EtatOp = 0 \wedge G'_0$
$p \in \{1,2\} \wedge EtatOp = 3$	**THEN**	**THEN**	**THEN**
THEN	$EtatOp := 0 \parallel S_1$	$EtatOp := 0$	S'_0
$EtatOp := p$	**END ;**	**END ;**	**END ;**
END ;			

Désactivation. La représentation de $T_0 ::= T_1[> T_2$ considère deux cas selon que T_1 est atomique ou pas.

1 - T_1 est atomique et vaut T_{At1} : comme T_1 est une tâche atomique, l'écriture de la tâche $T_0 ::= T_1[> T_2$ est traduite en $T_0 ::= T_1[]T_2$. Par conséquent la traduction de la tâche T_0 est identique à celle de l'opérateur *choix* puisqu'il ne peut y avoir que T_1 ou T_2 d'activable. Nous donnons ci-dessous le raffinement $RefDesactTache_{T0}$ correspondant à cette traduction :

RAFFINEMENT $RefDesactTache_{T0}$
REFINE $Tache_{T0}$
INVARIANT
$J(var_i, var_j) \wedge EtatDes \in \{0,1,2,3\} \wedge J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((\exists(p).(p \in \{1,2\} \wedge EtatDes = 3)) \vee (G_1 \wedge EtatDes = 1) \vee (G_2 \wedge EtatDes = 2) \vee (EtatDes = 0 \wedge G'_0))$
INITIALISATION
$EtatDes := 3$

L'invariant de collage $J(var_i, var_j)$ permet de garantir que la postcondition de T_0 décrite par S_0 est établie lorsque T_1 ou T_2 se déclenchent. Les tâches T_1 et T_2 « travaillent » ou se déclenchent pour que T_0 de l'abstraction se réalise correctement.

Le variant $EtatDes$ est utilisé pour déclencher les événements Evt_{InitOp}, Evt_1 et Evt_2. Si le variant est égal à 0, l'événement Evt_0 de l'abstraction est déclenché.

EVENTS

$Evt_{InitOp} =$	$Evt_1 =$	$Evt_2 =$	$Evt_0 =$
ANY p	**SELECT**	**SELECT**	**SELECT**
WHERE	$EtatDes = 1 \wedge G_1$	$EtatDes = 2 \wedge G_2$	$EtatDes = 0 \wedge G'_0$
$p \in \{1,2\} \wedge$	**THEN**	**THEN**	**THEN**
$EtatDes = 3$	$EtatDes := 0 \parallel S_1$	$EtatDes := 0 \parallel S_2$	S'_0
THEN	**END ;**	**END ;**	**END ;**
$EtatDes := p$			
END ;			

2 - T_1 n'est pas atomique et vaut $T_1 := T_{1,1}\ op_1\ \cdots\ op_n\ T_{1,n}$: la solution envisagée utilise la technique du raffinement pour effectuer la décomposition de la tâche T_1. Ainsi, nous décrivons dans l'abstraction l'opération de désactivation comme si T_1 se comportait comme une tâche atomique, c'est-à-dire comme la modélisation précédente. Le raffinement enrichit alors l'abstraction en décrivant le fait que T_1 n'est pas seulement atomique. Il suffit donc de traduire la décomposition suivante $T_1 := T_{1,1}\ op_1\ \cdots\ op_n\ T_{1,n}$ en autorisant la désactivation au niveau du raffinement.

Les avantages de cette technique sont doubles. D'une part, et nous l'avons vu pour l'approche explicite, le raffinement et l'invariant de collage $K(var_i, var_j, var_k)$ permettent de respecter la décomposition, c'est-à-dire que la postcondition de la tâche de plus haut niveau est maintenue dans la décomposition en sous-tâches. D'autre part, le raffinement

permet de traduire implicitement tous les chemins liés à la désactivation d'une tâche non atomique. Cependant, c'est aussi dans le raffinement que nous décrivons la sémantique de l'opérateur de désactivation, c'est-à-dire le fait que la tâche T_2 désactive T_1 pour tous ses états observables.

Plus précisément, l'abstraction est donnée par le modèle $RefDesactTache_{T0}$ précédent et nous donnons dans la spécification $RefDesactTache_{T1}$ ci-dessous son raffinement :

RAFFINEMENT $RefDesactTache_{T1}$
REFINE $RefDesactTache_{T0}$
INVARIANT
$K(var_i, var_j, var_k) \wedge EtatVar \in \{0..n\} \wedge K'(var_k)$
ASSERTIONS
$((\exists(p).(p \in \{1,2\} \wedge EtatDes = 3)) \vee (G_1 \wedge EtatDes = 1) \vee$
$(G_2 \wedge EtatDes = 2) \vee (EtatDes = 0 \wedge G_0)) \Rightarrow (EtatVar = \ldots \vee \ldots)$
INITIALISATION
$EtatVar := \ldots \parallel \ldots$

EVENTS

$Evt_1 =$
SELECT
$EtatVar = 0 \wedge G'_1 \wedge$
$EtatDes = 1$
THEN
S'_1
END ;

$Evt_{11} =$ … $Evt_{1n} =$
SELECT … **SELECT**
$EtatVar = 1$ … $EtatVar = n$
THEN … **THEN**
… …
END ; … **END ;**

$Evt_{Desact1} =$
SELECT
$EtatVar \in (1..n)$
THEN
$EtatVar := 0 \parallel$
EtatDes := 2
END ;

La décomposition en sous-tâches de T_1 est représentée dans le premier rectangle en partant de la gauche par deux événements. Le raffinement $RefDesactTache_{T1}$ enrichit la modélisation en introduisant de nouvelles variables concernant le variant décrivant la décomposition de la tâche T_1 (noté sur la modélisation $EtatVar$). Pendant le déroulement des sous-tâches de T_1, le variant décroît (effets de bord réalisés par les événements $Evt_{11}, Evt_{12}, \ldots, Evt1n$) jusqu'à valoir zéro. L'événement de l'abstraction Evt_1 reprend donc la main.

L'événement $Evt_{Desact1}$ (second rectangle) décrit quant à lui le fait que la désactivation intervient dans les états observables de la tâche T_1 ($EtatVar \in (1..n)$). Son corps modifie les valeurs des variants ($EtatVar$ et $EtatDes$) de telle sorte que la désactivation de T_1 se produise ($EtatVar := 0$) et que l'événement Evt_2 du raffinement $RefDesactTache_{T0}$ se déclenche ($EtatDes := 2$) et désactive ainsi la tâche T_1.

Exemple : pour préciser le raffinement précédent, considérons que T_1 n'est pas atomique et est décomposée en $T_1 ::= T_{11} >> T_{12}$:

RAFFINEMENT $RefDesactTache_{T1}$
REFINE $RefDesactTache_{T0}$
INVARIANT
$K(var_i, var_j, var_k) \wedge EtatAct \in \{0,1,2\} \wedge K'(var_k)$
ASSERTIONS
$((\exists(p).(p \in \{1,2\} \wedge EtatDes = 3)) \vee (G_1 \wedge EtatDes = 1) \vee$
$(G_2 \wedge EtatDes = 2) \vee (EtatDes = 0 \wedge G_0)) \Rightarrow (EtatAct = 2 \wedge G_{11}) \vee \ldots$
INITIALISATION
$EtatAct := 2 \parallel \ldots$

EVENTS

$Evt_1 =$	$Evt_{11} =$	$Evt_{12} =$	$Evt_{Desact1} =$
SELECT	**SELECT**	**SELECT**	**SELECT**
$EtatAct = 0 \wedge G'_1 \wedge$	$EtatAct = 2 \wedge G_{11}$	$EtatAct = 1 \wedge G_{12}$	$EtatAct \in \{1,2\}$
$EtatDes = 1$	**THEN**	**THEN**	**THEN**
THEN	$EtatAct := 1 \parallel$	$EtatAct = 0 \parallel$	$EtatDes := 2 \parallel$
S'_1	S_{11}	S_{12}	$EtatAct := 0$
END ;	**END ;**	**END ;**	**END ;**

Nous avons modélisé la décomposition en sous-tâches de la tâche T_1 par deux événements appelés Evt_{11} et Evt_{12}. Le rôle du variant $EtatAct$ est la description de la séquence entre les tâches T_{11} et T_{12}. A la fin de la séquence d'activation, l'événement Evt_{12} rend la main à l'événement Evt_1 du raffinement $RefDesactTache_{T0}$.

L'événement $Evt_{Desact1}$ désactive à tout instant les états observables de T_1 ($EtatAct \in \{1,2\}$). S'il désactive la tâche T_1, l'événement Evt_2 de l'abstraction se déclenche et désactive la tâche T_1 ($EtatAct := 0$).

Désactivation d'une tâche itérative. $T_0 ::= T_1^*[> T_2$ est représentée en B événementiel selon que T_1 est atomique ou non. Nous proposons ci-dessous deux niveaux de modélisation à la manière de l'opérateur de désactivation étudié précédemment.

1 - T_1 est atomique et vaut T_{At1} : comme T_1 est atomique la transformation en opérateurs basiques de $T_0 ::= T_1^*[> T_2$ est $T_0 ::= (T_1)^N >> T_2$ où N est un entier naturel.

RAFFINEMENT $RefDesactIteraTache_{T0}$
REFINE $Tache_{T0}$
INVARIANT
$J(var_i, var_j) \wedge EtatDes \in \{0,1,2\} \wedge EtatLoop \in NAT \wedge Depart \in \{0,1\}$
$\wedge\ J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((\exists(N).(N \in NAT \wedge Depart = 0 \wedge EtatDes = 2)) \vee \ldots)$
INITIALISATION
$EtatLoop :: NAT \parallel EtatDes := 2 \parallel Depart := 0$

EVENTS

$Evt_{InitLoop} =$
ANY N
WHERE
$EtatDes = 2 \wedge N \in NAT \wedge$
$Depart = 0$
THEN
$EtatLoop := N \parallel Depart := 1$
END ;

$Evt_{Loop1} =$
SELECT
$EtatLoop > 0 \wedge G_1 \wedge$
$EtatDes = 2 \wedge Depart = 1$
THEN
$EtatLoop := EtatLoop - 1$
$\parallel S_1$
END ;

$Evt_1 =$
SELECT
$EtatLoop = 0 \wedge$
$EtatDes = 2 \wedge$
$Depart = 1$
THEN
$EtatDes := 1$
END ;

$Evt_2 =$
SELECT
$EtatDes = 1 \wedge G_2$
THEN
$EtatDes := 0 \parallel S_2$
END ;

$Evt_0 =$
SELECT
$EtatDes = 0 \wedge G'_0$
THEN
S'_0
END ;

Cinq événements modélisent cette construction. Le variant $EtatLoop$ traduit la boucle de la tâche T_1 et $EtatDes$ décrit la séquence entre la tâche T_1 et T_2. Quand le variant $EtatDes$ vaut 0, l'événement Evt_0 de l'abstraction reprend la main.

Nous retrouvons dans la clause **INVARIANT** l'invariant de collage qui permet de garantir que la postcondition de T_0 décrite par S_0 est établie lorsque T_1 et T_2 sont déclenchées en séquence. Nous trouvons par ailleurs, les invariants de typage et l'invariant $J'(var_j)$ qui exprime des propriétés complémentaires.

2 - T_1 n'est pas atomique et vaut $T_1 := T_{1,1}\ op_1\ \cdots\ op_n\ T_{1,n}$: à l'abstraction précédente ($RefDesactIteraTache_{T0}$), nous l'enrichissons par la décomposition de la tâche $T_1 := T_{1,1}\ op_1\ \cdots\ op_n\ T_{1,n}$. Cette modélisation exploite la même approche que pour la traduction de l'aspect désactivation des états observables de la tâches T_1 :

RAFFINEMENT $RefDesactIteraTache_{T1}$
REFINE $RefDesactIteraTache_{T0}$
INVARIANT
$K(var_i, var_j, var_k) \wedge EtatVar \in (0..n) \wedge J'(var_j)$
ASSERTIONS
$((\exists(N).(N \in NAT \wedge Depart = 0 \wedge EtatDes = 2)) \vee \ldots)$
$\Rightarrow (EtatVar = \ldots \vee \ldots)$
INITIALISATION
$EtatVar := \ldots \parallel \ldots$

EVENTS

$Evt_1 =$	$Evt_{11} =$	$Evt_{1n} =$	$Evt_{Desact1} =$
SELECT	**SELECT**	**SELECT**	**SELECT**
$EtatVar = 0 \wedge$	$EtatVar = 1$	$EtatVar = n$	$EtatVar \in (1..n)$
$EtatDes = 2 \wedge \ldots$	**THEN**	**THEN**	**THEN**
THEN	$\ldots$	$\ldots$	$EtatVar := 0 \parallel$
S'_1	**END ;**	**END ;**	**EtatDes := 1**
END ;			**END ;**

La décomposition en sous-tâches de la tâche T_1 est également représentée sur le premier rectangle où $EtatVar$ est le variant de déclenchements des événements $Evt_{11} \ldots Evt_{1n}$.

L'événement $Evt_{Desact1}$ (second rectangle) se charge de désactiver à tout moment le déroulement des sous-tâches de T_1 ($EtatVar \in (1..n)$). Son traitement modifie le variant de la décomposition ($EtatDes := 1$) et le variant du déroulement des sous-tâches ($EtatVar := 0$) de T_1. L'événement $Evt_{Desact1}$ permet donc de redonner la main à l'événement Evt_1 de l'abstraction et de continuer le déroulement avec la tâche T_2 en déclenchant l'événement Evt_2.

Exemple : pour préciser le raffinement précédent, considérons que T_1 n'est pas atomique et est décomposée en $T_1 ::= T_{11} >> T_{12}$. Nous proposons ci-dessous le raffinement $RefDesactIteraTache_{T1}$ dans le cas où la tâche T_1 se décompose en $T_1 ::= T_{11} >> T_{12}$ (similaire à la construction désactivation) :

RAFFINEMENT $RefDesactIteraTache_{T1}$
REFINE $RefDesactIteraTache_{T0}$
INVARIANT
$K(var_i, var_j, var_k) \wedge EtatAct \in \{0,1,2\} \wedge K'(var_k)$
ASSERTIONS
$((\exists(N).(N \in NAT \wedge Depart = 0 \wedge EtatDes = 2)) \vee \ldots)$
$\Rightarrow ((EtatAct = 0 \wedge EtatDes = 2 \wedge \ldots) \vee (EtatAct = 2 \wedge G_{11}) \vee \ldots)$
INITIALISATION
$EtatAct := 2 \parallel \ldots$

EVENTS

$Evt_1 =$	$Evt_{11} =$	$Evt_{12} =$	$Evt_{Desact1} =$
SELECT	**SELECT**	**SELECT**	**SELECT**
$EtatAct = 0 \wedge$	$EtatAct = 2 \wedge G_{11}$	$EtatAct = 1 \wedge G_{12}$	$EtatAct \in (1..2)$
$EtatDes = 2 \wedge \ldots$	**THEN**	**THEN**	**THEN**
THEN	$EtatAct := 1 \parallel$	$EtatAct = 0 \parallel$	$EtatDes := 1 \parallel$
S'_1	S_{11}	S_{12}	$EtatAct := 0$
END ;	**END ;**	**END ;**	**END ;**

A l'initialisation l'événement Evt_{11} est déclenché. En fonction des effets de bords de cet événement ($EtatAct := 1$), l'événement Evt_{12} est ensuite déclenché. Ce dernier événement rend ensuite la main à l'abstraction, c'est-à-dire à l'événement Evt_1.

Cependant, l'événement $Evt_{Desact1}$ peut se déclencher dans tous les états observables de la tâche T_1 ($EtatAct \in \{1..2\}$). L'événement de désactivation rend la main à l'événement de l'abstraction Evt_2.

Interruption. la modélisation de la traduction de $T_0 ::= T_1| > T_2$ est différente selon que T_1 est atomique ou pas.

1 - T_1 est atomique et vaut T_{At1} : comme T_1 est atomique, l'écriture en opérateurs basiques de la tâche $T_0 ::= T_1| > T_2$ est donnée par $T_0 ::= (T_2)^N >> T_1$. Cette forme ressemble à celle de la désactivation. Nous donnons ci-dessous la modélisation en B événementiel de cette représentation :

RAFFINEMENT $RefInterupTache_{T0}$
REFINE $Tache_{T0}$
INVARIANT
$J(var_i, var_j) \wedge EtatInt \in \{0,1,2\} \wedge EtatLoop \in NAT \wedge Depart \in \{0,1\} \wedge J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((\exists(N).(N \in NAT \wedge Depart = 0 \wedge EtatInt = 2)) \vee \ldots)$
INITIALISATION
$EtatLoop :: NAT \parallel Depart := 0 \parallel EtatInt := 2$

EVENTS

$Evt_{InitLoop2} =$
ANY N
WHERE
$EtatInt = 2 \wedge N \in NAT \wedge$
$Depart = 0$
THEN
$EtatLoop := N \parallel Depart := 1$
END ;

$Evt_{Loop2} =$
SELECT
$EtatLoop > 0 \wedge G_2 \wedge$
$EtatInt = 2 \wedge Depart = 1$
THEN
$EtatLoop := EtatLoop - 1$
$\parallel S_2$
END ;

$Evt_2 =$
SELECT
$EtatLoop = 0 \wedge$
$EtatInt = 2$
$Depart = 1$
THEN
$EtatInt := 1$
END ;

$Evt_1 =$
SELECT
$EtatInt = 1 \wedge G_1$
THEN
$EtatInt := 0 \parallel S_1$
END ;

$Evt_0 =$
SELECT
$EtatInt = 0 \wedge G'_0$
THEN
S'_0
END ;

Cinq événements modélisent cette construction. Le variant *EtatLoop* traduit la boucle de la tâche T_2 alors que *EtatInt* traduit l'opérateur de séquence entre la tâche T_2 et T_1.

Nous retrouvons dans la clause **INVARIANT**, l'invariant de collage $J(var_i, var_j)$ et les deux invariants qui typent les variants *EtatInt* et *EtatLoop*. Dans la clause **ASSERTIONS** nous exprimons la propriété de non blocage par la disjonction des gardes abstraites impliquant la disjonction des gardes concrètes.

2 - T_1 n'est pas atomique et vaut $T_1 := T_{1,1}\ op_1\ \cdots\ op_n\ T_{1,n}$: nous procédons de la même manière que pour les constructions *désactivation* et *désactivation d'une tâche itérative*. Nous donnons sur le raffinement $RefInterupTache_{T1}$ ci-dessous la modélisation de la décomposition de la tâche T_1 :

RAFFINEMENT $RefInterupTache_{T1}$
REFINE $RefInterupTache_{T0}$
INVARIANT
$K(var_i, var_j, var_k) \wedge EtatVar \in (0..n) \wedge Interrupt \in \{0,1\} \wedge K'(var_k)$
ASSERTIONS
$((\exists(N).(N \in NAT \wedge Depart = 0 \wedge EtatInt = 2)) \vee \ldots)$
$\Rightarrow ((EtatVar = \ldots) \vee \ldots)$
INITIALISATION
$Interrup := 0 \parallel EtatVar := \ldots \parallel \ldots$

EVENTS

$Evt_1 =$
SELECT
$EtatVar = 0 \wedge$
$EtatInt = 1 \wedge$
G'_1
THEN
S'_1
END ;

$Evt_{11} =$
SELECT
$EtatVar = 1 \wedge$
$Interrup = 0 \wedge \ldots$
THEN
$\ldots$
END ;

$Evt_{1n} =$
SELECT
$EtatVar = n \wedge$
$Interrup = 0 \wedge \ldots$
THEN
$\ldots$
END ;

$Evt_{Interupt1} =$
SELECT
$EtatVar \in (1..n) \wedge$
$Interrup = 0$
THEN
$EtatLoop := -1 \parallel$
$EtatInt := 2 \parallel$
Interrup := 1
END ;

$Evt_2 =$
SELECT
$EtatLoop = 0 \wedge \ EtatInt = 2 \wedge$
$Interrup = 1$
THEN
$EtatInt := 1 \parallel \ Interrup := 0$
END ;

Nous identifions trois zones distinctes. La première (rectangle en haut à gauche) modélise la décomposition en sous-tâches de la tâche $T_1 := T_{1,1}\ op_1\ \cdots\ op_n\ T_{1,n}$ représentée ici par deux événements Evt_{11} et Evt_{1n}. Le variant $EtatVar$ est utilisé pour contrôler le déclenchement de ces deux événements. Par ailleurs le variant $Interrup$ indique si les sous-tâches de T_1 sont dans un état interrompu ($Interrup = 1$) ou pas ($Interrup = 0$).

Une deuxième zone (rectangle en haut à droite) se charge de modéliser l'interruption au moyen de l'événement $Evt_{Interupt1}$. Nous décrivons par sa garde deux aspects. Le premier concerne le fait que cet événement peut interrompre le déroulement des sous-tâches de T_1 à tous les niveaux des états observables de cette tâche ($EtatVar \in \{1..n\}$). Le second ($Interrup = 0$) modélise le fait que tant que l'interruption n'est pas terminée (interruption

en cours $Interrup$ vaut 1), l'événement $Evt_{Interupt1}$ ne peut interrompre de nouveau. Si cet événement se déclenche, nous indiquons à la modélisation que la tâche T_1 est dans un état interrompu $Interrup = 1$. Les variants ($EtatLoop$) et ($EtatInt$) sont initialisés de telle façon que le raffinement donne la main à l'abstraction en déclenchant l'événement $Evt_{InitLoop2}$.

L'événement $Evt_{InitLoop2}$ qui vient d'être déclenché se charge du traitement lié à la tâche d'interruption (voir déroulement du raffinement $RefInterupTache_{T0}$). La dernière zone désigne l'événement Evt_2 qui a été enrichi de la variable $Interrupt$. Il se charge d'avertir la modélisation que le traitement lié à l'interruption est terminé ($Interrup := 0$). Ce changement de valeur du variant $Interrup$ permet aux événements Evt_{11} et Evt_{1n} qui décrivent les sous-tâches de T_1 de poursuivre leurs traitements (rectangle en haut à gauche).

Exemple : pour préciser le raffinement précédent, considérons que T_1 n'est pas atomique et est décomposée en $T_1 ::= T_{11} >> T_{12}$. Nous procédons de la même manière que pour les opérations de *désactivation*. A l'abstraction précédente, nous l'enrichissons par la décomposition de la tâche $T_1 ::= T_{11} >> T_{12}$. Seulement, l'événement $Evt_{Desact1}$ est remplacé par $Evt_{Interupt1}$ pour traduire la notation de suspension et de reprise :

RAFFINEMENT $RefInterupTache_{T1}$
REFINE $RefInterupTache_{T0}$
INVARIANT
$K(var_i, var_j, var_k) \wedge EtatAct \in \{0,1,2\} \wedge Interrupt \in \{0,1\} \wedge K'(var_k)$
ASSERTIONS
$((\exists(N).(N \in NAT \wedge Depart = 0 \wedge EtatInt = 2)) \vee \ldots)$
$\Rightarrow ((EtatAct = 2 \wedge G_{11} \wedge Interrup = 0) \vee (EtatAct = 0 \wedge \ldots) \vee \ldots)$
INITIALISATION
$EtatAct := 2 \parallel Interrup := 0 \ldots$

L'invariant de collage $K(var_i, var_j, var_k)$ permet de garantir que la postcondition de T_1 décrite par S_0 est établie lorsque T_{11} et T_{12} sont déclenchées en séquence. Les tâches T_{11} et T_{12} se déclenchent pour que T_1 de l'abstraction se réalise correctement. Nous trouvons par ailleurs, l'invariant $K'(var_k)$ qui exprime des propriétés complémentaires à celles introduites dans la phase de conception.

EVENTS

$Evt_1 =$
SELECT
$EtatAct = 0 \wedge$
$EtatInt = 1 \wedge$
G'_1
THEN
S'_1
END ;

$Evt_{11} =$
SELECT
$EtatAct = 2 \wedge G_{11}$
$\wedge\ Interrup = 0$
THEN
$EtatAct := 1 \parallel$
S_{11}
END ;

$Evt_{12} =$
SELECT
$EtatAct = 1 \wedge G_{12}$
$\wedge\ Interrup = 0$
THEN
$EtatAct := 0 \parallel$
S_{12}
END ;

$Evt_{Interupt1} =$
SELECT
$EtatAct \in \{1, 2\} \wedge$
$Interrup = 0$
THEN
$EtatLoop := -1 \parallel$
$EtatInt := 2 \parallel$
$Interrup := 1$
$Interrup := 1$
END ;

$Evt_2 =$
SELECT
$EtatLoop = 0 \wedge\ EtatInt = 2 \wedge$
$Interrup := 0$
THEN
$EtatInt := 1 \parallel\ Interrup := 0$
END ;

Les événements Evt_{11} et Evt_{12} se déclenchent à la condition qu'ils ne soient pas interrompus ($Interrup = 0$) et que la valeur du variant qui traduit l'opération d'activation soit bonne ($EtatAct = 2$ ou $EtatAct = 1$). L'événement $Evt_{Interrupt1}$ est déclenchable dans les états observables de la séquence de tâches T_{11} et T_{12} ($EtatAct \in \{1..2\}$) et si la tâche T_1 n'est pas déjà interrompue. A ce moment là, le raffinement donne la main à l'abstraction $RefInterupTache_{T0}$ qui effectue son traitement. Quand le traitement est terminé $EtatLoop = 0$, l'événement Evt_2 est déclenché et modifie l'état d'interruption de telle façon que les événements Evt_{11} et Evt_{12} puissent de nouveau se déclencher. Quand Evt_{12} est déclenché et termine son traitement, il donne la main à l'abstraction par l'intermédiaire de l'événement Evt_1.

2.5.2.2 Formalisation des informations des tâches CTT

Nous avons présenté dans la section précédente une représentation des constructions CTT en B événementiel. Intéressons nous maintenant à la représentation en B des informations des tâches pour que notre représentation soit complète. A partir des informations de la sémantique originelle, nous avons extrait plusieurs aspects à formaliser :

- les objets à manipuler ;
- la précondition des tâches ;
- l'action ou corps des tâches ;
- finalement les catégories de tâches.

Il est à noter que la formalisation des informations des tâches a été représentée dans tous les raffinements donnés ci-dessus.

Objets à manipuler : les objets à manipuler par les tâches sont issus de la modélisation de la conception. Plus précisément, il s'agit des variables qui décrivent l'IHM, des opérations qui modifient l'état de ces variables et enfin les événements du contrôleur de dialogue qui spécifient la dynamique du dialogue.

La modélisation B du modèle de tâches doit donc étendre les composants de la conception dont le point d'entrée reste le module du contrôleur de dialogue.

Précondition des tâches : la précondition des tâches est relative aux aspects des contraintes de précédence (liées aux opérateurs temporels) et aux contraintes relatives aux objets de la conception. Le premier aspect a déjà été abordé, il s'agit de la formalisation des opérateurs par les variants. Il nous reste donc à traduire ce second aspect.

En général, nous avons vu qu'un événement était écrit sous la forme d'une substitution gardée dont le prédicat exprimait des relations sur les variants. A ce prédicat, nous y ajoutons des expressions relatives aux objets de la conception afin de traduire ce second aspect.

Cependant, tous les événements résultant de la représentation des constructions CTT ne sont pas directement associés à une tâche. C'est par exemple le cas de l'itération où il y a besoin de trois événements. Il ne faut donc exprimer des contraintes sur les objets de la conception uniquement que sur les événements qui sont directement associés à une tâche.

Action ou corps des tâches : lorsque la précondition d'une tâche est vraie, l'action de celle-ci est réalisable et modifie l'état des objets de la conception.

La substitution de l'événement ne modifie pas seulement l'état des variants mais aussi l'état des objets de la conception. De manière plus précise nous employons des substitutions généralisées de type *appel d'opération*. Il s'agit en fait des événements du contrôleur de dialogue s'il s'agit de tâches feuilles.

L'appel à des éléments de la conception permet d'une part de s'assurer de la cohérence du modèle de tâches (la postcondition d'une tâche mère est conforme à la postcondition de ses tâches filles) et d'autre part de la validation de tâches.

Catégories de tâches : rappelons qu'il existe quatre catégories de tâches (interaction, application, utilisateur et abstraite) qui sont dépendantes de l'action à effectuer.

En considérant les tâches feuilles d'un arbre de tâches et les événements associés, nous traduisons les catégories par la nature de l'événement du contrôleur de dialogue. S'il s'agit d'un événement dont les gardes font référence à des variables d'états associées à des périphériques d'entrée (curseur de la souris positionné sur un bouton, appui d'une touche du clavier, etc), l'événement est de nature « interactif ». Par rapport à l'étude de cas, c'est par exemple la tâche interaction *clicker sur le bouton E » F.*

Les tâches de plus haut niveau sont catégoriées selon les catégories des sous tâches. Par exemple, si les sous-tâches sont de catégories différentes, la tâche mère est donc abstraite. Enfin si une tâche feuille est de catégorie utilisateur alors aucune action n'est réalisée c'est-à-dire qu'aucun élément de la conception n'est référencé.

2.5.2.3 Application à l'étude de cas du convertisseur et du compteur

Les règles de transformation précédentes (section 2.5.2.1) couvrent l'ensemble des constructions (opérateurs et informations de tâches) du langage CTT pour la modélisation des tâches utilisateurs. Ces règles donnent une sémantique formelle qui utilise la sémantique à base de traces du B événementiel. Toutes ces règles sont implémentées et sont supportées par l'outil Atelier B.

Quand une analyse de tâches est définie par un modèle de tâches CTT, la décomposition en B événementiel s'appuie sur les règles de transformation pour construire cet arbre de tâches. L'arbre de décomposition correspond alors à un modèle et à des raffinements B. Le raffinement est effectué jusqu'à ce que les sous tâches (tâches terminales) correspondent à des événements du contrôleur de dialogue. Le niveau de raffinement est donc fortement dépendant de la profondeur de l'arbre de tâches.

Les spécifications B contiennent les clauses **ASSERTIONS** et **INVARIANT**. En plus de valider le modèle de tâches (contraintes de précédence), ces clauses permettent d'exprimer aussi des propriétés supplémentaires liées au fonctionnement attendu d'une tâche. En d'autres termes, il est possible de valider ou d'invalider des propriétés à partir de descriptions de tâches CTT.

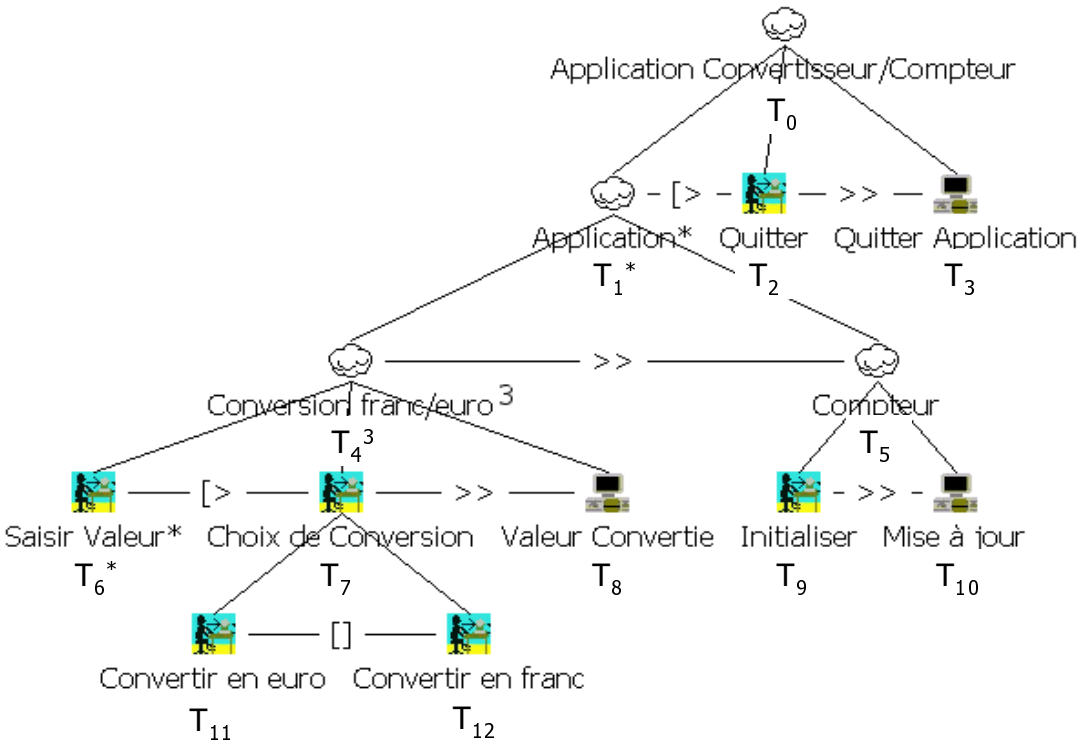

Fig. 2.16 – Modèle de tâches CTT de l'application convertisseur francs/euros et compteur.

Appliquons ces règles de transformation à notre étude de cas du convertisseur francs-/euros et compteur [7].

Sur la figure 2.16 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.

[7] Les sources B de l'étude de cas du convertisseur francs/euros et compteur sont disponibles à l'adresse *http ://www.lisi.ensma.fr/members/baron/*

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 $T_{11}[]T_{12}$.

Pour des raisons de simplification, nous avons utilisé la substitution *devient élément de* ($:\in$) à la place de la substitution *choix non borné* (**ANY** ...) puisque en se reportant à [Abr96a] nous obtenons $X :\in E \triangleq$ **ANY** Y **WHERE** $Y \in E$ **THEN** $X := Y$ **END**. Par ailleurs, nous ne détaillerons pas complètement le code, nous ferons apparaître uniquement les événements issus du contrôleur de dialogue.

Finalement, nous présentons sur le tableau 2.14 les événements que nous allons employer dans la formalisation du modèle de tâches CTT de l'étude de cas.

Nomination	Description
evtClickQuitButton	Cliquer sur le bouton quitter
evtClickQuitApplication	Fermeture de l'application
evtInputValue	Saisir un chiffre
evtDisplayValue	Met à jour les vues après une conversion
evtClickInitialiser	Cliquer sur le bouton initialiser du compteur
evtInitialiser	Met à jour les vues après l'initialisation du compteur
evtClickEuro	Cliquer sur le bouton $F >> E$
evtClickFranc	Cliquer sur le bouton $E >> F$

Tab. 2.14 – Extrait des événements contenus dans le modèle *BControleurConvertisseurCompteur*.

Premier modèle B : il s'agit de la modélisation de la tâche de plus haut niveau T_0. La liaison entre la modélisation de tâches et la phase de conception est ob-

tenue par l'extension du modèle *BControleurConvertisseurCompteur*. Pour des raisons de simplification de code, l'extension du modèle *BControleurConvertisseurCompteur* (**EXTENDS** *BControleurConvertisseurCompteur*) se retrouve dans toutes les modélisations.

