THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par **l'Institut Supérieur de l'Aéronautique et de l'Espace**
Spécialité : Informatique

Présentée et soutenue par **Alexandre Cortier**
le 6 juin 2008

Contribution à la validation formelle d'applications interactives Java

---

JURY

M^{me} Laurence Nigay, présidente du jury
M. Yamine Aït-Ameur
M. Bruno d'Ausbourg, directeur de thèse
M. Christophe Kolski, rapporteur
M. Dominique Méry
M. Ioannis Parissis, rapporteur

École doctorale : **Mathématiques, informatique et télécommunications de Toulouse**

Unité de recherche : **Équipe d'accueil ISAE-ONERA MOIS (ONERA-DTIM, centre de Toulouse)**

Directeur de thèse : **M. Bruno d'Ausbourg**

THÈSE

présentée en vue de l'obtention du titre de

DOCTEUR
de
l'Université de Toulouse
délivrée par
l'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)
École Doctorale MITT

Spécialité : Informatique

par

Alexandre CORTIER

Contribution à la Validation Formelle d'Applications Interactives Java.

JURY :

L.	**NIGAY**	(Pr., LIG, Grenoble)	**Président du Jury**
B.	**d'AUSBOURG**	(IR, ONERA-DTIM, Toulouse)	**Directeur de Thèse**
Y.	**AÏT-AMEUR**	(Pr., LISI-ENSMA, Poitiers)	**Co-Directeur de Thèse**
C.	**KOLSKI**	(Pr., LAMIH, Valenciennes)	**Rapporteur**
I.	**PARISSIS**	(MdC, LIG, Grenoble)	**Rapporteur**
D.	**MÉRY**	(Pr., LORIA, Nancy)	**Examinateur**

Thèse préparée au sein du Département Traitement de l'Information et Modélisation (DTIM)
de l'ONERA-CERT (Toulouse).

Remerciements

Je tiens à remercier en premier lieu **Ioannis Parissis** et **Christophe Kolski** d'avoir accepté d'être les rapporteurs de mes travaux de thèse, ainsi que **Dominique Méry** et **Laurence Nigay** qui ont accepté de se joindre à mon jury de thèse.

Je tiens à remercier **Yamine Aït-Ameur**, sans qui cette thèse n'aurait pas vu le jour. Merci à toi, Yamine, de m'avoir pris sous ton aile pour mes travaux de DEA puis de m'avoir suivi pendant ces trois années de thèse. Ton hyperactivité et ta rigueur m'ont permis de prendre un peu plus conscience de mon tempérament de rêveur. Certes, la distance (physique !) a parfois posé quelques soucis : lors de tes passages éclair sur Toulouse, il a bien souvent fallu courir pour te rattraper dans les couloirs ! Malgré cela, tu as toujours été présent aux moments importants et toujours disponible par téléphone ou par mail pour m'écouter, me conseiller et me demander de retravailler les articles ! J'apprécie ta gentillesse, ton énergie communicative et ton caractère de "bon vivant". Encore une fois : merci.

Je ne peux aller plus loin sans remercier **Bruno d'Ausbourg**, alias mon "chef". Je risque de me faire taper sur les doigts pour ce surnom... Le passage du surencadrement "méthode Yamine" à l'encadrement "méthode Bruno" a été quelque peu déconcertant à mon arrivée en thèse. "Plonger dans le grand bain" est sans doute la formule consacrée : Bruno a d'ailleurs bien cru à ma noyade les premiers temps ! Dès le début de la thèse, tu m'as laissé une autonomie quasi complète. Si cette autonomie a été difficile à gérer en début de parcours, je dois bien avouer que j'y ai pris goût rapidement et qu'elle m'a beaucoup enseigné. Bruno, je te remercie pour la confiance que tu as su m'accorder tout au long de cette thèse, pour ton écoute, ton ouverture d'esprit, et les quelques coups de pieds savamment placés qui m'ont permis de rebondir. Trois années en ta présence m'ont permis de découvrir un homme intègre, juste, honnête et chaleureux. Encore une fois : merci.

Je tiens également à remercier tous **les membres du DTIM de l'Onera** qui m'ont accueilli durant ces trois années (et demi...). Merci tout d'abord à **Jacques Cazin** de m'avoir accueilli au sein du DTIM pour cette thèse. Merci à **Frédéric Boniol** qui m'a permis d'encadrer les cours de Lustre à l'INSA et à Supaéro et avec qui je vais avoir le plaisir de travailler au cours de mon post-doc... Merci à **Guy Durrieu** pour son aide précieuse concernant l'analyse de code Java. Avec une épine de moins sous le pied, on avance plus vite ! Merci à **Claire Pagetti** pour ses conseils d'ex-thésarde...

Merci à, **Josette Brial**, la râleuse et chocolat-phage de l'étage : à une "époque", tu imposais une dîme de carrés de chocolat aux thésards du DTIM. Nos discussions autour d'un "coffee and cigarettes" et ton langage fleuri me manquent déjà ! Merci de m'avoir soutenu, écouté, conseillé et d'avoir su jouer de la semelle pour que je remonte en selle lorsque le moral était en baisse. Et promis : je te donnerai de mes nouvelles...

Merci aussi à **Christiane Payrau** qui s'est toujours révélée très efficace pour les petits soucis administratifs ! Croyez-moi futurs thésards, vous êtes entre de bonnes mains. Merci à **Jean-Loup Bussenot**, dit "Bubu", qui a dépanné plus d'une fois ma machine. C'est toujours un plaisir de discuter avec toi ! Merci également à **Pierre Bieber**, **Christelle Seguin**, **Patrice Cros**, **Pierre Michel**, **Jean Yves Rousselot**...

Au cours de ces trois années de thèse, j'ai eu l'occasion de faire la connaissance de nombreuses personnes dans le cadre de mes activités d'enseignement. Parmi elles, je souhaite remercier **Claude Baron**, responsable de l'enseignement temps-réel à l'INSA, qui m'a accueilli comme vacataire d'enseignement. Je remercie également "l'équipe marocaine" : **Bernard Pradin**, **Ernesto Exposito** et **Daniela Dragomirescu**. Cette semaine d'enseignement à Marrakech avec vous restera un très, très bon souvenir... Merci à **Maria Zrikem** pour son accueil très chaleureux à l'ENSA de Marrakech. Merci également à **Régine Leconte** de Supaéro (oups... ISAE) que j'ai eu l'occasion de connaître lors de mes TP !

J'ai eu également l'occasion de faire la connaissance avec quelques générations de thésards de l'ONERA. Je commencerai par remercier **Nil Geisweiler** qui m'a fait découvrir qu'il existait bien plus rêveur que moi ! J'ai beaucoup aimé nos discussions dont les sujets étaient des plus variés : informatique, mathématique, musique, philosophie... Merci de m'avoir fait l'honneur d'être ton témoin à ton mariage. J'espère vous revoir bientôt toi, **Yana** et votre enfant ! Dans la lignée des anciens, merci à **François Carcenac**, **Christophe Kheren** et **Rémi Delmas**...

Nous arrivons maintenant aux thésards de ma génération. Sans eux, jamais je n'aurais appris la coinche : c'est vous dire combien je suis redevable ! Merci **Laurent** pour ta bonne humeur journalière et tes coups de main sous Linux. On repart quand en Suède ? Bon, il y a tout de même un reproche que je peux te faire : pourquoi m'as-tu parlé de Tower Defense alors que j'étais en pleine rédaction ?

Merci à **Florian** pour ces parties de coinches interminables (on fait équipe pour le prochain tournoi ?), nos discussions philosophico-bacchusiennes, nos petits buffs de jazz manouche (plus manouche que Jazz) et j'en passe... Conserve ton énergie, ta curiosité et ton envie de construire un monde meilleur : on en a besoin !

Merci **Eric** pour ta couleur de cheveux, ton humour et ta "grande gueule" ;). Ton digicode est toujours en panne ?

Merci à **Maud**, compagne de coinche exceptionnelle ! Ne le dis pas à Flo, mais je préférerais que nous fassions équipe tous les deux pour le prochain tournoi...

Merci à **Manu** d'avoir Musclé nos conversations. Merci à **Thomas** le stagiaire (ah bon tu es en thèse ?) pour les petites pauses que tu venais faire dans le bureau.

Merci à **Julien** d'avoir co-encadré les TPs avec moi (je te jure qu'il marchait ce programme Lustre !), merci à **Stéphanie** pour sa bonne humeur et son très bon Calva que j'attends toujours de goûter (promis lors de ma soirée de thèse, je m'en souviens...) !

Merci aux autres coincheurs sans qui les pauses déjeuners auraient été bien plus tristes : **Domingo**, **Hélène**, **Cédric**, **Sophie** et ceux que j'oublie !

Je remercie également quelques habitants ou ex-habitants de la planète LEGOS dont j'ai fait la connaissance pendant ces trois années. Merci en premier lieu à **Momo** sans qui je n'aurais pas fait votre connaissance. Merci à **Benoit** pour les sorties escalade et les récits de ses pérégrinations (c'est pas faux...), à **Mathieu** pour son humour, à **Boubou** pour ces news de la Jet-Set (ça y est tu es pote avec Manaudou sur Facebook ?), à **Yoyo** et **Maria** alias "les Bisnounours"... Merci également à **Chacha** et **Julien** (alias Patrick) pour l'organisation des soirées poker, pour leur bonne humeur et leur gentillesse (vous nous manquez !).

Merci à **Guillaume** pour sa "fiffon attitude" et pour l'argent que j'ai pu mettre de côté grâce à lui lors des soirées poker... Merci à toi et **Marie** de m'avoir accepté dans le club très fermé des sessions Saint-Pé. Une dernière chose : vous êtes des parents exceptionnels !

Je ne peux que clore cette parenthèse océanographique par un énorme merci à **Julien**, alias Dédé, compère des sorties escalades et des soirées fructoses ! Merci à toi pour ton humour qui passe du sphincter au jeu de mots ciselé en moins de deux, pour tes sushis et tes brocolis, pour tes anecdotes et tes imitations essseptionnelles, pour ta tête en l'air et tes gaffes... Merci à **Caro** d'être rentrée à Toulouse et de continuer à supporter Ju (pwaaaât...) : je pense que je parviendrai à te battre à un jeu de cartes... un jour. J'inventerai de nouvelles règles s'il le faut. Ju et Caro, où que le vent nous mène, j'espère que nous continuerons à nous voir !

J'aimerais également remercier les fadas de la "bande à Mariole" que j'ai rencontrés un beau jour dans une grange... Merci à **Clermont** pour ses saucissons, ses coups de fil interminables, sa gentillesse et son humour... Merci à **Cédric** et **Marie** pour les week-ends dans le Lot, pour les cabécous et les fins de bouteilles ! Merci à **Bruno** & **Émilie** pour les 44 préparés en 30 minutes et pour les animations de mariage. Merci à **Justine** pour ses coups de gueule, sa gentillesse et son énergie (passe le bonjour à tes filles). Merci à **P'tite Marie** pour le week-end pâté/foie gras dans les Landes... Merci aussi à **Cheep** et **Cheepette** : promis, nous passerons sur Grenoble pour arpenter les montagnes ! Merci à **Guigui** et **Guiguette** (le plus beau mariage traditionnel auquel j'ai pu assister...). Merci également à **Dana et Dannette** (votre mariage aussi valait le détour : D. I. A. N. E !)...

Encore des remerciements, mais cette fois dans le désordre : merci à **Yann** de m'avoir accompagné à la Dent d'Orlu sur les "Enfants de la dalle" (que nous n'avons malheureusement pas terminé...). Quand remettons-nous cela? Merci aux anciens de l'ENSMA que je n'oublie pas même si nous nous voyons au compte goutte... Spéciale dédicace aux Kaaârfmen bien évidemment ! Merci à **Sabrina** pour le bout de chemin que nous avons parcouru ensemble : ne reste pas trop longtemps aux États-Unis Sabi ! Merci à **Loé** (le Tiken-Jah poitevin) pour son caractère chaleureux et jovial... Merci à **Stéphane** de m'avoir accueilli lors de mes quelques passages à Poitiers... Merci à **Bidou** d'avoir enfin daigné descendre sur Toulouse pour ma soutenance de thèse ! Merci à **Frédo** pour ses escapades Toulousaines, les sessions de grimpe et le raid : pourvu que ça dure !

Pour finir, je souhaite remercier les personnes qui me sont le plus proche... Tout d'abord merci aux "beaux parents" **Françis** et **Jo** qui nous accueillent très souvent à la gare de Leboulin : vous êtes fantastiques ! Encore merci pour le coup de main que vous m'avez donné pour les festivités du pot de thèse...

Merci à ma grande soeur **Cécile** et son boubou **Bruno** d'avoir déménagé en Ariège : c'est si bon d'avoir de la famille pas trop loin de chez soi !

Merci à mon frère **Adrien** qui est un homme en or et qui m'a fait découvrir l'informatique ! Merci d'avoir toujours été présent... Je ne désespère pas de te faire quitter Paris pour le Sud !

Merci à mes parents, **J-P.** et **Sissi** pour leur soutien durant ces longues années d'études ! Il est bon de venir recharger les batteries dans le cocon familial de temps en temps... Vive les parties d'échec avec le pater, les confitures de la mater et les bons repas

avec toute la famille. Malgré la distance, soyez-en sûrs : je pense beaucoup à vous où que je sois !

Enfin merci à “ma mie” **Marielle** qui me supporte depuis 2 ans déjà... Si j'étais fou, j'écrirais dans mes remerciements de thèse que je souhaite passer ma vie avec toi...

Résumé

Les travaux présentés dans ce manuscrit proposent une approche formelle pour la validation d'applications interactives Java-Swing vis-à-vis d'une spécification décrite par un modèle de tâches CTT. Cette approche de validation de l'utilisabilité du système s'appuie sur l'extraction d'un modèle formel décrivant le comportement dynamique de l'application (modèle de dialogue). Cette extraction est obtenue par analyse statique du code source Java-Swing de l'application. La validation du système consiste alors à démontrer formellement que les structures d'interaction encodées dans le programme s'inscrivent bien dans les scénarii d'usage représentés en compréhension par le modèle de tâches CTT. Cette étape de validation exploite d'une part le modèle formel extrait par analyse statique et d'autre part une formalisation du modèle de tâches. La démarche d'extraction et de validation est abordée suivant deux techniques formelles distinctes : la méthode B événementielle basée sur la démonstration de théorèmes (*theorem-proving*), et la méthode NuSMV basée sur la vérification exhaustive de modèles (*model-checking*). Une étude de cas permet d'illustrer tout au long du mémoire la démarche de validation proposée suivant ces deux techniques formelles.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

"La dernière chose qu'on trouve en faisant un ouvrage,
est de savoir celle qu'il faut mettre la première."
Blaise Pascal

Généralités.

Un système est qualifié d'*interactif* lorsqu'il interagit de manière continue avec son environnement. Les applications informatiques sont dans la plupart des cas considérées comme des systèmes interactifs qui communiquent avec l'utilisateur à travers une *Interface Homme-Machine* (IHM). On entend par IHM l'ensemble des mécanismes matériels et logiciels mis à la disposition des utilisateurs pour leur permettre d'interagir avec le système.

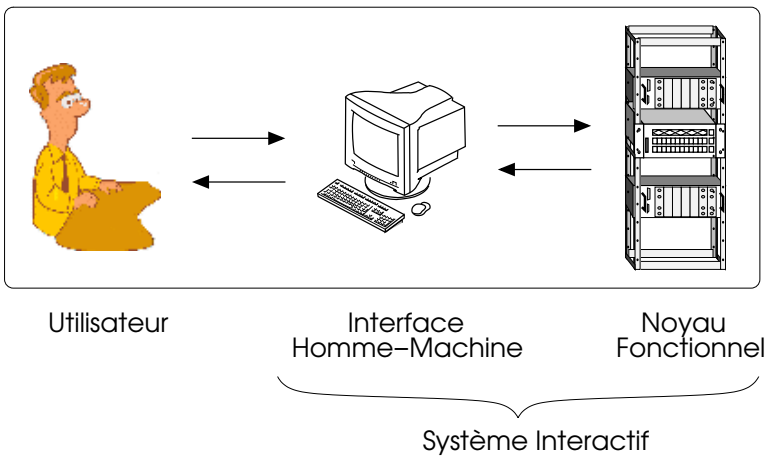


FIG. 1 – L'interface : médiateur entre l'homme et la machine

Selon [Myers, 1995b], 97% des applications possèdent une IHM. Dans la majorité des cas, ces IHM constituent le seul point d'entrée de l'utilisateur dans ces systèmes. Ces applications ont pour but d'établir un lien entre les fonctionnalités brutes du système, regroupées dans un composant appelé *Noyau Fonctionnel* (NF), et l'utilisateur. Un système interactif peut donc être vu comme la combinaison d'une IHM et d'un noyau fonctionnel (Fig. 1).

La naissance des IHM remonte aux débuts des années 60 avec l'apparition du clavier et des systèmes capables d'interpréter des lignes de commande. En 1964, Douglas C. Engelbart introduit pour la première fois la souris : il élabore alors un écran combinant des fenêtres dans lesquelles s'affichent des menus auxquels il est possible d'accéder en déplaçant un pointeur... dirigé à l'aide d'une souris à deux roues et en métal.

Depuis, le développement des IHM suit de près l'histoire de l'informatique, sur le terrain matériel et logiciel : aux interfaces de type "question-réponse" ont suivi des interfaces de type "grille de saisie" puis l'approche multi-fenêtrage avec une manipulation d'objets représentés graphiquement [Shneiderman, 1987]. Ces dernières interfaces, appelées interfaces WIMP (Windows, Icons, Menus and Pointers devices), ont offert une plus grande flexibilité à l'utilisateur et sont encore utilisées dans la majorité des applications. Aujourd'hui, le développement de domaines de recherche tels que la synthèse de la parole, la vision par ordinateur et la synthèse d'images apportent de nouvelles possibilités du point de vue de l'interaction homme-machine.

L'évolution des dispositifs d'interaction, en particulier à travers l'introduction de *modalités* et des combinaisons possibles de ces modalités, ont conduit à la naissance des IHM *multimodales*. L'objectif de ces interfaces est d'améliorer la qualité de la communication entre l'homme et la machine en se rapprochant au mieux de la qualité de la communication humaine.

À l'heure actuelle, du fait de cette évolution des dispositifs d'interaction et de la taille sans cesse croissante des systèmes à interfacer, la conception des IHM est de plus en plus complexe. En outre, celles-ci accompagnent aujourd'hui des applications "critiques" (système de contrôle de centrales nucléaires, cockpit des dernières générations d'aéronefs...) dont les défaillances peuvent avoir des conséquences catastrophiques.

Contexte des travaux.

L'IHM constituant le seul point d'entrée de l'utilisateur pour communiquer avec le système, il devient indispensable de disposer de modèles de spécification, de développement et de techniques de vérification et de validation pour pouvoir maîtriser la complexité évoquée précédemment et pour augmenter la flexibilité et la facilité d'utilisation des IHM, en permettant de répondre au critère d'*utilisabilité* de l'IHM. L'utilisabilité d'un système dénote l'efficience et la satisfaction avec lesquelles les utilisateurs peuvent utiliser un système interactif pour accomplir un but. Ce critère est une caractéristique particulière aux IHM souvent omise dans le processus de validation de logiciels interactifs.

Les travaux présentés dans ce mémoire s'inscrivent dans ce contexte. Ils visent à contribuer à la validation des systèmes interactifs du point de vue de l'utilisabilité du système. L'objectif est de proposer une approche de validation rigoureuse permettant d'assister

le développeur dans sa démarche de vérification et pouvant s'inscrire dans un cycle de développement traditionnel.

De nombreux travaux ont été menés dans ce domaine. Les travaux issus de disciplines telles que l'ergonomie ou la psychologie ont proposé des modèles permettant de mieux comprendre le comportement de l'homme face à la machine. Des règles permettant de définir ou de mesurer l'utilisabilité des IHM ont été définies à partir de ces modèles. Plusieurs notations ont également vu le jour. Ces notations définissent des méthodes d'analyse et de description de l'activité de l'utilisateur (modèles de tâches) et facilitent la conception et surtout la validation des applications interactives. Cependant, la distance entre ces disciplines (ergonomie et psychologie) et la programmation rend difficile l'application de ces résultats au développement des systèmes interactifs.

Les méthodes du génie logiciel, et plus particulièrement les méthodes formelles, ont apporté des solutions du point de vue de la validation de systèmes critiques (systèmes de sécurité, systèmes embarqués). Cependant, ces méthodes ne s'appliquent aujourd'hui qu'à une partie de ces logiciels souvent qualifiée de coeur du système critique.

Beaucoup de travaux ont exploité et adapté les techniques du génie logiciel pour traiter le problème de la validation des IHM. La majorité des travaux proposés aborde ce problème suivant une *approche descendante* de conception consistant à utiliser les méthodes formelles au plus tôt dans le cycle de développement. Cette approche se base sur une décomposition du système par raffinements successifs jusqu'à l'obtention d'un modèle concret proche de l'implémentation. Un tel modèle peut alors être exploité pour la génération automatique de code source.

Cependant, force est de constater qu'il est difficile d'utiliser de telles approches descendantes dans un cycle de développement traditionnel, principalement basé sur l'utilisation de générateurs d'interfaces ou de boîtes à outils graphiques. Ces outils du type générateur d'interfaces permettent de composer l'aspect visuel d'une interface par manipulation directe (et pour certains les aspects dynamiques du dialogue homme-machine) et permettent de générer tout ou partie du code source de l'interface. L'introduction des méthodes formelles nécessiterait une révision en profondeur de ces méthodes de travail d'autant plus que l'étape de génération de code source d'un système interactif à partir d'un modèle formel reste difficile. En outre, l'utilisation des méthodes formelles nécessite des compétences spécifiques : les difficultés rencontrées dans l'élaboration de modèles formels constituent un handicap non négligeable à leur usage par l'ensemble des intervenants dans les processus de développement des IHM.

Contribution.

L'objectif principal des travaux présentés dans ce mémoire est de proposer une démarche de validation basée sur l'utilisation des méthodes formelles qui permet de vérifier une partie des critères d'utilisabilité de l'IHM et qui respecte les méthodes de conception actuellement utilisées. Les travaux exposés dans ce mémoire proposent une démarche de validation s'appuyant directement sur le code source de l'application développée et pouvant s'intégrer en aval d'un outil de génération d'interface (ou d'une utilisation de boîte à outils spécifique). Cette approche repose sur l'extraction de modèles formels par analyse statique du code source de l'application développée. Les modèles formels obtenus sont

alors exploitables pour vérifier que le système produit se conforme aux spécifications du système représentées sous la forme de modèles de tâches.

À la différence des principaux travaux du domaine qui se basent sur une *approche descendante* pour la validation formelle d'IHM, il s'agit donc dans ce travail d'explorer une démarche de validation basée sur une *approche ascendante*. À notre connaissance, seuls quelques travaux ont exploré une telle approche (Projet IVY[1] [Silva *et al.*, 2006] et Projet VERBATIM[2] [d'Ausbourg & Durrieu, 2006]). Dans ces deux cas, l'approche ascendante proposée exploite des méthodes formelles basées sur la vérification exhaustive de modèles (*model-checking*) et ne traite pas explicitement de la validation de modèles de tâches CTT.

La principale contribution de nos travaux est la définition d'une approche de validation formelle ascendante basée sur une technique orientée démonstration de théorèmes (*theorem-proving*) : la méthode "B événementiel". La validation considérée consiste à montrer que les scénarii d'interaction encodés dans le code source de l'application développée sont conformes à ceux décrits en compréhension par une spécification de type modèle de tâches.

Afin de mettre en évidence la faisabilité de l'approche proposée suivant les deux techniques de preuve existantes (*model-checking* et *theorem-proving*) et dans le souci de pouvoir comparer ces techniques, ce mémoire présente également la démarche de validation suivant une méthode formelle orientée *model-checking*: la méthode NuSMV (*New Symbolic Model Verifier*).

Bien que l'approche présentée dans ce mémoire soit applicable pour la majorité des langages de programmation, nos travaux se sont focalisés sur le langage Java/Swing du fait de sa popularité grandissante et de sa portabilité intrinsèque. Parmi les différents modèles de tâches exploitables, la démarche de validation proposée s'est tournée vers l'utilisation du langage CTT actuellement le plus utilisé et le mieux outillé.

Plan du mémoire.

Ce mémoire est divisé en six chapitres.

État de l'Art. Les deux premiers chapitres de ce mémoire proposent un état de l'art aussi complet que possible sur le domaine de l'Interaction Homme-Machine et plus particulièrement sur le développement des systèmes interactifs.

Le premier chapitre de cet état de l'art présente les notations et modèles exploités dans un cycle de développement des systèmes interactifs. Le cycle de développement en V adapté au contexte du développement des systèmes interactifs et proposé par [Balbo, 1994] (Figure 2) est utilisé dans ce chapitre comme ligne directrice.

Suite à une courte introduction, les modèles utilisés pour le développement des IHM sont présentés en suivant les différentes étapes de ce cycle de développement : modèles de l'utilisateur et modèles de tâches (Analyse des besoins et Spécification), modèles d'architecture dédiés aux systèmes interactifs (Conception globale) et modèles de dialogue

[1] IVY - A model-based usability analysis environment. http://www.di.uminho.pt/ivy

[2] RNRT VERBATIM - VERification Biformelle et Automatisation du Test d'Interfaces Multimodales, http://iihm.imag.fr/nigay/VERBATIM

(Conception détaillée). Enfin, ce premier chapitre présente une classification des types de propriétés des IHM et un aperçu des outils de conception dédiés IHM.

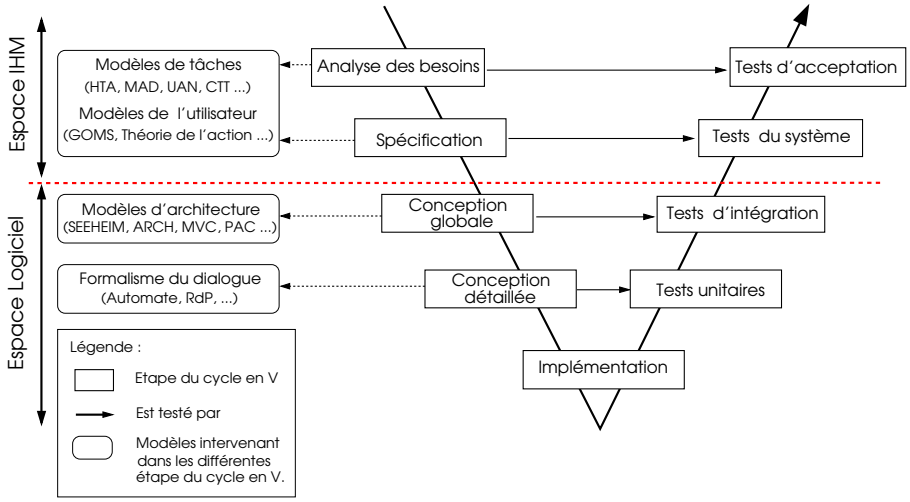


FIG. 2 – Cycle de développement en V et modèles intervenant dans les différentes étapes du développement.

Le deuxième chapitre de cet état de l'art présente l'état actuel des recherches relatives aux développements formels des systèmes interactifs et aux techniques d'analyse statique de code source. Pour conclure ce deuxième chapitre, une synthèse et un ensemble de constats sur le développement et la validation des IHM sont alors proposés. Basée sur ces constats, une approche de validation formelle des IHM est alors exposée en fin de chapitre.

Approche proposée : Validation formelle d'application Java-Swing par analyse statique de code source. Les chapitres 3, 4, 5 et 6 de ce mémoire développent l'approche de validation proposée. Le chapitre 3 présente quelques prérequis nécessaires à la compréhension de l'exposé (langage Java-Swing, Méthode B et méthode NuSMV) et une étude de cas (convertisseur Euros/Dollars) utilisée pour illustrer les différentes étapes de l'approche.

Le chapitre 4 présente les principes de la modélisation du dialogue d'une application Java/Swing suivant les techniques formelles B et NuSMV.

Le chapitre 5 détaille les techniques et algorithmes d'analyse statique permettant d'extraire ces modèles formels et de capturer la dynamique du dialogue homme-machine.

Le dernier chapitre présente l'exploitation des modèles formels extraits en vue de valider l'application vis-à-vis d'une description de tâches exprimée CTT. Une nouvelle fois, cette démarche de validation de l'utilisabilité du système est présentée suivant les deux méthodes formelles : B événementiel et NuSMV.

Enfin, ce mémoire de thèse se termine par une conclusion et une présentation des perspectives dégagées à l'issue de nos travaux.

Première partie

État de l'Art : Conception, Validation et Analyse de Systèmes Interactifs.

"Si l'automobile avait suivi le même développement que les ordinateurs, une Rolls-Royce coûterait aujourd'hui 500 francs, ferait du 700 kilomètres heure et exploserait une fois par an en faisant 10 morts..."
Robert Cringely

"Un ordinateur fait au bas mot 1 million d'opérations à la seconde, mais il a que ça à penser, aussi."
Jean-Marie Gourio (Brèves de comptoir, 1988)

Chapitre 1

Le domaine de l'Interaction Homme-Machine

"Lorsque l'énoncé d'un problème est exactement connu, le problème est résolu ;
ou bien c'est qu'il est impossible.
La solution n'est donc autre chose que le problème bien éclairé."
Emile-Auguste Chartier dit Alain (1868-1951)

1.1 État de l'art : Introduction

L'évolution des modalités d'interaction et la taille sans cesse croissante des systèmes à interfacer rendent les interfaces homme-machine 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 [Memon, 2001, Memon, 2002] et donc à un effort de développement tout aussi important. Le développement lié à l'interface représente 45% à 60% du code d'une application et monopolise 50% à 80% du temps imparti au processus de développement [Mahajan & Shneiderman, 1997], [Myers, 1995b], [Myers, 1995a], [Myers & Rosson, 1992].

En outre, les IHM accompagnent aujourd'hui des applications qualifiées de *critiques*. Les systèmes critiques sont les systèmes dont les défaillances peuvent avoir des conséquences catastrophiques ou extrêmement coûteuses. Tous les grands secteurs d'activité, tels que les transports (terrestre et aérien), l'énergie nucléaire, les télécommunications, la médecine et le secteur spatial [Wick *et al.*, 1993] sont concernés. Dans ce contexte plus que tout autre, la correction et l'utilisabilité de l'IHM dont dépendent l'exécution correcte de l'ensemble de l'application sont primordiaux. L'utilisabilité d'un système dénote l'efficacité et la satisfaction avec lesquelles les utilisateurs peuvent utiliser le système pour réaliser leurs buts.

Afin de prendre en compte cette complexité et cette criticité et par conséquent d'assurer la correction et l'utilisabilité de l'IHM, le recours à des modèles, à des notations et à des

critères d'évaluation est rendu nécessaire. La conception de systèmes interactifs utilisables nécessite la collaboration de spécialistes issus de différentes disciplines : informaticiens, ergonomes, sociologues, cognitiens et psychologues.

Ce chapitre présente les notations ou modèles proposés par ces différentes disciplines. Les différents types de modèles seront exposés suivant leur apparition dans un cycle de développement.

Cycles de développement des systèmes interactifs

Une modélisation du processus de développement de logiciels est un schéma générique utilisé par les concepteurs pour développer un système informatique.

D'un point de vue général, les modèles ou cycles de développement classiques du génie logiciel ont pour objectif d'assurer deux critères : la *faisabilité* et la *qualité* du système. La faisabilité est associée à la difficulté et au travail nécessaire à la réalisation du système. La qualité, quant à elle, fait référence à la correction, la sûreté et la maintenance du système.

Afin d'assurer le critère d'*utilisabilité* d'un système interactif, critère non pris en charge dans les cycles de développement classiques, les méthodes issues du domaine du génie logiciel ont été utilisées et adaptées au domaine de l'IHM.

Le modèle de développement en *cascade* [Boehm, 1981] fut le premier modèle à être utilisé par Royce [Royce, 1987]. Ce modèle distingue globalement quatre étapes de développement : l'analyse des besoins, la spécification, la conception du système et les tests du logiciel. La progression du développement dans un cycle en cascade se fait séquentiellement avec un possible retour aux étapes antérieures. Par conséquent, ce type de modèle ne favorise pas les éventuelles modifications de conception, mais tend à une stabilisation rapide du système. Du point de vue de la conception de l'IHM, le modèle en cascade ne fait aucune référence à l'utilisation de modélisation de l'utilisateur. Ce dernier intervient uniquement durant la phase de test lorsque le logiciel est achevé.

Le modèle en cascade a par la suite été raffiné sous diverses formes avec le *modèle en V* [McDermid & Ripkin, 1984], le *modèle* ∇ (prononcé "nabla") [Kolski, 1998], le *modèle en spirale* [Boehm, 1988] qui introduit un processus itératif, le *modèle en étoile* [Hartson & Hix, 1989] et le *modèle en couches* [Curtis & Hefley, 1994]. Ces modèles sont particulièrement adaptés au processus de développement des IHM. Par souci de simplicité, nous nous limiterons à la présentation du modèle en V (cf. Figure 1.1) proposé par [Balbo, 1994] qui explicite et structure les activités de tests. Parmi ces activités de test, Boehm différencie la validation et la vérification [Boehm, 1988] :

- la *validation* s'interroge sur l'adéquation du logiciel : "Construisons-nous le bon produit ?". L'adéquation n'est pas une caractéristique mesurable, mais relève d'un jugement subjectif ;
- la *vérification* concerne la conformité du produit vis-à-vis d'une description de référence : "construisons-nous le produit correctement ?". La vérification n'a de sens que s'il existe un document de référence, par exemple, un dossier de spécification ou un cahier des charges.

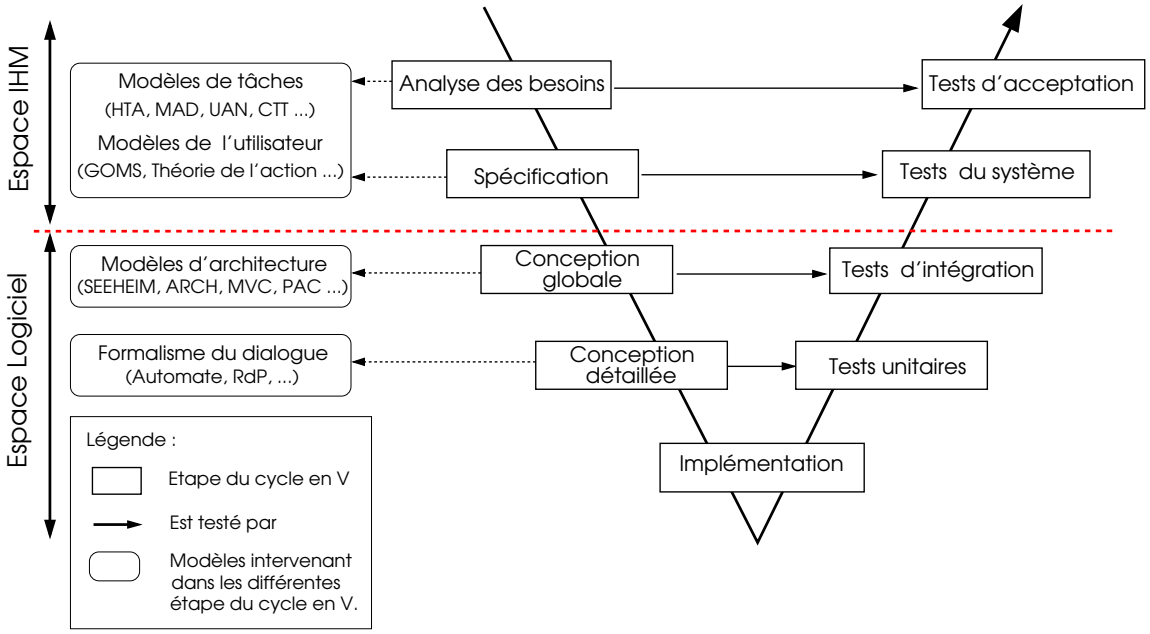


FIG. 1.1 – Cycle de développement en V et modèles intervenant dans les différentes étapes du développement.

La pente descendante du modèle en V comprend les étapes : analyse des besoins, spécification, conception globale et conception détaillée avant le codage. À chaque étape correspond un ensemble de tests permettant de valider ou vérifier l'étape. Les tests doivent être décrits lors de la phase de conception et effectués en séquence une fois le codage terminé. On distingue 4 étapes de tests :

- les **tests unitaires** permettent de vérifier que chaque module qui compose le système répond à sa spécification ;
- les **tests d'intégration** consistent à vérifier que les composants interagissent correctement une fois intégrés au système global. Dans le cas de systèmes de grande ampleur, les composants sont introduits un à un. À chaque étape, des modules de simulation remplacent les composants logiciels absents afin d'obtenir une vision du fonctionnement global du système et d'offrir la possibilité de tester graduellement l'intégration des composants ;
- les **tests système** ont pour objectif de s'assurer que la solution mise en place répond bien à la solution exprimée dans le dossier de spécifications externes ;
- enfin, les **tests d'acceptation** servent à vérifier la couverture du logiciel vis-à-vis des besoins exprimés dans le cahier des charges.

Le modèle en V permet de prendre en compte les exigences de l'utilisateur dès les premières étapes de la conception. Le cycle en V de la figure 1.1 est découpé en deux espaces : l'espace IHM et l'espace logiciel. Le premier espace se caractérise par la priorité qu'il accorde aux aspects ergonomiques et psychologiques. Le second espace se focalise sur les méthodes et techniques logicielles d'implémentation.

L'espace IHM, qui regroupe l'analyse des besoins, les spécifications et les tests du système et d'acceptation, s'appuie sur des modèles ou notations permettant de comprendre la logique de l'utilisateur (Modèles de l'utilisateur) et des modèles décrivant les tâches devant être accessibles à l'utilisateur (Modèles de tâches). A l'origine, les modèles de tâches

étaient absents du processus de conception : une des conséquences était la difficulté d'utilisation des systèmes conçus. L'introduction de ce type de modèles, construit à partir d'informations collectées lors d'interviews sur les méthodes de travail des futurs utilisateurs, permet notamment la validation du système en terme d'utilisabilité. Ces modèles viennent en complément des modèles traditionnellement utilisés en génie logiciel.

De nombreux travaux ont été réalisés par la communauté des ergonomes et des psychologues pour l'analyse des tâches et la modélisation de l'utilisateur lors de ces étapes d'analyse des besoins et de spécifications. Au niveau global, les modèles d'architecture offrent un schéma générique des différents composants du système interactif. Du point de vue de la conception détaillée, plusieurs modèles de description du dialogue proposent des solutions opérationnelles permettant la gestion de la communication homme-machine.

Dans ce qui suit, une description des modèles intervenant dans la phase de conception du cycle en V est présentée : modèles de l'utilisateur (section 1.2), modèles de tâches (section 1.3), les modèles d'architecture dédiés IHM (section 1.4) et modèles de description du dialogue (section 1.5). Nous nous limiterons à la présentation des formalismes les plus répandus dans les phases de développement en V. Ainsi que le note [Kamel, 2006], on constate l'absence quasi totale de modèles spécifiques aux systèmes multi-modaux en dehors de la proposition de [Rousseau *et al.*, 2004].

Ce chapitre fait ensuite le point sur les différents types de propriétés IHM devant être validées ou vérifiées (section 1.6) lors de la phase d'évaluation du logiciel (phase ascendante du cycle en V).

Enfin, une dernière section propose un tour d'horizon des différents outils de conception permettant d'assister le concepteur lors de la conception ou l'implémentation d'un système interactif.

1.2 Modèles utilisateur : les apports de la psychologie et de l'ergonomie

De nombreux travaux dans les domaines de l'ergonomie et de la psychologie ont proposé des modèles pour comprendre la logique de l'utilisateur et prévoir ou interpréter les difficultés d'interaction.

Il est possible de distinguer quatre modèles principaux parmi les modèles issus du domaine de la psychologie : un modèle théorique général (modèle du processeur Humain) [Card *et al.*, 1983], deux modèles fondés sur l'observation du comportement (modèles béhavioristes GOMS et Keystroke) [Card *et al.*, 1983] et un modèle cognitiviste (théorie de l'Action) [Norman, 1986].

Modèle du Processeur Humain. Le modèle du Processeur Humain [Card *et al.*, 1983] représente le sujet humain comme un système multiprocesseur (systèmes sensoriel, moteur et cognitif) de traitement de l'information. Ce modèle présente l'intérêt notable de définir un cadre fédérateur à la diversité des connaissances en psychologie cognitive et propose des équations permettant des calculs approximatifs pour évaluer *a priori* certaines performances de bas niveau de l'utilisateur (temps nécessaire à la réalisation d'une action). Cependant, ce modèle ne présente aucune information quant à la structure des représentations mentales de haut niveau (formation des concepts,

reconstruction de la mémoire...). En outre, cette représentation théorique ne fournit aucune méthode de conception des IHM.

GOMS. *Goal, Operator, Method and Selection* (GOMS) [Card *et al.*, 1983] est un modèle de description du comportement observable d'un utilisateur (modèle comportemental) et ne cherche pas à décrire les états mentaux et les traitements internes de l'utilisateur (approche cognitiviste). GOMS propose quatre niveaux d'analyse du comportement de l'utilisateur, chaque niveau correspondant à un degré d'abstraction particulier :

- **Niveau tâche** : l'activité de l'utilisateur est décomposée en une hiérarchie de sous-tâches dont la nature dépend uniquement du domaine.
- **Niveau fonctionnel** : les tâches élémentaires identifiées au niveau supérieur sont décrites en termes de fonctions offertes par le système informatique.
- **Niveau argument** : description pour chaque fonction de la suite des commandes et de ses arguments (interface texte ou graphique).
- **Niveau physique** : description pour chaque commande de la suite des mouvements physiques que doit faire l'utilisateur.

Modèle de Keystroke. GOMS permet aussi de prédire le temps d'exécution de ces actions. Sa version simplifiée Keystroke [Card *et al.*, 1983] concerne les aspects syntaxiques et lexicaux de l'interaction. Le modèle de Keystroke permet de décomposer l'activité de l'utilisateur (actions physiques sur l'interface et activités mentales) en un ensemble de tâches élémentaires et génériques pour prédire le temps d'exécution de cette activité. L'utilisation du modèle de Keystroke permet de comparer les différents choix lexicaux et syntaxiques possibles d'une interface. Par exemple, il est possible d'effectuer une évaluation comparative de procédés de déplacement sur une interface suivant deux méthodes M1 et M2 (M1 : déplacement du curseur dans un éditeur de texte en utilisant la souris, M2 : déplacement du curseur dans un éditeur de texte en utilisant le clavier).

Les modèles de GOMS et de Keystroke permettent une description minimaliste et simplifiée des actions de l'utilisateur sur l'interface en offrant le choix d'une analyse descendante (*top-down*) du comportement d'un utilisateur à partir d'un but, ou celui d'une analyse ascendante (*bottom-up*) par assemblage de comportements élémentaires. L'avantage est de proposer un cadre familier aux informaticiens et, de surcroît, de proposer un modèle prédictif puisque les modèles fournissent des mesures. Cependant, ces modèles ne prennent pas en compte les erreurs possibles de l'utilisateur et, tout comme le modèle du Processeur Humain, ils ne fournissent aucune donnée quant à la conception du système.

Théorie de l'Action. Si GOMS et Keystroke modélisent un comportement observé, la théorie de l'Action [Norman, 1986] modélise les processus psychologiques qui conduisent à ce comportement : l'utilisateur élabore un modèle conceptuel du système informatique et son comportement est conditionné par l'environnement et sa représentation interne du système. La théorie de l'Action associe notamment la réalisation d'une tâche au parcours d'une distance exprimant la différence entre la représentation de l'interface et celle maintenue dans l'idée que s'en fait l'utilisateur [Coutaz, 1990]. Les distances sémantiques et articulatoires traduisent (cf. figure 1.2) respectivement :

- la difficulté de l'utilisateur à interpréter l'état du système en fonction des informations et des grandeurs présentées sur l'interface ;

– et la difficulté d'un utilisateur à faire correspondre à un but l'enchaînement des commandes permettant sa réalisation.

Du point de vue de l'utilisabilité du système, le concepteur a pour objectif de diminuer la longueur de ces distances par l'intermédiaire de l'interface graphique. Notamment, l'utilisation de métaphores, le choix dans la disposition des composants graphiques et de leur intitulé permettent la réduction de ces distances. En résumé, la théorie de l'Action est un cadre permettant la mise en ordre d'un ensemble de règles ergonomiques trouvées expérimentalement et qui enrichit les modèles précédents en précisant la notion d'état (effectif et perçu). En outre, cette théorie permet d'expliquer les réussites, difficultés et erreurs d'un utilisateur.

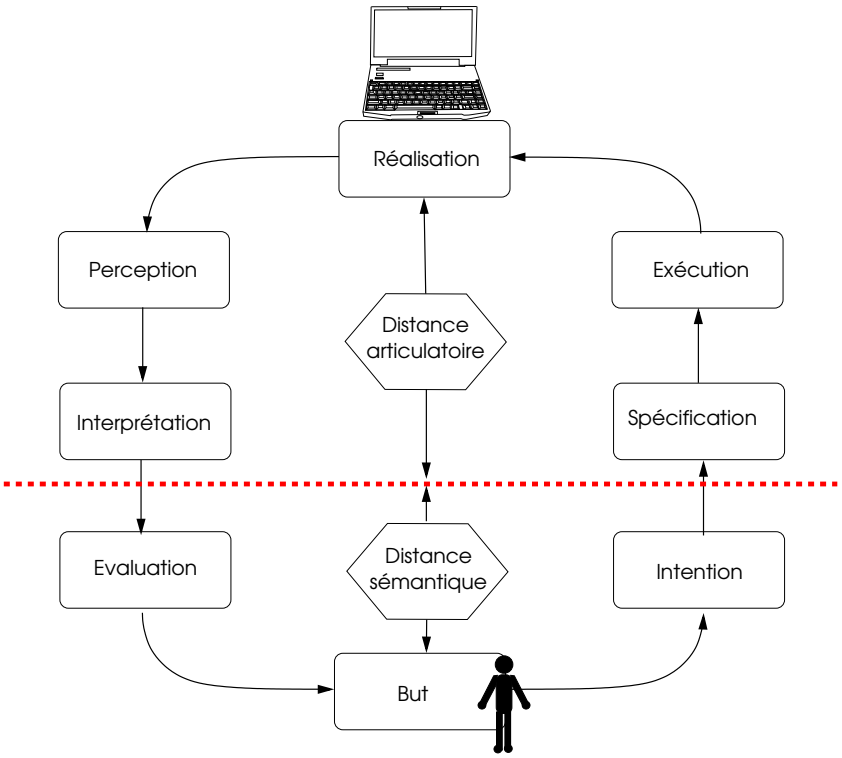


FIG. 1.2 – Théorie de l'action : distances sémantiques et articulatoires

L'ensemble des modèles présentés donne au concepteur un ensemble de données psychologiques et ergonomiques sur le comportement de l'utilisateur. Ces informations sont exploitables pour adapter l'interface en vue d'améliorer l'utilisabilité du système. Cependant, ces modèles ne fournissent aucune information sur la réalisation technique du système interactif, c'est-à-dire l'aspect informatique de l'IHM. Les sections 1.3, 1.4, 1.5 exposent quelques méthodes, outils et modèles utilisables du point de vue de la programmation des interfaces homme-machine.

Des informations supplémentaires sur les aspects ergonomique et psychologique des IHM peuvent être trouvées dans [Kolski, 1993], [Jacko & Sears, 2003].

1.3 Analyse et Modèles de Tâches

Les modèles de tâches expriment les besoins de l'utilisateur. Suivant la définition du dictionnaire [Robert, 2008], le mot *tâche* désigne un travail déterminé que l'on doit exécuter. Dans le domaine de l'interaction homme-machine, une *tâche* est définie comme un objectif à atteindre par l'utilisateur à l'aide d'un système interactif [Normand, 1992]. Des concepts liés à la définition d'une tâche ont été définis et sont résumés comme suit [Balbo, 1994] :

- une **tâche** représente un but que l'utilisateur souhaite atteindre via une procédure décrivant les moyens et la manière de l'atteindre ;
- un **but** est un état du système que l'utilisateur souhaite obtenir. Dans le cas où l'interface du système n'est pas *honnête*, l'état réel du système risque d'être différent de l'état perçu par l'utilisateur ;
- une **procédure** est définie par un ensemble d'opérations reliées entre elles par des relations temporelles et structurelles ;
- une **action** désigne une *opération terminale*. Elle intervient dans l'accomplissement d'un but terminal ;
- une **opération** est une action ou une tâche ;
- un **but terminal** et les actions qui permettent de l'atteindre définissent une **tâche élémentaire** ;
- une **procédure élémentaire** est la procédure associée à une tâche élémentaire ;
- les **relations temporelles** entre les opérations d'une procédure traduisent la séquentialité, l'interruptibilité ou le parallélisme ;
- les **relations structurelles** servent à exprimer la composition logique des opérations ou la possibilité de choix ;
- une **tâche composée** est une tâche non élémentaire. Elle inclut dans sa description celles de sous-tâches.

Reprise de [Balbo, 1994], la figure 1.3 représente l'ensemble de ces concepts.

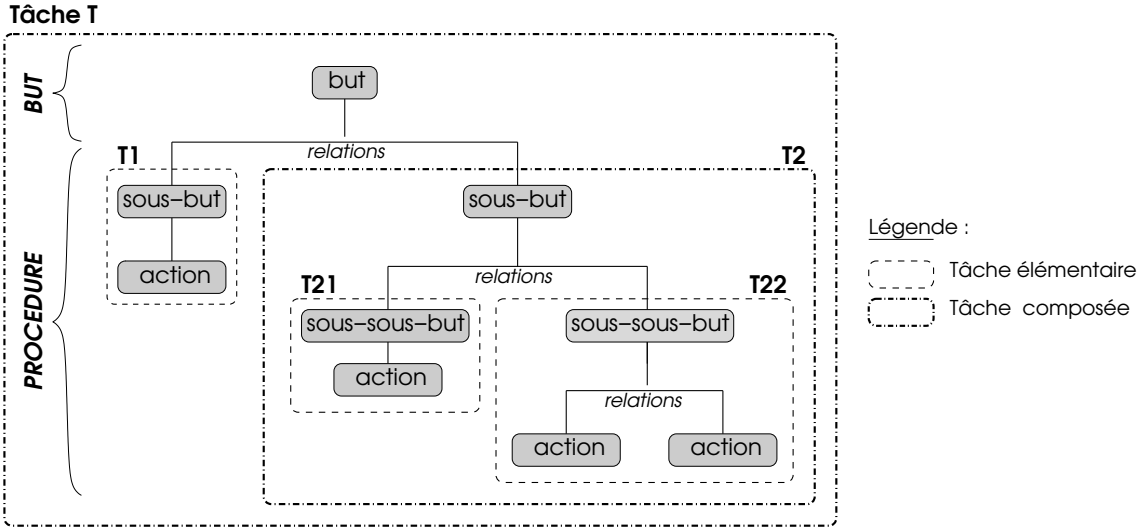


Fig. 1.3 – Décomposition de tâche et concepts associés

De nombreuses notations ont été proposées dans la littérature, la plupart d'entre elles provenant des ergonomes et des psychologues. Les formalismes proposés, la plupart du temps des formalismes graphiques, diffèrent suivant leur qualité et leur finalité. De nombreuses classifications ont été proposées afin d'aider le concepteur dans le choix du modèle adapté à ses besoins [Brun, 1998], [Balbo, 1994], [Jambon, 1996], [Tabary, 2001], [Navarre, 2001], [Limbourg & Vanderdonckt, 2003]. La présentation qui suit se limite à un certain nombre de modèles précurseurs et représentatifs, en distinguant deux types de modèles : les modèles d'*analyse de tâches* HTA [Shepherd, 1989] et MAD [Scapin & Pierret-Golbreich, 1989], les *modèles de description de l'interface* homme-machine UAN [Harston & Gray, 1992], et CTT [Paternò, 2001]. Ces modèles peuvent être utilisés de deux manières :

1. *Comme modèle de spécification* : dans ce cas les fonctionnalités d'une IHM sont vues comme un ensemble de tâches que l'utilisateur peut effectuer. Le concepteur commence dans ce cas à spécifier l'IHM par une tâche principale qui sera décomposée en un ensemble de sous-tâches elles-mêmes décomposées à leur tour jusqu'à atteindre les tâches élémentaires réalisées par des actions interactives de base. La spécification est obtenue en composant toutes les tâches principales grâce aux opérateurs de composition disponibles dans le modèle utilisé. Les modèles de tâches sont particulièrement utilisés à cet effet dans les outils de conception de type *Système Basé sur Modèles* ou les *Systèmes de Gestion d'Interface Utilisateur* dont il sera question en section 1.7.
2. *Comme modèle de validation* : dans ce cas, la tâche est modélisée pour vérifier que le système permet de la réaliser ou non. Si le système est représenté par un modèle, alors la validation du système vis-à-vis d'un modèle de tâche T consiste à vérifier que le modèle du système décrit des comportements prévus et spécifiés par le modèle de tâches.

1.3.1 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*. Ces informations sont obtenues par les récits des utilisateurs au moyen de lectures de rapports, d'interviews ou de simulations [Dix *et al.*, 1993]. Cette analyse est réalisée à haut niveau d'abstraction et doit être indépendante de toute idée d'implémentation du système. Par exemple, aucune référence liée aux dispositifs d'interaction ne doit apparaître.

Les *modèles d'analyse de tâches* correspondent aux notations utilisées pour représenter les informations collectées.

HTA. Le modèle HTA [Annett & Duncan, 1967], [Shepherd, 1989] (*Hierarchical Task Analysis*) est l'un des premiers formalismes proposés pour la collecte et l'analyse des informations concernant l'activité de l'utilisateur. Un modèle HTA, possédant une représentation graphique et textuelle, propose une décomposition hiérarchique des tâches en sous-tâches jusqu'à l'obtention de tâches élémentaires. La décomposition des tâches est structurée via l'utilisation de relations temporelles et conditionnelles entre les sous-tâches.

Il faut cependant noter le manque de clarté de la sémantique de ce modèle. De plus, l'absence de véritables opérateurs temporels rend difficile l'exploration d'un arbre de tâches HTA. Par ailleurs, ce formalisme n'est supporté par aucun outil.

MAD. Le modèle MAD [Scapin & Pierret-Golbreich, 1989] (*Méthode Analytique de Description*) est une méthode basée sur une approche psycho-ergonomique. MAD, tout comme HTA, dispose d'une notation graphique permettant la description hiérarchique de tâches. La différence entre HTA et MAD se situe au niveau de l'organisation structurelle de l'arbre de tâches. MAD introduit un ensemble de constructeurs qui jouent le rôle de liens temporels entre les sous-tâches : la séquence, l'alternative, le parallélisme et la simultanéité. Par ailleurs, cette notation associe également aux tâches un ensemble d'attributs, de pré et de post-conditions.

Le formalisme MAD a connu de nombreuses évolutions et améliorations pour aboutir à une version appelée MAD* [Gamboa & Scapin, 1997]. Cette version enrichit MAD par l'adjonction d'opérateurs temporels (interruption, désactivation, tâche multi-utilisateur) et la définition d'une sémantique précise des objets manipulés. Cette dernière notation est également outillée (outils IMAD*) pour permettre le recueil et l'édition de modèles.

1.3.2 Les modèles de description de l'interface homme-machine : UAN et CTT

Les modèles de description de l'interface homme-machine définissent *la vue que l'utilisateur aura du système interactif*. D'après [Balbo, 1994], cette vue doit donner accès aux services définis dans l'analyse des besoins. Ce type de modèle s'inscrit donc dans la continuité de l'analyse de tâches. Outre la description des activités de l'utilisateur pour atteindre un but, ces modèles doivent tenir compte des retours d'informations de l'interface lors de l'interaction. Ainsi, ces modèles définissent un pont entre les approches ergonomiques ou psychologiques et la conception du logiciel. Suivant la présentation de [Baron, 2003], on détaillera ici deux modèles particulièrement représentatifs à titre d'illustration : UAN et CTT.