Il n'y a qu'un seul événement Evt_0 avec une garde G_0 et une substitution *appel d'opération* (S_0) correspond à la postcondition globale. Le rôle de cette substitution S_0 est de préserver l'état final de la tâche T_0.

MODEL *Convertisseur/Compteur*
EXTENDS *BControleurConvertisseurCompteur*
INVARIANT
$I(var_i)$

EVENTS
$Evt_0 =$
SELECT
G_0
THEN
S_0
END ;

Notons que nous aurions pu immédiatement modéliser $T_0 = ((T_6^*[> (T_{11}[]T_{12}) >> T_8)^3 >> (T_9 >> T_10))^*[> (T_2 >> T_3)$. Cependant, sans l'utilisation de la technique de raffinement, la modélisation serait beaucoup plus complexe et rendraient les obligations de preuve difficiles à décharger. Par ailleurs, l'invariant permet d'exprimer des propriétés sur les variables d'état var_i du modèle.

Premier raffinement : le premier niveau de décomposition concerne la tâche $T_0 = T_1^*[> T_2 >> T_3$ qui est traduite en $T_0 = (T_1^N) >> T_2 >> T_3$ par l'intermédiaire de la règle de traduction de l'opération *désactivation d'une tâche itérative*. La modélisation B événementiel est donnée ci-dessous :

REFINEMENT T_0
REFINE *Convertisseur/Compteur*
INVARIANT
$J(var_i, var_j) \wedge EtatDesT_0 \in (0..3) \wedge EtatLoopT_0 \in NAT \wedge DepartT_0 \in \{0,1\}$
$\wedge\ J'(var_j)$
ASSERTIONS
$G_0 \Rightarrow ((G'_0 \wedge EtatDesT_0 = 0) \vee (EtatDesT_0 = 3 \wedge DepartT_0 = 0) \vee \ldots)$
INITIALISATION
$EtatDesT_0 := 3 \parallel EtatLoopT_0 :: NAT \parallel DepartT_0 := 0$

EVENTS

$Evt_0 =$
SELECT
$G'_0 \wedge EtatDesT_0 = 0$
THEN
S'_0
END ;

$Evt_{InitLoop1} =$
SELECT
$EtatDesT_0 = 3 \wedge$
$DepartT_0 = 0$
THEN
$EtatLoopT_0 :\in NAT \parallel$
$DepartT_0 := 1$
END ;

$Evt_{Loop1} =$
SELECT
$G_1 \wedge EtatDesT_0 = 3 \wedge$
$EtatLoopT_0 > 0 \wedge DepartT_0 = 1$
THEN
$EtatLoopT_0 := EtatLoopT_0 - 1$
$\parallel\ S_1$
END ;

$Evt_1 =$
SELECT
$DepartT_0 = 1 \wedge$
$EtatDesT_0 = 3 \wedge$
$EtatLoopT_0 = 0$
THEN
$EtatDesT_0 := 2$
END ;

$Evt_2 =$
SELECT
$G_2 \wedge$
$EtatDesT_0 = 2$
THEN
evtClickQuitButton
$\parallel\ EtatDesT_0 := 1$
END ;

$Evt_3 =$
SELECT
$G_3 \wedge$
$EtatDesT_0 = 1$
THEN
evtClickQuitApplication $\parallel$
$EtatDesT_0 := 0$
END ;

Ici, l'expression G_1 est la garde de la tâche T_1 et ne s'applique qu'au niveau de l'événement Evt_{Loop1} où le traitement de la tâche est effectué. N est un nombre naturel arbitraire montrant que l'itération est réalisée un nombre arbitraire de fois.

$EtatDesT_0$ est le variant qui contrôle la décomposition de la tâche T_0. Quand ce variant atteint la valeur 0, l'événement Evt_0 est déclenché et redonne la main à l'abstraction au moyen de la substitution *identité*.

Les événements associés aux tâches atomiques T_2 et T_3 font appels aux événements du contrôleur de dialogue *evtClickQuitButton* et *evtClickQuitApplication*.

Notons que l'utilisation de la substitution *devient élément de* à la place de la substitution *choix non borné* simplifie l'écriture du contenu de la clause **ASSERTIONS**.

Deuxième raffinement : le second niveau de décomposition introduit l'itération pour traduire la tâche $T_1 = T_4^3 >> T_5$ où la règle de l'itération est exploitée. La modélisation B événementiel est donnée ci-dessous :

REFINEMENT T_1
REFINE T_0
INVARIANT
$K(var_i, var_j, var_k) \wedge EtatLoopT_1 \in NAT \wedge EtatActT_1 \in (0..2) \wedge K'(var_k)$
ASSERTIONS
$((G_0 \wedge EtatDesT_0 = 0) \vee (EtatDesT_0 = 3 \wedge DepartT_0 = 0) \vee \ldots)$
$\Rightarrow ((EtatLoopT_1 > 0 \wedge G_4 \wedge EtatActT_1 = 2) \vee (EtatLoopT_1 = 0 \wedge \ldots) \vee \ldots)$
INITIALISATION
$EtatLoopT_1 := 3 \parallel EtatActT_1 := 2$

EVENTS

$Evt_4 =$
SELECT
$EtatLoopT_1 > 0 \wedge G_4 \wedge$
$EtatActT_1 = 2$
THEN
$EtatLoopT_1 := EtatLoopT_1 - 1 \parallel$
$S_{Loop4} \parallel EtatActT_1 := 1$
END ;

Evt_5
SELECT
$EtatLoopT_1 = 0 \wedge G_5$
$EtatActT_1 = 1$
THEN
$EtatActT_1 := 0 \parallel$
S_5
END ;

Evt_1
SELECT
$EtatActT_1 = 0 \wedge$
$EtatDesT_0 = 3 \wedge \ldots$
THEN
S_1'
END ;

$Evt_{Desact1}$
SELECT
$EtatActT_1 \in \{1, 2\}$
THEN
$EtatActT_1 := 0 \parallel$
$EtatDesT_0 := 2$
END ;

Le variant de la boucle est décrit par $EtatLoopT_1$. L'événement Evt_5 est déclenché après que Evt_4 ait été déclenché trois fois. A la fin du déclenchement de Evt_5, la modélisation redonne la main à l'abstraction.

En outre, l'événement $Evt_{Desact1}$ décrit le fait que la désactivation peut intervenir dans les états observables de la tâches T_1. Le corps de cet événement modifie les valeurs des variants de telle sorte que l'événement de l'abstraction Evt_2 soit déclenchable afin de traduire le fait que sa tâche associée T_2 désactive la tâche itérative T_1.

Troisième raffinement : ce troisième niveau de décomposition concerne les tâches $T_4 = T_6^*[> T_7 >> T_8$ et $T_5 = T_9 >> T_{10}$. Notons aussi que nous aurions pu modéliser ces deux tâches par deux raffinements successifs.

La tâche T_4 est traduite en opérateurs basiques par $T_6^N >> T_7 >> T_8$ et sa modélisation en B événementiel est donnée ci-dessous :

REFINEMENT T_4 *et* $T5$
REFINE T_1
INVARIANT
$L(var_i, var_j, var_k, var_l) \wedge EtatLoopT_4 \in NAT \wedge EtatDesT_4 \in (0..3) \wedge$
$EtatActT_5 \in (0..2) \wedge DepartT_4 \in \{0,1\} \wedge L'(var_l)$
ASSERTIONS
$((EtatLoopT_1 > 0 \wedge G_4 \wedge EtatActT_1 = 2) \vee (EtatLoopT_1 = 0 \wedge \ldots) \vee \ldots)$
$\Rightarrow ((G'_4 \wedge EtatDesT_4 = 0 \ \wedge \ldots) \vee (EtatDesT_4 = 3 \wedge DepartT_4 = 0) \vee \ldots)$
INITIALISATION
$EtatLoopT_4 :: NAT \parallel EtatDesT_4 := 3 \parallel EtatActT_5 := 2 \parallel DepartT_4 := 0$

EVENTS

$Evt_4 =$	$Evt_{InitLoop6}$	Evt_{Loop6}
SELECT	**SELECT**	**SELECT**
$G'_4 \wedge EtatDesT_4 = 0 \wedge$	$EtatDesT_4 = 3 \wedge$	$G_6 \wedge EtatDesT_4 = 3$
$EtatLoopT_1 > 0 \wedge \ldots$	$DepartT_4 = 0$	$EtatLoopT_4 > 0 \wedge DepartT_4 = 1$
THEN	**THEN**	**THEN**
S'_4	$DepartT_4 := 1 \parallel$	$evtInputValue \parallel$
END ;	$EtatLoopT_4 :\in NAT$	$EtatLoopT_4 := EtatLoopT_4 - 1$
	END ;	**END ;**

$Evt_6 =$	$Evt_7 =$	$Evt_8 =$
SELECT	**SELECT**	**SELECT**
$EtatLoopT_4 = 0 \wedge$	$G_7 \wedge$	$G_8 \wedge$
$EtatDesT_4 = 3 \wedge$	$EtatDesT_4 = 2$	$EtatDesT_4 = 1$
$DepartT_4 = 1$	**THEN**	**THEN**
THEN	$S_7 \parallel EtatDesT_4 := 1$	$evtDisplayValue \wedge$
$EtatDesT_4 = 2$	**END ;**	$EtatDesT_4 := 0$
END ;		**END ;**

Nous traduisons dans cette modélisation le fait que la tâche T_6 est une tâche itérative désactivable par la tâche T_7. Les tâches T_6 et T_8 sont atomiques et les événements du contrôleur de dialogue sont atteints $evtInputValue$ et $evtDisplayValue$.

La valeur du variant de la boucle est obtenue par la substitution *devient élément de* dans le corps de l'événement $Evt_{InitLoop6}$.

La tâche $T_5 = T_9 >> T_{10}$ est modélisée par trois événements présentés ci-dessous :

$Evt_9 =$	$Evt_{10} =$	$Evt_5 =$
SELECT	**SELECT**	**SELECT**
$EtatActT_5 = 2 \wedge G_9$	$EtatActT_5 = 1 \wedge G_{10}$	$G'_5 \wedge EtatActT_5 = 0 \wedge$
THEN	**THEN**	$EtatLoopT_1 = 0 \wedge \ldots$
$EtatActT_5 := 1 \parallel$	$EtatActT_5 := 0 \parallel$	**THEN**
$evtClickInitialiser$	$evtInitialiser$	S'_5
END ;	**END ;**	**END ;**

Deux nouveaux événements Evt_9 et Evt_{10} sont introduits qui appellent en séquence des événements du contrôleur de dialogue $evtClickInitialiser$ et $evtInitialiser$.

Par ailleurs, nous enrichissons l'événement $Evt_{Desact1}$ de telle sorte qu'il gère la désactivation dans tous les états observables de la tâche T_1 :

$Evt_{Desact1}$
SELECT
$EtatActT_1 \in \{1, 2\} \wedge EtatDesT_4 \in (1..3) \wedge EtatActT_5 \in \{1, 2\}$
THEN
$EtatActT_1 := 0 \parallel EtatDesT_4 := 0 \parallel EtatActT_5 := 0 \parallel$
$EtatDesT_0 := 2$
END ;

L'enrichissement de l'événement $Evt_{Desact1}$ fait intervenir tous les variants relatifs à la désactivation des tâches T_6, T_7, T_8, T_9, T_{10}. Cet événement est déclenché si les variants ($EtatActT_1$, $EtatDesT_4$, $EtatActT_5$) sont dans des états observables. Dans ce cas, le corps de l'événement modifie les variants ($EtatActT_1$, $EtatDesT_4$, $EtatActT_5$) de telle sorte que l'événement $Evt_{Desact1}$ redonne la main aux événements de l'abstraction (Evt_1, Evt_4 et Evt_5) (propriété de non blocage).

Finalement, la désactivation permet de rendre activable la tâche T_2 puisque le corps de l'événement $Evt_{Desact1}$ modifie le variant $EtatDesT_0 := 2$.

Quatrième raffinement : ce dernier raffinement traduit le choix entre la tâche T_{11} et T_{12}.

REFINEMENT T_7
REFINE T_4 *et* $T5$
INVARIANT
$M(var_i, var_j, var_k, var_l, var_m) \wedge EtatChoixT_7 \in (0..2)$
ASSERTIONS
$((G_4 \wedge EtatDesT_4 = 0 \ \wedge \ldots) \vee (EtatDesT_4 = 3 \wedge DepartT_4 = 0) \vee \ldots)$
$\Rightarrow ((EtatChoixT_7 = 1 \wedge G_{11}) \vee (EtatChoixT_7 = 2 \wedge G_{12}) \vee \ldots)$
INITIALISATION
$EtatChoixT_7 :\in \{1, 2\}$

EVENTS

$Evt_{11} =$	$Evt_{12} =$	$Evt_7 =$
SELECT	**SELECT**	**SELECT**
$EtatChoixT_7 = 1 \wedge G_{11}$	$EtatChoixT_7 = 2 \wedge G_{12}$	$G'_7 \wedge EtatChoixT_7 = 0$
THEN	**THEN**	**THEN**
$evtClickEuro \ \|\|$	$evtClickFranc \ \|\|$	S'_7
$EtatChoixT_7 := 0$	$EtatChoixT_7 := 0$	**END ;**
END ;	**END ;**	

L'opération du choix est obtenue par le variant $EtatChoixT_7$ dont sa valeur est déterminée dans la clause **INITIALISATION** par la substitution *devient élément de*. Après le traitement de l'événement Evt_{11} ou de celui de Evt_{12}, l'événement Evt_7 prend la main.

Dans la modélisation ci-dessous, nous présentons l'enrichissement de l'événement $Evt_{Desact1}$ qui décrit la désactivation de la tâche T_1 par T_2 :

$Evt_{Desact1}$
SELECT
$EtatActT_1 \in \{1,2\} \wedge EtatDesT_4 \in (1..3) \wedge EtatActT_5 \in \{1,2\} \wedge$
$EtatChoixT_7 \in \{1,2\}$
THEN
$EtatActT_1 := 0 \parallel EtatDesT_4 := 0 \parallel EtatActT_5 := 0 \parallel$
$EtatChoixT_7 := 0 \parallel EtatDesT_0 := 2$
END ;

En comparaison avec le raffinement précédent T_4 *et* $T5$, cette modélisation introduit le fait que la désactivation peut intervenir dans les états observables des sous-tâches de la tâche T_7 ($EtatChoixT_7$). Si la garde de cet événement est vraie, la tâche T_2 est activée.

2.5.2.4 Validation de tâches par l'approche implicite

La modélisation précédente a décrit la décomposition d'une tâche *Application Convertisseur/Compteur* par l'intermédiaire des constructions *tâche itérative*, *désactivation d'une tâche itérative*, *activation*, *choix*, *tâche atomique* pour valider notre étude de cas. 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.

Toutes ces propriétés ont été exprimées en B événementiel par l'intermédiaire des clauses **INVARIANT** (non détaillées) et **ASSERTIONS**. Toutes les obligations de preuve qui en sont générées par l'Atellier B, doivent être déchargées pour que les propriétés soient respectées, sinon il faudra réitérer la processus de conception du système interactif (en supposant que l'analyse de tâche est correcte).

Notons que l'utilisation d'un nombre naturel arbitraire dans la formalisation des opérateurs *interruption* et *désactivation* a été rendue possible par l'intermédiaire de la sub-

stitution généralisée *choix non borné* **ANY WHERE THEN**[8]. La possibilité d'utiliser un nombre naturel arbitraire permet de prendre en compte tous les cas possibles pour la modélisation des descriptions de tâches. Cette approche est difficile à représenter dans les techniques de vérification sur modèles, sauf si une valeur du nombre naturel est fixée. Le nombre d'états augmente quand ce nombre naturel augmente.

2.5.3 Bilan sur l'approche à base d'événements

Nous avons présenté dans cette section l'approche à base d'événements qui se fonde sur les travaux de l'approche du LISI pour modéliser formellement les modules boîte à outils, présentation, adaptateur du domaine et domaine de l'architecture ARCH. Nos travaux se sont tout d'abord intéressés à la description du comportement du contrôleur de dialogue au moyen de systèmes de transitions étiquetés. Le 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 les conditions d'apparition de l'événement (garde) et l'action résultante lorsque la garde est vraie.

De plus, ce travail ne se limite pas à la modélisation de systèmes finis et la technique de raffinement de la méthode B permet de répartir la complexité des preuves. L'invariant de collage permet de préserver les propriétés de l'abstraction après raffinement. Dans les approches ascendantes, chaque système de transitions est modélisé et la description du système complet est obtenue par composition de tous les automates. En général, à chaque composition, toutes les propriétés sont prouvées une nouvelle fois. L'augmentation de la complexité de la description du système après composition, fait que la preuve des propriétés devient plus compliquée également.

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 du LISI 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

[8]Simplifiée dans la modélisation de l'étude de cas par la substitution *devient élément de*

du modèles de tâches et de la conception à l'aide de la même technique formel 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 à faire une quelconque expertise sur les spécifications (à l'opposé de l'approche explicite expérimentée en premier).

2.6 Conclusion

Cette section achève le chapitre décrivant le développement d'interfaces homme-machine par une approche complète formelle basée sur la méthode B. Au travers d'une étude de cas interactive, nous avons montré que l'utilisation de la technique B permettait d'exprimer des propriétés à tous les niveaux du développement et de les vérifier.

Nous nous sommes tout d'abord intéressés à exploiter les travaux du LISI concernant la conception de systèmes interactifs par composition modulaire. A cela, nous avons ajouté la validation de tâches qui a permis, à la manière des diagrammes de séquences d'UML, la description explicite de traces de tâches. Les traces représentant chacune une séquence d'opérations du contrôleur de dialogue sont obtenues par des raffinements successifs. La validation de tâches consiste alors à construire explicitement la tâche utilisateur. Cependant, cette approche a besoin de l'expertise de l'utilisateur et de la connaissance des éléments de conception en général et du contrôleur de dialogue en particulier (ce qui n'est pas le cas des ergonomes ou psychologues en charge de ce type de validation).

Puis, nous avons étendu les travaux sur la conception en intégrant une description du contrôleur de dialogue à base d'événements en codant avec le B événementiel des systèmes de transitions et leur composition. Cette forme de conception dite descendante a permis l'introduction de nouveaux composants grâce au raffinement. Ces travaux ont permis la description d'un véritable dialogue pouvant être exploité pour la validation de tâches. Nous avons exploité la notation CTT et nous avons montré qu'il était possible de traduire toutes les constructions de cette notation par l'extension B événementiel. Du point de vue de l'utilisation des techniques formelles, la robustesse de notre approche a été rendue possible par le raffinement et par la technique basée sur la preuve :

- en effet, le raffinement permet d'introduire graduellement les différentes décompositions de tâches à partir d'une modélisation abstraite jusqu'à une modélisation concrète. Ces étapes de raffinement ont l'avantage d'éviter d'avoir à prouver à chaque enrichissement l'intégralité des preuves. Seul les invariants de collage et les assertions seront prouvés ;

– ensuite nous avons contourné les problèmes dus à l'explosion du nombre d'états. En effet, quand une itération est réalisée un nombre arbitraire de fois, il n'est pas nécessaire de prouver les propriétés pour des valeurs fixées de l'itération. Un nombre arbitraire de fois est utilisé dans la technique B.

D'un point de vue général, précisons que nous n'avons pas suggéré de nouveaux modèles ou de notations dans le domaine de l'interaction homme-machine. Cette approche a permis d'intégrer des notations et des modèles hétérogènes dans une technique formelle unique : la méthode B.

Chapitre 3

SUIDT : Une approche expérimentale pour la construction d'interfaces utilisateurs sûres

3.1 Introduction

Nous avons montré dans la section 1.4 qu'il existait principalement deux familles d'outils qui permettaient d'aider un concepteur à réaliser une application graphique interactive. Elles se différencient selon le point de départ à partir duquel elles cherchent à construire l'application.

D'une part, les approches ascendantes privilégient l'aspect « présentation » et permettent de réaliser le noyau fonctionnel au fur et à mesure de la construction de l'interface. L'association de ces environnements à des langages interprétés permet de raccourcir le cycle de développement, avec l'inconvénient majeur de rendre plus difficile la sécurisation de l'application finale. D'autre part, les approches descendantes tirent parti de spécifications, notamment du noyau fonctionnel, le plus souvent enrichies de descriptions supplémentaires, pour construire ou générer l'application interactive. L'apprentissage de langages spécifiques, la nécessaire traduction des modèles utilisés en code exécutable et l'absence de sémantique claire pour ces approches complexifient le processus de création.

Pourtant, lorsqu'un noyau fonctionnel peut être développé de façon indépendante de toute perspective d'interaction homme-machine, ses spécifications, pour peu qu'elles soient formalisées et complètes, constituent un vrai modèle, qui peut être pris comme point de départ pour la construction de l'application. Rapprocher les deux familles d'outils consis-

terait, en partant des spécifications du noyau fonctionnel, à créer un outil capable de construire interactivement une application directement exécutable tout en conservant la sémantique formelle. Par ailleurs, une telle approche permettrait d'assurer que les propriétés du noyau fonctionnel sont maintenues tout au long du processus de développement de l'IHM.

Cette séparation de la conception du noyau fonctionnel et de l'interface utilisateur est un des aspects originaux de l'approche que nous proposons. Elle permet naturellement de distinguer deux profils très différents qui interviennent dans le processus de développement d'une application interactive. Tout d'abord, la conception du noyau fonctionnel ne peut être effectuée que par un **spécialiste** qui maîtrise les aspects classiques de la programmation (programmation classique, méthodes formelles, etc). A l'inverse, la construction de l'interface utilisateur n'impose aucun pré-requis en matière de connaissances informatiques. Sous condition de lui fournir les outils adéquats, un utilisateur final (end-user) au sens de [Lie01] est à-même de construire une application interactive en totale cohérence avec les contraintes du noyau fonctionnel. Dans la suite de ce chapitre nous appelons **concepteur d'IHM** cet utilisateur final dont les connaissances ne lui permettent pas de construire la partie du spécialiste.

Dans un premier temps, nous discuterons sur les généralités de l'approche décrite ci-dessus en présentant différents outils qui ont permis de la mettre en place et plus particulièrement l'outil SUIDT, sujet principal de notre contribution concernant les approches expérimentales pour la construction d'interfaces utilisateurs sûres. Dans le but de justifier du caractère original de cette approche, nous aborderons dans une deuxième section l'apport d'un noyau fonctionnel développé formellement et existant dans une démarche de conception. Puis dans les troisième et quatrième sections nous étudierons la validation de la spécification abstraite et concrète. Nous verrons comment cette distinction entre la spécification abstraite et spécification concrète permet de séparer les activités de conception et de validation.

Tout au long de ce chapitre nous justifierons de l'originalité de l'approche face aux approches expérimentales étudiées dans la section 1.4 et nous montrerons comment cette approche est capable de vérifier et de valider des propriétés de l'IHM. Nous illustrerons nos propos avec l'étude de cas du convertisseur francs/euros et du compteur utilisé dans le chapitre précédent.

3.2 Généralités sur l'approche expérimentale

L'approche introduite précédemment a abouti à deux systèmes de conception d'application interactive : les outils GenBUILD[1] (Générateur Builder) et SUIDT[2] (Safe User Interface Development Tool). Le premier outil génère automatiquement une application interactive à partir de l'interface d'un noyau fonctionnel préalablement développé. L'originalité de l'outil réside dans le fait que cette application générée automatiquement est couplée à un constructeur d'interfaces qui permet de réaliser une véritable application interactive tout en conservant le lien avec le noyau fonctionnel. Cependant ce premier outil ne cherchait qu'à montrer la faisabilité d'une approche s'appuyant sur l'existence préalable d'un noyau fonctionnel et souffrait de nombreuses limites quant à la conception de l'interface utilisateur. Le deuxième outil, SUIDT, est le sujet de notre contribution. Il vise à dépasser les limites du système GenBUILD tout en garantissant que ses fonctionnalités de développement permettent aux concepteurs de construire des applications assurant des critères de qualité et notamment celui d'utilisabilité.

Tout naturellement, nous avons choisi de présenter dans cette section l'environnement GenBUILD, en discutant brièvement d'une part des solutions retenues pour la mise en œuvre de cette démarche et d'autre part des limites de l'outil. Ensuite, nous présentons l'outil SUIDT et ses caractéristiques.

3.2.1 L'approche initiale : GenBUILD

Ce premier outil a été développé pour démontrer qu'il était possible de fournir un environnement de conception susceptible d'assurer tout au long du développement d'une IHM des liaisons avec un noyau fonctionnel existant.

Dans un premier temps, le processus de développement a débuté par la conception d'un noyau fonctionnel. Pour cela, nous avons utilisé le langage impératif *C* et nous y avons ajouté des descriptions supplémentaires afin d'enrichir la sémantique de ce langage. Il a été aussi tenu compte des principes de développement modulaire dans [Pie91] et du travail de [Fek96a] sur les services minimums pour l'interaction homme-machine. Le modèle obtenu correspondait en fait à des pré (nécessite) et postconditions (entraîne). Les spécifications semi-formelles du noyau fonctionnel nous ont alors permis de fournir des services via une API[3]. Il a donc été possible d'automatiser le lien avec le noyau fonctionnel par un outil : le **Générateur**. Cet outil exploite la signature des fonctions

[1] Disponible en téléchargement à l'adresse *http ://www.lisi.ensma.fr/members/baron*
[2] Disponible en téléchargement à l'adresse *http ://www.lisi.ensma.fr/ihm/suidt*
[3] Application Program Interface : Interface de Programmation d'Applications

et des descriptions supplémentaires pour construire une interface utilisateur du noyau fonctionnel. A ce niveau, le rôle du spécialiste qui maîtrise les aspects classiques de la programmation est terminé. Le second acteur que nous avions qualifié de concepteur d'IHM s'occupe quant à lui de la construction de l'IHM du système interactif. Il utilise pour cela l'outil : **Builder**. Nous présentons sur la figure 3.1 une copie d'écran de cet outil.

Sur la gauche identifié par le repère 1 (figure 3.1), le module *animateur* qui a été généré automatiquement par l'outil **Générateur** permet d'exécuter interactivement des fonctions du noyau fonctionnel. Chaque fonction du noyau fonctionnel est associée à un bouton, qui permet de l'activer interactivement. Lorsque des paramètres sont requis, une boîte de dialogue s'affiche et permet à l'utilisateur de les saisir. La liaison en continu avec le noyau fonctionnel permet de vérifier que les valeurs des paramètres saisis sont correctes. Les fonctions sont décrites textuellement et l'état courant du noyau sémantique est retourné au travers de l'ensemble des résultats des fonctions. Cette interface permet au concepteur d'IHM de pouvoir tester un noyau fonctionnel sans avoir de connaissance en programmation impérative.

Sur la partie droite de la figure 3.1 identifiée par le repère 2, nous pouvons voir l'interface du module constructeur d'interfaces qui ressemble aux générateurs de présentation de la section 1.4.1.3. Plutôt que de partir de la présentation réalisée automatiquement pour chercher à l'améliorer comme le propose MOBI-D, nous avons choisi de fournir un espace de conception indépendant du module animateur. Ceci permet de conserver une visualisation de l'état du modèle en cours de conception. Ce module permet de construire l'interface en déposant des composants graphiques sur une zone de travail. La programmation du dialogue de l'application se fait par l'association des fonctions du noyau fonctionnel et des événements émis par les composants graphiques. Le retour d'information résultant d'une interaction est aussi développé suivant ce principe. Toutes les associations effectuées par l'utilisateur respectent les contraintes des descriptions du noyau fonctionnel.

L'outil GenBUILD combine donc les avantages des approches ascendantes et descendantes pour fournir un système basé sur modèles comparable à ceux étudiés dans la section 1.4.2. Nous montrons sur la figure 3.2 l'architecture de développement suivant l'architecture générique présentée sur la figure 1.25. Nous remarquons que les aspects liés aux règles de conception et aux guides de style n'ont pas été pris en compte (rectangles arrondis en traits pointillés). Par ailleurs, la génération de l'application finale (croix sur l'utilisateur) n'a pas été étudiée, puisque nous nous étions seulement occupés de la partie conception. Toutefois, l'utilisateur est à même de pouvoir tester l'application en cours de développement par l'intermédiaire de l'interpréteur. GenBUILD fournit un éditeur de développement interactif (EDI) pour construire de façon textuelle le noyau fonctionnel. Puis, le **générateur** automatise la construction de l'adaptateur de l'application. C'est

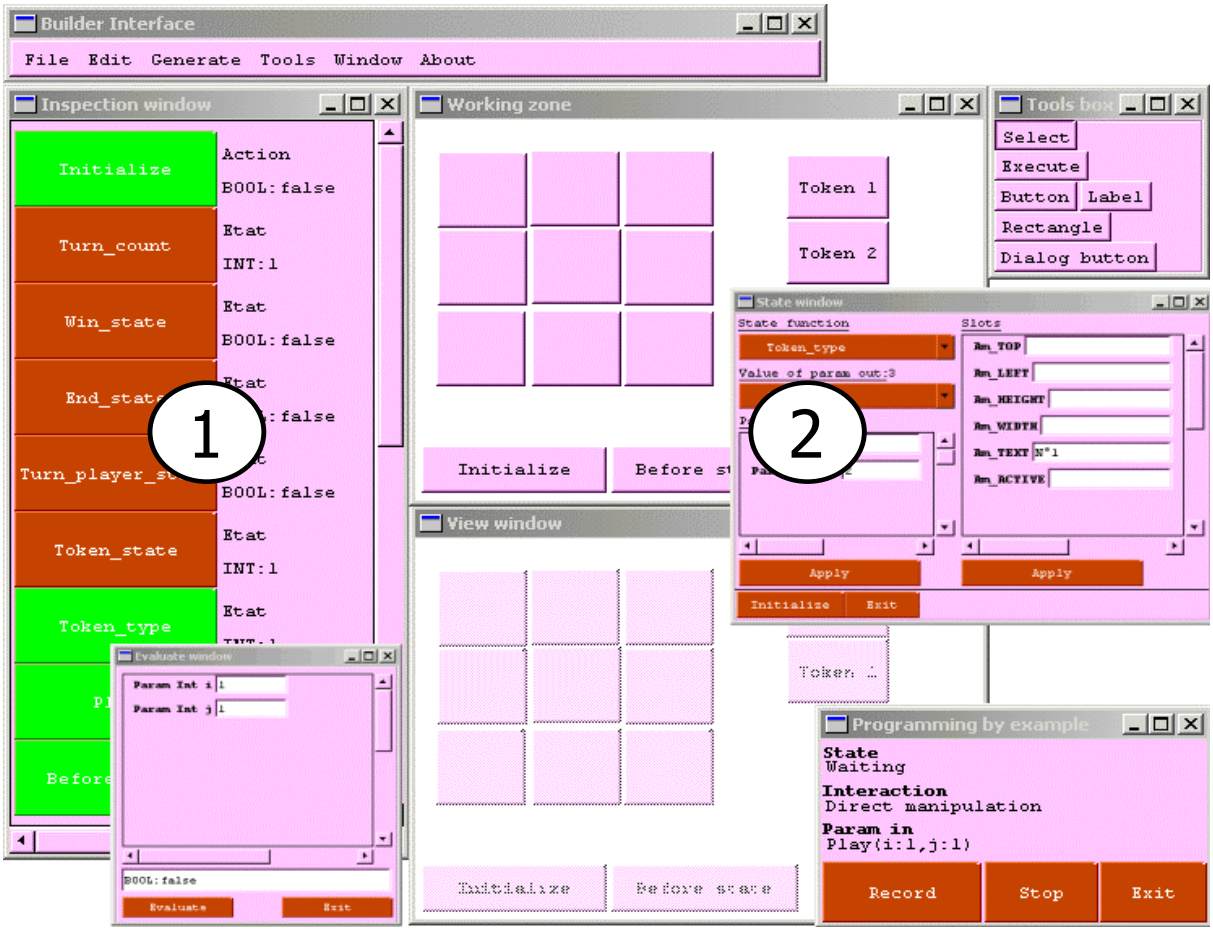

Fig. 3.1 – Capture d'écran de l'outil GenBUILD.

ensuite à l'aide de cet adaptateur et de l'outil **Builder** que le prototype de l'application est construit. Enfin, le concepteur d'IHM peut tester à deux niveaux de conception l'application en construction avec d'une part l'animateur de noyau fonctionnel et d'autre part avec l'interpréteur.

Les apports principaux de cet outil peuvent être caractérisés par :

1. en premier lieu, tout au long de la conception, le concepteur peut alterner phase de test et phase d'édition sans perte du contexte d'exécution puisque le noyau fonctionnel est toujours en exécution. Il n'existe pas de phase de compilation qui d'une part ralentirait le processus de développement et d'autre part supprimerait ce contexte d'exécution ;
2. ensuite, l'outil GenBUILD s'appuie sur des descriptions semi-formelles du noyau fonctionnel pour assurer que des appels aux fonctions sont corrects ; c'est un premier pas vers une plus grande sûreté de l'application ;

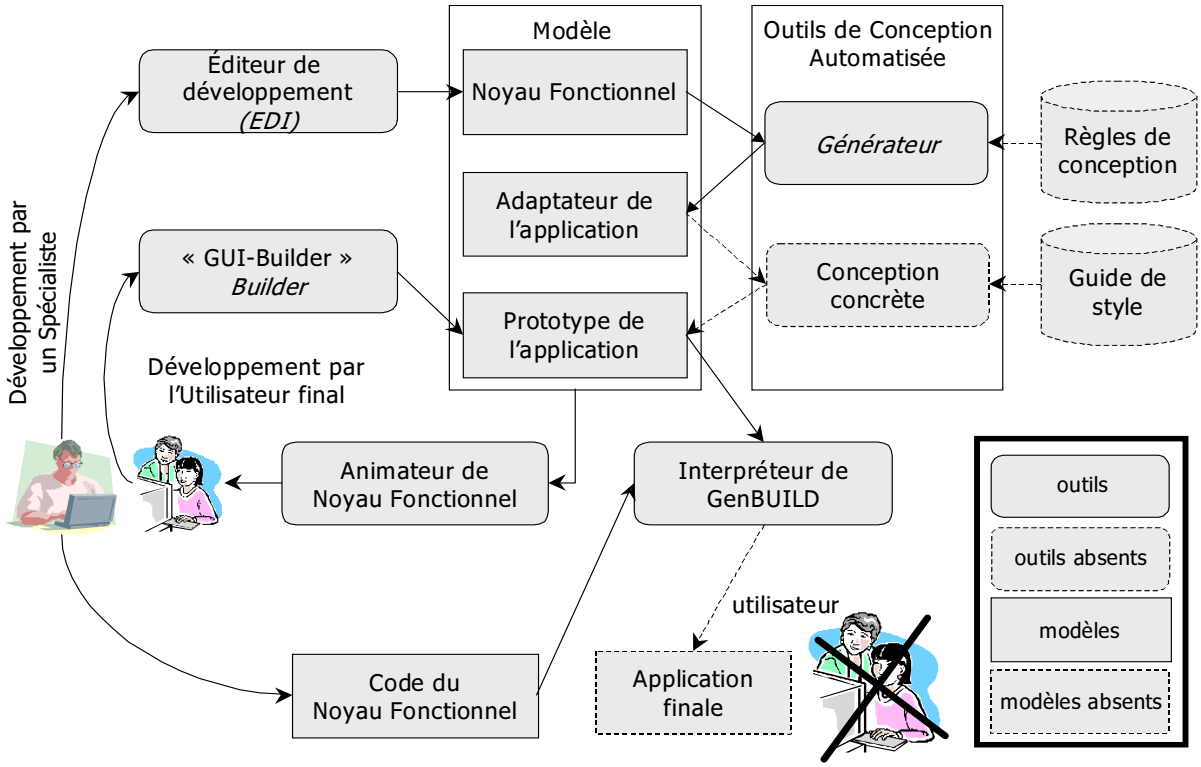

Fig. 3.2 – Architecture générique de l'environnement GenBUILD.

3. enfin, cette approche a l'avantage de décomposer la conception de l'application suivant des modules bien distinctes : le noyau fonctionnel, le module animateur comparable à l'adaptateur du domaine, le dialogue modélisé par une logique événementielle, le module constructeur d'interfaces construisant la présentation et enfin la boîte à outils AMULET [MMM+97].

Si GenBUILD a démontré la possibilité de garantir des appels corrects du noyau fonctionnel, le bénéfice n'est pas très intéressant à ce niveau pour les raisons suivantes :

- la description du noyau fonctionnel s'appuie sur une sémantique pauvre, c'est-à-dire qu'aucun raisonnement au moment de la conception du noyau ne peut être effectué. Les descriptions supplémentaires ne peuvent être utilisées dans le cadre d'une vérification de propriétés ;
- l'absence du point de vue de l'utilisateur : aucun modèle de tâches n'est pris en compte. Le système ne peut garantir que les actions de l'utilisateur sur l'application générée peuvent être atteintes ;
- le modèle d'interaction est pauvre, cet outil n'assure qu'une liaison directe entre les composants graphiques et les éléments du noyau fonctionnel ;
- l'absence d'outil de vérification et de validation.

3.2.2 Présentation générale de l'approche SUIDT

Les limites de l'outil GenBUILD évoquées précédemment nous ont conduit à étendre la démarche initiale en proposant les améliorations décrites ci-dessous :

- la présence d'une vraie description formelle permet le raisonnement sur le noyau fonctionnel. Nous avons choisi d'intégrer pour cela la méthode formelle B ;
- la prise en compte du point de vue de l'utilisateur est obtenue en intégrant un modèle de tâches CTT (ConcurTaskTrees) dans la démarche ;
- le modèle est enrichi par la décomposition en deux niveaux de la description de l'interface : la spécification **abstraite** et la spécification **concrète** ;
- l'intégration d'outils de test et de simulation pour vérifier les propriétés de l'IHM. Ces outils analysent le comportement des spécifications abstraites et concrètes et permet notamment de pouvoir décider si la description du système interactif doit être modifiée.

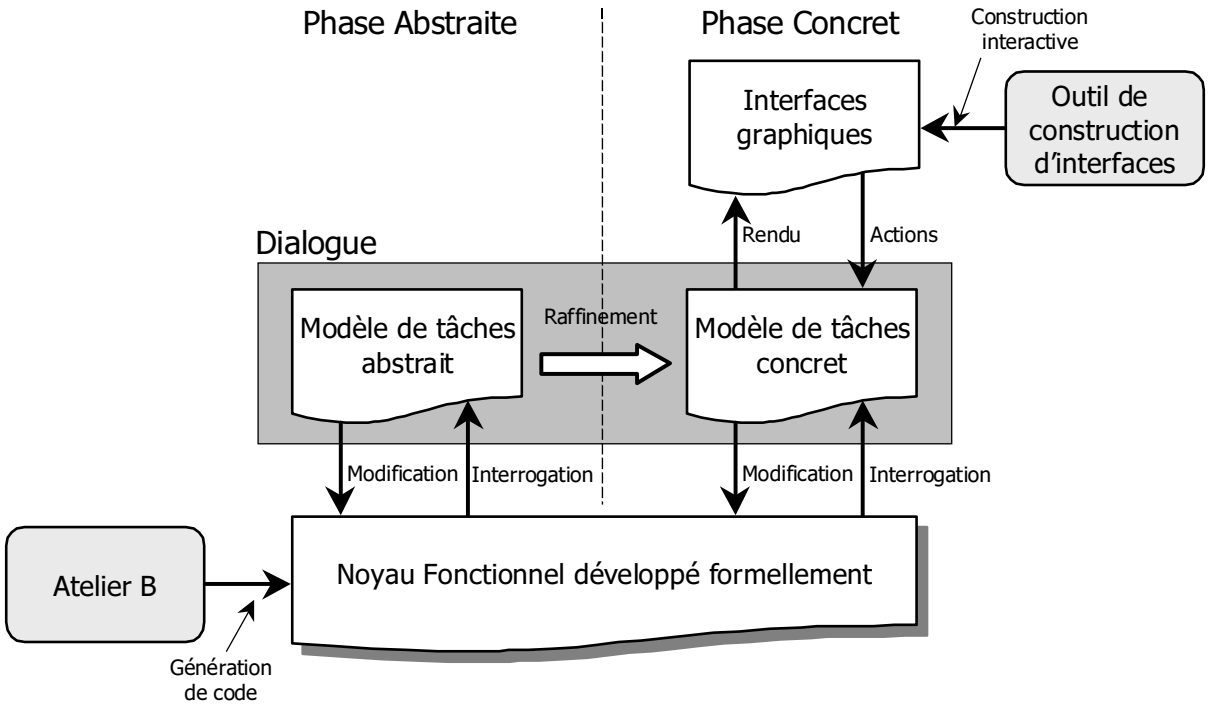

Fig. 3.3 – Description générale de l'approche SUIDT.

L'outil SUIDT associé à cette démarche permet de concevoir de manière interactive le dialogue entre un noyau fonctionnel existant développé formellement et une présentation graphique de l'interface construite de manière classique (comme montré sur la figure 3.3).

Ce dialogue est basé sur la notation de description de l'utilisateur CTT et sa construction se scinde en deux phases : une phase abstraite (partie gauche de la figure 3.3) et une phase concrète (partie droite de la figure 3.3).

1. Dans un premier temps, un modèle de tâches abstrait sans lien avec une interface graphique. Il s'appuie sur des travaux en amont issus des ergonomes. Ce modèle de tâches abstrait a pour objectif de valider (par simulation) la faisabilité des tâches en liaison avec les fonctionnalités d'un noyau fonctionnel formel existant. Ce dernier est développé formellement au moyen de la méthode B, de l'outil Atelier B et respecte les services préconisés par [Fek96a] (notification, prévention des erreurs et annulation). La dynamique du modèle de tâches abstrait se base sur des contraintes de précédence (opérateurs temporels) et sur les préconditions des fonctions accesseurs (flèche *Interrogation*) du noyau fonctionnel. Les postconditions des tâches, modifiant l'état du noyau fonctionnel par les modifieurs (flèche *Modification*), sont exprimées au niveau des feuilles du modèle de tâches.

2. Dans un second temps, des interfaces graphiques sont associées à ce modèle de tâches abstrait. Elles sont développées par l'intermédiaire d'un outil classique de construction d'interfaces. Le modèle de tâches abstrait est alors raffiné depuis un niveau fonctionnel jusqu'à la spécification des interactions de l'utilisateur (flèche *Action*) et des rendus du système (flèche *Rendu*). Plus précisément seules les tâches feuilles du modèle de tâches abstrait sont raffinées dans le but de spécifier sous forme de tâches d'interactions les liens (en entrée et en sortie) avec les objets des interfaces graphiques. Le modèle obtenu porte le nom de modèle de tâches concret.

Notons que nous n'avons pas traité dans notre contribution la dernière phase qui consiste à générer un prototype fonctionnel indépendamment de l'environnement SUIDT. Nous reviendrons sur cette étape dans les perspectives de cette thèse.

Par rapport aux modèles d'architecture existants, notre approche emploie une architecture basée sur ARCH section 1.2.4.2. En effet, notre approche propose d'un côté une présentation, s'appuyant sur la boîte à outils Java/Swing, un contrôleur de dialogue (modèle de tâches abstrait et concret), une interface du noyau fonctionnel et un noyau fonctionnel existant. Cependant, le dialogue homme-machine est conçu tout en respectant les contraintes de sûreté imposées par le noyau fonctionnel, par les besoins de l'utilisateur et par la présentation.

Le développement de l'environnement SUIDT a été réalisé au moyen du langage Java, de la boîte à outils Swing, du langage B et de l'Atelier B.

3.2.3 Bilan sur les généralités de l'approche expérimentale

Nous venons de présenter dans cette section la démarche générale de notre contribution sur les approches expérimentales pour la construction d'interfaces utilisateur. Cette démarche basée sur modèles s'appuie sur l'outil SUIDT pour construire à partir d'un noyau fonctionnel développé formellement, un modèle de tâches abstrait, un modèle de tâches concret, le dialogue et la présentation.

L'intérêt de cette approche est de pouvoir utiliser un noyau fonctionnel développé formellement et sur lequel des raisonnements peuvent être effectués. Par ailleurs, l'intégration d'un modèle de tâches permet de prendre en compte au plus tôt les besoins utilisateurs afin de garantir l'utilisabilité du système à construire. Finalement, l'apport principal de cette approche est l'utilisation d'outils de vérification et de validation qui permettent tout au long du développement de respecter les contraintes des modèles afin d'assurer un développement sûr d'une application interactive.

Intéressons nous maintenant à la description des modèles et des outils qui composent cette approche.

3.3 Intégration d'un noyau fonctionnel développé formellement dans une approche de système basés sur modèles

L'intégration d'un noyau fonctionnel développé formellement à une IHM ne peut être effectuée que si ce noyau fonctionnel est exempt de toute erreur et où des propriétés de robustesse sont vérifiées préalablement. Seul l'apport de techniques formelles permet de vérifier au plus tôt ces propriétés. La démarche initiale GenBUILD, à base d'expressions correspondant en fait à des pré/postconditions, nous a conduit tout naturellement à choisir comme formalisme une méthode basée sur modèles, où l'essentiel de la sémantique peut s'exprimer sous la forme d'invariants et de pré/postconditions : la méthode B. Les outils que nous avons développés peuvent ainsi s'appuyer sur la sémantique des spécifications du noyau fonctionnel pour en garantir les propriétés.

Nous résumons l'intégration d'un noyau fonctionnel développé formellement sur la figure 3.4. Dans un premier temps, des spécifications formelles B du noyau fonctionnel sont établies puis vérifiées et validées avec l'outil Atelier B. A la suite d'un développement formel, cet outil permet de générer du code C^{++} transformé en code Java par un traducteur. Le code obtenu correspond au noyau fonctionnel de l'application. De même, l'outil *géné-*

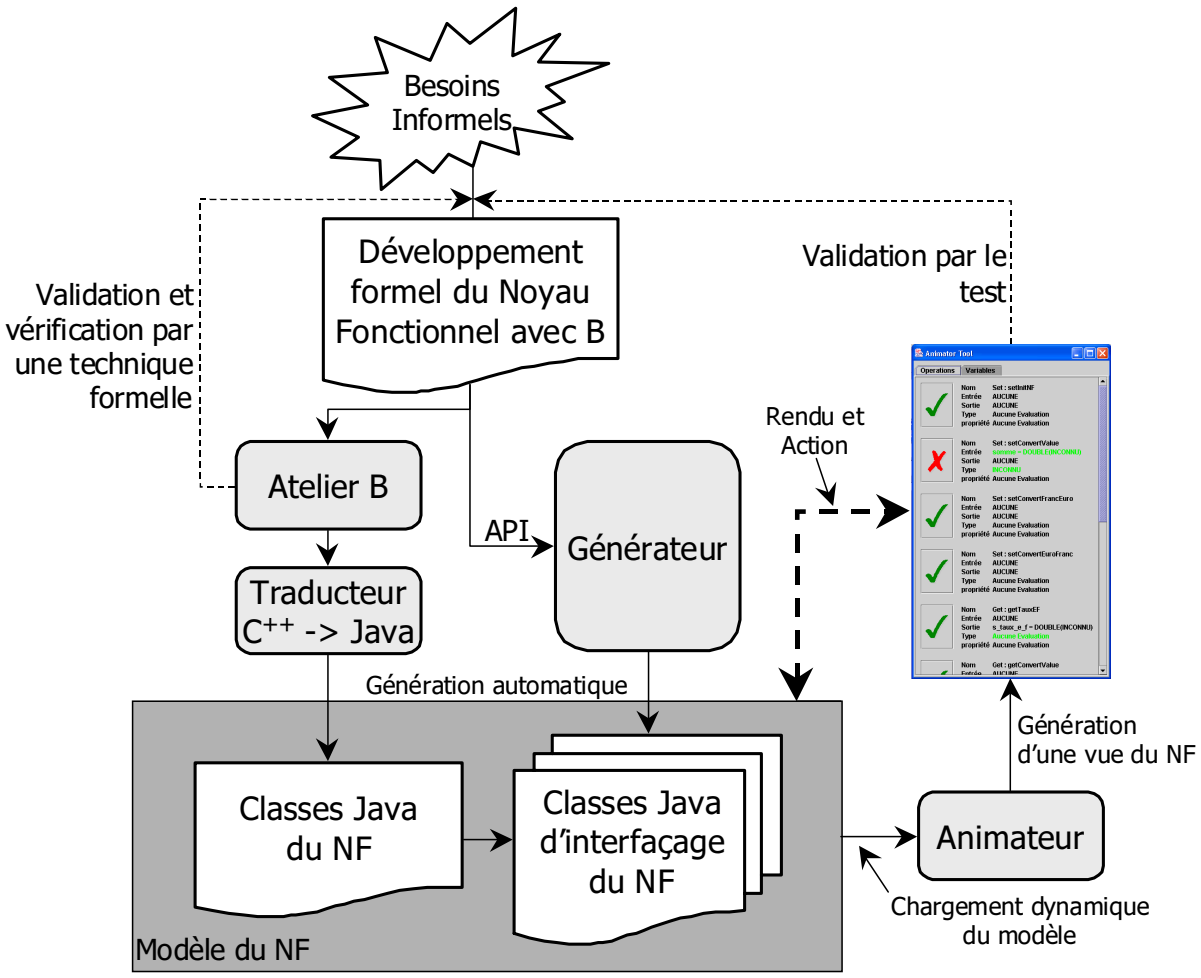

Fig. 3.4 – Démarche pour la prise en compte d'un noyau fonctionnel développé formellement

rateur extrait à partir des spécifications B des informations du noyau fonctionnel pour générer des classes Java d'interfaçage. Ces deux générations permettent de fournir un noyau fonctionnel développé formellement à notre approche. Ensuite, en chargeant dynamiquement les classes Java d'interfaçage du noyau fonctionnel, l'outil animateur permet de construire une interface graphique. Chaque action sur l'animateur est transmise au noyau fonctionnel qui notifie la présentation de l'animateur pour une mise à jour. Via cette interface graphique, le concepteur peut alors valider par le test la conformité du noyau fonctionnel.

Nous abordons dans cette section, deux points de notre approche. Tout d'abord la conception du noyau fonctionnel développé formellement qui devra d'une part respecter certains services liés au domaine des IHM (notification, annulation, prévention d'erreurs) et d'autre part fournir une API de communication. Dans un second temps, nous présen-

terons la conception du module de l'adaptateur de présentation et plus précisément des outils Générateur et Animateur. Enfin, dans une dernière section, nous discuterons de l'originalité de l'intégration d'un noyau fonctionnel développé formellement par rapport à des approches similaires.

3.3.1 Conception du noyau fonctionnel développé formellement

A ce niveau, nous adoptons une modélisation basée sur ARCH pour la conception du noyau fonctionnel, figure 3.5. Nous pouvons aussi mettre en avant la frontière des compétences exigées par cette conception et la nécessité pour le concepteur de posséder des connaissances précises en techniques formelles.

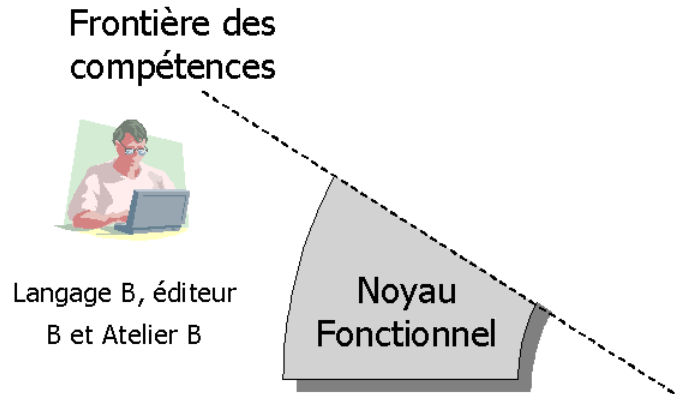

Fig. 3.5 – Conception du module Noyau Fonctionnel de l'architecture ARCH.

La méthode B a été choisie car, rappelons-le, c'est une méthode couvrant tout le spectre depuis la spécification jusqu'au code. De plus, la pertinence de la méthode B est d'assurer le respect des propriétés, exprimées dans les spécifications, tout au long du processus de développement. Par ailleurs, le langage B est parfaitement instrumenté par l'environnement de développement Atelier B. Finalement, l'utilisation de la méthode B nous permet de réutiliser des travaux de l'approche du LISI et plus précisément en ce qui concerne la modélisation du noyau fonctionnel.

Nous avons employé la méthode B dans sa version « classique ». Ici, notre besoin se limite à un modèle statique qui fournit un ensemble d'entités pouvant être interrogé par l'environnement du modèle.

Par ailleurs, nous avons restreint les possibilités de la méthode B pour satisfaire nos besoins en termes de conception. D'une part, nous avons choisi de nous limiter à l'exploitation des variables de types simples (*entier*, *chaînes de caractères*, *booléen* et *réel*).

D'autre part, nous nous sommes limités à des noyaux fonctionnels simples, définis plus dans une logique de programmation procédurale que de programmation objet. Nous avons préféré porter notre attention sur les aspects interactifs et les respects des contraintes du noyau fonctionnel plutôt que sur le développement de celui-ci, nous reviendrons sur ce point dans les perspectives de cette thèse.

3.3.1.1 Conception et méthode B

Le développement B débute par la construction d'une machine abstraite de base qui reprend toutes les descriptions des besoins utilisateurs. Elle décrit pour cela les principales variables du noyau fonctionnel, typées par l'intermédiaire de la clause **INVARIANT** par des types élémentaires et/ou des ensembles. De plus des propriétés complémentaires sur ces variables s'expriment dans les invariants. Enfin, nous trouvons des opérations, qui encapsulent ces variables (les fonctions ou les procédures qui accèdent ou modifient les variables), auxquelles des préconditions qui expriment des prédicats sur les paramètres d'entrée mais aussi des prédicats établissant des contraintes pour lesquels l'invariant doit être satisfait sont définies.

Ensuite le modèle B sera raffiné, jusqu'à obtenir un dernier raffinement qui sera l'implantation finale. Cette dernière étape représente le code du noyau fonctionnel. Pendant les étapes de raffinement, les variables abstraites sont raffinées en variables concrètes (par exemple transformer les ensembles en structures de tableau).

La validation du noyau fonctionnel concerne les propriétés du modèle, c'est-à-dire les invariants. Pour prouver les invariants, il suffit de montrer que la propriété de la clause **INVARIANT** est établie pendant l'opération d'initialisation et qu'elle est préservée par chaque appel d'opération. L'invariant est conservé pendant tout le développement. En rapprochant ce processus de vérification de propriétés à notre approche, nous proposons de vérifier les préconditions (sous forme de prédicat) d'une opération avant son exécution. Ainsi, nous pourrons garantir que l'appel de l'opération respecte les propriétés décrites dans le développement du noyau fonctionnel.

3.3.1.2 Services du noyau fonctionnel

Nous avons montré dans le chapitre 1 que la définition de propriétés permet de décrire la notion d'utilisabilité d'un système interactif. Afin de satisfaire ces propriétés, l'IHM doit fournir un ensemble de fonctionnalités (retour en arrière, traitement des erreurs et mise à jour de l'affichage). Par exemple, dans le cas du traitement des erreurs liées aux propriétés de gestion des erreurs, cette fonctionnalité doit être capable de prévenir les

erreurs de l'utilisateur. Un autre exemple concerne la fonctionnalité de la mise à jour de l'affichage. Elle permet de respecter la propriété liée à la visualisation du système interactif en assurant une représentation correcte de l'état de ce système via l'interface utilisateur.

C'est dans ce sens que nous avons choisi d'utiliser les trois services (annulation, notification et prévention des erreurs) du noyau fonctionnel indispensables à l'IHM définis par [Fek96a]. Une grande partie de la complexité des systèmes interactifs vient de la communication avec le noyau fonctionnel. Le concepteur d'une IHM doit réussir à extraire du noyau toutes les informations dont il a besoin, parfois avec une grande difficulté. [Fek96a] justifie aussi le fait que si ces services ne sont pas disponibles dans le noyau fonctionnel, ils sont impossibles à émuler convenablement.

Toujours selon [Fek96a], la modélisation du noyau fonctionnel doit offrir les trois services suivant :

- **la prévention des erreurs :** la possibilité de savoir si un appel d'une opération est licite dans un contexte donné ;
- **l'annulation :** la possibilité de revenir à des états précédents du noyau fonctionnel ;
- **la notification :** la possibilité pour un module externe (adaptateur de présentation par exemple) d'être prévenu lorsque l'état du noyau fonctionnel est modifié.

Examinons pour chaque service si à partir d'un noyau fonctionnel développé avec le langage B ces services sont directement implémentables. Dans le cas positif, nous étudierons la solution de l'implémentation et dans le cas contraire nous proposerons une solution alternative.

Prévention des erreurs. La syntaxe du langage B permet de proposer une solution. Parmi les informations disponibles dans les spécifications du noyau fonctionnel, les préconditions sont particulièrement intéressantes. Elles permettent d'implémenter directement la prévention d'erreurs, en fournissant au module qui interroge le noyau la possibilité d'interdire des appels aux opérations si les conditions exprimées ne sont pas remplies (programmation défensive). Ceci permet de garantir par exemple que les actions de l'utilisateur ne pourront déclencher d'erreur et plus précisément que l'appel des opérations respecte les contraintes imposées par le noyau fonctionnel. Il suffit d'être capable de tester à priori les préconditions.

Annulation. Ce service a été implémenté directement dans le noyau fonctionnel. Nous avons tout d'abord défini un outillage qui permet de stocker, à chaque changement d'état du noyau fonctionnel, l'intégralité des états des variables. Nous avons utilisé pour cela

une fonction qui du point de vue implémentation correspond à un tableau. Nous y avons ajouté aussi une valeur entière appelée *nbrAnnulation* qui permet de stocker le nombre de modifications du noyau qui ont été effectuées depuis le début de l'exécution. Chaque modification de l'état du noyau incrémente *nbrAnnulation* et stocke dans la fonction, à cet indice, l'ensemble des variables. Le service d'annulation est complété par l'ajout d'une opération particulière appelée *opAnnulation* dans la machine abstraite qui lorsqu'elle est appelée, modifie l'intégralité des états courants des variables par celles contenues dans la fonction de sauvegarde à l'indice *nbrAnnulation*.

Par ailleurs, pour conserver la propriété d'honnêteté de l'interface, il convient que l'opération *opAnnulation* vérifie l'invariant de la machine. Cette vérification est assurée par des préconditions de cette opération. Cette solution semble satisfaisante mais il serait plus efficace de pouvoir connaître les seules variables modifiées, pour ne stocker que ces variables. Cette solution imposerait une description supplémentaire dans le noyau fonctionnel.

Notification. Pour le service de notification, la construction du noyau fonctionnel est réalisée de telle façon que ce noyau ne connaît que lui même. Il ne peut y avoir de communication entre le noyau fonctionnel et l'interface du système puisque le premier ne connaît pas le second. À ce niveau de modélisation, nous ne pouvons donc pas décrire le mécanisme de notification. Nous reviendrons sur ce service dans la section 3.3.2.2.

3.3.1.3 API du noyau fonctionnel

L'API du noyau fonctionnel fournit une description qui permet à l'outil Générateur (figure 3.5) d'extraire les spécifications du noyau fonctionnel. Nous étudions ici les informations et les restrictions nécessaires à la préparation de cette extraction. En fait, il s'agit d'une sorte de guide de développement que le concepteur doit assurer au moment de la conception du noyau sémantique.

Le concepteur doit tout d'abord enrichir la modélisation du noyau fonctionnel en intégrant trois caractéristiques. La première concerne l'initialisation du noyau décrite par une opération. La deuxième caractéristique traite du service d'annulation et devra modéliser l'outillage que nous avons décrit précédemment. La dernière concerne l'accès aux variables d'état par les modules extérieurs. Ces variables d'états sont nécessaires à l'évaluation des préconditions des opérations afin de garantir que l'appel à une opération est correct. Pour chaque variables, nous devons donc ajouter une opération qui retourne la valeur de cette variable.

Lors du développement du noyau fonctionnel, l'API est fournie dans le dernier raffinement avant la phase d'implémentation. Cependant, cette API peut se trouver dans le niveau de la machine abstraite si par exemple le noyau fonctionnel n'est pas complexe et le seul raffinement correspond à l'implémentation. Au niveau le plus concret, nous avons choisi de définir des opérations déterministes qui sont logiquement décrites par les deux substitutions généralisées de la forme : $PRE\ P\ THEN\ S\ END$ (substitution précondition) et $BEGIN\ S\ END$ (substitution bloc).

OPERATIONS
$\backslash * \ Trois\ caractristiques\ du\ noyau\ fonctionnel\ * \backslash$

$\backslash * \ Une\ opration\ d'initialisation\ sur\ l'ensemble\ des\ variables\ * \backslash$
$OpInitialisation = \mathbf{PRE}\ P(var_1,\ \ldots,\ var_i)\ \mathbf{THEN}\ S_{Initialisation}\ \mathbf{END}\ ;$

$\backslash * \ Une\ opration\ d'annulation\ * \backslash$
$OpAnnulation = \mathbf{PRE}\ P(var_1,\ \ldots,\ var_i)\ \mathbf{THEN}\ S_{Annulation}\ \mathbf{END}\ ;$

$\backslash * \ Des\ accesseurs\ sur\ les\ varibles\ du\ noyau\ fonctionnel\ * \backslash$
$v\_variable_1 <-- getVariable_1 = \mathbf{BEGIN}\ S_1\ \mathbf{END}\ ;$
$v\_variable_i <-- getVariable_i = \mathbf{BEGIN}\ S_i\ \mathbf{END}\ ;$
$v\_nbrAnnulation <-- getNbrAnnulation = \mathbf{BEGIN}\ S_{Annulation}\ \mathbf{END}\ ;$

$\backslash * \ Le\ corps\ du\ noyau\ fonctionnel\ * \backslash$

$\backslash * \ Accesseurs\ * \backslash$
$v\_valeur_1 <-- getOp_1 = \mathbf{PRE}\ P(var_1,\ \ldots,\ var_i)\ \mathbf{THEN}\ S_{Op1}\ \mathbf{END}\ ;$
$v\_valeur_i <-- getOp_i = \mathbf{BEGIN}\ S_{Opi}\ \mathbf{END}\ ;$

$\backslash * \ Modifieurs\ * \backslash$
$setMod_1 = \mathbf{PRE}\ P(var_1,\ \ldots,\ var_i)\ \mathbf{THEN}\ S_{Mod1}\ \mathbf{END}\ ;$
$setMod_i = \mathbf{PRE}\ P(var_1,\ \ldots,\ var_i)\ \mathbf{THEN}\ S_{Modi}\ \mathbf{END.}$

...

Tab. 3.1 – Noyau fonctionnel générique B.

L'ensemble des opérateurs (décrit précédemment) constitue l'API du noyau fonctionnel. Nous trouvons les opérations *accesseurs* qui retournent une valeur d'une variable mais qui ne modifient pas l'état du noyau, puis il y a les opérations *modifieurs* qui modifient l'état du noyau fonctionnel. Si ces opérations possèdent des paramètres, l'écriture d'une opération est donnée par la substitution généralisées $PRE\ P\ THEN\ S\ END$ où P décrit

à la fois un prédicat de typage et un prédicat de propriétés. Dans le cas contraire elles utilisent l'écriture suivante $BEGIN\ S\ END$. En résumé, nous avons représenté dans un sous ensemble de B, la partie interface du noyau fonctionnel (API).

A titre d'exemple, nous montrons sur le tableau 3.1 l'API d'un noyau fonctionnel générique B.

$P(var_1, \ldots, var_i)$ désigne une précondition sur les variables d'état du noyau fonctionnel. Dans le cas des accesseurs, la présence d'une précondition n'est pas obligatoire. Nous devons également donner les substitutions généralisées des accesseurs ($S_{Annulation}$, S_1, etc). En effet, l'affectation de la valeur de retour (par conséquent son type) est définie dans cette substitution. Il faut donc pouvoir l'analyser.

3.3.1.4 Génération de code : Atelier B

L'Atelier B génère à partir de la modélisation du noyau fonctionnel du code exécutable C^{++}. La génération n'est possible que si toutes obligations de preuve de développement sont déchargées. Nous avons choisi de générer du code C^{++} car la version de l'Atelier B dont nous disposions ne permettait pas de générer du code Java. Cependant afin de proposer une homogénéité des langages avec celui utilisé pour développé SUIDT, nous avons transformé le code C^{++} en code Java. Cette traduction a été facilitée puisque le code C^{++} obtenu est indépendant de toute perspective d'interaction et, à ce niveau, la syntaxe des langages Java et C^{++} reste très proche.

Nous obtenons une classe Java où sont décrites d'une part les variables au travers des attributs et les opérations par l'intermédiaire des méthodes de la classe. Nous donnons ci-dessous, un extrait du noyau fonctionnel de l'étude de cas du convertisseur/compteur qui correspond au noyau fonctionnel développé en B décrit au chapitre 2.

Les variables du noyau fonctionnel sont définies par des types simples. Nous décrivons ici, la variable du compteur, les valeurs pour la conversion et l'état de conversion (en euros ou en francs).

```
public class NFConvertisseurCompteur {
 // Attributs qui d'écrivent les variables du noyau fonctionnel.
 private boolean etat_conversion;
 private double valeur_convertie;
 private double valeur_a_convertir;
 private int compteur;
 ...
```

Les opérations sont décrites quant à elles par des méthodes. Par exemple la méthode *convertValue* permet de calculer une somme en francs ou en euros selon la valeur de la variable *valeur_a_convertir*, puis finalement elle incrémente la valeur du compteur. Il est a noter qu'il n'y a ici aucune vérification sur le fait que le prédicat $compteur < 3$ doit être vérifié avant l'exécution de la méthode. Cette contrainte avait été exprimée dans la précondition de l'opération *convertValue* de la spécification B du noyau fonctionnel. La précondition est vue en B comme un prédicat qui n'a pas besoin d'être respecté pour appeler une opération. Au contraire, la garde est un prédicat qui doit être vrai pour qu'un événement par exemple soit émis.

```
// Méthodes qui décrivent les opérations du noyau fonctionnel.
public void convertValue() {
 if (etat_conversion) {
  valeur_convertie = valeur_a_convertir/taux_e_f;
 } else {
  valeur_convertie = valeur_a_convertir/taux_f_e;
 } compteur++;
}
```

La prise en compte des contraintes édictées par les spécifications du noyau fonctionnel est à la charge de l'adaptateur du noyau fonctionnel développé formellement.

3.3.2 Adaptateur du noyau fonctionnel développé formellement

Nous nous appuyons sur le modèle d'architecture ARCH pour la conception de l'adaptateur du noyau fonctionnel développé formellement, figure 3.6. A la différence de la conception du noyau fonctionnel, un concepteur d'IHM est en mesure de construire ce module par l'intermédiaire des outils Générateur et/ou Animateur.

Nous présentons dans cette section les outils qui permettent d'une part, d'extraire les informations du noyau fonctionnel via son API et d'autre part, fournir une interface interactive qui permet d'utiliser ce noyau.

3.3.2.1 Génération d'interfaçage : Générateur

Le générateur se charge d'extraire les informations de la modélisation du noyau fonctionnel décrit avec B via son API afin de réaliser l'interfaçage du noyau fonctionnel avec

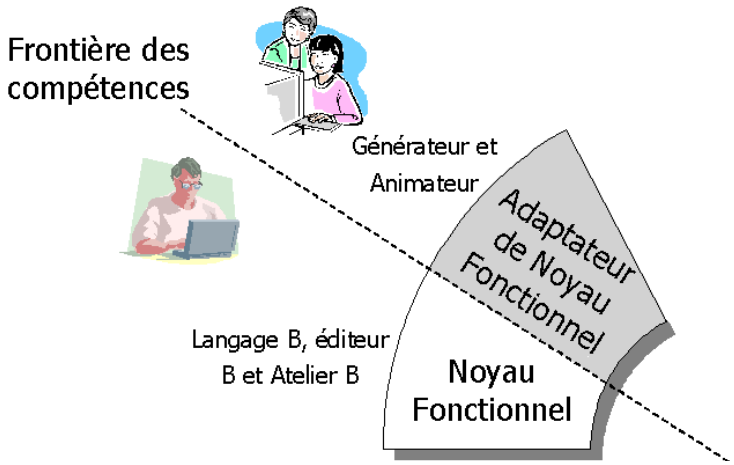

Fig. 3.6 – Conception du module Adaptateur de Noyau Fonctionnel de l'architecture ARCH.

l'environnement de développement interactif. Cet outil ne nécessite pas de connaissance réelle dans la pratique des techniques formelles.