UAN et XUAN. La notation UAN (*User Action Notation*) [Harston & Gray, 1992] est une notation permettant la spécification d'interfaces à manipulation directe. À la différence des modèles d'analyse de tâches, cette notation est en lien direct avec les choix d'implémentation du système. Elle permet la description de l'interaction de l'utilisateur sur l'interface sous forme textuelle, dans un tableau de trois colonnes regroupant : les actions physiques exécutées par l'utilisateur (clic souris), les retours d'informations de l'interface (sélection, activation d'un objet graphique) et finalement l'état de l'interface. Le tableau 1.1 présente un exemple de description UAN proposé par [Kamel, 2006].

Les actions de l'utilisateur $[file\_icon]$ et M_v signifient respectivement "*déplacer le curseur sur une icône de fichier non sélectionnée*" et "*presser le bouton de la souris*". En retour, le rendu de l'icône sélectionnée est modifié et apparaît en surbrillance $file\_icon$!. L'état de l'interface est modifié : la variable *selected* prend la valeur $file$. Enfin, l'action utilisateur $M\Diamond$ signifie "*relâcher le bouton de la souris*".

Cette notation a été utilisée pour spécifier l'interface en entrée du système multimodal MATIS dans les travaux de [Coutaz *et al.*, 1993a].

Tâche : Déplacer un fichier		
Action Utilisateur	**Retour Information**	**État Interface**
$[file\_icon]M_v$ $M\Diamond$	$file\_icon!$	$selected = file$

TAB. 1.1 – Déplacement d'une icône de fichier en UAN

Le formalisme UAN permet la description de l'interaction homme-machine à un faible degré d'abstraction : seule la description de tâches élémentaires est disponible. Sa version étendue XUAN [Gray *et al.*, 1994] (*Extended User Action Notation*) propose un niveau supplémentaire d'abstraction en définissant un niveau de tâches. Cette abstraction supplémentaire permet la décomposition hiérarchique des tâches en sous-tâches en tenant compte des descriptions du niveau des actions. XUAN introduit naturellement une structuration des tâches en définissant des opérateurs de composition, des relations temporelles et des pré- et post-conditions pour modéliser de manière plus complète la notion de tâche. Si l'on se réfère à la notation MAD présentée dans la sous-section précédente, on note toutefois l'absence des opérateurs d'interruption et de désactivation permettant de prendre en compte les erreurs de l'utilisateur [Jambon *et al.*, 2001].

CTT. La notation CTT (*ConcurTaskTree*, [Paternò, 2001]) met l'accent sur l'activité de l'utilisateur. CTT propose une vision hiérarchique des tâches sous forme d'arbre en différenciant quatre types de tâches : tâche *abstraite*, tâche *utilisateur*, tâche *interaction* et tâche *système*. Le séquencement des tâches est assuré par un ensemble d'opérateurs temporels empruntés à l'algèbre de processus LOTOS *(Language Of Temporal Ordering Specification)* [Systems, 1984]. CTT utilise une représentation graphique et propose un outil constituant un environnement d'édition, de simulation et de génération de scénarii de tâches, appelé CTTE (*CTT Environment*) [Paternò *et al.*, 2001], [Mori *et al.*, 2002].

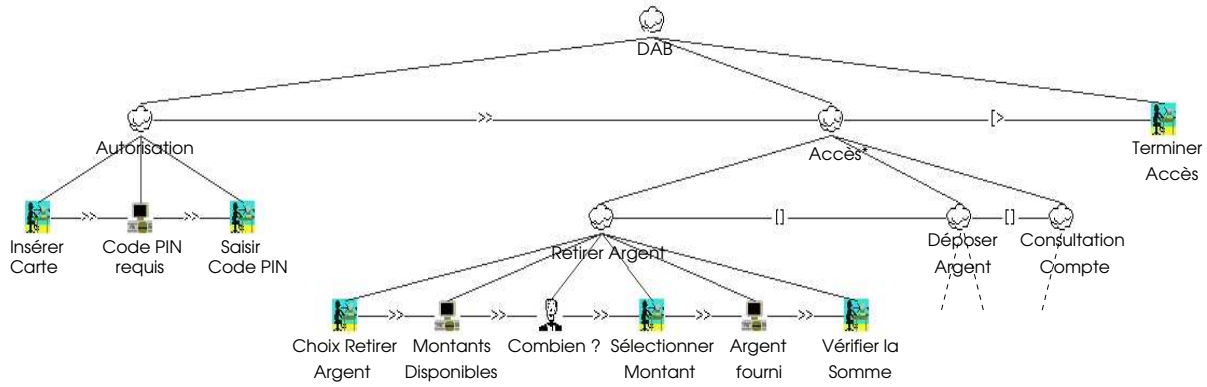


FIG. 1.4 – Exemple d'arbre de tâches CTT : Distributeur Automatique de Billets (DAB)

La figure 1.4 représente le modèle de tâche d'un distributeur automatique de billets. L'utilisateur doit dans un premier temps insérer sa carte puis (>>) saisir son code PIN afin d'obtenir l'autorisation d'accéder à son compte. Une fois l'autorisation obtenue, la tâche *Accès* peut être effectuée plusieurs fois (*) et désactivée à tout moment ([>) par la

tâche *Terminer Accès*. La tâche *Accès* consiste en un choix ([]) entre trois sous-tâches : *Retirer Argent*, *Déposer Argent* ou *Consultation compte*. Ici, seule la tâche *Retirer Argent* est détaillée.

CTT est une notation simple qui présente l'avantage d'associer dans une même notation graphique l'activité de l'utilisateur et les retours attendus de l'interface. Cependant, l'absence de description des objets contenus et manipulés dans les tâches constitue une faiblesse de la notation. En outre la sémantique des opérateurs utilisés manque de formalisation. Le manque de formalisme de cette notation rend difficiles l'évaluation et la validation de tâches CTT d'une application. Ce dernier point a fait l'objet de travaux sur la formalisation de modèles de tâches CTT dans le langage B événementiel [Baron, 2003], [Aït-Ameur & Baron, 2004], [Aït-Ameur *et al.*, 2005b].

1.4 Modèles d'architecture dédiés aux systèmes interactifs

Comme leur nom l'indique, les modèles d'architecture définissent l'architecture logicielle d'un système. Autrement dit, ils définissent une structure générique des différents composants logiciels d'un système et les différentes relations qui unissent ces composants. Ce type de modèle intervient au niveau de l'étape de "*Conception Globale*" du cycle de développement en V.

Dans le cas des systèmes interactifs, tous les modèles proposés reposent sur un principe commun : la séparation du *Noyau Fonctionnel* (NF) qui implémente les concepts propres à un domaine d'application particulier, et l'*interface* qui permet d'interagir avec ces concepts via le retour d'informations et la manipulation d'objets graphiques. Cette ségrégation modulaire présente un avantage puisqu'elle permet de modifier l'une des deux parties sans affecter la seconde. Cette structuration du logiciel en composants assure donc une modularité facilitant la réutilisation logicielle, la maintenance et son évolution. De manière générale, un modèle d'architecture doit [Kamel, 2006] :

- préconiser une séparation fonctionnelle entre les services de l'application et ceux de l'interface ;
- définir une répartition des services de l'interface, qui se traduit par un ensemble de composants logiciels ;
- définir un protocole d'échange entre les constituants logiciels qu'il identifie.

Une grande variété de modèles a été définie. Les sections qui suivent se contentent de présenter les modèles les plus cités dans la littérature. Ces modèles sont la plupart du temps classés suivant trois classes : les modèles globaux, les modèles multi-agents et les modèles hybrides.

1.4.1 Les modèles globaux

Les modèles globaux, également nommés macro-modèles, modèles généraux ou modèles centralisés [Fekete, 1996] ont été les premiers modèles proposés dans la littérature. Ils s'intéressent tout particulièrement à la séparation de la partie interactive du reste de l'application. Ils définissent le nombre, la nature et l'organisation structurelle des différents composants logiciels constituant le système. La structure de ces composants globaux n'est

pas traitée dans ce type de modèle.Le modèle de SEEHEIM [Pfaff, 1985] et le modèle ARCH [Bass *et al.*, 1991], [Paternò, 1993] appartiennent à cette classe de modèles.

Le modèle de SEEHEIM

Le modèle de SEEHEIM a été proposé au cours d'un séminaire sur les systèmes de gestion utilisateur en 1983 à Seeheim. Ce modèle préconise la décomposition logicielle d'un système interactif en trois modules logiques : la présentation, l'interface avec l'application et le contrôleur de dialogue. La figure 1.5 représente l'architecture du modèle de SEEHEIM.

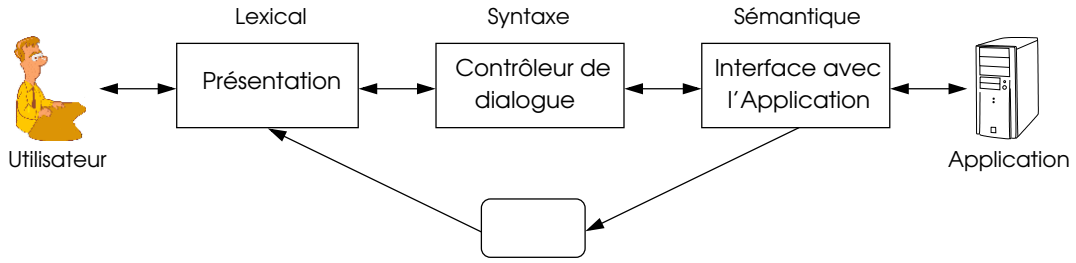


FIG. 1.5 – Le modèle d'architecture SEEHEIM

Présentation. Le composant "Présentation" prend en charge les entrées et les sorties du système interactif. Il présente donc les informations de l'application (image du système) et met à disposition de l'utilisateur les modalités d'entrée (souris, clavier) lui permettant d'agir sur le système. Ce module prend également en compte certaines règles d'ergonomie afin d'améliorer l'organisation graphique de l'interface et d'optimiser l'interaction. Du point de vue du dialogue homme-machine, le module de présentation correspond aux aspects lexicaux.

Interface avec l'application. Le composant "Interface avec l'application" est une surcouche de l'application qui définit la sémantique du dialogue homme-machine. Son rôle est de transformer les données provenant de la présentation par l'intermédiaire du contrôleur de dialogue en des données exploitables par l'application et vice-versa. Ce module établit donc un lien sémantique entre les concepts manipulés par l'utilisateur et les concepts de l'application.

Contrôleur de dialogue. Le module "Contrôleur de dialogue" établit un lien syntaxique entre la présentation et l'interface de l'application. Il détermine l'enchaînement possible des interactions entre l'utilisateur et le système. Il contrôle également les appels aux fonctionnalités de l'interface de l'application.

Le modèle de SEEHEIM a été le premier à proposer une architecture modulaire pour une application interactive. Cependant, on peut lui reconnaître quelques limites, notamment son manque de précision sur le fonctionnement du contrôleur de dialogue et la non-définition du lien entre l'interface de l'application et la présentation (cf figure 1.5). En outre, son caractère monolithique semble peu adapté à l'émergence de l'approche orientée objet.

Le modèle ARCH

Le modèle ARCH est une extension du modèle de SEEHEIM qui définit deux composants supplémentaires. Cinq composants constituent donc le modèle ARCH : boîte à outils, présentation, dialogue, adaptateur de domaine et domaine (figure 1.6).

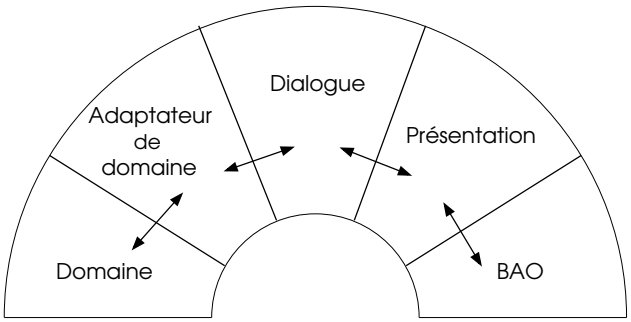


FIG. 1.6 – Le modèle d'architecture ARCH

Le domaine. Le domaine représente le noyau fonctionnel de l'application, c'est-à-dire les objets et les fonctions propres à l'application qui demeurent indépendants de l'application.

L'adaptateur de domaine. L'adaptateur de domaine joue le rôle d'intermédiaire entre le domaine et le dialogue en garantissant l'indépendance de ces deux modules. Il permet d'implémenter notamment les tâches utilisateurs relatives au domaine. L'adaptateur de domaine reste très proche de l'interface avec l'application du modèle SEEHEIM.

Le dialogue. Le dialogue est proche du rôle joué par le contrôleur de dialogue du modèle de SEEHEIM. À la différence de celui-ci, le modèle ARCH définit plus précisément l'objectif de ce module : il est chargé de maintenir la cohérence entre les différentes vues d'un même objet.

La présentation. La présentation joue un rôle intermédiaire entre le dialogue et la Boîte à Outils. Elle est composée de représentations abstraites d'objets d'interaction.

Boîte à Outils. Enfin, la Boîte à Outils implémente l'interaction physique avec l'utilisateur.

La décomposition plus fine d'un système interactif telle que présentée par le modèle ARCH supporte davantage les modifications dues aux évolutions de l'IHM. Cependant, tout comme le modèle de SEEHEIM, le modèle ARCH ne précise pas la structure interne des composants.

1.4.2 Les modèles multi-agents

Contrairement aux modèles globaux, les modèles d'architecture multi-agents (ou modèles génériques [Fekete, 1996]) définissent de manière précise la structure des composants de base de l'application ainsi que leurs liens de communication, mais ne précisent ni leur

nombre ni leur organisation. Dans ce type de modèle, il existe une séparation entre les aspects présentation et domaine de l'application, mais suivant une granularité beaucoup plus fine. Parmi cette classe de modèles d'architecture, on trouve entre autres les modèles MVC [Goldberg, 1984], PAC [Coutaz, 1987], ALV [Hill, 1992] et H4 [Depaulis *et al.*, 2006]. Par souci de concision, seuls les modèles MVC et PAC sont présentés dans ce qui suit.

Le modèle MVC

MVC pour *Model View Controller* a été utilisé dans l'architecture de Smalltalk [Goldberg, 1984]. Ce modèle est très utilisé dans l'industrie et particulièrement dans la boîte à outils Swing du langage Java. Ce modèle repose sur une décomposition hiérarchique de l'interface en agents autonomes. Chaque agent est décomposé en trois sous-modules (Modèle, Vue et Contrôleur) (cf. figure 1.7). La communication entre les agents est réalisée par envoi de messages au moyen de l'entité "Modèle".

Le *Modèle* réunit l'accès à l'ensemble des fonctionnalités du système et notifie les modifications à la *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*.

À la différence des modèles globaux, MVC est le premier modèle où le contrôle de l'interface est réparti plutôt que centralisé. De plus, ce modèle permet la création dynamique d'agents MVC.

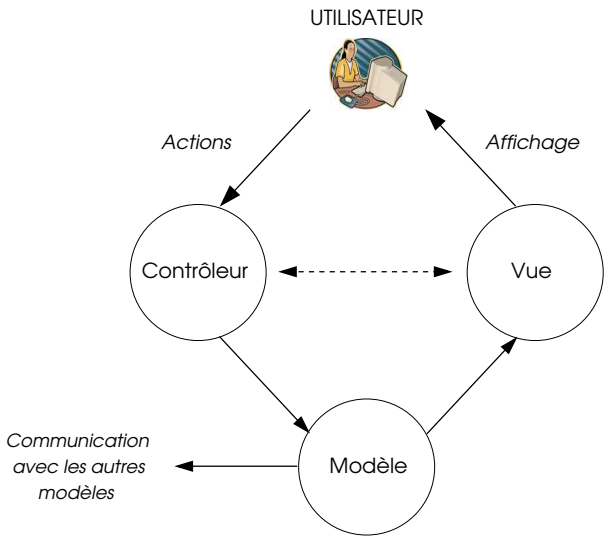


FIG. 1.7 – Le modèle d'architecture MVC

Le modèle PAC

Le modèle d'architecture PAC a été développé en 1987 par J. Coutaz [Coutaz, 1987]. Cette architecture structure une application sous la forme d'une hiérarchie d'agents interactifs. Tout comme MVC, le modèle PAC peut être vu comme une architecture SEEHEIM

répartie. La différence tient au fait que PAC peut modéliser l'ensemble de l'application là où MVC ne prend en compte que la décomposition de l'interface. Un agent PAC est constitué de trois sous-modules : la présentation, l'abstraction et le contrôle (cf. figure 1.8).

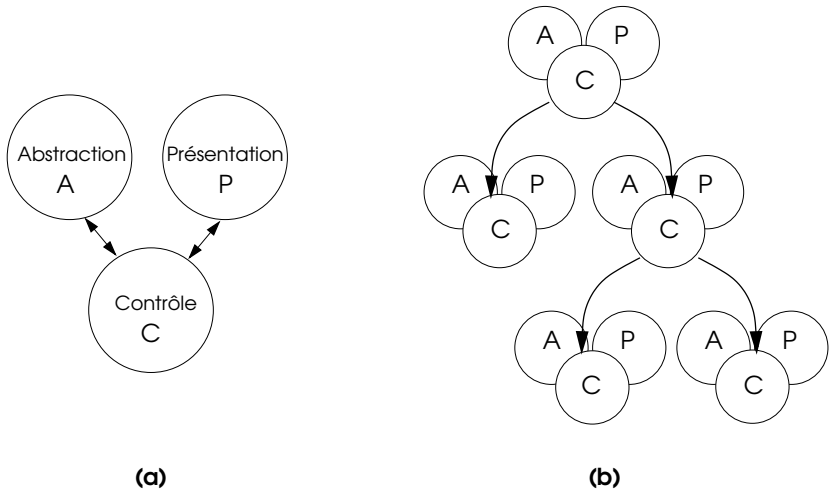


FIG. 1.8 – Le modèle d'architecture PAC

La *présentation* définit le comportement en entrée/sortie. L'*abstraction* définit les aspects graphiques et conceptuels de l'agent de manière indépendante. L'abstraction peut être vue comme le noyau fonctionnel de SEEHEIM. Enfin, le *contrôle* sert de pont entre l'abstraction et la présentation et gère les communications avec les autres agents.

1.4.3 Les modèles à base d'interacteurs : Cnuce, York, Cert

Les modèles à base d'interacteurs sont des modèles multi-agents qui, en plus des services offerts par ce type de modèles, offrent une description précise des différents événements manipulés et de leur composition. Différentes définitions d'interacteurs ont été proposées suivant la nature de l'approche employée [Markopoulos, 1997]. Les modèles d'York [Duke & Harrison, 1993], de Pise [Paternò, 1993], de Cnuce [Faconti & Paternó, 1990], du Cert [Roche, 1998, d'Ausbourg, 1998] et d'ADC [Markopoulos, 1995] appartiennent à cette classe de modèles à base d'interacteurs. La plupart de ces modèles ont été formalisés suivant différentes techniques formelles afin de les valider vis-à-vis d'un ensemble de propriétés. Les propriétés attendues des systèmes interactifs et non interactifs sont traitées en section 1.6. Une présentation des modèles et des techniques formelles utilisées pour la vérification de propriétés sera effectuée dans le second chapitre de cet état de l'art.

La notion d'interacteur repose sur un principe commun : la description d'un système interactif par composition de processus abstraits indépendants.

Interacteurs Cnuce

Les interacteurs de Cnuce ont été développés dans le cadre de la formalisation du projet GKS [GKS, 1985]. L'objectif de ce projet est de définir des composants de base (les

interacteurs) avec lesquels un système graphique interactif peut être modélisé, construit et vérifié.

Le modèle de CNUCE décrit la structure et le comportement de base d'un interacteur. Il distingue plusieurs types et sources d'événements, et introduit la notion de niveau d'abstraction. Au niveau le plus abstrait, l'interacteur est vu comme une "boîte noire" qui sert d'intermédiaire entre un côté utilisateur et un coté application. Il peut émettre ou recevoir des événements des deux côtés et traiter en interne les données qui transitent. À un niveau plus concret l'interacteur est vu comme une "boîte blanche" qui permet la description de son fonctionnement interne.

Les auteurs de ce modèle définissent un interacteur *comme une entité d'un système interactif capable de réagir à des stimuli externes, en traduisant des données d'un haut niveau d'abstraction vers un niveau plus bas d'abstraction et vice-versa.* Un système interactif est ainsi décrit comme un graphe d'interacteurs communiquant entre eux : le comportement d'un interacteur est réactif et fonctionne de manière parallèle avec d'autres interacteurs.

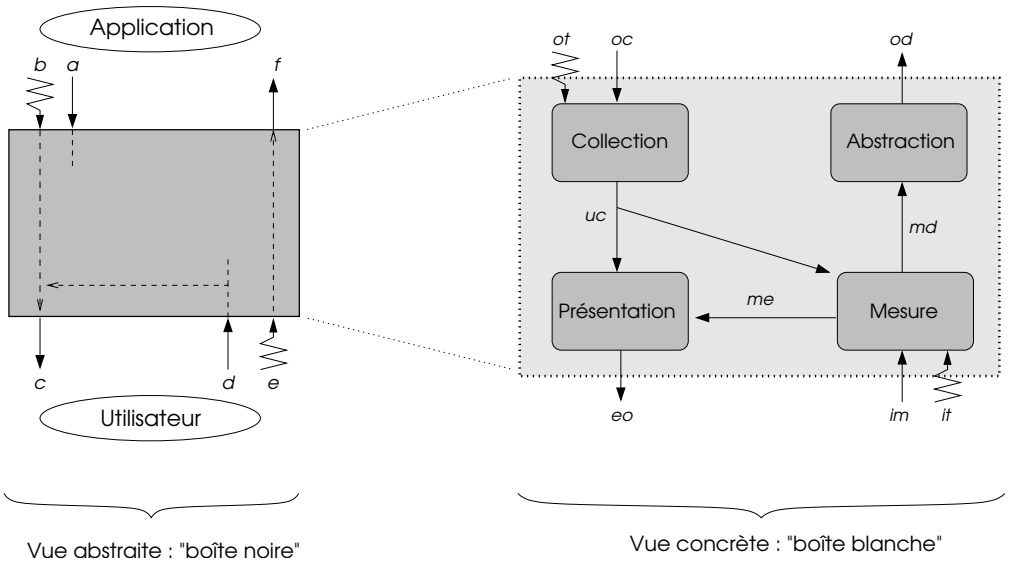


FIG. 1.9 – Interacteur Cnuce, adapté de [Paternò, 1993]

Vu de l'extérieur ("boîte noire"), un interacteur peut (figure 1.9 , schéma de gauche) :
- recevoir et accumuler des informations du côté application (a) ;
- recevoir du côté application un signal déclencheur (b) provoquant une émission d'informations du côté utilisateur (c) ;
- recevoir et accumuler des informations du côté utilisateur (d) et émettre un retour du côté utilisateur (c) ;
- recevoir du côté utilisateur un signal déclencheur (e) provoquant du côté application une émission des informations accumulées (f).

Vu de l'intérieur ("boite blanche"), un interacteur est structuré en quatre composants qui communiquent entre eux (figure 1.9, schéma de droite) : le composant *Collection* maintient une représentation abstraite de l'apparence externe de l'interacteur. Lorsque cet élément est déclenché (ot : output trigger), il transmet des informations (oc : output collection) au composant *Présentation* qui met à jour les éléments visibles par l'utilisateur

(*eo* : event output). D'autre part, le composant *Mesure* reçoit et accumule les informations provenant de l'utilisateur (*im* : input measure). Lorsqu'elle est déclenchée, elle les transmet au composant *Abstraction* qui les convertit en données abstraites manipulables par l'application (*od* : output date). La mesure peut également se servir d'informations provenant de la collection (*uc*).

Interacteurs de York

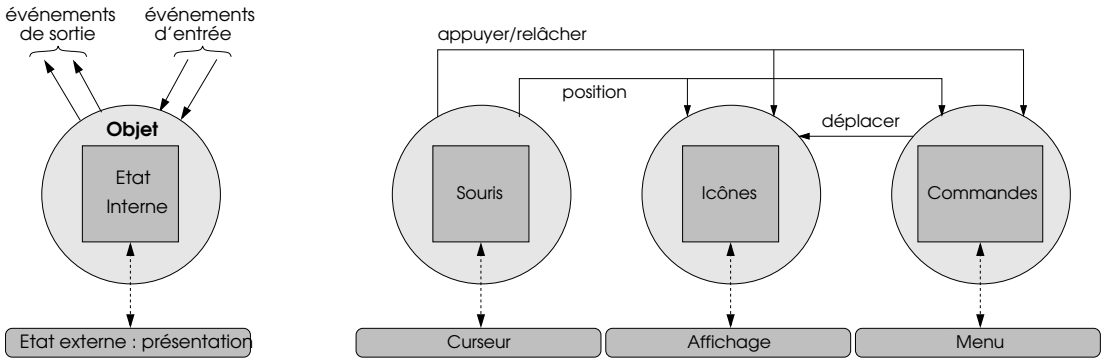


FIG. 1.10 – Interacteur de York : schéma général d'un interacteur et exemple de composition d'interacteurs

L'idée de modéliser un système interactif au moyen d'interacteurs a été reprise par les travaux de Duke et Harrison [Duke & Harrison, 1993] à l'université de York. Dans ce modèle, un interacteur est défini comme un *composant dans la description d'un système interactif qui encapsule un état, les événements qui manipulent cet état, et les moyens par lesquels cet état est rendu perceptible par l'utilisateur.*

À la différence des interacteurs de Cnuce, qui ne contiennent pas de structure de contrôle du système autre que l'émission et la réception d'événements, les interacteurs d'York permettent de modéliser plus finement le dialogue du système dans le sens où l'état de l'interacteur est implicitement décrit.

Le schéma général d'un interacteur d'York est représenté sur la figure 1.10. Un interacteur d'York consiste en un objet comportant un état et communiquant avec son environnement par des événements d'entrée et de sortie. Un interacteur possède également une présentation, qui reflète l'état de l'objet de façon perceptible par l'utilisateur. La *relation de rendu* spécifie les présentations possibles pour un état (et éventuellement un historique) donné de l'objet.

La figure 1.10 montre comment un comportement simple de système interactif peut être décrit par des combinaisons d'interacteurs. Dans cet exemple, des icônes (disques, fichiers, répertoires) peuvent être manipulées par une souris représentée par un curseur. Une icône peut être sélectionnée ou désélectionnée en cliquant à son emplacement, ce qui a pour effet de changer son apparence. Enfin, les icônes sélectionnées peuvent être déplacées en cliquant sur la commande "déplacer" dans un menu, puis en bougeant la souris.

Un ensemble d'opérateurs de composition a été défini. Le résultat d'une composition décrit alors un réseau de processus communicants.

Deux évolutions de ce modèle ont été proposées : les objets abstraits d'interaction ("*Abstract Interaction Object*") [Duke & Harrison, 1993] basés sur un modèle à états et une deuxième approche basée sur un modèle événementiel [Duke & Harrison, 1995].

Interacteurs du Cert

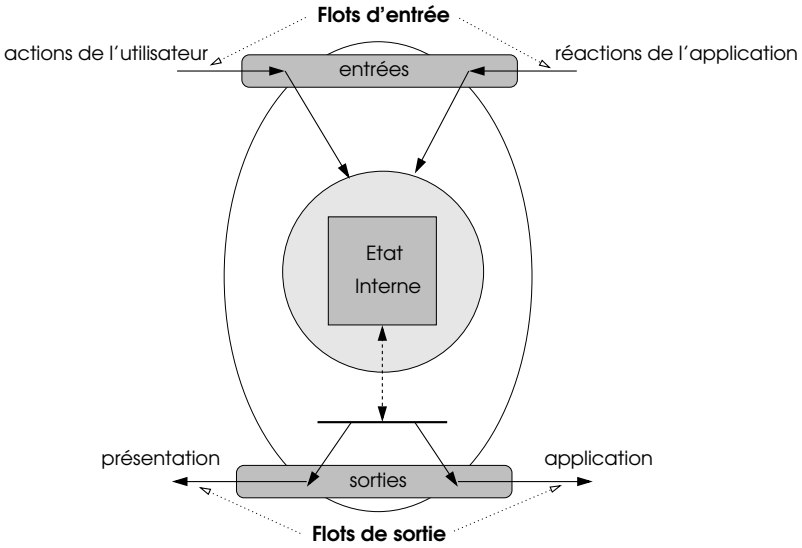


FIG. 1.11 – Interacteurs du Cert à base de flots de données et de contrôle

L'approche proposée par le Cert[1] [Roche, 1998] s'appuie sur le concept d'interacteur de York. Cette nouvelle approche permet la modélisation d'IHM suivant le paradigme de l'approche synchrone, paradigme très exploité dans le domaine des systèmes réactifs.

L'auteur propose une modélisation des interacteurs de York dans le langage Lustre [Halbwachs *et al.*, 1991]. Le langage Lustre est un *langage à flots de données synchrones*. Un système est décrit en Lustre comme un réseau d'opérateurs (ou nœuds) agissant en parallèle. Les opérateurs, définissant un système équationnel reliant les flots de sortie aux flots d'entrée, transforment les flots d'entrée en flots de sortie à chaque top d'horloge. Un flot, constituant une séquence de valeurs peut également être interprété comme un événement.

Suivant la figure 1.11, 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. Le système équationnel définissant le corps du nœud Lustre, définit alors de manière déterministe la valeur des flots de sortie en fonction des flots d'entrée. Ces derniers sont envoyés à la présentation ou à un autre interacteur.

Par souci de précision, notons que les notations Lustre des interacteurs du Cert dénotent des Systèmes de Transitions Étiquetés (STE). En ce sens, les interacteurs du Cert sont bien plus qu'un simple modèle d'architecture puisqu'ils permettent également de décrire le dialogue d'une interaction Homme-Machine et d'en faire la vérification. Les notions

[1] Centre d'Études et de Recherches de Toulouse (ONERA)

de modèles de dialogue et de vérifications de propriétés sont exposées dans les sections suivantes.

L'approche par interacteur à flots de données permet également la description de systèmes complexes au moyen de la composition d'interacteurs. Cette composition est codée naturellement en Lustre par la composition de nœuds.

1.4.4 Les modèles hybrides

Ce type de modèle utilise les modèles globaux pour la structuration globale des modules et exploite les modèles multi-agents pour la description interne des modules. Le modèle le plus représentatif de ce type est PAC-Amodeus [Coutaz, 1990] qui utilise les mêmes composants que le modèle ARCH et détaille le composant contrôleur de dialogue en un ensemble d'objets coopératifs de type PAC.

1.5 Les modèles de description du dialogue

Les modèles de description du dialogue ou *modèles de dialogue* permettent de représenter la dynamique de l'interaction homme-machine. Ce dialogue est lié à la sémantique du système interactif (ce qu'il doit faire) et à la présentation (visualisation du système) [Dix *et al.*, 1993]. Parmi la multitude de formalismes permettant la description du dialogue [Brun, 1998], seules les formalismes à base d'états et les formalismes à base d'événements sont présentés dans ce qui suit.

1.5.1 Modèles à base de Systèmes de Transitions

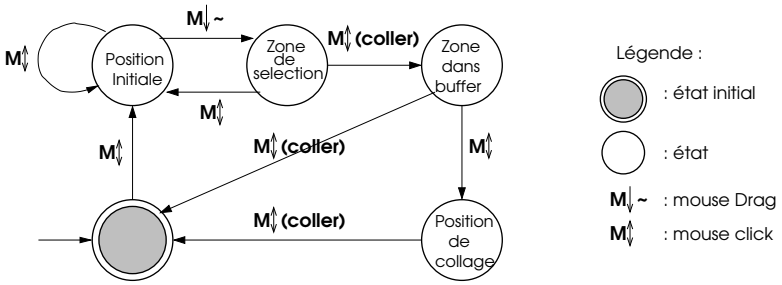


FIG. 1.12 – Exemple de système de transition : *Copier/Coller une zone de texte.*

Historiquement, les travaux sur les automates à états, appelés Systèmes de Transitions Étiquetés (STE), sont les premiers à avoir représenté le dialogue d'une application interactive. Un STE est un cadre mathématique formalisé permettant le raisonnement et qui dispose d'une représentation graphique.

Dans ce cadre, un système interactif est représenté 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 des transitions) qui permettent le passage d'un état à un autre au moyen des transitions.

[Jacob, 1983] fut le premier à spécifier le comportement d'une interface à l'aide de STE. Dans cette description, les états constituent les différents états possibles de l'interaction et les arcs sont étiquetés par les actions de l'utilisateur. L'exemple donné en figure 1.12, repris de [Baron, 2003] décrit le système d'état-transitions de l'opération "*Copier/Coller*" sur une zone de texte.

La modélisation de systèmes complexes peut être obtenue par composition des systèmes de transitions représentant chaque sous-système en utilisant des opérations de produits cartésiens et de produits synchronisés. Ce type de technique tend à une complexification des modèles obtenus, voire à une explosion combinatoire du nombre d'états du modèle. La validation et la construction de tels systèmes sont alors délicates.

Plusieurs extensions des automates ont été proposées afin de résoudre les problèmes évoqués ci-dessus. [Woods, 1986] a proposé les réseaux de transitions récursifs (RTN : *Recursive Transition Networks*). Les RTN permettent une structuration hiérarchique d'un automate à états finis. [Wasserman, 1981] définit les réseaux de transitions augmentés (ATN : *Augmented Transition Networks*) qui introduisent la notion de registres (variables d'état de l'automate). Le formalisme des *Statecharts*[Harel, 1987] est également une extension des automates qui permet l'expression du parallélisme et de la synchronisation.

Enfin, les STE sont associés à différentes techniques de preuve permettant la validation de nombreuses propriétés relatives au dialogue homme-machine (sûreté, vivacité, atteignabilité). C'est notamment le cas des notations Lustre des interacteurs du Cert évoquées dans la section précédente. Les différentes techniques de preuves utilisées dans le domaine de l'IHM, ainsi qu'une définition formelle des STE, seront présentées en section 2.2.

1.5.2 Les Réseaux de Petri

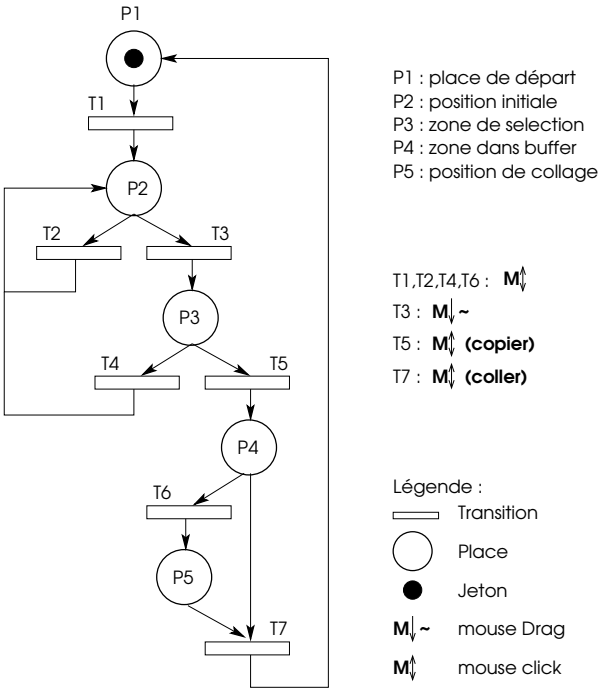


FIG. 1.13 – Exemple de Réseau de Petri : *Copier/Coller une zone de texte.*

Le formalisme dit des Réseaux de Petri (RdP) est un formalisme dérivé des automates. Ce formalisme possède une sémantique formelle et est particulièrement bien adapté à la modélisation des systèmes concurrents.

Un réseau de Petri est composé de deux types d'objet : *les places* et les *transitions*. Les places correspondent aux 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 définissant le *marquage du réseau*. Places et transitions sont reliées par des *arcs orientés*. La figure 1.13 présente le réseau de Petri de l'exemple du "*Copier/Coller*" utilisé précédemment.

Les réseaux de Petri présentent les mêmes inconvénients que les automates du point de vue de la modularité et de la composition : le système est représenté par un réseau de Petri unique amenant de ce fait le problème de l'explosion du nombre d'états. Pour résoudre ces problèmes, des extensions aux réseaux de Petri ont été définies dans la littérature :

- les *Réseaux de Petri à Objets* (RPO) [Sibertin-Blanc, 1985] utilisent le concept de structuration lié aux langages à objets ;
- les *Objets Coopératifs* [Bastide, 1992] introduisent la notion de modularité en définissant un système de communication entre les objets ;
- les *Objets Coopératifs Interactifs* (ICO) [Palanque, 1992], à la différence des Objets Coopératifs sont spécialement dédiés à la modélisation des IHM. Ils introduisent notamment la notion de présentation au sein des objets manipulés.

Si les réseaux de Petri ont principalement été utilisés pour la modélisation du dialogue d'une application, on trouve également quelques travaux [Ezzedine & Kolski, 2005] qui exploitent ce formalisme pour la modélisation des activités cognitives de l'utilisateur suivant différents modes de fonctionnement du système (normal ou anormal).

1.5.3 Modèles à base d'événements

Contrairement aux modèles précédents, les modèles à base d'événements ne se concentrent pas sur la notion d'état, mais sur les événements permettant d'accéder à ces états. Ces modèles reposent sur les concepts d'événements, de gestionnaire d'événements et de traitements [Palanque, 1992]. Lorsqu'un événement est émis, il est pris en charge par le gestionnaire d'événements qui le dirige vers une procédure de traitement qui exécute une modification d'état du système suivant le type et les paramètres de l'événement.

Les modèles à base d'événements possèdent un pouvoir d'expression important (concurrence par exemple) et sont largement utilisés dans la plupart des langages de programmation (notamment le langage Java/Swing) et dans plusieurs langages de modélisation (Lustre [Halbwachs *et al.*, 1991], B événementiel [Abrial, 1996], [Abrial, 2003b]).

De plus amples renseignements sur les modèles à base d'événements peuvent être trouvés dans [Tarby, 1993].

1.6 Propriétés des IHM

Comme tout système logiciel, les IHM doivent satisfaire un ensemble de propriétés. Parmi ces propriétés, on retrouve les propriétés classiques abordées en génie logiciel (sûreté, équité, vivacité, atteignabilité) auxquelles s'ajoutent des propriétés liées à

l'*utilisabilité* du système. Ces dernières propriétés, issues des exigences d'ergonomes et de psychologues, traitent de la capacité de l'utilisateur à réaliser des tâches de manière efficace, confortable et sûre.

Encore une fois, de nombreuses classifications de ces propriétés ont été proposées dans le domaine de l'IHM [Roche, 1998], [Dix *et al.*, 1993], [Duke & Harrison, 1995], [Gram & Cockton, 1997]. On utilise ici la classification proposée par [Roche, 1998] et reprise par [Jambon *et al.*, 2001], [Baron, 2003], [Kamel, 2006]. Cette classification distingue deux classes de propriétés :

1. les **propriétés de validité** caractérisent le fonctionnement attendu et souhaité par l'utilisateur ;
2. les **propriétés de robustesse** concernent la sûreté de fonctionnement du système.

1.6.1 Propriétés de validité

Ce type de propriétés regroupe les **propriétés de complétude** pour la réalisation d'un objectif donné et les **propriétés de flexibilité** pour la représentation de l'information, le déroulement des tâches et pour l'adaptation du dialogue vis-à-vis de l'utilisateur.

Propriétés de complétude

[Gram & Cockton, 1997] distingue la complétude des traitements qui est une propriété du système (*Task Completeness*) et la complétude des objectifs qui est une propriété considérant 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 de complétude des objectifs si l'utilisateur peut atteindre un objectif donné au moyen du système.

Propriétés de flexibilité

La flexibilité d'une IHM caractérise la manière avec laquelle l'utilisateur et le système échangent des informations lors de l'exécution du système. [Gram & Cockton, 1997] propose la distinction de trois sous-catégories pour ce type de propriétés : la représentation de l'information, l'adaptation du dialogue et le déroulement des tâches.

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

- la *multiplicité des périphériques* est la capacité du système à offrir plusieurs périphériques d'entrée (souris, clavier, caméra vidéo...) et de sortie (écran, son...) ;
- la *multiplicité de la représentation* est la capacité du système à offrir plusieurs représentations d'un même concept. Dans le cas d'une horloge, il existe différentes formes différentes de représentation : numérique ou 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 illustre l'usage de données de sortie comme nouvelles données d'entrée. À l'inverse, les valeurs entrées par l'utilisateur sont réutilisables par le système en sortie.

Adaptation du dialogue. Cette catégorie englobe principalement les propriétés liées à l'adaptativité et l'adaptabilité [Browne *et al.*, 1990], [Schneider-Hufschmidt *et al.*, 1993]:

- l'*adaptativité* est la capacité du système à s'adapter à l'utilisateur sans intervention explicite de sa part ;
- l'*adaptabilité* (ou la *reconfigurabilité*) est la capacité du système à supporter la personnalisation de l'interface par l'utilisateur.

Déroulement des tâches. Cette dernière sous-classe de propriétés de flexibilité englobe les propriétés liées à l'atteignabilité, la non-préemption et l'interaction de plusieurs fils *(multithreading)* :

- l'*atteignabilité* est la capacité du système à atteindre un état désiré du système à partir de l'état actuel ;
- la *non-préemption* est la capacité du système à fournir directement le prochain but à l'utilisateur ;
- l'*interaction à plusieurs fils* est la capacité du système à gérer plusieurs processus concurrents, ce qui permet à l'utilisateur de lancer plusieurs traitements en même temps.

1.6.2 Propriétés de robustesse

Selon [Gram & Cockton, 1997], "*un système interactif est dit robuste s'il assiste l'utilisateur dans l'accomplissement de ses tâches sans que des erreurs irréversibles se produisent et s'il donne aux utilisateurs une représentation correcte et complète de la progression de la tâche*". Les propriétés de robustesse englobent les propriétés liées à la visualisation du système comme l'observabilité, l'insistance et l'honnêteté, et les propriétés liées à la gestion des erreurs comme la prédictibilité et la tolérance aux écarts.

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

- l'*observabilité* caractérise la capacité qu'a l'utilisateur d'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.

Gestion des erreurs. Il s'agit des propriétés qui caractérisent la capacité du système à recouvrer ou prévenir les erreurs des utilisateurs :

– la *prédictibilité* 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.

La vérification de ces propriétés permet de garantir l'utilisabilité du système. Face à l'apparition de nouveaux types d'interface plus complexes en termes d'interaction et de présentation (les interfaces multimodales), plusieurs propriétés ont été définies. Ces propriétés, qui s'ajoutent aux propriétés précédentes, permettent de caractériser l'aspect multimodal d'une interface.

1.6.3 Propriétés d'utilisabilité des IHM multimodales : CARE

Les propriétés CARE [Coutaz *et al.*, 1995] pour *Complémentarité, Assignation, Équivalence et Redondance*, définies initialement dans l'espace TYCOON (TYpes and goals of COOperatioN between modalities) [Martin, 1998, Martin & Broule, 1993], ont servi de base pour définir les propriétés d'utilisabilité des interfaces multimodales. Elles définissent les différentes manières avec lesquelles l'utilisateur peut combiner les modalités pour réaliser ses tâches. Ces propriétés sont définies comme suit :

1. la *Complémentarité* désigne l'usage de plusieurs modalités pour la réalisation d'un but ;
2. l'*Assignation* désigne l'usage exclusif d'une seule modalité pour la réalisation d'un but ;
3. l'*Équivalence* désigne le choix offert à l'utilisateur pour réaliser son but avec une modalité de son choix parmi un ensemble de modalités permettant de la réaliser ;
4. enfin, la *Redondance* désigne l'usage parallèle de plusieurs modalités pour réaliser un même but.

Une définition formelle de ces propriétés reposant sur les concepts d'état, de but, de modalité et de relations temporelles a été proposée par [Coutaz *et al.*, 1995]. Une autre définition a également été proposée dans [Nigay & Coutaz, 1997]. Cette dernière définition permet de préciser la définition précédente en prenant en compte les dispositifs physiques et les langages d'interactions associés.

Mise en oeuvre des propriétés CARE. Les propriétés CARE ont servi de base pour la définition de la plateforme de conception multimodale ICARE [Bouchet *et al.*, 2004]. Cette plateforme se base sur un modèle à base de composants qui décrivent les composants programmes manipulés. La plateforme distingue deux types de composants conceptuels :

1. les *composants élémentaires* qui distiguent les modalités ;
2. les *composants de composition* qui se basent sur les propriétés CARE pour définir les opérations de composition des modalités.

Le concepteur choisit les composants élémentaires et les compose en utilisant les composants de composition afin d'obtenir des interactions plus complexes, vérifiant les propriétés CARE qu'il aura choisies. La méthodologie développée consiste à associer un composant ICARE à chaque tâche élémentaire apparaissant dans les feuilles d'un arbre de tâche.

Les travaux de [Kamel, 2006] proposent un cadre formel générique pour la modélisation d'IHM multimodales. Ce cadre générique est notamment utilisé pour la formalisation des propriétés CARE dans la méthode B événementiel [Ait-Sadoune, 2005].

1.6.4 Vérification des propriétés dédiées IHM

Suivant [Kamel, 2006], les propriétés des sytèmes interactifs sont en général établies de trois manières :

1. par le *test* tel que mis en oeuvre en génie logiciel. Majoritairement, il s'agit de tester les propriétés liées au système. Généralement, ces tests ne sont pas exhaustifs et ne peuvent donc pas établir ces propriétés pour tous les cas possibles sauf dans le cadre de systèmes à états finis. Notons que plusieurs travaux dans le domaine de l'IHM s'intéressent à la génération automatique de tests ;
2. par l'*expérimentation* en observant les utilisateurs. Ceci concerne en général les propriétés d'utilisabilité. Dans ce cas, plusieurs méthodes et techniques ont été définies par les ergonomes et psychologues. Comme pour les tests effectués pour valider le système, l'observation ne permet pas de confirmer dans l'absolu la vérification des propriétés puisque l'observation ne fait appel qu'à une catégorie particulière d'utilisateurs ;
3. par *la preuve* en utilisant des formalismes mathématiques pour modéliser le système et les propriétés. Ces techniques formelles, association d'un langage formel pour la modélisation et d'une technique de preuve exploitant ces modèles, peuvent être utilisées pour prouver des propriétés liées au système et à l'utilisateur. Ces méthodes formelles permettent de confirmer la vérification des propriétés avant ou après l'implémentation.

Bien entendu, ces différentes techniques pour la vérification des systèmes interactifs sont complémentaires, et aucune d'entre elles ne peut se vanter de prendre en compte l'ensemble des propriétés dédiées IHM. Ces méthodes doivent donc être utilisées conjointement dans un cadre méthodologique restant à définir.

Vérification des IHM multimodales. Du point de vue des applications multimodales, la vérification ou la validation des propriétés CARE sont effectuées soit par le test et l'expérimentation soit par construction comme le propose la plateforme ICARE. Cependant dans le cas des approches de vérification par construction se pose toujours le problème de la validité des constructions, composants et opérateurs de composition utilisés sur la plateforme concernée. Les travaux menés au sein du projet RNRT VERBATIM [VERBATIM, 2003 2007] sont à notre connaissance les premiers travaux à proposer un cadre formel permettant la vérification de propriétés CARE.

Le projet VERBATIM[2] a étudié l'utilisation des méthode formelles dans une démarche d'aide à l'ingénierie d'interfaces multimodales construites à partir de la plateforme ICARE. Plus particulièrement, ce projet a permis l'élaboration d'une méthode de validation et de test à la fois formelle et garantissant une acceptabilité optimale de la part de l'utilisateur [Jourde *et al.*, 2006]. Les travaux de [Aït-Ameur *et al.*, 2005a],

[2]Rapports du projet RNRT VERBATIM disponible sur : http://iihm.imag.fr/nigay/VERBATIM/

[Aït-Ameur *et al.*, 2006b] et [Kamel, 2006] ont exploité les langages formels B et SMV pour la modélisation et la validation de systèmes interactifs multimodaux. Enfin, les travaux de [d'Ausbourg & Durrieu, 2006] ont proposé une technique d'analyse statique pour la validation de code exploitant le langage formel Promela.

1.7 Développement des IHM : Outils de conception

Le développement des systèmes interactifs est de plus en plus complexe, d'une part du fait de la complexification des modalités et des dispositifs d'interaction utilisés et d'autre part en raison de la taille sans cesse croissante des systèmes à interfaçer. En outre, le concepteur doit assurer l'utilisabilité du système en intégrant au plus tôt l'utilisateur dans le cycle de développement.

Rappelons que le développement de l'IHM représente 48% du code dévolu à un système interactif et monopolise 50% du temps imparti au processus de développement [Myers & Rosson, 1992].

Afin d'alléger la charge du concepteur d'IHM, plusieurs outils d'aide au développement ont été créés ces vingt dernières années. Ces outils se sont développés successivement, chaque étape s'appuyant sur la précédente. [Party, 1999] a utilisé la métaphore de la *boîte gigogne* où chaque nouvelle boîte est conçue au-dessus de la précédente formant plusieurs couches (cf. Figure 1.14). De ce point de vue, plus l'outil est de haut niveau, plus il offre des mécanismes évolués (vérification, aide au développement...) et plus la conception du système est "facilitée".

Plusieurs études ont présenté une classification des outils de conception [Coutaz, 1990], [Party, 1999], [Fekete & Girard, 2001]. On retiendra pour cet exposé les travaux de [Texier, 2000] qui propose de classer ces outils suivant deux approches : l'*approche ascendante* regroupant les outils de type boîte à outils, squelette d'applications et générateur d'interfaces et l'*approche descendante* regroupant les outils de type Système de Gestion d'Interfaces Utilisateurs et Système Basé sur Modèles.

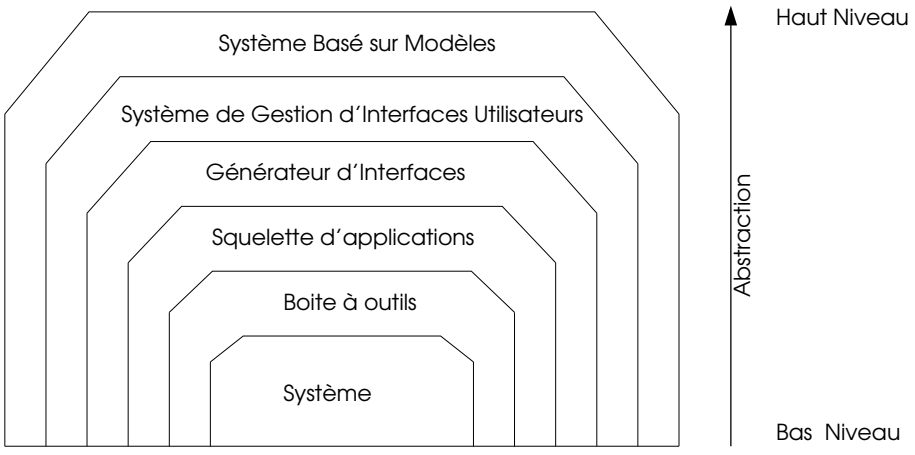


FIG. 1.14 – Boite gigogne : Représentation des familles d'outils de développement.

1.7.1 Approche ascendante : boîte à outils, squelette d'application et générateur d'interfaces

Le principe de l'approche ascendante consiste à construire la partie présentation de l'application à partir de composants graphiques élémentaires tels que des boutons, des fenêtres et des champs de saisie. Le concepteur doit alors relier cette présentation au noyau fonctionnel de l'application. Concrètement, en réaction à une action de l'utilisateur, l'état du noyau fonctionnel est modifié par le biais de fonctions appelées *modifieurs*. Le noyau fonctionnel retourne alors son état à la présentation par le biais de fonctions *accesseurs* conduisant à une modification de l'état de la présentation (modifications des attributs des composants graphiques).

Boîte à outils

Ces composants graphiques élémentaires sont communément appelés *widgets* pour "window gadgets" et sont regroupés dans une *boîte à outils*. Les boîtes à outils fournissent une interface de programmation (API[3]) orientée objet qui encapsule les données. Parmi ces API on trouve entre autres MFC (Microsoft Fundation Class)[4], QT[5], Tk [Ousterhout, 1994], AMULET [Myers *et al.*, 1997] ou Java-Swing[6].

Les composants graphiques définis par ces bibliothèques doivent réagir aux actions de l'utilisateur. Dans ce but, les boîtes à outils définissent un mécanisme d'événements pour gérer le dialogue entre l'utilisateur et la présentation. Ainsi, lors d'une action utilisateur, un événement est généré au sein du système. Les widgets recevant ces événements vont réagir suivant deux comportements :

- un *comportement interne* correspondant à une modification de leur propre apparence. Par exemple, l'apparence d'un bouton dans l'état "pressé" et "non pressé" possèdera une représentation graphique différente. Ce comportement est encodé dans la bibliothèque utilisée ;
- un *comportement externe* associé au résultat souhaité par le concepteur. Ce comportement doit être implémenté par le concepteur. Suivant le type de boîte à outils utilisée, différents mécanismes de traitement d'événements sont utilisés.

L'utilisation de boîtes à outils présente plusieurs avantages : elle permet notamment de donner un style à l'interface et d'intégrer des critères ergonomiques à travers les comportements internes de chaque composant, réduisant ainsi l'effort d'implémentation du concepteur.

Cependant, plusieurs inconvénients peuvent être formulés. Tout d'abord, ces bibliothèques sont *difficiles à utiliser* : elles sont réservées aux informaticiens et par conséquent elles excluent de leur utilisation les personnes issues du domaine de l'ergonomie ou de la psychologie. De plus, si la conception est mal menée, l'utilisation des boîtes à outils peut aboutir à des *architectures logicielles floues* c'est-à-dire à un manque de structuration du code. Ce manque de structuration rend toute maintenance ou réutilisation de code très

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

difficile. Enfin, la *vérification de telles applications est également difficile*. Afin de juger de l'utilisabilité, il est nécessaire de passer par de longues phases de tests.

Squelettes d'applications

D'autres outils se sont basés sur les boîtes à outils pour aider le concepteur dans sa tâche. Les squelettes d'applications réalisent les fonctions usuelles de l'interface sous la forme d'un logiciel réutilisable et extensible [Coutaz, 1990]. En effet, lorsqu'un concepteur utilise une boite à outils, il est bien souvent amené à programmer un grand nombre de fois les mêmes séquences de code dans des applications différentes. Les outils de type squelettes d'applications codent une fois pour toutes ces séquences sous la forme d'une structure d'application interactive et adaptable. Le concepteur peut alors utiliser ces squelettes en supprimant les portions de code inutiles et en adaptant à son problème celles qui sont inadaptées en ajoutant de nouvelles fonctionnalités.

Selon [Schmucker, 1987], les squelettes d'applications permettent la factorisation des parties de code récurrentes et la diminution de l'effort de développement d'un facteur quatre à cinq par rapport à la seule utilisation de boîtes à outils.

Java-Swing propose plusieurs squelettes d'applications pour gérer le mécanisme *Défaire/Refaire*, la sauvegarde au moyen du mécanisme de *sérialisation*, ou pour se connecter à d'autres sous-systèmes (base de données, serveurs HTTP...).

En contrepartie, l'utilisation d'un squelette d'applications peut se révéler difficile du point de vue de la *compréhension de son fonctionnement*. En effet, ces squelettes sont souvent très abstraits et nécessitent de la part du concepteur une bonne connaissance de leur fonctionnement ainsi que de la boîte à outils sous-jacente. Enfin, l'utilisation des squelettes *limite le domaine d'application* du fait des stéréotypes qu'ils définissent.

Générateurs de présentations

Les générateurs de présentations, également appelés GUI-Builders (*Graphical User Interface Builders*) permettent de créer facilement la présentation graphique d'une interface. Pour cela, le concepteur dessine l'interface par manipulation directe de primitives graphiques représentant les widgets de la boîte à outils sous-jacente. Ainsi, le concepteur visualise directement le résultat et peut dans certains cas tester l'interface comme dans l'outil GILT [Myers *et al.*, 1997]. Ces outils permettent donc le maquettage d'applications de manière simple et rapide et ne nécessitent pas du concepteur d'être un expert en programmation.

Les générateurs de présentation produisent un squelette d'application à partir de l'interface construite. Le concepteur doit alors implémenter la dynamique du dialogue et les appels au noyau fonctionnel pour construire l'application ce qui nécessite cette fois une bonne connaissance de la programmation.

Plusieurs outils commerciaux disposent aujourd'hui de constructeurs d'interfaces intégrés à leur environnement de développement : par exemple Uim/x [Foundation, 1990], Visual Studio[7], Delphi et Borland[8]. Notons qu'Uim/x permet d'assister le concepteur dans la construction de la dynamique du dialogue.

[7]Microsoft Corporation : hhtp://www.microsoft.com
[8]Borland Software Corporation

Encore une fois, ce type d'outils ne prend généralement en charge que la couche présentation de l'application interactive, omettant donc la partie dynamique de l'application : succession des objets graphiques à l'écran. Dans ce sens, BeanBuilder[9] associé à la technologie Java/Beans apporte une contribution dans ce domaine en autorisant la construction de la couche de présentation et l'ébauche d'un *pseudo* dialogue.

Bilan sur les approches ascendantes

Les approches ascendantes privilégient la couche présentation de l'application interactive. La liaison entre la couche présentation et le noyau fonctionnel doit être réalisée par le concepteur au fur et à mesure de la construction de l'interface.

La prise en compte des besoins utilisateur et d'une architecture logicielle est laissée au bon vouloir du concepteur, pouvant mener à des architectures logicielles floues et un logiciel peu utilisable. De plus, ce manque de structuration rend plus difficile la vérification des systèmes construits.

1.7.2 Approches descendantes

Les approches descendantes tentent de résoudre les problèmes évoqués précédemment : manque de structuration du code généré par les approches ascendantes et non-prise en compte des besoins utilisateurs. L'approche consiste à générer la couche présentation d'une application en traduisant ou en interprétant un ensemble de spécifications. Ces spécifications sont écrites dans un langage de haut niveau.

Principes de l'approche descendante : UIMS et MBS

Les outils utilisant les approches descendantes, également appelés *Système de Gestion d'Interface Utilisateur* (SGUI ou UIMS[10]) se basent sur des langages spécialisés de haut niveau d'abstraction. Ces outils permettent de générer des interfaces de plus en plus riches et de plus en plus complexes en intégrant différents types de descriptions : tâches de l'utilisateur, structure et relations entre les informations manipulées par l'application, couche présentation, dialogue, etc.

Les *Systèmes Basés sur Modèles* (MBS[11]) représentent une évolution des UIMS et font référence à un ensemble d'outils qui permettent de construire l'interface de l'application au moyen de plusieurs modèles, chacun d'eux correspondant à une vue particulière du système (modèle de tâches, modèles utilisateur, etc.). Les MBS sont de véritables outils de développement accompagnant le concepteur dans la construction d'un système interactif.

D'après [Szekely, 1996] et adaptée par [Baron, 2003], la figure 1.15 montre les principaux composants d'un environnement MBS.

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

[9] Sun Microsystems : http://www.sun.com
[10] User Interface Management Systems
[11] Model Based Systems

Modèle. Le modèle est le composant principal d'un MBS. Il décrit à haut niveau d'abstraction les caractéristiques de l'application interactive. Ce modèle peut contenir des informations relatives au dialogue, au domaine ou à la présentation. Le modèle peut être décomposé en trois catégories [Szekely, 1996] :

1. au plus haut niveau, on trouve les **modèles de tâches** et le **modèle du domaine** de l'application. Le modèle du domaine représente les données et les fonctions (accesseurs et modifieurs) supportées par l'application. En d'autres termes, il s'agit d'une abstraction du noyau fonctionnel de l'application. Les modèles de tâches décrivent les tâches que l'utilisateur doit pouvoir réaliser sur l'interface. Les notations présentées en section 1.3 (UAN, MAD, CTT) sont exploitables à ce niveau. Les feuilles de ces arbres de tâches correspondent aux opérations élémentaires directement réalisables avec l'application ;
2. le second niveau du modèle est appelé **spécification abstraite** de l'interface. Il représente la structure générale de l'interface. Ce modèle est décrit en termes d'objets d'interaction de bas niveau (sélection d'éléments dans un ensemble), d'éléments de présentation (valeur ou ensemble de valeurs du domaine, constantes de l'application) et d'unités de présentation (abstraction 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 est la **spécification concrète** de l'interface utilisateur. Elle raffine les éléments du deuxième niveau d'abstraction au moyen d'éléments issus d'une boîte à outils (bouton, champ de saisie...)

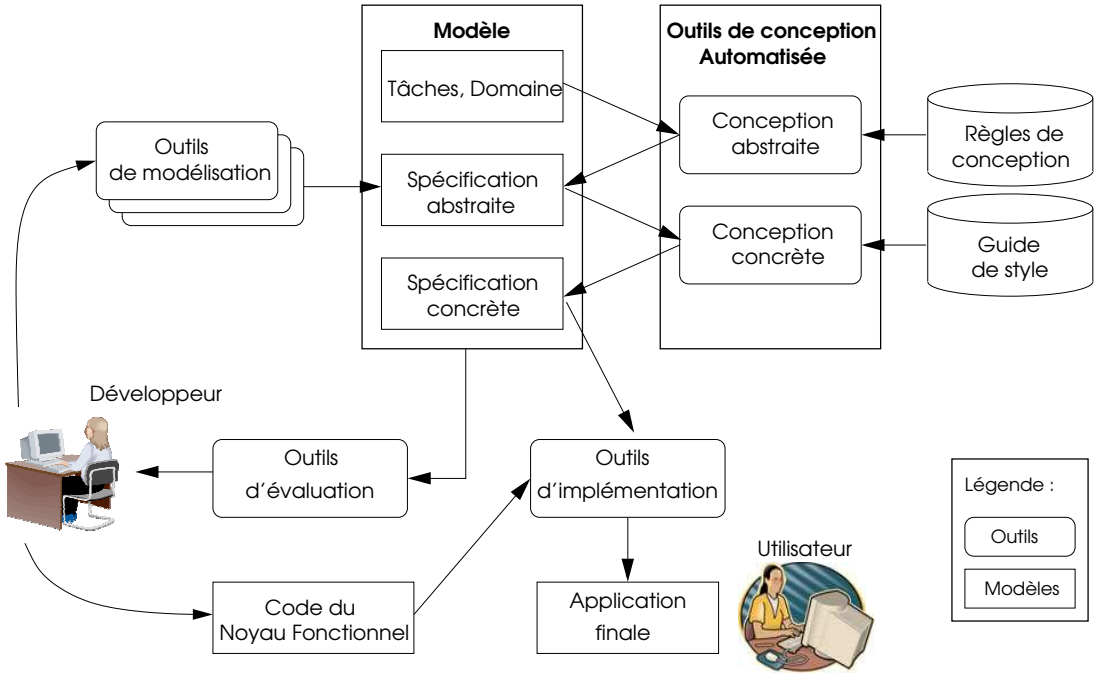


FIG. 1.15 – Environnement de développement d'interfaces à base de modèles

Outils de modélisation. Les outils de modélisation assistent le concepteur dans la construction des modèles de l'application. L'objectif de ces outils est de cacher la syn-

taxe des langages de modélisation et de fournir une interface conviviale pour faciliter la description des spécifications contenues dans les modèles. Ces outils peuvent être des éditeurs de textes pour construire les spécifications textuelles du modèle comme dans ITS [Wiecha *et al.*, 1990] ou MASTERMIND [Szekely *et al.*, 1995], ou encore des formulaires de création d'éléments comme dans l'outil MECANO [Puerta, 1996].

Outils de conception automatisée. Les outils de conception automatisée sont utilisés pour effectuer des transformations sur les modèles. Ces outils permettent aux concepteurs de spécifier seulement certains aspects de l'interface. Notamment, ils sont capables de déduire, à partir du modèle de l'application fourni et non finalisé, les aspects manquants et de les générer. Par exemple, avec Janus [Balzert *et al.*, 1996], le concepteur spécifie uniquement le modèle du domaine : les spécifications abstraites et concrètes sont alors générées par les outils de conception automatique. Si ces outils facilitent grandement la tâche du concepteur, leur utilisation réduit grandement les domaines d'application qu'il est possible de créer. Dans l'exemple de Janus, il est possible de générer uniquement 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 directement être utilisable par une boîte à outils ou un générateur d'interfaces. Certains MBS comme ITS disposent d'un interpréteur permettant d'exécuter les modèles. L'interprétation des modèles offre l'avantage de pouvoir tester rapidement l'interface construite sans avoir à passer par de longues phases de compilation.

Outils de validation. À la différence des approches ascendantes, les MBS disposent d'outils de validation. Ces derniers analysent et évaluent les modèles. C'est ensuite au concepteur de prendre en compte les résultats de la validation et éventuellement de modifier les modèles en cas d'incohérences. Par exemple, ces outils permettent de vérifier la possibilité d'atteindre toutes les fonctionnalités du système (propriétés d'atteignabilité).

Cependant, ces outils de validation n'interviennent, pour leur majorité, qu'au stade final de la modélisation. En effet, pour la plupart, ces outils ne s'intéressent qu'au niveau de la spécification concrète. À notre connaissance seul l'outil SUIDT (Safe User Interface Development Tool) [Baron, 2003] base sa conception sur une description formelle du noyau fonctionnel et permet la validation de l'application par simulation des modèles de tâches.

De plus amples informations concernant les MBS peuvent être trouvées dans [Tabary, 2001], [Pinheiro da Silva, 2000], [Calvary, 1998], [Baron, 2003].

Bilan sur l'approche descendante

Même si les outils fondés sur l'approche basée sur modèles sont plus sophistiqués que les outils de l'approche ascendante, ils demeurent aujourd'hui confinés aux études de recherche peinant à trouver leur place dans des outils commerciaux. Leur principal atout est de permettre au concepteur de raisonner sur des modèles et par conséquent de créer des applications mieux structurées et plus sûres du point de vue de l'utilisabilité du système. Cependant, le raisonnement ne s'effectue que sur des éléments partiels de la modélisation sans permettre la validation à tous les niveaux de la modélisation.

Une des raisons de cette lacune est l'absence de sémantique formelle dans les langages permettant la description des modèles. Peu d'approches proposent des modèles à sémantique formelle mis à part l'outil SUIDT [Baron, 2003].

1.8 Bilan sur le domaine de l'IHM

La conception de systèmes interactifs utilisables, c'est-à-dire respectant des critères d'utilisabilité, nécessite d'intégrer les besoins de l'utilisateur au plus tôt dans le cycle de développement.

Un grand nombre de propriétés inhérentes aux IHM (propriétés de validité, de robustesse) doivent être vérifiées pour garantir l'utilisabilité du système interactif.

Pour assurer la qualité et l'utilisabilité du système produit, la conception d'un système interactif repose sur l'utilisation de techniques issues du monde du génie logiciel (cycle de développement, modélisation, vérification, validation, etc.) et sur l'exploitation de notations et de modèles (modèles de l'utilisateur, modèles de tâches, modèles d'architecture et modèles de dialogue) provenant de différentes communautés (informaticiens, ergonomes, psychologues).

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

Notamment, 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 utilisées ne peut être effectué. Une spécification semi-formelle ne permet pas de garantir que le typage et les propriétés pertinentes du système pourront être validées.

Du point de vue des modèles de tâches, leur sémantique est généralement pauvre : il est difficile d'établir de manière non ambiguë l'ordonnancement des tâches spécifiées par le modèle. Actuellement, la phase de vérification de l'ordonnancement des tâches passe par de lourdes phases de tests. En outre, ces modèles ne sont pas exploitables directement du point de vue de la programmation logicielle.

Ainsi, l'absence de sémantique formelle rend empirique l'application de ces notations et modèles.

Un des objectifs de recherche dans le domaine de l'Interaction Homme-Machine est d'étendre les techniques traditionnelles de conception et de vérification des IHM par l'introduction de méthodes formelles. Ces techniques font l'objet du chapitre suivant.

Chapitre 2

Développement formel et analyse statique de systèmes interactifs

"La mathématique n'est pas une science dans le même sens que les autres.
Elle est certes scientifique, et même de façon exemplaire, par sa rigueur, sa précision,
sa certitude, mais elle n'est pas une connaissance des choses.
C'est un langage cohérent, mais indifférent au réel."
Blanché, L'Epistémologie

2.1 Introduction

Techniques formelles et IHM. Un des objectifs de recherche dans le domaine de l'Interaction Homme-Machine est d'étendre les techniques traditionnelles de conception des IHM par l'introduction de méthodes formelles. L'objectif est non seulement de répondre au problème de la qualité et de la fiabilité des systèmes, mais également de réduire le coût de leur développement et de leur vérification. La section 2.2 présente l'état actuel des travaux concernant des techniques formelles pour la conception des IHM qui exploitent les modèles et notations exposées dans le chapitre précédent.

Analyse Statique et IHM. Si l'utilisation des méthodes formelles connaît des résultats prometteurs, force est de constater que ces techniques peinent à trouver leur place au sein d'un processus de développement où l'utilisation de boîtes à outils et de générateurs d'interfaces est pratique courante. Du point de vue de la validation du système, le test logiciel, tâche laborieuse pour le concepteur de logiciels et très coûteuse [Myers *et al.*, 1996, Perry, 1995] est actuellement la seule technique utilisée en pratique.

Une approche récente dans le domaine de l'interaction Homme-Machine se tourne vers les possibilités offertes par l'*Analyse Statique* ou *Dynamique* de code source. Ces analyses peuvent être utilisées suivant différents objectifs : génération de cas de test [Memon *et al.*, 2003], reverse-engineering [Gannod & Cheng, 1999], [Suberri, 2003],

[Silva *et al.*, 2006], extraction de modèles (formels ou non) et abstraction pour la validation [Schmidt & Steffen, 1998], [d'Ausbourg & Durrieu, 2006], [VERBATIM, 2003 2007]. La section 2.3 de ce chapitre présente les principes de l'Analyse Statique et les différentes applications de ces principes dans le cadre des applications interactives ou non interactives. En particulier, un point sera réalisé sur l'utilisation de ces techniques dans le cadre d'applications codées en Java et sur une technique particulière d'Analyse Statique appelée opération de *slicing*.

Enfin, la dernière section de ce chapitre présente une synthèse des travaux de recherche en cours dans le domaine de l'interaction homme-machine et présente la contribution de ce mémoire dont l'objectif est la définition d'une approche de validation formelle des systèmes interactifs.

2.2 Développement formel de systèmes interactifs

L'utilisation des méthodes formelles dans le développement des systèmes a été motivée par le besoin de développer des logiciels sûrs. Les méthodes formelles permettent un raisonnement rigoureux à base mathématique permettant d'appréhender le comportement et les propriétés d'un système.

Une *modélisation formelle* consiste en la description d'un système et de ses propriétés en utilisant un langage à *syntaxe* et à *sémantique formelles*. La syntaxe décrit les constructions autorisées du langage et la sémantique décrit mathématiquement, et par conséquent sans ambiguïté, le sens des constructions syntaxiques. Suivant le degré d'abstraction utilisé dans la modélisation, on peut parler de spécification, conception ou implémentation pour désigner un modèle formel.

En général, le développement formel d'un système nécessite un modèle formel pour la représentation du système en question, un modèle formel pour l'expression des propriétés à vérifier et un *système de preuve* (ou procédure de vérification) permettant de prouver que le modèle du système vérifie les propriétés. On distingue deux systèmes de preuve : la vérification sur modèle (*model-checking*) et la preuve sur modèle (*theorem-proving*). Suivant la nature du langage formel utilisé pour la modélisation, l'une ou l'autre des techniques de preuve peut être utilisée.

Parmi les différents langages formels définis, certains sont utilisés pour modéliser uniquement le système, d'autres permettent de modéliser uniquement les propriétés et enfin certains peuvent représenter à la fois le système et ses propriétés.

Cette section présente une classification des différents types de modèles existants, les systèmes de preuve utilisés pour la vérification ainsi que quelques modèles formels utilisés dans le domaine de l'IHM.

2.2.1 Classification des modèles formels

De nombreuses approches de modélisation formelle ont été proposées dans la littérature. Suivant [Kamel, 2006], on peut classer ces modèles suivant trois types :

1. les **modèles sémantiques** décrivent aussi bien le système que les propriétés. Ces modèles décrivent les comportements d'un système et les propriétés sont alors définies comme des comportements attendus du système. Il est possible de distin-

guer dans cette approche celles se basant sur la définition explicite d'un état du système (système de transitions) et celles basées sur des systèmes d'événements. Parmi la multitude de modèles sémantiques existante, on peut citer : les Statecharts [Harel, 1987], les réseaux de Petri [Peterson, 1981], Lustre [Halbwachs *et al.*, 1991] et les algèbres de processus comme CCS [Milner, 1980], CSP [Hoare, 1978] et LOTOS [ISO84, 1984] ;

2. Les **modèles syntaxiques** sont fondés sur deux modélisations complémentaires : une modélisation statique décrivant les entités constituant le système (les variables du système) et une modélisation dynamique décrivant les changements d'états autorisés. Les changements d'état sont exprimés en termes d'actions définies sur les entités et de propriétés devant être vérifiées avant l'action (préconditions) et après l'action (post-conditions). Ces modèles sont qualifiés de syntaxiques car ils utilisent une logique associée à un système de preuve fondé sur la réécriture d'expressions logiques. Les méthodes Z [Spivey, 1988], VDM [Jones, 1990] et B [Abrial, 1996] font partie de cette classe.
3. Les **modèles logiques** décrivent les propriétés de systèmes sous forme de formules logiques. Une procédure de vérification est mise en oeuvre pour la vérification de ces propriétés (*model-checking*) sur un modèle. On distingue dans cette classe les logiques temporelles CTL, PLTL, etc., et la logique du premier ordre.

2.2.2 Les techniques de vérification formelle

Comme évoqué précédemment, les techniques de vérification formelles sont traditionnellement classées selon deux grandes catégories : la vérification sur modèle ou "*model-checking*" et la démonstration de théorème ou "*theorem-proving*".

Model-Checking

Ce système de preuve est principalement adapté aux formalismes de type automates. Cette technique est basée sur le parcours exhaustif des états d'un système. Il existe deux approches pour le *model-checking*.

Première approche. La première approche, développée par [Clarke & Emerson, 1982] et [Queille & Sifakis, 1981], consiste à modéliser le système par un système de transitions fini et à exprimer ses propriétés dans une logique temporelle [Manna & Pnueli, 1992]. La procédure de vérification consiste alors en un algorithme vérifiant que le système de transitions est un modèle pour la spécification exprimée par la propriété temporelle.

Seconde approche. Dans cette seconde approche, système et propriétés sont exprimés dans le même formalisme, en l'occurrence des formalismes basés sur les automates. Les deux automates sont alors comparés pour vérifier si le système est conforme à sa spécification. La conformité peut être exprimée soit par une relation d'inclusion de langage reconnu [Kurshan, 1994] ou par une relation d'équivalence observationnelle [Cleaveland *et al.*, 1993]. Cette approche concerne les propriétés opérationnelles qui peuvent être exprimées par un système de transitions.

Ainsi, la vérification de la propriété consiste à comparer deux ensembles de comportements. Deux voies existent pour représenter ces comportements : linéaire avec les traces ou arborescente avec des arbres. De plus amples informations sur ces deux approches (linéaire et arborescente) sont accessibles dans [Kamel, 2006].

La technique du *model-checking* est implémentée dans des outils appelés *model-checkers*. Les *model-checkers* se différencient par le langage de spécification du système et des propriétés utilisé, et par l'algorithme de vérification. Parmi les *model-checkers* on peut citer SMV et SPIN.

L'avantage du *model-checking* est que cette technique est complètement automatique. En outre, dans le cas d'une propriété non vérifiée, un *model-checker* est capable de retourner un contre-exemple mettant en évidence la propriété prise en défaut. Ces informations sont alors exploitables par le concepteur pour réaliser des mises au point sur sa spécification.

Le principal problème auquel peut être confronté le *model-checking* est le problème de l'*explosion combinatoire*. Plusieurs travaux se sont intéressés à la définition de représentations efficaces des systèmes de transitions afin d'augmenter le nombre d'états manipulables. Aujourd'hui, les model-checkers peuvent aisément traiter des systèmes composés de 100 à 200 variables d'état[1] et constitués de 10^{120} états atteignables [Burch *et al.*, 1994]. L'utilisation de techniques d'abstraction appropriées peut permettre d'aborder les systèmes à états infinis [Cousot & Cousot, 1999].

Technique de preuve sur modèle

La technique de preuve sur modèle ou *theorem-proving* repose sur la définition d'un système formel. Un système formel est constitué d'un ensemble d'axiomes et de règles d'inférence. Avec cette technique, le système ainsi que ses propriétés sont exprimés par des formules logiques du système formel. La vérification d'une propriété consiste à démontrer un théorème, c'est-à-dire à trouver la preuve d'une propriété en exploitant les axiomes du système et les règles de déduction.

Plusieurs outils permettent aujourd'hui de guider le concepteur dans la construction de sa preuve par une séquence de lemmes ou de définitions. Dans de nombreux cas, les théorèmes peuvent être prouvés automatiquement par un *theorem-prover* en utilisant des heuristiques pour l'induction, la réécriture et la simplification.

Il existe également plusieurs outils combinant les deux approches présentées. Par exemple, Analytica combine la démonstration de théorèmes avec l'algèbre symbolique de Mathematica, PVS et STep permettent d'utiliser preuve et model-checking. Enfin, l'Atelier B combine un développement par raffinement, un prouveur et un générateur de code. La présentation de la méthode B fera l'objet de la section 3.2 du chapitre 3.

2.2.3 Modèle formel : Systèmes de transitions et logiques temporelles

Dans ce qui suit un modèle utilisé pour la spécification des systèmes et des propriétés est présenté. La présentation est limitée aux systèmes de transitions étiquetés et à la logique temporelle CTL* dont une restriction donne les logiques LTL et CTL. Ces deux

[1] Voire plus pour SMV par exemple.

formalismes (STE et CTL*) constituent des techniques de modélisation générique adaptées à la modélisation de systèmes interactifs et à la formalisation de propriétés.

Systèmes de transitions

Un système de transitions étiquetées (STE) est une représentation formelle modélisant le comportement dynamique d'un système. Comme évoquée en section 1.5, cette formalisation est notamment utilisée pour modéliser le dialogue entre l'utilisateur et le système (étape de *Conception détaillée* du cycle de développement en V).

Un système de transitions est constitué d'un ensemble d'états, d'un ensemble d'événements, d'un état initial et d'une relation de transition entre les états. Chaque transition est étiquetée par un événement. L'occurrence d'un événement permet la transition éventuelle d'un état à un autre.

Définition : STE. *Un système de transitions étiquetées est défini formellement par un quadruplet* $\mathcal{A} = (Q, E, T, q_0)$ *où :*
- Q *est l'ensemble des états du système ;*
- E *est un ensemble d'étiquettes désignant les actions ou les événements du système ;*
- T *est une relation de transition entre états telle que* $T \subseteq Q \times E \times Q$ *;*
- q_0 *est l'état initial du système tel que* $q_0 \in Q$*.*

Une transition t est donc un triplet $t = (p, a, q)$ que l'on note $p \xrightarrow{a} q$. Un système de transitions dispose d'une représentation graphique : les états sont représentés par des cercles et les transitions par des arcs. Un système de transitions est alors un graphe où une transition $q_i \xrightarrow{e_k} q_j$ est représentée telle que décrite par la figure 2.1.

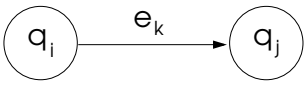


Fig. 2.1 – Représentation graphique d'une transition

L'état initial est représenté comme suit :

Fig. 2.2 – Représentation graphique de l'état initial d'un STE

Une *trace* ou encore *chemin* dans un système de transitions A est une suite σ de transitions $q_i \xrightarrow{e_i} q'_i$ telle que :
- q_i est l'état initial du STE pour $i = 0$
- et $\forall i.(q'_i = q_{i+1})$

Un chemin est donc une suite de transitions qui s'enchaînent et que l'on note souvent : $q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} q_2 \ldots$

On appelle communément *comportement* un chemin dans un graphe représentant le système de transitions.

Structure de Kripke

Une structure de Kripe est un STE enrichi d'une fonction associant à chaque état un ensemble de *propositions* vérifiées dans cet état. Ce formalisme est utilisé comme modèle sémantique pour certaines logiques temporelles.

Définition : Structure de Kripke. *Considérons $P = p_1, p_2, \ldots, p_n$ un ensemble de propositions décrivant des propriétés élémentaires. Une structure de Kripke est un modèle représenté par le quintuplet $\mathcal{K} = (Q, E, T, q_0, l)$ où :*
- *Q est un ensemble fini d'états ;*
- *E est un ensemble fini d'étiquettes de transitions ;*
- *T avec $T \subseteq Q \times E \times Q$ est l'ensemble des transitions ;*
- *q_0 est l'état initial du système tel que $q_0 \in Q$*
- *l avec $l : Q \to \mathbb{P}(P)$ est l'application qui associe à chaque état de Q l'ensemble fini des propriétés élémentaires vérifiées dans cet état.*

L'application l permet d'obtenir les différentes propriétés vérifiées dans un état donné. Elle permet entre autres de décrire les variables d'état qui caractérisent un système de processus et de les observer lors d'un changement d'état. Le lien entre variables d'état et automates (STE) peut être de deux types :
- *Affectations* : une transition peut modifier la valeur d'une (ou plusieurs) variable(s) ;
- *Gardes* : une transition peut être gardée par une condition sur les variables d'états.

Assemblage d'automates : produit cartésien et produit synchronisé

Lors de la description d'un système, on procède souvent par la description des sous-systèmes qui le composent. Dans ce cas, chaque sous-système peut être décrit par un système de transitions. Le système complet est alors décrit par l'ensemble des systèmes de transitions décrivant les sous-systèmes. Le produit cartésien (ou libre) et le produit synchronisé permettent de décrire des systèmes par assemblage des systèmes de transitions de base. Le résultat de ce produit décrit alors le comportement global du système. Généralement, le système de transitions obtenu possède un très grand nombre d'états, si bien qu'il est difficile, voire impossible, de le construire.

Produit cartésien ou produit libre. Considérons un ensemble de n structures de Kripke $\mathcal{A}_i = (Q_i, E_i, T_i, q_{0,i}, l_i)$. Soit "$-$" une nouvelle étiquette permettant d'exprimer l'action fictive. Cette action fictive, également appelée transition de bégaiement, sert à décrire qu'un sous-système de transitions n'effectue aucune transition dans le système de transition global.

Définition : Produit cartésien (produit libre). *Le produit cartésien $\mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_n$ de ces automates, est l'automate $\mathcal{A} = (Q, E, T, q_0, l)$ tel que :*
- $Q = Q_1 \times Q_2 \times \ldots \times Q_n$
- $E = \prod_{i=1}^{n}(E_i \cup \{-\})$
- $T = \{((q_1, q_2, \ldots, q_n)(e_1, e_2, \ldots, e_n)(q_1^{'}, q_2^{'}, \ldots, q_n^{'}) | \forall i \in 1..n, (e_i =' -' \text{ et } q_i = q_i^{'}) \text{ ou bien } (e_i \neq' -' \text{ et } (q_i, e_i, q_i^{'}) \in T_i)\}$
- $q_0 = (q_{0,1}, q_{0,2}, \ldots, q_{0,n})$
- $l = \bigcup_{i=1}^{n}(l_i(q_i))$

Dans un produit cartésien, chaque composante locale (ou automate) $\mathcal{A}_i$ 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 différentes composantes. De plus, le produit cartésien permet des transitions où toutes les composantes ne font rien.

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

Considérons un ensemble de synchronisations *Sync* (ou bien un vecteur de synchronisation) défini par :

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

Sync indique, parmi les étiquettes du produit cartésien, lesquelles correspondent réellement à des synchronisations (transitions groupées autorisées).

On peut également définir le vecteur de synchronisation en utilisant les états du produit cartésien avec $Sync \subset Q_1 \times Q_2 \times \ldots \times Q_n$.

Informellement, le produit synchronisé est un produit cartésien dans lequel seules les transitions prises dans l'ensemble de synchronisation *Sync* sont autorisées.

Définition : Produit synchronisé. *Formellement, le produit synchronisé* $(\mathcal{A}_1 \parallel \mathcal{A}_2 \parallel \ldots \parallel \mathcal{A}_n)_{Sync}$ *des automates* $\mathcal{A}_i$ *est l'automate* $\mathcal{A} = (Q, E, T, q_0, l)$ *tel que :*

- $Q = Q_1 \times Q_2 \times \ldots \times Q_n$
- $E = \prod_{i=1}^{n}(E_i \cup \{-\})$
- $T = \{((q_1, q_2, \ldots, q_n)(e_1, e_2, \ldots, e_n)(q_1^{'}, q_2^{'}, \ldots, q_n^{'}) \mid (e_1, e_2, \ldots, e_n) \in Sync\ \forall i \in 1..n, (e_i =' -' \text{ et } q_i = q_i^{'}) \text{ ou bien } (e_i \neq' -' \text{ et } (q_i, e_i, q_i^{'}) \in T_i)\}$
- $q_0 = (q_{0,1}, q_{0,2}, \ldots, q_{0,n})$
- $l = \bigcup_{i=1}^{n}(l_i(q_i))$

La définition précédente est fondée sur l'utilisation d'un vecteur de synchronisation composé de transitions.

Logique temporelle

La logique temporelle est une forme de logique spécialisée dans les énoncés et raisonnements faisant intervenir la notion d'ordonnancement dans le temps. C'est en 1977 que A. Pnueli a proposé pour la première fois de l'utiliser pour la spécification formelle des propriétés comportementales des systèmes [Pnueli, 1977].

L'avantage des logiques temporelles est qu'elles disposent d'une *sémantique formelle* et que les propriétés décrites dans ces logiques peuvent être vérifiées de manière automatique sur des systèmes de transitions par *model-checking* grâce à l'énumération des traces du système modélisé. Cette vérification n'est possible que si les systèmes sont finis. Plusieurs logiques ont été définies. Elles se classent en deux catégories : les *logiques temporelles linéaires* et les *logiques temporelles arborescentes*. Les logiques temporelles

linéaires utilisent les traces comme modèle d'interprétation tandis que les logiques temporelles arborescentes utilisent les arbres. La présentation se limite ici à la logique temporelle arborescente CTL* introduite par Emerson et Halpern [Emerson & Halpern, 1986] dont une restriction donne les logiques LTL et CTL.

Syntaxe et sémantique informelle de CTL*. La logique CTL* est une logique arborescente dont découlent les deux logiques CTL (*Computation Tree Logic*) et LTL (*Linear Temporal Logic*). La figure 2.3 présente la grammaire de cette logique.

ϕ, ψ	$::=$	$p_1 \mid p_2 \mid \ldots$	(propositions atomiques)
		$\mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Longrightarrow \psi \ldots$	(combinateurs booléens)
		$\mid \mathrm{X}\phi \mid \mathrm{F}\phi \mid \mathrm{G}\phi \mid \phi \mathrm{U} \psi$	(combinateurs temporels)
		$\mid \mathrm{E}\phi \mid \mathrm{A}\phi$	(quantificateurs de chemins)

FIG. 2.3 – Grammaire formelle de la logique temporelle $CTL^\star$

Les *propositions atomiques* sont utilisées pour caractériser les états qui apparaissent dans les comportements. Ces propositions sont regroupées dans l'ensemble des propositions $P = \{p_1, p_2, \ldots, p_n\}$. Une proposition p_i est vraie dans l'état q d'une structure de Kripke si $p_i \in l(q)$. Un ensemble de combinateurs permet alors de construire des formules logiques plus complexes.

Les combinateurs temporels $\boldsymbol{X}$, $\boldsymbol{F}$ et $\boldsymbol{G}$ permettent de décrire les enchaînements des états au sein des différents comportements. Soit p une proposition de l'état courant. Alors :

- $\boldsymbol{X}p$ énonce que p est vraie dans l'état suivant ;
- $\boldsymbol{F}p$ énonce que p est vraie dans un état futur (suivant l'état courant) sans préciser quel état. Cette expression peut se lire "*la propriété p sera vraie un jour*" ;
- $\boldsymbol{G}p$ énonce que tous les états futurs satisfont la propriété p. Cette expression peut se lire "la propriété p sera toujours vraie" ;
- $\phi \boldsymbol{U} \psi$ énonce que ϕ est vraie jusqu'à ce que ψ le soit. En d'autres termes, "ψ *sera vérifiée jusqu'à ce que* ϕ *le devienne*".

Le combinateur $\boldsymbol{G}$ est le dual de $\boldsymbol{F}$. En effet, $\boldsymbol{G}\phi \equiv \neg\boldsymbol{F}\neg\phi$. Le combinateur $\boldsymbol{F}$ est un cas particulier de $\boldsymbol{U}$ dans la mesure où $\boldsymbol{F}\phi \equiv (true\boldsymbol{U}\phi)$. La combinaison des opérateurs $\boldsymbol{F}$ et $\boldsymbol{G}$ est souvent utilisée. Notamment :

- $\boldsymbol{GF}\phi$ se lit "*toujours il y aura un état tel que* ϕ". L'expression ϕ sera vraie infiniment souvent ;
- $\boldsymbol{FG}\phi$ se lit "*tout le temps à partir d'un certain moment* ϕ *sera vraie*". L'expression ϕ sera vraie tout le temps sauf un nombre fini de fois.

À partir d'un état, plusieurs futurs sont possibles. Ceci peut être représenté par une vision arborescente des comportements. Dans ce cas, les chemins ou les comportements sont des parcours ou des branches particuliers dans l'arborescence des comportements. Lorsque les comportements sont arborescents, la logique temporelle introduit des quantificateurs de chemins $\boldsymbol{A}$ et $\boldsymbol{E}$ qui permettent de quantifier l'ensemble des exécutions sur l'ensemble des chemins.

- $\boldsymbol{A}\phi$ énonce que toutes les exécutions partant de l'état courant satisfont la propriété ϕ ;

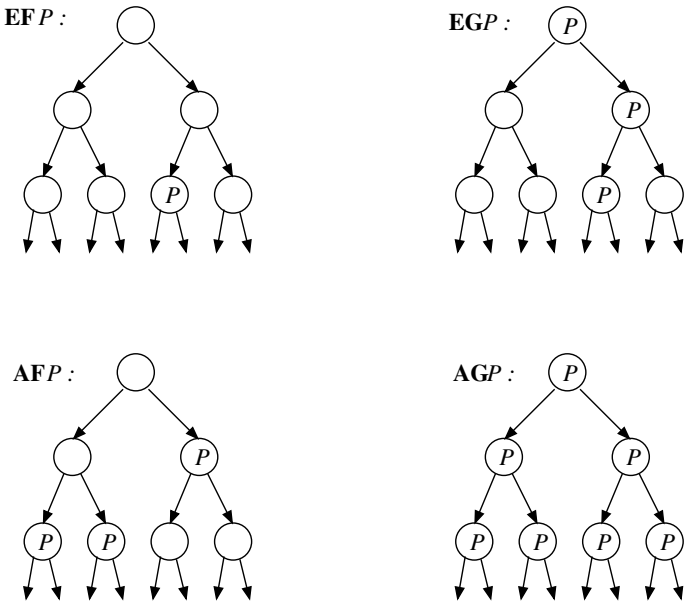


FIG. 2.4 – Quatre manières de combiner E et F

– $\boldsymbol{E}\phi$ énonce qu'il existe une exécution, partant de l'état courant, qui satisfait la formule ϕ.

Bien entendu pour augmenter l'expressivité des propriétés construites, il est également possible de combiner les quantifieurs et les combinateurs temporels. La figure 2.4 présente quatre manières de combiner ces opérateurs temporels. La sémantique informelle de ces combinaisons est la suivante :

– $\boldsymbol{EF}p$ énonce qu'il existe une exécution telle que p sera vraie un jour ;
– $\boldsymbol{EG}p$ énonce qu'il existe une exécution telle que p sera toujours vraie ;
– $\boldsymbol{AF}p$ énonce que quelque soit l'exécution choisie, p sera vraie un jour ;
– $\boldsymbol{AG}p$ énonce que quelque soit l'exécution choisie, p sera toujours vraie.

Sémantique formelle de la logique CTL*. La sémantique des formules CTL* est définie à l'aide de structures de Kripke $\mathcal{A} = (Q, E, T, q_0, l)$. Pour définir cette sémantique, il faut définir la relation de satisfaction $\models$ qui indique dans quelles conditions une propriété est satisfaite.

Définition : Satisfaction. *$\mathcal{A}, \sigma, i \models \phi$, et on lira "au temps i de l'exécution σ, ϕ est vraie", cela en parlant d'exécutions de $\mathcal{A}$ dont on n'exige pas qu'elles soient issues de l'état initial.*

Le *contexte* $\mathcal{A}$ est très souvent laissé implicite et on l'omet dans les écritures. On écrit $\sigma, i \not\models \phi$ pour dire que ϕ *n'est pas satisfaite* au temps i de σ.

La définition de $\sigma, i \models \phi$ se fait par induction sur la structure de ϕ, c'est-à-dire que la valeur de vérité d'une formule composée est donnée à partir des valeurs de vérité de ses sous-formules. La définition de $\sigma, i \models \phi$ est donnée par les expressions présentées en figure 2.5. Dans ces expressions, $|\sigma|$ et $\sigma(i)$ dénotent respectivement la longueur de la trace σ et le i-ème état de σ.

$\sigma, i \models \mathrm{P},$	ssi	$P \in l(\sigma(i))$
$\sigma, i \models \neg\phi,$	ssi	il n'est pas vrai que $\sigma, i \models \phi$
$\sigma, i \models \phi \wedge \psi,$	ssi	$\sigma, i \models \phi$ et $\sigma, i \models \psi$
$\sigma, i \models \mathrm{X}\phi,$	ssi	$i < \lvert\sigma\rvert$ et $\sigma, i+1 \models \phi$
$\sigma, i \models \mathrm{F}\phi,$	ssi	il existe j tel que $i < j \leq \lvert\sigma\rvert$ et $\sigma, j \models \phi$
$\sigma, i \models \mathrm{G}\phi,$	ssi	pour tout j tel que $i \leq j \leq \lvert\sigma\rvert$, on a $\sigma, j \models \phi$
$\sigma, i \models \phi\mathrm{U}\psi,$	ssi	il existe $j, i \leq j \leq \lvert\sigma\rvert$ tel que $\sigma, j \models \psi$ et
	ssi	pour tout k tel que $i \leq k < j$, on a $\sigma, k \models \phi$
$\sigma, i \models \mathrm{E}\phi,$	ssi	il existe σ' tel que $\sigma(0)\ldots\sigma(i) = \sigma'(0)\ldots\sigma'(i)$ et $\sigma', i \models \phi$
$\sigma, i \models \mathrm{A}\phi,$	ssi	pour tout σ' tel que $\sigma(0)\ldots\sigma(i) = \sigma'(0)\ldots\sigma'(i)$, on a $\sigma', i \models \phi$

FIG. 2.5 – La sémantique de CTL*

On peut alors introduire une notion dérivée, "l'automate $\mathcal{A}$ satisfait ϕ", notée $\mathcal{A} \models \phi$ et définie par : $\mathcal{A} \models \phi$ *ssi* $\sigma, 0 \models \phi$ *pour toute exécution* σ *de* $\mathcal{A}$.

C'est une notion bien commode pour parler de la correction d'un modèle. Mais elle n'est pas élémentaire au sens où elle regroupe la correction de toutes les exécutions (issues de q_0) d'un modèle. Ainsi, $\mathcal{A} \not\models \phi$ n'implique pas nécessairement $\mathcal{A} \models \neg\phi$ (alors que $\sigma, i \not\models \phi$ est équivalent à $\sigma, i \models \neg\phi$).

Logique temporelle linéaire LTL. LTL est une logique temporelle linéaire. Les formules LTL sont interprétées sur des chemins d'exécution d'un système de transitions. Les quantificateurs $\boldsymbol{A}$ et $\boldsymbol{E}$ n'existent pas dans cette logique. Ainsi, le langage de la logique LTL est obtenu à partir du langage de la logique CTL* en retirant A et E.

Une formule de LTL ne peut pas examiner les chemins d'exécutions alternatives issues du choix non déterministe. LTL s'intéresse à toutes les exécutions prises d'une manière linéaire indépendamment les unes des autres. Toutes les formules LTL sont des formules de chemins.

LTL ne permet pas d'exprimer qu'à un certain moment le long d'un chemin, il serait possible de prolonger l'exécution de telle ou telle manière. SPIN est un exemple de model-checker implémentant la vérification de formules de la logique temporelle LTL.

Logique temporelle arborescente CTL. CTL est une logique arborescente obtenue à partir du langage de la logique de CTL* en exigeant que chaque utilisation d'un combinateur temporel $\boldsymbol{X}$, $\boldsymbol{F}$, $\boldsymbol{G}$ et $\boldsymbol{U}$ soit immédiatement sous la portée d'un quantificateur $\boldsymbol{A}$ ou $\boldsymbol{E}$. Cette restriction fait que les formules de CTL sont des formules d'états. Les propriétés exprimées dans cette logique peuvent être vérifiées en utilisant l'état courant et les états qui peuvent être atteints à partir de l'état courant dans l'automate, et non en utilisant une exécution courante.

Cette logique a été utilisée par [Abowd *et al.*, 1995] pour vérifier des propriétés de dialogue dans les systèmes interactifs. SMV est un exemple de model-checker implémentant la vérification de formules de la logique CTL. On trouve également dans [Loer & Harrison, 2000] des schémas génériques de propriétés d'utilisabilité des systèmes

interactifs. Par exemple, la propriété d'atteignabilité est exprimée par la formule CTL ***AGEF***(<*Configuration-cible*>) où <*Configuration-cible*> désigne l'état du système à atteindre.

2.2.4 Démarches de conception formelle

Il existe principalement deux approches pour modéliser formellement un système.

Modélisation descendante ou par décomposition. Cette démarche consiste à modéliser le système global à haut degré d'abstraction. Ce modèle abstrait est ensuite concrétisé en effectuant plusieurs étapes de raffinements. Le raffinement permet d'enrichir la spécification avec de nouveaux éléments de description de la spécification. Cette démarche permet de préciser les modèles de la spécification au fur et à mesure des raffinements c'est-à-dire à chaque étape du développement. Les propriétés sont introduites au fur et à mesure et vérifiées à chaque étape du développement. Le raffinement permet de garantir que les propriétés vérifiées dans les étapes précédentes sont toujours conservées au niveau de la nouvelle étape de raffinement : c'est le principal atout de cette démarche.

Modélisation ascendante ou par composition. Cette approche consiste à modéliser le système par composition de ses sous-systèmes. C'est une démarche permettant la réutilisation de systèmes déjà définis dans la conception de nouveaux systèmes. L'inconvénient majeur de cette approche est qu'elle ne garantit pas la conservation des propriétés des sous-systèmes. Le concepteur doit alors vérifier cette préservation de propriétés.

2.2.5 Utilisation des méthodes formelles pour les IHM

Plusieurs méthodes et techniques formelles, basées sur la preuve ou sur la vérification exhaustive de modèles, ont été utilisées pour la conception et la vérification des IHM. Ces techniques formelles se basent également sur différents formalismes permettant la construction de modèles : systèmes de transitions, réseaux de Petri, algèbre de processus... Cette section présente, sans doute de manière non exhaustive, différents travaux utilisant les techniques formelles dans une démarche de conception et de validation des IHM de type WIMP ainsi que les IHM multimodales.

Model-checking : Utilisation de SMV, LOTOS et Lustre

SMV. SMV a été utilisé par [Abowd *et al.*, 1995] pour la vérification des propriétés du dialogue d'une spécification effectuée par le Simulateur d'Actions (*Action Simulator*). L'IHM est spécifiée à l'aide du simulateur d'actions puis cette spécification est traduite dans le langage d'entrée du model-checker SMV. SMV permet alors de vérifier des propriétés exprimées dans la logique temporelle CTL. Dans ce travail, aucun modèle d'architecture n'est utilisé pour représenter l'IHM. Seul le dialogue est spécifié de manière tabulaire à l'aide du système de production propositionnelle (PPS) [Olsen, 1990]. On trouve également dans ces travaux un ensemble de schémas génériques de propriétés de dialogue comme le non-blocage, la complétude des tâches, etc.

SMV a également été utilisé par [Campos & Harrison, 1999]. Ces travaux proposent de spécifier les interacteurs de York (cf. section 1.4) dans la logique modale MAL [Ryan *et al.*, 1991]. Une traduction de cette spécification vers le langage d'entrée de SMV est alors effectuée afin de vérifier que la spécification satisfait un ensemble de propriétés formalisées en CTL. Contrairement aux travaux précédents, cette spécification ne prend pas en compte uniquement le dialogue de l'application, mais bien toute l'IHM en utilisant l'architecture à base d'interacteurs de York.

Enfin, [Kamel, 2006] propose également une utilisation de SMV pour vérifier les propriétés d'utilisabilité des IHM multimodales. Ces travaux reposent sur la définition d'un cadre générique formalisant les propriétés CARE.

Lotos. Lotos est une algèbre de processus [ISO84, 1984]. Lotos a été utilisé pour modéliser les interacteurs de Pise, Cnuce dans les travaux [Paternò, 1995], [Paternò & Faconti, 1993] et les interacteurs ADC dans les travaux de [Markopoulos *et al.*, 1996], [Markopoulos, 1997]. Dans ces travaux, chaque interacteur est modélisé par un processus Lotos. Le système global est alors obtenu par l'interconnexion des interacteurs via leurs ports de communication.

Paternò centre sa démarche de conception sur l'utilisateur en traduisant une spécification hiérarchique de tâches en une hiérarchie d'interacteurs. Chaque interacteur est alors traduit dans le langage de modélisation Full-Lotos, langage basé sur Lotos pour la spécification du contrôle et sur le langage algébrique ACT-ONE [Quemada *et al.*, 1993] pour la spécification des données. Enfin, la spécification Lotos est traduite en un système de transitions. Les propriétés attendues du système sont alors formalisées dans la logique temporelle basée sur les actions ACTL et vérifiées à l'aide du model-checker Lite. Un ensemble de propriétés vérifiées à l'aide de la logique temporelle ACTL sont définies dans [Paternò, 1995].

Markopoulos [Markopoulos *et al.*, 1996] associe l'approche de Paternò à une autre approche consistant à spécifier les propriétés en Lotos. La relation de conformité entre le système de transitions de l'interface et celui de la propriété est réalisée à l'aide de l'outil CAESAR/ALDEBARAN [Fernandez *et al.*, 1992].

L'avantage de l'utilisation de Lotos est de permettre une description formelle de l'architecture à base d'interacteurs. Cependant, dans les travaux présentés ci-dessus, la spécification, au moment de la génération des systèmes de transitions, doit être traduite vers Lotos basique, ce qui implique la perte d'une partie des données spécifiées. Le problème se pose notamment pour la définition d'événements gardés. Avec la perte des gardes, le système obtenu n'est pas équivalent à celui d'origine. Pour éviter ce problème, il faut donc éviter l'utilisation des gardes dans la spécification Lotos ce qui limite grandement le pouvoir d'expression du langage. En outre, si le langage Lotos est relativement abordable par des non-experts du domaine IHM, ce n'est pas le cas du langage algébrique ACT-ONE. Enfin, ces travaux n'offrent aucune interface aidant le concepteur d'IHM à utiliser cette technique sans être un expert.

Notons que le langage Lotos a également été utilisé pour la spécification d'IHM multimodales. Les travaux de [Coutaz *et al.*, 1993b] et de [Paternò & Mezzanotte, 1994] utilisent Lotos pour décrire les interacteurs modélisant l'interface de l'application multimodale *Matis*. La modélisation de l'IHM débute par la modélisation des tâches en utilisant la notation UAN. Cette modélisation est raffinée jusqu'au niveau des tâches interactives

élémentaires. Ces travaux ont permis de vérifier un ensemble de propriétés liées à l'utilisabilité du système à l'aide de la logique ACTL et du model-checker Lite.

Approche synchrone : Lustre. Lustre [Halbwachs *et al.*, 1991] est un langage synchrone à flots de données. Ce langage a été utilisé dans les travaux de [d'Ausbourg & Roche, 1995], [d'Ausbourg *et al.*, 1996a], [d'Ausbourg *et al.*, 1996b], [d'Ausbourg *et al.*, 1996c], [d'Ausbourg *et al.*, 1997], [Roche, 1998]. Dans ces travaux, les interacteurs du Cert (cf. section 1.4) sont modélisés par un ensemble de nœuds Lustre. L'interface est alors conçue en interconnectant l'ensemble des nœuds interacteurs. Dans cette approche, les propriétés sont exprimées dans le même formalisme que l'interface. Une propriété est spécifiée par un nœud observateur Lustre et elle est vérifiée en connectant le nœud modélisant l'interface avec le nœud observateur. Si la sortie du nœud observateur est toujours vraie, la propriété est alors vérifiée. La vérification est effectuée avec le model-checker Lesar. Ainsi contrairement aux autres model-checker, l'automate représentant le système n'est pas généré avant le processus de vérification, mais pour chaque propriété un nouvel automate est généré.

Cette approche de vérification a été implantée dans une maquette d'environnement permettant d'effectuer un ensemble d'opérations de validation formelle et s'intégrant dans un cycle de conception itératif. Cette approche à la fois expérimentale et formelle est représentée par la figure 2.6.

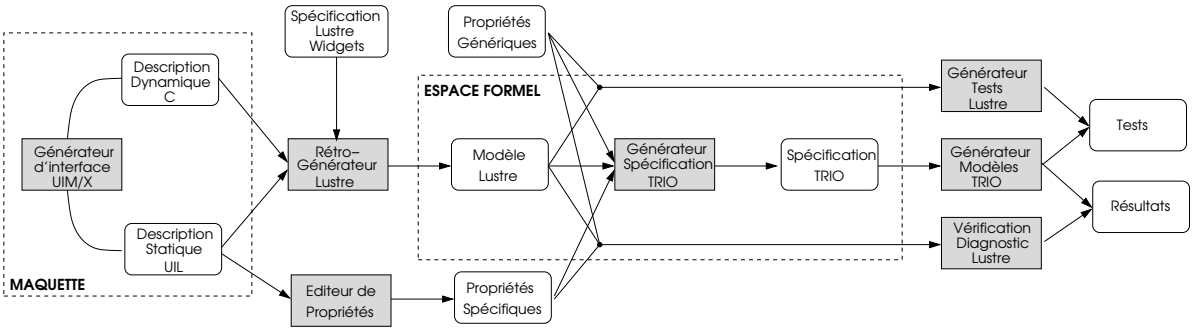


FIG. 2.6 – Maquette d'environnement de conception et de vérification itérative basé sur Lustre.

L'approche se base sur l'utilisation du générateur de présentation UIM/X [Hall, 1991] 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 (*User Interface Language*) [Hall, 1991]. L'aspect comportemental du système est décrit dans le langage C. Une opération de formalisation est effectuée à partir des descriptions UIL et C de l'interface afin d'obtenir une spécification Lustre du système à base d'interacteurs. Cette spécification peut alors être exploitée de deux manières différentes : soit pour la vérification de propriétés en utilisant le model-checker Lesar, soit pour la génération de tests. Dans ce second cas, la spécification Lustre ainsi que la définition de propriétés spécifiques et génériques sont traduites dans la logique TRIO [Ghezzi *et al.*, 1990], [Morzenti *et al.*, 1992]. L'utilisation d'un générateur de modèle TRIO permet alors la génération de tests pour la vérification des propriétés spécifiées [Morasca *et al.*, 1996], [Cazin *et al.*, 1996].

L'avantage de cette méthodologie est de couvrir l'ensemble du cycle de développement. Par contre, notons que les propriétés vérifiées par Lustre sont limitées par rapport à celle exprimée dans des logiques temporelles plus évoluées telles que LTL ou CTL. Seules les propriétés de sûreté et une forme restrictive de vivacité peuvent être vérifiées.

D'autres travaux se sont également penchés sur l'utilisation du langage de spécification Lustre pour la réalisation de tests afin de vérifier des propriétés sur le système étudié. Les travaux de [Jourde *et al.*, 2006] et [Bouchet *et al.*, 2006] utilisent l'outil de tests de logiciels synchrones Lutess [du Bousquet *et al.*, 1999] pour vérifier un système interactif développé avec la plateforme ICARE [Bouchet *et al.*, 2004]. Lutess construit automatiquement un générateur de données de tests pour le programme sous test. Dans ces travaux, les propriétés CARE ont été spécifiées en Lustre puis vérifiées à l'aide de l'outil Lutess. Ces travaux n'utilisent pas les méthodes formelles au cours de la conception du système, mais y font appel uniquement à la fin du développement du système multimodal pour les tests. Cette démarche peut être intéressante pour effectuer des vérifications lorsque le *model-checking* n'est pas possible (explosion combinatoire).

Les réseaux de Petri : Utilisation des ICO

[Bastide, 1992] et [Palanque *et al.*, 1995] utilisent le formalisme des Objets Coopératifs Interactifs (ICO[2]). Cette technique de description formelle est dédiée à la spécification et à l'implémentation des systèmes interactifs. Elle utilise les concepts de l'approche orientée objet (instanciation dynamique, classification, encapsulation, héritage) pour la description de la partie statique du système, de même que les réseaux de Petri pour la description de sa partie dynamique.

Dans cette approche le système est modélisé suivant le modèle d'architecture ARCH. Une fois l'interface représentée dans le formalisme des ICO, la description est ensuite traduite vers les réseaux de Petri standards. La vérification est alors effectuée par analyse du graphe de marquage du réseau de Petri obtenu.

La vérification effectuée peut concerner toute ou partie du système. Cependant, dans cette approche la spécification de propriétés n'est pas assistée par un langage de description des propriétés d'interaction. De même, aucune validation de tâche n'est effectuée et seules les propriétés d'atteignabilité sont vérifiées.

Les travaux de [Palanque & Schyn, 2003], [Schyn *et al.*, 2003], [Navarre *et al.*, 2005], [Schyn, 2005] utilisent le modèle ICO pour représenter la fusion de deux modalités : le geste et la parole. Pour prendre en compte les spécificités de l'interaction multimodale, le formalisme des ICO a été étendu avec un ensemble de mécanismes tel que le mécanisme de communication par production et consommation d'événements. Ce mécanisme permet de synchroniser un ensemble de transitions sur un événement, et permet d'émettre un événement lors du franchissement d'une transition. Notons que ces travaux ne traitent pas les propriétés d'utilisabilités CARE des IHM multimodales.

[2]Interactive Cooperative Object

Theorem-proving : Utilisation de la méthode B et de Z

Utilisation de la Méthode B et B événementiel. Plusieurs travaux utilisant la méthode B [Abrial, 1996] ont été réalisés au sein de l'équipe IHM du LISI[3]. Dans un premier temps [Aït-Ameur *et al.*, 1998a], [Aït-Ameur *et al.*, 1998b] ces travaux se sont intéressés à la formalisation des spécifications d'un système interactif opérationnel tout en assurant certaines propriétés ergonomiques. À partir d'études de cas, ces travaux ont permis de vérifier des propriétés comme l'observabilité, l'insistance ou la robustesse. Dans les travaux de [Aït-Ameur, 2000] une application est implémentée par raffinement jusqu'à la génération du code dans un langage de programmation. Un ensemble de propriétés ont été vérifiées. Dans [Aït-Ameur *et al.*, 2003], une expérience comparative est réalisée en utilisant deux techniques : la preuve avec l'utilisation de la méthode B et le *model-checking* avec l'utilisation du langage Promela et du *model-checker* SPIN.

Dans ces travaux, l'architecture ARCH est utilisée comme modèle d'architecture pour la description et la spécification des différents composants logiciels. Contrairement aux travaux basés sur la technique du *model-checking*, ces travaux échappent[4] au problème d'explosion combinatoire. Le processus de raffinement et l'outil Atelier B [Atelier-B,] génère automatiquement toutes les obligations de preuve à décharger. Par contre, le processus de preuve n'est pas complètement automatique. L'utilisateur peut avoir à conduire la preuve de manière semi-automatique.

D'autres travaux utilisant la même approche ont été réalisés en s'intéressant à une architecture orientée objet, telle que MVC ou PAC. Comme les contraintes du langage B ne permettaient pas d'utiliser directement les modèles à agents, une nouvelle architecture CAV (*Control-Abstraction-View*) [Jambon, 2002] a été développée. Ces travaux s'intéressent à la vérification des propriétés du noyau fonctionnel.

Dans les travaux évoqués précédemment, le contrôleur de dialogue n'est pas modélisé et les tâches ne sont pas validées. Dans [Baron, 2003] B événementiel a été utilisé pour la modélisation du contrôleur de dialogue. Cette démarche a permis de valider des tâches modélisées en CTT. Enfin, citons également les travaux de [Ait-Sadoune, 2005], [Aït-Ameur *et al.*, 2006a], [Aït-Ameur *et al.*, 2006b] qui présentent l'utilisation de cette technique pour la vérification des IHM multimodales.