En se basant sur l'API, nous pouvons extraire toutes les signatures des opérations, comprenant le nom, les paramètres d'entrée, le paramètre de sortie et finalement les préconditions qui décrivent les contraintes d'appels. Chaque opération est décrite dans une classe Java qui hérite soit de la classe abstraite *AbstractFunction* soit de *AbstractVariable* également définies dans l'adaptateur du noyau fonctionnel. Dans le cas où l'opération est utilisée pour connaître la valeur d'une variable, c'est la classe abstraite *AbstractVariable* qui est héritée, inversement il s'agit de la classe abstraite *AbstractFunction*. Dans la suite, nous présentons des implémentations de ces deux classes (*AbstractVariable* et *AbstractFunction*.

Nous montrons ci-dessous la classe *FCConvertValue* qui interface l'opération *convertValue* du noyau fonctionnel. Cette opération est appelée *convertValue*, ne possède pas de paramètre d'entrée ni de paramètre de sortie mais une précondition sur la valeur du compteur.

```
public class FCConvertValue extends AbstractFunction {
 public FCConvertValue(int p_ind, NFConvertisseurCompteur p_nf) {
  operation_name = "convertValue";
  output_parameter = new OutputParameter(null,"","");
  input_parameter = newInputParameter(null,null,null);
  propertie_predicat = "v_compteur_value < 3";
  ...
 }
```

A cela s'ajoutent deux méthodes : *checkPreCondition* évalue le prédicat de la précondition pour savoir si l'invariant et la précondition sont respectés. Nous utilisons pour cela, les accesseurs des variables, ici *getVCompteurValue*, avec lesquels nous exprimons le prédicat de précondition. La seconde méthode *executeOperation* appelle la méthode *convertValue* si la précondition de l'opération soit respectée.

```
public boolean executeOperation() {
 if (this.checkPreCondition()) {
  inst_nf.convertValue();
  return true;
 } else return false;
}
public boolean checkPreCondition() {
 return(inst_nf.getVCompteurValue() < 3);
}
```

La valeur de retour de la méthode *convertValue* permet de signaler l'appel correct ou non de l'opération du noyau fonctionnel.

3.3.2.2 L'animateur de noyau fonctionnel

Les précédentes sections ont montré l'élaboration d'un modèle de description d'un noyau fonctionnel développé formellement. Ce modèle fournit des services (annulation, prévention d'erreurs, signatures, etc.) pour exécuter des opérations du noyau fonctionnel tout en respectant les contraintes de sûreté imposées par ce noyau. Toutefois, l'utilisation de ce modèle dans sa forme actuelle (des classes Java) n'est pas à la portée de concepteurs d'IHM puisqu'ils leur incombent de connaître le langage Java, la compilation, etc.

Suivant les résultats encourageants obtenus par l'outil GenBUILD et de son animateur de noyau fonctionnel, nous avons choisi d'utiliser cet outil dans cette optique. Nous proposons donc un outil similaire, toujours appelé Animateur, qui permet de générer également une interface interactive pour un noyau fonctionnel développé formellement. Pour un modèle donné, il s'agit de fournir une interface permettant d'interagir avec chacune des opérations du noyau fonctionnel et de représenter l'état de ce noyau.

Nous présentons sur la figure 3.7 des captures d'écran de l'outil Animateur. Chaque classe d'interfaçage définie dans le modèle est associée à un groupe de composants graphiques. L'exécution de l'opération est associée à un bouton (repère 2, figure 3.7). Le nom de l'opération (repère 3, figure 3.7), les paramètres d'entrée (repère 4, figure 3.7), la valeur

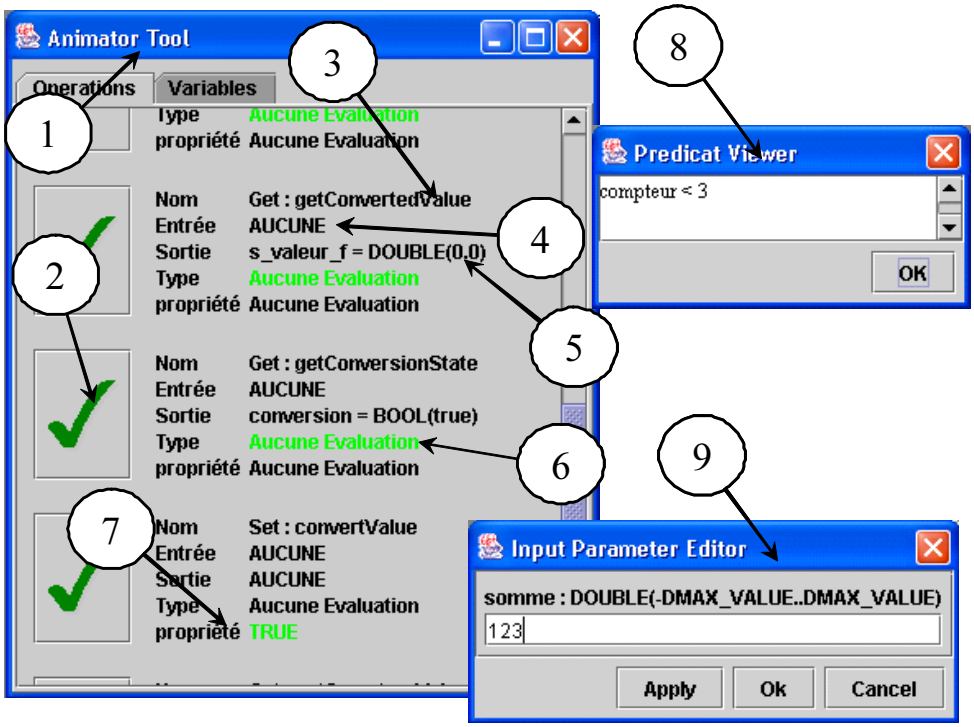

Fig. 3.7 – Capture d'écran de l'outil Animateur de noyau fonctionnel formel.

de retour (repère 5, figure 3.7), la cohérence des types des paramètres d'entrée (repère 6, figure 3.7) et la précondition de l'opération (repère 7, figure 3.7) sont visualisés dans des champs de textes. Toutes les opérations sont visualisées suivant la même hiérarchie de composants et sont regroupées dans une fenêtre (repère 1, figure 3.7). L'utilisateur peut visualiser les préconditions (repère 8, figure 3.7) et saisir les paramètres d'entrées (repère 9, figure 3.7).

A chaque appel d'une opération *modifieur*, l'état du noyau fonctionnel est modifié et l'outil Animateur doit être averti de ces changements. Nous avons donc implémenté le service de notification de l'outil Animateur par l'interrogation de toutes les opérations accesseurs à chaque modification ce qui implique que la vue de l'interface retourne instantanément l'état du noyau fonctionnel.

Les avantages de l'outil Animateur sont divers :

- fournir une plate-forme qui permet de valider et de vérifier par des tests la conformité du noyau fonctionnel. Cet outil agit donc comme un complément à la validation du noyau fonctionnel par la technique formelle B et l'Atelier B. Si par contre les tests ne sont pas concluants la modélisation du noyau fonctionnel peut être revue ;
- fournir un outil d'aide à la compréhension de techniques formelles pour le concep-

teur d'IHM. Ce concepteur manipule directement un noyau fonctionnel développé formellement sans connaissance préalable du langage B. Il se familiarise avec les concepts de contraintes du noyau fonctionnel développé formellement, précondition, invariant, etc.

3.3.3 Bilan sur l'intégration d'un noyau fonctionnel développé formellement

L'intégration d'un noyau fonctionnel développé formellement dans une approche de conception d'IHM présente des avantages que d'autres approches, étudiées dans la section 1.4, ne possèdent pas. Nos travaux se situent principalement dans une approche descendante dans le sens où l'on part d'une spécification (en l'occurrence celle du noyau fonctionnel) pour en générer une description.

Le premier aspect que nous jugeons intéressant concerne la prise en compte d'une sémantique formelle pour la description du noyau fonctionnel. A notre connaissance seule l'approche de Petshop associée au formalisme des ICO se rapproche de cette idée. Rappelons que la description d'une partie du noyau fonctionnel et de celle de l'adaptateur du noyau fonctionnel est obtenue par un ensemble de CO-classes (les signatures) et utilise les réseaux de Petri à objets pour décrire leurs comportements. Cependant, la procédure de vérifications employée est la technique de vérification sur modèles (*model checking*) qui ne peut être efficace lorsqu'il y a des modifications apportées à la description de la spécification. Les propriétés devront être une nouvelle fois re-vérifiées. À l'inverse notre approche s'appuie sur un système de preuve par démonstration de théorèmes pour vérifier les propriétés du noyau fonctionnel. Il permet grace au raffinement d'enrichir la spécification tout en conservant les propriétés exprimées et prouvées dans l'abstraction. Le code généré est alors considéré comme sûr sous condition de respecter les contraintes édictées dans la spécification.

Le deuxième aspect concerne la prise en compte de services dans le noyau fonctionnel. Ce point a été partiellement pris en compte dans l'outil Mastermind et Petshop au travers du modèle à objets CORBA. Les deux outils décrivent le service de prévention d'erreur et de notification par l'intermédiaire de préconditions et de mécanisme basé sur les exceptions. L'originalité de notre approche est de pouvoir prendre en compte tous les services et notamment celui d'annulation.

Le troisième aspect s'intéresse aux corps de métiers qui participent à la conception de l'application. Nous identifions, l'expert qui spécifie formellement le noyau fonctionnel, du concepteur d'IHM qui exploite cette description pour générer l'adaptateur du noyau

fonctionnel. Ce non spécialiste en programmation peut facilement comprendre les usages d'une méthode formelle sans avoir à apprendre le langage formel lui-même. En comparaison avec Petshop, ce dernier n'identifie pas les compétences du concepteur. Même si Petshop propose des outillages interactifs pour faciliter l'édition de réseaux de Petri, il ne s'abstrait pas du langage ni de la sémantique formelle.

Enfin, un dernier aspect concerne la génération (semi-)automatique d'interfaces à partir du noyau fonctionnel. Notre approche peut être comparée aux approches fondées sur la génération et plus précisément les SGIU[4] ou les UIMS[5]. Les outils Mike [Ols86] et Mickey [Ols89] ont été les premiers à s'intéresser à la démarche de génération (semi-)automatique de l'IHM à partir du noyau fonctionnel. La génération s'effectue à partir de la signature des fonctions du noyau fonctionnel écrites en Pascal. Nous pouvons également citer l'outil PDGen [EFF96] qui génère un programme complet possédant une interface graphique afin d'éditer des données scientifiques à partir d'un fichier de description. Se basant sur une analyse du code C^{++}, et plus particulièrement de la définition des classes, PDGen génère une interface permettant d'accéder directement aux données, de naviguer parmi elles et de les modifier, le seul but du programme étant cet accès aux données. La principale différence avec notre approche est que la sémantique de l'application est figée. PDGen ne traite pas les fonctions membres des classes et ne sait opérer que des actions limitées sur les données.

3.4 Validation sur le noyau fonctionnel

De nombreuses méthodes de développement d'applications interactives préconisent d'effectuer au préalable une analyse de l'activité [SB01] pour construire un modèle de tâches. En premier, cette analyse doit être indépendante de toute idée d'interface et ne doit pas être composée d'insinuations sur les systèmes d'interaction à implémenter. C'est une vue abstraite du fonctionnement de l'application sans élément de l'interface. Il nous a semblé naturel d'introduire la notion de modèle de tâches dans notre approche à la manière des approches descendantes, afin de permettre de décrire au mieux le système interactif du point de vue de l'utilisateur.

Notre choix du formalisme de tâches s'est porté sur la notation ConcurTaskTrees car nous souhaitons disposer d'une notation graphique, à sémantique définie et si possible disposant d'outils d'analyse (en l'occurrence CTTE) se connectant à notre approche. De plus nous souhaitons ré-utiliser la même notation que celle utilisée dans le chapitre 2.

[4]Système de Gestion d'Interfaces Utilisateur
[5]User Interface Management Systems

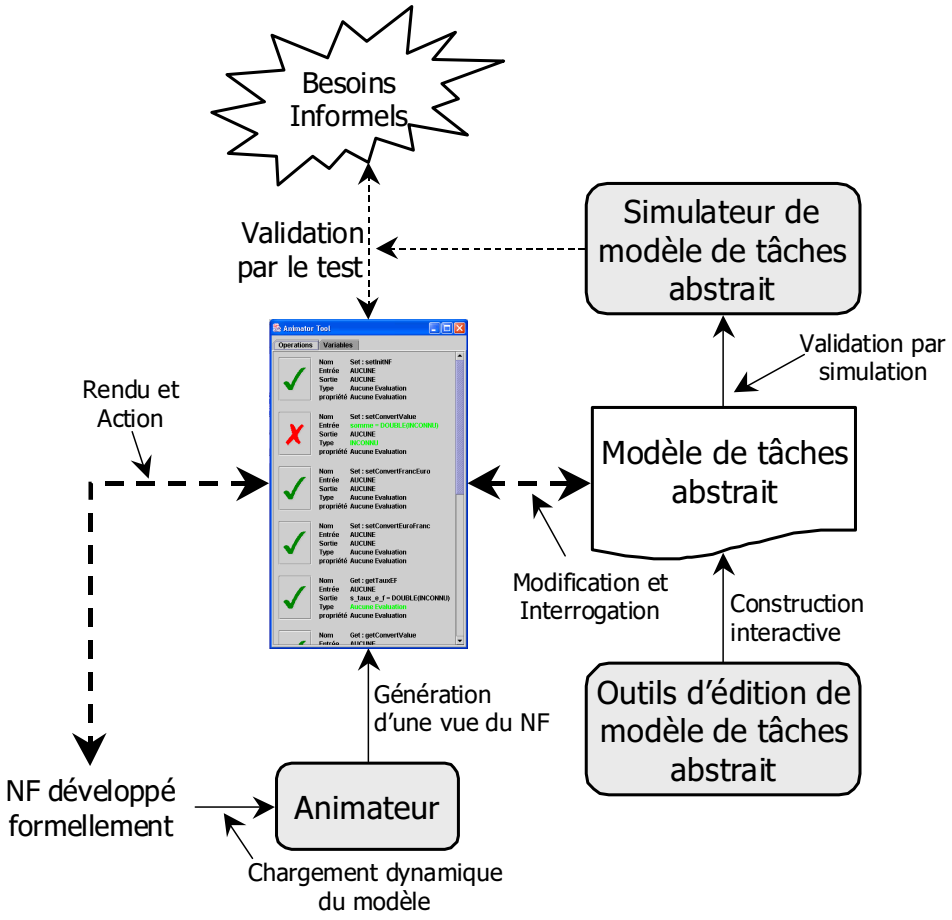

Fig. 3.8 – Démarche pour la modélisation du modèle de tâches abstrait.

La notation CTT est adaptée à la description de tâches afin d'exprimer les besoins des utilisateurs, souvent des non-informaticiens. Nous avons défini à partir de cette notation, un modèle qui se situe au niveau de la spécification abstraite d'une approche descendante, appelé **modèle de tâches abstrait** où aucun choix de technique d'interaction n'est autorisé. C'est à partir du modèle de tâches abstrait et dans le respect des propriétés du noyau fonctionnel développé formellement, que le concepteur d'application interactive sera en mesure de construire son système. Plus précisément, le modèle de tâches abstrait fournit une base pour la conception d'un dialogue abstrait notamment via les contraintes de précédence.

L'approche de description du modèle de tâches abstrait est présentée sur la figure 3.8. Sur la partie de gauche, nous retrouvons l'animateur de noyau fonctionnel qui joue le rôle central de la conception. Les outils d'édition du modèle de tâches abstrait permettent de construire hiérarchiquement un arbre de tâches abstrait et associent aussi les objets du noyau fonctionnel formel aux tâches tout en respectant ses contraintes. Les objets du

noyau fonctionnel sont disponibles via l'animateur de noyau fonctionnel. La validation du modèle de tâches abstrait est effectuée au moyen du simulateur. Cet outil anime le dialogue abstrait en déclenchant les tâches à la manière de la simulation de CTTE. Mais en plus de vérifier l'ordonnancement des tâches, le concepteur modifie et interroge aussi le noyau fonctionnel (double flèche *Modification et Interrogation* entre le modèle de tâches abstrait et l'animateur). Par ailleurs, toute modification apportée pendant le déroulement du modèle de tâches abstrait influence l'interface utilisateur de l'animateur (*Rendu et Action*).

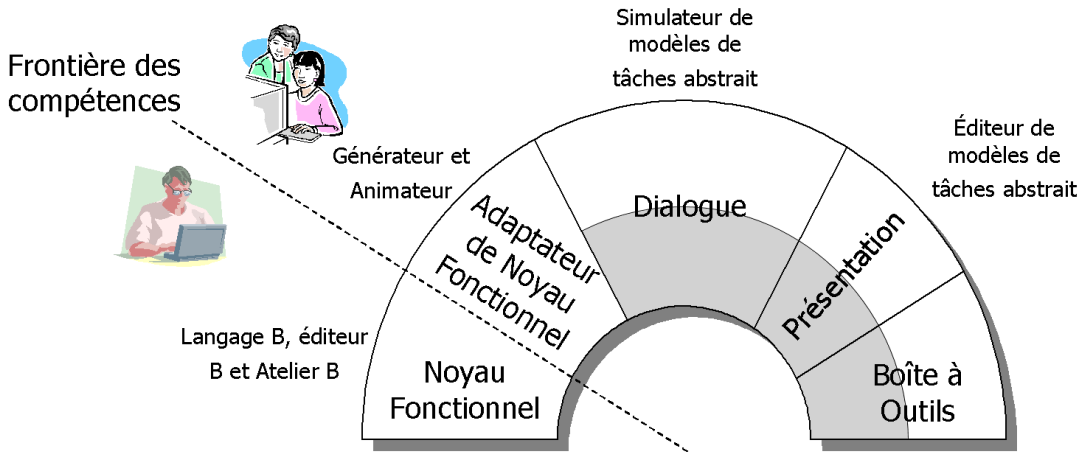

Fig. 3.9 – Conception « abstraite » du module Dialogue, Présentation et Boîte à outils de l'architecture ARCH.

La conception du modèle de tâches abstrait peut être vue comme une conception « abstraite » des modules dialogue, présentation et boîte à outils de l'architecture ARCH, illustrée par la figure 3.9. La partie grisée des modules dialogue, présentation et boîte à outils met en avant le fait que la conception « abstraite » n'exprime pas complètement le développement de ces modules et que seuls les objets du noyau fonctionnel développé formellement sont utilisés.

Nous proposons le plan suivant : une première partie est consacrée à la présentation de la sémantique utilisée pour définir la spécification abstraite de notre approche. Nous étudions plus particulièrement la relation entre le noyau fonctionnel formel et le modèle de tâches abstrait. Dans une deuxième, troisième et quatrième partie, nous discutons les outils proposés en mettant l'accent sur l'outil de simulation. Finalement, dans la conclusion partielle, nous confrontons notre démarche à des approches similaires.

3.4.1 Description du modèle de tâches abstrait

La modélisation de la spécification abstraite débute par une intégration d'un modèle de tâches CTT établi au préalable avec l'aide d'un ergonome. Du point de vue de la forme, le modèle de tâches abstrait s'identifie (décomposition hiérarchique des tâches, même nombre de tâches, catégorie de tâches identique, etc.) à celui du modèle de tâches CTT.

Toutefois du point de vue du fond, nous avons utilisé CTT en le restreignant (réduction du nombre d'opérateurs). Nous avons également proposé une extension de son pouvoir d'expression pour améliorer la gestion des objets qui constitue une faiblesse de cette notation actuellement. En effet, dans CTT les descriptions des objets des tâches sont réalisées de façon informelle. Ceci interdit tout raisonnement sur ces objets et ne permet de vérifier que l'atteignabilité d'une tâche par rapport aux contraintes de précédence des opérateurs temporels.

Au niveau de la spécification abstraite, nous considérons les objets relatifs à ceux du noyau fonctionnel formel existant, c'est-à-dire les opérations de type accesseur et modifieur. A cela, s'ajoutent toutes les descriptions que nous avons extraites du noyau fonctionnel développé formellement. La prise en compte d'objets ayant une existence propre et où des raisonnements peuvent être effectués révèle de nouveaux problèmes auxquels nous proposons de répondre dans cette partie.

Nous discutons tout d'abord des préconditions de tâches, puis des actions, des restrictions sur les opérateurs et enfin de l'établissement d'un dialogue abstrait et de sa présentation abstraite au moyen du modèle de tâches CTT.

3.4.1.1 Les préconditions des tâches

Dans la sémantique de CTT, les préconditions des tâches ne concernent que les contraintes de précédence définies par les opérateurs temporels et les caractéristiques de tâches (les objets sont décrits de façon informelle, le raisonnement est donc impossible). Prenons le cas de la tâche $T_0 ::= T_1 >> T_2$. La précondition de la tâche T_2 exprime le fait que si T_1 est activée alors T_2 est activable. Cependant, pour que des raisonnements soit effectués sur les objets de l'application, il faut pouvoir intégrer l'expression de propriétés dans la description des objets (preconditions, invariants, gardes, etc).

Rappelons que nous avions traité ce problème dans le chapitre précédent en intégrant à chaque tâche un prédicat décrit sur des variables de l'architecture et sur des variants liés aux opérateurs temporels. Nous proposons une approche similaire dans le sens où

nous enrichissons la notion de précondition par l'intégration d'un prédicat établi avec les objets du noyau fonctionnel. Cette approche est un peu similaire à ce que MAD [SPG89] (voir section 1.2.3) propose, en permettant de définir à la fois des préconditions sur les relations de précédence et sur les objets.

À la définition d'une tâche, nous ajoutons donc, un attribut de type booléen qui désigne la précondition sur les objets. Si cet attribut est vrai, la précondition (garde) est respectée sinon elle ne l'est pas. Cet attribut booléen peut combiner des accesseurs du noyau fonctionnel, des fonctions diverses (fonction qui retourne par exemple la longueur d'une chaîne de caractères), des opérateurs de comparaison ($==, \leq$, etc), des variables, des constantes, etc. Par exemple, pour la tâche *conversion franc/euro* de l'exemple du convertisseur francs/euros et compteur, la précondition (garde) est définie par $compteur < 3$. Du point de vue de la construction de ce prédicat, *compteur* désigne une opération accesseur du noyau fonctionnel, $<$ un opérateur de comparaison et 3 une constante de type entier. Toutes les catégories de tâches CTT sont concernées, à l'exception de la tâche utilisateur qui concerne plus particulièrement des actions sur l'utilisateur (réflexion cognitive par exemple) qu'une action sur le système. Sa précondition est donc toujours vraie.

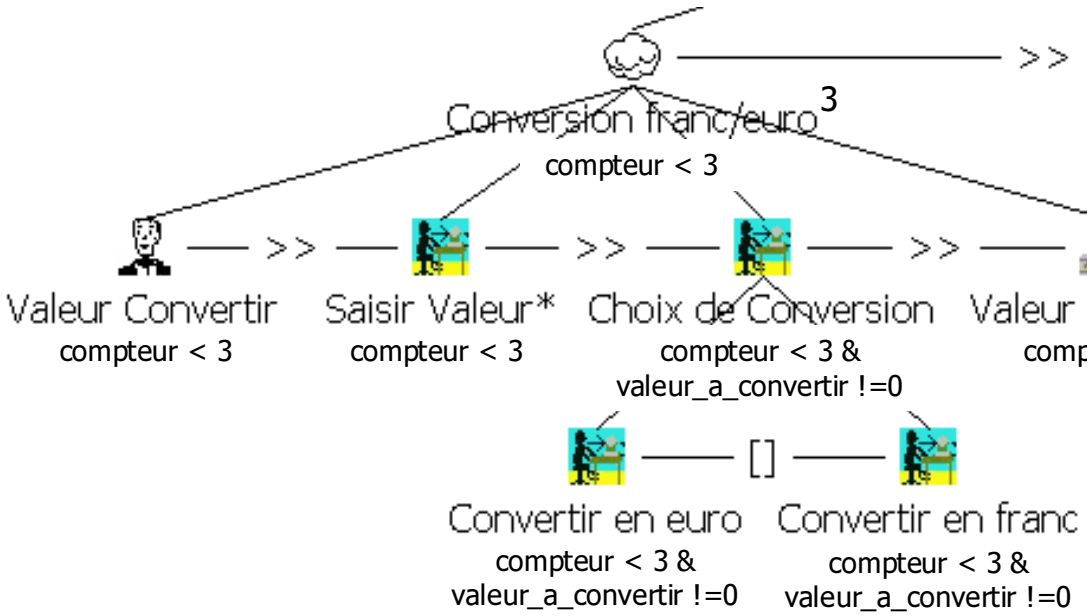

Fig. 3.10 – Exemple de préconditions (gardes) de tâches du modèle de tâches abstrait.

Du point de vue de la décomposition hiérarchique du modèle de tâches abstrait, lors des raffinements successifs, la précondition (garde) de chaque tâche fille correspond à la conjonction de toutes les préconditions (gardes) sur les objets des tâches mères. Nous illustrons ce propos sur la figure 3.10. Par exemple, la précondition (garde) de la tâche *Convertir en Euro*, est obtenue par la conjonction des préconditions (gardes) des tâches mères *Choix de Conversion*, *Conversion franc/euro*, etc. Enfin, même si la précondition

(garde) de la tâche utilisateur *ValeurConvertir* n'est pas modifiable, elle est quand même sujette à la conjonction des préconditions (gardes) afin de respecter la décomposition hiérarchique des tâches.

3.4.1.2 Les actions

Lorsqu'une précondition (garde) est satisfaite (contrainte de précédence et prédicat sur les objets respectés), l'action associée à la tâche modifie l'état de l'application.

Nous emploierons dans la suite de ce chapitre indifféremment corps de tâche ou post-condition pour désigner une action de tâche. Il peut s'agir d'une modification de l'état du noyau fonctionnel (appel d'opérations modifieurs) mais aussi d'un rendu de la présentation abstraite (affichage d'une valeur quelconque).

Toutefois, selon la catégorie de la tâche exécutée (abstraite, application, interaction ou utilisateur), la nature du traitement a effectuer est différente :

- tâche **abstraite** et tâche **utilisateur** : dans les deux cas, ces catégories de tâches ne peuvent modifier l'état de l'application. La catégorie abstraite est une tâche de plus haut niveau sans connaître exactement la portée de son action. A l'inverse la tâche utilisateur est plutôt une action mentale qu'une action dirigée vers la modification de l'application ;
- tâche **application** : l'action d'une tâche application modifie l'état de la présentation abstraite et peut aussi modifier l'état du noyau fonctionnel. La présentation abstraite se limite ici à l'affichage de messages envoyés par les tâches application. Toutefois, une tâche application est limitée à l'envoi d'un seul message. Suivant le principe de la précondition sur les objets, un message est défini par une expression de type chaîne de caractères. Il combine aussi des accesseurs du noyau fonctionnel, des fonctions diverses, etc. Par exemple, pour la tâche *Valeur Convertie*, son action émet un message qui retourne la valeur de la variable *valeur_convertie* (par un accesseur du noyau fonctionnel) ;
- tâche **interaction** : l'action d'une tâche interaction modifie l'état du noyau fonctionnel et permet dans certains cas de saisir des paramètres utilisateur au moyen de la présentation abstraite. Toujours par rapport à notre exemple, la tâche *Saisir Valeur* permet de choisir la valeur qui sera convertie. Nous associons à l'action de cette tâche l'opération modifieur *setConvertValue*. Elle modifie la valeur correspondant à la somme à convertir et possède, un paramètre de type *double*.

Du point de vue de la décomposition hiérarchique des tâches, seules les tâches élémen-

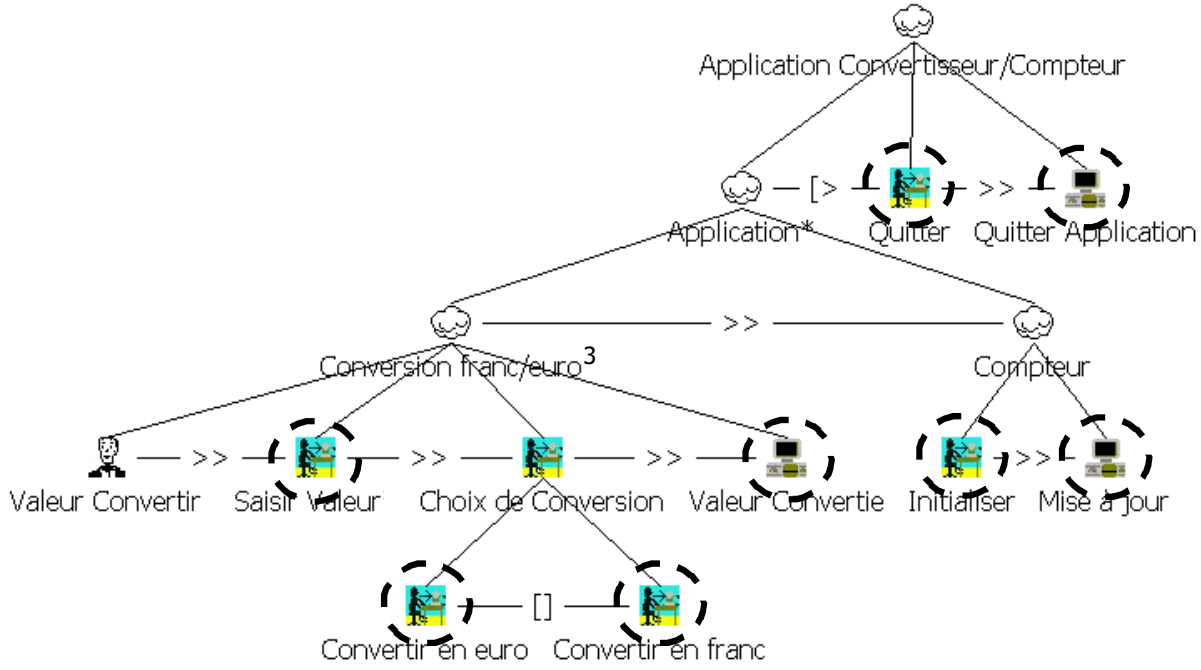

Fig. 3.11 – Modèle de tâches abstrait du convertisseur/compteur.

taires ou feuilles sont susceptibles d'effectuer une action sur l'application. Nous montrons sur la figure 3.11 le modèle de tâches abstrait de notre étude de cas. Les tâches encerclées ciblent celles où la notion d'action de la tâche est autorisée. La tâche utilisateur *Valeur Convertir* ne peut affecter l'état de l'application, il s'agit d'une action cognitive (réfléchir sur la valeur à convertir).

3.4.1.3 Le dialogue abstrait

Le dialogue abstrait joue le rôle de médiateur entre la présentation abstraite (messages et interfaces de saisie des paramètres utilisateur) et l'adaptateur du noyau fonctionnel. C'est le dialogue qui est à même d'autoriser et contrôler l'enchaînement des interactions de l'utilisateur (saisir des paramètres), et qui déclenche les appels des fonctionnalités de l'interface (affichage des messages) avec l'application. Il définit donc le comportement de la présentation abstraite.

Le dialogue abstrait se base sur la dynamique du modèle de tâches abstrait et plus précisément sur les contraintes de précédence et sur les préconditions sur les objets. Nous rapprochons cela avec un formalisme à base d'états comme les automates par exemple. Les tâches du modèle de tâches abstrait sont vues comme l'ensemble des états, les transitions

étant représentées par les opérations et par les préconditions des tâches ; cela établit le comportement du dialogue abstrait. L'avantage de baser la dynamique du dialogue abstrait sur celle du modèle de tâches abstrait est de pouvoir garantir avec exactitude que ce passage est conforme, aux relations temporelles et structurelles du modèle de tâches CTT.

L'avantage du formalisme CTT est d'avoir une forte capacité à structurer. Cette caractéristique est offerte par la richesse des opérateurs. Elle permet donc d'exprimer un dialogue abstrait suffisant pour jouer le rôle de médiateur entre l'adaptateur de noyau fonctionnel développé formellement et la présentation abstraite simplifiée à l'affichage de message et à la saisie de paramètres utilisateur.

Toutefois, il est à noter que l'utilisation d'une notation de description de tâches utilisateur ne peut prétendre se substituer aux formalismes de description du dialogue (voir les travaux de [Jam96]) que nous avions décrits dans la section 1.2.5. Cette solution nous semble parfaitement adaptée au niveau de la spécification abstraite pour effectuer des vérifications sur les contraintes du noyau fonctionnel et pour vérifier aussi l'atteignabilité de tâches utilisateur. Nous reviendrons plus précisément sur ces détails dans la section 3.4.4 quand nous présenterons l'outil de simulation.

3.4.1.4 Restrictions sur les opérateurs temporels

Dans les sections précédentes, nous avons étendu la notion d'objet sur les préconditions (gardes) et actions de tâche. Nous avons aussi montré que la source des objets était d'une part le noyau fonctionnel par le biais de son adaptateur et d'autre part la présentation abstraite.

Ces objets sont donc « visibles » par l'ensemble des tâches ; c'est pourquoi nous avons choisi d'exclure les opérateurs temporels suivants : activation avec passage d'information ([] >>) et synchronisation (|[]|). Ils seront remplacés par les opérateurs activation (>>) et parallèle (||).

3.4.2 Outil d'édition de la structure du modèle de tâches abstrait

Le premier outil que nous présentons est l'éditeur du modèle de tâches abstrait qui permet de construire l'arbre de tâches. Cet outil est comparable à celui de CTTE. Notre éditeur à la faculté d'importer des modèles de tâches CTT relatifs à l'étape d'analyse d'activité dans l'objectif de faciliter la construction du modèle de tâches abstrait. Cependant,

le passage du modèle de tâches CTT en modèle de tâches abstrait transforme le modèle CTT de telle sorte que les restrictions décrites ci-dessus, soient respectées (réduction du nombre d'opérateurs en l'occurrence).

La figure 3.12 fournit une vue de l'éditeur du modèle de tâches abstrait de notre étude de cas. La zone d'édition de l'arbre de tâches est désignée par le repère 1 (figure 3.12). Tandis que CTTE fournit une vue purement graphique de son arbre de tâches, comme illustrée précédemment sur la figure 3.11, nous avons choisi de représenter le modèle de tâches abstrait sous la forme du composant graphique *JTree* de la boîte à outils Swing par défaut. Ce choix a été motivé principalement par des raisons de maîtrise de la taille et de facilité d'utilisation de l'arbre de tâches.