Utilisation de Z. La méthode Z a été utilisée dans les travaux de [Duke & Harisson, 1997] pour la description des IHM. La démarche consiste à modéliser les interacteurs en utilisant le langage Z. Une spécification Z est décrite par plusieurs étapes de raffinement et les propriétés sont vérifiées à l'aide d'invariants.

Les travaux de [MacColl & D. Carrington, 1998] utilisent conjointement Z et l'algèbre de processus CSP[5] [Hoare, 1985] pour modéliser les systèmes multimodaux. Les auteurs utilisent Z pour modéliser l'aspect statique du système et l'algèbre CSP pour la partie dynamique du dialogue. Ces travaux font cependant abstraction des modalités et évitent

[3]Laboratoire d'Informatique Scientifique et Industriel - ENSMA, Poitiers

[4]Cette affirmation reste relative : l'augmentation du nombre d'états, même si ceux-ci ne sont pas explicitement représentés peut amener à une augmentation ou une complexification des Obligations de Preuve à décharger. Le principe de raffinement de la méthode B permet cependant une diminution de ce nombre d'Obligations de Preuve.

[5]Communicating Sequential Process

ainsi de prendre en charge les spécificités des systèmes multimodaux : les problèmes de fusion et fission des modalités ne sont pas abordés.

2.3 Analyse Statique de code source et IHM

Une approche récente dans le domaine de l'interaction Homme-Machine se tourne vers les possibilités offertes par l'Analyse Statique ou Dynamique de codes sources.

Traditionnellement l'analyse statique était principalement utilisée pour l'optimisation des compilateurs. De nombreux ouvrages [Aho *et al.*, 2006], [Appel, 1998], [Muchnick, 1997], [Nielson *et al.*, 1999] traitent de l'implémentation et du design des compilateurs ou des principes de l'analyse statique en particulier.

Par la suite, l'analyse statique a également été utilisée dans de nombreux outils pour réaliser des tâches d'ingénierie logicielle telles que la compréhension de programmes [Grothoff *et al.*, 2001], [Wright, 1991], la détection de fautes ou de code mort [Flanagan *et al.*, 2002], [Detlefs *et al.*, 1998], le test [Richardson, 1994] ou la rétro-ingénierie (*Reverse Engineering*) [Gopal & Schach, 1989].

Selon la nature des programmes et des propriétés considérés, on parle de *preuve* (les propriétés étant données par l'utilisateur et vérifiées par un assistant de preuve), de *typage* (permettant de vérifier automatiquement la cohérence des structures de données de certaines fonctionnalités du programme), d'*analyse de flot de données* (utilisées à la compilation pour déterminer si certaines optimisations sont applicables), de *vérification exhaustive de modèles* (un modèle fini de l'exécution du programme étant exploré systématiquement, ou du moins en partie si les ressources sont suffisantes), etc.

Toutes ces analyses statiques se formalisent en fin de compte dans la théorie de l'interprétation abstraite ainsi que le démontre un certain nombre de publications :

- pour la preuve : [Cousot, 2002] et [Cousot & Cousot, 1992] ;
- pour le typage : [Cousot, 1997] ;
- pour la vérification exhaustive de modèles : [Cousot & Cousot, 1999], [Cousot, 2000], [Cousot & Cousot, 2000] et [Cousot & Cousot, 2002] ;
- pour l'analyse de flot de données : [Cousot & Cousot, 1977] et [Cousot & Cousot, 2000] ;
- pour l'analyse de programme : [Cousot & Cousot, 1976] et [Cousot & Cousot, 1977].

Depuis quelques années on observe l'émergence d'un nouveau paradigme consistant à utiliser les principes de l'analyse statique afin d'extraire un modèle formel utilisable pour la vérification de propriétés via une technique de preuve (*model-checking* ou *theorem-proving*) [Corbett *et al.*, 2000].

Cette section présente quelques techniques et applications de l'analyse statique. Après une présentation sommaire de la notion de compilation et des représentations intermédiaires utilisées en analyse statique, telles que les arbres de syntaxe abstraite et les graphes de contrôle, les graphes de dépendances système sont présentés. Le graphe de dépendance Java, représentation intermédiaire très utile pour la manipulation de programmes est présenté plus en détail. La sous-section 2.3.7 présente la technique de découpage d'application (*program slicing*) permettant d'obtenir une projection d'un code source original, projection guidée par un ensemble de variables de référence dont on souhaite suivre l'exécution au cours de l'exécution d'un système. Un *slice* est donc un programme dérivé du pro-

gramme original et dont on a supprimé les instructions du code dont les variables de référence ne dépendent pas.

Enfin, les travaux relatifs à l'analyse statique de code source dans le domaine particulier de l'interaction homme-machine sont exposés.

2.3.1 Compilation : Analyses lexicale, syntaxique et sémantique

Un compilateur est un système logiciel qui traduit un programme écrit dans un langage de haut niveau en un programme équivalent en code objet ou dans un langage machine de manière à pouvoir réaliser une exécution de ce programme sur un calculateur. Cette définition peut être étendue pour inclure tout système traducteur d'un langage dans un autre, d'un langage machine vers un autre, d'un langage de haut niveau dans un langage intermédiaire...

Un compilateur met en œuvre un ensemble de phases de traitements qui analysent séquentiellement diverses formes d'un même programme pour en synthétiser de nouvelles. Ces phases de traitement partent initialement d'une séquence de caractères qui constitue le *programme source* à compiler pour produire finalement une représentation exécutable sur un calculateur. Comme le décrit la figure 2.7, on peut distinguer au moins quatre étapes dans la mise en œuvre d'une compilation : l'*analyse lexicale*, l'*analyse syntaxique*, l'*analyse sémantique* et la *génération de code*.

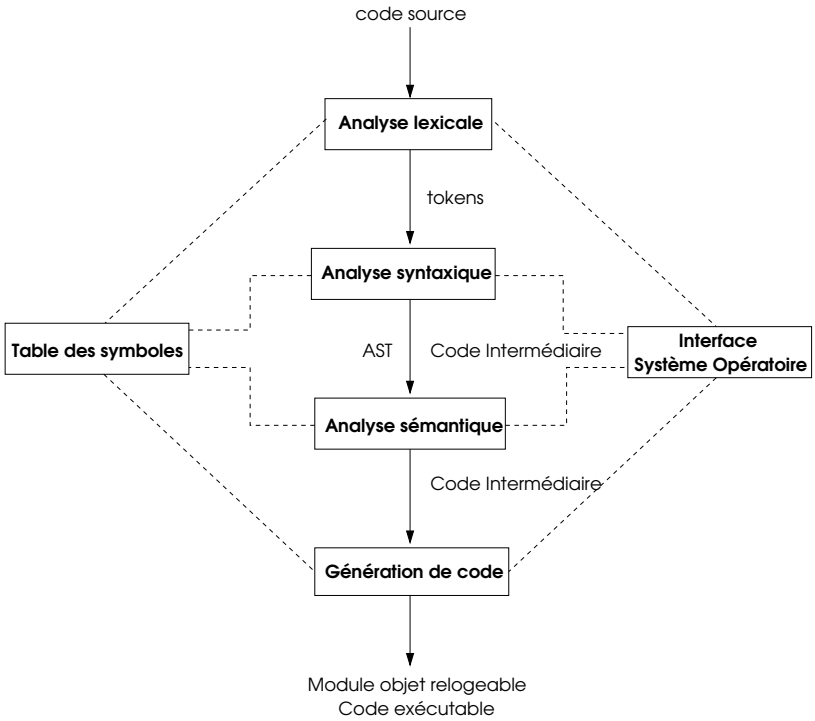


Fig. 2.7 – Structure d'un compilateur

Analyse lexicale. L'étape d'analyse lexicale analyse la chaîne de caractères présentée en entrée du compilateur et identifie dans cette chaîne les lexèmes (ou *tokens* en anglais) qui comportent principalement les mots appartenant au vocabulaire du langage dans lequel

le programme est écrit ainsi que les symboles de ponctuation : cette étape peut produire des messages d'erreurs si la chaîne de caractères ne peut être analysée comme une chaîne de mots du langage.

Analyse syntaxique. L'étape d'analyse syntaxique s'opère sur la séquence de *tokens* du langage et vérifie que cette séquence est structurée en conformité avec les règles de syntaxe du langage. Les règles de syntaxe sont formalisées pour chaque langage par une *grammaire*. Cette étape produit une représentation intermédiaire du programme, généralement un arbre de syntaxe abstraite (*Abstract Syntax Tree* ou *AST*) ainsi qu'une table de symboles qui mémorise les identifieurs utilisés dans le programme ainsi que leurs attributs ; cette étape peut produire des messages d'erreur si la chaîne de *tokens* n'est pas structurée conformément à la grammaire du langage.

Analyse sémantique. L'analyse sémantique prend en entrée la représentation intermédiaire et la table des symboles générée par l'étape précédente et détermine si le programme a un sens dans le langage source, et en particulier s'il satisfait les propriétés sémantiques statiques requises par la définition de ce langage source : par exemple si les identifieurs du programme sont correctement déclarés et utilisés et si les expressions sont correctement typées. Cette étape peut aussi réaliser des *analyses de flots de données* afin de détecter d'éventuels risques d'erreurs ou pour effectuer des optimisations indépendantes du langage cible. Cette étape peut enfin produire des messages d'erreurs lorsque le programme n'est pas sémantiquement correct ou qu'il présente des constructions qui échappent à la définition du langage.

Génération de code. Enfin, l'étape de génération de code transforme la représentation intermédiaire en un code intermédiaire ou machine équivalent sous forme d'un module objet relogeable ou d'un code objet directement exécutable.

Les analyses lexicale et syntaxique sont souvent regroupées lors de la compilation et font largement appel aux automates. Elles s'appuient sur le cadre plus restreint des automates finis et des expressions régulières (ou rationnelles) qui sont largement utilisées dans de nombreux outils.

L'analyse sémantique, et tout particulièrement pour la vérification de la correction des types, fait appel aux graphes de dépendances.

2.3.2 Analyse statique et représentations intermédiaires

De manière générale, l'objectif d'une analyse statique est de découvrir des informations sur le comportement d'un programme à l'exécution. Il s'agit de déterminer *statiquement* (à l'analyse) certaines propriétés *dynamiques* (à l'exécution) des programmes. Traditionnellement, les principales applications de l'analyse statique concernent l'optimisation des programmes (éliminations de sous-expressions communes, propagations de copies, élimination de code mort, déplacement de code hors boucle, détection de variables d'induction...), la vérification (divisions par zéro, dépassements des bornes d'un tableau, détections de mauvaises utilisations de pointeurs, détections de code malicieux...), ou bien la parallélisation de codes séquentiels.

De nombreux travaux exploitent également les principes de l'analyse statique pour la rétro-ingénierie ou ingénierie inverse d'applications (*Reverse Engineering*). Plusieurs acceptions peuvent être attribuées au terme de *Reverse Engineering*. Initialement, la rétro-ingénierie consiste à produire les données d'une étape N du cycle de développement à partir des données de l'étape $N+1$. Cependant, de nombreux travaux utilisent également le terme de rétro-ingénierie pour désigner le processus consistant à extraire un ou des modèles abstraits de l'application. Les modèles obtenus peuvent être utilisés de différentes manières comme par exemple pour une meilleure compréhension du système. La rétro-ingénierie est une technique particulièrement exploitée dans le cadre de la maintenance de système ou de l'adaptation de systèmes d'un contexte d'utilisation (langage d'implémentation, plateforme d'exécution) vers un autre. Dans ce dernier cas, certains travaux exploitent ces modèles obtenus pour la génération de nouveaux systèmes.

Une pratique plus récente consiste à analyser un programme afin de générer un modèle formel de l'application. Ce modèle formel, constituant une abstraction du code source originel, peut alors être exploité pour vérifier un certain nombre de propriétés du système : atteignabilité, vivacité, sûreté, etc.

Quel que soit l'objectif de l'analyse statique, il est nécessaire de disposer d'une représentation intermédiaire du code source étudié afin d'effectuer des abstractions ou des transformations permettant l'analyse du système ou la génération d'un modèle exploitable en terme de vérification formelle. Dans ce qui suit, les modèles intermédiaires pouvant être utilisés à cet égard, ainsi que quelques techniques permettant la manipulation de ces représentations, telles que l'abstraction ou le *slicing*, sont présentés.

2.3.3 Représentations intermédiaires : AST, Graphes de Flots de Données et Graphes de Flots de Contrôle

Il est possible de classer les représentations intermédiaires en deux grandes catégories :
- les représentations intermédiaires fondées sur des graphes ;
- les représentations intermédiaires linéaires : elles s'apparentent à une forme de pseudo-code pour des machines abstraites.

En pratique on trouve en compilation de nombreuses représentations intermédiaires du code source qui hybrident les deux.

La sous-section suivante présente la représentation par AST (*Abstract Syntax Tree*) d'un programme. Cette représentation peut être analysée puis transformée en une représentation permettant l'analyse du comportement dynamique d'un programme : les graphes de flots de contrôle.

AST

Les étapes d'analyse d'un code source utilisent la plupart du temps une représentation intermédiaire. L'Arbre de Syntaxe Abstraite ou AST (*Abstract Syntax Tree*) est une des représentations exploitées pour l'analyse de programme. Cet arbre est produit à partir de l'arbre de dérivation fourni par une grammaire du langage. La figure 2.8 présente un programme Java simple et sa représentation sous forme d'arbre de syntaxe abstraite.

```
1 public class MyClass{
2     public int main(int b, int a) {
3         if (b>c)
4       {a=b-c;
5       return a;}
6   else{a=b+c;
7       return a;}
8     }
9 }
```

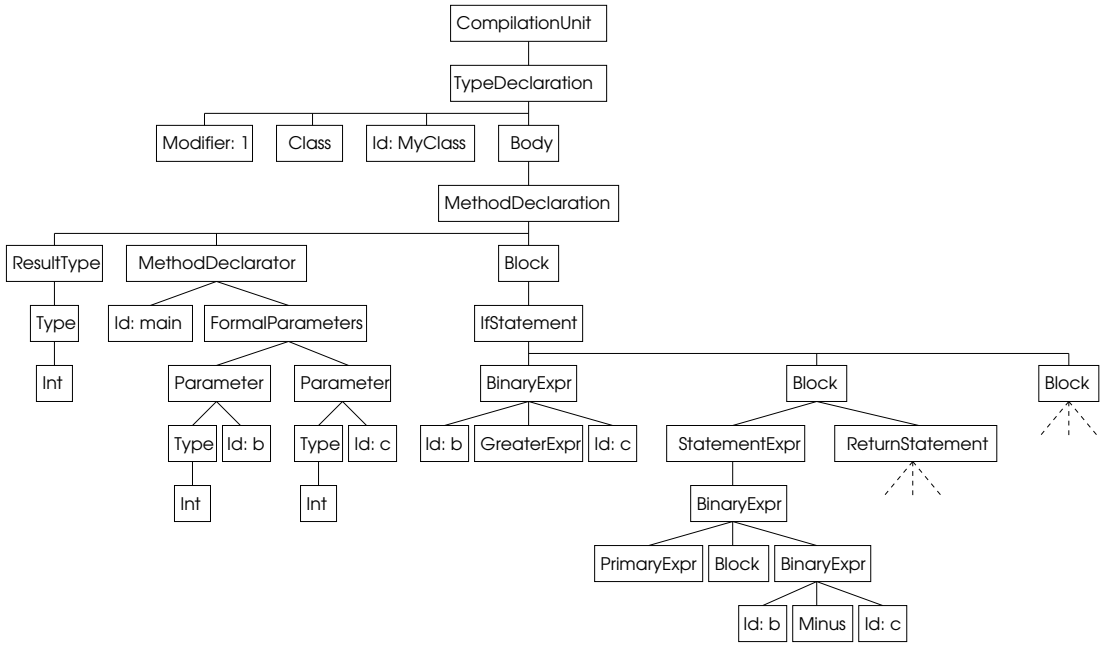


FIG. 2.8 – Un programme Java simple et sa représentation sous forme d'AST.

L'arbre de syntaxe abstraite d'un programme est un outil idéal pour les traductions (ou compilations) d'un langage source vers un autre.

Graphes de flots de contrôle : CFG

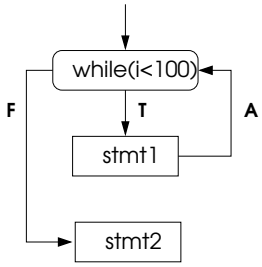
Un graphe de contrôle (ou *Control Flow Graph* : CFG) est une représentation abstraite du contrôle de l'exécution d'un code source. Un CFG peut être dérivé en temps linéaire en utilisant une traduction dirigée par la syntaxe [Aho *et al.*, 2006].

Les graphes de flots de contrôle représentent la manière dont le programme transfère le contrôle entre différents blocs d'instructions. Les nœuds du graphe représentent les blocs de base : ce sont les séquences d'instructions dépourvues de toute instruction de contrôle et de branchement.

La figure 2.9 présente deux codes et leur représentation sous forme de CFG où les blocs de base sont représentés par des rectangles.

```
1 while (i<100)
2    begin
3      stmt1
4    end
5 stmt2
```

```
1 if(x=y)
2    then stmt1
3    else stmt2
4 stmt3
5 fi
```



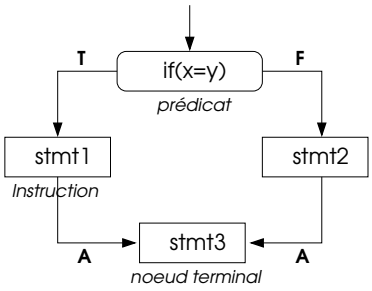


FIG. 2.9 – Deux programmes simples et leur représentation sous forme de CFG.

On impose que les blocs de base ne disposent que d'un seul point d'entrée : la première instruction du bloc. Par ailleurs, ces blocs de base peuvent ne pas être directement représentés dans le graphe de flot de contrôle. Ils peuvent disposer d'une autre représentation particulière comme par exemple un arbre de syntaxe abstraite. D'autres représentations non présentées dans ce mémoire sont possibles : graphe dirigé sans cycle ou représentation linéaire.

Formellement, un CFG est défini de la manière suivante (définition adaptée de [Ball & Horwitz, 1993]) :

Définition : Control Flow Graph. *Le graphe de contrôle $\mathcal{G} = (N, E, n_{Start})$ d'un programme est un graphe dirigé et étiqueté où :*
- *N est un ensemble de nœuds représentant les commandes du programme ;*
- *l'ensemble N est partitionné en deux sous-ensembles N^I et N^P où :*
  - *N^I est l'ensemble des nœuds représentant une instruction qui possèdent au plus un successeur ;*
  - *N^P est l'ensemble des nœuds représentant un prédicat qui possèdent deux successeurs ;*
  - *et $N^E \subseteq N^I$ contient les nœuds de N^I qui ne possèdent pas de successeur (i.e les nœuds terminaux de $\mathcal{G}$).*
- *E est un ensemble d'arcs étiquetés qui représente le flot de contrôle entre les nœuds du graphe tel que chacun des nœuds $n_p \in N^P$ possède deux arcs sortants étiquetés T et F respectivement, et chacun des nœuds $n_i \in (N^I - N^E)$ possède un arc sortant étiqueté A (pour "Always taken") ;*
- *enfin, le nœud d'entrée n_{Start} ne possède pas d'arcs entrants et tous les nœuds de N sont atteignables à partir de n_{Start}.*

Dépendances de données et dépendances de contrôle dans un CFG

Les dépendances de données et de contrôle sont définies à partir du CFG d'un programme. Les ensembles DEF(i) et REF(i) dénotent les ensembles de variables référencées et définies dans le bloc d'instructions définissant le nœud i du CFG.

Définition : Dépendance de flots de données. *Un nœud j est dépendant d'un nœud i du point de vue des données s'il existe une variable x telle que :*
- $x \in \text{DEF}(i)$
- $x \in \text{REF}(i)$
- *et il existe un chemin du CFG de i à j sans intervention de la définition de x.*

Autrement dit, la définition de x au nœud i est une définition qui atteint le nœud j.

La dépendance vis-à-vis du contrôle est habituellement définie en terme de *post-domination*.

Définition : Post-domination et dépendance de contrôle. *Un nœud i du CFG est post-dominé par un nœud j si tous les chemins de i au nœud de sortie du CFG passent par j. Un nœud j est dépendant du nœud i du point de vue du contrôle si :*
- *il existe un chemin P de i à j tel que tout nœud $u \in P$ et $u \neq i, j$ est post-dominé par j, et*
- *i n'est pas post-dominé par j.*

De manière informelle, une dépendance de contrôle $A \xrightarrow{c} B$ existe si l'exécution de l'instruction B est reliée à l'exécution d'un nœud de type prédicat A. Une dépendance de données $A \xrightarrow{d} B$ existe si l'exécution de l'instruction B référence une variable qui est définie ou modifiée par l'instruction A.

2.3.4 Représentations intermédiaires (2) : Graphes de Dépendance Système

[Ottenstein & Ottenstein, 1984] ont été les premiers à suggérer que les graphes de dépendances pouvaient être utilisés pour des opérations d'ingénierie logicielle basées sur l'analyse et la représentation de programmes. Ces travaux proposent un graphe de dépendance de programme (*Program Dependence Graph* : PDG) capable de représenter un bloc de code exécuté séquentiellement. Par la suite, les travaux de [Horwitz *et al.*, 1988] ont introduit le graphe de dépendance système (*System Dependence Graph* : SDG) pour l'analyse de programmes multiprocéduraux. Dans cette représentation, chaque procédure est représentée individuellement par un graphe de dépendance. Ces graphes sont ensuite reliés à un graphe de dépendance central qui représente le programme principal.

De nombreux travaux se sont penchés sur la modification du SDG afin de prendre en compte la représentation de programmes orientés objet. Ce type de représentation doit être capable de capturer des propriétés telles que le polymorphisme, l'association dynamique et l'héritage. [Larsen & Harrold, 1996] ont notamment proposé un graphe capable de prendre en compte ces propriétés pour des programmes C++. Cette représentation a été modifiée par [Kovács *et al.*, 1996] et [Zhao *et al.*, 1996] pour prendre en compte certaines fonctionnalités spécifiques du langage Java telles que les interfaces, les paquetages et l'héritage simple. [Liang & Harrold, 1998] ont également augmenté les graphes proposés par Larsen et Harrold afin de distinguer les attributs d'objets paramétrés éliminant de ce fait des dépendances superflues au niveau de certains sites d'appel de méthodes et par conséquent améliorant ainsi la précision d'opérations basées sur les graphes.

[Walkinshaw & Roper, 2003] ont également proposé un graphe de dépendance basé sur le langage Java, baptisé *Java System Dependance Graph* (JSysDG) qui intègre les bénéfices des différentes approches mentionnées ci-dessus. Ces travaux présentent en outre la construction d'un JSysDG d'un point de vue pratique.

Enfin, les travaux de [Hatcliff *et al.*, 1999], [Hatcliff *et al.*, 2000a], [Hatcliff *et al.*, 2000b] introduisent plusieurs types de dépendance permettant la représentation d'applications Java composées de plusieurs processus (*threads*) communicants. Cette représentation est notamment utilisée au sein de l'outil Bandera[6] pour le *slicing* d'applications.

La présentation des opérations effectuées sur les codes Java-Swing pour l'extraction d'un modèle formel B événementiel (Chapitre 5) repose sur la représentation intermédiaire proposée par [Walkinshaw & Roper, 2003] : JsysDG. La section suivante est consacrée à une présentation plus détaillée de ce graphe de dépendance.

2.3.5 Java System Dependence Graph : JsysDG

Un graphe JSysDG est un multi-graphes qui décrit les dépendances de données et de contrôle entre les instructions d'un programme Java. Les instructions appartiennent à des catégories distinctes suivant qu'elles participent à la structure du programme (en-têtes de déclarations de paquetages, de classes, d'interfaces et de méthodes) ou bien au comportement du programme (instructions appartenant au corps d'une méthode). Chacune de ces catégories est représentée de manière différente dans un graphe. La combinaison de ces graphes fournit une représentation du programme complet.

La construction d'un JSysDG se fonde sur une construction préalable des CFG des différentes méthodes déclarées dans le code source de l'application.

Instructions

L'instruction représente la couche la plus basse d'un JSysDG. Il s'agit d'une construction atomique représentant une seule expression dans le code source du programme. Une instruction représentant un appel à une autre méthode (site d'appel ou "*call site*") nécessite une représentation spécifique décrite par la suite.

Graphe de dépendance de méthodes ou "Method Dependence Graph" (MDG)

Le MDG représente les méthodes d'un programme. Le nœud d'entrée de la méthode (*method entry vertex*) est connecté à tous les autres nœuds appartenant au corps de la méthode par des arcs de dépendance (*control dependence edges*) vis-à-vis du contrôle. Les passages de paramètres, dans le cas d'une instruction du type "appel de méthode", sont modélisés par l'introduction de nœuds de type "paramètre réel" *actual* et "paramètre formel" *formal*. Du point de vue de la méthode appelante, les nœuds *actual-in* et *actual-out* sont construits pour copier la valeur des paramètres d'entrée et de sortie dans des variables intermédiaires. La méthode appelée contient les nœuds *formal-in* et *formal-out*

[6]Bandera project : http://bandera.projects.cis.ksu.edu/

qui copient respectivement les paramètres d'entrée et la valeur de sortie. Enfin des arcs *parameter-in* connectent les nœuds *actual-in* et *formal-in* tandis que des arcs *parameter-out* connectent les nœuds *formal-out* et *actual-out* [Horwitz *et al.*, 1988]. En outre tous les nœuds *formal* sont connectés au nœud d'entrée de la méthode et les nœuds *actual* sont connectés au nœud correspondant à l'appel (*call site*) par des arcs de dépendance vis-à-vis du contrôle. Le flot de données au sein d'une méthode est représenté par des arcs de dépendance sur données. Enfin, un nœud de dépendance d'appel (*call dependence*) relie le nœud d'appel au nœud d'entrée de la méthode appelée. La figure 2.10 représente un exemple simple d'appel de méthode.

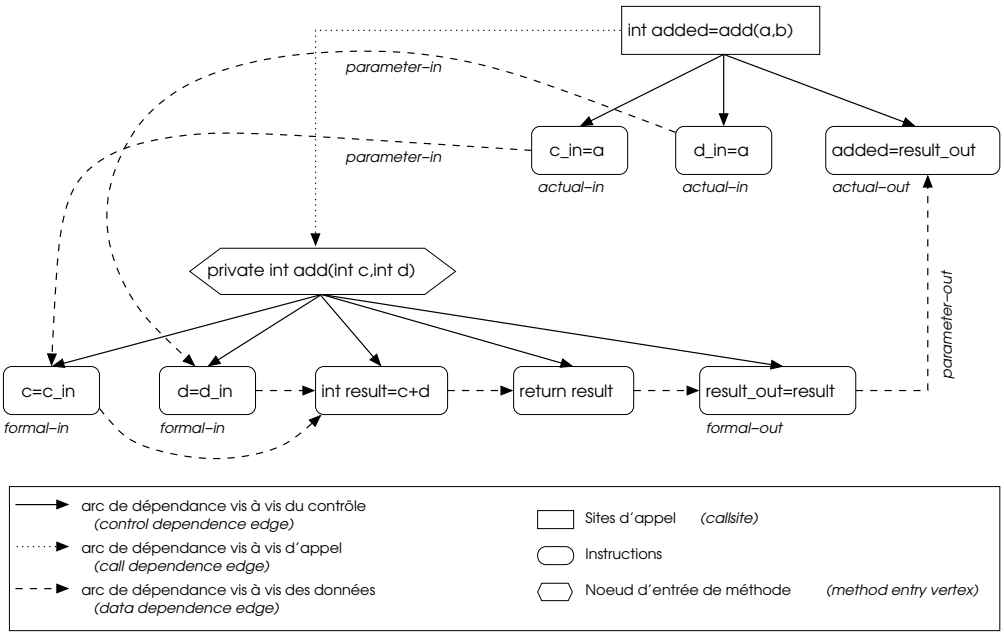


FIG. 2.10 – Exemple : MDG d'un appel de méthode simple.

Graphe de dépendance de classes ou "Class Dependence Graph" (ClDG)

Le ClDG représente les classes d'un programme [Larsen & Harrold, 1996]. Pour chaque classe il existe un nœud d'entrée (*class entry vertex*) qui est connecté aux nœuds d'entrée des méthodes de la classe par des arcs d'appartenance de classe (*class membership edge*). Ces arcs peuvent être étiquetés pour indiquer la visibilité de la méthode (ex. : public, protected) [Kovács *et al.*, 1996].

Dans le cas où une classe hérite d'une autre, les deux classes sont liées par un arc de dépendance sur classes (*class dependence edge*). Le nœud d'entrée de la classe est connecté à ses attributs de classe par des arcs (*data member edge*). La figure 2.11 présente un exemple simple de ClDG.

Représentation des objets et polymorphisme. Un JSysDG représente les différentes instances d'une classe individuellement ; ceci permet de prendre en considération les objets individuellement lors d'opérations sur le graphe de dépendance telles que le

slicing. Un nœud n qui référence un objet est étendu en un arbre suivant le contexte d'utilisation de n. Les différents contextes d'extension sont les suivants :

- n est un nœud de type paramètre représentant un objet de type statique[7] ;
- n est un nœud de type paramètre représentant un objet de type dynamique[8] ;
- n est un site d'appel et la méthode est appelée sur un objet de type statique (ex : `objet1.callMethod_M()`) ;
- n est un site d'appel et la méthode est appelée sur un objet de type dynamique.

Voir [Walkinshaw & Roper, 2003] pour plus d'informations.

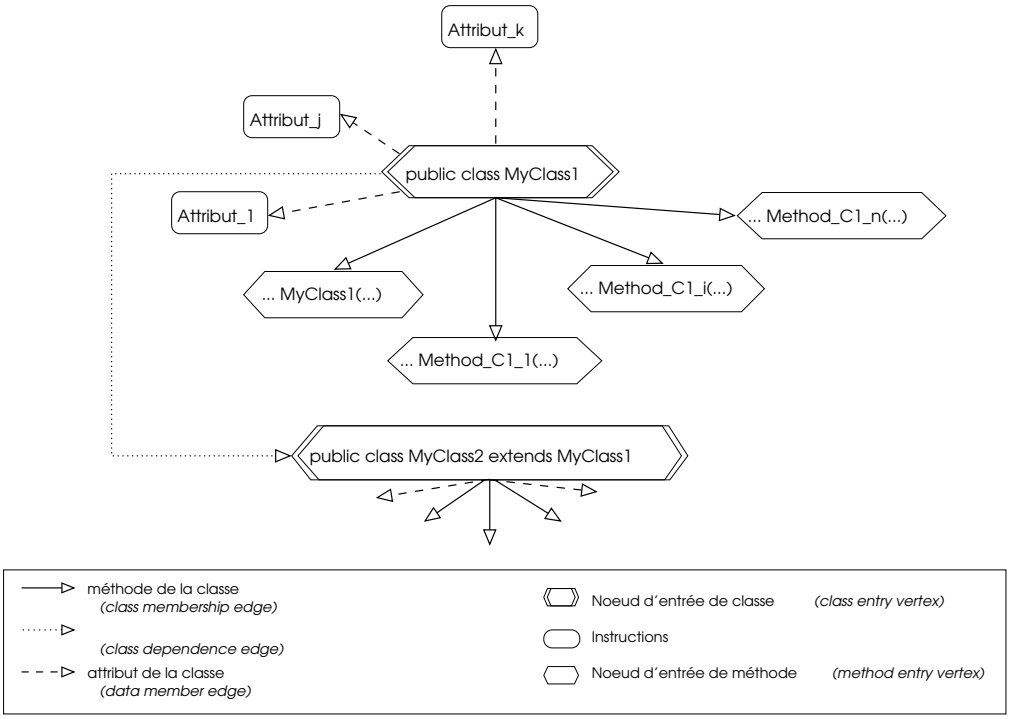


FIG. 2.11 – Exemple de ClDG simple.

Graphe de dépendance d'interface ou "Interface Dependence Graph (InDG)"

Le JsysDG traite une interface Java comme une forme particulière de classe abstraite. Les méthodes abstraites sont représentées dans un JSysDG par un nœud d'entrée. Pour représenter complètement la signature d'une méthode, si la méthode retourne une valeur (i.e qui n'est pas `void`), le JsysDG connecte le nœud d'entrée de la méthode à un nœud de type sortie de paramètre (*parameter-out*).

Le graphe de dépendance sur interfaces (InDG) consiste en un nœud d'entrée d'interface (*interface entry vertex*) qui est connecté à un ensemble de nœuds d'entrée de méthode représentant ses méthodes abstraites par un arc (*absract member edge*). Les nœuds d'entrée de ces méthodes sont connectés aux nœuds de type paramètre représentant leurs paramètres d'entrée. Ces derniers nœuds n'ont pas besoin d'assigner la valeur d'entrée à

[7] Le type de l'objet peut être déterminé statiquement sans exécuter le programme.
[8] Le type de l'objet peut uniquement être déterminé dynamiquement : cas du polymorphisme.

une variable temporaire puisque la méthode est abstraite. Chaque nœud d'entrée de la méthode abstraite est connecté au nœud d'entrée de la méthode l'implémentant par un arc de type implémentation de méthode abstraite (*implement abstract method*). Si une classe implémente une interface, celle-ci est connectée à l'interface par un arc d'implémentation (*implements*). Si une classe $C1$ étend une classe $C2$ et que $C2$ implémente une interface, $C1$ doit également implémenter l'interface. Il n'est alors pas nécessaire de connecter $C1$ à l'interface, cette connexion étant implicite du fait de la relation d'héritage.

La figure 2.12 propose un exemple de graphe de dépendance d'interface simple.

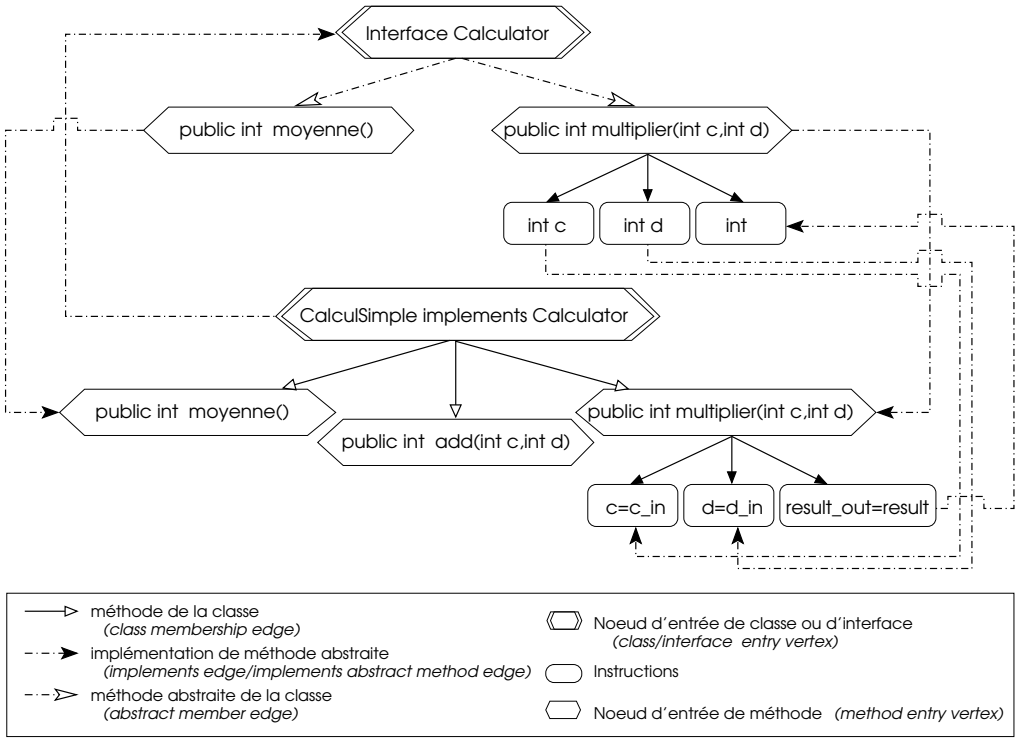


FIG. 2.12 – Exemple de InDG simple.

Graphe de dépendance de paquetage ou "Package Dependence Graph" (PaDG)

Un "paquetage" définit une collection de classes qui sont similaires ou relatives à un même propos. Ceci est représenté par un graphe de dépendances sur paquetages (PaDG). Il s'agit de la dernière couche d'un JsysDG. Les paquetages sont importants pour des opérations de *slicing* puisqu'ils sont nécessaires au calcul adéquat de la visibilité des données. Un nœud d'entrée de paquetage (*package entry*) représente le paquetage. Ce nœud d'entrée est connecté aux nœuds d'entrée des classes appartenant au paquetage par un arc d'appartenance au paquetage (*package member*).

2.3.6 Opération de découpage : "program slicing"

Le concept du "découpage d'applications" (*program slicing*) a été introduit par [Weiser, 1979]. Dans la définition de Weiser, un *slice* correspond à l'ensemble des ins-

tructions d'un programme qui affectent (potentiellement) les variables d'un ensemble V en un point du programme p. La paire (V, p) est définie comme étant le critère de *slice* (*slicing criterion)*, et dans certains cas [Horwitz *et al.*, 1988], l'ensemble des variables appartenant à V est supposé être présent au point de contrôle p.

Depuis, plusieurs notions de *slicing* légèrement différentes ont été proposées, ainsi qu'un grand nombre de méthodes pour les calculer. Une importante distinction est faite entre les opérations de *slicing statique* et *dynamique*. Les opérations de *slicing* statique sont réalisées sans avoir recours à l'exécution du programme analysé et sans hypothèse sur les entrées du programme. Une opération de *slicing* statique est valide pour toutes les exécutions du programme, mais peut être imprécise pour certaines expressions particulières. A contrario, l'opération de *slicing dynamique* exploite l'exécution du programme analysé et se fonde sur des cas de tests spécifiques.

Procédures, contrôle de flots arbitraire, types de données composés, pointeurs et communication interprocédurale : chacune de ces spécificités d'un programme nécessite une solution particulière pour effectuer une opération de *slicing*.

La sous-section suivante donne un aperçu de différentes méthodes utilisées pour le *slicing* de programmes. Loin d'être exhaustif, il s'agit de présenter uniquement les techniques liées au *slicing* statique d'applications. Des informations supplémentaires peuvent être trouvées dans [Tip, 1995].

Le slicing statique

La figure 2.13 propose un exemple de programme emprunté à [Tip, 1995]. Ce programme lit une variable `n` et calcule la somme et le produit des `n` premiers nombres positifs. La partie droite de la figure 2.13 montre le résultat d'une opération de *slicing* sur ce programme suivant le critère de *slicing* `(10,product)`. Tous les calculs relatifs à la variable `sum` ont été supprimés.

Exemple de programme P

```
1  read(n)
2  i:=1;
3  sum:=0;
4  product:=1;
5  while (i ≤ n) do {
6     sum:=sum+i;
7     product:=product*i;
8     i=:i+1;}
9  write(sum);
10 write(product);
```

Slice de P : critère ≡ `(10,product)`

```
1  read(n)
2  i:=1;
3
4  product:=1;
5  while i ≤ n do{
6
7     product:=product*i;
8     i=:i+1;}
9
10 write(product);
```

FIG. 2.13 – Un example de *slicing* sur un programme simple

Dans l'approche de Weiser, les *slices* sont calculés en répétant et en ôtant les ensembles d'instructions non directement pertinentes du point de vue des dépendances de flots de contrôle et de flots de données. Ici, seules les informations disponibles statiquement sont utilisées pour le calcul du *slice*.

Une méthode alternative pour le calcul de *slices* statiques a été proposée par [Ottenstein & Ottenstein, 1984] qui redéfinit le problème du *slicing* statique en termes

d'atteignabilité dans un graphe de dépendances de programme (*program dependence graph* : PDG) [Ferrante *et al.*, 1987]. Un PDG est un graphe dirigé dont les nœuds correspondent aux instructions ou aux prédicats de contrôle du programme et dont les arcs correspondent aux dépendances vis-à-vis des données et vis-à-vis du contrôle. Le critère de *slicing* est identifié par un nœud du PDG et le *slice* correspond à tous les nœuds du PDG pouvant être atteints à partir du nœud considéré par le critère.

De nombreuses autres approches de *slicing* basées sur les graphes utilisent des versions étendues ou modifiées du PDG comme représentation interne.

Une autre approche proposée par [Bergeretti & Carré, 1985] définit l'opération de *slicing* en termes de relations de flots d'informations dérivés directement à partir du programme suivant une méthode dirigée par la syntaxe.

Les *slices* mentionnés ci-dessus sont calculés en collectant les déclarations (instructions) et les prédicats de contrôle par une analyse arrière (*backward*) qui traverse le programme en partant du point de contrôle défini par le critère de *slicing*. Pour cette raison, ces *slices* sont appelés *slices arrière* (*backward (static) slice*). [Bergeretti & Carré, 1985] furent les premiers à définir la notion de *slice statique avant* (*forward slice*), bien que [Reps & Bricker, 1989] aient été les premiers à utiliser cette terminologie. Informellement, un *slice avant* est constitué de toutes les instructions et de tous les prédicats de contrôle qui dépendent du critère de *slicing*. Une instruction est "dépendante" du critère de *slicing* si les valeurs calculées par cette instruction dépendent des valeurs calculées au point de contrôle du critère, ou bien si les valeurs calculées au point défini par le critère déterminent le fait que l'instruction considérée est exécutée ou non. Les *slices* arrière et avant sont calculés d'une manière similaire ; il est cependant nécessaire d'effectuer un traçage des dépendances dans le cas d'une analyse avant.

2.3.7 Méthodes de slicing statique

Cette section présente les algorithmes de base pour le *slicing* statique de programmes structurés sans variables non scalaires, sans procédures et sans communications interprocessus.

Une revue et un comparatif d'algorithmes de *slicing* plus élaborés peuvent être trouvés dans [Tip, 1995].

Equations de flots de données

La définition originale du *slicing* de programme introduite par [Weiser, 1979] est basée sur un calcul itératif d'équations de flots de données. Weiser définit un *slice* comme un programme exécutable obtenu à partir du programme original en supprimant une ou plusieurs instructions. Un critère de *slice* consiste en une paire (n, V) où n est un nœud dans le CFG du programme, et V un sous-ensemble des variables du programme. Afin d'être un *slice* respectant le critère (n, V), le sous-ensemble S des instructions d'un programme P doit satisfaire les propriétés suivantes : chaque fois que P s'arrête pour une entrée donnée, S s'arrête également pour cette entrée, et S calcule les mêmes valeurs pour les variables appartenant à V chaque fois que l'instruction correspondante au nœud n est exécutée. Il existe au moins un *slice* pour un critère de *slicing* quelconque : le programme lui-même. Un *slice* est dit minimal du point de vue des instructions (*statement-minimal*) s'il n'existe

aucun *slice* pour le même critère contenant moins d'instructions. D'après Weiser, les *slices* minimaux ne sont pas nécessairement uniques, et le problème de la détermination des *slices* minimaux est indécidable.

Les approximations de *slices* minimaux sont calculées suivant un processus itératif qui calcule les ensembles consécutifs de *variables pertinentes* pour chaque nœud du CFG associé au programme. Tout d'abord, les *variables directement pertinentes* sont déterminées en prenant en compte les dépendances de données. Dans ce qui suit $i \xrightarrow{CFG} j$ indique l'existence d'un arc dans le CFG entre les nœuds i et j. Pour un critère de *slicing* $C \equiv (n, \mathcal{V})$, l'ensemble des variables directement pertinentes au nœud i du CFG, noté $R^0_C(i)$, est défini comme suit :

1. $R^0_C(i) = V$ pour $i = n$;
2. pour chaque $i \xrightarrow{CFG} j$, $R^0_C(i)$ contient toutes les variables v telles que :
   (a) $v \in R^0_C(j)$ et $v \notin \textsc{Def}(i)$, ou
   (b) $v \in \textsc{Ref}(i)$ et $\textsc{Def}(i) \cap R^0_C(j) \neq \varnothing$

Partant de cela, un ensemble d'*instructions directement pertinentes* S^0_C est dérivé. S^0_C est défini comme l'ensemble des nœuds i qui définissent une variable v qui est un successeur pertinent de i dans le CFG :

$$S^0_C \equiv \{i | \textsc{Def}(i) \cap R^0_C(j) \neq \varnothing, i \xrightarrow{CFG} j)\}$$

Les variables référencées dans un prédicat de contrôle (instructions `if` ou `while`) sont *indirectement pertinentes* si (au moins) une des instructions dans les branches contrôlées par le prédicat est pertinente. La *classe d'influence* $\textsc{Infl}(b)$ d'un prédicat b est définie comme l'ensemble des instructions qui sont dépendantes du contrôle de b. Les branches d'instructions B^k_C qui sont pertinentes du fait de leur influence sur un nœud i de S^k_C sont définies comme suit :

$$B^k_C \equiv \{b | i \in S^k_c, i \in \textsc{Infl}(b)\}$$

Les ensembles de *variables indirectement pertinentes* R^{k+1}_C sont déterminés en considérant les variables dans les prédicats des branches d'instructions B^k_C pertinentes.

$$R^{k+1}_C(i) \equiv R^k_C(i) \cup \bigcup_{b \in B^k_C} R^0(i)_{(b, \textsc{Ref}(b))}$$

Les ensembles d'*instructions indirectement pertinentes* S^{k+1}_c sont constitués des nœuds appartenant à B^k_C et des nœuds i qui définissent une variable qui est pertinente pour le successeur j dans le CFG :

$$S^{k+1}_C \equiv B^k_C \{i | \textsc{Def}(i) \cap R^{k+1}_C(j) \neq \varnothing, i \xrightarrow{CFG} j)\}$$

NODE #	DEF	REF	INFL	R_C^0	R_C^1
1	{n}	∅	∅	∅	∅
2	{i}	∅	∅	∅	{n}
3	{sum}	∅	∅	{i}	{i,n}
4	{product}	∅	∅	{i}	{i,n}
5	∅	{i,n}	{6,7,8}	{product,i}	{product,i,n}
6	{sum}	{sum,i}	∅	{product,i}	{product,i,n}
7	{product}	{product,i}	∅	{product,i}	{product,i,n}
8	{i}	{i}	∅	{product,i}	{product,i,n}
9	∅	{sum}	∅	{product}	{product}
10	∅	{product}	∅	{product}	{product}

TAB. 2.1 – Résultats obtenus en applicant l'algorithme de Weiser sur l'exemple de la figure 2.13

Les ensembles R_C^{k+1} et S_C^{k+1} sont des sous-ensembles non décroissants de l'ensemble des variables du programme et des instructions du programme respectivement ; le point du calcul des ensembles S_C^{k+1} constitue le *slice* souhaité du programme.

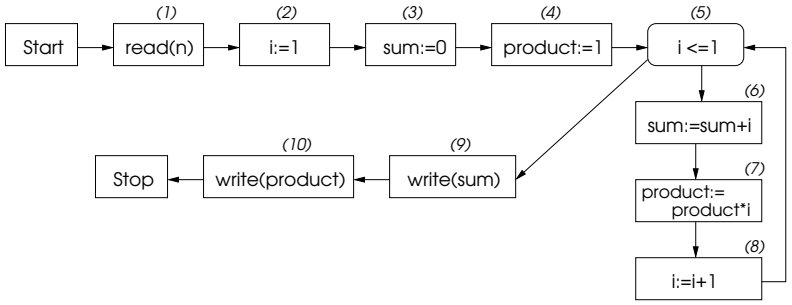


FIG. 2.14 – CFG de l'exemple de la figure 2.13.

À titre d'exemple, considérons le programme de la figure 2.13 et le critère de *slicing* `(10,product)`. Le CFG représentant le programme initial est donné en figure 2.14. Le tableau 2.1 résume la valeur des ensembles DEF, REF et INFL et les ensembles de variables pertinentes calculés par l'algorithme de Weiser. À partir des informations fournies par le tableau et de la définition d'un *slice* on obtient $S_C^0 = \{2, 4, 7, 8\}$, $B_C^0 = \{5\}$, et $S_C^1 = \{1, 2, 4, 5, 7, 8\}$.

Pour cet exemple le point fixe des ensembles de variables non directement pertinentes est obtenu pour l'ensemble S_C^1. Le *slice* correspondant au critère $C \equiv$`(10,product)` calculé suivant l'algorithme de Weiser est identique au *slice* présenté en figure 2.13 sans l'instruction `write(product)` qui n'est pas contenue dans le *slice*.

[Lyle, 1984] présente une version modifiée de l'algorithme de Weiser pour le calcul de *slices*. À l'exception de quelques changements mineurs du point de vue de la terminologie, cet algorithme est essentiellement le même que dans [Weiser, 1981].

[Hausler, 1989] reformule l'algorithme de Weiser suivant une sémantique dénotationnelle. Dans une sémantique dénotationnelle, le comportement d'une instruction, ou d'une séquence d'instructions, est caractérisé en définissant la manière dont il transforme l'état. Dans le *slicing* dénotationnel une fonction δ caractérise une construction du langage en définissant comment celle-ci affecte l'ensemble des variables pertinentes. Une autre fonction α utilise δ pour exprimer comment les *slices* peuvent être construits.

Relations de flots d'informations

λ_ϵ	$=$	$\varnothing$
μ_ϵ	$=$	$\varnothing$
ρ_ϵ	$=$	(Id)
$\lambda_{v:=e}$	$=$	$\textsc{Vars}(e) \times \{e\}$
$\mu_{v:=e}$	$=$	$\{(e, v)\}$
$\rho_{v:=e}$	$=$	$(\textsc{Vars}(e) \times \{v\} \cup (\textsc{Id}) - (v, v'))$
$\lambda_{S_1;S_2}$	$=$	$\lambda_{S_1} \cup (\rho_{S_1} \cdot \lambda_{S_2})$
$\mu_{S_1;S_2}$	$=$	$(\mu_{S_1} \cdot \rho_{S_2}) \cup \mu_{S_2}$
$\rho_{S_1;S_2}$	$=$	$\rho_{S_1} \cdot \rho_{S_2}$
$\lambda_{\textbf{if } e \textbf{ then } S}$	$=$	$(\textsc{Vars}(e) \times \{e\}) \cup \lambda_S$
$\mu_{\textbf{if } e \textbf{ then } S}$	$=$	$(\{e\} \times \textsc{Defs}(S)) \cup \mu_S$
$\rho_{\textbf{if } e \textbf{ then } S}$	$=$	$(\textsc{Vars}(e) \times \textsc{Defs}(S)) \cup \rho_S \cup \textsc{Id}$
$\lambda_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2}$	$=$	$(\textsc{Vars}(e) \times \{e\}) \cup \lambda_{S_1} \cup \lambda_{S_2}$
$\mu_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2}$	$=$	$(\{e\} \times (\textsc{Defs}(S_1) \cup \textsc{Defs}(S_2))) \cup \mu_{S_1} \cup \mu_{S_2}$
$\rho_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2}$	$=$	$(\textsc{Vars}(e) \times (\textsc{Defs}(S_1) \cup \textsc{Defs}(S_2))) \cup \rho_{S_1} \cup \rho_{S_2} \cup \textsc{Id}$
$\lambda_{\textbf{while } e \textbf{ do } S}$	$=$	$\rho_S^\star \cdot ((\textsc{Vars}(e) \times \{e\}) \cup \lambda_S)$
$\mu_{\textbf{while } e \textbf{ do } S}$	$=$	$(\{e\} \times \textsc{Defs}(S)) \cup \mu_S \cdot \rho_S^\star \cdot ((\{e\} \times \textsc{Vars}(e)) \cup \textsc{Id})$
$\rho_{\textbf{while } e \textbf{ do } S}$	$=$	$\rho_S^\star \cdot ((\textsc{Vars}(e) \times \textsc{Defs}(S)) \cup \textsc{Id})$

FIG. 2.15 – Définition récursive des relations de flots d'informations de Bergetti et Carré

Les travaux de [Bergeretti & Carré, 1985] définissent des relations de flots d'informations pour des programmes. Ces relations peuvent être utilisées pour calculer des *slices*. Pour une instruction (ou une séquence d'instructions) S, une variable v, et une expression (i.e, un prédicat de contrôle ou la partie droite d'une affectation) e qui apparaît dans S, les relations λ_S, μ_S et ρ_S sont définies. Ces relations possèdent les propriétés suivantes :

- $(v, e) \in \lambda_S$ si et seulement si la valeur de v en entrée de S affecte potentiellement la valeur calculée pour e ;
- $(e, v) \in \mu_S$ si et seulement si la valeur calculée pour e affecte potentiellement la valeur de v à la sortie de S ;
- $(v, v') \in \rho_S$ si et seulement si la valeur de v en entrée de S affecte éventuellement la valeur de v' à la sortie de S.

L'ensemble E_S^v dénote l'ensemble de toutes les expressions e telles que $(e, v) \in \mu_S$. Cet ensemble peut être utilisé pour construire des *instructions partielles*. L'instruction partielle d'une instruction S associée à la variable v est obtenue en remplaçant toutes les instructions de S qui ne contiennent pas d'expressions dans E_S^v par des instructions vides.

Les relations de flots d'informations sont calculées de manière ascendante et ce calcul est dirigé par la syntaxe. Pour une instruction vide, les relations λ_S et μ_S sont vides, et ρ_S correspond à la relation d'identité. Pour une affectation du type $v := e$, λ_S contient (v', e) pour toutes les variables v' qui apparaissent dans e, μ_S contient (e, v) et ρ_S contient

(v, v') pour toutes les variables qui apparaissent dans e ainsi que (v'', v'') pour toutes les variables $v'' \neq v$.

La figure 2.15 montre comment les relations de flots d'informations pour une séquence d'instructions, des instructions conditionnelles, des instructions de boucles sont construites à partir des relations de flots d'informations de ses constituants. Dans cette figure, ϵ dénote l'instruction vide, "$\cdot$" la jointure relationnelle, ID la relation identité, VARS(e) l'ensemble des variables apparaissant dans e, et DEFS(S) l'ensemble des variables définies dans l'instruction S. La définition pour la construction ***while*** est obtenue en la transformant de manière effective en une séquence infinie d'instructions ***if*** imbriquées à une seule branche. La relation $\rho_S^\star$ utilisée dans cette définition correspond à la fermeture transitive et réflexive de ρ.

Un *slice* vis-à-vis de la valeur d'une variable v en un point arbitraire peut être calculé en insérant une affectation fictive $v' := v$ à la place appropriée, où v' est une variable qui n'est pas apparue précédemment dans S. Le *slice* vis-à-vis de la valeur finale de v' dans le programme modifié est équivalent à un *slice* suivant la variable v au point sélectionné dans le programme d'origine. Des *slices* statiques avant peuvent être dérivés d'une relation λ_S d'une manière similaire à la méthode calculant des *slices* statiques arrière à partir de la relation μ_S.

Approches basées sur les graphes de dépendances

Dans toutes les approches basées sur les graphes de dépendance, le critère de slice est identifié par un nœud v dans le graphe de dépendance du programme. Selon la terminologie de Weiser, ceci correspond à un critère (n, V) où n est un nœud du CFG correspondant à v, et V l'ensemble de toutes les variables définies ou utilisées au point v. Dans ce cas, c'est l'ensemble des nœuds atteignables à partir de v qui définit le slice désiré. Les parties relatives au code source du programme peuvent être trouvées en maintenant une correspondance entre les nœuds du graphe de dépendance et le code source pendant la construction du graphe.

En conséquence, le critère de *slicing* des méthodes basées sur les graphes de dépendance de programme est moins général que celui des méthodes basées sur les équations de flots de données ou les relations de flots d'informations. Cependant, une méthode de *slicing* basée sur les graphes de dépendances de programme peut calculer un slice suivant le critère de *slicing* (n, V) pour un ensemble V arbitraire en réalisant les trois étapes suivantes :

1. tout d'abord, le nœud n du CFG correspondant au nœud du graphe de dépendance du programme est déterminé ;
2. puis l'ensemble des nœuds N du CFG correspondant à toutes les définitions atteignables pour les variables de V à partir n est déterminé ;
3. enfin, l'ensemble S des nœuds du graphe de dépendence du programme correspondant à l'ensemble N est déterminé.

Le slice désiré correspond alors l'ensemble des nœuds à partir desquels un nœud dans S peut être atteint suivant le graphe de dépendance du programme.

Une étude détaillée concernant les différentes méthodes de *slicing* (statique et dynamique, méthodes pour le *slicing* interprocédural, comparaison des méthodes, complexité des algorithmes) peut être trouvée dans [Tip, 1995].

Certaines notations utilisées dans les algorithmes de Weiser seront exploitées au chapitre 5.

2.3.8 Analyse statique et IHM : applications

Rétro-ingénierie

La démarche de Reverse Engineering consiste à repenser ce qui a été conçu dans une démarche d'ingénierie. En informatique, l'ingénierie inverse consiste à analyser un produit fini (un système d'information, des processus, un logiciel ou des interfaces) pour déterminer la manière dont celui-ci a été conçu et d'identifier des composants et leurs dépendances [Müller *et al.*, 2000].

La rétro-ingénierie d'applications peut être dirigée par différents objectifs : compréhension du code source, génération de documentation, maintenance, réutilisation ou adaptation. On s'intéresse plus particulièrement ici aux travaux de "rétro-ingénierie" visant la génération de modèles abstraits, modèles éventuellement exploités pour l'adaptation d'IHM ou dans le cadre d'outils de conception itérative dirigée par les modèles.

Le processus de rétro-ingénierie d'adaptation d'IHM [Chikofsky *et al.*, 1990], c'est-à-dire le processus permettant le passage d'un contexte d'utilisation vers un autre, peut être décomposé en deux étapes successives :

- l'ingénierie régressive (ou abstraction) qui consite à analyser une application de manière à en extraire une représentation abstraite (indépendante de la plateforme et du langage) ;
- l'ingénierie progressive (ou concrétisation) durant laquelle une nouvelle interface est générée à partir des spécifications actualisées et d'une représentation abstraite.

[Calvary *et al.*, 2005] utilise les termes de *réification* et *réification inverse* pour dénoter respectivement la concrétisation et l'abstraction. La majorité des travaux dans ce domaine traite des IHM pour le web. Les premiers travaux sur la rétro-ingénierie avaient pour objectif de remplacer l'interface textuelle d'applications anciennes par une interface graphique. Aujourd'hui les travaux s'orientent plutôt vers la rétro-ingénierie de pages web pour les rendre disponibles sur plusieurs plateformes. Javahery et al. [Javahery *et al.*, 2003] proposent une démarche qui se base sur les *patterns*[9] et principalement sur les *patterns d'utilisabilité* dans l'objectif de générer des interfaces utilisateurs multiples (MUI). [Paganelli & Paternò, 2003] proposent une rétro-ingénierie des pages web qui permet d'obtenir une description du modèle de tâches exprimée en CTT. VAQUITA [Vanderdonckt *et al.*, 2001, Bouillon *et al.*, 2005] est un autre environnement qui permet d'effectuer la rétro-ingénierie des pages web de manière à obtenir un modèle de présentation indépendant du contexte d'utilisation. Il offre la possibilité d'obtenir plusieurs modèles de présentation à partir d'une même page par la sélection d'heuristiques de rétro-ingénierie différentes. En revanche, cette approche est utilisée uniquement pour générer une représentation abstraite d'une page web. Elle ne s'intéresse pas à la phase de regénération de nouvelles interfaces.

Bien que certains travaux parlent de rétro-ingénierie pour désigner l'opération consistant à extraire des modèles formels d'un code source afin de vérifier un certain nombre de

[9] *Pattern* : Solution récurrente décrivant et résolvant un problème général dans un contexte particulier. Il existe différents types de patterns : d'analyse, de conception, d'implémentation...

propriétés de l'application par model-checking, cette activité particulière est désignée par "abstraction ou formalisation" dans ce qui suit.

Extraction de modèles pour la vérification : abstraction ou formalisation

Les travaux de [Silva *et al.*, 2006] traitent de l'abstraction d'applications Java/Swing. Ces travaux sont menés dans le cadre d'un projet de R&D (IVY - A model-based usability analysis environment[10]) dont le but est de développer un outil basé sur les modèles pour l'analyse de systèmes interactifs (*JSR : Java Swing Reverse*). L'outil, en cours de développement, s'intéresse à l'analyse de programmes Java générés par la technologie NETBEANS produisant un code structuré. L'outil JSR travaille sur l'arbre de syntaxe abstraite de l'application pour générer, suite à des opérations de *slicing* et d'abstractions, trois modèles distincts : un modèle structurel de l'interface et deux modèles comportementaux dont un modèle d'interacteurs et un système d'états transitions. Le modèle structurel est représenté par un graphe de flots d'événements (*Event Flow Graph*) représentant une vision abstraite de l'ensemble des widgets de l'application ainsi que leurs relations. Le premier modèle comportemental est représenté dans le langage à base d'interacteurs MAL (*Modal Action Logic*) tel que défini dans [Campos & Harrison, 2001]. Enfin, le second modèle extrait est une machine à états finis. Plusieurs vérifications sont effectuées par traduction du modèle d'interacteurs MAL dans le langage d'entrée du *model-checker* NuSMV.

Les travaux de [d'Ausbourg, 2001] présentés en section 2.2 proposent également une rétro-génération d'un modèle Lustre à base d'interacteurs à partir d'une spécification de la partie statique de l'interface et d'un code source C représentant la partie dynamique liée à l'interaction sur l'interface. Dans le cadre du projet VERBATIM[11], Ausbourg et al. se sont également intéressés à la construction automatique de modèles Promela d'applications Java-Swing multimodales. Ces travaux prennent en compte des applications construites suivant l'architecture ICARE. Les modèles extraits sont ensuite vérifiés via le *model-checker* SPIN.

2.4 Synthèse, constats et proposition

Cette section propose une synthèse concernant le domaine de l'interaction homme-machine. Quelques constats, découlant de cette synthèse, sont par la suite mis en évidence et conduisent à une proposition pour la validation de l'utilisabilité d'un système interactif.

2.4.1 Synthèse et constats à propos de l'interaction Homme-Machine

Le domaine de l'Interaction Homme-Machine est un domaine de recherche pluridisciplinaire (ergonomie, psychologie et informatique) qui étudie la façon dont les humains interagissent avec les ordinateurs ou entre eux à l'aide d'ordinateurs, ainsi que la façon

[10] http://www.di.uminho.pt/ivy

[11] RNRT VERBATIM-VERification Biformelle et Automatisation du Test d'Interfaces Multimodales, http://iihm.imag.fr/nigay/VERBATIM/

de concevoir des systèmes informatiques qui soient ergonomiques, c'est-à-dire efficaces, faciles à utiliser ou plus généralement adaptés à leur contexte d'utilisation.

L'évolution des *modalités* et des dispositifs d'interaction utilisés et la taille sans cesse croissante des systèmes à interfacer complexifient la conception et la validation des IHM. Il est nécessaire pour les concepteurs de ces systèmes de disposer d'une panoplie de modèles, d'outils et de méthodes permettant de prendre en charge cette complexité.

Développement des IHM et utilisabilité

Ergonomes et psychologues sont à l'origine de plusieurs modèles, dits *modèles de l'utilisateur*, permettant de mieux appréhender la logique de l'utilisateur (modèle du processeur humain, GOMS, Keystroke) voire de mieux prévoir ou interpréter ses difficultés d'interaction (Théorie de l'Action). Si ces modèles fournissent au concepteur un ensemble de données psychologiques et ergonomiques sur le comportement de l'utilisateur, ils ne fournissent aucune information sur la réalisation technique d'un système interactif.

Quoi qu'il en soit, l'interface développée doit satisfaire des critères d'*utilisabilité*. L'utilisabilité d'un système dénote l'efficacité, l'efficience et la satisfaction avec lesquelles les utilisateurs peuvent utiliser le système pour atteindre leurs buts.

Afin d'exprimer les besoins de l'utilisateur, de nombreuses notations de description ont été proposées. Ces notations, appelées *modèles de tâches* tels que HTA, MAD, UAN, XUAN, ou encore CTT, sont pour la plupart centrées sur l'utilisateur. Elles permettent d'une part la collecte d'informations sur la façon dont les utilisateurs accomplissent une activité, et d'autre part elles définissent la vue que l'utilisateur aura du système interactif (description de l'interface homme-machine). Ces notations constituent, en général, la base de la réalisation d'une IHM.

Face aux déboires rencontrés lors de la conception des premiers systèmes interactifs, peu utilisables, les informaticiens ont vite compris la nécessité de prendre en compte l'utilisateur du système au plus tôt dans le *cycle de développement* des IHM. Les cycles de développement, comme le cycle en V, permettent la prise en compte des besoins utilisateurs en intégrant un modèle de tâches. Un modèle de tâches est un outil qui aide le concepteur lors des phases de conception, d'évaluation de l'utilisabilité et de rédaction de la documentation (contenu et structure). En outre, les modèles de tâches s'adressent également à l'utilisateur pour l'aider dans son travail en lui proposant une aide sous forme de documentation[12]. Cependant, les ergonomes et les psychologues, souvent à l'origine de ces notations, privilégient des approches graphiques basées sur une décomposition hiérarchique des tâches en sous-tâches et représentées par des arbres. Ces notations sont informelles et peuvent amener quelques réserves du point de vue des difficultés rencontrées lors du passage de l'analyse de tâche au développement. Elles obligent le concepteur à transcrire les besoins de l'utilisateur dans sa conception suivant un processus informel au risque de définir des représentations conceptuelles erronées ou ambiguës.

Outre les cycles de développement, plusieurs techniques, méthodes et modèles ont été empruntés au domaine du *génie logiciel* afin d'améliorer la conception logicielle des systèmes interactifs. En adaptant ces techniques au domaine de l'interaction Homme-machine, de nombreux *modèles d'architectures dédiés IHM* ont vu le jour (SEEHEIM,

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

ARCH, MVC, PAC...). Un modèle d'architecture, qu'il soit global ou multi-agent, définit un cadre générique d'architecture logicielle. La plupart de ces modèles reposent sur un principe commun : la séparation du noyau fonctionnel de l'application, qui contient les fonctionnalités brutes du système, et de la partie purement interactive. La gestion du dialogue entre l'utilisateur et le système est alors prise en charge par un contrôleur de dialogue pouvant être réparti ou non au sein de l'application suivant le type d'architecture utilisé.

Plusieurs *modèles de dialogue* permettant de spécifier la dynamique du dialogue homme-machine ont également été proposés. Ces modèles de dialogue utilisent des formalismes comme les systèmes de transitions étiquetées (STE) ou les réseaux de Petri. Dans ces modèles, le système interactif est vu soit comme un ensemble d'états liés par des transitions (et le passage d'un état à un autre établit alors la dynamique du système), soit comme un ensemble d'événements identifiant des états.

La modélisation du dialogue doit intégrer l'utilisateur afin de garantir l'utilisabilité du système. Toute trace ou tout scénario issu du modèle de tâches qui définit les besoins de l'utilisateur doit donc se retrouver dans la dynamique du dialogue. Les modèles de tâches privilégient les notations graphiques au détriment d'une clarté sémantique ce qui rend cette étape de validation et vérification délicate.

En ce sens, seuls les tests exhaustifs permettent de vérifier que les traces issues d'un modèle de tâches sont incluses dans la modélisation du dialogue de l'application. Il s'agit de la seule méthode couramment utilisée pour assurer la fiabilité du système du point de vu de l'utilisabilité de l'interface.

Outils de conception et approches formelles de conception

Si l'évaluation de l'utilisabilité d'un système interactif n'est pas "indispensable" pour la plupart des systèmes, force est de constater qu'il s'agit d'une priorité dans le cas des systèmes qualifiés de "critiques". Les IHM assistent de nos jours des activités à haut risque, notamment dans la plupart des secteurs d'activité liés aux transports (interface de pilotage pour les dernières générations d'aéronefs, contrôle aérien), dans les activités nucléaires (système de contrôle d'une centrale) ou dans le secteur médical (interface de contrôle de laser). L'IHM correspondant au seul point d'entrée du système, la fiabilité du système global dépend en grande partie de la fiabilité de l'IHM.

Afin d'aider le concepteur dans sa tâche, d'améliorer la conception ou d'améliorer les processus de validation des IHM, plusieurs approches ont été proposées. On distingue principalement deux types d'approches : les *approches expérimentales* et les *approches formelles*. Cette frontière est malléable et certaines approches peuvent être qualifiées de "mixtes" lorsqu'elles associent l'approche formelle et l'approche expérimentale.

Approches expérimentales. Les approches que l'on qualifie d'expérimentales s'appuient sur un *outil de conception*. Les outils de bas niveau d'abstraction, tels que les *générateurs d'interface* ou les *squelettes d'applications*, sont actuellement les seuls utilisés dans un contexte industriel. Ces outils permettent de créer la partie graphique de l'application et laissent une très grande liberté au concepteur dans le choix de l'aspect visuel de l'interface. Cependant, ces outils sont souvent trop proches d'un langage de programmation et par conséquent aucun raisonnement ne peut être établi sur une modélisation

préalable du système. La vérification et la validation sont alors effectuées au moyen de longues phases de tests ou de relecture de code.

Les outils d'un plus haut niveau d'abstraction tels que les *systèmes de gestion d'interfaces utilisateurs* (SGUI) et les *systèmes basés sur modèles* (SGBD) suivent une approche ascendante qui consiste à créer l'application au moyen de modèles définis dans un ou plusieurs langages. 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 par les modèles. Contrairement aux générateurs d'interfaces classiques, cette approche autorise le raisonnement sur les modèles. Cependant, le raisonnement ne s'effectue 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, mises à part quelques exceptions [Baron, 2003], la vérification et la validation à tous les niveaux de la modélisation. En outre, ces outils de haut niveau restent à l'heure actuelle à l'état de recherche.

Les approches formelles. Une autre approche qualifiée de formelle consiste à utiliser conjointement un langage à sémantique mathématique permettant la définition de modèles et un outil de preuve (*model-checking* ou *theorem-proving*) permettant la vérification de propriétés. Suivant le formalisme utilisé la conception peut alors être ascendante ou descendante.

Parmi les techniques formelles présentées dans ce chapitre les techniques basées sur la vérification de modèles (model-checking) permettent de rendre exécutables des spécifications formelles et prennent en charge de manière automatique la preuve de propriétés. En contre-partie, elles peuvent se heurter au problème d'explosion du nombre de leurs états.

À l'opposé, les techniques basées sur la preuve autorisent le découpage de la preuve au moyen du raffinement et de la préservation des propriétés d'un raffinement à un autre. Cependant, ces techniques sont pénalisées par l'aspect semi-automatique de la preuve.

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 du système de preuve employé.

De ce fait, malgré les résultats très prometteurs des méthodes formelles, leurs utilisations dans les phases de conception ou de spécification d'un cycle de développement industriel sont actuellement peu envisagées. Ceci nécessiterait un changement trop important des méthodes de travail, des langages et des modèles utilisés par les concepteurs actuels.

Les approches "mixtes". Enfin d'autres approches proposent des solutions "mixtes". Parmi ces approches on peut citer les travaux de [Baron, 2003], proposant une approche expérimentale en intégrant l'utilisation de modèles B (outil SUIDT), et ceux de [d'Ausbourg & Cazin, 1999] qui utilisent des modèles formels Lustre pour la génération de cas de tests. Enfin, une approche plus récente dans le domaine de l'IHM consiste à réaliser une analyse statique de code source afin d'en extraire des modèles exploitables pour la vérification [d'Ausbourg & Durrieu, 2006], [Silva *et al.*, 2006].

Analyse Statique : rétro-génération de modèles formels pour la validation

L'analyse statique de code source consiste à découvrir des informations sur le comportement d'un programme à l'exécution. L'analyse statique, quel que soit l'objectif (optimisation de code, rétro-ingénierie, adaptativité), exploite des représentations intermédiaires du programme source afin d'analyser de manière statique son comportement dynamique. Quelques travaux proposent d'utiliser les techniques d'analyse statique afin d'extraire des modèles formels exploitables pour la vérification de certaines propriétés des systèmes interactifs [d'Ausbourg, 2001], [Silva *et al.*, 2006], [d'Ausbourg & Durrieu, 2006]. Toutes les approches proposées exploitent une technique basée sur le model-checking (Lustre, SMV et Promela) et proposent de vérifier formellement certaines propriétés sans avoir à modifier les méthodes ou outils utilisés par les concepteurs.

Une telle approche peut être exploitable dans un développement de type itératif, permettant la preuve de propriétés au fur et à mesure de l'implémentation.

En résumé :

1. étant données la complexité et la criticité des applications interactives développées, il est *nécessaire d'assurer l'utilisabilité et la fiabilité des systèmes conçus*. La prise en compte d'un modèle de tâche dans le processus de validation semble être un bon moyen pour assurer une partie des exigences liées à l'utilisabilité du système ;

2. à l'heure actuelle, la vérification et la validation des systèmes interactifs sont réalisées par le biais de *tests coûteux, fastidieux, et la plupart du temps non exhaustifs* ;

3. les outils de conception dirigés par les modèles prennent en compte ces modèles de tâches, mais la majorité de ces outils ne propose pas de démarche de validation et de vérification liées à l'*utilisabilité du système*. En outre, ces outils restent à l'état de recherche, contrairement aux outils tels que les générateurs d'interfaces ;

4. l'utilisation des *techniques formelles* tend à se populariser. Ces techniques permettent de vérifier certaines classes de propriétés du système et, dans une certaine mesure, permettent de diminuer les phases de tests nécessaires à la vérification d'une IHM. Cependant, ces techniques semblent difficilement exploitables dans une phase de développement et d'implémentation puisqu'elles nécessiteraient de la part du développeur une remise en cause complète des méthodes et outils de travail. En outre, la réalisation de modèles formels nécessite un haut degré d'expertise ;

5. plusieurs travaux se dirigent vers les possibilités offertes par les *techniques d'analyse statique* afin d'extraire des modèles formels exploitables en terme de validation du système. Ces travaux possèdent l'avantage d'utiliser les méthodes formelles uniquement lors de la phase de validation du système. Par conséquent, les développeurs d'applications n'ont pas à revoir leurs méthodes de conception et de développement. Cependant, tous ces travaux exploitent la technique du *model-checking* pour la preuve de propriétés et, à notre connaissance, aucun de ces travaux ne propose de méthode pour la validation des systèmes interactifs vis-à-vis de modèles de tâches.

2.4.2 Proposition

La figure 2.16 présente la vue globale de la proposition formulée dans ce mémoire dont l'objectif est de définir une méthodologie de validation formelle des systèmes interactifs critiques du point de vue de leur utilisabilité (voir également [Cortier *et al.*, 2007b], [Cortier *et al.*, 2007a]). Plus particulièrement, l'objectif est de vérifier que le système est en adéquation avec un modèle de tâches CTT que l'on considère comme une spécification du système.

Cette approche formelle intervient uniquement dans la phase de validation du système. Ce choix est guidé par le fait que l'utilisation des méthodes formelles est difficile à mettre en place dans le cadre d'un processus de développement : redéfinition du cycle de développement, changement d'habitude des développeurs (nouveaux langages, nouvelles techniques). Dans ce cadre, le code source de l'application est le seul matériel dont on dispose. Ce code source peut être le résultat d'une génération de code (partielle ou non) à partir d'outils spécifiques (générateur de présentations, SGUI).

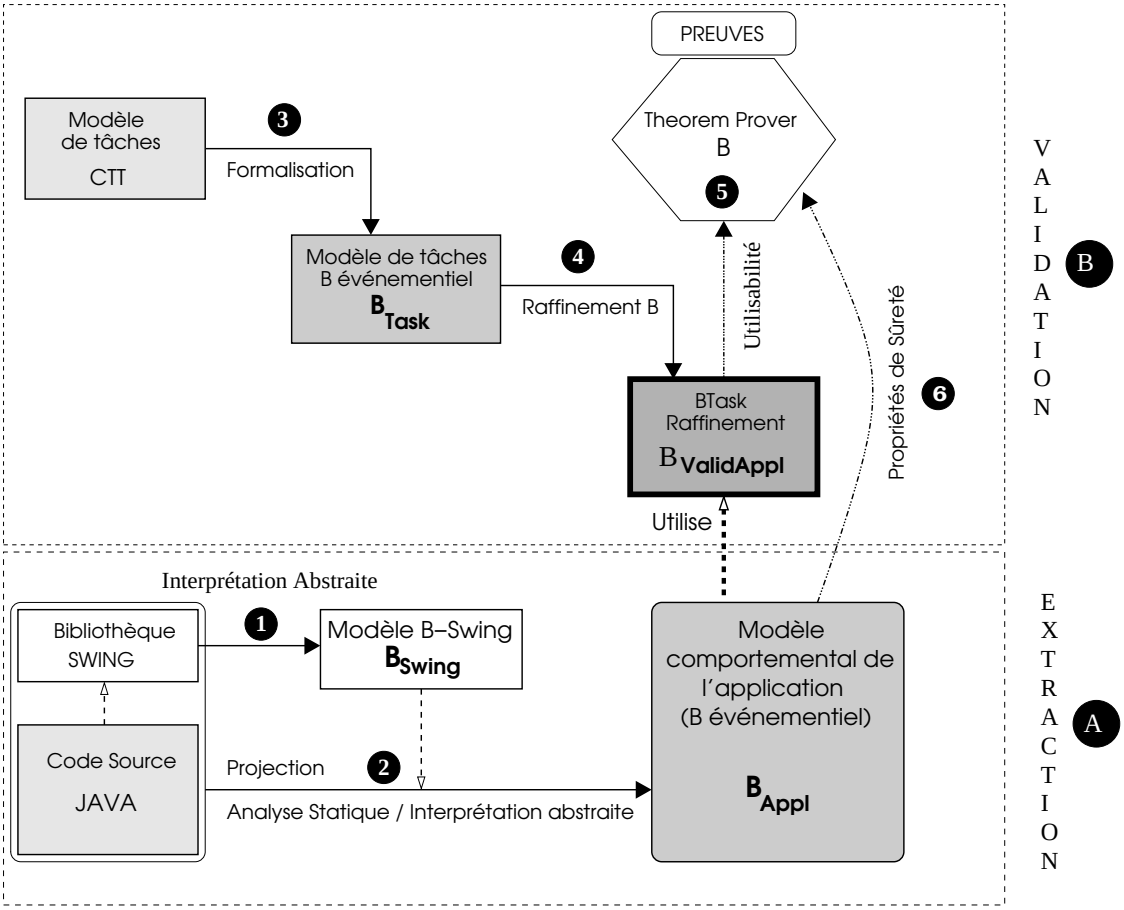


FIG. 2.16 – Principes de l'approche.

L'approche proposée peut être divisée en deux grandes étapes : l'extraction d'un modèle formel à partir du code source d'une application (Fig. 2.16, tag.**(A)**) et l'utilisation de ce modèle pour la validation de l'application vis-à-vis d'un modèle de tâches (Fig. 2.16, tag.**(B)**).

Extraction de modèles formels par analyse statique de code source Java/Swing

Les travaux présentés dans ce mémoire s'intéressent donc à l'extraction de modèles formels à partir d'applications développées en Java/Swing par analyse statique de leurs codes sources (Fig. 2.16,❶,❷). Le choix du langage de programmation Java/Swing est guidé par la popularité grandissante de celui-ci. La méthodologie proposée reste cependant applicable pour tout autre type de langage reposant sur des principes similaires et notamment sur l'utilisation de boîte à outils.

La méthode "B événementiel" a été choisie pour exprimer le modèle formel extrait de l'application. Ce modèle est une représentation formelle de la dynamique du dialogue de l'application. Le choix de la méthode B événementiel est guidé par le fait que le dialogue d'un système interactif est fortement combinatoire. En effet le nombre de chemins d'interactions d'un système interactif et par conséquent le nombre d'états permettant de représenter la dynamique du dialogue d'un système interactif est très important. Par conséquent, le choix d'une technique formelle orientée démonstration de preuves apparaît comme judicieuse puisque les techniques de vérifications exhaustives de modèles peuvent se heurter au problème de l'explosion combinatoire. En outre, la méthode B est aujourd'hui bien outillée (Atelier B, Balbulette et B4free, plateforme Rodin) et en voie de démocratisation comme le prouve son utilisation dans le secteur ferroviaire[13]. À notre connaissance, aucune autre approche de validation de systèmes interactifs reposant sur l'analyse statique de programmes et sur une technique formelle fondée sur la preuve n'existe à ce jour.

Toutefois, afin d'établir une comparaison entre les approches de preuve (*model-checking* et *theorem-proving*), les principes de modélisation d'une application Java-Swing dans un autre langage formel appelé NuSMV sont également présentés.

Le modèle formel extrait peut être directement utilisé pour la validation de propriétés de sûreté (Fig. 2.16,❺,❻) .

Validation de l'utilisabilité du système

La validation du système vis-à-vis d'une spécification CTT est délicate du fait de la nature semi-formelle de la notation CTT. Le principe de la validation est de confronter le comportement de l'application au modèle de tâche reflétant les exigences attendues de l'application. Il s'agit de démontrer que les structures d'interaction encodées dans le programme s'inscrivent bien dans les scénarii d'usage représentés en compréhension dans un modèle de tâches CTT. Pour cela, le modèle de tâche est formalisé en un modèle B événementiel B_{Task} (Fig.2.16,❸). Ce modèle est raffiné en ajoutant de nouveaux événements et de nouvelles variables (Fig.2.16,❹). Les événements introduits sont les événements issus du modèle extrait B_{ApplM} qui reflètent le comportement de l'application (déclenchement de méthodes Java) lorsqu'elle répond aux actions de l'usager sur l'interface. La preuve de correction du raffinement ainsi que la preuve de non-blocage du système, établies à l'aide d'un prouveur de théorème B (Fig.2.16,❺), assurent alors la correction de l'application vis-à-vis du modèle de tâche.

[13]Notamment, l'entreprise Clearsy utilise la méthode B pour le développement des systèmes de signalisation ferroviaires.

Il est possible également de valider le système vis-à-vis d'une spécification CTT à partir d'un modèle NuSMV : une comparaison des deux approches sera alors réalisée.

Plan de travail

Le chapitre 3 de ce mémoire présente quelques prérequis nécessaires à la compréhension de la démarche : présentation du langage Java, de la méthode B et B événementiel et du langage NuSMV. Une étude de cas est également présentée. Cette étude de cas est utilisée tout au long du mémoire afin d'illustrer par des exemples simples la mise en application des différentes étapes de l'approche.

Le chapitre 4 présente en détail les principes de la modélisation B événementielle et NuSMV d'une application Java/Swing et montre comment ces modèles peuvent être utilisés afin de vérifier certaines propriétés de sûreté du système.

Le chapitre 5 présente en détail l'analyse statique permettant l'extraction de modèles formels afin de mettre en évidence la faisabilité d'une automatisation de la démarche d'extraction. Quelques algorithmes ainsi qu'une réflexion sur la structure du code analysé accompagnent notre propos.

Le chapitre 6 présente la deuxième étape de l'approche : la validation de l'application vis-à-vis de modèles de tâches CTT. La notation CTT est présentée en annexe A.

Enfin, le dernier chapitre de ce mémoire propose une conclusion générale et des perspectives de travaux relatives à l'approche méthodologique proposée.

Deuxième partie

Validation formelle d'applications Java-Swing par analyse statique de code source.

"Les mathématiques consistent à prouver une chose évidente par des moyens complexes."
Georges Polya

"L'ordinateur est un appareil sophistiqué auquel on fait porter une housse la nuit en cas de poussière et le chapeau durant la journée en cas d'erreur."
Jean-Marie Gourio

Chapitre 3

Prérequis

"Tout ça n'est rien si je n'ai pas le Swing."
Duke Ellington

L'objectif de ce chapitre est de présenter quelques prérequis sur les langages utilisés dans cette partie et de présenter d'un point de vue général la méthodologie employée pour la validation formelle de systèmes interactifs développés en Java-Swing. En section 3.1, on présente les principaux concepts de la programmation Java-Swing ainsi qu'une étude de cas simple (convertisseur Euro/Dollar) qui servira de support à l'exposé. Les sections 3.2 et 3.3 présenterons respectivement la méthode B et plus particulièrement sa vision événementielle ainsi que la méthode NuSMV.

3.1 Le langage Java-Swing : Présentation et Étude de cas

Les bibliothèques Swing et AWT définissent un ensemble de composants graphiques et d'écouteurs d'événements permettant de programmer les aspects structurel, visuel et comportemental d'une interface.

3.1.1 Étude de cas : Un convertisseur Euros/Dollars

Afin de présenter le langage Java/Swing, cette section commence par la présentation d'une étude de cas simple : un convertisseur Euros/Dollars. Cette étude de cas est utilisée tout au long de l'exposé.

Description de l'interface. La figure 3.1 présente l'interface du convertisseur étudié. L'interface est constituée de quatre composants graphiques élémentaires principaux : deux champs de saisie et deux boutons. Le champ de saisie situé à gauche (❶) permet la saisie d'une valeur à convertir. Le deuxième champ de texte, situé à droite (❹) permet l'affichage du résultat de la conversion. Enfin, les deux boutons permettent respectivement d'effectuer la conversion d'Euros en Dollars (❶) (❸) et de Dollars en Euros.

Comportement de l'interface. La figure 3.1 présente trois états distincts de l'interface. Dans l'état initial (Etat 1), seul le champ de texte permettant la saisie d'une valeur est actif. Ce champ de texte est vide et les deux boutons de conversion sont inactifs : l'utilisateur ne peut donc pas presser l'un des deux boutons pour appeler la fonction de conversion du Noyau Fonctionnel.

Lorsque l'utilisateur saisit une valeur en entrée (Etat 2) les deux boutons de conversion deviennent actifs. Dans ce cas l'utilisateur peut effectuer une conversion, en l'occurrence en appuyant sur le bouton *Dollars -> Euros*. Le résultat est alors affiché dans le deuxième champ de texte.

Cette dernière action a pour effet de désactiver le bouton pressé (Etat 3). Le second bouton reste quant à lui actif. L'utilisateur peut alors soit :

- effectuer une conversion dans le sens inverse en pressant le bouton *Euros -> Dollars*. Dans ce cas, le bouton *Euros -> Dollars* est désactivé et le bouton *Dollars -> Euros* redevient actif. L'opération peut être réitérée un nombre illimité de fois.
- ou bien entrer une nouvelle valeur. Si le champ de saisie est vide, les deux boutons sont désactivés (Etat 1). Si le champ de saisie est non-vide, les boutons sont activés (Etat 2). Dans les deux cas, le champ de texte affichant le résultat est vidé.

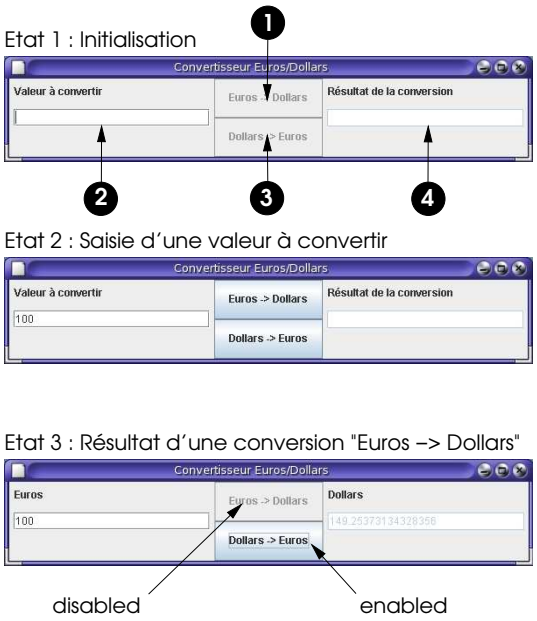


FIG. 3.1 – Etude de cas : convertisseur Euros/Dollars.

3.1.2 Le langage Java-Swing : Concepts

Swing est le nom officiel du kit de développement d'interface graphique léger qui fait partie des *Java Foundation Classes* (JFC). Swing ne remplace pas complètement AWT, mais fournit des composants d'interface plus performants. Notamment, les anciens composants AWT restaient liés à la plate-forme locale, car écrits en code natif, ce qui provoquait des différences d'apparence. Les composants SWING sont écrits en Java, ce qui leur confère plus de souplesse et d'adaptabilité. L'architecture de base de l'AWT, en

particulier la gestion des événements, demeure identique à celle de Java 1.1. Swing utilise le modèle de gestion des événements de Java 1.1.

3.1.3 Aspect Structurel d'une interface Java : les Widgets

Hiérarchie de widgets. Une interface graphique en Java est constituée d'un ensemble de composants graphiques (*widgets*) définis par les classes de la librairie AWT/Swing. Il y a principalement deux sortes de widgets : les **widgets élémentaires** et les **widgets composés de sous-widgets**.

1. les **widgets élémentaires** disposent :
   – d'une représentation graphique à l'écran ;
   – de la possibilité de réagir à des événements qui sont généralement la conséquence des actions de l'utilisateur, fournissant ainsi à ce dernier la possibilité d'exprimer des commandes ;
   Exemple de tels widgets : `JButton`, `JLabel`, `JTextfield`...
2. les **widgets conteneurs ou widgets parents.** Ces widgets sont composés de sous-widgets et structurent le code et la présentation visuelle. Ils définissent notamment :
   – une hiérarchie de widgets ;
   – et la gestion de la mise en page graphique de leurs widgets enfants grâce aux `LayoutManagers` ;
   Exemple de tels widgets : `Container`, `JPanel`, `JFrame`, etc.

Les widgets conteneurs, de la même manière que les widgets élémentaires, disposent d'une représentation graphique (ex. : couleur en fond) et de la possibilité de réagir à des événements.

La figure 3.2 présente la hiérarchie de widgets utilisée dans l'exemple du convertisseur.

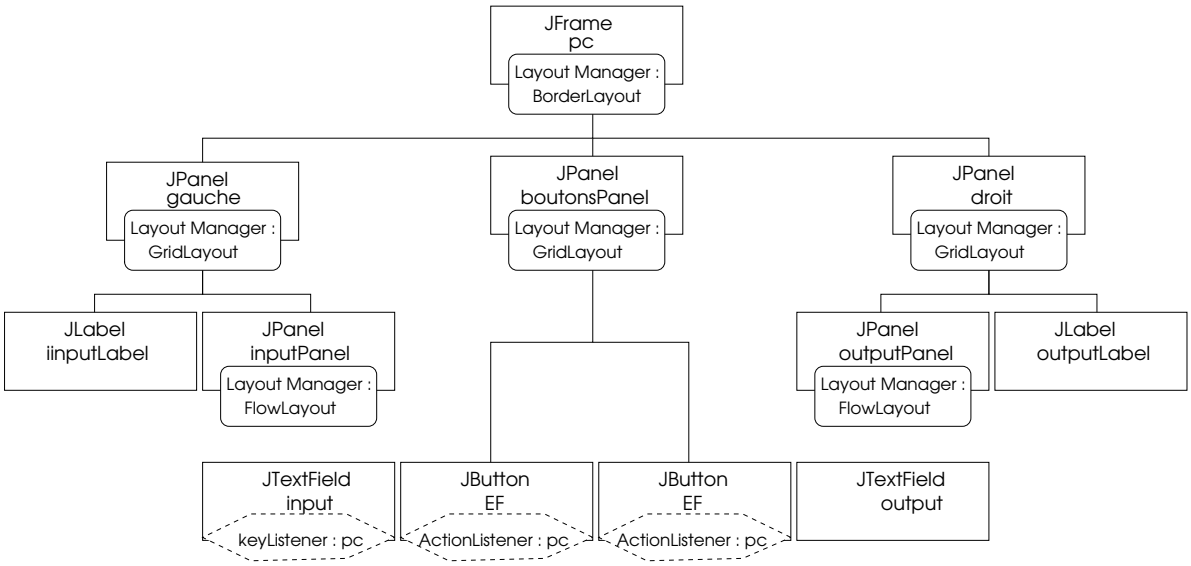


FIG. 3.2 – Hierarchie de widgets relative à l'étude de cas.

Ici, l'interface est constituée d'une fenêtre `JFrame`. Cette fenêtre est un conteneur possédant trois fils de type `JPanel`. La mise en page de ces panneaux est définie par un

`LayoutManager` de type `GridLayout`. Les feuilles de cet arbre sont des widgets élémentaires : deux `JLabel`, deux `JTextField` et deux `JButton`.

D'un point de vue pratique, une association widget/widget permettant de construire une hiérarchie de widgets est réalisée par l'utilisation de la méthode `add()`. La figure 3.3 présente un exemple de code Java-Swing associant à un widget conteneur `container` de type `JPanel` un widget élémentaire `button` de type `JButton`.

```
1 /* Declaration et instanciation de widgets */
2
3 JPanel container=new JPanel();
4 JButton button=new JButton();
5
6 /* Association des widgets */
7
8 container.add(button);
```

FIG. 3.3 – Exemple de création d'une hiérarchie de widget : association widget/widget.

Attributs des widgets. D'une manière générale chaque composant possède trois caractéristiques :

1. son *contenu* ou son *état*, tel que l'état coché ou non d'un bouton radio ;
2. son *apparence*, par exemple sa couleur, sa taille, etc. ;
3. son *comportement*, c'est-à-dire sa réaction suite à une action utilisateur.

Le contenu d'un widget est défini par un ensemble d'attributs. Une partie de ces attributs, héritée de classes abstraites, est commune à la majorité des widgets. Deux attributs sont tout particulièrement observés dans la suite : `visible` et `enabled`. Le premier attribut, de type booléen, conditionne la visibilité du widget sur l'interface. Le second, également de type booléen, conditionne l'activité du widget. Un widget *actif* et *visible* est en mesure d'accepter une action de l'utilisateur. Dans le cas contraire, l'utilisateur ne pourra pas interagir avec le widget. L'aspect visuel (Look'n Feel) du widget est modifié selon son état d'activité dans la majorité des cas (cf. figure 3.1).

La visibilité et l'activité d'un widget peuvent être modifiés en utilisant respectivement les méthodes `setVisible()` et `setEnabled()`. La non-visibilité d'un widget est transitive : si le widget conteneur est non visible sur l'interface, alors ses fils sont non visibles. Ceci ne concerne pas l'état d'activité d'un widget. Les widgets fils d'un widget conteneur inactif peuvent être actifs.

En résumé, concernant l'aspect structurel d'une interface graphique Java-Swing, on retient les faits suivants :

1. La structure d'une interface graphique Java est une **hiérarchie de widgets** : un widget conteneur peut avoir plusieurs fils, chaque fils pouvant être un widget conteneur ou un widget élémentaire ;
2. L'action d'un utilisateur sur un widget est possible si et seulement si le widget est **visible** et **actif** et si l'ensemble de ses ancêtres (parents, grands-parents...) dans la hiérarchie de widgets sont **visibles** ;

Gestion des événements

Tout système d'exploitation qui supporte les interfaces graphiques doit constamment surveiller l'environnement afin de détecter des événements tels que la pression sur une touche de clavier ou sur un bouton de la souris. Le système d'exploitation en informe alors les programmes en cours d'exécution. Chaque programme détermine ensuite s'il doit répondre à ces événements ou non.

Événement et écouteur d'événements. Dans les limites des événements connus d'AWT/SWING, il est possible de contrôler complètement la manière dont les événements sont transmis de la *source de l'événement* (par exemple, un bouton ou une barre de défilement) à *l'écouteur d'événement* (*event listener*).

La gestion des événements est donc réalisée par des objets (écouteurs) qui implantent une ou plusieurs interfaces *listeners* et qui sont connectés à différents widgets.

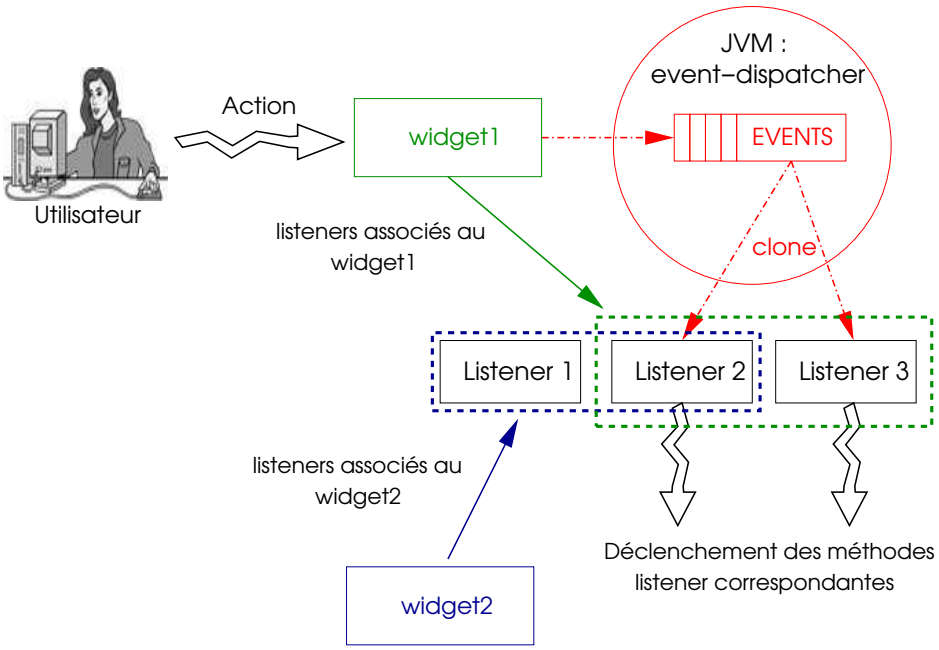


FIG. 3.4 – Gestion des événements : fonctionnement général.

Fonctionnement général. À à la suite d'une action de l'utilisateur sur un widget, ce widget génère un événement `AWTEvent`. Différents types d'événement peuvent être émis suivant le type de dispositif d'interaction ou du type d'action effectué par l'utilisateur : `ActionEvent` lors d'un clic sur un bouton, `WindowEvent` dans le cas d'une action sur une fenêtre, `KeyEvent` lors d'une pression sur une touche de clavier, `MouseEvent` lors d'un clic souris, etc.

Ensuite, le widget fait traiter une copie de cet événement par chacun des écouteurs d'événements qui lui sont connectés et qui sont capables de traiter ce type d'événement. Par exemple, un `ActionListener` traitera uniquement les `ActionEvent` et un `WindowListener` les `WindowEvent`. Plus généralement, un `XxxListener` traite uniquement les `XxxEvent`.

Le traitement d'un événement est réalisé en appelant la méthode (ou les méthodes) d'écouteurs correspondant à la nature de l'événement. La figure 3.4 récapitule le fonctionnement général précédemment décrit. On remarquera qu'un même écouteur peut être connecté à plusieurs widgets.

Déclarations et connexions des écouteurs d'événements. Un écouteur est une instance de classe qui implémente une interface spéciale appelée *interface écouteur* (*listener interface*). Un écouteur est connecté à un widget lorsque l'on souhaite définir le *comportement* de ce widget, c'est-à-dire lorsque l'on souhaite définir la réaction de ce widget suite à une action utilisateur.

Pour connecter un écouteur de type `Xxx` à un widget, on utilise la méthode `addXxxListener()` de ce dernier. Il existe une bijection entre les types d'événements, les écouteurs d'événements et les méthodes de connexion. Par exemple : un écouteur `XxxListener` écoute des événements du type `XxxEvent` et se connecte à un widget via la méthode `addXxxListener`.

Comme pour toutes les interfaces Java, lorsque qu'une classe implémente une interface *listener* elle doit nécessairement implémenter les méthodes définies par l'interface avec une signature correcte. Par exemple, pour implémenter l'interface `ActionListener`, la classe implémentant l'interface doit définir une méthode nommée `actionPerformed` qui recevra l'objet `ActionEvent` comme paramètre.

L'exemple de la figure 3.5 illustre ces principes.

```
1  public class MyPanel extends JPanel
2                       implements ActionListener
3      // MyPanel est un widget conteneur de type JPanel
4      // MyPanel est un écouteur d'événements de type ActionEvent
5  {...
6     public MyPanel(){ // Constructeur de classe
7      ...
8      // Declaration et instanciation d'un widget de type JButton
9      JButton myButton=new JButton();
10    // Connexion du widget à l'écouteur.
11    // this = MyPanel
12     myButton.addActionListener(this);
13    }
14     ...
15     public void actionPerformed(ActionEvent evt){
16     // Routine de réponse au clic sur le bouton
17     }
18 }
```

FIG. 3.5 – Exemple Java-Swing : création de widget et association widget/écouteur d'événements.

Traitement des événements. Comme déjà évoqué, un écouteur peut proposer plusieurs méthodes pour traiter un même événement. Le choix de la méthode appelée dépend alors de la nature précise de l'événement. Par exemple, dans le cas d'un widget auquel est connecté un `KeyListener` :

- un `KeyListener` doit implémenter les méthodes `KeyPressed`, `KeyReleased` et `KeyTyped` ;
- quand on enfonce la touche [a] du clavier, les méthodes `keyPressed()` puis `keyTyped()` sont appelées ;

– quand on relâche la touche, c'est la méthode `keyReleased()` qui est appelée.

La figure 3.6 présente une partie de l'architecture Événement/Écouteur de la bibliothèque AWT.

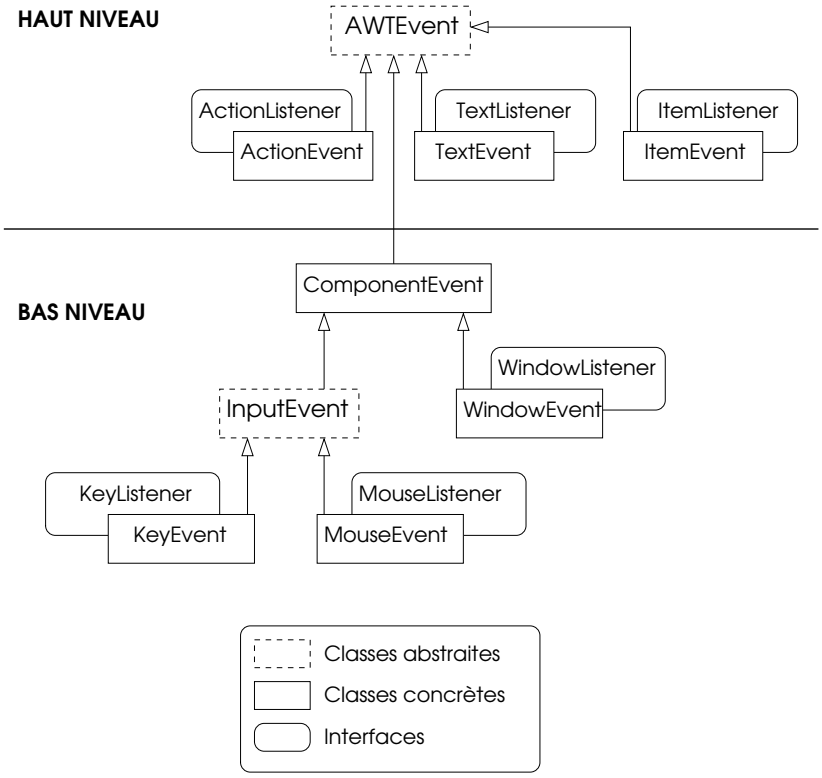


FIG. 3.6 – Architecture partielle Événement/Écouteur.

Ces événements disposent d'un certain nombre d'attributs caractérisant l'événement. Voici par exemple en figure 3.7 les informations qu'il est possible d'obtenir auprès de tout événement (`AWTEvent`) et auprès des événements concrets `ActionEvent` et `MouseEvent` :

`AWTEvent`
`getSource()` : renvoie le composant graphique qui a généré l'événement.
`getId()` : renvoie le type d'événement.
`ActionEvent`
`getModifiers()` : renvoie les modificateurs activés pendant cet événement. Ex : `ALT_MASK`, `CTRL_MASK`, `META_MASK`, ...
`MouseEvent`
`getX()`, `getY()` : renvoie la position relative à `getSource()`.

FIG. 3.7 – Exemple d'informations relatives aux événements.

3.1.4 Retour sur l'étude de cas

L'application présentée en début de chapitre est construite suivant une architecture de type SEEHEIM (cf. section 1.4). Le code source de l'application est constitué de 5 classes. La classe `Presentation.java` définit la partie graphique et interactive de l'interface.

Une classe `Controleur.java` gère le séquencement des entrées et des sorties : il s'agit du contrôleur de dialogue. Il dispose d'une visibilité sur la présentation et communique avec le noyau fonctionnel (classe `NF_Convertisseur.java`) via un adaptateur d'interface `ANF.java`. Enfin, une classe `SEEHEIM.java` permet d'instancier les différentes classes et possède la méthode principale `main()` permettant d'exécuter l'application.

Les figures 3.8, 3.9, 3.10 présentent une partie du code source extraite du module présentation de l'application.

La classe `Presentation` étend la classe `JFrame` et implémente les interfaces de type écouteur d'événements `ActionListener` et `KeyListener` (figure 3.8). Cette classe présente 12 attributs privés : 2 champs de saisie de texte de type `JTextField`, deux boutons de type `JButton`, un contrôleur de dialogue, 4 widgets conteneurs de type `JPanel` et enfin 2 champs de texte `JLabel`. La méthode d'instanciation de la classe (`Presention()`) assure l'instanciation ainsi que l'initialisation des widgets et l'abonnement des widgets aux écouteurs d'événements.

```
1  public class Presention extends JFrame
2              implements ActionListener, KeyListener {
3     private
4       JTextField input,output;//déclaration des zones d'édition
5       JButton EF,FE;//déclaration des boutons
6       JPanel gauche,droit,boutonsPanel,inputPanel,outputPanel;
7       JLabel inputLabel, outputLabel;
8       Controleur controleur;
9
10   public Presention() {
11         // Instanciation et initialisation de la présentation
12         initialiser();
13         activerboutons(false); // Désactivation des boutons
14         abonner();} // Abonnements des widgets aux écouteurs d'événements
15
16    public void activerboutons(boolean b) {
17         EF.setEnabled(b);
18         FE.setEnabled(b);}
19    }
```

FIG. 3.8 – Etude de cas : initialisation de la présentation (1).

La figure 3.9, présente la méthode `initialiser()` appelée par le constructeur de la classe `Presentation`. Elle effectue dans un premier temps l'instanciation des widgets élémentaires, la création des widgets conteneurs et la configuration de la mise en page des conteneurs (`LayoutManager`). La dernière opération consiste à construire la hiérarchie des widgets présentée en figure 3.2. Ici le conteneur principal est la classe elle-même : en effet, la classe étend la classe `JFrame` et peut donc intervenir en tant que conteneur. Cette fenêtre est constituée de trois panneaux principaux `inputPanel`, `outputPanel` et `boutonsPanel` qui contiennent respectivement : le champ de saisie `input`, les deux boutons de conversion `EF` et `FE` et enfin le champ de saisie permettant d'afficher le résultat `output`.

```
1     public void initialiser(){
2        [...]
3         /* Création des widgets élémentaires */
4
5         input=new JTextField(20);
6         output=new JTextField(20);
7         EF=new JButton("Euros -> Dollars");
8         FE=new JButton("Dollars -> Euros");
9         inputLabel=new JLabel("Valeur à convertir");
10        outputLabel=new JLabel("Résultat de la conversion");
11        [...]
12
13        /* Création des widgets conteneurs */
14
15        gauche=new JPanel();
16        droit=new JPanel();
17        boutonsPanel= new JPanel();
18        inputPanel = new JPanel();
19        outputPanel = new JPanel();
20        Container c = this.getContentPane();
21
22        /* Configuration des layout managers des Panel */
23
24        inputPanel.setLayout(new FlowLayout());
25        outputPanel.setLayout(new FlowLayout());
26        gauche.setLayout(new GridLayout(3,1));
27        [...]
28
29        /* Association widget-widget :  hiérarchie de widgets */
30
31        inputPanel.add(input);
32        outputPanel.add(output);
33        gauche.add(inputLabel);
34        gauche.add(inputPanel);
35        droit.add(outputLabel);
36        droit.add(outputPanel);
37        boutonsPanel.add(EF);
38        boutonsPanel.add(FE);
39        c.add(gauche,BorderLayout.WEST);
40        c.add(boutonsPanel,BorderLayout.CENTER);
41        c.add(droit,BorderLayout.EAST);
42
43        output.setEnabled(false);
44        this.pack();
45        this.setVisible(true);    }
```

FIG. 3.9 – Etude de cas : initialisation de la présentation (2).

La dernière méthode appelée (figure 3.10) lors de l'instanciation de la classe `Presentation` est la méthode `abonner()`. Un même écouteur est ici connecté à trois widgets distincts. L'écouteur est une fois de plus l'instance de la classe `Presentation` qui implémente les interfaces `ActionListener` et `KeyListener`. Cet écouteur est connecté au champ de saisie `input` et aux deux boutons permettant d'effectuer une conversion `EF` et `FE`.

Enfin, les deux méthodes présentées en figure 3.10 correspondent à une partie de l'ensemble des méthodes d'écouteurs d'événements devant être implémentées lorsqu'une classe étend les interfaces `ActionListener` et `KeyListener` (les corps des autres méthodes liées aux interfaces `ActionListener` et `KeyListener`, étant vides, ils ne sont pas présentés ici). La première méthode `actionPerformed(ActionEvent e)` répond aux événements émis lors d'un clic sur l'un des deux boutons.

Suivant la source de l'événement reçu (`EF` ou `FE`), l'écouteur appelle la méthode `convertirEF()` du contrôleur de dialogue : la valeur booléenne du paramètre d'entrée définit alors le sens de conversion choisi. Cette dernière méthode fait appel au noyau fonctionnel de l'application qui effectue la conversion explicite puis appelle à nouveau des méthodes de la présentation pour gérer le feed-back : en l'occurrence, la source de l'événement est désactivée, le second bouton est activé et le résultat de la conversion est affiché dans le champ de saisie `output`.

Le principe est identique en ce qui concerne la méthode `keyPressed(KeyEvent e)`. Cette méthode, déclenchée lors d'une entrée clavier, réalise un appel à la méthode `gestionInput()` du contrôleur de dialogue. Le résultat final de cet appel dépend du contenu du champ de saisie en entrée (`input`) : si celui-ci est vide les deux boutons sont désactivés, dans le cas contraire les deux boutons sont activés.

```
1   public void abonner() {
2     input.addKeyListener(this);
3     EF.addActionListener(this);
4     FE.addActionListener(this);
5     };
6
7     /** Méthodes listeners */
8
9  [...]
10    public void actionPerformed(ActionEvent e)
11    {if (e.getSource()==EF)
12     {controleur.convertirEF(true);
13     }
14     if (e.getSource()==FE)
15     {controleur.convertirEF(false);
16     }
17    }
18
19    public void keyPressed(KeyEvent e){
20      controleur.gestionInput();
21     }
22 [...]
23 }
```

FIG. 3.10 – Etude de cas : méthodes d'écouteurs d'événements.

3.2 La Méthode B et "B événementiel"

La méthode B, introduite par J.R. Abrial [Abrial, 1996], couvre les différentes étapes du développement d'un logiciel depuis la spécification abstraite jusqu'à une spécification concrète directement implémentable dans un langage de programmation. Cette méthode est basée sur le principe de raffinements successifs : un modèle abstrait est raffiné en un modèle plus concret par l'ajout successif de détails dans la modélisation. S'il respecte un certain nombre de propriétés, ce raffinement assure que les propriétés du modèle abstrait sont conservées par le modèle concret.

La méthode B a été conçue initialement comme une méthode de développement de logiciels dans lesquels on s'intéresse aux données et aux traitements. L'introduction des événements dans la méthode B événementiel permet la prise en compte des aspects comportementaux des systèmes. Concrètement un modèle B événementiel représente un sys-

tème de transitions codées par des événements et dont les propriétés sont exprimées dans la logique du premier ordre.

Cette section présente uniquement les éléments des méthodes B et B événementiel nécessaires à la compréhension de l'utilisation de B dans les travaux exposés dans ce mémoire. De plus amples informations sur la méthode B événementiel sont disponibles dans [Abrial, 1996], [Abrial & Mussat, 1998], [Metayer *et al.*, 2005], [Méry & Cansell, 2004], [Cansell & Méry, 2006], [ClearSy, 2001].

3.2.1 Machine Abstraite

Une *machine abstraite* est l'entité de base du mécanisme de structuration utilisé dans le processus de développement en B. Une machine abstraite permet de modéliser un système par un ensemble de variables qui définissent l'état du système et un ensemble d'opérations sur ces variables.

Une machine abstraite B est constituée d'un ensemble de clauses qui permettent de structurer les données et le contrôle (i.e. les traitements) du système modélisé. Le tableau 3.1 présente l'organisation des différentes clauses composant la spécification d'une machine abstraite.