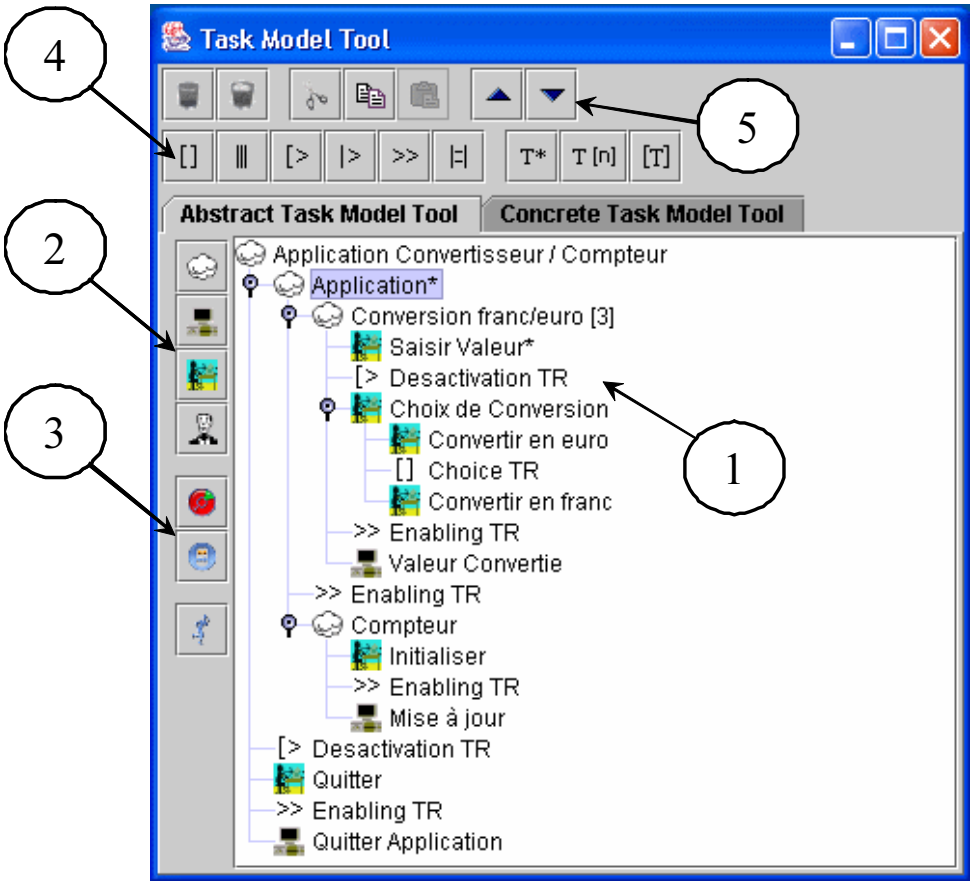

Fig. 3.12 – Capture d'écran de l'éditeur de modèle de tâches abstrait.

Le repère 2 (figure 3.12) désigne les catégories de tâches (abstraite, interaction, utilisateur et application), le repère 3 (figure 3.12), l'accès aux autres outils que nous présentons dans cette section. Le repère 4 (figure 3.12), désigne l'ensemble des opérateurs temporels autorisés dans la spécification abstraite. À la différence de l'environnement CTTE, notre outil d'édition prend en compte l'itération finie.

Enfin, une boite à outils, (repère 5, figure 3.12), facilite le travail d'édition en proposant les outillages classiques d'éditeur (copier/coller, effacer, descendre ou monter une tâche, etc).

3.4.3 Outil de liaison entre les objets du noyau fonctionnel et les tâches du modèle de tâches abstrait

Tandis que l'outil d'édition, présenté précédemment, ne propose qu'une « interface utilisateur » différente de celle de l'outil CTTE, l'outil de liaison apporte une réelle contribution à la manipulation du formalisme CTT. La partie originale de cet outil consiste en la liaison du modèle de tâches abstrait avec les objets d'un noyau fonctionnel développé formellement. Plus précisément, il permet de « construire » les préconditions (gardes) et les actions des tâches du modèle de tâches abstrait. Cette construction pouvant s'apparenter à de la programmation, nous avons équipé cet outil de mécanismes de construction adaptés à un concepteur d'IHM. Dans ce sens, nous avons volontairement caché les aspects propres à la programmation classique (phases d'édition/compilation) que nous avions présentés en partie dans les approches ascendantes.

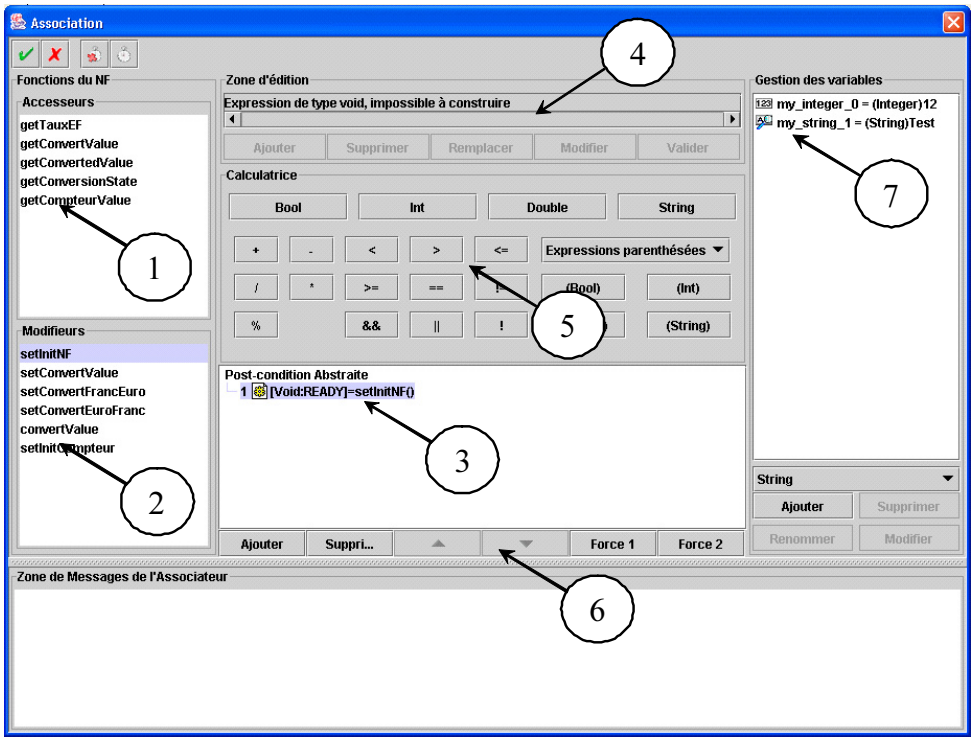

Fig. 3.13 – Capture d'écran de l'outil de liaison entre les objets du noyau fonctionnel et les tâches du modèle de tâches abstrait.

La figure 3.13 représente l'interface utilisateur de cet outil. Les repères 1 et 2 (figure 3.13) désignent respectivement les opérations du noyau fonctionnel développé formellement séparés selon les catégories « accesseurs » et « modifieurs ». Le repère 3 (figure 3.13) désigne un arbre où sont entreposées les expressions de liaison entre les tâches et les opérations du noyau fonctionnel. La construction de l'expression en cours (repère 4, figure 3.13) est réalisée au moyen d'opérateurs (arithmétique, comparaison) et de fonctions diverses fournies par une interface de type calculatrice (repère 5, figure 3.13). Des fonctionnalités (repère 6, figure 3.13) permettent d'ajouter, de supprimer ou d'évaluer directement les expressions. Ce dernier point est intéressant puisqu'il permet au concepteur d'IHM de tester rapidement les résultats de la programmation sans avoir à passer par une phase d'édition/compilation qui ralentirait le processus de conception. Finalement, une liste de variables complémentaires (repère 7, figure 3.13) à celles du noyau fonctionnel permettent de faciliter la construction des expressions par étapes successives (par exemple pour simplifier l'écriture des expressions en les abstrayant par des variables).

Exemple de précondition (garde) sur les objets de la tâche *Conversionfranc/euro* **:** rappelons que le prédicat de cette précondition (garde) exprime le fait que la valeur du compteur doit être inférieure à 3. Pour la construction de la précondition (garde), il ne peut y avoir qu'une seule expression dans la liste qui définit un prédicat (voir repère 1 de la figure 3.14.

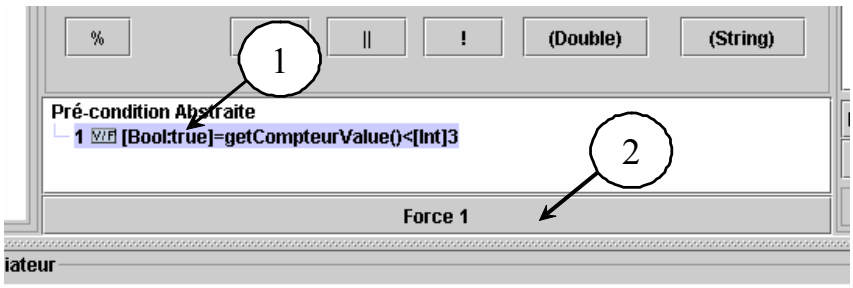

Fig. 3.14 – Exemple d'expressions pour la précondition (garde) sur les objets de la tâche *Conversionfranc/euro*.

L'originalité de l'approche se situe dans la phase de construction de cette expression qui s'appuie sur l'existence d'un noyau fonctionnel développé formellement permettant d'évaluer directement la valeur de l'expression suivant le contexte dans lequel se trouve ce noyau. Cette évaluation est effectuée soit à chaque fin de construction, soit en demandant implicitement une évaluation *force 1* (repère 2, figure 3.14). L'évaluation en *force 1* n'autorise que l'interrogation de l'état du noyau fonctionnel et ne permet pas sa modification.

Exemple d'action de la tâche *Saisir Valeur* **:** nous montrons sur la figure 3.15, l'expression associée à l'action de la tâche *Saisir Valeur*. Sur le repère 1 (figure 3.15), nous désignons une expression qui effectue une modification de l'état de l'application. Cette expression appelle une opération du noyau fonctionnel *setConvertValue* qui possède un paramètre (il s'agit de la valeur à convertir). Ce dernier est représenté sur l'interface par un sous noeud (repère 2, figure 3.15) où nous lui avons affecté dans ce cas précis un paramètre utilisateur de type *double*. Sa valeur doit absolument respecter les spécifications de la précondition de cette opération sous peine de ne pas exécuter l'opération associée (respect des propriétés du noyau fonctionnel développé formellement).

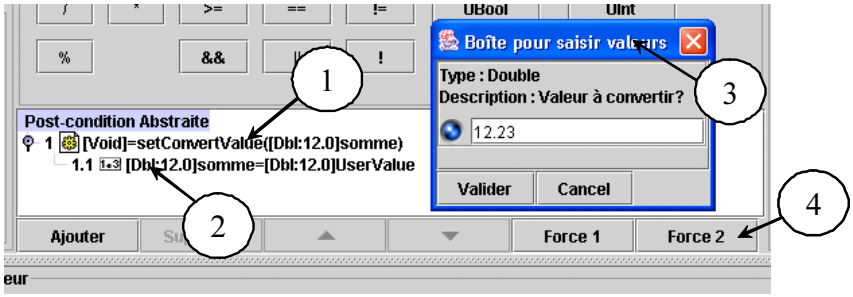

Fig. 3.15 – Exemple d'expressions pour l'action de la tâche interaction *Saisir Valeur*.

L'utilisateur peut tester la conformité de son expression en l'évaluant au moyen de l'option *force 2* (repère 4, figure 3.15). A la différence de l'option *force 1*, celle-ci permet la modification de l'état de l'application, en l'occurrence l'état du noyau fonctionnel. En conséquence de l'évaluation de cette expression, l'utilisateur doit saisir une valeur pour le paramètre de type *double* par le biais d'une interface de saisie (repère 3, figure 3.15) de la présentation abstraite. Suivant la valeur du paramètre et de la précondition (garde) de l'opération, l'action de la tâche est exécutée.

Cet exemple a montré deux aspects intéressants de notre approche. Le premier aspect concerne la satisfaction des propriétés de la spécification du noyau fonctionnel développé formellement qui sont maintenues à chaque modification de son état. Le second aspect est relatif à l'absence de phases d'édition/compilation qui ralentiraient la phase d'évaluation.

Exemple d'action de la tâche *Valeur Convertie* **:** une action d'une tâche application fournit au moins une expression de type *chaîne de caractères* qui ne peut être supprimée (repère 1, figure 3.16). Dans cet exemple, nous avons associé l'opération *getConvertedValue* à cette expression. La valeur retournée par l'opération correspond donc à la valeur de l'expression.

L'ordre d'effectuer l'action de la tâche est donné par l'utilisateur au moyen de l'option

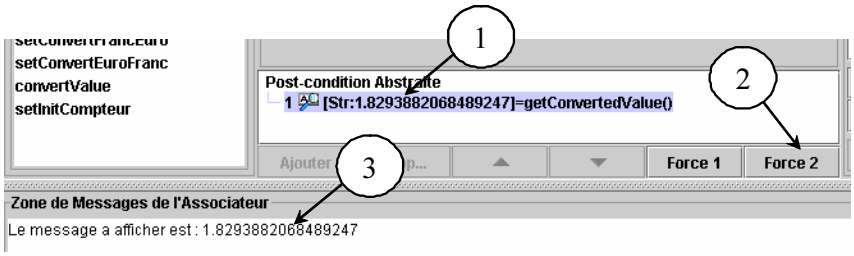

Fig. 3.16 – Exemple d'expressions pour l'action de la tâche application *Valeur Convertie*.

force 2 (repère 2, figure 3.16). Dans cet exemple, l'action n'émet qu'un message dont le contenu est donné par l'expression de type *chaîne de caractères*. Plus précisément c'est la présentation abstraite qui se charge d'afficher le message (repère 3, figure 3.16). Si le contenu de cette chaîne est vide, aucun message ne sera envoyé.

3.4.4 Outil de simulation du modèle de tâches abstrait

La validation du modèle de tâches abstrait est réalisée grâce à l'outil de simulation à tout moment de l'édition du modèle. Le concepteur anime le dialogue abstrait en déclenchant les tâches à la manière de l'outil de simulation de CTTE. Nous présentons sur la figure 3.17, une capture d'écran de cet outil pendant l'enregistrement d'un scénario.

La simulation du modèle de tâches abstrait (repère 1, figure 3.17) implique à chaque instant du déroulement du modèle une modification de l'état du noyau fonctionnel développé formellement (effet de bord provoqué par les actions des tâches). L'outil de simulation est alors capable d'évaluer les préconditions (gardes) sur les objets des tâches activables avant l'activation de celles-ci par le concepteur d'IHM. Cette activation de tâches est effectuée par l'utilisateur soit en sélectionnant une tâche parmi l'ensemble des tâches activables (repère 2, figure 3.17), soit directement sur le modèle de tâches abstrait (repère 1, figure 3.17). Suivant la valeur des préconditions, l'outil de simulation interdit ou pas, la possibilité d'activer les tâches. L'activation provoque l'exécution de l'action qui, selon la nature de la tâche, effectue des traitements sur l'état du noyau fonctionnel formel et/ou sur la présentation abstraite (affichage des messages (repère 6, figure 3.17) ou interface de saisie des paramètres utilisateurs (repère 5, figure 3.17)). Si l'outil de simulation rencontre des problèmes liés à l'exécution des actions (il peut s'agir par exemple d'une contrainte liée au noyau fonctionnel qui n'a pas été respectée), il interrompt le déroulement en localisant et en retournant l'erreur à l'utilisateur.

Au moment de la simulation, le concepteur conserve la possibilité de modifier certains éléments du modèle de tâches abstrait. Sont concernées par cette modification, les pré-

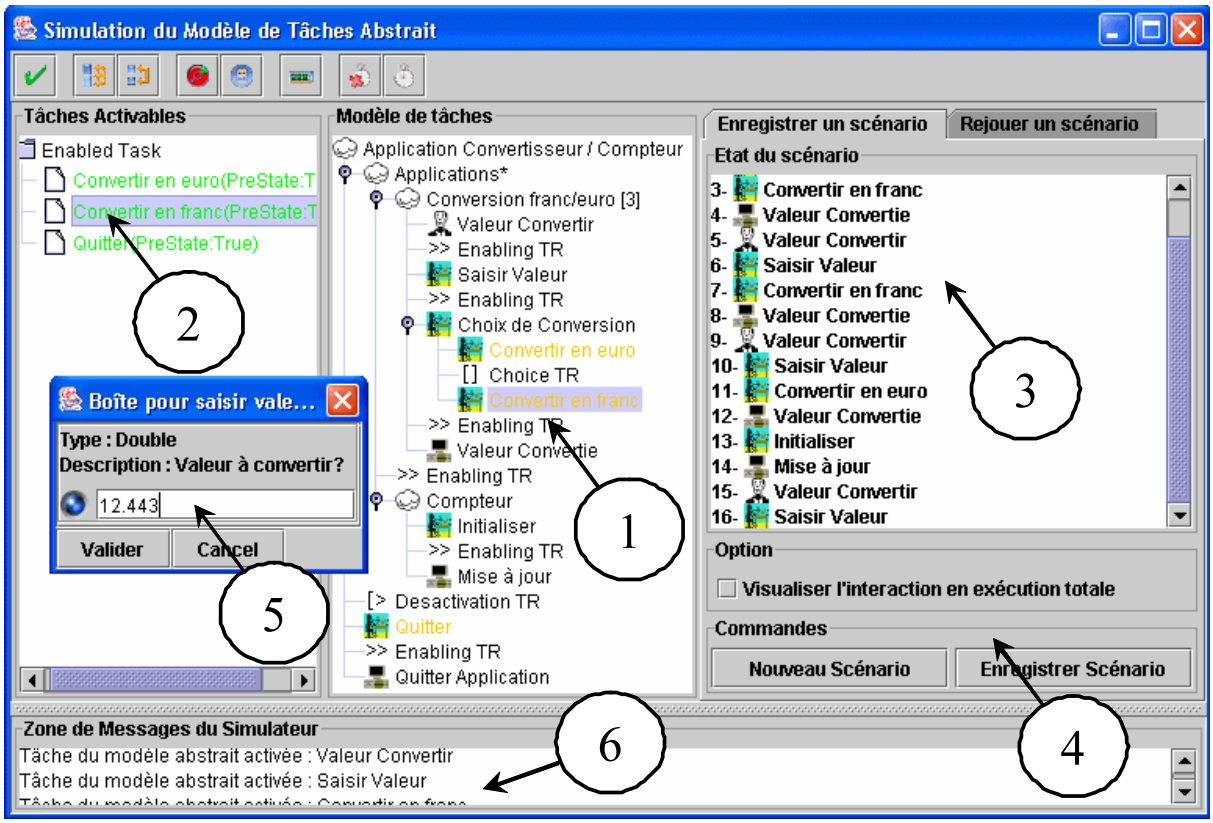

Fig. 3.17 – Capture d'écran de l'outil de simulation du modèle de tâches abstrait : Vue de l'enregistrement d'un scénario.

conditions (gardes) sur les objets ou bien les actions des tâches qui sont toutes les deux modifiées par l'outil de liaison présenté précédemment. L'avantage de cette approche est de pouvoir modifier le contenu des tâches sans la perte du contexte de simulation. Cependant, la modification de la structure hiérarchique de l'arbre de tâches ne peut se faire sans perte de ce contexte. En effet, le modèle de tâches abstrait s'appuie sur une décomposition hiérarchique où l'évaluation des tâches de plus bas niveau (tâches feuilles) est obtenue par celle de plus haut niveau (tâches mères). Par opposition, les formalismes de types automates, comme par exemple les réseaux de Petri dans Petshop peuvent ajouter, modifier ou supprimer de nouvelles places et transitions à tout moment pendant l'exécution des CO-instances sans perte du contexte.

L'activation des tâches construit, au fur et à mesure, un scénario (repère 3, figure 3.17). Le scénario en construction représente une trace (tâches en séquence) vue comme un chemin dans le modèle de tâches abstrait. L'outil de simulation enregistre les tâches qui ont été activées mais aussi, les valeurs des paramètres utilisateurs saisies implicitement dans le cas des tâches interactions.

La lecture de scénarii est obtenue par l'interface présentée sur la figure 3.18 où le repère 3 (figure 3.18) désigne le scénario à rejouer et le repère 4 (figure 3.18) l'état de progression de la lecture du scénario.

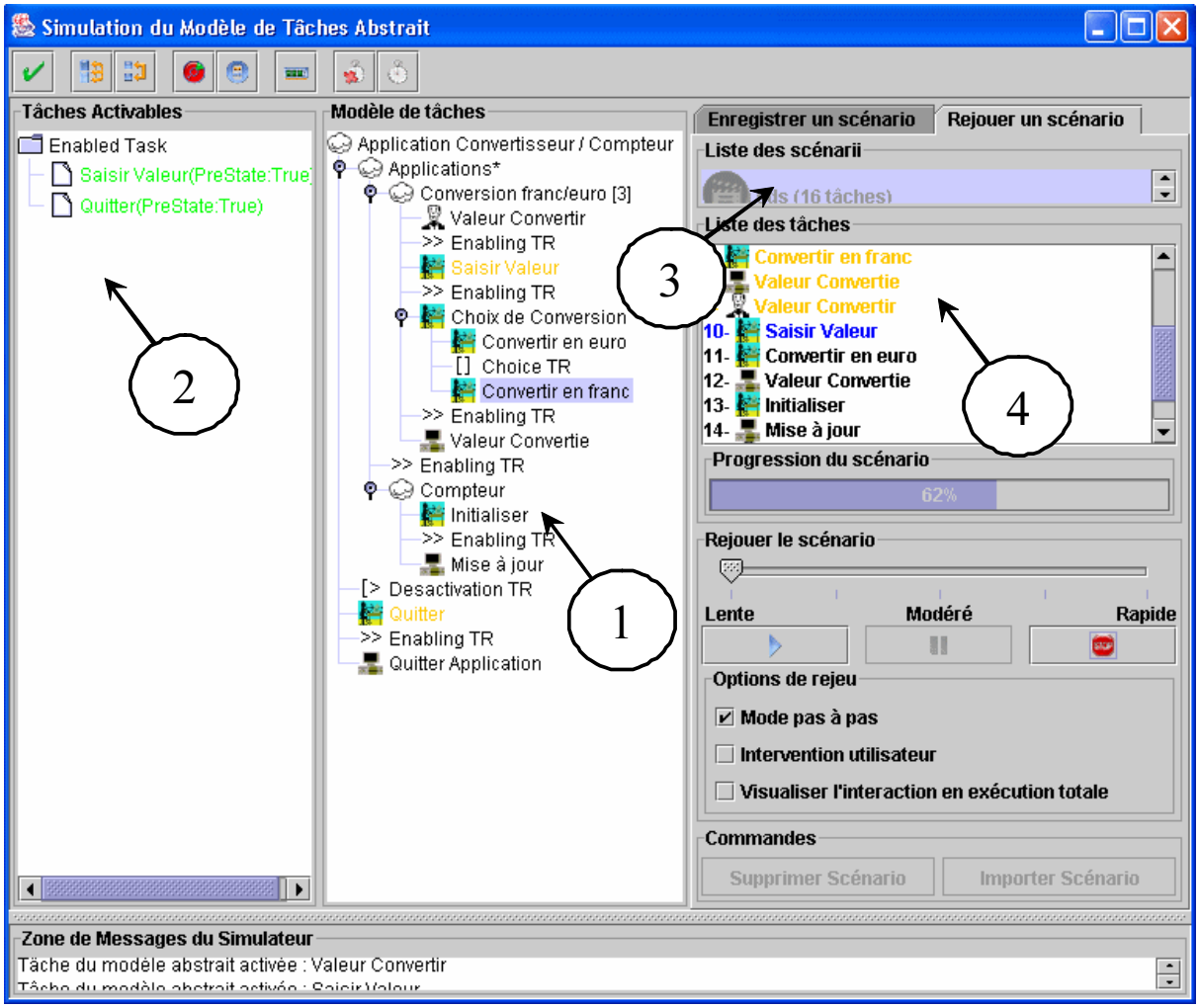

Fig. 3.18 – Capture d'écran de l'outil de simulation du modèle de tâches abstrait : Vue de la lecture d'un scénario.

Le rôle de la simulation abstraite est avant tout de vérifier que les besoins utilisateurs sont correctement pris en compte dans la conception de l'application. Il permet de valider les tâches utilisateurs et de vérifier des propriétés concernant l'atteignabilité et la robustesse du noyau fonctionnel développé formellement. La propriété d'atteignabilité, rappelons-le permet à un utilisateur d'atteindre un état désiré du système depuis l'état actuel, est vérifiée en parcourant l'ensemble des chemins possibles du modèle de tâches abstrait, un peu à la manière de la technique de vérification sur modèles.

Cependant, il faut aussi vérifier que les spécifications du noyau fonctionnel sont main-

tenues par le biais des descriptions de l'adaptateur du noyau fonctionnel. Toutes ces vérifications se font par des tests exhaustifs, qui sont assurés et facilités par l'outil de simulation qui offre au concepteur des outils adaptés (enregistrement de scénarii, lecture, etc). De plus, la prise en compte du noyau fonctionnel développé formellement permet de modifier le contenu des tâches sans perte du contexte. Toutefois, si la vérification n'est pas concluante, les processus de modélisation et de tests devront être réitérés.

3.4.5 Validation sur le noyau fonctionnel : bilan

Cette section a présenté le niveau de la spécification abstraite de l'approche SUIDT. Nous avons montré que ce niveau permettait de prendre en compte les besoins utilisateur au moyen de la notation CTT qui servait à décrire à la fois le dialogue abstrait et la présentation abstraite. La spécification abstraite, appelée modèle de tâches abstrait, a permis d'établir une logique d'appel des opérations du noyau fonctionnel développé formellement en définissant une structure hiérarchique des appels des opérations. Plus concrètement, le modèle de tâches abstrait qui s'occupe du dialogue abstrait lie les éléments de la présentation abstraite aux opérations de ce noyau fonctionnel. Nous avons aussi présenté de nombreux outils (édition, liaison et simulation) qui permettent d'exploiter ce modèle de tâches abstrait en mettant en avant les points originaux de l'approche.

Le premier point concerne l'utilisation d'une notation graphique pour l'édition de la structure du modèle de tâches abstrait et des liaisons entre le noyau fonctionnel et la présentation abstraite. Du point de vue de l'édition du modèle de tâches, notre outil est proche de CTTE ou de VTMB [BBS99] (Visual Task Model Builder) dans le sens où il offre aussi une édition et une visualisation graphique du modèle. Cependant, ces outils ne permettent pas d'associer aux tâches (préconditions (gardes) sur les objets et/ou les actions) des objets existants issus de modèles de l'application (noyau fonctionnel et présentation) car il n'existe pas de liaison avec ces modèles. L'inconvénient majeur de l'absence de liaison se situe au niveau de la validation du modèle de tâches. Les simulateurs de CTTE et VTMB contrôlent le modèle de tâches de toutes sortes d'incohérences (boucle sans fin, tâches inatteignable) et vérifient aussi l'exactitude par rapport aux spécifications du cahier des charges. En revanche, l'outil du modèle de tâches abstrait de SUIDT vérifie aussi les erreurs de construction du modèle en assurant que les appels aux opérations du noyau fonctionnel développé formellement sont cohérents. Enfin, notre démarche permet à l'utilisateur d'« interagir » au moment de la simulation avec la sémantique de l'application en fournissant en entrée une interface de saisie et en sortie des messages textuels.

Ce deuxième point est relié à l'absence de phase de compilation qui ralentirait le processus de conception. D'autres approches possèdent cette caractéristique et notamment celles qui exploitent le modèle à objets CORBA comme MASTERMIND ou Petshop. Ce

résultat est obtenu par l'interprétation des modèles. Cependant, la limite de notre outil se situe au niveau de la perte du contexte de simulation au moment de l'édition de la structure du modèle de tâches abstrait. Toutefois, ce désavantage peut être minimisé face à la présence d'un modèle de tâches qui intègre à la fois les besoins utilisateurs et le dialogue abstrait de l'application, ce que l'outil de Petshop ne propose pas au moment de la conception du dialogue.

Le troisième point est relatif à la possibilité, offerte au concepteur d'IHM, de pouvoir simuler le modèle de tâches abstrait même si il n'est pas finalisé. Il rejoint les approches qui exploitent l'interprétation des modèles comme MASTERMIND.

Finalement, le dernier point est la prise en compte d'un niveau de spécification abstraite indépendant de toute idée d'interaction. L'intérêt principal est de pouvoir vérifier la cohérence des appels des opérations du noyau fonctionnel développé formellement sans se soucier de problèmes supplémentaires liés aux interactions d'une interface utilisateur. Nous ne l'avons pas montré mais, avec ce même noyau fonctionnel, il est possible d'obtenir différents modèles de tâches abstraits. Cependant, quand le concepteur d'IHM conçoit ce modèle, il a déjà une vision globale de la future interface de l'application. Dans la phase suivante celle de la prise en compte des aspects interactions, il peut concrétiser son interface, tout en restant dans la logique de son modèle de tâche abstrait.

3.5 Validation sur la présentation

Nous détaillons dans cette section la spécification concrète de notre approche MBS appelée modèle de tâches concret. Il s'agit d'un modèle de tâches, tout comme le modèle de tâches abstrait, qui permet d'associer aux tâches (préconditions et actions) des éléments d'une présentation concrète. Celle-ci est composée d'un ensemble de composants graphiques utilisés dans les boîtes à outils classiques (bouton, champs de texte, menu, etc).

Toutefois, la notation CTT ne permet pas de décrire explicitement le niveau interaction (clicker sur un bouton) ni le rendu de l'interface (modifier la couleur d'un bouton). Notre objectif est de pouvoir à la manière de la notation UAN décrire une décomposition des interactions de bas-niveau et des rendus de l'interface. Cela permet d'une part, de connaître l'ensemble des possibilités d'interaction que l'utilisateur effectue sur la présentation et d'autre part, d'établir l'ensemble des rendus qui sont réalisés sur cette présentation. Chaque tâche interaction ou application de bas-niveau est alors décomposée en sous-tâches afin de décrire l'interaction et le rendu. Nous nommons cette décomposition : la concrétisation.

Par exemple, dans l'étude de cas du convertisseur/compteur, nous pouvons concrétiser la tâche (*choix du sens de conversion*) appartenant au modèle de tâches abstrait, par la tâche **concrète** permettant de la réaliser sur l'interface (par exemple cliquer sur un bouton ou sélectionner un bouton radio). Dans le modèle de tâches abstrait, aucun choix de technique d'interaction n'était autorisé. A l'inverse, le modèle de tâches concret est destiné uniquement à concrétiser, en termes d'interactions, les feuilles de l'arbre de tâches abstrait. De ce fait, il s'appuie complètement sur ce dernier, et consiste en un raffinement du modèle de tâches abstrait. Tout comme le modèle de tâches abstrait, le modèle de tâches concret fournit une base pour la conception d'un dialogue concret, via les contraintes de précédence et notamment les événements issus de l'interface utilisateur.

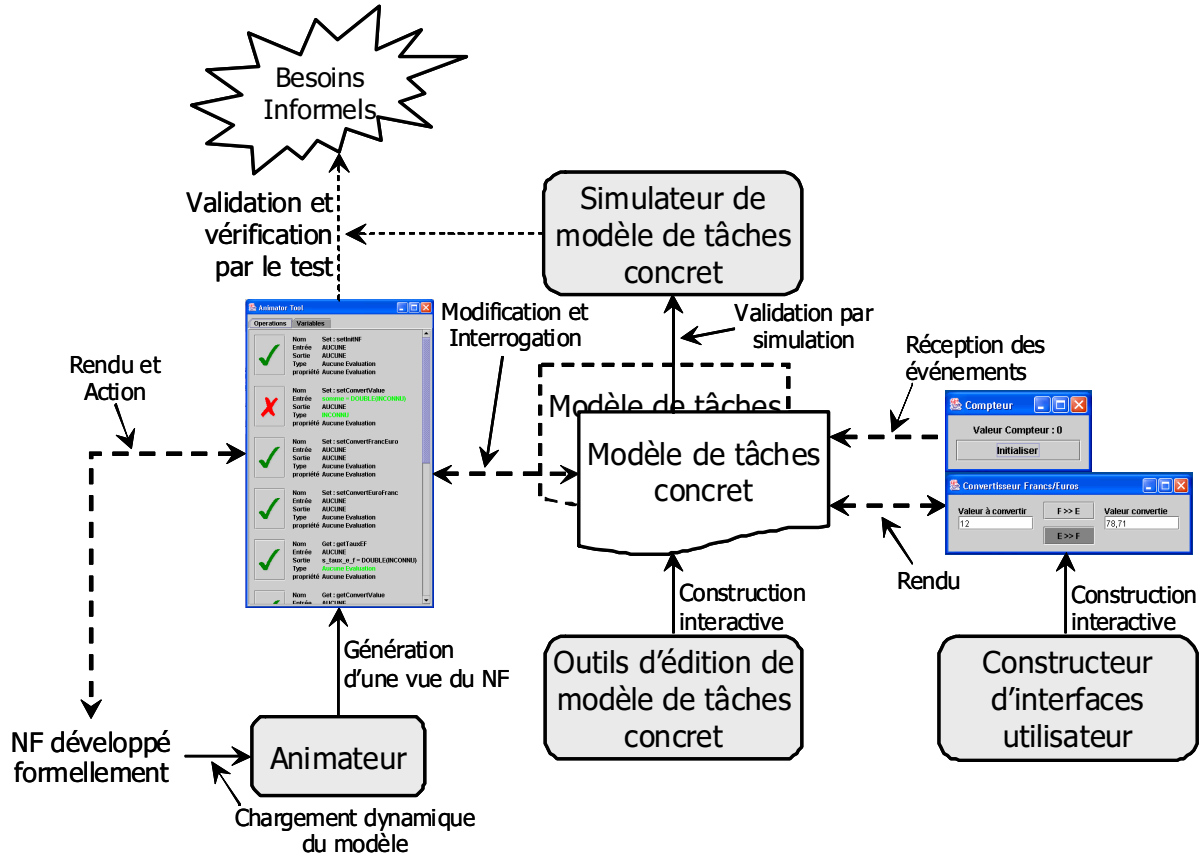

Fig. 3.19 – Démarche pour la modélisation du modèle de tâches concret.

L'approche de modélisation du modèle de tâches concret est présentée sur la figure 3.19. Les outils d'édition de modèles de tâches concrets permettent, d'une part, de construire hiérarchiquement un arbre de tâches concret par raffinement du modèle de tâches abstrait (rectangle en pointillé) et d'autre part, d'associer des objets aux tâches concrètes. Ces objets sont issus du noyau fonctionnel développé formellement par l'intermédiaire de l'outil animateur et de la présentation via un constructeur d'interfaces utilisateur identique à ceux présentés dans la section 1.4.1.3. La validation du modèle de tâches concret

est toujours effectuée au moyen du simulateur. Ce dernier permet de vérifier d'une part le respect des contraintes du noyau fonctionnel (communication entre le modèle de tâches concret et l'animateur) mais il assure aussi que le comportement de l'interface utilisateur correspond bien aux attentes de l'utilisateur. Plus précisément, le comportement de l'interface utilisateur est assuré par la communication entre le modèle de tâches concret et la présentation. Cette communication permet d'autoriser ou pas les actions de l'utilisateur sur des composants graphiques (activer ou désactiver les objets graphiques) mais elle permet aussi d'assurer la mise à jour (modification d'attributs des composants graphiques) de l'interface. Le modèle de tâches concret joue ici le rôle de dialogue concret. Le déroulement de la simulation est identique à celui du modèle de tâches abstrait. Le concepteur d'IHM anime le dialogue concret correspondant au modèle de tâches concret soit en déclenchant des tâches activables soit en interagissant directement sur l'interface utilisateur (flèche *réception des événements*).

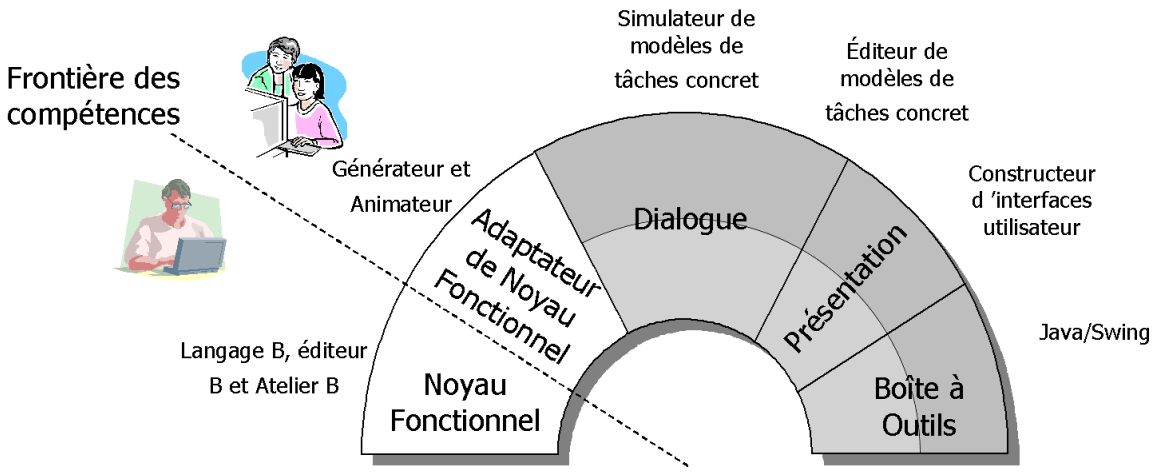

Fig. 3.20 – Concrétisation du module Dialogue, Présentation et Boîte à outils de l'architecture ARCH.

La conception des modules dialogue, présentation et boîte à outils de l'architecture ARCH complète la conception du modèle de tâches abstrait précédente que nous illustrons par la figure 3.20. À la différence de la figure 3.9, nous montrons ici (zones grisées sur la totalité des modules) le fait que la concrétisation du module dialogue, présentation et boîte à outils s'intéresse aux objets du noyau fonctionnel développé formellement et à ceux de la présentation.

Le plan proposé reprend celui de la section précédente, c'est-à-dire que nous proposons dans un premier point, une étude approfondie du modèle de tâches concret, puis nous étudierons dans les autres parties les outils qui servent à éditer et à manipuler ce modèle et plus particulièrement l'outil de simulation. Enfin, nous discuterons de l'apport original de notre contribution face aux approches similaires.

3.5.1 Description du modèle de tâches concret

La modélisation du modèle de tâches concret s'appuie complètement sur le modèle de tâches abstrait. Il emploie le formalisme de tâches CTT sur lequel aucune modification de la sémantique (celle des opérateurs) n'a été effectuée. Nous utilisons aussi les travaux réalisés dans la section précédente concernant les préconditions (gardes) sur les actions des tâches CTT. Par ailleurs, la concrétisation fait apparaître de nouveaux objets liés à la présentation sur lesquels les préconditions et les actions des tâches peuvent être associées.

Toutefois, l'aspect lié aux objets de la présentation n'a pas été, à notre connaissance, traité dans la notation CTT. Le problème reste identique aux objets du noyau fonctionnel. Il faut d'une part être capable de définir une présentation (ensemble de composants graphiques) et d'autre part accéder aux objets de cette présentation afin de pouvoir par exemple interroger ou modifier les attributs de ces objets. Il est à noter que la prise en compte d'objets issus de la présentation a été intégrée dans l'approche TERESA [PS02] qui exploite aussi la notation CTT.

Par ailleurs, sous leur forme actuelle les catégories de tâches (interaction, application, abstraite et utilisateur) proposées par le formalisme CTT ne suffisent pas à décrire la décomposition des interactions et des rendus. L'introduction de nouvelles catégories, comme *tâche concrète interaction* ou *tâche concrète application*, nous amène à définir précisément le rôle des tâches sur la présentation.

Par conséquent, nous proposons dans une première partie, la description de ces objets et nous étudierons plus particulièrement la manière d'analyser les composants graphiques de la présentation. Dans une deuxième partie, nous discutons sur la sémantique des différentes catégories des tâches concrètes (actions et objets de la présentation). Une troisième partie présente le niveau structurel de l'arbre et plus particulièrement du niveau le plus bas. Nous détaillons enfin l'élaboration du dialogue concret.

3.5.1.1 Objets disponibles dans le modèle de tâches concret

Les objets de la spécification concrète sont ceux utilisés dans la modélisation sous-jacente du modèle de tâches abstrait, c'est-à-dire les objets du noyau fonctionnel développé formellement auxquels sont ajoutés les objets de la présentation concrète.

Nous exploitons, dans le modèle de tâches concret, à la manière du modèle de tâches abstrait, les opérations du noyau fonctionnel tout en assurant que les contraintes du noyau fonctionnel sont toujours respectées. En ce qui concerne les objets de la présentation concrète, ils sont issus de la boîte à outils Java/Swing. Les composants graphiques de

cette boîte à outils sont utilisés pour construire l'interface utilisateur au moyen d'un constructeur d'interfaces que nous étudierons dans la section 3.5.2.

L'approche utilisée pour exploiter les composants graphiques s'appuie sur le principe de la technologie Java/Beans[6] où un Beans est vu comme un composant graphique. La force de cette technologie est de pouvoir combiner plusieurs Beans pour élaborer un Beans plus complexe. De plus, cette phase est interactive et ne nécessite pas l'exploitation directe du code Java des Beans pour les combiner entre eux. N'ayant pas exploité directement cette technologie pour des raisons de simplification, nous avons tout de même utilisé la technique d'introspection qui permet d'analyser le contenu des Beans pour extraire les attributs des objets (rendu) et les événements (interaction). L'introspection consiste à étudier le code Java des composants graphiques en extrayant les classes associées, les classes dont ils héritent, les méthodes et les attributs. En comparaison avec la technique de retro-conception employée dans le chapitre 2, la technique d'introspection extrait toutes les informations du code sans perte d'information par abstraction.

Nous nous servons des méthodes de type accesseur pour évaluer la valeur des attributs et des méthodes de type modifieur pour changer la valeur de ces attributs. Plus précisément, la distinction entre les méthodes qui modifient et interrogent les attributs ont une signature précise en Java comme *getAttribut* et *setAttribut*. Par exemple, prenons le cas du composant graphique *bouton*, il possède un attribut de type chaîne de caractères qui représente le texte à afficher sur le bouton. Les méthodes à rechercher sont donc *getText* et *setText*. Cependant, nous avons enrichi la signature des méthodes (de la forme *get_suidt_Attribut* et *set_suidt_Attribut*) afin de filtrer les attributs que nous désirions prendre en compte. Au final, pour un composant graphique donné, nous connaissons ses attributs en sachant s'ils sont modifiables par la présence ou pas d'une méthode de type modifieur. Notons que les types des attributs sont tous ramenés aux types élémentaires utilisés dans le noyau fonctionnel développé formellement (entier, chaîne caractères, double et booléen).

Le principe d'extraction des informations relatives aux événements reste le même. Il s'agit d'introspecter le composant graphique, c'est-à-dire la source[7] (*Event Source* en anglais) afin de reconnaître des méthodes dont la signature est de la forme :

public SUIDTNomAdapter get_event_listener_NomEcouteur() ...

où *NomEcouteur* est le nom donné à l'écouteur[8] (en anglais *EventListener*) et

[6]Sun Microsystems : http ://www.sun.com

[7]Chaque composant graphique peut émettre certains types d'événement.

[8]Des objets (spécifiés par des interfaces Java) correspondent à chaque type d'événement et peuvent recevoir ces événements.

SUIDTMouseAdapter est la référence de l'objet qui écoute la source. Si des méthodes de cette forme sont présentes dans le composant graphique introspecté, cela sous entend que la liaison par abonnement entre la source et l'écouteur existe. A partir de cela, une dernière étape consiste à introspecter chaque objet des écouteurs pour connaître tout naturellement les événements à écouter (clic souris, caractère frappé, etc).

Nous ne nous attarderons pas plus sur le sujet de l'introspection des composants graphiques. Nous dirigeons les lecteurs sur [Eng97] pour de plus amples informations concernant la technologie des Java/Beans et de la technique d'introspection.

Ce mécanisme d'introspection nous a permis d'extraire les informations des composants graphiques : attributs pour le rendu et événements pour les interactions. Voyons maintenant comment exploiter ces informations dans le modèle de tâches concret.

3.5.1.2 Les tâches concrètes

Les catégories de tâches du modèle de tâches abstrait (abstraite, interaction, application et utilisateur) ne permettent pas la description de la décomposition des interactions de bas niveau et des rendus de la présentation. Aussi, nous proposons de nouvelles catégories qui dépendent soit des tâches interactions, soit des tâches applications. Ces nouvelles catégories de tâches portent le nom de **tâche concrète interaction** et **tâche concrète application** où la première est une sous-tâche d'une tâche interaction et la seconde est une sous-tâche d'une tâche application. Les tâches concrètes se distinguent des catégories classiques par le fait qu'elles sont obligatoirement associées à un composant graphique de la présentation. Voyons précisément le rôle de chacune de ces tâches concrètes.

Tâche concrète interaction. Une tâche concrète d'interaction est liée au traitement d'un événement particulier émis par la présentation. Il s'agit donc d'un couple composé d'un composant graphique (bouton par exemple) et d'un événement particulier (cliquer). Plus précisément, cette tâche décrit l'action (cliquer) de l'utilisateur sur la présentation (bouton).

En fait, les actions de l'utilisateur sont les éléments déclencheurs aux traitements d'une tâche concrète interaction et enrichissent donc la précondition (garde) de cette tâche. À cela s'ajoute aussi les propriétés des objets et les contraintes de précédence. Pour les propriétés sur des objets, le prédicat est constructible à partir des opérations du noyau fonctionnel et des attributs de la présentation. Enfin, le traitement de la tâche concrète interaction, c'est-à-dire l'action de la tâche, ne s'applique qu'à la modification de l'état du noyau fonctionnel, tandis que l'état de la présentation est à la charge de la tâche concrète application.

Tâche concrète application. Une tâche concrète application est liée à la modification de l'état de l'application interactive dans sa globalité. Elle concerne l'état du noyau fonctionnel développé formellement et celui de la présentation. Nous avons aussi distingué deux sortes de tâches concrètes application : les **tâches concrètes rendu** et les **tâches concrètes système** :

- les **tâches concrètes rendu** modifient l'état de la présentation. Elles agissent directement sur les valeurs des attributs de cette présentation. Une tâche concrète rendu est donc un couple composé d'un composant graphique (bouton) et d'un attribut appartenant à ce composant (hauteur du bouton). L'action de la tâche modifie donc la valeur de cet attribut en y affectant la valeur d'une expression du type de l'attribut. Cette expression est exprimée notamment par les opérations accesseurs du noyau fonctionnel, les attributs de tous les composants graphiques et différents éléments que nous avions utilisés dans la spécification abstraite ;
- les **tâches concrètes système** servent à appeler des objets n'appartenant pas directement à l'application en cours de construction, c'est-à-dire qu'ils ne sont pas présents dans le noyau fonctionnel ni dans la présentation. Il peut s'agir par exemple d'une boîte de dialogue pour la gestion de l'impression ou bien de l'affichage d'un message sur la console du terminal. Ces objets sont préexistants dans l'environnement SUIDT. Ils permettent d'enrichir la conception par des options supplémentaires que les autres objets ne peuvent fournir. Dans notre étude de cas, la conversion d'une somme pourrait être signalée à l'utilisateur par un objet système qui émet un signal sonore. L'action de la tâche concrète système appelle donc l'objet en lui passant des paramètres via une expression spécifique construite suivant le principe des autres expressions.

3.5.1.3 Structure du modèle de tâches concret

La structure hiérarchique du modèle de tâches concret s'appuie sur la base de celle du modèle de tâches abstrait. Nous y retrouvons toutes les catégories de tâches CTT aussi que les tâches concrètes. La concrétisation (passage du modèle de tâches abstrait au modèle de tâches concret) est comparable aux approches descendantes des techniques formelles (voir section 1.3.1.3) dans le sens où le passage d'un modèle à un autre est vu comme un raffinement. Dans le modèle de tâches abstrait, seuls les objets du noyau fonctionnel développé formellement sont manipulés. Nous complétons la modélisation du modèle de tâches concret en intégrant les objets de la présentation et les tâches concrètes. Ainsi, les préconditions (gardes) des tâches du modèle de tâches concret sont exprimables à la fois avec les accesseurs du noyau fonctionnel mais aussi avec les attributs de la présentation. Les tâches feuilles du modèle de tâches abstrait ne le sont plus dans le modèle de tâches

concret puisqu'elles ont été concrétisées et un niveau de décomposition a été ajouté. Par conséquent leurs actions ne sont plus prises en compte.

Nous présentons sur la figure 3.21 les interfaces utilisateurs et une partie du modèle de tâches concret relatif à notre étude de cas. Nous avons gardé la représentation de CTTE puisqu'elle permet une meilleure compréhension de la décomposition. Par manque de place nous n'avons pas pu préciser tous les composants graphiques associés aux tâches concrètes.

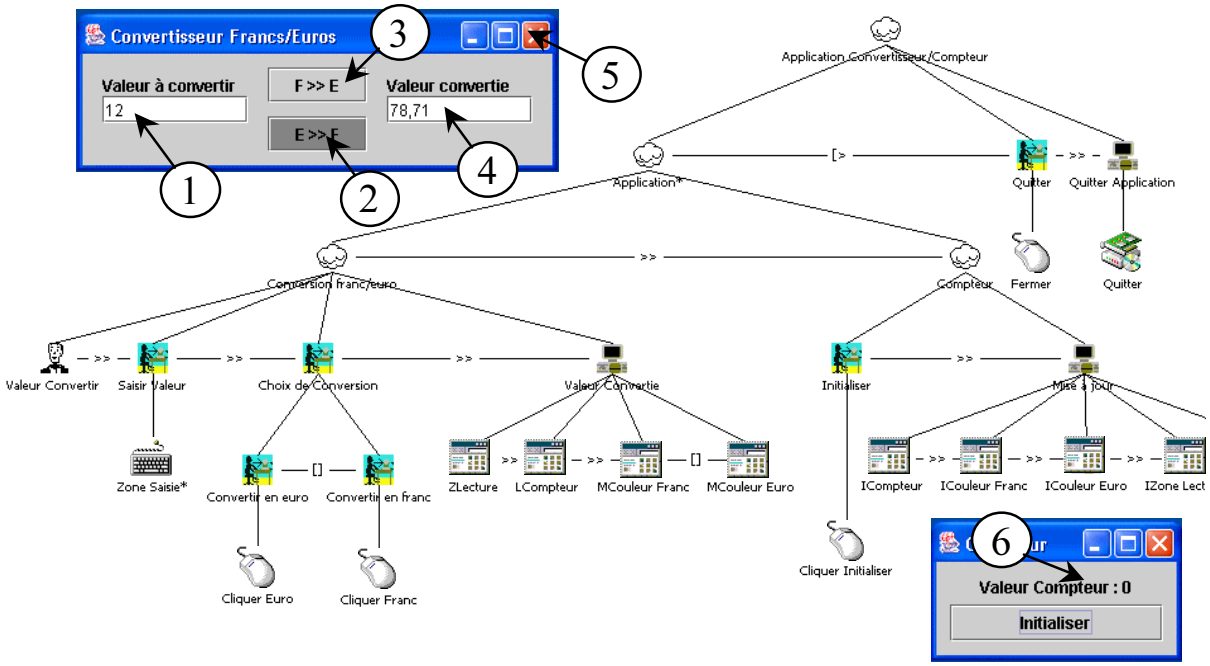

Fig. 3.21 – Extrait du modèle de tâches concret.

Par exemple, la tâche *Saisir Valeur* est concrétisée par la tâche concrète *Zone Saisie*. Cette tâche est associée à la zone de saisie (repère 1, figure 3.21) et à l'événement *keyTyped* de la boîte à outils Java/Swing. La tâche *Valeur Convertie* modifie l'affichage après la conversion. Elle a été concrétisée par un ensemble de tâches concrètes rendues qui modifient les attributs de la zone de lecture (repère 4, figure 3.21) pour afficher la valeur de conversion, du texte (repère 6, figure 3.21) pour mettre à jour la valeur du compteur et enfin des boutons (repère 2 et 3 de la figure 3.21) pour modifier leur couleur de fond afin de préciser l'état de conversion. La tâche *Mise à jour* est, comme son nom l'indique, une mise à jour de la présentation. Enfin, la tâche *Quitter Application* est concrétisée par une tâche concrète système appelée *Quitter* associée à la fenêtre (repère 5, figure 3.21). Son rôle est d'appeler une option du système pour terminer l'exécution de l'application.

Notons que ces ajouts ne modifient en rien la sémantique de CTT. La seule entorse au formalisme réside dans le fait que, alors qu'en CTT « pur » une tâche ne peut-être

décomposée en une seule sous-tâche (deux sous-tâches sont nécessaires au minimum), le passage du modèle de tâches abstrait au modèle de tâches concret peut se faire par une décomposition en un seul fils (si l'interface ne propose pas deux manières d'exécuter une tâche).

Précisons aussi que le niveau de concrétisation se limite à un seul niveau de décomposition comme le montre sur la figure 3.21 et où tous les opérateurs temporels de la sémantique du modèle de tâches abstrait sont utilisables.

Enfin, nous avons distingué deux types d'interactions relatives à l'action de l'utilisateur sur la présentation :

- les interactions **continues** se répètent indéfiniment jusqu'à ce qu'elles soient désactivées par une action. Dans notre exemple, il s'agit de la tâche concrète interaction *Zone Saisie* qui correspond à la saisie en continue d'une somme jusqu'à ce que l'utilisateur choisisse d'effectuer la conversion. Du point de vue de la modélisation, nous traduisons le fait qu'une tâche concrète interaction est continue par la caractéristique itération infinie. Plus précisément, la tâche de désactivation n'est pas nécessairement une tâche concrète interaction, il peut s'agir aussi d'une tâche concrète application ;
- les interactions dites **uniques** ne se produisent qu'une fois. C'est par exemple le cas pour la tâche *Cliquer Initialiser* qui est suivie après son exécution par la tâche *ICompteur*.

3.5.1.4 Le dialogue concret

Le dialogue concret, comme celui du modèle de tâches abstrait, joue le rôle de médiateur entre la présentation concrète (interfaces utilisateur Java/Swing) et l'adaptateur du noyau fonctionnel. Il offre cependant l'avantage de pouvoir retourner son état au travers d'interfaces interactives existantes ce qui dans le modèle de tâches abstrait se déroulait par messages textuels.

Étant en liaison directe avec les besoins de l'utilisateur, le dialogue concret est à même de contrôler et autoriser les interactions de l'utilisateur. Par exemple, dans notre étude de cas, tant que l'utilisateur n'a pas initialisé la valeur du compteur en cliquant sur le bouton, aucune interaction sur l'interface du convertisseur n'est autorisée puisque les composants graphiques boutons et zones de saisie sont inactifs. Ce traitement est implicite, le concepteur n'indique pas que tel ou tel composant doit être inactif dans tel état du dialogue. Nous reviendrons plus précisément sur cet point au moment de la simulation du modèle de tâches concret.

Le dialogue concret s'appuie aussi sur la dynamique du modèle de tâches concret via la logique des opérateurs de Lotos, sur les propriétés sur les objets mais aussi sur le formalisme à base d'événements de la boîte à outils Java/Swing qui traite ces événements par abonnement.

Le déroulement du dialogue concret est le suivant :

- quand le déroulement du dialogue concret est dans un état où il existe une ou plusieurs tâche(s) concrète(s) interaction, il attend la réception d'un événement de la présentation. Le dialogue exécute alors la tâche correspondant à l'événement reçu (associé au composant graphique) à condition que sa précondition (garde) sur les objets soit respectée. Le dialogue exécute alors l'action de cette tâche et met à jour l'état du dialogue ;
- quand le déroulement du dialogue concret est dans un état où il existe une tâche concrète application, elle est automatiquement exécutée ;
- dans le cas où il existe plusieurs tâches concrètes applications (par exemple les tâches concrètes rendu *MCouleur Franc* [] *MCouleur Euro*), les propriétés sur les objets déterminent la tâche à exécuter. Si la précondition (garde) sur les objets ne permet pas de faire ce choix, une tâche application est exécutée ;
- dans le cas où il existe une ou plusieurs tâche(s) concrète(s) interaction et une ou plusieurs tâche(s) concrète(s) application, le dialogue concret prend en priorité les tâches concrètes application et détermine la valeur de leur précondition sur les objets pour choisir la tâche concrète application à exécuter (scénario similaire au précédent). Cependant, si aucune précondition sur les objets n'est respectée, le dialogue concret est en attente, tout en évaluant constamment les préconditions des tâches concrètes application, d'un événement de la présentation.

3.5.2 Constructeur d'interfaces utilisateur

Cet outil permet la construction de la présentation concrète de l'application. Une vue de cet outil est donnée sur la figure 3.22. Ses fonctionnalités sont proches d'un constructeur classique tels que ceux que l'on trouve dans les environnements de développement des approches ascendantes. Il manipule interactivement les composants graphiques de la boîte à outils Java/Swing (repère 5, figure 3.22) afin de construire la présentation dans une zone de travail (repère 1, figure 3.22).

C'est un outil interactif qui permet de visualiser directement le résultat sans passer par une phase de compilation. Il permet aussi d'introspecter les composants graphiques sélectionnés afin d'accéder aux attributs (repère 2, figure 3.22) qui sont pour certains inat-

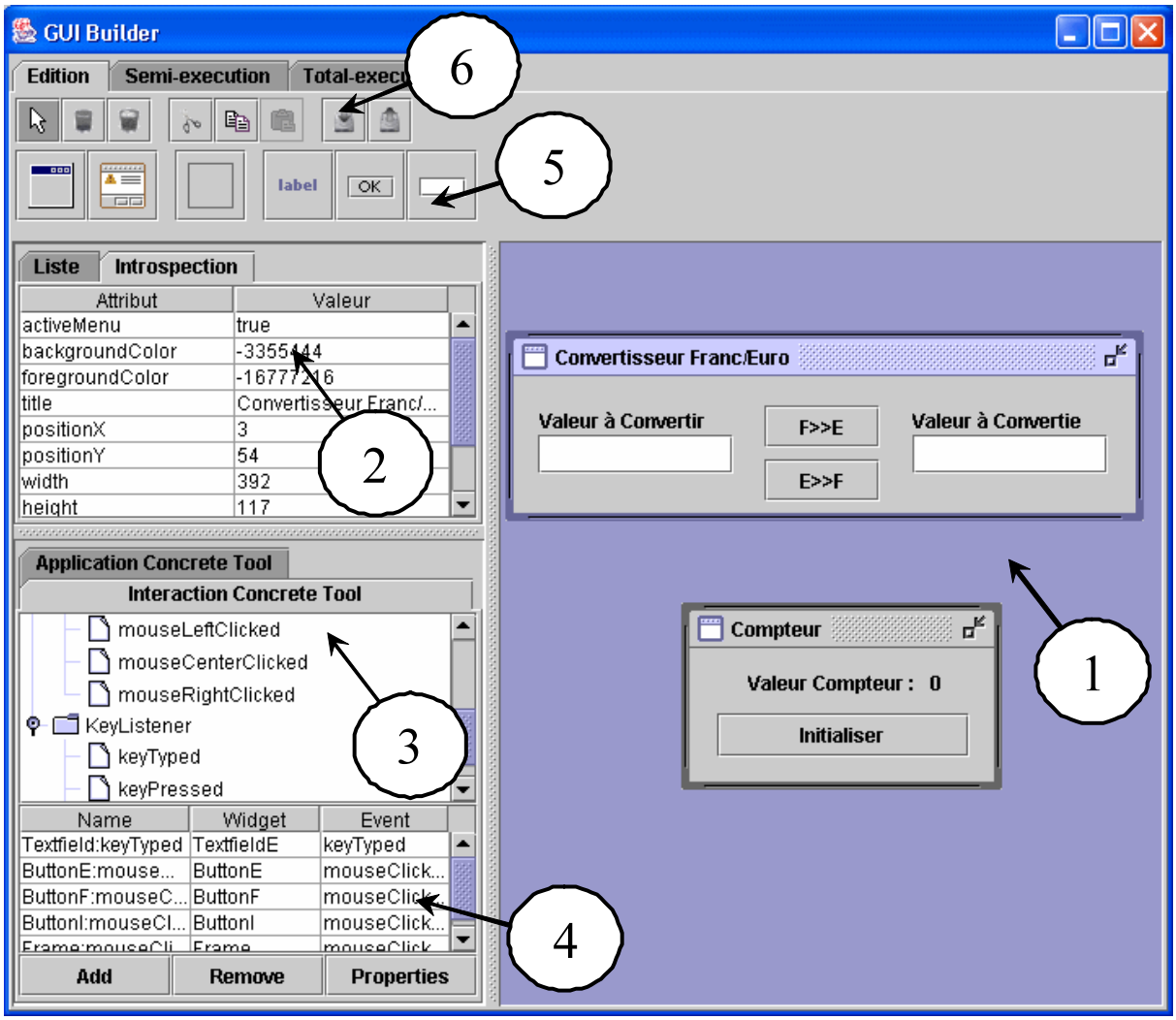

Fig. 3.22 – Capture d'écran de l'outil de construction d'interfaces utilisateur.

teignables graphiquement ou tout simplement pour effectuer des modifications spécifiques sur ces attributs. De nombreuses options (copier/coller, gestionnaire de composants graphiques préconçus, etc) ont été ajoutées afin de faciliter le travail du concepteur d'IHM (repère 6, figure 3.22).

Ce qui différencie ce constructeur d'interfaces utilisateur par rapport aux autres outils du même genre, c'est la possibilité de s'abstraire du code pour effectuer la liaison avec le reste du développement de l'application. En effet, alors que dans les générateurs de présentation la dynamique du dialogue et la mise en place des effets de bords sont réalisées par programmation textuelle où une connaissance en programmation est nécessaire, notre outil est lié directement à l'environnement de développement. Le travail du concepteur d'IHM utilisant cet outil consiste à construire interactivement les interfaces graphiques et à

associer les composants graphiques de l'interface avec des attributs ou des événements afin de créer les tâches concrètes. Sur les repère 3 et 4 (figure 3.22) nous montrons l'association des composants graphiques avec les événements Java/Swing.

3.5.3 Outils d'édition du modèle de tâches concret et de liaison entre les objets du noyau fonctionnel, de la présentation et des tâches

Les outils d'édition et de liaison du modèle de tâches abstrait ont été enrichis afin de prendre en compte les objets issus de la présentation concrète.

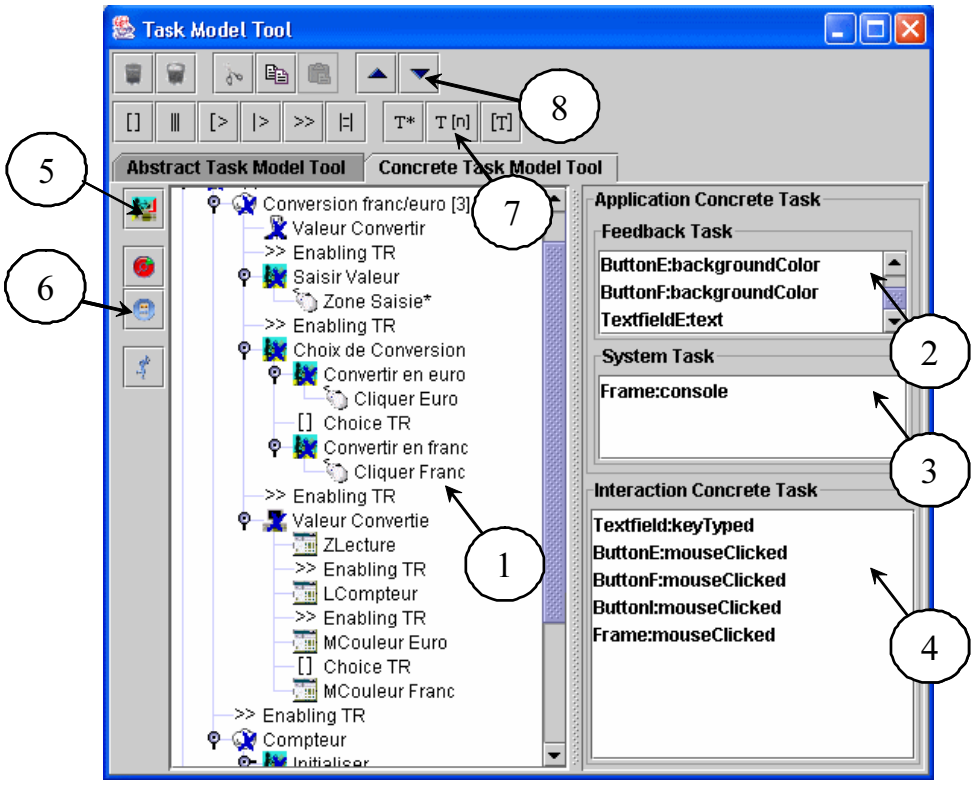

Fig. 3.23 – Capture d'écran de l'éditeur de modèle de tâches concret.

Le premier outil permet d'éditer graphiquement le modèle de tâches concret en s'appuyant sur une représentation identique de l'arbre de tâches. Nous présentons sur la figure 3.23 une capture d'écran de cet outil où nous retrouvons sur le repère 1 (figure 3.23) la zone d'édition, les opérateurs sur le repère 7 (figure 3.23), etc. Sur les repères 2, 3 et 4 (figure 3.23), les couples composants/attributs ou composants/événements qui ont été associés dans l'outil précédent sont présentés. L'édition consiste à utiliser ces couples pour concrétiser les tâches de bas niveau du modèle de tâches abstrait. L'utilisateur peut définir

alors des tâches concrètes application et interaction. L'originalité de cet éditeur, est de pouvoir respecter, au moment de la création de l'arbre de tâches, la syntaxe du modèle de tâches concret que nous avons définie précédemment (niveau de décomposition, respect des catégories à concrétiser, etc).

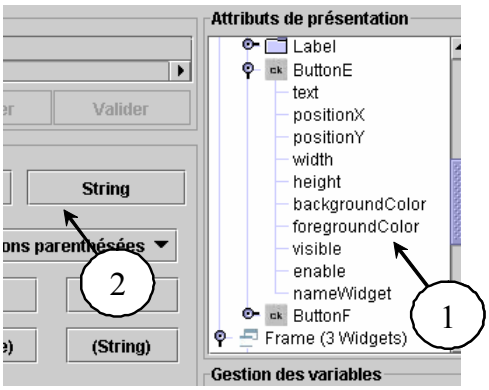

Fig. 3.24 – Capture d'écran des attributs de la présentation dans l'outil de liaison entre les objets du noyau fonctionnel et les tâches du modèle de tâches concret.

L'outil de liaison entre les tâches et les objets de l'application est enrichi par la gestion des attributs de la présentation concrète. Nous donnons une capture d'écran de l'interface de cet outil sur la figure 3.24 où le repère 1 (figure 3.24) désigne les attributs de la présentation qui permettent de définir les expressions de la précondition et de l'action des tâches.

Exemple d'action de la tâche concrète interaction *Zone Saisie*. Nous présentons sur la figure 3.25 la liaison entre la tâche *Zone Saisie*, l'opération *setConvertValue* (modification de la valeur à convertir) du noyau fonctionnel développé formellement et de l'attribut qui retourne le contenu de la zone de saisie. La conversion de type entre cet attribut (chaîne de caractères) et le paramètre de l'opération (double) est effectuée de façon implicite.

Cet outil permet de tester directement l'effet de l'action de la tâche itérative *Zone-Saisie* (repère 1, figure 3.25) en utilisant l'option *force 2* (repère 2, figure 3.25) (voir section 3.4.3). A la différence du modèle de tâches abstrait où l'utilisateur saisit une valeur dans une interface de saisie générique liée à la présentation abstraite, l'option *force 2* lui permet d'agir directement sur le composant concerné (zone de saisie) désigné par le repère 3 (figure 3.25). A chaque interaction sur la présentation, l'action de la tâche est exécutée. A chaque activation de la tâche *Zone Saisie*, la valeur à convertir du noyau fonctionnel est modifiée.

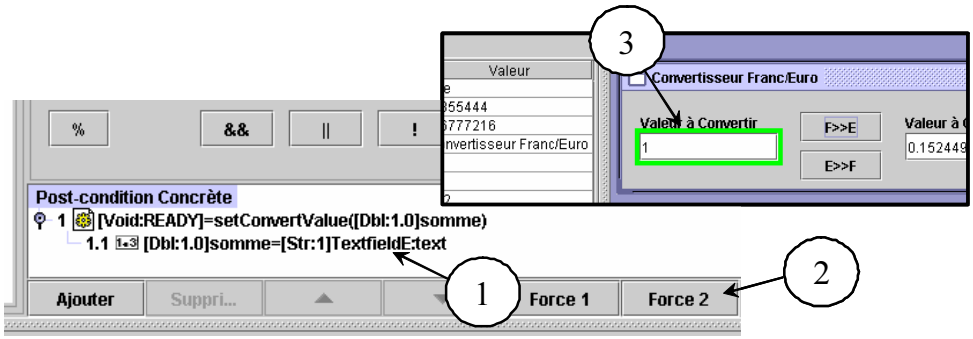

Fig. 3.25 – Expressions pour l'action de la tâche interaction concrete *Zone Saisie*.