MACHINE `Nom_De_La_Machine`
SETS
Déclaration des noms de type et d'ensembles
CONSTANTS
Déclaration des noms des constantes du système
PROPERTIES
Définitions des propriétés logiques sur les constantes
VARIABLES
Déclarations des noms de variables du modèle
INVARIANT
Définitions des invariants du système
ASSERTIONS
Définition des assertions
INITIALISATION
Définition des valeurs intiales des variables
OPERATIONS
Définition des opérations de la machine
END

TAB. 3.1 – Structure générique d'une machine abstraite.

Les variables de la machine, déclarées dans la clause VARIABLES, sont contraintes par des propriétés déclarées dans la clause INVARIANT. C'est notamment dans cette dernière clause que les variables seront typées en fonction des ensembles définis dans la clause SETS de la machine. La clause PROPERTIES permet d'exprimer des propriétés logiques sur les ensembles et les constantes.

Les opérations, ainsi que la clause INITIALISATION qui est une opération particulière, sont exprimées par des substitutions généralisées. Ces opérations font évoluer l'état du système en modifiant les valeurs des variables : le nouvel état doit alors satisfaire les invariants du système formulés dans la logique des prédicats du premier ordre.

3.2.2 Obligations de preuve et substitutions généralisées

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 dont l'énoncé est généré à partir de la description du système en question.

La modification des données par un événement s'effectue à l'aide d'un pseudo-code nommé *substitution*. Une substitution est une notation mathématique qui joue le rôle de transformateur de prédicats. Ces substitutions se basent sur le calcul de la plus faible précondition de [Dijkstra, 1976].

L'utilisation des substitutions pour l'expression des opérations permet d'établir systématiquement les OP 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 automatique s'assure que cette substitution est valide compte tenu des invariants de la machine et des préconditions de l'opération concernée.

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

Tab. 3.2 – Machine abstraite B avec une opération.

On note S une substitution et P un prédicat exprimant une post-condition. Suivant ces notations, $[S]P$ désigne la plus faible précondition qui établit P après l'exécution de la substitution S.

Le tableau 3.2 présente une machine B générique et simplifiée (sans définition d'ensembles, de constantes et de propriétés). Dans l'expression $u \leftarrow Op(w) = S$ de cette machine abstraite, Op désigne le nom de l'opération, w et u les paramètres d'entrée et de sortie de l'opération, et S la substitution définissant le corps de l'opération. Les OP de cette machine à vérifier sont décrites dans le tableau 3.3.

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

La première obligation de preuve INV_1 concerne l'initialisation : l'initialisation doit établir l'invariant du système après son appel (aucune valeur de variable n'est définie avant l'initialisation). La deuxième obligation de preuve INV_2 concerne l'opération Op : celle-ci doit préserver l'invariant, c'est-à-dire que sous couvert de l'invariant l'opération établit l'invariant après son exécution. La troisième OP INV_3 concerne la clause ASSERTIONS. $A(x)$ est un prédicat qui doit être vérifié dans tous les états : il s'agit d'une propriété du système qu'il est possible de vérifier à partir de l'invariant $I(x)$ du système.

Substitutions. Une opération Op peut revêtir différentes formes suivant la nature de la substitution utilisée pour la décrire. Par conséquent, l'obligation de preuve INV_2 possédera une expression particulière suivant le type de substitution utilisé pour décrire l'opération. Le tableau 3.4 présente les substitutions de haut niveau utilisées dans nos modèles ainsi que les transformations de prédicats utilisées pour le calcul de la plus faible précondition $\mathcal{WP}$. Dans ce tableau, x et l désignent des variables (ou des ensembles de variables), E et F des expressions, Q et G des prédicats et T une substitution élémentaire.

Nom	Substitution	Réduction
bloc	[**BEGIN** $T(x)$ **END**]$P(x)$	$[T(x)]P(x)$
précondition	[**PRE** $Q(x)$ **THEN** $T(x)$ **END**]$P(x)$	$Q(x) \wedge [T(x)]P(x)$
garde	[**SELECT** $G(x)$ **THEN** $T(x)$ **END**]$P(x)$	$G(x) \Rightarrow [T(x)]P(x)$
choix non borné	[**ANY** l **WHERE** $G(x,l)$ **THEN** $T(x)$ **END**]$P(x)$	$\forall x.(G(x,l) \Rightarrow [T(x)]P(x))$

TAB. 3.4 – Les différentes formes de substitutions utilisées pour définir le corps d'une opération.

Par exemple, dans le cas où l'opération est définie à l'aide d'une substitution de type *précondition*, l'obligation de preuve résultante est :

$$(INV_2) \qquad I(x) \wedge Q(x) \Rightarrow [T(x)]I(x)$$

Le tableau 3.5 définit les différentes formes de substitutions employées pour $T(x)$. Dans ce tableau E dénote une expression et T_1 et T_2 des substitutions élémentaires.

Nom de la substitution	Substitution	Réduction
devient égal	$[x := E]P(x)$	$P(x/E)$
appel d'opération	$[R \leftarrow Op(E)]P(x)$	$[X := E; S(x); R := Y]P(x)$
identité	$[skip]P(x)$	$P(x)$
simultanée	$[T_1(x) \parallel T_2(y)]P$	$[T_1(x)]P \wedge [T_2(y)]P$ avec $x \neq y$
séquencement	$[T_1(x); T_2(x)]P(x)$	$[T_1(x)][T_2(x)]P(x)$

TAB. 3.5 – Formes des 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 les paramètres effectifs. L'appel d'opération se définit sous quatre formes différentes, selon la présence de paramètres d'entrée et de sortie. Si Op est définie par $Y \leftarrow Op(X) = S$, la signification d'un appel de $R \leftarrow Op(E)$ (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) est obtenue en remplaçant les paramètres formels par l'expression des paramètres réels (idem pour le résultat).

La substitution *simultanée* permet la composition parallèle de deux (ou plusieurs) substitutions.

Décomposition d'une machine. La méthode B propose des techniques de structuration qui permettent la décomposition d'une machine B. En effet, il n'est pas envisageable de spécifier complètement un système au moyen d'une machine abstraite unique : le nombre d'OP serait alors trop important. L'utilisation de raffinements successifs est une des solutions utilisées pour résoudre ce problème.

3.2.3 Modèle B événementiel

En B événementiel, on parle de *modèle* plutôt que de *machine*. Les *opérations* définies en B classique sont remplacées par des *événements* et la clause OPERATIONS est remplacée par une clause EVENTS. Le tableau 3.6 présente la structure d'un modèle B événementiel générique.

MODEL `Nom_De_La_Machine`
SETS
Noms de types et noms d'ensembles
CONSTANTS
Déclaration des noms de constantes
PROPERTIES
Définition des propriétés logiques des constantes
VARIABLES
Déclaration des noms de variables
INVARIANT
Définition des propriétés statiques par des formules logiques
ASSERTIONS
Définition des assertions
INITIALISATION
Définition des valeurs intiales des variables
EVENTS
Énumeration des événements associés au modèle
END

Tab. 3.6 – Structure générique d'un modèle B événementiel.

Les événements

Un événement correspond à un changement d'état et modélise donc une transition discrète du système à modéliser. À ce propos, on peut citer les travaux de [Bert & Cave, 2000], [Bert *et al.*, 2005] et [Stouls & Potet, 2004] traitant de la construction de systèmes de transitions à partir de modèles B.

Un événement peut être représenté sous la forme $Evt \mathrel{\hat{=}} G \Rightarrow S$ où :

- Evt correspond au *nom* de l'événement ;
- G représente la *garde de l'événement* ;
- et S l'*action* ou *corps* de l'événement.

Une *action* est décrite par une substitution généralisée définissant la manière dont l'événement modifie la valeur des variables du système. L'action S d'un événement est exécutée uniquement lorsque sa garde G est vérifiée.

Le langage B événementiel est *asynchrone*. Si plusieurs 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 indéterminé. La durée d'exécution d'un événement est toutefois considérée comme nulle et cet instant correspond au changement d'état.

B événementiel n'admet que trois formes possibles d'événements. Ces différentes formes sont présentées dans le tableau 3.7. Dans ce qui suit, x désignera l'ensemble des variables d'état du système. x est définie dans la clause VARIABLES du modèle.

Événement simple. La garde d'un événement simple est toujours vraie. $S(x)$ est une substitution qui modifie l'état de la variable x (Tab.3.7,A).

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 (Tab.3.7,B).

Événement indéterministe. L'événement indéterministe est gardé par l'expression logique $\exists l.G(x,l)$. Cet événement ne peut se déclencher que s'il existe des valeurs pour les variables locales l qui satisfont la condition $G(x,l)$ (Tab.3.7,C).

(A)	(B)	(C)
$Evt =$ **BEGIN** $S(x)$ **END**	$Evt =$ **SELECT** $G(x)$ **THEN** $S(x)$ **END**	$Evt =$ **ANY** l **WHERE** $G(x,l)$ **THEN** $S(x,l)$ **END**

TAB. 3.7 – Les différents types d'événements.

Preuves de consistance d'un modèle B événementiel

Prédicats avant-après associés aux substitutions généralisées. Le prédicat avant-après $prd_x(S)$ d'une substitution S dénote une relation binaire entre les valeurs avant et après la substitution S pour les variables x. Par convention, la valeur d'une variable var avant la substitution est notée var et celle après la substitution est notée var'. Plus généralement, si x dénote un ensemble de variables d'état du modèle, on notera x et x' leurs valeurs avant et après la transition définie par l'événement. La définition formelle du prédicat avant-après est la suivante :

$$prd_x(S) \mathrel{\hat{=}} < S > (x = x')$$

$$prd_x(S) \mathrel{\hat{=}} \neg[S](x \neq x')$$

$< S > R$ désigne le conjugué de la plus faible pré-condition $[S]R$, c'est à dire $\neg[S]\neg R$. Intuitivement, si la formule $< S > R$ est vérifiée alors il existe une exécution de S qui vérifie la post-condition R. Si x désigne l'ensemble des variables du système alors $prd_x(S)$ caractérise l'ensemble des couples (x, x') tels que x' désigne les valeurs de l'état après l'exécution de S et x désigne celles avant l'exécution de S.

A titre d'exemple, le tableau 3.8 présente l'expression des prédicats avant-après pour trois substitutions généralisées.

Substitution	Prédicat avant-après
$x := E(v)$	$x' = E(v) \wedge y' = y$
skip	$x' = x \wedge y' = y$
$Evt =$ **ANY** l **WHERE** $G(v, l)$ **THEN** $x := F(v, l)$ **END**	$\exists l.(G(v, l) \wedge x' = F(v, l)) \wedge y' = y)$

Tab. 3.8 – Exemples de prédicats avant-après de substitutions généralisées.

Dans le tableau 3.8, la lettre y représente l'ensemble des variables de v (espace des états) qui sont distinctes de celles de x.

Preuves de consistance d'un modèle B événementiel. Soit un modèle B événementiel $\mathcal{M}$ constitué d'un ensemble de variables v et d'un contexte défini par les ensembles s (clause SETS) et les constantes c (clause CONSTANTS). Les propriétés des constantes sont dénotées par $P(s, c)$ et l'invariant par $I(s, c, v)$. Soit également un événement Evt de $\mathcal{M}$ possédant une garde $G(s, c, v)$ et un prédicat avant-après $R(s, c, v, v')$.

Il convient tout d'abord de vérifier la faisabilité de l'événement. Pour cela il faut qu'il existe une valeur v' telle que le prédicat avant-après de l'événement Evt soit vérifié. Ce prédicat est à vérifier sous couvert des propriétés $P(s, c)$, de l'invariant $I(s, c, v)$, et de la garde $G(s, c, v)$. Cette obligation de preuve (*faisabilité de l'événement*) est notée FIS. La deuxième obligation de preuve à fournir afin d'assurer la consistance d'un événement est

la *préservation de l'invariant* (INV). Ces deux obligations de preuve sont représentées dans le tableau 3.9 suivant :

$P(s,c) \wedge I(s,c,v) \wedge G(s,c,v) \Rightarrow \exists v'.R(s,c,v,v')$	FIS
$P(s,c) \wedge I(s,c,v) \wedge G(s,c,v) \wedge R(s,c,v,v') \Rightarrow I(s,c,v')$	INV

Tab. 3.9 – Obligations de preuve : Faisabilité et préservation de l'invariant.

En fonction de la forme de l'événement considéré, l'expression de ces obligations de preuve est légèrement modifiée (du fait, par exemple, de la différence d'expression existante entre les gardes d'événement déterministe ou indéterministe). Ces expressions peuvent être trouvées dans [Metayer *et al.*, 2005].

Il existe une règle spécifique pour l'événement d'initialisation. Avec $RI(s,c,v')$ le prédicat avant-après de la substitution associée à l'initialisation[1], les deux OP simplifiées à prouver INI_FIS et INI_INV sont données par le tableau 3.10.

$P(s,c) \Rightarrow \exists v'.RI(s,c,v')$	INI_FIS
$P(s,c) \wedge RI(s,c,v,v') \Rightarrow I(s,c,v')$	INI_INV

Tab. 3.10 – Obligations de preuve : Faisabilité et préservation de l'invariant pour l'initialisation.

Enfin, il est parfois utile de prouver que le modèle défini ne se bloque pas. Autrement dit, il s'agit de prouver que quel que soit l'état du système, il existe toujours une transition réalisable permettant un changement d'état du système. Ceci est obtenu simplement en stipulant que la disjonction des gardes des événements du modèle est toujours vraie sous couvert des propriétés des constantes et de l'invariant du système. Le tableau 3.11 présente cette OP spécifique, où $G_1(s,c,v)$, ..., $G_n(s,c,v)$ représentent les gardes des événements.

$P(s,c) \wedge I(s,c,v) \Rightarrow G_1(s,c,v) \vee \ldots G_n(s,c,v)$	$DLKF$

Tab. 3.11 – Propriété de non-blocage du système.

Raffinements

"The art of progress is to preserve order amid change
and to preserve change amid order"
Alfred North Whitehead

Un modèle B abstrait peut être raffiné en un modèle plus concret. Lors d'un raffinement, les éléments du modèle abstrait sont remplacés progressivement par d'autres éléments plus concrets qui respectent les conditions définies par le modèle abstrait. Le processus de raffinement consiste à :

– remplacer les structures de données abstraites par des structures de données concrètes ;

[1] Il s'agit en réalité d'un prédicat "après" puisque l'événement d'initialisation ne possède pas de valeurs pour les variables avant l'événement.

- remplacer les substitutions abstraites par des substitutions concrètes ;
- ajouter éventuellement de nouvelles variables apparaissant du fait du niveau de granularité plus fin du modèle concret ;
- ajouter éventuellement de nouveaux événements.

REFINEMENT `Nom_Du_Raffinement`
REFINES
`Nom_du_modèle_abstrait`
SETS
Noms de types et noms d'ensemble
CONSTANTS
Déclaration du nom des constantes
PROPERTIES
Définitions des propriétés logiques sur les constantes
VARIABLES
Définitions des noms de variables du système
INVARIANT
Définitions des propriétés statiques du modèle
VARIANT
Définition des variants décrémenté par les événements.
ASSERTIONS
Définition des prorpiétés sur les variables et les constantes
INITIALISATION
Définition des valeurs intiales des variables
EVENTS
Énumération des événements associés au modèle
END

Tab. 3.12 – Structure générique d'un raffinement B événementiel.

Un raffinement est un modèle dont les comportements sont des comportements du modèle abstrait. Le tableau 3.12 présente la structure générique d'un raffinement. À la différence d'un modèle, un raffinement possède une clause supplémentaire : la clause VARIANT discutée ci-après.

Un raffinement doit satisfaire l'invariant du modèle abstrait. Un nouvel invariant est ajouté au modèle du raffinement. Cet invariant consiste à lier les variables abstraites (du modèle abstrait) aux variables concrètes (du modèle du raffinement). Cet invariant est appelé *invariant de collage*. Celui-ci est nécessaire pour établir que le raffinement vérifie l'invariant abstrait.

Dans un raffinement, on retrouve les événements du modèle abstrait avec le même nom, mais avec des gardes et des actions (corps) éventuellement modifiés. Un raffinement est considéré comme correct vis-à-vis de son abstraction si :

- l'initialisation préserve l'invariant abstrait (et éventuellement les nouveaux invariants contraignant les nouvelles variables introduites) ;

– chaque événement du système est faisable et préserve l'invariant abstrait (et éventuellement les nouveaux invariants).

Les modèles B événementiel sont généralement représentés en B classique car, jusqu'à présent, il n'existait pas encore d'outil autorisant la description de modèles B événementiel. Lors de la traduction, il faut alors ajouter dans la clause ASSERTIONS une propriété qui permet d'assurer que le système est réactif. Cette propriété est exprimée en indiquant que la disjonction des gardes abstraites implique la disjonction des gardes concrètes. Informellement, ceci revient à assurer que le système concret ne se bloque pas plus que son abstraction.

Enfin, dans un raffinement B événementiel, un événement abstrait peut être raffiné par plusieurs événements concrets. Il est alors envisageable qu'un événement puisse prendre la main indéfiniment, c'est-à-dire que la substitution de cet événement concret établisse sa propre garde. Dans ce cas, l'événement garde la main et la progression du système n'est plus assurée : le raffinement n'est alors plus en adéquation avec son abstraction.

Afin d'assurer la progression des nouveaux événements, et par conséquent d'empêcher un événement de prendre la main indéfiniment, un ou des variants sont introduits dans le raffinement. Un variant est un entier naturel strictement décroissant défini dans la clause VARIANT du modèle. Chaque nouvel événement introduit doit faire décroître ce variant : ceci assure que les événements concrets finiront par redonner le contrôle à l'abstraction.

Des informations plus complètes sur ces aspects et sur la méthode B événementiel en général peuvent être trouvées dans [Abrial, 2000], [Abrial *et al.*, 2005], [Cansell, 2003].

Exemple de raffinement B événementiel

On reprend ici l'exemple d'une horloge de Cansell [Cansell, 2003]. Si l'on veut définir uniquement le comportement de l'heure d'une horloge on peut construire le modèle suivant :

MODEL *horloge*
VARIABLES
h
INVARIANT
$h \in 0..23$
INITIALISATION
$h := 13$ **EVENTS**
incr= **select** $h \neq 23$ **then** $h := h + 1$ **end**;
zero= **select** $h = 23$ **then** $h := 0$ **end**;
END

TAB. 3.13 – Exemple de modèle B événementiel : comportement d'une horloge.

Dans ce modèle, une seule variable h représentant l'heure est définie. L'événement *incr* incrémente l'heure et l'événement *zero* réinitialise l'heure à 0.

Il est possible de raffiner ce modèle en ajoutant à cette modélisation les minutes : une variable m est ajoutée au modèle et un nouvel événement *tictac* est introduit. La figure 3.14 présente le raffinement obtenu.

REFINEMENT *horloge_minute*
REFINES *horloge*
VARIABLES
h, m
INVARIANT
$m \in 0..59 \wedge$ /*invariant de typage*/
$h = h_1$ /*invariant de collage*/
ASSERTION
/* *Propriété de non-blocage* */
$h \in 0..23 \wedge (m \in 0..59 \wedge h = h_1) \wedge$
$h \neq 23 \vee h = 23$
$\Longrightarrow$
$(h_1 \neq 23 \wedge m = 59) \vee (h_1 = 23 \wedge m = 59) \vee m \neq 59$
VARIANT
$59 - m$
INITIALISATION
$h_1 := 13 \parallel m := 0$ **EVENTS**
incr= **select** $h_1 \neq 23 \wedge m = 59$ **then** $h_1 := h_1 + 1 \parallel m := 0$ **end**;
zero= **select** $h_1 = 23 \wedge m = 59$ **then** $h_1 := 0 \parallel m := 0$ **end**;
tictac= **select** $m \neq 59$ **then** $m := m + 1$ **end**;
END

Tab. 3.14 – Exemple de raffinement B événementiel : comportement d'une horloge.

Dans ce raffinement, la variable h est partagée par les deux modèles : h est alors renommée h_1 dans le modèle raffiné et $(h = h_1)$ est ajouté à la clause INVARIANT du modèle raffiné : il s'agit de l'invariant de collage.

Nouveaux événements. Les nouveaux événements apparaissant dans le modèle concret raffinent l'événement abstrait **skip** $(x := x)$. Dans le cas du raffinement *horloge_minute*, on peut remarquer que le nouvel événement *tictac* incrémente les minutes et laisse l'heure h inchangée : il raffine donc bien l'événement **skip**.

Propriété de non-blocage. Si le raffinement de modèle était défini comme le raffinement de chaque événement, cela ne fonctionnerait pas, car comme les gardes des événements sont renforcées il se pourrait, si les gardes des nouveaux événements ne compensent pas, que le modèle raffiné se bloque plus que le modèle abstrait. Pour éviter cet écueil, il faut ajouter une obligation de preuve qui assure cette propriété : il s'agit de la propriété de non-blocage. Cette propriété exprime que, sous couvert des invariants et de la disjonction

des gardes abstraites, il faut que l'on puisse déduire la disjonction des gardes concrètes. Cette propriété est ajoutée au modèle dans la clause ASSERTION du raffinement.

Variant. Même si la propriété de non-blocage est vérifiée par rapport à l'abstraction, il se peut que les nouveaux événements puissent se déclencher indéfiniment. Dans ce cas, les événements abstraits ne sont plus observés puisque les nouveaux événements raffinent l'événement **skip**. C'est comme si le modèle abstrait se bloquait. Cette situation n'est pas souhaitable : les nouveaux événements ne doivent pas garder le contrôle indéfiniment. Pour cela, un variant (entier naturel) est introduit et chaque nouvel événement doit faire décroître ce variant. Les nouveaux événements doivent alors satisfaire les Obligations de Preuve (OP) présentées dans le tableau 3.15. Dans ce tableau, I dénote l'invariant du système, J l'invariant de collage entre les variables abstraites x et les variables concrètes y du raffinement, RC dénote le prédicat "avant-après" de l'événement concret et V dénote le variant du raffinement.

Première OP	Deuxième OP
$I(x) \wedge J(x,y)$	$I(x) \wedge J(x,y) \wedge RC(y,y')$
$\Longrightarrow$	$\Longrightarrow$
$V(y) \in \mathbb{N}$	$V(y') < V(y)$

TAB. 3.15 – Obligations de Preuve liées à l'introduction d'un variant.

La première OP peut aisément se prouver en assertion. Pour l'exemple présenté, les deux obligations de preuve à prouver sur l'événement *tictac* sont les suivantes :

Première OP	Deuxième OP
$h \in 0..23 \wedge (m \in 0..59 \wedge h = h_1)$	$h \in 0..23 \wedge (m \in 0..59 \wedge h = h_1) \wedge m \neq 59$
$\Longrightarrow$	$\Longrightarrow$
$59 - m \in \mathbb{N}$	$59 - (m+1) < 59 - m$

TAB. 3.16 – Obligations de Preuve liées à l'introduction d'un variant : exemple.

Quelques notations B événementiel

Le tableau 3.17 référence les notations B utilisées dans les chapitres suivants. Dans ce qui suit : x et y dénotent les élements appartenant respectivement aux ensembles X et Y.

Outils pour le B événementiel

Un nouvel outil permettant d'implémenter des modèles B événementiel est disponible en open source depuis peu : il s'agit de la plateforme RODIN[2].

[2]Renseignements et téléchargement de l'outil : http://rodin.cs.ncl.ac.uk/, http://rodin-b-sharp.sourceforge.net/.

Concernant la méthode B plusieurs outils sont diponibles : l'Atelier B[3] et sa version libre B4free[4] développée par Clearsy et son interface graphique Click'n'Prove[5] [Abrial & Cansell, 2003] développé par J.R Abrial et D. Cansell.

Symboles B	Sémantique
$\vee$, $\wedge$, $\neg$	disjonction, conjonction et négation
$\forall$, $\exists$	quantificateurs universel et existentiel
$\cup$, $\cap$, $\subseteq$, $\subset$	union, intersection, inclusion, inclusion stricte d'ensembles
$\mathbb{P}(X)$	ensemble des parties de X
$\leftrightarrow$	ensemble de relations : $X \leftrightarrow Y \widehat{=} \mathbb{P}(X \times Y)$
$x \mapsto y$	définition du couple (x, y)
$\nrightarrow$, $\rightarrow$	ensemble de fonctions partielles, ensemble de fonctions totales : $X \nrightarrow Y \widehat{=} \{r \mid r \in X \leftrightarrow Y \wedge (r^{-1}; r) \subseteq id(Y)\}$ avec ; la composition et id la relation d'identité. $X \rightarrow Y \widehat{=} \{f \mid f \in X \nrightarrow Y \wedge dom(f) = X\}$ avec dom le domaine de f.
$\Leftarrow$	surcharge d'une relation : si $R \in X \leftrightarrow Y$ et $Q \in X \leftrightarrow Y$ alors $Q \Leftarrow R = \{x, y \mid (x, y) \in X \times Y \wedge (((x \mapsto y) \in Q \wedge x \notin dom(R)) \vee (x \mapsto y) \in R)\}$
$\parallel$	substitution simultanée

Tab. 3.17 – Quelques notations du langage B.

3.3 Méthode formelle : NuSMV

SMV [Cadence Berkeley Labs, 1998] est un outil de model-checking symbolique développé à l'université de Carnegie Mellon [Burch *et al.*, 1990]. Il comporte un langage de description des automates (machines à états finis). Ce langage permet de traiter des systèmes manipulant uniquement des structures de données finies (booléen, type énuméré, intervalle fixé, tableau de dimension fixe) et de valider des formules LTL. Le langage d'entrée de SMV permet de décrire le modèle du système et sa spécification.

Le modèle utilisé par SMV est une structure de Kripke dont les états sont définis par un ensemble de variables d'états qui peuvent être du type booléen ou scalaire. La spécification est définie dans la classe SPEC par un ensemble de formules de logique temporelle LTL. Ces formules LTL représentent les propriétés attendues du système. Enfin, un générateur de code est associé à SMV.

NuSMV [Cavada *et al.*, 2005], [Cimatti *et al.*, 2000] est un model-checker qui tire son origine d'une ré-implémentation et d'une extension de SMV. NuSMV permet la représentation de systèmes de transitions finis synchrones et asynchrones, et la spécification des systèmes exprimée en CTL ou LTL. La syntaxe du langage NuSMV est très proche de celle de SMV.

[3]Renseignements : http://www.atelierb.eu/
[4]Téléchargement : http://www.b4free.com/
[5]Téléchargement : http://www.loria.fr/ cansell/cnp.html

Dans ce qui suit, on se limite à la présentation du sous-ensemble du langage utilisé par la suite.

MODULE *Nom_du_module*(*parametres*)
VAR
Déclarations des attributs du modules
$x_1 : Type; \ldots; x_k : Type;$
DEFINE
Définitions de descriptions concises (symboles)
$Id_1 := Exp(x_1, \ldots, x_k)$
$\ldots$
$Id_p := Exp(x_1, \ldots, x_k)$
INIT
Initialisation du module
$x_k := \ldots;$
$\ldots$
TRANS
Axiomes définissant les relations de transitions
ASSIGN
Initialisation du module et définitions des relations de transitions
init$(x_i) := \ldots;$
$\ldots$
init$(x_j) := \ldots;$
next$(x_l) :=$
case
$\ldots$
esac;
next$(x_m) := \ldots;$
case
$\ldots$
esac;
SPEC
Propriétés exprimées en CTL ou LTL

Tab. 3.18 – Structure générique d'un module NuSMV.

3.3.1 Principe de modélisation en NuSMV

Un modèle NuSMV est constitué d'un ensemble de modules. Le tableau 3.18 présente la structure générique et simplifiée d'un module NuSMV. Un module NuSMV est identifié par un *nom* et un ensemble de *paramètres d'entrée*. Les attributs du module, dont les valuations définissent les états du système de transitions décrit, sont déclarés et typés dans la clause VAR. L'*état initial* du module est défini dans la clause INIT (ou dans la clause ASSIGN par l'intermédiaire du mot clef *init*) et la *relation de transition* définissant

l'évolution de l'état du module dans la clause ASSIGN. Cette relation de transition peut également être définie par des *axiomes* dans une clause TRANS. La clause DEFINE permet la définition de raccourcis d'écriture en associant à un identifieur une expression construite à partir des variables du module. Enfin, on traite dans la clause SPEC toutes les formules logiques exprimées en CTL ou LTL devant être vérifiées par le système de transitions décrit par le module.

3.3.2 Systèmes de transitions

Le langage d'entrée de NuSMV, permet la description d'un système de transitions par la donnée de sa relation de transition.

1. **Les variables** dans NuSMV peuvent être déclarées comme des booléens ou bien des types énumérés. Elles peuvent être déclarées également comme des tableaux.
2. **Les états du système** de transitions sont composés d'une valuation de toutes les variables du système spécifié.
3. **L'état initial** est défini par la valeur initiale de chaque variable (ou attribut) du système. L'état initial peut être défini de deux manières :
   - en utilisant la clause $init()$. $init(x)$ où x est une variable, définit l'état initial de la variable x. Par exemple, l'instruction $init(x) := 0$ affecte la valeur 0 à la variable x dans l'état initial ;
   - en utilisant l'instruction d'affectation après le mot réservé INIT. L'instruction $x := 3$ introduite après le mot clef INIT définit la valeur initiale de x à 3.
4. **La relation de transition** définit à partir d'un état donné l'état suivant du système. Cette relation de transition est définie pour chaque variable et peut être exprimée de deux manières :
   - en utilisant la notation $next()$ qui détermine l'état suivant d'une variable. $next(x) := 1$ affecte la valeur 1 à x dans l'état suivant l'état courant ;
   - en utilisant une expression booléenne introduite par le mot clef TRANS qui définit une relation de transition entre les états. Par exemple $next(x) = 0 \Rightarrow next(y) = 1$ signifie que quand la variable x passe à la valeur 0, y prend la valeur 1.

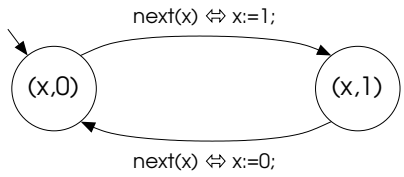
Toutes ces déclarations, ainsi que les définitions introduites par le mot clef ASSIGN peuvent être combinées. Elles s'ajoutent l'une après l'autre et restreignent l'ensemble des états initiaux ou des transitions possibles.

Produit synchronisé. Chaque variable dans le système est définie par son automate. Le système de transitions global est alors obtenu en effectuant le produit synchronisé des différents systèmes de transitions associés à chaque variable. On peut noter à ce propos que l'intérêt des model-checkers symboliques réside dans leur capacité à ne pas construire explicitement (i.e. complètement) cet automate. Rappelons que ceci reste vrai uniquement dans le cas de systèmes à états finis.

Cette opération de produit synchronisé permet l'assemblage de système. À la différence de B événementiel où la modélisation d'un système est obtenue par décomposition, en NuSMV le modèle global est le résultat d'une composition de sous-systèmes.

La figure 3.11 présente un exemple de module NuSMV et son système de transitions associé. Cet exemple définit un système à une seule variable dont les valeurs possibles sont

0 ou 1 telle que déclarée dans la clause VAR du module. Initialement, la variable x prend la valeur 0 (état initial du système). L'état suivant est défini par la notation $next()$: dans le cas où la valeur de x vaut 0, alors la valeur suivante de x sera 1, et vice versa. Ceci est exprimé par l'instruction choix $case \ldots esac$.



MODULE Exemple
VAR
$x : \{0, 1\}$;
ASSIGN
$init(x) := 0$;
$next(x) :=$ **case**
$0 : 1$;
$1 : 0$;
esac;

FIG. 3.11 – Exemple de module NuSMV et son automate associé.

Transition indéterministe. Il est possible de définir une relation de transition indéterministe par le biais de la notation *next* et de l'instruction $case \ldots esac$. Par exemple, soit x une variable prenant ses valeurs dans l'ensemble $\{0, 1, 2, 3\}$. Dans ce cas, le tableau 3.12 présente la définition d'une relation de transition indéterministe pour la variable x. Notamment dans les cas où la variable x vaut 0, alors la valeur suivante de x sera de manière indéterministe 0 ou 1.

$next(x) :=$ **case**
$0 : \{1, 2\}$;
$1 : 3$;
$2 : 3$;
$3 : \{0, 3\}$;
esac;

FIG. 3.12 – Exemple de transition indéterministe en NuSMV.

3.3.3 Expression de propriétés NuSMV

Plusieurs propriétés peuvent être spécifiées pour un même système. Ces propriétés sont exprimées sous forme de formules de logique temporelle CTL ou LTL et sont introduites dans la clause SPEC du module décrivant le système. Le model-checker NuSMV est capable d'effectuer la vérification de ces propriétés. Dans le cas où l'une des propriétés spécifiées est fausse, NuSMV arrête son analyse et fournit un contre-exemple sous la forme d'une trace violant la propriété. Cette trace peut alors être utilisée pour trouver l'erreur qui a causé l'échec de l'analyse.

3.4 Conclusion

Ce chapitre a présenté en guise de prérequis le langage Java-Swing et les langages formels B et NuSMV.

Le langage Java, associé à la bibliothèque Swing, est un langage adapté à l'implémentation d'IHM. La présentation de l'IHM (*aspect structurel* de l'interface) est constituée d'une hiérarchie de widgets : des widgets de type conteneur peuvent contenir des widgets conteneurs ou des widgets élémentaires. L'*aspect comportemental* de l'interface est encodé par l'intermédiaire d'interface Java jouant le rôle d'écouteur d'événements. Lors d'une action de l'utilisateur, un événement est émis au sein de la JVM. Cet événement est traité et distribué aux écouteurs d'événements concernés, c'est-à-dire aux écouteurs associés au widget utilisé pour l'interaction et capable de traiter le type d'événement émis. Une ou plusieurs méthodes d'écouteurs peuvent alors être exécutées : il s'agit du traitement de l'interface associé à l'action de l'utilisateur.

La méthode B événementiel est une méthode formelle basée sur un système de preuve de type démonstration de théorème (*Theorem Proving*). Un modèle B est constitué d'un ensemble de clauses permettant de définir les variables d'état du système et d'une clause permettant de représenter l'évolution de cet espace d'états sous la forme d'événements. Un ensemble d'invariants (contraintes devant être satisfaites par les variables à tout instant) peut être défini. Des Obligations de Preuves doivent alors être déchargées afin de prouver que l'exécution des événements respecte la préservation de ces invariants. Un événement est constitué d'une *garde* (condition booléenne exprimée en logique du premier ordre) et d'une *action* (substitutions généralisées). L'*action* est exécutée uniquement lorsque la garde de l'événement est évaluée à vrai. En outre, la méthode B propose un mécanisme de raffinement. Un modèle B abstrait peut être raffiné en un modèle plus concret par l'ajout de variables et d'événements permettant de représenter plus finement le comportement du système. La construction d'un raffinement nécessite une preuve de correction de raffinement. L'approche B événementiel est adaptée à la représentation de système interactif et en particulier à la représentation du dialogue de l'interaction homme-machine.

La méthode NuSMV est une méthode formelle basée sur un système de preuve de type vérification exhaustive de modèle (*model-checking*). Un modèle NuSMV permet de représenter les variables d'état du système. Le comportement du système est encodé par des systèmes de transitions, un système de transitions étant défini pour chaque variable du système. Les propriétés du système sont exprimées par des formules de logique temporelle CTL ou LTL. En plus des propriétés de sûreté qui peuvent être vérifiées en B événementiel (invariants du système), il est également possible en NuSMV de vérifier des propriétés de vivacité et d'atteignabilité.

Le chapitre suivant présente les principes de modélisation formelle d'applications Java-Swing suivant les méthodes B événementiel et NuSMV.

Chapitre 4

Modélisation formelle d'applications Java-Swing : mise en oeuvre avec B événementiel et NuSMV

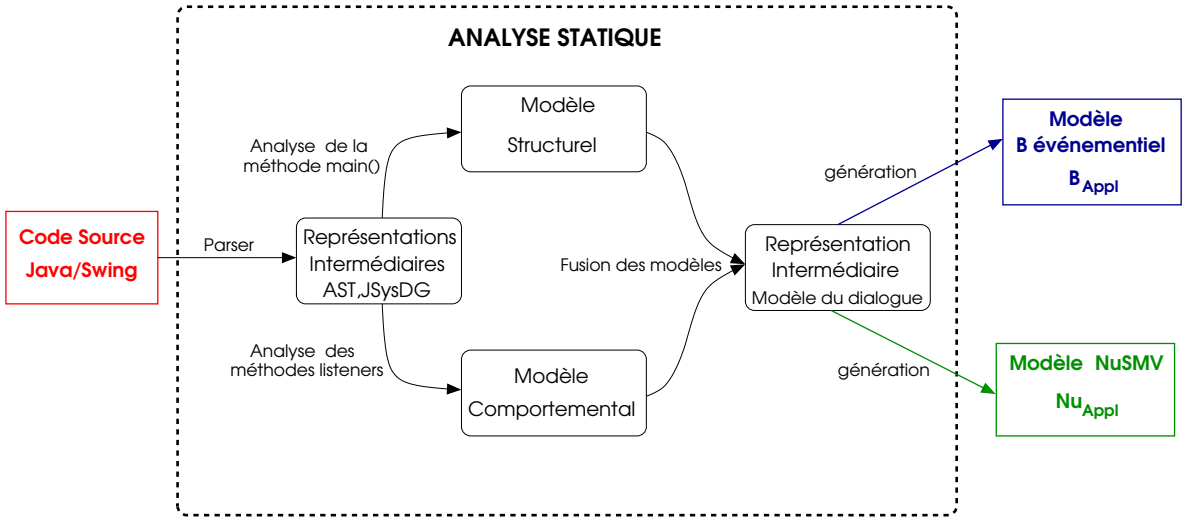


FIG. 4.1 – Schéma de principe simplifié du processus d'extraction.

4.1 Introduction

Ce chapitre expose les principes utilisés pour la modélisation formelle (B événementiel et NuSMV) d'applications interactives codées en Java-Swing. Cette étape constitue la première étape de la méthodologie de validation des systèmes interactifs présentée dans cette thèse. On s'attache ici au problème de l'extraction d'un modèle B événemen-

tiel. Concernant l'extraction d'un modèle NuSMV, on présente de manière informelle sa construction à partir du premier modèle B extrait.

Les détails de l'analyse statique d'un code source Java-Swing pour l'extraction de modèles formels (techniques et algorithmes) font l'objet du chapitre 5.

Objectifs. L'objectif de l'analyse statique d'une application est d'extraire un modèle exploitable du point de vue de la validation du système. Suivant le type de validation effectué, et par conséquent du type de propriétés du système que l'on souhaite vérifier, différents types de modèles peuvent être exploités. Chacun de ces modèles doit capturer les aspects pertinents du système du point de vue des propriétés à vérifier. Autrement dit, un modèle formel extrait par analyse d'un code source est le résultat d'une projection dont le résultat est une abstraction du système d'origine qui conserve les informations relatives aux propriétés à vérifier.

L'objectif final de l'approche présentée dans ce mémoire est la validation du système du point de vue de son utilisabilité. On souhaite plus particulièrement capturer l'évolution des *disponibilités d'interaction* au cours de l'exécution du système afin de s'assurer que les scénarios d'interaction acceptés par l'interface sont en adéquation avec les scénarios d'interaction attendus par l'utilisateur.

Disponibilité d'interaction. La *disponibilité d'interaction* d'une interface en un instant de l'exécution est caractérisée par l'état de l'ensemble des widgets présents à l'écran avec lesquels l'utilisateur peut interagir. Les conditions nécessaires permettant l'action d'un utilisateur sur un widget ont été présentées au chapitre précédent (section 3.1). Un *widget est disponible du point de vue de l'interaction* en un instant donné si et seulement si :

1. le widget est *actif* et *visible* sur l'interface ;
2. tous ses ancêtres dans la hiérarchie de widgets de l'application sont *visibles* ;
3. le widget est connecté à un écouteur d'événements.

Ainsi, l'évolution des disponibilités d'interaction d'une application interactive implémentée en Java-Swing dépend à la fois :

1. de la structure hiérarchique des widgets composant l'interface (vue statique) ;
2. et des réactions de ces widgets qui modifient le contenu des widgets de l'interface. Ces réactions sont encodées par les méthodes d'écouteurs de l'application (vue dynamique).

On désignera par "*modèle structurel*" d'une application interactive un modèle capturant les aspects statiques d'une application, et par "*modèle comportemental*" les aspects dynamiques de l'interface liés à la modification des disponibilités de l'interface au cours de l'interaction (cf. Fig.4.1).

L'association des modèles structurel et comportemental définit un modèle permettant de capturer l'évolution des disponibilités d'interaction au cours de l'interaction homme-machine : il s'agit d'un *modèle de dialogue* tel que décrit dans la section 1.5.

Modèle structurel. Concernant le modèle structurel de l'interface, il ne s'agit pas ici d'extraire un modèle précis du type modèle de classes, mais d'abstraire la structure de l'interface pour ne conserver que les objets, et leurs attributs, nécessaires à l'étude du comportement global de l'interface. Par conséquent les attributs caractérisant l'*apparence* d'un widget (taille, position, couleur, mise en page...) sont absents d'un modèle structurel puisque les aspects ergonomiques liés à la présentation visuelle de l'interface ne participent pas à son comportement.

On appelle modèle structurel un modèle capturant :

- la **structure hiérarchique de widgets** qui compose l'interface de l'application telle que définie au chapitre précédent ;
- le **contenu** de chacun de ces widgets.

Le **contenu** d'un widget correspond aux informations décrivant l'état du widget et ses associations possibles avec des écouteurs d'événements. Ces informations sont encapsulées par les attributs du widget. Deux attributs communs à l'ensemble des widgets sont indispensables pour définir la disponibilité d'un widget :

- l'attribut `visible`, attribut booléen caractérisant la visibilité du widget sur l'interface;
- l'attribut `enabled`, attribut booléen caractérisant l'état d'activation du widget.

D'autres attributs peuvent être indispensables à l'étude suivant le type de widget considéré. Par exemple, l'attribut `text` d'un champ de saisie qui représente son contenu ou bien l'attribut `selected` d'un bouton radio ou d'une `CheckBox` qui caractérise l'état coché ou non du widget.

La section 4.2 de ce chapitre présente la modélisation B événementiel générique de la boite à outils Swing définissant le modèle structurel de l'interface.

Modèle comportemental. On appelle modèle comportemental d'une interface un modèle capturant l'évolution des disponibilités d'interaction avec l'IU au cours du temps. Un modèle comportemental doit prendre en compte l'ensemble des chemins d'interaction réalisables sur l'interface.

La mise en place d'un modèle comportemental repose sur l'étude des réactions du système suite à une action utilisateur. Cette réaction modifie le contenu des widgets composant cette interface et conditionne par conséquent les prochaines actions possibles de l'utilisateur : si un bouton est rendu inactif ou invisible sur l'interface, l'utilisateur ne pourra plus interagir avec ce bouton.

L'aspect comportemental d'une interface Java-Swing est contenu dans la définition des écouteurs d'événements associés aux widgets. Plus précisément, ce sont les méthodes définies par ces écouteurs qui encodent la réaction de l'interface à une action de l'utilisateur. L'analyse de ces méthodes permet la définition du modèle comportemental du système.

Modèle B événementiel. Pratiquement, les modèles structurel et comportemental d'une interface sont représentés au sein d'un seul modèle B événementiel noté B_{Appl}. L'analyse de la méthode `main()` de l'application permet la construction des clauses SETS, PROPERTIES , VARIABLES et INITIALISATION du modèle. L'analyse des méthodes d'écouteurs permet quant à elle de construire la clause EVENTS du modèle B (modèle comportemental du système).

La section 4.2 présente l' abstraction B utilisée pour représenter la boite à outils Java-Swing. Cette abstraction constitue le contexte du modèle B_{Appl}. La modélisation comportementale d'une interface Java-Swing en B fait l'objet de la section 4.3.

Afin de mettre en évidence la faisabilité d'une telle démarche d'extraction de modèles formels vers un langage cible différent du B événementiel, la section 4.4 présente la construction d'un modèle de dialogue en NuSMV. Les principes d'extraction d'un modèle NuSMV étant identiques à ceux utilisés pour l'extraction d'un modèle B événementiel, on présente de manière informelle la construction de ce modèle à partir du modèle B_{Appl}. On désigne par Nu_{Appl} le modèle NuSMV extrait.

Les modèles B_{Appl} et Nu_{Appl} sont directement exploitables pour la validation de propriétés de sûreté du système. La section 4.5 présente quelques résultats obtenus sur l'exemple du convertisseur (étude de cas).

Enfin, la section 4.6 propose une comparaison des deux approches (B et NuSMV) en guise de conclusion.

4.2 Modèle structurel et modélisation de la boîte à outils Java-Swing

Afin de représenter le modèle structurel de l'application analysée, il convient de disposer d'une représentation B événementiel de la boîte à outils Java-Swing. On désigne ce modèle par B_{Swing}. Il constitue le contexte du modèle B_{Appl} final obtenu lors de l'analyse. Ce modèle représente les instances de widgets de l'application, leurs attributs et leurs associations.

Les associations *widgets*/écouteurs lient une réaction de l'interface à un type d'événement particulier généré lors de l'action de l'utilisateur sur un widget. Si ces informations sont indispensables à la construction du modèle lors de l'analyse statique, il n'est pas nécessaire de les représenter directement dans le modèle. Comme on le verra dans la section suivante, cette association est réalisée de manière implicite par la définition de gardes au niveau des événements du modèle B_{Appl}.

Les figures 4.2 et 4.3 présentent une partie du modèle B_{Swing}. Ce modèle représente le modèle structurel extrait par analyse statique de la méthode `main()` de l'application. Le détail de l'analyse statique de la méthode `main()` est présenté au chapitre 5. L'analyse consiste à rechercher dans le corps de la méthode `main()` l'ensemble des créations d'instances de widgets, les valeurs initiales de leurs attributs et les associations widgets/écouteurs d'événements. La représentation des données utilisée dans le modèle B_{Swing} est générique : tous les modèles générés lors d'une analyse utilisent une représentation identique des instances de widgets, de leurs attributs et de leurs associations.

Représentation des instances de widgets

Comme illustré par la figure 4.2, toutes les instances de widgets sont représentées par des constantes (clause CONSTANTS du modèle). Ceci est justifié par l'hypothèse selon laquelle aucune création dynamique de widget n'est autorisée en cours d'exécution de l'application. Ces instances sont ici représentées par l'identifiant du widget préfixé par

`BW_`. Pour l'étude de cas du convertisseur, on retrouve donc : `BW_input`, `BW_ED`, `BW_DE`, etc.

Le type des widgets est représenté par l'intermédiaire des ensembles `JFrame`, `JTextFields`, *JPanel*... Ces ensembles sont définis comme des sous-ensembles de l'ensemble des instances de widgets dénotés par l'ensemble `WIDGETS`. Deux propriétés spécifient que l'ensemble `WIDGETS` est égal à l'union de ces ensembles et un ensemble de propriétés stipulent qu'un widget ne possède qu'un seul type spécifique.

Enfin, toujours sur la figure 4.2, une fonction totale `WIDGETS_parents` permet d'associer à chaque widget appartenant à `WIDGETS`, l'ensemble des ancêtres du widget dans la hiérarchie de widgets définie par l'application. Il s'agit ici d'une représentation "plane" de l'arbre hiérarchique de widgets décrit en section 3.1.3.

La définition de l'ensemble `STRING` permet d'abstraire la notion de chaîne de caractère. Ici une chaîne de caractère est considérée comme vide (`empty`) ou pleine (`full`), c'est-à-dire non vide.

```
1  MODEL BSwing
2
3  SETS
4
5    WIDGETS; STRING={empty,full};
6  /* Eventuellement d'autres abstractions... */
7    [...]
8
9  CONSTANTS
10
11     /* Les instances de widget créées à l'initialisation */
12   BW_pc, BW_gauche, BW_boutonsPanel, BW_droit, BW_inputLabel, BW_inputPanel,
13   BW_input, BW_ED, BW_DE, BW_droit, BW_outputLabel, BW_outputPanel, BW_output,
14     /* Hierarchie de widgets, associations widget/ancêtres */
15   WIDGETS_parents,
16     /* Les différents types de widgets présents dans l'application */
17   JFrame,JTextFields,JButton,JPanel, [...]
18
19 PROPERTIES
20
21     /* Propriétés de typage */
22   BW_pc ∈ WIDGETS ∧ BW_gauche ∈ WIDGETS ∧ BW_boutonsPanel ∈ WIDGETS ∧ [...]
23   JFrame ⊆ WIDGETS ∧ JTextFields ⊆ WIDGETS ∧ JButton ⊆ WIDGETS ∧
24   WIDGETS_parents ∈ WIDGETS → ℙ(WIDGETS) ∧
25     /* Prédicats */
26   JButton={BW_ED,BW_DE} ∧ JFrame={pc} ∧ [...] ∧
27   WIDGETS={BW_input, BW_output, BW_DE, BW_ED,...} ∧
28   WIDGETS=JFrame ∪ JTextFields ∪ JButton ∪ JPanel ∪ [...] ∧
29
30   ∀(w1,w2).(w1 ∈ JFrame ∧ w2 ∈ JTextFields ⇒ w1≠w2) ∧
31   ∀(w1,w2).(w1 ∈ JFrame ∧ w2 ∈ JButton ⇒ w1≠w2) ∧ [...] ∧
32   WIDGETS_parents ={pc ↦ ∅,gauche ↦ {pc},
33                     ED ↦ {boutonsPanel,pc},
34                     input ↦ {inputPanel,gauche,pc},[...]} ∧
35   empty ≠ full ∧
```

FIG. 4.2 – Modèle B_{Swing} (1).

```
1  VARIABLES
2
3     /* Decalarations des attributs de widgets */
4    visible, enabled, Jtext
5
6  INVARIANT
7
8  /* Représentations des attributs de widgets */
9     /* Attributs associés à tous les widgets */
10   visible ∈ WIDGETS → BOOL ∧ enabled ∈ WIDGETS → BOOL ∧
11    /* Attributs spécifiques à certains widgets */ Jtext ∈
12   Jtext → STRING ∧ [...]
13
14 INITIALISATION
15
16   visible := {input ↦ TRUE, ED ↦ TRUE, DE ↦ TRUE,...}   ||
17   enabled := {input ↦ TRUE, ED ↦ FALSE, DE ↦ FALSE,...}   ||
18   Jtext := {input ↦ empty} || [...]
```

FIG. 4.3 – Modèle B_{Swing} (2).

Représentations du contenu des widgets

Une fois définies les instances de widgets et leurs associations, les attributs de chaque widget et leurs valeurs initiales doivent être représentés. Contrairement aux définitions statiques précédentes, les valeurs des attributs de widgets sont amenées à évoluer au cours de l'interaction. De ce fait, les attributs de widgets sont définis dans les clauses VARIABLES et INVARIANT du modèle (Figure 4.3).

Dans un premier temps, les attributs génériques caractérisant l'ensemble des widgets sont définis. Il s'agit des attributs communs à l'ensemble des widgets de la bibliothèque Java-Swing. C'est le cas des attributs `visible` et `enabled` caractérisant l'état de visibilité et d'activité d'un widget. Ces attributs sont représentés par une fonction totale associant à chaque widget appartenant à `WIDGETS` une valeur booléenne. Certains widgets disposent d'attributs spécifiques : c'est le cas par exemple des `JTextFields` possédant un attribut de type texte. Cet attribut est représenté de la même manière que les attributs précédents par une fonction totale `JText` associant à chaque instance de widget de type `JTextField` une valeur prise dans l'ensemble `STRING`. De manière générique, l'attribut *att* d'un widget de type *JTyp* prenant ses valeurs dans *attTyp* sera modélisé par :

$$\texttt{att} \in \texttt{JTyp} \rightarrow \texttt{attTyp}$$

Enfin, la clause INITIALISATION permet l'initialisation des attributs des widgets. Les valeurs initiales de ces attributs sont obtenues par analyse de la méthode `main()`, analyse présentée en détail dans le chapitre 5.

D'autres variables viendront s'ajouter à la clause VARIABLES du modèle. Il s'agira :

- de *variables du contrôleur de dialogue* de l'application nécessaires à la modélisation du système. En effet certaines variables globales du contrôleur de dialogue peuvent être utilisées pour stocker certaines informations relatives à l'état courant de l'interface et gérer le contrôle de l'application ;
- ou bien de *variables de contrôle* permettant d'ordonnancer l'exécution des événements du modèle.

4.3 Modèle comportemental : modélisation des méthodes d'écouteurs d'événements

L'analyse statique des méthodes d'écouteurs d'événements d'une application Java-Swing permet d'établir une représentation abstraite des réactions de l'interface sous la forme d'événements B. Il s'agit ici d'extraire du corps des méthodes d'écouteurs les instructions pertinentes du point de vue de la disponibilité des widgets. Principalement, il s'agit de la modification de la valeur des attributs des instances de widgets de l'application.

Hypothèse de travail

L'analyse des méthodes d'écouteurs repose sur l'hypothèse selon laquelle l'application Java-Swing analysée est *mono-thread* (pas de processus concurrents). L'hypothèse *mono-thread* revient à considérer l'application Java-Swing comme un ensemble de processus séquentiels exécutés en séquence, séquence dépendant des actions de l'utilisateur sur l'interface. Les processus séquentiels correspondent à l'exécution des méthodes d'écouteurs de l'application analysée. Cette considération repose sur les mécanismes de traitement des événements Swing par la machine virtuelle Java (JVM, *Java Virtual Machine*). Concrètement, les événements Swing émis lors d'une action utilisateur sont stockés dans une file de type FIFO (*First In First Out*) par la JVM. Le distributeur d'événements (*event dispatcher*) extrait les événements de la file, dans l'ordre de leur arrivée, et les distribue aux écouteurs associés à la source de l'événement. Le choix de la méthode d'écouteur à exécuter dépend alors de la nature de l'événement émis (`actionEvent`, `keyEvent`,...). Le distributeur d'événements ne distribue aucun nouvel événement tant que l'exécution de la méthode d'écouteur liée à l'événement précédent n'est pas terminée. De ce fait, l'exécution d'une méthode d'écouteur ne peut pas être interrompue par l'exécution d'une autre méthode.

Modélisation des méthodes d'écouteurs

Concrètement, chaque méthode d'écouteur présente dans l'application est modélisée par un ensemble d'événements B. L'ensemble de ces événements définit alors le modèle comportemental de l'application étudiée.

Le corps d'une méthode d'écouteur peut être vu comme un ensemble d'instructions d'affectation modifiant l'état de l'interface. Chacune de ces instructions est traduite en un événement B. Le séquencement de ces événements est réalisé par l'intermédiaire d'une (ou plusieurs) variable(s) de contrôle afin que l'enchaînement des événements reflète l'ordre d'exécution des instructions du code source de la méthode d'écouteur modélisée.

En outre, un deuxième type de variable de contrôle est introduit afin d'assurer que l'enchaînement des événements représentant une méthode d'écouteur M_1 ne soit entrelacé avec l'exécution d'autres événements représentant une méthode d'écouteur M_2.

Formellement, soit une application $\mathcal{A}$ possédant k méthodes d'écouteurs $M_i, i \in 1..k$. Chaque méthode est représentée par un ensemble EVTS_{M_i} d'événements B. Les événements de EVTS_{M_i} sont ordonnancés par l'intermédiaire de variables de contrôle afin de refléter l'ordre d'exécution des instructions de la méthode M_i. Suivant ces notations, seuls

les premiers événements déclenchés $evt_{M_i,1}$ des ensembles EVTS_{M_i}, $i \in 1..k$, peuvent entrer en concurrence. L'événement déclenché parmi $\bigcup_{i \in 1..k} evt_{M_i,1}$ est alors choisi suivant la garde de ces événements. Si plusieurs de ces événements possèdent leur garde vraie au même instant alors le choix de l'événement déclenché est indéterministe tel que défini par la méthode B événementiel. Si l'événement $evt_{M_p,1}$ est déclenché, alors les prochains événements déclenchés sont les événements de $\text{EVTS}_{M_p} - \{evt_{M_p,1}\}$. Tant que tous les événements appartenant à $\text{EVTS}_{M_p} - \{evt_{M_p,1}\}$ n'ont pas été déclenchés, aucun événement evt tel que $evt \in \bigcup_{i \in 1..k, i \neq p} \text{EVTS}_{M_i}$ ne pourra prendre la main. Ceci est assuré par un ensemble de variables de contrôle permettant de garder l'exécution des différents événements modélisant l'application.