Exemple d'action de la tâche concrète application *MCouleur Euro*. Nous montrons sur la figure 3.26, l'exemple de la liaison entre la tâche concrète rendue *MCouleur-Euro* et l'attribut de couleur de fond du bouton *F»E*. Nous avons affecté à cet attribut une valeur constante (repère 1, figure 3.26) qui modifiera alors la couleur du bouton pour retourner l'état de conversion.

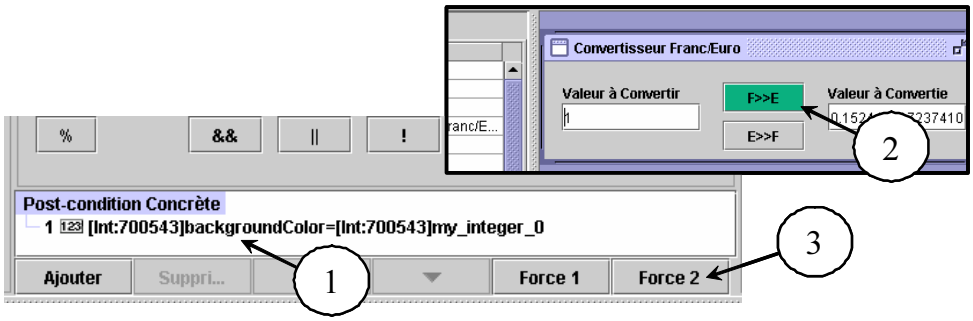

Fig. 3.26 – Expressions pour l'action de la tâche concrete rendue *ICouleur Euro*.

L'option *Force 2* (repère 2, figure 3.26) permet d'évaluer l'action de la tâche et de visualiser directement le résultat sur la présentation (repère 3, figure 3.26). Cette étape est instantanée est ne nécessite pas de phase de compilation puisque l'environnement SUIDT interprète la conception.

3.5.4 Outil de simulation du modèle de tâches concret

Cet outil, présenté sur la figure 3.27, permet la simulation du modèle de tâches concret (repère 1, figure 3.27) et son fonctionnement est similaire à celui du modèle de tâches abstrait. Cependant, grâce à cet outil, l'utilisateur peut tester l'application en cours de construction en visualisant l'état de celle-ci soit sur l'animateur du noyau fonctionnel (à

la manière du modèle de tâches abstrait) ou soit directement sur la présentation concrète, repère 4 (figure 3.27). Le simulateur agit maintenant sur les objets de la présentation et plus particulièrement sur les attributs pour le rendu, puis sur les événements pour les actions de l'utilisateur. Le déroulement du modèle de tâches concret s'appuie sur le dialogue concret que nous avons détaillé précédemment. Enfin, il existe toujours deux phases de simulation, une pour l'enregistrement de scénarii et l'autre pour la relecture de ces scénarii.

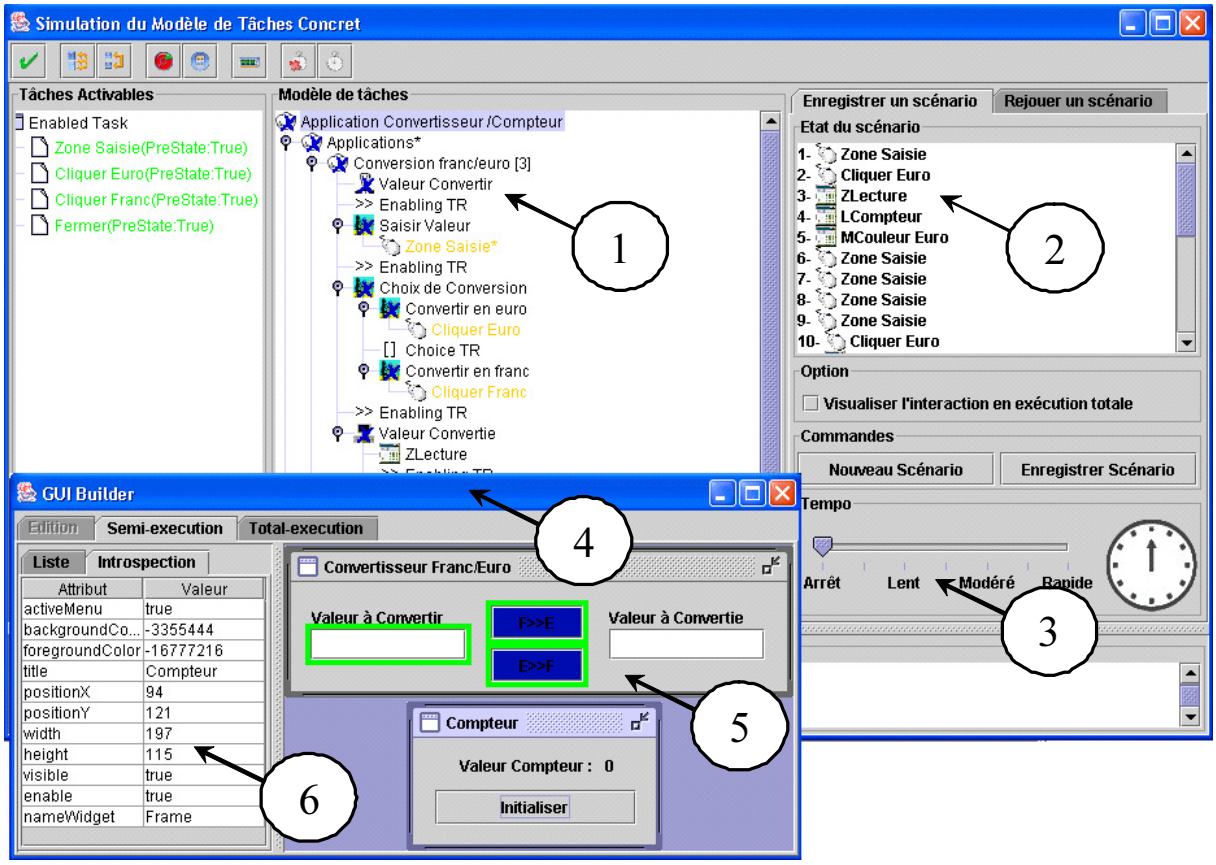

Fig. 3.27 – Capture d'écran de l'outil de simulation du modèle de tâches concret : vue de l'enregistrement d'un scénario.

Même si le modèle de tâches concret n'est pas complètement finalisé, la simulation le complète par les tâches du modèle de tâches abstrait. Plus précisément, le simulateur s'appuie sur le modèle sous-jacent du modèle de tâches concret pour le simuler si celui-ci n'est pas finalisé. Par exemple, si la tâche *Saisir Valeur* n'est pas encore concrétisée, le simulateur exploitera cette tâche pour déterminer la valeur de la précondition et pour exécuter l'action. Dans cet exemple précis, la présentation abstraite liée à une interface de saisie sera utilisée pour saisir la valeur à convertir.

Le simulateur évalue les préconditions (gardes) des tâches activables en utilisant la

conjonction des préconditions (gardes). Ainsi, une tâche ne pourra être activable que si le prédicat correspondant à la précondition est correct. Dans ce cas, le simulateur exécute l'action associée à ces tâches. Le simulateur vérifie aussi l'intégrité des contraintes des opérations du noyau fonctionnel.

A tout moment de la simulation, l'utilisateur peut modifier les préconditions (gardes) et les actions des tâches mais il peut aussi modifier l'agencement de la présentation au travers du constructeur d'interfaces utilisateur qui reste en partie actif pendant la simulation, repère 4 (figure 3.27). Cependant, la modification ne s'applique qu'aux objets existants et l'utilisateur ne peut ajouter ou supprimer des objets. Plus précisément, il ne peut agir que sur les attributs (couleurs, tailles, etc) des objets existants de la présentation, repère 5 et 6 (figure 3.27).

Du point de vue des activations des tâches concrètes, on distingue celles des tâches concrètes interaction et des tâches concrète application.

Pour les premières, l'utilisateur a la possibilité de les activer directement à partir de l'arbre concret (repère 1) (à la manière de la simulation du modèle de tâches abstrait) ou directement en agissant directement sur la présentation (repère 5). L'utilisateur effectue alors l'action correspondant à l'événement attendu par la tâche. Afin de distinguer les composants graphiques de la présentation sur lesquels l'utilisateur peut interagir, le simulateur les signale par une zone de surbrillance. De plus, connaissant l'état de déroulement du modèle de tâches concret, le simulateur peut modifier l'état de la présentation pour faire correspondre ces deux états. Par exemple, si la valeur du compteur est égale à trois, l'utilisateur doit initialiser cette valeur pour effectuer de nouvelles conversions. Cependant, tant qu'il n'a pas cliqué sur le bouton initialiser pour effectuer cette action, les interactions concernant la saisie de la valeur à convertir ou les boutons de conversion ne sont pas accessibles (désactivés). L'originalité de l'approche est de pouvoir modifier implicitement l'état de la présentation suivant celui du dialogue sans avoir à intervenir directement sur les attributs de la présentation. Nous modélisons ainsi la propriété d'insistance.

Pour les activations des tâches concrètes application, nous avons intégré la notion de temps d'exécution par la présence d'un variateur qui permet de le réguler (repère 3). Cette option a été mise en place afin que le déroulement du modèle de tâches concret puisse se faire de façon continue et de façon automatique. Ainsi, si le variateur est dans l'état *pas à pas*, c'est à l'utilisateur d'activer les tâches, sinon les tâches sont activées automatiquement suivant une durée d'attente fixée par le variateur. Dans l'exemple de la tâche *Mise à jour*, cinq tâches se succèdent séquentiellement. L'activation de ces tâches suit la logique décrite dans la section 3.5.1.4. L'avantage de la régulation du temps d'exécution est de pouvoir contrôler au fur et à mesure du déroulement du modèle de tâches concret les modifications apportées sur la présentation.

Finalement, au cours de l'enregistrement des scénarii, toutes les informations relatives aux actions de l'utilisateur sont enregistrées au moyen des tâches concrètes interaction. Dans le cas de la tâche *Zone Saisie*, il s'agit par exemple des codes clavier utilisés pour saisir la valeur à convertir. Ainsi, la relecture du scénario peut rejouer sur la présentation, les mêmes actions que l'utilisateur. Le simulateur peut cependant laisser à l'utilisateur le soin d'agir directement sur la présentation mais en le forçant à suivre un chemin précis du modèle de tâches concret.

3.5.5 Validation sur la présentation : bilan

Cette section a présenté le dernier niveau de modélisation de notre système basé sur modèles (SUIDT), appelé modèle de tâches concret. Il a permis de montrer qu'une interface graphique de type WIMP pouvait être conçue de façon à intégrer les besoins de l'utilisateur au plus tôt dans le processus de conception. Le modèle de tâches concret a rempli son rôle de dialogue concret en contrôlant que les liaisons entre les objets du noyau fonctionnel et ceux de la présentation étaient corrects. Plus précisément, et par son intermédiaire, nous avons pu vérifier pendant la simulation de ce modèle que les propriétés du noyau fonctionnel étaient respectées et que l'état de la présentation (rendu et actions disponibles) était conforme au souhait exprimé par la décomposition de tâches. Toute la modélisation et la vérification de ce modèle ont été facilitées par un ensemble d'outils interactifs orientés pour le concepteur d'IHM (édition d'arbre de tâches, constructeur d'interfaces, liaison et simulation). Plusieurs aspects intéressants à ce niveau de modélisation ont été mis en avant, nous en proposons un résumé tout en discutant de l'apport original en comparaison avec les différents systèmes basés sur modèles.

Le premier aspect concerne la construction de la présentation. Nous pouvons rapprocher notre démarche à celle de Teresa puis à MOBI-D. Toutes ces approches utilisent la description du dialogue (issue de la modélisation de tâches) comme une base pour construire l'interface de l'application. Dans notre cas, l'établissement du modèle de tâches abstrait (dialogue abstrait) laisse déjà présager d'interfaces types et oblige donc le concepteur d'IHM à se conformer à cette description. Teresa, par exemple, s'appuie sur des règles de conception et de guides de style pour élaborer la présentation. De même, MOBILE s'appuie sur une démarche mixte dans le sens où il propose à l'utilisateur une première phase où l'outil TIMM automatise la construction de la présentation (proposition de composants graphiques adaptés aux tâches) et d'une deuxième phase s'appuyant sur la précédente afin de créer de manière ascendante la présentation avec l'outil MOBILE. Notre approche au contraire ne propose pas de phase automatique, mais la construction de la présentation se fait de manière ascendante et doit obligatoirement correspondre à la description du dialogue. Nous reviendrons sur l'aspect des guides de style et des règles de conception dans les perspectives de cette thèse.

Le deuxième aspect avait déjà été évoqué au niveau du modèle de tâches abstrait. Il concerne la possibilité d'alterner phases d'édition et de conception tout en préservant les liens sémantiques entre les modèles. L'intérêt de l'approche SUIDT est que chaque modèle est exécutable. Les modèles peuvent être exécutés pour l'inspection, la simulation et la vérification. Le modèle du domaine, c'est-à-dire le noyau fonctionnel, joue un rôle central dans l'approche SUIDT. Chaque modèle est en fait relié à ce modèle du domaine, et permet de faire fonctionner l'application en cours de développement à chaque fois qu'un modèle est exécuté. Cet aspect donne à SUIDT un haut niveau de degré d'interactivité. De plus, tant que les liaisons sont toujours actives, l'utilisateur ne perd pas les bénéfices de la modélisation et peut modifier une partie de l'application tout en respectant les contraintes exprimées dans les modèles. Au niveau de la modélisation de la spécification concrète, cet aspect se traduit par la possiblité de modifier l'agencement de la présentation de l'application tout en simulant le modèle de tâches concret. A l'inverse Petshop par exemple, ne permet pas la modification directe de la présentation puisque celle-ci est construite par l'outil JBuilder qui n'est pas directement intégré à Petshop.

Dans [PV02], l'auteur présente des heuristiques de conception qui aident progressivement à obtenir la présentation à partir du modèle de tâches et du modèle de domaine. Notre approche est similaire, puisque le développement de l'application débute avec le modèle du domaine (noyau fonctionnel) et le modèle de tâches abstrait sans introduire une vue de l'interface. Quand ces deux modèles sont construits, la présentation peut être établie.

3.6 Conclusion générale sur l'approche SUIDT

Deux acteurs dans le cycle de conception ayant des connaissances particulières ont été distingués. Le **spécialiste**, maîtrisant les méthodes formelles, s'occupe de la conception du code du noyau fonctionnel. Il utilise pour cela un **éditeur de développement interactif** (EDI) et l'outil **Atelier B** pour spécifier, vérifier et valider le noyau fonctionnel. Un second acteur que nous avons qualifié de **concepteur d'IHM** construit interactivement l'interface utilisateur de l'application à développer.

Le niveau le plus haut du modèle est composé de la spécification B du **noyau fonctionnel développé formellement** et du **modèle de tâches CTT**. Le niveau intermédiaire du modèle définit l'**adaptateur de l'application** et le **modèle de tâches abstrait**. Le premier est obtenu par le **générateur** avec la spécification B du noyau fonctionnel, tandis que le second est basé sur le **modèle de tâches CTT** construit au moyen de l'**éditeur de modèles de tâches abstrait**. Le niveau le plus bas correspondant au niveau concret représente le **modèle de tâches concret**. Ce niveau se base sur le précédent

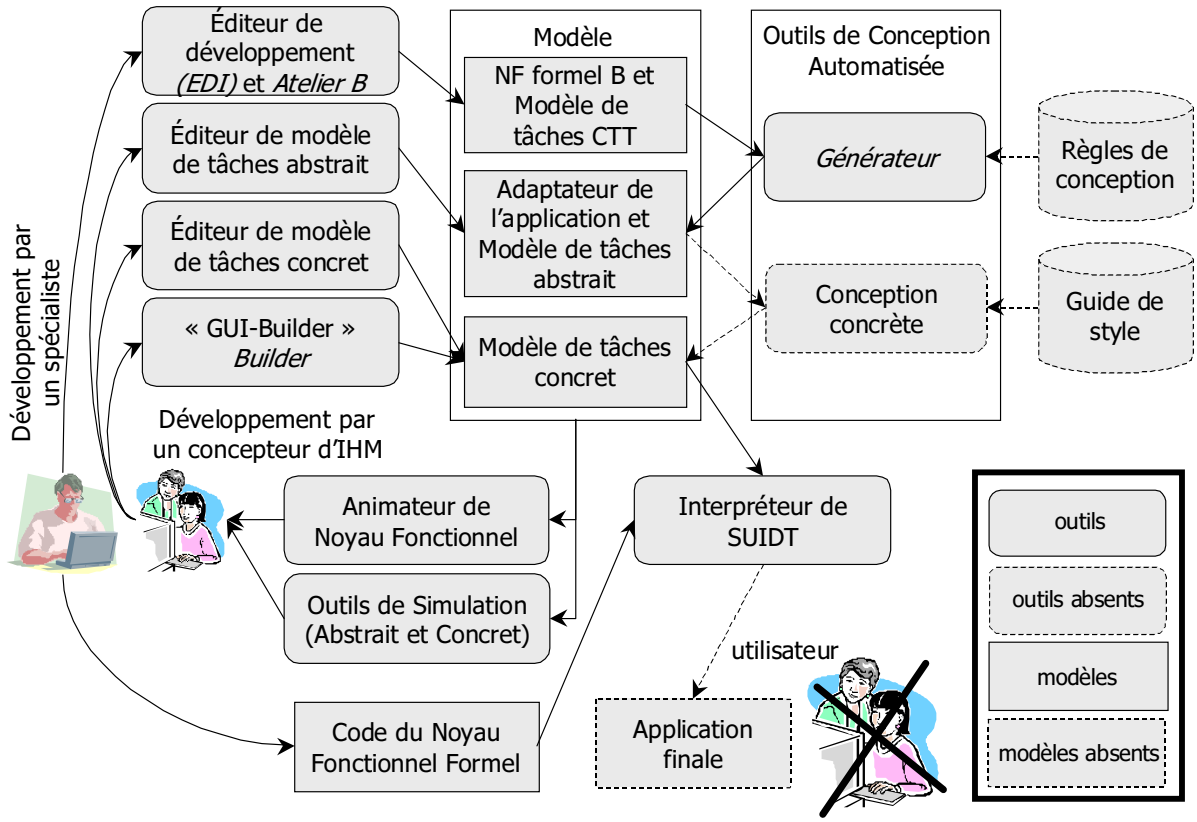

Fig. 3.28 – Architecture générique de l'environnement SUIDT.

et est construit par l'**éditeur de modèle de tâches concret** et par un **constructeur d'interfaces utilisateur** (noté *GUI-Builder* sur la figure).

Une contribution importante apportée par l'approche SUIDT se situe au niveau des outils d'évaluation, qui, dans notre démarche est obtenue par l'**animateur de noyau fonctionnel** et le **simulateur de modèle de tâches abstrait et concret**. Le premier outil permet au concepteur d'IHM de pouvoir manipuler les objets du noyau fonctionnel développement formellement via une interface interactive (obtenue par l'intermédiaire de l'outil **générateur**) sans avoir à comprendre les usages d'une méthode formelle. Un second outil permet de vérifier que le déroulement des **modèles de tâches abstrait et concret** est conforme aux besoins de l'utilisateur, de visualiser l'état de l'application sur la présentation et/ou sur l'animateur du noyau fonctionnel. Il aide à comprendre pourquoi une tâche ne peut être exécutable (état de la précondition). Par ailleurs, l'utilisation de l'**interpréteur SUIDT** permet d'alterner phase d'édition/conception sans passer par de longues étapes de compilation. Cette aspect permet aussi de garder un contexte du noyau sémantique identique à chaque phase de validation.

Pour terminer cette description, signalons que nous n'avons pas exploité de règles

de conception ni de guide de style pendant le développement (rectangles en pointillés). Nous ne nous sommes pas non plus occupé de la partie génération de l'application finale. L'application finale ne peut être testée qu'au travers de l'outil de conception SUIDT.

SUIDT et MOBI-D sont tous les deux des outils de conception puisqu'ils permettent de couvrir l'intégralité du processus de développement. Les outils que nous proposons permettent le développement du noyau fonctionnel jusqu'à l'interface utilisateur tout en décrivant le dialogue de l'application. Dans cette optique, seuls, les outils PETSHOP et TERESA sont proches et proposent des outils de validation. Dans le premier, l'utilisation des réseaux de Petri couplé à un outil d'animation de réseaux de Petri permet de tester, vérifier et valider la modélisation. Dans TERESA, il s'agit d'un simulateur de modèle de tâches qui ne vérifie que la cohérence des enchaînements des tâches.

Un autre point concerne l'interprétation de la modélisation qui permet de tester le système interactif en développement sans passer par une phase de compilation, comme par exemple le cas de MASTERMIND et PETSHOP. Notre approche par l'intermédiaire d'un noyau fonctionnel développé formellement existant permet un développement autour de ce noyau exécutable. Ainsi, le concepteur d'IHM peut modifier la présentation sans que le contexte du noyau fonctionnel soit perdu. Il est aussi possible en cours de simulation de modifier les préconditions ou le corps des tâches.

En outre, l'architecture de l'outil est définie de telle façon que le noyau fonctionnel se trouve dans une position centrale. L'ajout de modules extérieurs peut être envisagée sans avoir à reconstruire tous les développements des systèmes interactifs. En termes de modules, nous pensons à des composants graphiques (table, arbre, etc) qui enrichiraient le constructeur d'interfaces utilisateur ou bien à des outils supplémentaires permettant, par exemple, à l'utilisateur de l'assister dans sa conception.

Cependant plusieurs limites de notre approche sont à considérer suivant la présentation que nous venons de réaliser dans ce chapitre. Elles concernent la sémantique des modèles et les fonctionnalités des outils.

Limites liées à la sémantique des modèles :

1. l'approche SUIDT se base sur la modélisation de tâches (relations temporelles et structurelles) comme modèle du dialogue. Elle exploite aussi un modèle à base d'événements pour coder les transitions de passage d'une tâche interaction à une autre. Il s'agit ici d'une limite que nous avions mis en lumière pour l'approche TERESA. En contre exemple, nous pouvons donner l'outil PETSHOP qui s'appuie sur des réseaux de Petri pour décrire le dialogue du système. L'inconvénient d'utiliser un dialogue sous forme d'arbre de tâches est de se limiter principalement à des interac-

tions simples (interfaces de type formulaire par exemple) où le système ne peut être décrit par plusieurs processus (interaction) entrelacés. Nous pensons par exemple aux applications avec plusieurs modalités ;

2. la logique de programmation procédurale et l'utilisation de données de type élémentaire pour le noyau fonctionnel ne permettent pas de définir des objets structurés qui pourrait être ré-utilisés comme des composants à la manière des composants graphiques de la présentation. Par exemple, Petshop utilise une définition des objets sous forme de classes.

Limites liées aux fonctionnalités des outils :

1. l'outil SUIDT ne permet d'importer que des modèles de tâches construits à l'aide de l'outil CTTE. Pour contourner cette difficulté le concepteur d'IHM devra définir une transformation de l'analyse de tâches de la notation qu'il utilise dans la notation CTT ;
2. l'outil de simulation ne permet pas de comparer deux ou plusieurs modèles de tâches abstrait ou concret. Cet aspect pourrait être intéressant par exemple pour comparer la manière dont une tâche interaction a été concrétisée (plusieurs interfaces possibles pour une même tâche interaction) ;
3. en dernier point concerne l'absence de guide de style et de règles de conception qui pourraient par exemple aider le conception d'IHM a choisir, pour une tâche interaction, des solutions prédéfinies de concrétisation, un peu à la manière de MOBI-D.

Conclusion et perspectives

Le présent mémoire décrit les deux axes de recherche que nous avons explorés.

Nos premiers travaux se sont intéressés à prendre en compte les besoins de l'utilisateur dans les travaux de l'approche du LISI qui permettent la modélisation de la conception d'un système interactif en utilisant exclusivement la méthode B dite « classique ». La validation de tâches que nous avons intégrée, appelée approche explicite, consiste à définir explicitement des traces d'opérations issues de la conception, un peu à la manière de ce qui se fait pour le diagramme de séquence d'UML. Cependant, cette approche de validation de tâches a montré ses limites car seul l'opérateur de séquence a été employé ce qui aboutit à des modélisations de grande taille. Nous avons remarqué tout d'abord qu'une conception ascendante obligeait à reprouver à chaque composition des obligations de preuve plus complexes, puis que la modélisation du contrôleur de dialogue ne définissait qu'un ensemble d'opérations, mais pas les événements qui permettent de les déclencher ni leur ordre.

Par la suite, nous avons cherché à répondre aux limites précédentes grâce à l'utilisation de B événementiel. Nous avons ainsi défini un comportement dynamique du contrôleur de dialogue. Ce dernier a été modélisé par des systèmes de transitions et codé en B événementiel. L'approche de construction employée est dite descendante puisque que nous décomposons un système complexe pour introduire au fur et à mesure les sous-systèmes par la technique du raffinement. La validation de tâches a exploité un modèle de tâches, en l'occurrence ConcurTaskTrees. Nous avons réussi à traduire toutes les constructions de CTT par l'intermédiaire de B événementiel qui facilite le codage d'opérateurs autres que la simple séquence. Par ailleurs, nous avons également exprimé des opérations comme l'interruption et la désactivation.

Dans le chapitre 3 nous nous sommes intéressé à la définition d'une nouvelle approche par outils appelée SUIDT (Safe User Interface Development Tool). Elle se base sur une approche antérieure appelée GenBUILD qui a montré la faisabilité de l'intégration au préalable d'un noyau fonctionnel existant dans un outil de développement. SUIDT est capable de construire interactivement une application directement exécutable en conservant

le lien avec le noyau et en assurant que les propriétés de ce noyau fonctionnel sont maintenues tout au long du processus de développement de l'IHM. Cette approche appartient à la famille des systèmes basés sur modèles (Model-Based Systems). Nous l'avons présentée en décrivant la sémantique de chaque modèle et en montrant comment ces modèles sont édités, vérifiés et validés.

Le premier aspect que nous avons étudié est l'intégration d'un noyau fonctionnel développé formellement. Nous sommes parti de la méthode formelle B pour modéliser ce noyau et vérifier que les propriétés de robustesse étaient garanties. Par ailleurs, nous avons fourni un outil permettant d'exploiter le noyau fonctionnel par un concepteur d'IHM pour qu'il puisse manipuler interactivement la sémantique de l'application.

À partir du noyau fonctionnel développé formellement, nous nous sommes ensuite intéressé à la modélisation des tâches. Nous avons défini la sémantique du modèle de tâches abstrait qui s'appuie sur la notation de tâches CTT pour intégrer les besoins de l'utilisateur dans notre approche. Cette sémantique décrit la manière dont les objets du noyau fonctionnel sont manipulés. Nous avons ainsi pu expliciter clairement les préconditions et le corps de tâches à l'aide des objets du noyau. Ce modèle, conçu indépendamment de toute idée d'interaction, est alors à même de pouvoir vérifier si les besoins de l'utilisateur sont correctement pris en compte. Deux outils ont ensuite été présentés : l'éditeur et le simulateur de modèle de tâches abstrait. Nous avons montré que le simulateur permettait de dérouler l'arbre de tâches tout en conservant le lien sémantique avec le noyau fonctionnel. Par ailleurs l'absence de phase d'édition/compilation rend plus rapide la phase de validation.

Enfin, dans un dernier niveau de spécification appelé modèle de tâches concret, nous avons introduit des objets issus de la présentation. Nous avons ainsi concrétisé en terme d'interaction le modèle de tâches abstrait. Le modèle de tâches concret s'appuyant sur le formalisme CTT est en mesure d'utiliser des objets de la présentation. Finalement, nous avons fourni des outils qui permettent d'éditer et de simuler le modèle de tâches concret et un outil qui permet de construire interactivement la présentation. L'aspect original de cette approche est de pouvoir construire le modèle de tâches concret tout en respectant les propriétés du noyau fonctionnel développé formellement avec B et de permettre également d'animer le modèle de tâches concret à tout moment. Il est donc possible de tester au plus tôt le système interactif pour valider les besoins utilisateur.

Nous avons démontré, grâce à SUIDT, qu'il était possible de concevoir une application interactive en s'appuyant sur un noyau fonctionnel développé formellement, tout en prenant en compte les besoins utilisateurs, sans avoir recours à un spécialiste des techniques formelles dans la phase de conception et de réalisation de l'interface.

Dans de futurs travaux, il conviendra d'étudier en détail des aspects propres à chacune des approches puis à tenter de les combiner dans un processus global d'ingénierie des systèmes interactifs.

En ce qui concerne l'approche de modélisation et validation formelles de descriptions de l'interaction dans les IHM, nous pouvons envisager les perspectives suivantes :

- **généralisation à différentes notations et modèles**. Il serait intéressant d'étudier la généralisation de nos travaux à d'autres notations de descriptions utilisateur et de structuration de l'architecture issues du domaine de l'interaction homme-machine. Nous gardons à l'esprit que nous ne suggérons pas de nouvelles techniques ni de nouvelles notations, mais que nous assistons les concepteurs afin de les aider à appliquer ces modèles et ces notations déjà définis en fournissant des outils supports formels au plus tôt du développement ;
- **extension au domaine de la multi-modalité**. Nous pensons que l'utilisation de l'approche fondée sur le B événementiel est parfaitement adaptée au domaine de la multi-modalité. De nouvelles propriétés devront être exprimées et vérifiées par la modélisation formelle de modèles et notations spécialement adaptés à ce domaine ;
- **boîte à outils de développements B**. Nous pourrions aussi envisager d'étudier le rapprochement de nos travaux autour de la validation de tâches aux travaux sur les Interacteurs. La conception du système interactif pourrait être obtenue par composition d'Interacteurs modélisés avec la méthode B (par une approche descendante par exemple) et nous appliquerions la modélisation formelle de tâches au système interactif pour la validation de tâches ;
- **prise en compte de propriétés ergonomiques**. Nous pourrions selon le guide ergonomique des interfaces homme-machine proposé par [Van94] définir puis formaliser des propriétés ergonomiques qui n'ont pas été traitées dans cette approche.

En ce qui concerne l'approche expérimentale pour la construction d'interfaces utilisateurs sûres, les perspectives suivantes peuvent être retenues :

- **enrichissement de la sémantique du modèle de dialogue et du noyau fonctionnel**. Notre approche est adaptée à des applications simples où le dialogue est simple et où le modèle de tâches convient parfaitement à la description de ce type de dialogue. Il conviendra donc d'améliorer la capacité à modéliser des applications interactives plus complexes en ajoutant un modèle de dialogue, qui serait à même par exemple de contrôler des applications à plusieurs modalités. De même l'extension du noyau fonctionnel à des objets structurés, et la multiplication des objets induira des problèmes nouveaux ;

- **manipulation des données**. Les données visualisées sont toutes de type élémentaire, et ne font pas intervenir d'espace de visualisation propre. Une étude approfondie des types de visualisation possibles en relation avec les composants graphiques usuels est ici nécessaire ;
- **assistance de l'utilisateur dans ses choix de développement**. Notre approche s'est principalement intéressée à fournir des outils qui permettent au concepteur d'IHM de valider sa modélisation. Cependant, il nous semble intéressant d'assister ce concepteur en lui fournissant des règles de conception et des guides de styles. Ces aspects permettraient d'améliorer ce développement ;
- **incorporation des différents modèles de description des besoins utilisateurs**. Il conviendra également de s'intéresser à la prise en compte de modèles de tâches autre que la notation CTT. En suivant [LV03], nous pensons que la méta-modélisation des modèles de tâches serait une solution pour fournir en entrée de notre approche SUIDT une sémantique de notation de description de tâches utilisateurs quelconque.

De façon plus générale, concernant les deux approches, nous pouvons avancer les perspectives suivantes :

- **scalabilité des approches**. Il conviendra d'insister sur la faisabilité de ces deux approches en envisageant de les évaluer sur divers domaines d'application et plus particulièrement dans le domaine des systèmes critiques. Nous évaluerons dés lors la complexité de ces interfaces homme-machine pouvant être obtenues et la charge de travail à fournir pour atteindre ce résultat.
- **démarche méthodologique**. L'étude de l'aspect méthodologique pour définir une méthode pour chacune de ces approches et la manière de combiner ces deux approches dans un processus complet de conception constituent une perspective importante à ce travail.
  Dans le cas, par exemple de l'outil SUIDT, nous sommes partis d'une description des fonctions sémantiques du noyau fonctionnel que nous avons relié à un modèle de tâches et que nous avons concrétisé par la relation à une présentation. Il serait intéressant d'étudier la réversibilité de cette approche en partant cette fois de la présentation, la description de la tâche puis de la liaison au noyau fonctionnel à la manière de l'outil Teallach [BGM+99].
  Plus généralement, nous pensons qu'une démarche méthodologique pourrait s'inspirer de celle proposée par [Roc98] dans le sens où il utilise une approche expérimentale pour générer des spécifications formelles et des jeux de tests. L'approche SUIDT serait à même de pouvoir générer automatiquement, à partir des modélisations, des spécifications B sur lesquelles des propriétés pourraient être exprimées et vérifiées formellement. La mise en place de méthodes induira inévitablement de nouveaux problèmes.

- **plasticité des approches**. L'utilisation du modèle de tâches dans les deux approches nous amènera à explorer des perspectives liées à la plasticité des IHM [Thé01, Ste01]. Dans le cas, de l'environnement SUIDT, un premier travail serait d'approfondir la possibilité de concrétiser de plusieurs manières une même tâche interaction et de pouvoir ainsi générer différentes interfaces suivant la plate-forme utilisée (téléphone portable, assistant personnel, ...). De même, dans le cas de l'approche de modélisation et validation formelles de descriptions de l'interaction dans les IHM, nous pensons que la définition d'un modèle de tâches générique sur lequel nous proposerons plusieurs instantiations d'IHM pourrait permettre par exemple la validation d'interfaces homme-machine différentes.

Bibliographie

[AA00a] Yamine Aït Ameur. *Cooperation of formal methods in an engineering based software development process*, pages 136–155. 2000.

[AA00b] Yamine Aït Ameur. Cooperation of formal methods in an engineering based software development process. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods, IFM 2000*, pages 136–155, Dagstuhl Castle, Germany, 2000. Springer Verlag.

[AA00c] Yamine Aït-Ameur. *Développements Contrôlés de Programmes par Modélisations et Vérifications de Propriétés.* Habilitation à diriger les recherches, Université de Poitiers, 2000.

[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. 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.

[Bal94] Sandrine Balbo. *Évaluation ergonomique des interfaces utilisateur : un pas vers l'automatisation.* Doctorat d'université (phd thesis), Univerfsité Joseph Fourier, 1994.

[Bar88] Marie-France Barthet. *Logiciels Interactifs et Ergonomie, Modèles et méthodes de conception.* Dunod Informatique, Paris, 1988.

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

[BBS99] M. Biere, Birgit Bomsdorf, and Gerd Szwillus. The visual task model builder. In Jean Vanderdonkt and Angel Puerta, editors, *Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)*, pages 245–256, Louvain-la-neuve, Belgique, 1999. Kluwer Academic Publishers.

[BGM+99] Peter J. Barclay, Tony Griffiths, Jo McKirdy, Norman W. Paton, Richard Cooper, and Jessie Kennedy. The teallach tool : Using models for flexible user interface design. In Jean Vanderdonckt and Angel Puerta, editors, *Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)*, pages 139–157, Louvain-la-Neuve, Belgique, 1999. Kluwer Academics.

[BHH+88] L. Bass, E. Hardy, K. Hoyt, R. Little, and R. Seacord. The arch model : Seeheim revisited, the serpent run time architecture and dialog model. Technical Report CMU/SEI-88-TR-6, Carnegie Melon University, 1988.

[BHKN96] H. Balzert, F. Hofmann, V. Kruschinski, and C. Niemann. The janus application development environment-generating more than the user interface. In Jean Vanderdonckt, editor, *Computer-Aided Design of User iterface (CADUI'96)*, pages 183–206, Namur, Belgium, 1996. Presse Universitaire de Namur.

[BHL+95] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, B. Sacré, and J. Vanderdonckt. Towards a systematic building of software architectures : the trident methodological guide. In Philippe Palanque and Rémi Bastide, editors, *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'95)*, pages 262–278, Bonas, France, 1995. Springer-Verlag/Wien.

[Bjo87] D Bjorner. Vdm a formal method at work. In *VDM Europe Symposium'87*. Springer-Verlag, 1987.

[Boe81] B.W Boehm. Software engineering economics, 1981.

[Boe88] B.W Boehm. A spiral model of software developement and enhancement. 21(5) :61–72, 1988.

[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. Szezur. The arch model : Seeheim revisited. In *User Interface Developper's Workshop*, 1991.

[Bru98] Philippe Brun. *XTL : une logique temporelle pour le spécification formelle des systèmes interactifs*. Doctorat d'université (phd thesis), Université Paris -Sud, 1998.

[Bur92] Steve Burbeck. Application programming in smalltalk-80 : How to use the model-view-controller (mvc). Report, 1992.

[Cal98] Gaëlle Calvary. *Proactivité et réactivité : de l'assignation à la complémentarité en conception et évaluation d'interfaces homme-machine.* Doctorat d'université (phd thesis), Université Joseph Fourier, 1998. Point de vue intéressant sur les méthodes de conception des IHM en industrie : la part belle est faite au maquettage avec un cycle de style spirale.

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

[CH94] B Curtis and B Hefley. A wimp no more : The maturing of user interface engineering. *Interactions*, pages 23–34, 1994.

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

[CMN83] S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, 1983.

[Cou87] J. Coutaz. Pac, an implementation model for the user interface. In *IFIP TC13 Human-Computer Interaction (INTERACT'87)*, pages 431–436, Stuttgart, 1987. North-Holland.

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

[DFAB93] Alan Dix, Janet Finlay, Gregory Abowd, and Rusell Beale. *Human-Computer Interaction.* Prentice Hall, 1993.

[DH93] David J. Duke and Michael D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3) :25–36, 1993.

[DH95] D Duke and M. D. Harrison. Event model of human-system interaction. *IEEE Software*, 1(10) :3–10, 1995.

[Dij76] E.W. Dijkstra. In *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

[EFF96] Vadim Engelson, Drag Fritzson, and Peter Fritzson. Automatic generation of user interfaces from data structure specifications and object-oriented application models. In Pierre Cointe, editor, *ECOOP96*, volume 1098, pages 114–141, Linz Austria, 1996. Springer Verlag.

[Eng97] Robert Englander. *Java Beans - Guide du programmeur.* O'Reilly, 1997.

[Fek96a] Jean-Daniel Fekete. Les trois services du noyau sémantique indispensables à l'ihm. In *IHM'96 - 8ème journée sur l'ingénierie de l'Interaction Homme-Machine*, pages 45–50, Grenoble, France, 1996. Cépaduès Editions.

[Fek96b] Jean-Daniel Fekete. *Un modèle multicouche pour la construction d'applications graphiques interactives.* Doctorat d'université (phd thesis), Université Paris-Sud, 1996.

[FG01] Daniel Fekete and Patrick Girard. *Environnements de développement des systèmes interactifs (chapitre 1)*, volume 2 of *Interaction homme-machine pour les S.I.*, pages 23–52. Hermès Science, Paris, France, 2001.

[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.

[FS94] James D. Foley and Piyawadee "Noi" Sukaviriya. History, results and bibliography of the user interface design environment (uide), an early model-based system for user interface designe and implementations. In Fabio Paternò, editor, *Interactive Systems : Design, Specification, and Verification (DSV-IS'94)*, pages 3–14, Bocca di Magra, Italy, 1994. Springer.

[Gau95] M. C Gaudel. Formal specification techniques for interactive systems. In Philippe Palanque and Rémi Bastide, editors, *2nd Workshop on Design, Specification and Verification of Interactive Systems (DSVIS)*, Springer Computer Science, pages 21–26, Bonas, 1995. Springer-Verlag.

[GC96] Christian Gram and Gilbert Cockton. *Design Principles for Interactive Software*. Chapman & Hall, 1996.

[GEM94] Phil Gray, David England, and Steve McGowan. Xuan : Enhancing the uan to capture temporal relation among actions. Department research report IS-94-02, Department of Computing Science, University of Glasgow, February 1994.

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

[Gol84] A. Goldberg. *Smalltalk-80 : The Interactive Programming Environment*. Addison-Wesley, 1984.

[GS97] R. F. Gamboa and Dominique L. Scapin. Editing mad* task description for specifying user interfaces, at both semantic and presentation levels. In Michael Douglas Harrison and Juan Carlos Torres, editors, *Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'97)*, SpringerComputerScience, pages 193–208, Granada, Spain, 1997. Springer-Verlag.

[Gui95] 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.

[HH89] H.R Hartson and D Hix. Towards empirically derived methodologies and tools for human-computer interface development. *International Journal of Man-Machine Studies*, 31 :477–494, 1989.

[HH93] Deborah Hix and H Rex Hartson. *Developping user interfaces : Ensuring usability through product & process.* Wiley professional computing. John Wiley & Sons, inc., Newyork, USA, 1993.

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

[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.

[Jam96] Francis Jambon. *Erreurs et interruptions du point de vue de l'ingénierie de l'interaction homme-machine.* Doctorat d'université (phd thesis), Université Joseph Fourier (Grenoble 1), 1996.

[Jam02] Francis Jambon. From formal specifications to secure implementations. In *Computer-Aided Design of User Interfaces (CADUI'2002)*, pages 43–54, Valenciennes, France, 2002. Kluwer Acedemics.

[Jam03] Francis Jambon. Premiers par vers la rétro-conception en langage b d'une bibliothèque de composants graphiques. In *IHM 2003*, Caen, France, 2003.

[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.

[JGAA01] Francis Jambon, Patrick Girard, and Yamine Aït-Ameur. Interactive system safety and usability enforced with the development process. In Reed Murray Little and Laurence Nigay, editors, *Engineering for Human-Computer Interaction (8th IFIP International Conference, EHCI'01, Toronto, Canada, May 2001)*, volume 2254 of *Lecture Notes in Computer Science*, pages 39–55, Berlin, 2001. Springer.

[JGB99] Francis Jambon, Patrick Girard, and Yohann Boisdron. Dialogue validation from task analysis. In D J Duke and A Puerta, editors, *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99)*, SpringerComputerScience, pages 205–224, Universidade do Minho, Braga, Portugal, 1999. Springer-Verlag.

[Joh96] Chris Johnson. The namur principles : criteria for the evaluation of user interface notations. In M D Harrison and J Vanderdonckt, editors, *Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'96)*, Namur, Belgique, 1996. Berlin, Germany : Springer-Verlag.

[Kri59] S. Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 24 :1–14, 1959.

[Kri63] S. Kripke. Semantical considerations on modal logics. In *Philosophica Fennica - Modal and Many-Valued Logics*, pages 83–94, 1963.

[Lie01] Henry Lieberman. *Your Wish is my command.* Morgan Kaufmann, 2001.

[LS96] F. Lonczewski and S. Schreiber. The fuse-system : an integrated user interface design environment,. In Jean Vanderdonckt, editor, *Computer-Aided Design of User iterface (CADUI'96)*, pages pp. 37–56, Namur, Belgium, 1996.

[LV03] Quentin Limbourg and Jean Vanderdonckt. *Comparing Task Models for User Interface Design (Chapter 6).* Lawrence Erlbaum Associates, 2003.

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

[MMM+97] Brad. A. Myers, Robert. G. McDaniel, Robert. C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The amulet environment : New models for effective user interface software development. *IEEE Transactions on Software Engineering,*, 23(6) :347–365, 1997.

[MPS03] Giulio Mori, Fabio Paternò, and Carmen Santoro. Tool support for designing nomadic applications. In *Intelligent User Interfaces (IUI'2003)*, pages 141–148, Miami, Floride, 2003.

[MR84] J McDermid and K Ripkin. *Life cycle support in the ADA environment.* Cambridge University Press, 1984.

[MR92] A. Brad Myers and M. B. Survey Rosson. Survey on user interface programming. In *Human Factors in Computing Systems (CHI'92)*, pages 195–202, Monterey, USA, 1992. ACM/SIGCHI.

[Mye95] Brad A. Myers. User interface software tools. *ACM Transactions on Computer Human Interaction*, 2(1) :64–103, 1995.

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

[NC91] Laurence Nigay and Joëlle Coutaz. Building user interfaces : Organizing software agents. In *ESPRIT'91 Conference*, 1991.

[Nig94] Laurence Nigay. *Conception et Modélisation Logicielle des Systèmes Interactifs : Application aux Interfaces Multimodales.* Doctorat d'université (phd thesis), Université Joseph Fourier, 1994.

[Nor86] D. Norman. *User Centered System Design.* Lawrence Erlbaum Associates, 1986.

[Ols86] R. Olsen. Mike : the menu interaction kontrol environment. *ACM Transaction on Graphics*, 5(3) :318–344, 1986.

[Ols89] R. Olsen. A programming language basis for user interface management. In ACM Transaction, editor, *Conference on Human Factors in Computing Systems*, pages 171–176. ACM Press, 1989.

[Ous94] John Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Ous01] Sy Ousmane. *Spécification comportementale de composants CORBA*. Doctorat d'université (phd thesis), Université de Toulouse 1, 2001.

[Pal92] Philippe Palanque. *Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur*. Doctorat d'université (phd thesis), Université de Toulouse I, 1992.

[Pat99] Guillaume Patry. *Contribution à la conception du dialogue Homme-Machine dans les applications graphiques interactives de conception technique : Le système GIPSE*. Thèse, Université de Poitiers, 1999.

[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.

[PdS00] Paulo Pinheiro da Silva. User interface declarative models and development environments : A survey. In Philippe Palanque and Fabio Paterno, editors, *7th Eurographics workshop on Design, Specification and Verification of Interactive Systems DSVIS 2000*, pages 207–226, Limerick, Ireland, 2000. Springer.

[PE98] A. Puerta and J. Eisenstein. Interactively mapping task model to interfaces in mobi-d. In Panos Markopoulos and Peter Johnson, editors, *Eurographics Workshop on Design, Specification and Validation of Interactive Systems (DSV-IS'98)*, volume Proceedings, pages 261–274, Abingdon, UK, 1998.

[PF92] Fabio Paternò and Giorgio P. Faconti. *On the LOTOS use to describe graphical interaction*, pages 155–173. Cambridge University Press, 1992.

[Pfa85] Günther E Pfaff, editor. *User Interface Management Systems, Proceedings of the Workshop on User Interface Management Systems held in Seeheim*. Eurographic Seminars. Springer-Verlag, Berlin, 1985.

[Pie91] Guy Pierra. *Les bases de la programmation et du Génie Logiciel*. Dunod informatique Paris, 1991.

[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.

[PS02] Fabio Paternò and Carmen Santoro. One model, many interfaces. In Christophe Kolski and Jean Vanderdonckt, editors, *Computer-Aided Design of User Interfaces (CADUI'2002)*, pages 143–154, Valenciennes, Frances, 2002. Kluwer Academics.

[Pue96] Angel Puerta. The mecano project : comprehensive and integrated support for model-based interface development. In Jean Vanderdonckt, editor, *Computer-Aided Design of User interface (CADUI'96)*, pages 19–35, Namur, Belgium, 1996. Presse Universitaire de Namur.

[PV02] Costin Pribeanu and Jean Vanderdonckt. Exploring design heuristics for user interface derivation from task and domain models. In Christophe Kolski and Jean Vanderdonckt, editors, *Computer-Aided Design of User Interfaces (CADUI'2002)*, pages 103–110, Valenciennes, France, 2002. Kluwer Academics.

[Roc98] 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.

[SB85] Christophe Sibertin-Blanc. High level petri nets with data structure. In *6th European Workshop on Petri Nets and Applications*, Espoo, Finland, 1985.

[SB01] Dominique Scapin and J-M Christian Bastien. *Analyse des tâches et aide ergonomique à la conception : l'approche MAD* (chapitre 3)*, volume 1 of *Interaction homme-machine pour les S.I.* Hermès Science, Paris, France, 2001.

[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.

[She89] A Shepherd. *Analysis and training in information technology tasks*, pages 15–55. Books in Information Technology. Ellis Horwood, Chichester, USA, 1989.

[Shn98] Ben Shneiderman. *Designing the User Interface.* Addison-Wesley, 3 edition, 1998.

[Shu86] K. Shumerck. Macapp : An application framework. *Byte*, 11(8) :189–193, 1986.

[SLN93] P. Szekely, P. Luo, and R. Neches. Beyond interface builders : Model-based interface tools. In *InterCHI93*, pages 383–390, 1993.

[SPG89] D L Scapin and C Pierret-Golbreich. Mad : Une méthode analytique de description des tâches. In *Colloque sur l'Ingénierie des Interfaces Homme-Machine (IHM'89)*, pages 131–148, Sophia-Antipolis, France, 1989.

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

[SSC+95] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. Declarative interface models for user interface construction tools : the mastermind approach. In Leonard J Bass and Claus Unger, editors, *IFIP TC2-WG2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI'95)*, pages 120–150, Grand Targhee Resort (Yellowstone Park), USA, 1995. Chapman and Hall.

[Ste01] Constantine Stephanidis. *User Interfaces For All.* 2001.

[Sys84] ISO Information Processing Systems. Definition of the temporal ordering specification language lotos. Technical Report TC 97/16 N1987, ISO, 1984.

[Sze96] P. Szekely. *Retrospective and challenge for Model Based Interface Development*, pages 1–27. SpringerComputerScience. Springer-Verlag, Namur, Belgium, 1996.

[Tab01] Dimitri Tabary. *Contribution à TOOD, Une méthode à base de modèles pour la spécification et la conception des systèmes interactifs.* Thèse, Université de Valenciennes, 2001.

[Tar93] Jean-Claude Tarby. *Gestion Automatique de Dialogue Homme-Machine à partir de spécification conceptuelles.* Doctorat d'université (phd thesis), Université de Toulouse I, 1993.

[Tex00] Guillaume Texier. *Contribution à l'ingénierie des systèmes interactifs : Un environnement de conception graphique d'applications spécialisées de conception.* Thèse, Université de Poitiers, 2000.

[Thé01] David Thévenin. *Adaptation en Interaction Homme-Machine : le cas de la plasticité.* Thèse de doctorat, Université Joseph Fourier, 2001.

[Van94] Jean Vanderdonckt. *Guide ergonomique des interfaces homme-machine*, volume 1. Presses Universitaires de Namur, 1994.

[Van99] Jean Vanderdonckt. Development milestones towards a tool for working with guidelines. *Interacting with Computers*, 12(2) :81–118, 1999.

[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.

[WBB+90] Charles Wiecha, William Bennet, Stephen Boies, John Gould, and Sharon Greene. Its : a tool for rapidly developing interactive applications. *ACM Transactions on Information Systems*, 8(3) :204–236, 1990.

[Win70] W. Royce Winston. Managing the development of large software systems : Concept and techniques. In IEEE Computer Society Press, editor, *WESCOn*, pages 1–9, Los Alamitos, 1970.

[Woo70] W. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10) :591–606, 1970.

Publications liées à ce mémoire de thèse

Conférences internationales avec comité de lecture

- 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, *IEEE - The 2003 International Conference on Software Engineering Research and Practice - SERP 2003*, volume 2, pages 732-738, Las Vegas, Nevada USA, 2003. CSREA Press.
- 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.
- Mickaël Baron and Patrick Girard. SUIDT : A task model based gui-builder. In Costin Pribeanu and Jean Vanderdonckt, editors, *TAMODIA : Task MOdels and DIAgrams for user interface design*, volume 1, pages 64-71, Romania, Bucharest, 2002. Inforec Printing House.
- Mickaël Baron and Patrick Girard. Bringing robustness to end-user programming. In IEEE, editor, *2001 IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 142-149, Stresa, Italy, 2001. Entergraphica.

Conférences nationales avec comité de lecture

- Mickaël Baron. Intégration d'un modèle de tâche dans une démarche sûre de construction d'interface. In Girard Patrick and Baudel Thomas, editors, *14ř Conférence Francophone sur l'Interaction Homme-Machine (IHM'2002)*, pages 73-80, Poitiers, 2002. ACM Press.
- Mickaël Baron and Patrick Girard. Construction interactive d'application à partir du

noyau fonctionnel. In D Scapin and E Vergisson, editors, *Ergonomie et informatique avancées (Ergo-IHM'2000)*, pages 85-93, Biarritz, France, 2000. ESTIA.

Autres formes de publications

- Mickaël Baron and Patrick Girard. SUIDT : Safe user interface design tool (demo). In *International Conference on Intelligent User Interfaces Computer- Aided Design of User Interfaces*, pages 350-351, Madeira, Portugal, 2004. ACM Press.
- Mickaël Baron and Patrick Girard. SUIDT : Un outil de construction d'interfaces utilisateurs sûres (demo). In *15ème Conférence Francophone sur l'Interaction Homme-Machine*, volume 1, pages 198-201, Caen, 2003. ACM Press.
- Mickaël Baron. De la conception à la construction d'application sûre (rencontre doctorale). In Girard Patrick and Baudel Thomas, editors, *14ř Conférence Francophone sur l'Interaction Homme-Machine (IHM'2002)*, volume 1, pages 285-286, Poitiers, 2002. ACM Press.
- Mickaël Baron and Patrick Girard. Vers un développement sûr d'applications interactives (poster). In Jean Vanderdonckt, Ann Blandford, and Alain Derycke, editors, *IHM-HCI 2001*, volume 2, pages 155-158, Lille, France, 2001. Cépaduès-Éditions.

Annexe A

Présentation de la notation ConcurTaskTrees

Nous présentons, dans cette annexe, la notation CTT [Pat01] (ConcurTaskTrees) et l'outil CTTE [PMG01] (ConcurTaskTrees Environment) qui permet d'éditer des modèles de tâches CTT et d'en vérifier la consistance. Nous illustrerons cette notation à travers l'étude de cas du convertisseur francs/euros et compteur.

A.1 La sémantique de la notation CTT

CTT est une notation de spécifications de tâches fondée sur une structuration hiérarchique des tâches. Une tâche CTT définit comment un utilisateur peut atteindre un but en utilisant une application interactive. Selon la classification des tâches proposées par [Bar88], une tâche CTT est identifiée comme une tâche prescrite c'est-à-dire qu'elle est conçue par celui qui en commande l'exécution.

Une tâche CTT est définie par différentes caractéristiques résumées ci-dessous :

- un *nom* identifie la tâche ;
- une *catégorie* ;
- un ensemble d'objets ;
- des caractéristiques associés aux opérateurs temporels.

A.1.1 Les catégories des tâches

CTT inclut quatre catégories de tâches qui aident à construire l'arbre des tâches.

Les **tâches utilisateur**, désignées par le repère 1 de la figure A.1, sont accomplies par l'utilisateur. Il s'agit d'activités cognitives ou physiques indépendamment de toute interaction sur le système.

Les **tâches application**, désignées par le repère 2 de la figure A.1, sont effectuées complètement par le système et sont utilisées pour rendre compte de l'état du système.

Les **tâches interaction**, désignées par le repère 3 de la figure A.1, sont réalisées par les interactions de l'utilisateur avec le système.

Les **tâches abstraite**, désignées par le repère 4 de la figure A.1, sont raffinées par les trois catégories précédentes. Il s'agit de tâches de plus haut niveau sans savoir exactement la portée de son action.

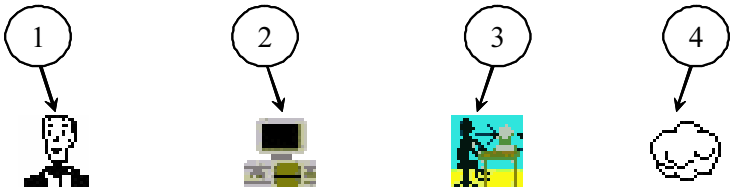

Fig. A.1 – Représentation graphiques des catégories de tâches CTT.

A.1.2 Les objets des tâches

Les objets des tâches sont des entités manipulées pour accomplir une tâche. Elles sont regroupées en deux catégories :

- la première concerne les objets dits *perçus* qui peuvent représenter des objets pour le retour d'information (champs de textes, arbre, etc) ou des techniques d'interactions pour agir sur le système (bouton, zone de saisie) ;
- les objets *application* sont des entités du noyau fonctionnel qui peuvent être associées à des objets perçus pour retourner l'état du noyau.

Cependant, nous ne nous attarderons pas à décrire les objets puisque que cette partie de la sémantique de CTT n'est pas précisément définie. L'auteur ne décrit pas précisément le rôle de chacun des objets et la manière dont s'effectue la modélisation des objets. Selon

notre expérience à partir de l'outil CTTE (puisque les articles ne détaillent pas ce point) il s'agit d'une description informelle où les objets sont décrits au moyen d'une simple chaîne de caractères sans relation directe entre eux.

A.1.3 Les opérateurs temporels

Les relations temporelles entre deux tâches sont décrites par des opérateurs temporels issus de Lotos [Sys84]. La notation CTT dispose de plusieurs opérateurs que nous décrivons. Considérons tout d'abord la tâche $T_0 ::= T_1$ *op* T_2 où T_0, T_1 et T_2 sont de catégories quelconques puisque la catégorie d'une tâche n'a pas d'influence sur les relations temporelles.

- $T_1 >> T_2$ **Activation** : il s'agit de l'opérateur de séquence. La tâche T_2 n'est activable que si T_1 est activée et a terminé son exécution.
- $T_1 [] T_2$ **Choix** : deux tâches sont disponibles, mais une seule tâche est activable. Si T_1 est activée alors T_2 n'est plus activable et réciproquement.
- $T_1 \parallel T_2$ **Parallèle** : T_1 et T_2 sont activables en même temps, il s'agit de l'opérateur de parallélisme.
- $T_1 |=| T_2$ **Ordre indépendant** : T_1 et T_2 sont activables dans n'importe quel ordre.
- $T_1 | > T_2$ **Interruption** : si T_1 est en cours de traitement, la tâche T_2 peut l'interrompre. Une fois T_2 activée, T_1 reprend son traitement.
- $T_1 [> T_2$ **Désactivation** : si T_1 est en cours de traitement, la tâche T_2 peut la désactiver.
- $T_1 [] >> T_2$ **Activation avec passage d'information** : identique à l'opérateur d'activation, avec un passage d'information de T_1 vers T_2 au moment d'activation.
- $T_1 |[]| T_2$ **Synchronisation** : synchronisation des objets des tâches T_1 et T_2.

Dans nos travaux, nous avons considéré que les opérateurs **activation avec passage d'information** et **synchronisation** étaient similaires aux opérateurs **activation** et **parallèle**.

A.1.4 Caractéristiques de tâches

Aux opérateurs temporels, s'ajoutent des caractéristiques temporelles aux tâches. Considérons une tâche T_1 de catégorie quelconque.

- T_1^* **Itération** : la tâche T_1 se répète indéfiniment tant qu'une autre tâche ne la désactive pas. L'expression $T_1^* >> T_2$ n'est pas autorisée car T_2 n'est jamais attei-

gnable ;
- T_1^N **Itération finie** : la tâche T_1 re répète N fois à moins qu'une autre tâche ne la désactive ;
- $[T_1]$ **Optionnelle** : elle indique que la tâche T_1 n'est pas obligatoirement exécutable. Les tâches à gauche ou à droite des opérateurs $|>$, $[>$ et $[]$ ne peuvent être optionnelles.

A.2 L'outil CTTE

L'outil CTTE[1] (ConcurTaskTrees Environment) fournit un ensemble d'outils pour décrire, dans la notation CTT, des modèles de tâches. Il permet de vérifier automatiquement la syntaxe d'un arbre CTT (combinaison de tâches avec des opérateurs), information sur les modèles de tâches, vérification du comportement du déroulement d'un modèle, construction de scénarii, etc.

A.2.1 L'éditeur de modèle de tâches

La figure A.2 présente l'éditeur de modèles de tâches CTT. Nous illustrons cette capture d'écran par l'étude de cas traitée dans ce mémoire. Sur la gauche de l'outil se trouve une palette d'outils qui proposent les quatres catégories (repère 1) de tâches, les opérateurs temporels (repère 2) puis les caractéristiques de tâches (repère 3) triés par ordre chronologique. Sur la partie supérieure, une palette d'outils (repère 4) pour paramétrer le modèle de tâches en cours et enfin sur la partie centrale (repère 5) le modèle de tâches à éditer. C'est un outil graphique qui permet à un utilisateur final d'éditer et d'exploiter des modèles de tâches CTT.

L'outil CTTE n'exploite pas complètement la sémantique de la notation CTT. Par exemple l'itération finie n'est pas prise en compte.

A.2.2 Le simulateur de modèle de tâches

Le simulateur de modèles de tâches de CTTE permet de vérifier le comportement du déroulement d'un modèle qui se base sur les contraintes de précédence, c'est-à-dire la fonction des opérateurs temporels. La figure A.3 montre cet outil. La partie centrale (repère 1)

[1] Adresse de téléchargement : http ://giove.cnuce.cnr.it/ctte.html

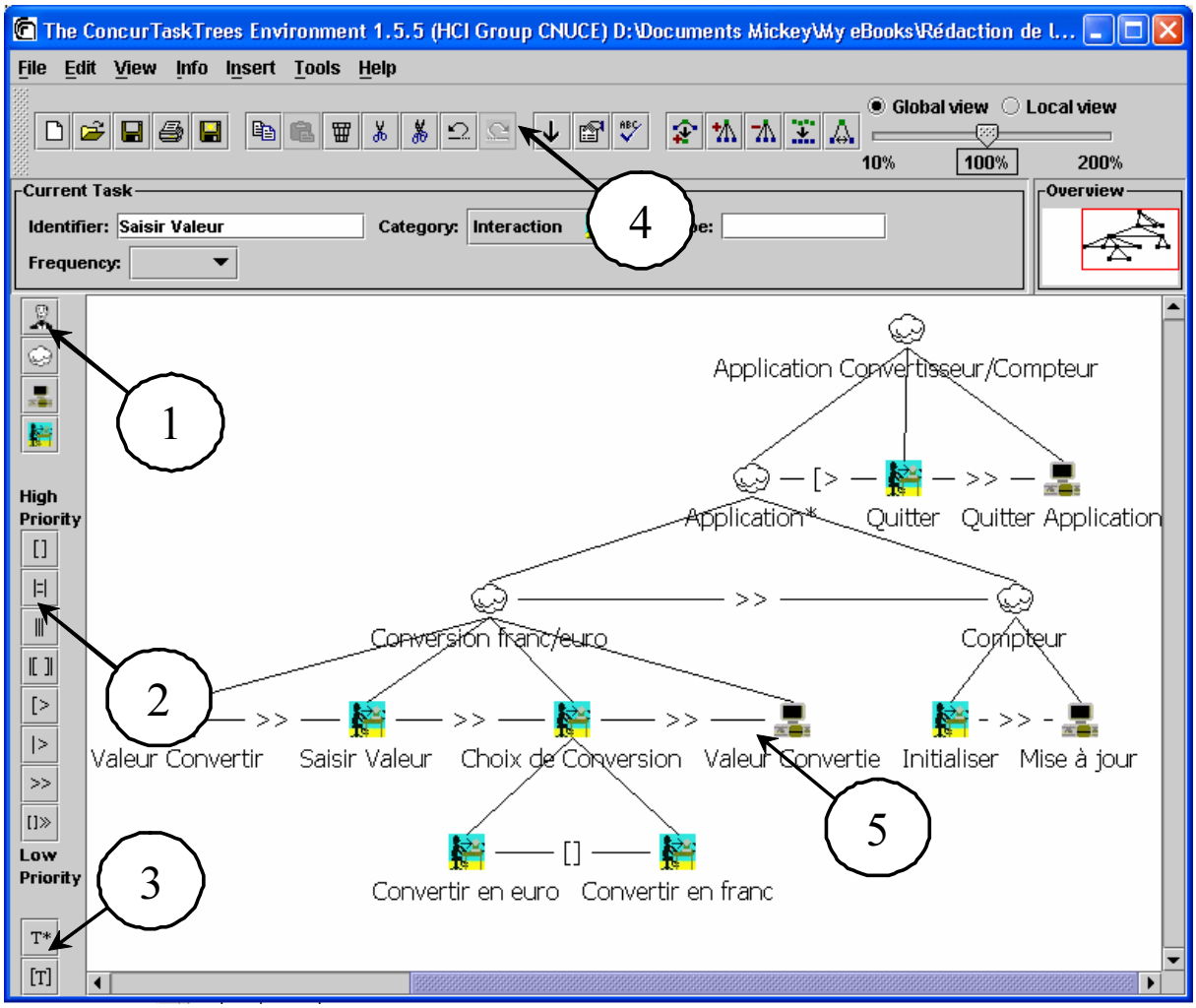

FIG. A.2 – Capture d'écran de l'éditeur de modèles de tâches CTT.

visualise le modèle de tâches en cours de simulation. La partie de droite fournit les tâches qui sont activables à l'instant donné (repère 2), et finalement des options (repère 3) qui permettent entre autres de sauvegarder un scénario, de revenir dans un état précédent, etc.

La simulation peut être effectuée par deux déroulements. Si l'utilisateur définit lui même un chemin à l'intérieur de l'arbre de tâche on parle de déroulement explicite. Au contraire, le déroulement implicite traduit un déroulement par un scénario. Les scénarii sont obtenus après un déroulement explicite.

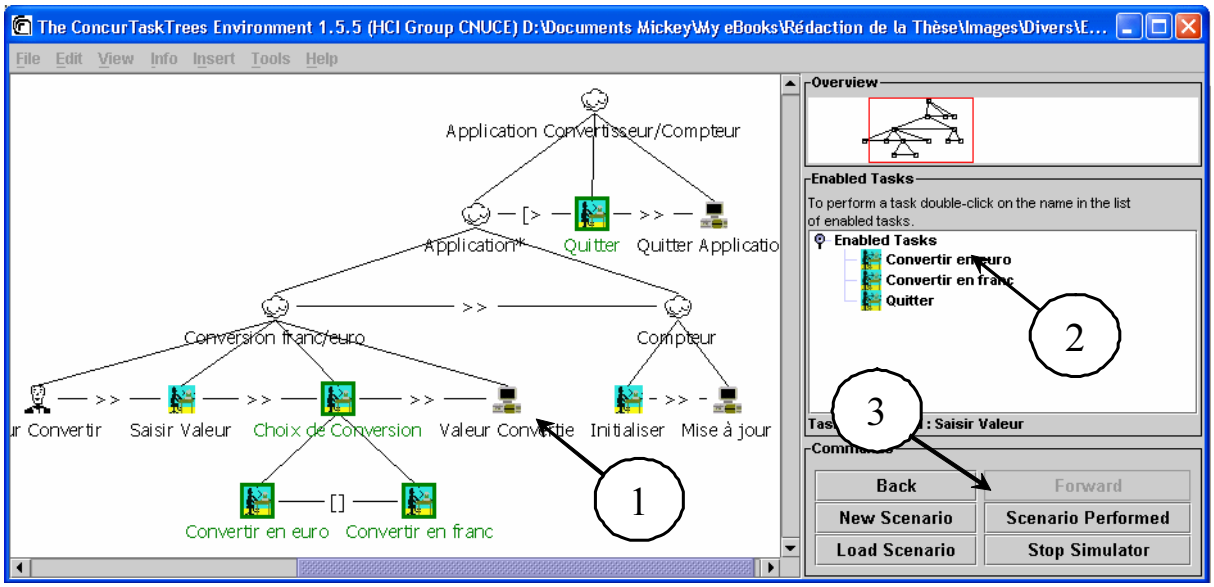

FIG. A.3 – Capture d'écran du simulateur de modèles de tâches CTT.

Vers une démarche sûre du développement des Interfaces Homme-Machine

Présentée par

Mickaël Baron

Directeurs de Thèse

Yamine Aït-Ameur et Patrick Girard

Résumé. Les interfaces homme-machine (IHM) constituent une part indispensable dans la quasi-totalité des systèmes informatiques. Le recours à des notations de description des IHM, et à des modèles de spécification, de développement, de vérification et de validation devient indispensable pour assurer que le système satisfait les propriétés définissant la notion d'utilisabilté. Aujourd'hui, on peut considérer que deux approches exploitant les modèles du domaine de l'IHM peuvent être mises en parallèle pour la vérification de propriétés : les approches fondées sur le développement formel et les approches fondées sur la définition d'outils. Malgré des avancées intéressantes, aucune d'elles n'est encore parvenue à s'imposer. Nous proposons dans cette thèse deux nouvelles approches permettant le développement sûr d'interfaces homme-machine, fondées sur une même méthode formelle (la méthode B). La première fondée sur le développement formel permet, d'intégrer des notations et des techniques hétérogènes du domaine de l'IHM dans une seule et unique méthode formelle (la méthode B), afin d'exprimer, vérifier et valider des propriétés du système interactif. La seconde, fondée sur la définition d'outils, (SUIDT), permet de concevoir de manière interactive le dialogue entre un noyau fonctionnel développé formellement en B et une présentation graphique de l'interface, tout en garantissant le respect des propriétés exprimées à la fois dans le noyau fonctionnel et au niveau des tâches de l'utilisateur.

Mots-Clés : interaction homme-machine, méthodes formelles, méthode B, systèmes basés sur modèles, validation de tâches.