Étapes du processus de modélisation

Le processus de formalisation d'une méthode d'écouteur d'événements peut être décomposé en cinq étapes.

Conditions d'activation. Le premier événement modélisant l'exécution d'une méthode d'écouteur doit satisfaire aux conditions d'activation de la méthode d'écouteur représentée. Ces conditions d'activation, présentées en section 4.3.1, sont formalisées dans la garde de l'événement.

Traitement des instructions complexes. Le code source d'une méthode d'écouteur est rarement constitué d'une simple séquence d'affectations mais plus généralement d'un ensemble de blocs d'instructions complexes (affectation, appel de méthode, déclaration de variables locales). Ces instructions complexes sont ordonnés par l'intermédiaire de structures de contrôle : séquence, conditionnelle ou structure itérative. Le traitement de ces constructions nécessite une étape préalable consistant à remplacer une instruction de type appel de méthode par le corps de la méthode appelée. Cette opération d'*inlining* effectuée de manière itérative permet de déplier le corps de la méthode d'écouteur afin d'obtenir uniquement des instructions élémentaires.

Abstraction. D'un point de vue pratique, certaines de ces instructions élémentaires présentes dans le corps de la méthode analysée ne sont pas pertinentes du point de vue de l'interaction. On peut distinguer deux grandes classes d'instructions sans impact sur l'évolution de la disponibilité des widgets :

1. les instructions faisant appel au noyau fonctionnel de l'application ;
2. les instructions modifiant l'apparence des widgets, et plus généralement modifiant la valeur de tout attribut non pertinent du point de vue de l'interaction : modification de la couleur, de la position du widget, etc.

Ces instructions doivent être abstraites avant toute traduction d'une méthode d'écouteur en événements B.

Règles de traduction. Une fois ces opérations d'*inlining* et d'abstraction effectuées, la traduction explicite des instructions de la méthode d'écouteur en événements B est réalisée. Cette traduction est établie suivant un ensemble de règles de traduction permettant la représentation des structures de contrôle : séquence, conditionnelle et boucle.

Réduction du nombre d'événements B. Le choix ad hoc consistant à traduire chaque instruction d'affectation par un événement B conduit à un nombre très important d'événements. Une solution pour réduire ce nombre d'événements consiste à regrouper un ensemble d'instructions séquentielles au sein d'un même événement B. Ces instructions séquentielles sont groupées dans le corps de l'événement en utilisant l'instruction simultanée || du langage B.

Ce regroupement d'instructions séquentielles nécessite une analyse de dépendance des données. Dans le cas où les instructions sont indépendantes du point de vue des données qu'elles manipulent, alors celles-ci peuvent être exécutées dans un ordre indifférent. Dans le cas contraire, l'utilisation de la règle de traduction d'instructions séquentielles doit être utilisée.

Les différentes étapes abordées ci-dessus sont détaillées dans les sous-sections suivantes. Chaque étape sera illustrée par un exemple issu de l'étude de cas.

4.3.1 Formalisation des conditions d'activation

Quel que soit le nombre d'événements nécessaires à la modélisation d'une méthode d'écouteur, le premier événement B *déclenché* (i.e. exécuté) doit satisfaire aux conditions d'activation de la méthode d'écouteur qu'il représente. Ces conditions sont exprimées dans la garde de l'événement. La formalisation de ces différentes conditions est donnée par le tableau 4.1. Dans ce tableau `wid` représente le widget à la *source* de l'événement déclenchant la méthode de l'écouteur.

Garde 1 :	`enabled(wid)=TRUE`
Garde 2 :	`visible(wid)=TRUE`
Garde 3 :	$\forall$`w.(w`$\in$`WIDGETS`$\wedge$`w`$\in$`WIDGETS_parents(wid)`$\Rightarrow$ `visible(w)=TRUE)`

TAB. 4.1 – Formalisation des conditions d'activation.

Les gardes 1 et 2 expriment que le widget source doit être actif et visible. En effet, l'utilisateur ne peut pas interagir avec le widget si celui-ci n'apparaît pas sur l'interface ou si celui-ci est inactif.

La garde 3 exprime que l'ensemble des ancêtres du widget source dans la hiérarchie de widgets définie par l'application doivent être visibles. Si l'un d'entre-eux est invisible sur l'interface, ses fils sont également invisibles : l'utilisateur est alors dans l'impossibilité d'interagir avec le widget et par conséquent la méthode d'écouteur ne se déclenche pas.

4.3.2 Inlining et abstraction

La traduction d'une méthode d'écouteur en un ensemble d'événements B nécessite une étape préalable de dépliage des instructions de type "appel de méthode" et l'abstraction des instructions non pertinentes.

Inlining des méthodes d'écouteurs

La traduction d'une instruction d'affectation Java en B événementiel est simple : elle consiste simplement à reproduire cette affectation au sein d'un événement B.

Cependant, le corps d'une méthode d'écouteur Java est rarement constitué de simples affectations. On y trouve en particulier des instructions du type "appel de méthode". Il est nécessaire d'analyser le corps des méthodes appelées afin de traiter ce type d'instruction. Pour ce faire, l'instruction d'appel est remplacée par le corps de la méthode appelée. Cette opération d'*inlining*, ou de "dépliage", est effectuée récursivement jusqu'à l'obtention d'un ensemble d'instructions qu'il n'est plus possible de déplier. Il s'agit alors soit d'une affectation simple, soit d'une instruction de type "appel de méthode" dont le corps n'est pas disponible lors de l'analyse.

Cette opération d'inlining n'est pas explicitement réalisée en pratique : le parcours du graphe de dépendance système de l'application (JSysDG) réalise cette opération de manière implicite.

Abstractions

Parmi l'ensemble des instructions qui composent la méthode d'écouteur, certaines instructions n'affectent pas le comportement de l'application du point de vue de l'interaction. Ces instructions sont par conséquent complètement abstraites, c'est-à-dire non traduites au sein des événements B modélisant la méthode d'écouteur. Cette abstraction est le résultat d'une opération de *slicing*[1] réalisée sur le code source de l'application.

L'interface d'une application interactive fait généralement appel au Noyau Fonctionnel (NF) de l'application qui encapsule les fonctionnalités brutes du système. Dans l'exemple de l'étude de cas, ce NF regroupe les méthodes de calcul d'une conversion. Les méthodes appartenant au NF n'interviennent pas du point de vue de l'interaction. Ces méthodes sont par conséquent abstraites. Dans la plupart des cas, cette abstraction consiste à supprimer l'instruction réalisant un appel au Noyau Fonctionnel (NF).

Dans le cas où l'appel au NF retourne une valeur utilisée par l'interface, l'appel de la méthode est alors abstrait en remplaçant l'instruction d'appel par une valeur constante dont le type est identique au paramètre de retour de la méthode appelée. De manière informelle, ceci revient à considérer, au niveau de la modélisation, qu'un appel au noyau fonctionnel a bien été effectué et une valeur retournée.

Cette étape d'abstraction du Noyau Fonctionnel est étroitement liée au modèle d'architecture utilisé lors de la conception de l'application. Notamment, ces opérations d'abstraction sont simplifiées lorsque l'architecture du système repose sur la séparation du NF et de la partie purement interactive du système.

[1] cf. Chapitre 1, section 2.3

Presque tous les modèles d'architecture dédiés IHM se fondent sur cette séparation. La donnée du modèle d'architecture utilisé permet d'optimiser l'analyse statique en facilitant la recherche des instructions d'appel des méthodes du noyau fonctionnel.

Application à l'étude de cas

```
1 public void
2   actionPerformed(ActionEvent e){
3  if (e.getSource()==ED)
4  {controleur.convertirED(true);}
5  if (e.getSource()==DE)
6  {controleur.convertirED(false);}
7 }
```

(A.1) Méthode d'écouteur `actionPerformed`.

```
1
2 public void keyTyped(KeyEvent e){
3   controleur.gestionInput();
4 }
```

(B.1) Méthode d'écouteur `keyTyped`.

```
1 public void
2   actionPerformed(ActionEvent e)
3 {if (e.getSource()==ED)
4  {anfc.setED(true);
5   anfc.setConvert(input.getText());
6   output.setText(anfc.getConvert());
7   ED.setEnabled(false);
8   DE.setEnabled(true);
9  }
10  if (e.getSource()==DE)
11  {anfc.setED(false);
12   anfc.setConvert(anfc.getConvert());
13   output.setText();
14   ED.setEnabled(false);
15   DE.setEnabled(true);
16  }
17  output.setColor(red);
18 }
```

(A.2) Inlining : méthode `actionPerformed`.

```
1 public void keyTyped(KeyEvent e){
2  if(e.getSource()==input){
3   output.setText("");
4   output.setColor(null);
5    if (input.getText()=="")
6    { ED.setEnabled(false);
7      DE.setEnabled(false);}
8    else
9    { ED.setEnabled(true);
10     DE.setEnabled(true);}
11  }
12 }
```

(B.2) Inlining : méthode `keyPressed`.

FIG. 4.4 – Méthodes d'écouteurs présentes dans l'étude de cas.

Inlining. L'étude de cas du convertisseur est constituée de deux méthodes d'écouteurs : `actionPerformed` (Fig.4.4, A.1) et `keyPressed` (Fig.4.4, B.1). La première étape consiste à remplacer les instructions de type "appel de méthodes" par le corps des méthodes appelées (inlining). Le résultat obtenu est présenté par les sous figures A.2 et B.2. Le corps des méthodes d'écouteurs est alors constitué d'un ensemble d'instructions "élémentaires" ordonnancées par des structures de contrôle : séquences et conditionnelles dans le cas présent. Les instructions élémentaires correspondent soit à des modifications de valeurs des attributs de widgets soit à des appels au noyau fonctionnel de l'application via l'adaptateur au noyau fonctionnel (*anf*).

Abstraction. La seconde étape réalisée avant la traduction des méthodes en événements B consiste à abstraire les instructions faisant appel au noyau fonctionnel. Dans le cas de la méthode `actionPerformed` (Fig.4.5, A), deux appels de méthodes au noyau fonctionnel (lignes 4 et 5) et une instruction non pertinente (ligne 17) sont supprimés.

```
1  public void
2    actionPerformed(ActionEvent e)
3  {if (e.getSource()==ED)
4   { anfc.setED(true);
5    anfc.setConvert(input.getText());
6     output.setText(full);
7     ED.setEnabled(false);
8     DE.setEnabled(true);
9   }
10  if (e.getSource()==DE)
11  {anfc.setED(false);
12   anfc.setConvert(input.getText());
13    output.setText(full);
14    ED.setEnabled(false);
15    DE.setEnabled(true);
16  }
17  output.setColor(red);
18 }
```

(A) Abstraction : méthode `actionPerformed`.

```
1  public void keyTyped(KeyEvent e){
2    output.setText("");
3   output.setColor(null);
4     if (input.text == empty)
5     { ED.setEnabled(false);
6       DE.setEnabled(false);}
7     else
8     { ED.setEnabled(true);
9       DE.setEnabled(true);}
10  }
11 }
```

(B) Abstraction : méthode `keyTyped`.

FIG. 4.5 – Abstraction des méthodes d'écouteurs.

On remarque que le troisième appel à l'adaptateur du NF (ligne 6, Fig. 4.5(A)) correspond à un paramètre d'appel de la méthode `setText` permettant de modifier la valeur du champ de texte *output* (affichage du résultat de la conversion). La valeur de ce résultat n'est pas significative du point de vue de l'aspect comportemental de l'interaction puisqu'elle ne modifie pas l'état de disponibilité des widgets composant l'interface. En contrepartie, on souhaite conserver l'information d'un affichage du résultat de la conversion. Cet appel de méthode est alors abstrait par une valeur du type du paramètre de retour de la méthode appelée. Par défaut, la valeur de retour de cet appel est fixée à `full` : cette abstraction consiste à considérer que la conversion explicite effectuée par le noyau fonctionnel renvoie une valeur non vide.

4.3.3 Règles de traduction des structures de contrôle

Une fois les opérations d'inlining et d'abstraction réalisées, le code source de la méthode est constitué d'un ensemble de blocs d'instructions ordonnancés par les structures de contrôle du langage : séquence, conditionnelle, boucle. Afin d'effectuer la traduction de la méthode en un ensemble d'événements B de manière automatique, il est nécessaire de disposer de règles de traduction permettant d'exprimer ces différentes structures de contrôle. Les règles de traduction exposées par les tableaux 4.2, 4.3 et 4.4 sont construites suivant les règles définies par J.R. Abrial. Dans [Abrial, 2003a], J.R. Abrial propose des règles de fusion d'événements dans un contexte de développement de programmes séquentiels en B événementiel. Ces règles permettent de fusionner certains événements en un événement concret faisant intervenir des structures de contrôle telles que la conditionnelle ou la boucle.

Ces règles sont utilisées dans le sens inverse afin d'éclater une instruction de contionnelle ou d'itération en un ensemble d'événements.

Séquence d'instructions. Le tableau 4.2 présente la règle de traduction associée à un bloc de `p` instructions en séquence. Une séquence de `p` instructions est traduite par

p événements. L'exécution en séquence de ces événements est assurée par l'utilisation d'une variable de contrôle V dont la valeur initiale est fixée à p-1. Chaque événement fait décroître cette variable de contrôle, donnant ainsi le contrôle à l'événement suivant. Le dernier événement Evt_p réinitialise la variable de contrôle à p-1 : la séquence d'instructions peut alors être exécutée une nouvelle fois si la garde de contexte associée au premier événement est vraie.

Dans le cas d'une séquence de blocs d'instructions, la traduction est réalisée de manière itérative en introduisant de nouvelles variables de contrôle pour chaque sous-bloc.

$I_1;I_2;\ldots;I_p;$ $\rightsquigarrow$	Evt_1= **SELECT** $G_{C1} \wedge$ V=p-1 **THEN** V:=V-1 $\parallel$ I_1 **END**; Evt_i= **SELECT** V=(p-i) **THEN** V:=V-1 $\parallel$ I_i **END**;	Evt_2= **SELECT** V=(p-2) **THEN** V:=V-1 $\parallel$ I_2 **END**; Evt_p= **SELECT** $G_{Cp} \wedge$ V=0 **THEN** V:=p-1 $\parallel$ I_p **END**;

TAB. 4.2 – Traduction d'une structure de contrôle : séquence.

Conditionnelle. La règle de traduction d'une structure de contrôle conditionnelle est présentée dans le tableau 4.3. L'instruction if(E) then {I_1;} else {I_2;} est traduite par deux événements. Ces deux événements possèdent une garde commune notée G_C, dépendant du contexte dans lequel la conditionnelle intervient. Le premier événement est déclenché lorsque la condition E est réalisée, le second lorsque la condition $\neg$(E) est vraie. Le corps des deux événements est alors constitué des instructions I_1 et I_2 respectivement et d'une éventuelle instruction Op(G_C) permettant une mise à jour du contexte d'exécution (incrémentation ou décrémentation d'autres variables de contrôle par exemple). Dans le cas ou l'instruction I_1 (ou I_2) est un bloc d'instructions, les règles de regroupement d'instructions et de traduction sont alors appliquées de manière itérative jusqu'à la fin du traitement.

if(E) then {I_1;} else {I_2;} $\rightsquigarrow$	Evt_1= **SELECT** $G_C \wedge$ E **THEN** $I_1 \parallel$ Op(G_C) **END**; Evt_2= **SELECT** $G_C \wedge \neg$(E) **THEN** $I_2 \parallel$ Op(G_C) **END**;

TAB. 4.3 – Traduction d'une structure de contrôle : conditionnelle.

Boucles itératives. La règle de traduction appliquée dans le cas d'une structure de contrôle de type "boucle" est présentée dans le tableau 4.4. La traduction est similaire du

point de vue de la forme à celle utilisée pour la conditionnelle avec cependant une différence qui n'apparaît pas explicitement : l'introduction du premier événement doit assurer l'existence d'un variant. L'existence d'un tel variant permet la preuve de terminaison de la boucle.

Ceci nécessite d'un point de vue pratique l'introduction du premier événement dans un précédent raffinement.

Cependant, notons qu'une telle construction est peu probable dans le cadre de l'étude : la programmation de la partie interactive d'un système fait rarement appel à une construction de type "boucle".

`while(E) {I1;} I2;` $\rightsquigarrow$	`Evt_1=`
	SELECT $G_C \wedge$ `E`
	THEN `I1` $\|\|$ `Op(`G_C`)`
	END;
	
	`Evt_2=`
	SELECT `G'`$_C \wedge \neg$`(E)`
	THEN `I2` $\|\|$ `Op(G'`$_C$`)`
	END;

TAB. 4.4 – Traduction d'une structure de contrôle : boucle while.

4.3.4 Réduction du nombre d'événements : parallélisation d'instructions séquentielles

La traduction ad hoc consistant à traduire un bloc de p instructions séquentielles en p événements B ordonnancés par des variables de contrôle conduit à un nombre très important d'événements. Afin de réduire ce nombre d'événements, il est possible, dans la majorité des cas, de regrouper ces instructions au sein d'un même événement B en utilisant l'instruction simultanée du langage B ($\|$).

Analyse de dépendances de données.

Le regroupement d'instructions Java au sein d'un événement B nécessite une analyse de dépendances des instructions (dépendances de données).

Considérons une séquence $\mathcal{S}$ de p instructions telle que $\mathcal{S} \hat{=} I_1; I_2; \ldots; I_p$; avec I_i, $i \in 1..p$ de la forme :

$$Var_i^0 = Exp_i(Var_i^1, \ldots, Var_i^{Ni}),\ Exp_i \text{ étant une expression quelconque.}$$

Chaque instruction possède donc $Ni - 1$ variables. Notons $\mathcal{VI}$ l'ensemble des variables d'intérêt de l'application. Les variables d'intérêt sont dans le cadre de l'étude :

1. soit des *variables de type widget* ou les attributs de ces widgets ;
2. soit des *variables du contrôleur de dialogue* pertinentes du point de vue de la gestion du dialogue homme-machine.

Enfin, soient W et R les fonctions associant à une instruction I l'ensemble des variables modifiées (*Write-set*) et l'ensemble des variables lues (*Read-set*) par l'instruction I. Par définition :

$$W(I_i) = Var_i^0 \text{ et } R(I_i) = \bigcup_{j \in 1..N_i} Var_i^j$$

Par extensions de notations :

$$W(\mathcal{S}) = \bigcup_{k \in 1..p} W(I_k) \text{ et } R(\mathcal{S}) = \bigcup_{k \in 1..p} R(I_k)$$

Dépendance de données. Il existe une dépendance de données entre deux instructions I_i et I_j de $\mathcal{S}$ si : $W(I_i) \cap R(I_j) \neq \varnothing$.

1er cas : Aucune dépendance de données. Dans le cas où les p instructions d'une séquence $\mathcal{S}$ sont indépendantes du point de vue des données, ces instructions peuvent être regroupées dans le corps d'un événement B déterministe tel que représenté par le tableau 4.5.

Evt=
SELECT G_C
THEN
$I_1 \parallel \ldots \parallel I_p$
END;

TAB. 4.5 – Traduction d'une séquence d'instructions non dépendantes.

2nd cas : Existence de dépendances. On suppose que seules deux instructions I_i et I_j de $\mathcal{S}$, avec $i < j$, sont dépendantes :

$$I_i \hat{=} (VarDep = Exp_i(Var_i^1, \ldots, Var_i^{Ni}) \text{ et}$$
$$I_j \hat{=} (Var_j^0 = Exp_j(Var_j^1, \ldots, VarDep, \ldots, Var_j^{Nj}))$$

Il est alors possible de paralléliser les instructions de $\mathcal{S}$ au sein d'un même événement B de type indéterministe `ANY`. L'opération consiste à introduire une variable locale x au sein de l'événement. La garde de l'événement définit la variable x par son type et sa valeur courante, en l'occurrence : $x = Exp_i(Var_i^1, \ldots, Var_i^{Ni})$. Les instructions I_i et I_j sont alors regroupées dans le corps de l'événement en remplaçant l'expression de Exp_i par x dans I_i et en remplaçant la variable $VarDep$ apparaissant dans Exp_j par x. Le tableau 4.6 présente la traduction obtenue :

Evt=
ANY x
WHERE $G_C \wedge$
$x \in Type(VarDep) \wedge x = Exp_i(Var_i^1, \ldots, Var_i^{Ni})$
THEN
$I_1 \parallel \ldots \parallel$
$VarDep := x \parallel \ldots \parallel$
$Var_i^0 := Exp_j(Var_j^1, \ldots, x, \ldots, Var_j^{Nj}) \parallel \ldots \parallel$
I_p
END;

TAB. 4.6 – Traduction d'une séquence d'instructions avec dépendance de données.

Dans le cas d'une dépendance multiple, cette transformation reste applicable en introduisant pour chaque dépendance une nouvelle variable locale au sein de l'événement B.

Cas particulier : Traitement des variables non pertinentes. En se référant aux notations introduites, toute variable v considérée comme non pertinente appartient à l'ensemble $\mathcal{VNP} = (W(S) \cup R(S)) - \mathcal{VI}$. Étant donné que les instructions liées au noyau fonctionnel et que les instructions liées à l'apparence des widgets ont été abstraites dans les précédentes étapes de l'analyse, v correspond à une variable locale utilisée dans le corps de la méthode d'écouteur. Afin de réduire l'espace des variables dans la représentation B événementiel, on ne souhaite pas représenter les variables locales au sein de notre système.

Ce cas nécessite un traitement particulier :

1. soit la variable locale v appartenant à $\mathcal{VNP}$ est modifiée, et modifiée seulement, dans l'une des instructions I de $\mathcal{S}$: $v \in W(\mathcal{S})$ et $v \notin R(\mathcal{S})$. Dans ce cas, l'instruction peut être abstraite en supprimant tout simplement celle-ci au sein de la séquence $\mathcal{S}$[2] ;
2. ou bien la variable locale v appartenant à $\mathcal{VNP}$ est modifiée dans une instruction I_i de $\mathcal{S}$ et lue une ou plusieurs fois dans des instructions I_j de $\mathcal{S}$ telle que $j > i$. Formellement : $\exists i, j \in 1..p$ tels que $v \in W(I_i) \wedge v \in R(I_j) \wedge i < j$. Dans ce cas, cette instruction participe à l'évolution des variables pertinentes du système et il existe une *dépendance de données* entre les instructions I_i et I_j. La transformation de la séquence est alors identique à celle présentée par le tableau 4.6 à la différence près que l'instruction I_i est supprimée du corps de l'événement B.

Autres cas. Dans tout autre cas non traité dans les paragraphes précédents, la règle de traduction pour les séquences d'instructions est utilisée. La séquence $\mathcal{S}$ est alors divisée en blocs d'instructions B_i, où toutes les instructions appartenant à B_i sont parallélisables. Chaque bloc est alors représenté par un événement B et une variable de contrôle permettant de représenter la séquentialité de ces blocs est introduite.

[2]L'instruction I est potentiellement un code mort.

4.3.5 Exemple : étude de cas.

Les figures 4.6 et 4.7 présentent la modélisation B événementiel des méthodes d'écouteur `actionPerformed` et `keyPressed` dont l'inlining et l'abstraction ont été présentés dans la figure 4.5.

La méthode `actionPerformed` est modélisée à l'aide de deux événements B permettant de traduire la structure de contrôle conditionnelle. La condition imposée par la structure conditionnelle est ici particulière puisqu'elle fait intervenir la source de l'événement : `e.getSource()`. L'analyse statique permet la reconnaissance de cette instruction particulière en utilisant des reconnaissances de patterns.

Une représentation des associations widgets/écouteurs est construite au cours de l'analyse. Dans cet exemple, les deux widgets associées à l'écouteur d'action sont `ED` et `DE`. La première instruction de la conditionnelle est ainsi exécutée lorsque la source de l'événement émis est `ED`, et la seconde, selon les règles de traduction définies précédemment, lorsque `not(e.getSource()=ED)` est évalué à vrai. On en déduit alors que le deuxième bloc de la conditionnelle est exécuté lorsque la source de l'événement est `DE`.

Les gardes des deux événements `actionP_ED` et `actionP_DE` sont par conséquent constituées des conditions d'activation des widgets `ED` et `DE`. Les corps de ces événements sont construits en analysant les deux blocs d'instructions qui composent la conditionnelle. Ces blocs contiennent un ensemble d'instructions en séquence. Ces instructions sont indépendantes et peuvent être regroupées au sein des événements en utilisant l'instruction simultanée du langage B.

```
1  actionP_ED=
2    SELECT
3      /* Conditions d'activation */
4      enabled(ED)=true ∧
5      visible(ED)=true ∧
6      ∀ w.(w ∈ WIDGETS ∧ w ∈ WIDGETS_parents(ED) ⇒ visible(ED)=TRUE) ∧
7      Vkp ≠ 0 /* Garde de contexte */
8    THEN
9      Jtext:=Jtext⩤{output ↦ full} ||
10     enabled:=enabled⩤{ED ↦ false, DE ↦ true}
11   END
12
13 actionP_DE=
14   SELECT
15     /* Conditions d'activation */
16     enabled(DE)=true ∧
17     visible(DE)=true ∧
18     ∀w.(w ∈ WIDGETS ∧ w ∈ WIDGETS_parents(DE) ⇒ visible(DE)=TRUE) ∧
19     Vkp ≠ 0 /* Garde de contexte */
20   THEN
21     Jtext:=Jtext⩤{output ↦ full} ||
22     enabled:=enabled⩤{DE ↦ false, ED ↦ true}
23   END
```

FIG. 4.6 – Modélisation B événementiel de la méthode `actionPerformed`.

Dans le cas de la méthode `keyPressed`, seul le widget `input` est associé à l'écouteur d'événements `KeyListener`. La garde du premier événement représentant la méthode (`keyPressed_0`) possède donc les conditions d'activation de la méthode.

Lors d'une modification du champ de texte `input`, et plus généralement de tout champ de texte, la valeur du champ de texte est modifiée puis une notification de ce changement

est émise par la JVM sous la forme de l'événement `keyPressed`. Les instructions présentes dans le corps de la méthode `keyPressed` peuvent dépendre de la nouvelle valeur du champ de texte. Afin de représenter cette modification de la valeur du champ de texte, le premier événement `keyPressed_0` est représenté par une substitution de type indéterministe. Le corps de cet événement modifie la valeur du champ de texte `input` en lui assignant de manière indéterministe la valeur `full` ou `empty` afin de refléter un choix de l'utilisateur. Une variable de contrôle $\mathtt{V}_{kp}$ est introduite. La garde du premier événement exprime que $\mathtt{V}_{kp}$ doit être égale à `2` pour que l'événement soit déclenché. Le corps de l'événement fait alors décroître cette variable de contrôle afin de donner la main aux événements représentant le corps de la méthode `keyPressed`.

```
keyPressed_0=
  ANY xx WHERE
    xx ∈ TEXT ∧
    /* Conditions d'activation */
    enabled(input)=true ∧
    visible(input)=true ∧
    ∀w.(w ∈ WIDGETS ∧ w ∈ WIDGETS_parents(input) ⇒ visible(input)=TRUE) ∧
    /* Variable de contrôle */
    Vkp=2 ∧
  THEN
    Jtext(input) := xx || Vkp := 1
  END


keyPressed_1=
  SELECT
    /* Conditions d'activation */
    enabled(input)=true ∧
    visible(input)=true ∧
    ∀w.(w ∈ WIDGETS ∧ w ∈ WIDGETS_parents(input) ⇒ visible(input)=TRUE) ∧
    Jtext(input)=empty ∧ Vkp=1
  THEN
    enabled:=enabled ⩤ {DE ↦ false,DE ↦ false} ||
    Vkp := 0
  END

keyPressed_2=
  SELECT
    /* Conditions d'activation */
    enabled(input)=true ∧
    visible(input)=true ∧
    ∀w.(w ∈ WIDGETS ∧ w ∈ WIDGETS_parents(input) ⇒ visible(input)=TRUE) ∧
    Jtext(input)= ¬(empty) ∧ Vkp=1
  THEN
    enabled:=enabled ⩤ {DE ↦ true,DE ↦ true} ||
    Vkp := 0
  END
```

Fig. 4.7 – Modélisation B événementiel de la méthode `keyPressed`.

Le corps de la méthode `keyPressed` est constitué d'une structure conditionnelle et est donc représenté par deux événements B `keyPressed_1` et `keyPressed_2` comme établi par les règles de traduction. Le premier de ces événements est déclenché lorsque la valeur du champ de saisie `input` est égale à `empty` et que la variable de contrôle $\mathtt{V}_{kp}$ vaut `1`, le second lorsque la valeur du champ de saisie vaut `¬(empty)` et que la variable de contrôle vaut également `1`.

Dans les deux événements `keyPressed_1` et `keyPressed_2` les instructions composant le corps des blocs d'instructions de la conditionnelle sont indépendantes du point de vue des données : les instructions peuvent alors être regroupées dans le corps de l'événement. Par exemple, les deux instructions de `keyPressed_1` modifient l'attribut *enabled* des boutons `ED` et `DE` en lui assignant la valeur `false`. Le regroupement consiste à changer la valeur de ces attributs au sein d'une seule instruction modifiant la valeur de la variable B `enabled` représentée par une fonction : l'opérateur $\lhd\!\!+$ du langage B représente en effet la surcharge d'une relation (cf. section 3.2.3). On retrouve dans la traduction des instructions les variables définies dans la modélisation de la boite à outils Swing B_{Swing} présentée dans la section précédente : `WIDGETS_parents`, `visible`, `enabled`, `JText`...

L'association du contexte de l'application (modèle structurel B_{Swing}) et du modèle comportemental définit le modèle formel B_{Appl}. Ce modèle formalise le dialogue de l'application interactive en suivant l'évolution des disponibilités d'interaction de l'application convertisseur.

Deux événements `open` et `close`, non présentés ici, sont ajoutés à la clause EVENTS. Ces événements représentent respectivement le lancement et la fermeture de l'application. La garde de l'événement `open` exprime que le widget à la base de la hiérarchie de widgets de l'application est inactif et invisible. Dans l'exemple de l'étude de cas, ce widget est le conteneur `pc` de type `JFrame`, et la garde de l'événement est alors : `enabled(pc)=false` $\wedge$ `visible(pc)=false`. Le corps de l'événement `open` est identique à celui de l'événement d'INITIALISATION.

L'événement `close` est gardé par les conditions d'activation de `pc`. Le corps de cet événement désactive et rend invisible l'ensemble des widgets de l'application.

Modèle B_{Appl}. Le modèle complet de l'application analysée est constitué du modèle B_Swing présenté dans la section précédente et des événements B modélisant l'exécution des méthodes d'écouteurs. L'invariant du modèle B_{Appl} extrait est constitué (clause INVARIANT du modèle) de l'invariant de typage du dans le modèle B_Swing. Une assertion est également ajoutée afin d'assurer le non-blocage de l'application (disjonction des gardes des événements du modèle).

Le modèle de dialogue B_{Appl} de l'application est exploitable pour prouver des propriétés de sûreté de l'application (section 4.5). Ces propriétés sont à ajouter à la clause ASSERTION du modèle.

4.4 Construction d'un modèle NuSMV

La section précédente a montré comment modéliser une application Java-Swing dans le langage B événementiel. Ce modèle peut être exploité afin de prouver des propriétés sur le système par démonstration de théorème (*theorem-proving*). Bien entendu, il est possible d'extraire un modèle de dialogue d'une application Java/Swing en utilisant d'autres langages de modélisation et d'autres techniques de preuve. Notamment, les langages basés sur les systèmes de transitions sont adaptés à la représentation de la dynamique de l'interaction homme-machine. Afin de mettre en évidence cette possibilité, et de comparer les atouts et inconvénients de deux techniques formelles, l'une s'appuyant sur la démons-

tration de théorème et l'autre sur la vérification exhaustive de modèles, on propose dans cette section une modélisation formelle d'une application Java-Swing en NuSMV.

Les étapes de modélisation en NuSMV d'un système interactif permettant de capturer les aspects comportementaux du système sont identiques à celles présentées pour la modélisation B événementiel B_{Appl}. L'analyse de la méthode `main()` de l'application Java-Swing permet de construire l'initialisation du modèle et l'analyse des méthodes listeners, préalablement dépliées et abstraites, et permet la construction du système de transitions. Par conséquent, la construction du modèle NuSMV est exposée en référence au modèle B_{Appl} précédemment décrit.

4.4.1 Abstraction de la boite à outils Java-Swing

La première étape consiste à se doter d'une représentation de la boîte à outils Java-Swing constituant le modèle structurel de l'application. La boite à outils Java-Swing est représentée par un ensemble de modules NuSMV. Ces modules définissent les différents types de widgets présents dans l'application et possèdent uniquement une clause VAR. À titre d'exemple, les modules définis en figure 4.8 représentent les objets de type `widget` et sa spécialisation `JTextField`. Contrairement à une représentation B, il n'est pas possible de définir les relations entre un `widget` et un `JTextfield` par le biais d'inclusions d'ensembles : chaque type spécialisant le type widget doit donc déclarer à nouveau les attributs communs à l'ensemble des widgets, notamment les attributs `enabled` et `visible`.

```
1  /* Exemple : Abstraction d'un widget*/
2  Module widget
3   VAR
4     visible, enabled : boolean;
5
6  /* Exemple : Abstraction d'un JTextField */
7  Module jtextfield
8   VAR
9     visible, enabled, : boolean;
10    text : boolean; /* 0≡empty et 1≡full */
```

FIG. 4.8 – Abstraction de la boîte à outils Swing : modules NuSMV.

4.4.2 Module principal

Le module principal de la modélisation (figures 4.9 et 4.10) déclare l'ensemble des widgets qui composent l'application ainsi que les variables de contrôle (ici $\mathtt{V}_{kp}$). Deux nouvelles variables `ua` (pour *user action*) et `preua` sont également introduites. `ua` représente au sein du modèle le prochain événement réalisé et `preua` (pour *previous user action*) la valeur précédente de `ua`, c'est-à-dire l'événement en cours d'exécution. La variable `ua` prend ses valeurs dans l'ensemble des événements de l'application : ils correspondent à l'ensemble des événements B définis dans le modèle B_{Appl} auxquel on ajoute les actions `nil` et `LM_action`. Les notations ont été modifiées par souci de concision : `keyPressed_i` devient `kpi` et `actionP_XX` devient `apXX`.

Un état pour lequel la variable `ua` prend la valeur `LM_action` correspond à l'état résultant d'un événement B (post-condition d'un événement). La necessité de l'introduction

d'une telle variable est explicitée par la suite (construction du système de transitions de la variable `ua`).

`nil` est une action spécifique du modèle NuSMV permettant de représenter un blocage de l'application : si aucun événement n'est réalisable dans l'instant courant, alors l'événement réalisant une action nulle `nil` sera tiré infiniment souvent. Le système reste alors dans le même état correspondant à un état de blocage.

Dans la clause DEFINE, un ensemble de variables internes (raccourcis d'écriture) est défini. Ces variables représentent les conditions d'activation de chaque événement : on y retrouve l'ensemble des expressions définissant les gardes des événements du modèle B_{Appl}. Du fait du manque d'expressivité du langage NuSMV (absence de la théorie des ensembles et de la logique du premier ordre), la condition traduisant le fait qu'un widget est disponible si l'ensemble de ses parents dans la hiérarchie de widgets sont visibles est traduite de manière explicite en énumérant cet ensemble.

```
1  Module main
2    VAR
3    /* Déclarations des widgets de l'application */
4     ED,DE,pc : widget;
5     input,output : jtextfield;
6     [...]
7    /* Variable de contrôle */
8     Vkp : {0,1}; /* Assure l'enchainement de l'événement
9                     kp0 avec l'un des deux évenements kp1 ou kp2 */
10   /* Déclaration de l'ensemble des événements */
11    ua : {kp1, kp2, aP_ED, aP_DE, kp0, open, close,nil,LM_action}
12    preua : {kp1, kp2, aP_ED, aP_DE, kp0, open, close,nil,LM_action}
13   DEFINE
14    G_open:=(pc.enabled=0) ∧(pc.visible=0)
15                 ∧ (DE.enabled=0 ∧ DE.visible=0)
16                 ∧ (ED.enabled=0 ∧ ED.visible=0)
17                 ∧ (input.enabled=0 ∧ input.visible=0)
18                 ∧  ¬(Vkp=0)[...];
19    G_close:=(pc.enabled=1) ∧(pc.enabled=1) ;
20    G_aP_ED:=(DE.enabled=1 ∧DE.visible=1)
21                  ∧ /* Enumération explicite
22                    pour tout w : WIDGETS_parents(ED) */
23                    w.visible=1
24                  ∧ ¬(Vkp=0)
25    G_kp0:= /* condition d'activation de KeyPressed */
26                   ∧ (Vkp=1);
27    G_kp1:= /* condition d'activation de KeyPressed */ ∧
28                   input.text=empty;
29                   ∧ (Vkp=0);
30    G_kp2:= /* condition d'activation de KeyPressed*/ ∧
31                   input.text=full;
32                   ∧ (Vkp=0);
33    [...]
```

FIG. 4.9 – Module principal du modèle Nu_{Appl}.

Initialisation. La figure 4.10 présente la clause ASSIGN du modèle NuSMV traduisant sous la forme d'un système de transitions le comportement de l'application. L'initialisation de l'application est effectuée par la clause INIT. Les valeurs initiales peuvent être définies directement à partir de la clause INITIALISATION du modèle B_{Appl}.

Système de transitions. Le système de transitions est construit en définissant l'ensemble des systèmes de transitions de chaque variable du module via le mot clef `next()`.

```
1  Module main /* suite */
2   ASSIGN
3      /* initialisation des variables */
4    init(Vkp):=1; init(ua):=open; init(preua):=nil; init(ef.visible):=1;
5    init(DE.enabled):=0;  [...]
6
7      /* Définition du système de transition */
8    next(preua):=ua;
9    next(ua):=
10     case
11       /* Enumération explicite de toutes les possibilités dans le choix
12          de l'évènement. Choix indéterministe parmi l'ensemble des
13          événements réalisables */
14      ua ≠ LM_action : LM_action;
15      ua=LM_action ∧ G_kp1 ∧ G_kp2 ∧ G_apED ∧ G_apDE ∧ G_kp0 ∧ G_open ∧ G_close
16            : UA_evt;
17      ua=LM_action ∧ G_kp1 ∧ G_kp2 ∧ G_apED ∧ G_apDE ∧ G_kp0 ∧ G_open ∧ ¬G_close
18            : UA_evt-{close,nil};
19      ua=LM_action ∧ G_kp1 ∧ G_kp2 ∧ G_apED ∧ G_apDE ∧ G_kp0 ∧ ¬G_open ∧ G_close
20            : UA_evt-{open,nil};
21      [...]
22      1:nil; /* Choix par défaut */
23    esac;
24
25    /* Transition des valeurs des attributs de widgets */
26    next(ED.enabled) := case
27                          (ua=kp1) : 0;
28                          (ua=kp2) : 1;
29                          (ua=ap_ED) : 0;
30                          (ua=ap_DE) : 1;
31                          1 : ED.enabled;
32                        esac;
33
34    next(input.text) := case
35                          (ua=kp0) : {fill,empty} ;
36                          1 : input.text;
37                        esac;
38
39    next(output.text) :=case
40                          (ua=kp0) : 0;
41                          (ua=ap_ED) : 1;
42                          (ua=ap_DE) : 1;
43                          1 : out_T.text;
44                       esac;
45
46    next(Vkp) := case
47                   (ua=kp1) : 1;
48                   (ua=kp2) : 1;
49                   (ua=kp0) : 0;
50                   1 : Vkp;
51                 esac;
52    [...]
```

FIG. 4.10 – Module main : définition des transitions du modèle Nu_{Appl}.

Le système de transitions de la variable `ua` est construit en énumérant l'ensemble des événements possibles suivant l'état courant. Le prochain événement du système de transitions est choisi de manière indéterministe parmi l'ensemble des actions réalisables dans l'état courant. L'ensemble des actions réalisables est constitué par l'ensemble des événements dont la garde est vraie dans l'instant courant. `ua` prend la valeur `LM_action` pour tout état suivant un état dans lequel la valeur de `ua` appartient à `UA_evt={open,close,kp0,kp1,kp2,apED,apDE}`. Un état pour lequel `ua=LM_action` correspond à une reconfiguration de l'interface. Ceci est rendu nécessaire par le fait que

la prochaine action à réaliser est calculée en fonction de l'état courant (conditions d'activation représentées par les gardes). Lorsque la prochaine action est choisie, il est donc nécessaire de passer par un état transitoire dans lequel l'action choisie est réalisée de manière effective.

Les systèmes de transitions des variables du modèle sont construits en fonction du choix de l'événement courant `ua`. Par exemple, la valeur de l'attribut `enabled` du widget `ED` est modifiée par les événements `kp1`, `kp2`, `apED`, `apDE`, `open` et `close`.

D'un point de vue général, le système de transitions d'une variable `x` autre que `ua` peut être construit directement en considérant les événements du modèle B_{Appl}. Par exemple, si l'on suppose que la variable `x` apparaît dans le corps des événements B $\texttt{evt}_1$,..., $\texttt{evt}_k$ sous la forme `x :=`$\texttt{y}_i$, $i \in 1..k$, le système de transitions associé à la variable `x` dans le modèle NuSMV est : :

```
next(x)=case
    (ua=evt_1) : y_1 ;
      ...
    (ua=evt_i) : y_i ;
    1:x;
esac;
```

Pour plus de détails concernant le modèle Nu_{Appl}, l'annexe 3 propose le listing complet du modèle.

4.5 Validation de propriétés des modèles B_{Appl} et Nu_{Appl}

Les modèles B événementiel et NuSMV (B_{Appl} et Nu_{Appl}) présentés dans ce chapitre sont directement exploitables pour vérifier certaines propriétés de sûreté.

4.5.1 Validation en B événementiel : consistance du modèle B et propriétés de sûreté.

Consistance du modèle. La consistance du modèle B_{Appl} extrait de l'application Java-Swing est établie par la démonstration des Obligations de Preuve (OP) du modèle. Ces OP sont générées de manière automatique par le générateur de preuve. Le tableau 4.7 présente le nombre d'OP à décharger pour l'exemple du convertisseur. 40 OP sont générées. Parmi elles 24 sont déchargées automatiquement par le prouveur automatique et 16 OP doivent être prouvées de manière interactive.

Nombres d'OP générées	Preuves automatiques	Preuves interactives
40	24	16

TAB. 4.7 – Nombre d'obligations de preuve de consistance du modèle B_{Appl}.

Comme la structure de données utilisée pour la modélisation de l'application est abstraite, donc simplifiée, la plupart des OP à prouver interactivement sont également

simples. Notamment, toutes les OP devant être prouvées interactivement épousent l'une des deux formes présentées dans le tableau 4.8, où :

- SET_1 et SET_2 représentent des ensembles (en l'occurrence $SET_1 \in \{WIDGETS, JTextFields, \ldots\}$ et $SET_2 \in \{BOOL, TEXT\}$;
- $\{x_1, \ldots, x_k\} = SET_1$ et $y_{i,i\in 1..k} \in SET_2$;
- $f \in SET_1 \rightarrow SET_2$ et $g \in SET_1 \nrightarrow SET_2$.

Première forme :	$\{x_1 \mapsto y_1, \ldots, x_k \mapsto y_k\} \in SET_1 \rightarrow SET_2$
Seconde forme :	$f \leftarrow g \in SET_1 \rightarrow SET_2$

TAB. 4.8 – Formes des OP à décharger interactivement.

La preuve de ces propriétés consiste uniquement à aider l'assistant de preuve dans le choix des bonnes hypothèses[3] : la preuve devient ensuite automatique.

Propriétés de sûreté. Une *propriété de sûreté* énonce que, sous certaines conditions, quelque chose ne se produit jamais (ou se produit toujours). B événementiel est tout particulièrement adapté à l'expression de propriétés de sûreté : celles-ci sont représentées par des invariants du système.

Les propriétés de sûreté sont ainsi définies dans les clauses INVARIANTS et ASSERTIONS du modèle B_{Appl}. Le tableau 4.9 présente quelques propriétés qu'il est possible de formuler. Parmi elles, les propriétés $\mathbf{PROP1}_B$, $\mathbf{PROP2}_B$, $\mathbf{PROP3}_B$, $\mathbf{PROP4}_B$ sont vérifiées par le modèle du système et la propriété $\mathbf{PROP5}_B$ est fausse.

$\mathbf{PROP1}_B$	$V_{Open} = 1 \Rightarrow$ $enabled(input) = true \wedge visible(input) = true \wedge$ $visible(ED) = true \wedge visible(DE) = true) \wedge$ $visible(output) = true)$	✓
$\mathbf{PROP2}_B$	$(V_{Open} = 1 \wedge Jtext(input) = full \wedge Vkp \neq 0) \Rightarrow$ $(enabled(ED) = true \vee enabled(DE) = true)$	✓
$\mathbf{PROP3}_B$	$(V_{Open} = 1 \wedge Jtext(input) = empty \wedge Vkp \neq 0) \Rightarrow$ $(enabled(ED) = false \wedge enabled(DE) = false)$	✓
$\mathbf{PROP4}_B$	$(enabled(ED) = false \wedge enabled(DE) = true)$ $\vee(enabled(ED) = true \wedge enabled(DE) = false) \Rightarrow$ $JText(output) = full$	✓
$\mathbf{PROP5}_B$	$(V_{Open} = 1 \wedge Jtext(output) = full) \Rightarrow$ $(enabled(ED) = false \wedge enabled(DE) = true)$ $\vee(enabled(ED) = true \wedge enabled(DE) = false)$	⊗

TAB. 4.9 – Exemple de propriétés de sûreté exprimées en B événementiel.

La propriété $\mathbf{PROP1}_B$ exprime que si l'application est en cours d'exécution ($V_{Open} = 1$) alors le champ de texte *input* est toujours *actif* et *visible* et les widgets ED, DE

[3]Les hypothèses découlent directement de la forme des OPs. Dans le premier cas, les hypothèses à utiliser sont $\{x_1, \ldots, x_k\} = SET_1$ et $y_{i,i\in 1..k} \in SET_2$ (axiomes du modèle B_{Swing}). Dans le second cas, les hypothèses à utiliser sont : $f \in SET_1 \rightarrow SET_2$ et $g \in SET_1 \nrightarrow SET_2$

et *output* sont toujours visibles. Dit autrement, il est toujours possible pour l'utilisateur d'entrer une nouvelle valeur à convertir.

La propriété $\mathbf{PROP2}_B$ exprime que si l'application est ouverte et que le champ de texte *input* est rempli alors un des deux boutons ED ou DE est actif.

La propriété $\mathbf{PROP3}_B$ exprime que si l'application est ouverte et que le champ de texte *input* est vide, alors les deux boutons de conversion sont inactifs. Ainsi, l'utilisateur ne peut pas demander à l'application d'effectuer une conversion s'il n'a pas préalablement fourni une valeur à convertir.

La propriété $\mathbf{PROP4}_B$ exprime que si l'un des deux boutons est actif et l'autre inactif alors nécessairement le champ de texte *output* n'est pas vide. L'interprétation que l'on peut faire est la suivante : lorsque l'utilisateur appuie sur l'un des deux boutons de conversion le résultat de la conversion est affiché dans le champ de texte *output*.

Enfin, la $\mathbf{PROP5}_B$ exprime que si l'application est ouverte et que le champ de texte *output* est non vide, alors l'un des deux boutons est inactif. L'interprétation de cette propriété est la suivante : si le résultat d'une conversion est affiché alors il y a nécessairement eu une demande de conversion de la part de l'utilisateur. Cette propriété est fausse. En effet, une fois une conversion effectuée, l'utilisateur peut vider le champ de texte *input*. Dans ce cas, les deux boutons deviennent inactifs alors que le champ de texte *output* conserve la valeur de la précédente conversion. Cette propriété fausse peut éventuellement alerter le concepteur si celui-ci souhaite que lors d'un changement de la valeur du champ de texte *input* le champ de texte *output* soit vidé.

Il convient d'insister ici sur le fait que les propriétés "fausses" sont très intéressantes dans une démarche de conception/validation.

4.5.2 Validation en NuSMV : propriétés de sûreté, de vivacité et d'équité

NuSMV présente un avantage sur le B événementiel du point de vue de la vérification de propriétés. En effet, outre les *propriétés de sûreté*, l'utilisation de la logique temporelle permet de vérifier des *propriétés de vivacité* et *d'équité*.

Propriétés de sûreté. Ce sont les combinateurs AG en CTL, qui expriment le plus naturellement les propriétés de sûreté.

Le tableau 4.10 présente la formalisation de quelques propriétés de sûreté exprimées dans la logique temporelle CTL.

PROP1$_{Nu}$	AG($V_{Open} = 1 \Rightarrow$ $(input.enabled = 1 \wedge input.visible = 1)$ $\wedge\ (ED.visible = 1 \wedge DE.visible = 1)$ $\wedge\ output.visible = 1)$	✓
PROP2$_{Nu}$	AG($V_{Open} = 1 \wedge \neg(Vkp = 0) \wedge input.text = 1 \Rightarrow$ $(ED.enabled = 1 \vee DE.enabled = 1))$	✓
PROP3$_{Nu}$	AG($V_{Open} = 1 \wedge Vkp = 1 \wedge input.text = 0 \Rightarrow$ $(ED.enabled = 0 \wedge DE.enabled = 0))$	✓
PROP4$_{Nu}$	AG($V_{Open} = 1$ $(ED.enabled = 1 \vee DE.enabled = 0)$ $\wedge(ED.enabled = 0 \vee DE.enabled = 1)) \Rightarrow$ $output.text = 1)$	✓
PROP5$_{Nu}$	AG($preua = kp2 \Rightarrow$ EG($G_{apED} = 1$))	⊗
PROP6$_{Nu}$	AG($preua = kp2 \Rightarrow$ EG($AG_{apED} = 1 \vee G_{apDE} = 1$))	✓
PROP7$_{Nu}$	AG($preua = kp2 \Rightarrow$ EG($ua = LM\_action \vee ua = apDE \vee ua = apED$))	✓

Tab. 4.10 – Exemple de propriétés exprimées en CTL.

Les propriétés **PROP1**$_{Nu}$, **PROP2**$_{Nu}$, **PROP3**$_{Nu}$ et **PROP4**$_{Nu}$ sont identiques aux propriétés **PROP1**$_B$, **PROP2**$_B$, **PROP3**$_B$ et **PROP4**$_B$ mais exprimées à l'aide de la logique temporelle CTL (cf. section 2.2.3).

La propriété **PROP5**$_{Nu}$ exprime que, si la dernière action réalisée est $kp2$ alors il existe une séquence éventuellement infinie d'états suivant l'état courant pour lesquels la garde de l'action $apED$ est vraie. Autrement dit, si la dernière action de l'utilisateur a été de saisir une valeur (non vide) dans le champ de saisie *input* alors il peut cliquer indéfiniment sur le bouton ED. Cette propriété est fausse : lorsque l'utilisateur clique sur le bouton ED l'interface désactive ce bouton et par conséquent l'utilisateur ne peut plus appuyer de nouveau sur ce bouton.

La propriété **PROP6**$_{Nu}$ est vraie. Cette propriété exprime que si la dernière action de l'utilisateur est $kp2$ alors il existe une séquence éventuellement infinie d'états pour lesquels l'une des deux gardes G_{apDE} et G_{apED} est vraie. Cette propriété ne permet pas de préciser exactement la séquence éventuellement infinie en question. En remplaçant l'expression à droite de l'implication par EG(E $[(G_{apED} = 1)\mathbf{U}(G_{apDE} = 1)]$), on peut alors conclure que la séquence éventuellement infinie $(G_{apED}; G_{apED})*$ est réalisable.

La propriété **PROP7**$_{Nu}$ est identique à la propriété **PROP6**$_{Nu}$ mais est exprimée non pas en fonction des gardes des événements mais de la valeur de la variable ua.

Propriétés de vivacité. Une *propriété de vivacité* énonce que, sous certaines conditions, quelque chose finira par avoir lieu. Le tableau 4.11 présente quatre propriétés de vivacité.

PROP8$_{Nu}$	AG(EF(Gap_ED))	✓
PROP9$_{Nu}$	AG(EF($UI.output.text = 1$))	✓
PROP10$_{Nu}$	AG(($ua = ap\_DE \vee ua = ap\_ED$) $\Rightarrow$ $AF(UI.output.text = 1)$)	✓
PROP11$_{Nu}$	AG($ua = open \Rightarrow$ AF($Gap\_DE \vee Gap\_ED$))	⊗

TAB. 4.11 – Exemple de propriétés de vivacité exprimées en CTL.

La propriété **PROP8**$_{Nu}$ exprime que le système peut toujours retourner dans un état tel que Gap_ED soit vraie, c'est-à-dire dans un état où il est possible d'effectuer une conversion.

La propriété **PROP9**$_{Nu}$ est presque identique à la propriété précédente : le système peut toujours retourner dans un état pour lequel il est possible d'effectuer une conversion de dollars en euros ou d'euros en dollars.

La propriété **PROP10**$_{Nu}$ exprime que toute requête de conversion est satisfaite un jour, c'est-à-dire qu'une demande de conversion est toujours suivie par l'affichage de la valeur convertie.

Enfin, la propriété **PROP11**$_{Nu}$ exprime que toute ouverture de l'application finira par une action de conversion de la part de l'utilisateur. Cette dernière propriété est naturellement fausse.

Propriété d'équité. Une *propriété d'équité* énonce que, sous certaines conditions, quelque chose aura lieu (ou n'aura pas lieu) un *nombre infini* de fois. On parle aussi de *vivacité répétée*. Le combinateur GF (ou F$^\infty$) correspond exactement à "un nombre infini de fois", ou "infiniment souvent". Une exécution vérifie F$^\infty$ P si elle passe une infinité de fois par un état satisfaisant P.

G$^\infty$, le dual de F$^\infty$, est également très utile. Une exécution satisfait G$^\infty$ P si P est vérifiée pour tous les états visités, sauf peut-être un nombre fini d'entre eux, ou, de façon équivalente, si P est toujours vraie à partir d'un certain moment.

Cependant, il faut noter qu'une version pure de CTL ne permet pas d'exprimer les propriétés d'équité. En effet, CTL ne permet d'emboîter G et F qu'à la condition d'insérer un quantificateur de chemin, A ou E, entre le G et le F. Dans le cas de AF$^\infty$ P, il existe une façon de respecter la syntaxe CTL : AF$^\infty P$ est équivalent à AG AF P. Mais il n'y a pas de solution pour EF$^\infty P$, ni pour des énoncés plus riches [Schnoebelen *et al.*, 1999].

À défaut d'une logique permettant d'exprimer des propriétés d'équité, NuSMV propose de considérer que les hypothèses d'équité font partie du modèle (clause FAIRNESS du langage). Il est alors possible de vérifier des propriétés sous une hypothèse d'équité.

L'exemple du convertisseur Euros/Dollars ne se prête pas à l'expression de propriétés d'équité de la forme AG AF P. En effet, plusieurs boucles d'exécution interviennent dans cet exemple. Par exemple, il est possible de boucler sur la séquence d'actions ($kp0$ >> $kp1$ >> $kp2$) : dans cette situation, il est impossible de vérifier que quelque chose aura lieu infiniment souvent.

4.6 Conclusion

4.6.1 Génération de modèles

Ce chapitre a présenté les différentes étapes, techniques et règles de traduction permettant de construire des modèles de dialogue B événementiel et NuSMV d'une application par analyse de son code source Java-Swing. Par extension, ce type d'analyse statique est exploitable pour d'autres types de langages de programmation et n'est pas restreint à l'utilisation du langage Java-Swing.

Il n'est pas envisageable de proposer un outil permettant d'automatiser cette extraction de modèles formels pour toute application du fait de la complexité du langage Java-Swing. La connaissance de l'architecture du code source, des règles de codage et des widgets élémentaires utilisés s'avère indispensable pour permettre ou faciliter une analyse automatisée.

L'utilisation en amont d'outils de génération d'applications interactives (générateur d'interface, système de gestion d'interface utilisateur ou système basé sur modèle) respectant une architecture et des règles de programmation est par conséquent nécessaire à une analyse fiable en aval.

Ce dernier point ne constitue pas une limite à la méthodologie proposée : la nécessité d'une utilisation des méthodes formelles pour la validation des interfaces homme-machine ne concerne que les systèmes interactifs dits "critiques". Il est aujourd'hui reconnu que le développement de tels systèmes passe par la définition d'architectures logicielles précises et par des règles de codage strictes. En outre, l'utilisation de générateurs d'interfaces est aujourd'hui une pratique courante dans le développement de ces systèmes.

Modèles d'architecture. L'étude s'est fondée sur une architecture de type SEEHEIM. L'avantage d'une telle architecture est de proposer une séparation de la partie interactive du système et du noyau fonctionnel de l'application. Cette séparation permet le développement en parallèle des deux parties du système interactif (Noyau Fonctionnel et Interface) et facilite les processus d'abstraction permettant d'isoler la partie comportementale de l'interface lors de l'analyse statique. D'une manière générale, toute architecture basée sur la séparation du noyau fonctionnel et de la partie interactive de l'interface peut être analysée suivant les principes exposés dans ce chapitre. Dans le cas d'une architecture de type multi-agents, telle que MVC, l'analyse est plus délicate puisque les fonctionnalités du noyau fonctionnel peuvent être réparties au sein des différents agents autonomes. Cependant, il est tout à fait envisageable d'établir une ségrégation entre la partie purement interactive et le noyau fonctionnel dans le cas d'une architecture MVC. En conclusion, l'analyse d'un code source Java-Swing peut être effectuée quelle que soit l'architecture utilisée à partir du moment où cette architecture est connue et qu'il est possible d'isoler de manière conceptuelle le noyau fonctionnel du reste de l'application : les stratégies d'extraction de modèles formels peuvent alors être optimisées en fonction du type d'architecture retenu.

Modélisation de la boîte à outils Java-Swing. Les modélisations formelles proposées nécessitent une modélisation des composants élémentaires de la boite à outils Java-Swing : ces composants sont en nombre important. Il est indispensable de disposer pour

chacun de ces composants d'une abstraction capturant les attributs nécessaires à la modélisation globale de l'interface. Suivant la nature des propriétés à valider sur le modèle, ces abstractions peuvent être menées de différentes manières.

Seules les informations relatives à la disponibilité du widget (visibilité et activité) et quelques informations relatives à son contenu (valeur du texte pour un champ de saisie) sont conservées lors de l'abstraction proposée ici. Ainsi, le développeur réalisant l'analyse d'un code source doit s'assurer qu'à chaque widget utilisé dans le code correspond une abstraction. Ces abstractions de widgets peuvent être définies directement par le développeur en Java-Swing. Par conséquent, au fur et à mesure des analyses, il est possible d'utiliser les abstractions de widgets préalablement définies, de modifier une abstraction ou encore d'ajouter de nouvelles abstractions et ceci sans passer par la syntaxe des langages formels. Une bibliothèque d'abstractions est définie. Cette bibliothèque est facilement modifiable dans le langage source du développeur et réutilisable pour différentes analyses.

Atout de la démarche. Si l'utilisation des méthodes formelles pour le développement du noyau fonctionnel d'une application tend à se populariser, notamment dans les domaines ferroviaire et avionique, ces techniques sont difficilement exploitables dans le domaine de l'interaction homme-machine. La solution consistant à utiliser les méthodes formelles uniquement dans la phase de validation supprime l'étape délicate de la construction d'un modèle formel lors de la phase de développement de l'application. L'utilisation conjointe d'un générateur d'interface et d'un outil permettant d'extraire de manière automatique un modèle formel à partir du code source généré permet au développeur de conserver ses méthodes de travail habituelles et plus particulièrement le langage de programmation qu'il manipule.

L'extraction d'un modèle formel B événementiel, ou suivant un autre formalisme comme démontré avec l'utilisation de NuSMV, est réalisable de manière automatique par analyse du code source de l'application. Afin de mettre en évidence la faisabilité de cette automatisation (choix des représentations intermédiaires, algorithmes...), les détails relatifs à l'analyse statique d'un code Java-Swing sont proposés au chapitre suivant.

Comparaison : B événementiel et NuSMV. Comme on le constate sur l'exemple de notre étude de cas, il est possible de construire un modèle NuSMV à partir d'un modèle B événementiel de manière ad hoc. La notation B événementiel reste cependant plus puissante du point de vue de son pouvoir d'expression. La présence de la logique du premier ordre assure une description concise du modèle et des événements.

L'absence des quantificateurs existentiel et universel dans le cas de la méthode NuSMV nécessite une énumération explicite des ensembles manipulés : par exemple la définition du système de transitions associé à la variable `ua` du modèle doit expliciter l'ensemble des combinaisons possibles afin que le choix indéterministe du prochain événement déclenché soit assuré. Dans le cas de B événementiel cette énumération reste implicite et codée dans la garde des événements.

Le modèle B événementiel met l'accent sur les événements : il est donc aisé de déterminer quelle est la réaction de l'interface (c'est-à-dire la modification de l'espace d'état du système) en fonction d'une action utilisateur particulière. Pour cela, il suffit de trouver les événements B qui modélisent l'exécution de la méthode d'écouteur d'événement déclenchée lors de l'action utilisateur concernée.

Le modèle NuSMV met plus l'accent sur la notion de variable puisque celui-ci définit explicitement le système de transition associé à chacune d'elle. Dans le modèle NuSMV il est donc plus aisé de déterminer quelles sont les actions de l'utilisateur qui modifient une variable particulière.

Ces deux modèles permettent donc d'obtenir deux perspectives différentes du système. L'utilisation conjointe de ces deux modèles peut se révéler utile pour le développeur afin de mieux appréhender le fonctionnement du système.

4.6.2 Validation de propriétés

Les modèles formels obtenus sont directement exploitables afin de vérifier que le système satisfait un ensemble de propriétés de sûreté. Ces vérifications permettent de s'assurer de la fiabilité du système conçu et déchargent le développeur d'une partie des activités de tests à réaliser. Bien que le développeur n'ait pas besoin d'utiliser les langages formels au cours du développement, la validation de propriétés de sûreté nécessite, elle, l'intervention d'un expert. La formulation des propriétés du système s'appuie sur la connaissance du logiciel développé et une bonne compréhension du modèle formel extrait.

Propriétés : B événementiel. La validation de la consistance du modèle B_{Appl} est relativement aisée. Les principales obligations de preuves à prouver manuellement correspondent aux obligations de préservation par les événements de l'invariant de typage des fonctions totales du modèle. La démonstration de ces invariants est systématique et pourrait être automatisée.

Suivant le nombre d'événements du modèle B_{Appl}, la formulation des assertions peut être délicate. Elle nécessite une bonne connaissance du modèle. Dès lors que plusieurs méthodes d'écouteurs sont traduites par plusieurs événements B, le nombre de variables de contrôle permettant le séquencement des événements est important. L'expression d'une assertion fait alors appel à ces variables de contrôle afin d'exprimer des propriétés relatives à certains états du système.

Propriétés : NuSMV. La validation de propriétés du modèle Nu_{Appl} nécessite également une bonne compréhension du modèle et une bonne connaissance de la logique temporelle CTL pour l'expression de propriétés. NuSMV possède un avantage du point de vue des propriétés vis-à-vis de B événementiel puisqu'il permet l'expression de propriétés de sûreté, de vivacité et d'équité là où B permet seulement d'exprimer des propriétés de sûreté. En revanche, NuSMV permet de modéliser uniquement des systèmes finis.

Comparaison : B événementiel et NuSMV. Du point de vue de la preuve, la formalisation des propriétés à vérifier présente la même difficulté qu'il s'agisse d'utiliser le langage B ou la logique temporelle CTL. En ce qui concerne la démonstration de ces propriétés, on retrouve les avantages et désavantages liés à la technique de preuve utilisée. Dans le cas d'une preuve exploitant la méthode B, l'utilisateur est confronté à la preuve interactive de propriétés (*theorem-proving*). Dans le cas de l'utilisation du *model-checking* (NuSMV) la preuve est plus simple puisqu'elle est automatisée. En revanche l'utilisateur peut être confronté au problème de l'explosion du nombre d'états. Ces problèmes n'apparaissent pas directement dans l'étude de cas du fait de la simplicité de l'interface étudiée.

Pour des systèmes interactifs plus complexes, le nombre de widgets et d'actions possibles sur l'interface peut être très grand. Dans ce cas, l'ensemble des chemins d'interaction possibles sur l'interface est très important ce qui pose le problème du nombre d'états atteignables dans le cas ou le système d'états-transitions du système est construit de manière effective comme c'est le cas avec NuSMV. La méthode B événementiel s'affranchit de ce problème d'explosion du nombre d'état, ou du moins en partie : le nombre d'obligations de preuves suit de manière proportionnelle le nombre d'événements B introduits.

Une utilisation conjointe des deux techniques de preuve présentées peut s'avérer intéressante. La formulation des propriétés diffère suivant le langage adopté. Ceci ouvre un champ d'expression plus grand et une meilleure conceptualisation des propriétés que l'on souhaite vérifier.

Chapitre 5

Techniques d'analyse statique de codes Java-Swing

Le chapitre précédent a présenté les principes de la modélisation du dialogue d'une application Java-Swing suivant les méthodes formelles B événementiel et NuSMV. Ce chapitre a pour objectif de présenter les différentes étapes de l'analyse statique d'un code source Java-Swing permettant l'extraction de ces modèles formels afin de mettre en évidence la faisabilité d'une automatisation de la démarche. Les différentes étapes de l'analyse ainsi que les modèles intermédiaires et une partie des algorithmes utilisés sont présentés.

5.1 Présentation détaillée du processus d'analyse statique

La figure 5.1 présente les différentes étapes ainsi que les différentes représentations intermédiaires utilisées au cours de l'analyse statique. Dans ce schéma toute la partie encadrée par des pointillés correspond aux différentes étapes de l'analyse réalisées de manière automatique. Le processus d'analyse nécessite trois entrées :

1. le *code source* de l'application analysée ;
2. une *abstraction du noyau fonctionnel* codée en Java ;
3. une *abstraction des widgets élémentaires* de la bibliothèque Swing également écrite dans le langage Java.

Première étape. La première étape (Fig.5.1,❶) est une analyse syntaxique du code source de l'application. Cette analyse produit un Arbre de Syntaxe Abstraite (AST) de l'application. Une première opération sur cet arbre, non représentée sur ce schéma, transforme cet AST en un Graphe de Dépendance Java (JSysDG, cf. section 2.3). Des renseignements sur la construction d'un graphe de dépendance d'une application Java peuvent

être trouvés dans [Walkinshaw & Roper, 2003]. Le choix de cette représentation intermédiaire (JSysDG) a été fait afin de simplifier l'exposé des algorithmes d'analyse présentés dans ce chapitre. Cette représentation n'est pas indispensable : toutes les opérations effectuées sur le graphe de dépendances peuvent être réalisées directement sur l'arbre de syntaxe abstraite.

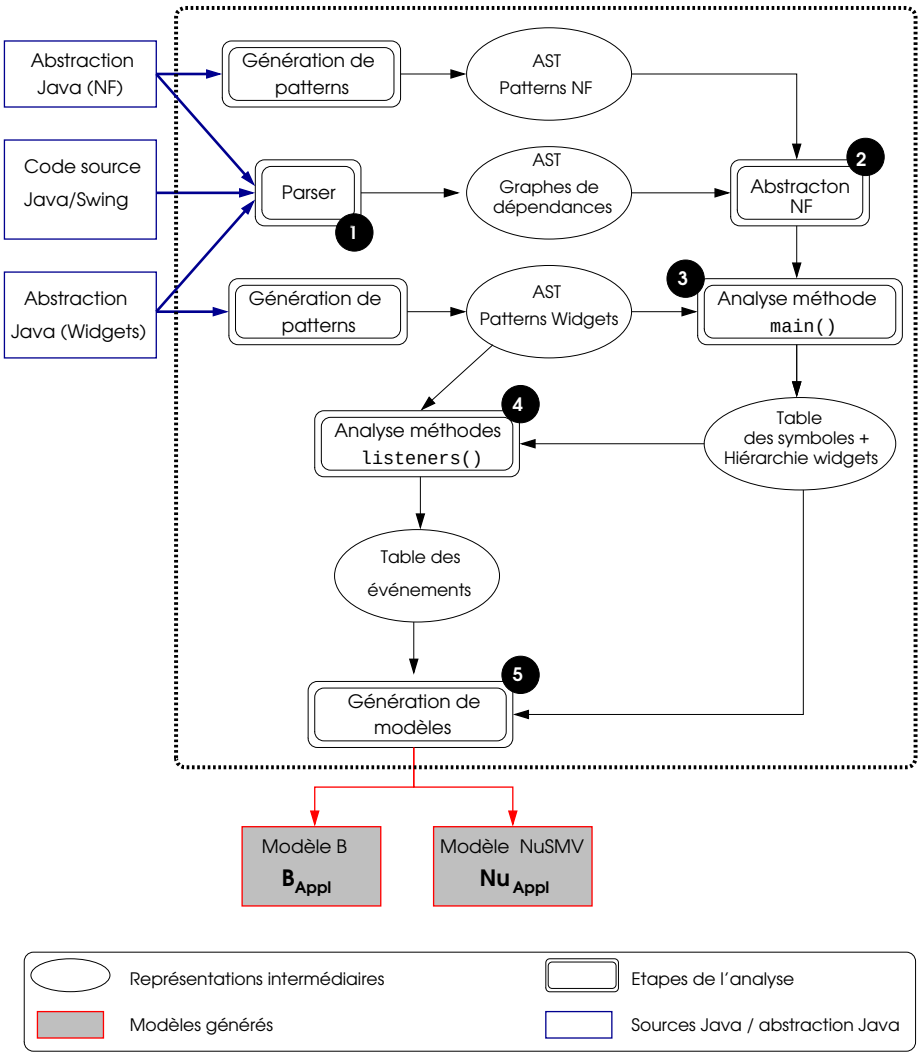


FIG. 5.1 – Schéma détaillé du processus d'analyse statique.

Abstraction du noyau fonctionnel. La seconde étape (Fig.5.1,❷) de l'analyse consiste à abstraire le graphe de dépendances Java de l'application. Cette abstraction concerne le noyau fonctionnel de l'application. Afin d'abstraire les instructions relatives au noyau fonctionnel, cette étape réalise une reconnaissance de "patterns". Ces patterns sont construits à partir du nom des méthodes appartenant au noyau fonctionnel. A chaque méthode du noyau fonctionnel est associé un ensemble de patterns Java représentés sous forme d'AST. Ces patterns permettent de représenter un appel au noyau fonctionnel. La comparaison d'un de ces patterns avec une instruction du code source permet alors de conclure si l'instruction analysée correspond à un appel au noyau fonctionnel. Suivant la nature du pattern reconnu, le noeud du graphe de dépendances correspondant à l'ins-

truction est alors soit supprimé (appel simple sans retour de valeur) ou bien abstrait en remplaçant l'expression d'appel par une constante dont le type est identique au type de retour de la méthode appelée (cf. section 4.3).

Analyse de la méthode `main()`. Une fois l'abstraction du noyau fonctionnel réalisée, l'étape suivante (Fig.5.1,❸) consiste à analyser la méthode `main()` de l'application. Comme évoqué au chapitre précédent, cette analyse permet la construction du modèle structurel de l'application. Cette étape consiste à définir la hiérarchie des widgets de l'application (association widgets/widgets), les associations widgets/écouteurs et les valeurs initiales des widgets et de leurs attributs caractéristiques. Les attributs caractéristiques dépendent du modèle à extraire. Les abstractions Java en entrée de l'application définissent ces attributs caractéristiques. La reconnaissance d'une instruction à prendre en compte dans la modélisation est effectuée par une reconnaissance de patterns.

L'analyse de la méthode `main()` génère plusieurs tables de symboles fournissant une représentation des objets de l'application. Une table des symboles "générale", constituée en réalité d'une liste de tables chaînées, représente l'ensemble des objets manipulés par l'application. Une table spécifique est également construite afin de représenter la hiérarchie des widgets de l'application et leurs associations avec des écouteurs d'événements. Cette dernière table permet de construire le contexte et l'initialisation du modèle B événementiel B_{Appl} généré par l'analyse (clauses SETS, PROPERTIES, VARIABLES, INVARIANT, INITIALISATION du modèle).

Analyse des méthodes d'écouteurs. Les tables de symboles générées lors de l'analyse de la méthode d'initialisation `main()` sont utilisées lors de l'étape d'analyse des méthodes d'écouteur de l'application (Fig.5.1,❹). L'analyse des méthodes d'écouteur est basée sur l'utilisation de patterns.

5.2 Reconnaissance d'instructions spécifiques : patterns

Plusieurs étapes de l'analyse statique utilisent la reconnaissance de patterns. Un *pattern* est un "schéma" décrivant un ensemble d'instructions Java dont la structure est cohérente avec celle du schéma. Par exemple, lors de l'analyse des méthodes `main()` et des méthodes d'écouteurs on souhaite reconnaître des instructions du type : `input.setVisible(true)` (changement de visibilité d'un widget), `pc.add(inputPanel)` (construction de la hiérarchie de widget), `EF.addActionListener(this)`, etc.

Un pattern Java sera représenté sous la forme d'un AST permettant de représenter ces diverses instructions. Par exemple, un pattern générique pour reconnaître une modification de la visibilité d'un widget sera représenté par : `$$$$$.setVisible($$$$$)`.

Reconnaissance d'instructions. La reconnaissance d'une instruction spécifique consiste à comparer l'arbre de syntaxe abstraite de l'instruction analysée avec l'arbre de syntaxe abstraite d'un pattern. Cette comparaison est effectuée pour chaque noeud de l'arbre de syntaxe. Si l'un des noeuds de l'AST de l'instruction est différent de celui du pattern, la comparaison est arrêtée et le pattern est considéré comme non reconnu. Dans le cas contraire, si l'ensemble des noeuds de l'AST est identique à l'en-

semble des noeuds du pattern, alors l'instruction est reconnue. En réalité, l'égalité stricte de ces ensembles n'est pas nécessaire : c'est le rôle joué par l'identifiant `$$$$$` qui est utilisé dans le pattern. Un noeud du pattern correspondant à l'identifiant `$$$$$` accepte n'importe quel sous-arbre correspondant à une expression Java. Ainsi le pattern `$$$$$.setVisible($$$$$)` reconnaît les instructions : `pc.input.setVisible(false)`, `FE.setVisible(true)`, `VariableInexistante.setVisible('du texte qui n'a rien à faire là')`.

5.3 Pré-Process : Abstraction de l'application

5.3.1 Abstraction du noyau fonctionnel

L'abstraction du noyau fonctionnel est codée dans le texte du fichier `NF_convertisseur.java`. Ce fichier est quasiment vide et ne contient qu'une seule méthode : le constructeur de la classe.

Le listing 5.2 présente l'abstraction réalisée :

```
1 class FunctionalCore{
2   public FunctionalCore(){
3   }
4 }
```

FIG. 5.2 – Abstraction du noyau fonctionnel : `FunctionalCore.java`.

Certaines classes Java qui définissent l'application interactive sont susceptibles de faire appel aux méthodes du noyau fonctionnel de l'application, méthodes qui ne sont pas contenues par définition dans l'abstraction du NF réalisée. Dans le cas d'une architecture logicielle de type SEEHEIM, les appels au noyau fonctionnel se font exclusivement par l'intermédiaire du contrôleur de dialogue.

Quelle que soit l'architecture utilisée, il est nécessaire d'abstraire tout appel de méthode du NF, sauf la méthode constructrice de la classe. L'abstraction réalisée consiste à remplacer l'appel de la méthode par un identifiant factice qui sera traité lors des analyses suivantes. Cet identifiant prendra la forme `NewType` ou `Type` est remplacé par le nom du type de retour de la méthode appelée. Dans le cas d'un appel de méthode ne possédant pas de retour (`void`), l'instruction est tout simplement supprimée.

Ceci nécessite de disposer des classes définissant le NF en entrée de l'analyse. Ces classes sont analysées afin de construire une table répertoriant les noms (identifiants) et type de retour de l'ensemble des méthodes du NF. Cette analyse peut se faire directement sur l'AST des différentes classes composant le NF.

Pour chaque identifiant de méthode, des patterns représentant les diverses formes d'un appel à cette méthode sont générés sous forme d'AST. Parmi ces patterns, on trouve les instructions abstraites Java présentées ci-dessous, ou `IdMethod` représente le nom de la méthode en cours de traitement :

– pattern 1 : `IdMethod($$$$$);`
– pattern 2 : `$$$$$=IdMethod($$$$$);`
– pattern 3 : `$$$$$.IdMethod($$$$$);`

– pattern 4 : `$$$$$=$$$$$.IdMethod($$$$$);`

La dernière étape consiste à parcourir de manière exhaustive les AST des classes Java représentant l'application interactive (classes du NF exclues) à la recherche des patterns préalablement établis. Comme évoqué plus haut, lorsqu'un pattern est reconnu, le sous arbre de syntaxe abstraite correspondant au pattern est abstrait suivant la nature de l'instruction et du type de retour de la méthode appelée.

```
1  class widget{
2    private widget[] fils;
3    private boolean visible;
4    private boolean enabled;
5
6    /* Constructeur de la classe */
7    public widget(){
8      visible=false;
9      enabled=false;
10     parent={};
11   }
12
13   /* Méthodes pertinentes du point de vue de l'interaction */
14   public void setVisible(boolean v){
15     visible=v;
16   }
17
18   public void setEnabled(boolean v){
19     enabled=v;
20   }
21
22   public boolean isVisible(){
23     return visible;
24   }
25
26   public boolean isEnabled(){
27   return enabled;
28   }
29
30   public void add(widget w){
31   fils.add(w);
32   }
33
34   /* Méthodes non pertinentes du point de vue de l'interaction */
35   public void setBackground(Color c){
36   }
37   [...]
38 }
```

Fig. 5.3 – Modèle de widget générique : `widget.java`.

5.3.2 Bibliothèque de modèles : abstraction des objets d'interaction

Afin d'effectuer une analyse de l'application, quelques abstractions sont à réaliser. Les programmes font appel à des classes Java importées dont, en particulier, les librairies implantant les objets d'interaction. Ces librairies sont principalement: `javax.swing` et `java.awt`.

Principalement parce que les objets de ces classes sont manipulés par des opérateurs qui sont les méthodes de ces classes et par des opérateurs de base du langage Java, on construira systématiquement des abstractions de certaines classes Java-Swing ou de certaines classes Java. Ces abstractions doivent permettre de représenter les aspects structu-

rels et comportementaux des widgets : visibilité, activité, association d'écouteurs d'événements et association widgets/widgets permettant de représenter la structure hiérarchique des widgets composant l'interface.

À l'inverse, certaines informations sur l'état des widgets ne sont pas nécessaires : couleur du widget, taille, position, etc. Ces informations pourraient être utiles pour la validation de propriétés IHM telles que l'insistance par exemple.

Ainsi, on se donne un ensemble de classes Java afin de réaliser une bibliothèque de modèles des composants d'interaction, bibliothèque définissant une abstraction des objets d'interaction intervenant dans le programme source. Ces classes définissent des modèles d'exécution des composants d'interaction. Elles constituent une entrée de l'analyse statique.

La première classe `widget.java` constitue un modèle générique d'un composant d'interaction dont tous les composants plus spécifiques hériteront (`JTextField`, `JButton`, `JFrame`, etc...). La figure 5.3 présente le code de ce premier modèle.

Dans ce modèle, seules les informations génériques communes à tous les widgets sont répertoriées comme des attributs de classes : visibilité du widget sur l'interface, activité du widget, et les fils du widget dans la hiérarchie construite lors de l'initialisation de l'application. Les méthodes accesseurs de ces attributs sont également définies : `setVisible`, `isVisible`, `setEnabled`... Les méthodes faisant référence à des attributs sans intérêt du point de vue de l'analyse peuvent également être définies dans cette classe avec un corps de méthode vide : c'est le cas de la méthode `setBackground`.

Par défaut, lors des étapes suivantes, si une instruction correspond à un appel de méthode sur un objet de type widget, et que cette méthode n'est pas définie dans les modèles de la bibliothèque, alors cette instruction sera ignorée (corps de la méthode considéré comme vide).

Les figures 5.4 et 5.5 présentent respectivement les classes `JButton` et `JTextField`, exemples de modèles de widgets spécifiques.

```
1  class JButton extends widget{
2    private ActionListener[] listeners;
3
4    public JButton(){
5      super();
6      listeners={};
7    }
8
9    public void addActionListener(ActionListener al){
10     listeners.add(al);
11   }
12
13 }
```

FIG. 5.4 – Modèle de widget spécifique : `JButton.java`.

```
1  class JTextField{
2    private String text;
3    private KeyListeners[] keyListeners;
4
5    public JTextFields(){
6      super();
7      keyListeners={};
8      text="";
9    }
10
11   public void addKeyListener(KeyListener kl){
12   keyListeners.add(kl);
13   }
14
15   public void setText(String txt){
16     text=txt;
17   }
18
19   public String getText(){
20     return text;
21   }
22 }
```

FIG. 5.5 – Modèle de widget spécifique : `JTextField.java`.

5.4 Analyse de la méthode `main()`

L'analyse de la méthode `main()` de l'application assure la construction du contexte de l'application. Cette analyse produit un ensemble de représentations intermédiaires qui permettent la définition des clauses SETS, PROPERTIES, VARIABLES, INVARIANT et INITIALISATION du modèle d'exécution B_{Appl}.

L'analyse de la méthode `main()` utilise l'AST obtenu après abstraction de l'AST original. D'un point de vue pratique, cette opération est réalisable directement par analyse de l'AST abstrait. Cependant, afin d'améliorer la lisibilité des algorithmes présentés dans cette section et dans les sections suivantes, on considère que l'AST abstrait est transformé en un graphe de dépendances système Java (JSysDG) (cf. section 2.3).

L'analyse présentée dans cette section vise trois objectifs principaux :

1. la *définition de tables des symboles* permettant de répertorier l'ensemble des instances créées à l'initialisation, ainsi que leurs valeurs initiales. Ces tables sont indispensables à l'analyse des méthodes d'écouteurs ;
2. la définition d'une *table répertoriant uniquement les instances de widgets créés à l'initialisation* de l'application. Les objets de cette table pointent sur l'objet correspondant dans la table des symboles principale. En fin d'analyse, cette table contient toutes les informations permettant de construire la hiérarchie de widgets. On désigne cette table par `WidgetsTree` ;
3. la *définition d'une table répertoriant l'ensemble des écouteurs* de l'application.

Avant de présenter l'algorithme implémentant l'analyse de la méthode `main()`, il est nécessaire de définir :

- l'organisation des tables de symboles créées lors de l'analyse et les opérations permettant de les manipuler ;
- l'organisation de la structure `WidgetsTree` ;

– et quelques notations intermédiaires.

5.4.1 Tables des symboles

Portée des identificateurs

Dans les différentes phases de l'analyse statique du programme Java/Swing, un parcours du graphe de dépendances du système est effectué. Au cours de l'analyse, il est indispensable de connaître la valeur des attributs (type,valeur...) associés à un identificateur. La *table des symboles* permet de centraliser l'information concernant l'environnement de l'analyse.

Définition : Liaison. *On notera dans la suite $x \mapsto v$ la liaison entre un identificateur x et un objet v. Cet objet v représente la valeur des attributs de l'identificateur.*

Définition : Environnement. *On appelle environnement un ensemble de liaisons. Selon la nature des valeurs associées aux identificateurs (constantes, cases mémoire, fonctions, types, etc.), plusieurs environnements peuvent être définis. Un environnement s'écrira $\{x1 \mapsto v1, \ldots, xn \mapsto vn\}$.*

L'analyse statique doit être capable de déterminer, lors de l'utilisation d'un identificateur, quelle est la définition *active* pour cet usage. Un même nom d'identificateur défini à l'entrée d'un bloc peut être *redéfini* dans un bloc d'instructions plus interne : il ne référencera donc plus le même objet.

Ainsi, pour les langages à structures de bloc tels que Java on définit les termes suivants :

– **portée statique** : la portée d'une définition d'identificateur (variable, méthode, type), c'est-à-dire la partie du programme où la liaison pour cet identificateur est active, peut être obtenue *statiquement* en analysant le texte du programme ;
– **redéfinition/hiding** : une liaison peut être *cachée temporairement* par une nouvelle liaison de *même nature* pour un même identificateur, mais seulement dans un bloc interne du bloc de la première définition ;
– **LIFO** : l'ordre dans lequel les liaisons sont introduites et détruites est du type "Last In First Out", ce qui suggère une pile comme structure de données adaptée. On désigne cette pile comme la *pile undo*.

Table des symboles

Plusieurs environnements sont utilisés au cours de l'analyse et évoluent au cours de cette analyse.

Portée des identificateurs : structure de données. On souhaite que les structures de données permettent :

– d'*accéder rapidement aux liaisons* par le nom de l'identificateur ;
– de *gérer facilement l'évolution des environnements*, notamment l'opération d'ajout d'une liaison et de suppression d'une liaison restaurant (éventuellement) la liaison précédente.

Solution impérative. Parmi les représentations possibles, on utilisera des *tables de hachage* avec une pile *undo* permettant la gestion des redéfinitions (cf. figure 5.6). Pour la gestion des redéfinitions, les actions suivantes devront être réalisées :

- *entrée dans un bloc* : on empile un marqueur dans la pile *undo*, puis on insère les définitions locales dans la table de symboles principale ;
- *insertion* : on ajoute la valeur en tête de la liste correspondant à la clef dans la table (liaison) et on empile la clef sur la pile *undo* ;
- *sortie de bloc* : on dépile et on supprime toutes les clefs jusqu'au marqueur de bloc ;
- *suppression* : on enlève l'élément en tête de liste dans la table.

Représentation des objets et organisation des tables de symboles

Représentation des objets. Un objet associé à une clef (liaison) de la table des symboles représente une instance de classe créée lors de l'initialisation de l'application. Une instance est définie par son type (`Type`) et sa valeur (`Value`). Dans le cas où l'objet possède un type primitif (booléen, entier), l'attribut `Value` correspond à la valeur de l'objet. Dans le cas où le type de l'objet est représenté par une classe, la valeur correspond alors au nom de la classe. Étant donné qu'une classe peut étendre ou implémenter d'autres interfaces ou d'autres classes, un même objet peut posséder plusieurs types. Ainsi, l'attribut `Type` d'un objet est représenté par un ensemble de types.

Dans le cas où le type de l'objet n'est pas un type élémentaire (instance de classe), une nouvelle table de symboles représentant les valeurs des attributs et les méthodes associées à l'instance est construite. L'attribut `Env` possède alors l'identifiant correspondant à la nouvelle table créée. Chaque table possède également un lien (`previous`) vers l'objet qu'elle représente. Ce lien est constitué de l'identifiant de l'objet et de l'identifiant de la table dans laquelle cet objet est présent.

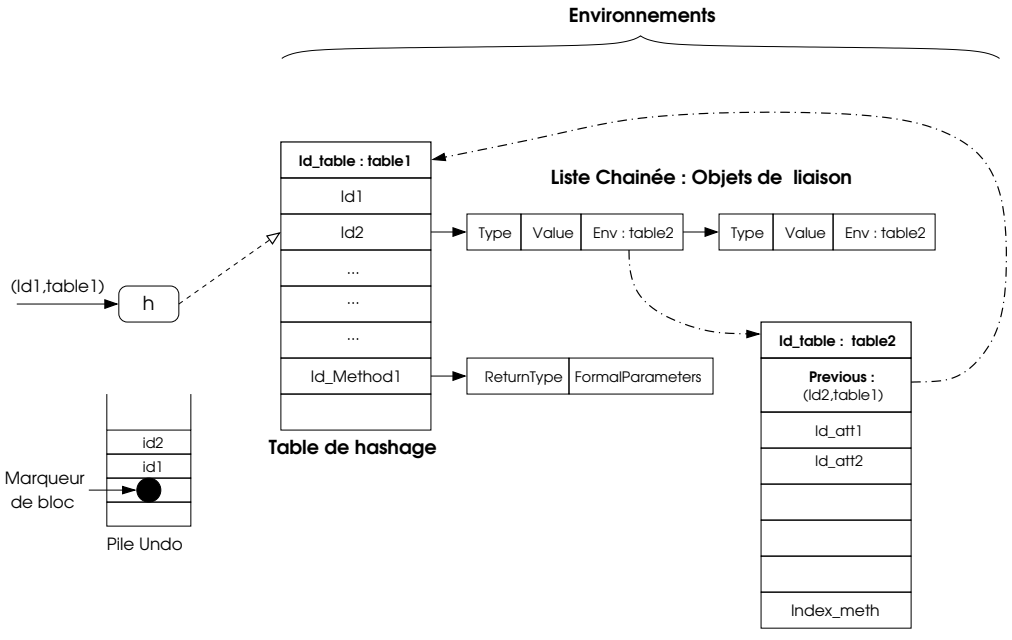


FIG. 5.6 – Tables des symboles et pile de *undo*.

Organisation des tables de symboles. L'environnement d'exécution est par conséquent constitué d'un ensemble de tables de symboles représentant les objets de l'application. Ces tables sont chaînées ce qui facilite l'accès aux objets de l'application lors de l'analyse d'une instruction. Par exemple, la référence à un objet de l'application apparaissant dans une instruction sous la forme `IdObj.Att1` peut facilement être repérée dans la table des symboles : il s'agit de trouver l'objet `Att1` dans la table référencée par l'attribut `Env` de l'objet `IdObj`. La figure 5.6 représente l'organisation des tables de symboles définies.

Primitives de manipulation des tables de symboles et de la pile undo

Afin de manipuler les différents environnements de l'analyse, on fait appel aux primitives de gestion de tables de symboles et de pile *undo* suivantes :

1. `mktable(previous,IdTable)` : création d'une table de symboles `IdTable` liée à la table `previous` ;
2. `insert(IdTable,IdObject,Object)` : insertion de l'index `IdObject` dans la table `IdTable`. `Object` représente le symbole lié à `IdObject`. Cet objet est représenté par `[Type,Value,Env]` comme sur la figure 5.6 ;
3. `previous(IdTable)` : renvoie la table `previous` associée à `IdTable` ;
4. `remove()` : dépile toutes les clefs de la pile *undo* jusqu'au premier marqueur de bloc et supprime les premiers objets des listes chainées qui sont associées à ces clefs dans l'environnement ;
5. `enterObjectEnv(IdTable, IdObject, IdEnv)`, `enterObjectValue(IdTable, IdObject, IdValue)` : modifient respectivement l'environnement et la valeur de l'objet accessible par la clef `IdObject` dans la table `IdTable`.

5.4.2 Structure hiérarchique de widgets : table WidgetsTree

Cette structure sera représentée par un arbre tel que présenté par la figure 5.7. Du point de vue de l'implémentation, cet arbre est représenté par une nouvelle table de symboles. Les objets de cette table correspondent aux widgets de l'application. À chaque identifiant de widget instancié `Widget_Id`, la table de hachage fait correspondre un objet possédant 4 attributs :

1. `Widget_Types` dénote le type du widget ;
2. `Id_table` correspond à l'identifiant de la table des symboles de l'environnement principal où l'objet est également défini : ce lien permet d'accéder rapidement aux valeurs des différents attributs du widget ;
3. `listeners` est un ensemble de couples `(Id,ListenerType)` représentant l'ensemble des écouteurs associés au widget ;
4. `fils` et `parent` désignent l'ensemble des fils du widget dans la hiérarchie de widgets de l'application, et le parent direct de ce widget dans cette même hiérarchie.

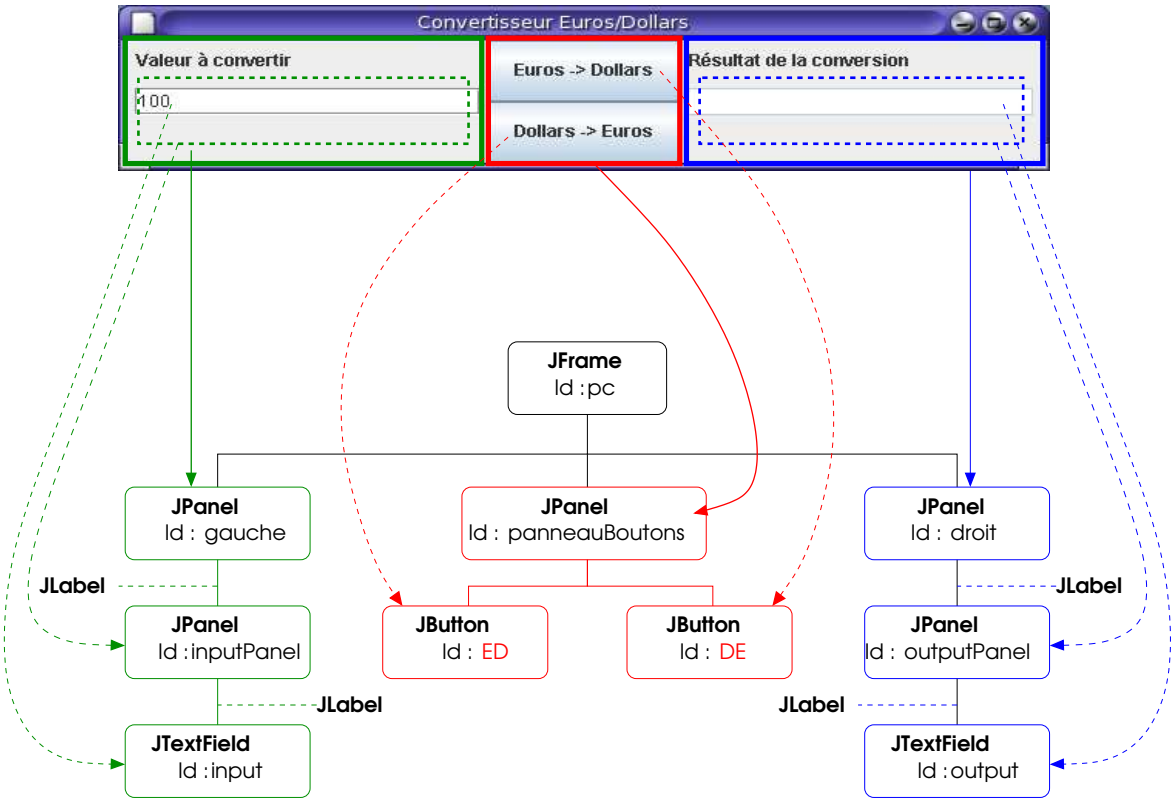


FIG. 5.7 – Structure hiérarchique de widgets : `WidgetsTree`.

Table d'écouteurs. Enfin, une table regroupant l'ensemble des écouteurs de l'application est définie. Cette table fait correspondre à chaque identifiant d'écouteur le type de l'écouteur et la clef permettant d'accéder à l'objet correspondant dans la table de l'environnement principal. L'objet correspondant est une instance de classe implémentant une interface *listener*.

5.4.3 Notations intermédiaires

Afin de présenter l'algorithme d'analyse de la méthode `main()` le plus clairement possible, on fait appel aux notations intermédiaires suivantes qui permettent de manipuler le graphe de dépendances système Java (JSysDG). La compréhension de ces notations intermédiaires nécessite la connaissance de l'organisation d'un graphe de dépendances de système Java (JSysDG) (cf. section 2.3.5). On note :

1. α(`n`) : la fonction faisant correspondre au noeud `n` appartenant au graphe de flots de contrôle (CFG) étendu d'une méthode `M`, le noeud d'entrée de la méthode `M` (*Method Entry Vertex*). Il s'agit de trouver l'antécédent du noeud `n` par l'arc de dépendance vis-à-vis du contrôle dans le JSysDG ;
2. β(`n`) : la fonction faisant correspondre au noeud `n` appartenant au CFG de la méthode `M`, le noeud d'entrée de la classe `C` (*Class Entry Vertex*) définissant la méthode `M`. Il s'agit de trouver l'antécédent du noeud d'entrée de `M` par l'arc de méthode de classe du JSysDG ;

3. γ(`n`) : la fonction faisant correspondre au noeud d'entrée `n` d'une classe `C`, le noeud d'entrée du constructeur de la classe. Il s'agit de trouver le successeur de `n` par un arc de méthode de classe tel que `ClassName=MethodName` ;
4. δ(`n`) : fonction faisant correspondre à un noeud d'entrée `n` de classe `C`, l'ensemble des noeuds $\mathtt{n}_i$ correspondant aux déclarations de variables globales de la classe `C`. Il s'agit de renvoyer les successeurs de l'ensemble des arcs de type attribut de classe ayant pour antécédent `n` ;
5. λ(`ClassName`) : fonction faisant correspondre à un identifiant `ClassName` le noeud d'entrée de la classe ayant pour nom `ClassName` ;
6. `extendsClauses(n)` : fonction faisant correspondre au noeud `n` d'entrée de la classe `C` l'ensemble des noms de classes étendues par `C` ;
7. `implementsClauses(n)` : fonction faisant correspondre au noeud `n` d'entrée de la classe `C` l'ensemble des noms d'interfaces implémentées par `C`.
8. $\mathtt{next}_{CFG}$`(n)` : fonction faisant correspondre au noeud `n` le noeud suivant dans le CFG de la méthode.

Enfin, on note :

1. `n.code` : le code de l'instruction associé au noeud `n`. Ce code est représenté sous la forme d'un arbre de syntaxe abstraite (AST) ;
2. et dans le cas où nc.code est une déclaration de variable (locale ou globale) :
   - `nc.id` l'identifiant de la variable déclarée ;
   - `nc.type` le type de la variable déclarée ;

La figure 5.8 représente de manière schématique une partie des notations introduites.

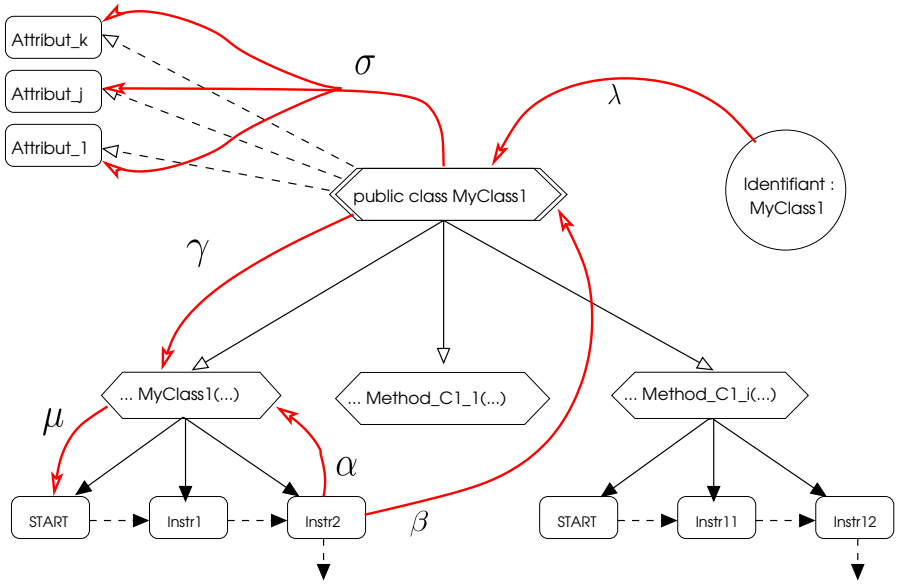


FIG. 5.8 – Notations intermédiaires.

5.4.4 Algorithme : analyse de la méthode `main()`

L'algorithme permettant l'analyse de la méthode `main()` est un algorithme récursif parcourant l'ensemble des instructions de la méthode. Durant ce parcours, l'environnement d'exécution de l'application (tables de symboles) est modifié afin de représenter l'ensemble des identifiants présents après l'initialisation de l'application.

Parallèlement, une table de widgets et une table d'écouteurs sont également mises à jour afin de représenter les diverses instances de widgets et d'écouteurs d'événements présents dans l'application. Ces deux tables, également utiles lors de l'analyse des méthodes d'écouteurs d'événements, permettent la construction du contexte du modèle B événementiel B_{Appl} : clauses SETS, PROPERTIES, VARIABLES, INVARIANT et INITIALISATION.

L'algorithme présenté dans cette section est simplifié. L'objectif n'est pas de présenter le traitement de l'ensemble des constructions Java, mais de donner un aperçu des diverses opérations effectuées sur le programme et de mettre en exergue la faisabilité d'une telle analyse.

L'algorithme présenté est constitué de quatre parties :

1. `analyze_main` : il s'agit du préambule de l'algorithme. Cet algorithme fait appel à `analyze_node` correspondant au coeur de l'algorithme récursif ;

2. `analyze_node` : algorithme parcourant les noeuds du CFG de la méthode `main()`. Cet algorithme fait appel à `analyze_constructor` et `analyze_method` qui, récursivement, font éventuellement appel à `analyze_node` ;

3. `analyze_constructor` : algorithme permettant de traiter des noeuds du type création d'instance faisant appel au constructeur de la classe ;

4. `analyze_method` : algorithme permettant de traiter des noeuds du type appel de méthode. L'algorithme de cette fonction, très similaire à `analyze_constructor`, n'est pas présenté dans ce qui suit.

Algorithme `analyze_main` :

Le listing de la figure 5.9 présente le début de l'algorithme permettant l'analyse de la méthode `main()`. Cet algorithme initialise les tables de symboles utilisées dans la suite de l'algorithme. Trois tables sont construites : la table des symboles principale `TabSymb`, une table `TabListener` pour répertorier l'ensemble des écouteurs d'événements de l'application et une table `TabWidgetsTree` pour répertorier les widgets créés lors de l'initialisation de la méthode. La pile `undo` permettant la gestion des redéfinitions est également créée. Enfin, l'algorithme appelle la méthode `analyze_node`. La méthode `analyze_node` a pour fonction d'analyser les instructions de la méthode `main()` l'une après l'autre. Ici `nc` (noeud courant) désigne le premier noeud (noeud START) de la méthode `main()`.

```
1  analyze_main()
2  begin_algo
3    /* findMainMethod renvoie un noeud le premier noeud START
4       de la méthode main  */
5    nc:=findFirstNodeOfMainMethod();
6    mktable(null,TabSymb);
7    mktable(null,TabListeners);
8    mktable(null,TabWidgetsTree);
9    undo:=new pile();
10         this:=null;
11   analyze_node(nc,this,TabSymb,TabListeners,TabWidgetsTree,undo);
12 end_algo
```

FIG. 5.9 – Algorithme d'analyse de la méthode `main()`.

Algorithme `analyze_node` :

La méthode `analyze_node` constitue le corps de l'analyse. La méthode `main()` est analysée instruction par instruction. Le noeud du graphe de dépendances de la méthode en cours d'analyse est donné par la variable `nc`. Le corps de la méthode est constitué d'une boucle dont la condition d'arrêt est définie par `nc`$\neq$ `null` (dernier noeud du CFG d'une méthode).

La première étape consiste à déterminer si l'instruction en cours d'analyse est un pattern reconnu. La méthode `findPattern` compare le noeud courant `nc` à l'ensemble des patterns appartenant à `WidgetsPatterns`. Parmi les patterns définis dans l'ensemble `WidgetsPatterns`, on trouve quatre types de patterns distincts :

1. *Pattern de Type 1* : pattern définissant la création d'un widget, par exemple `Jframe $$$$$ = new JFrame()` ;

2. *Pattern de Type 2* : pattern définissant une association widget/listener, c'est le cas des instructions de la forme `$$$$$.addXXXListener($$$$$)` ;

3. *Pattern de Type 3* : pattern définissant une association widget/widget (construction de la hiérarchie de widget de l'application). On trouve dans ce type un pattern qui est : `$$$$$.add($$$$$)` ;

4. *Pattern de Type 4* : pattern définissant l'appel d'un accesseur ou d'un modifieur de l'attribut d'un widget. Un tel pattern est de la forme : `$$$$$.NomMethode($$$$$)`;

La méthode `findPattern` renvoie l'identifiant du pattern reconnu, noté `pattern`, au sein de l'algorithme. Si `nc` ne correspond à aucun des patterns définis, le mot clef `null` est renvoyé.

```
1  analyze_node(nc,this,TabSymb,TabListeners,TabWidgetsTree,undo) =
2  begin_algo
3    while (nc≠STOP)
4      pattern:=findPattern(nc.code,WidgetsPatterns);
5
6      if (nc correspond à une entrée dans un bloc d'instructions)
7            undo.put(nc.id);
8      end_if
9
10     if (nc correspond à une sortie de bloc d'instructions)
11       remove();
12     end_if
13
14   if (pattern=null) /* Aucun pattern reconnu */
15           if (nc.code ≡ Type IdVar;) then {
16             insert(TabSymb, IdVar,[Typ,Val,null]);
17             nc:=next_CFG(nc);}
18           else if (nc.code ≡ IdVar=new constructeur(p1,...,pi);) then {
19             mktable(TabSymb, NewTable);
20             addObjectEnvironment(TabSymb,IdVar,NewTable);
21             analyseConstructeur(λ(constructeur),NewTable,IdVar,...);
22             nc:=next_CFG(nc);}
23           else if [...]
24           else if (nc.code ≡ STOP : noeud d'arrêt ) then {
25             id:=undo.pop();
26             remove();
27             nc:=null;}
28           end_if;
29   else /* nc.code est un pattern reconnu */
30           case : pattern de type 1 /* Création d'instance */
31             /* nc.code ≡ WidgetId=new WidgetType(p1,...,pi); */
32             mktable(tabSymb,newTable);
33             addObjectEnvironment(TabSymb,WidgetId,NewTable);
34             insert(TabWidgetsTree,WidgetId,[WidgetsType,null,...,newTable]);
35             analyze_constructor(λ(WidgetType),newTable,WidgetId,...);
36             nc:=next_CFG(nc);
37           case : pattern de type 2.
38             /* nc.code ≡ IdWidget.addXXXListener(IdListener); */
39             addListeners(TabWidgetsTree,IdWidget,(IdListener,XXXListener));
40             analyze_method(TabSymb,IdWidget,add);
41             nc:=next_CFG(nc);
42           case : pattern de type 3.
43             /* nc.code ≡ IdWidget1.add(IdXidget2); */
44             addComponent(TabWidgetsTree,IdWidget1,IdWidget2);
45             analyze_method(TabSymb,IdWidget1,add,...);
46             nc:=next_CFG(nc);
47           case : pattern de type 4.
48             /* nc.code ≡ IdWidget1.setAttributX(IdVal); */
49              if(AttributX est un attribut pertinent)
50              then addAttributX(TabWidgetsTree,IdWidget1,IdVal);
51                   analyze_method(TabSymb,IdWidget1,setAttributX,...);
52              else skip; /* Abstraction */
53             nc:=next_CFG(nc);
54           case : [...]
55   end_if
56   end_while
57 end_algo
```

FIG. 5.10 – Algorithme d'analyse de la méthode `main()`.

Les instructions suivantes assurent la gestion de la pile `undo` et les redéfinitions associées au niveau de la table des symboles principale `TabSymb`.

Cas d'une instruction ne correspondant à aucun pattern. Suivant la nature de l'instruction `nc` analysée, un traitement spécifique est réalisé. Dans le cas où aucun pattern

n'est reconnu, le traitement effectué dépend de la nature de l'instruction. Ici seulement trois cas sont présentés :

- 1^{er} cas : l'instruction est une déclaration de variable sans affectation de valeur. Le traitement consiste simplement à créer une nouvelle entrée dans la table des symboles `TabSymb`. Une fois cette opération effectuée, le noeud courant `nc` est modifié en prenant le prochain noeud présent dans le graphe de contrôle ;
- 2^{nd} cas : l'instruction est une création d'instance avec appel au constructeur de classe. Ne s'agissant pas d'une déclaration de variable, l'identifiant de l'instance appartient déjà à la table des symboles. Le traitement consiste à créer une nouvelle table `newTable` permettant de recenser les attributs de la nouvelle instance. La méthode `addObjectEnvironment` permet d'associer l'environnement défini par la nouvelle table à l'objet de la nouvelle instance. L'opération `analyze_constructor` permet alors de traiter le corps de la méthode du constructeur. La table `newTable` devient alors la table de référence lors de l'analyse. L'identifiant de l'instance créée est également passé en paramètre afin de pouvoir traiter des instructions faisant appel au mot clef *this* du langage Java. Une fois le constructeur de la classe analysée, le noeud courant devient le prochain noeud dans le CFG de la méthode en cours d'analyse ;
- 3^{me} cas : le noeud courant marque la fin de la méthode analysée. Dans ce cas le noeud courant est assigné à la valeur `null`, ce qui constitue la condition d'arrêt de la boucle.

Cas d'une instruction correspondant à un pattern. Dans le cas où un pattern est reconnu, le traitement dépend du type de pattern reconnu :

- 1^{er} cas : le pattern correspond à une création d'instance de widget (pattern de type 1). Dans ce cas, une nouvelle table `NewTable` liée à la table des symboles `TabSymb` est créée. L'environnement de l'objet `WidgetId` de la table `TabSymb` est associé à la nouvelle table créée. Enfin, le constructeur de la méthode est analysé. Dans ce cas, la classe analysée est une classe appartenant à la bibliothèque des abstractions de widgets (voir section 5.3) ;
- 2^{nd} cas : le pattern reconnu correspond à une association widget/écouteur. Dans ce cas l'attribut `listeners` de l'objet référencé par `IdWidget` dans la table `TabWidgetsTree` est modifié en ajoutant à l'ensemble des écouteurs associés au widget le couple `(IdListener, XXXListener)`. La méthode est ensuite analysée, puis la valeur du noeud courant est mise à jour ;
- 3^{me} cas : le pattern reconnu est une association widget/widget. L'appel à la méthode `addComponent` permet la mise à jour des attributs `parent` et `fils` des objets référencés par `IdWidget1` et `IdWidget2` dans la table `TabWidgetsTree`. La méthode est ensuite analysée, puis la valeur du noeud courant est mise à jour.
- 4^{me} cas : le pattern reconnu est une modification d'un attribut de widget relevant. La valeur de cet attribut est mise à jour dans la table `TabWidgetsTree` si et seulement si l'attribut est pertinent. Dans le cas contraire, l'instruction n'est pas prise en compte : il s'agit ici d'une abstraction. Une instruction du type `inputPanel.setColor(blue)`, par exemple, n'est pas analysée. Si l'instruction est d'intérêt pour l'analyse, alors la méthode est analysée (`analyze_method`) ce qui as-

sure la mise à jour de cette valeur dans la table des symboles globale `TabSymb`. Enfin, la valeur du noeud courant analysé est mise à jour.

Algorithme `analyze_constructor` et `analyze_method` :

Les algorithmes `analyze_constructor` et `analyze_method` réalisent le dépliage d'un appel de méthode contenu dans une instruction du code source. Cette opération d'*inlining* (cf. section 4.3.2) est réalisée de manière explicite en analysant les instructions du corps de la méthode appelée. `analyze_constructor` est un cas particulier de la méthode `analyze_method` qui nécessite des traitements spécifiques. On se limite ici à l'exposé de la méthode `analyze_constructor`.

La figure 5.11 présente le listing de cet algorithme.

```
1  analyze_constructor(nc,this,TabSymb,TabWidgets,TabListeners)=
2     /* nc : noeud d'entrée de classe du constructeur */
3     /* this= identifiant de l'objet en cours de construction */
4     /* TabSymb : table des symboles courante */
5
6  begin_algo
7    if(implementsClause(nc)≠∅)
8      for each InterfaceType ∈ implementsClause(nc)
9        if(InterfaceType ∈ ListenersSet)
10       then insert(TabListeners,this,[InterfaceType,this]);
11       end_if
12     end_for
13   end_if
14
15   if(extendsClause(nc)≠∅)
16     for each ClassType ∈ extendsClause(nc)
17       if(ClassType ∈ WidgetsSet)
18       then  insert(TabWidgets,this,[ClassType,...,this]);
19       end_if
20       addType(TabSymb,this,ClassType);
21       analyze_constructor(λ(ClassType),this,...);
22     end_for
23   end_if
24
25   if (σ(nc)≠∅)
26   then  /* la classe possède des déclarations globales */
27        for each tempNode ∈ σ(nc))
28         insert(TabSymb,tempNode.id,[tempNode.type,null,null]);
29        end_for
30        analyze_node(μ(γ(nc)),this,newTab,TabWidgets,...);
31   else analyze_node(μ(γ(nc)),this,TabSymb,TabWidgets,...);
32   end_if
33 end_algo
```

FIG. 5.11 – Algorithme d'analyse d'un constructeur de classe.

L'analyse consiste dans un premier temps à analyser les clauses `implements` et `extends` de la classe correspondant au constructeur en cours d'analyse.

Analyse de la clause `implements`. Le traitement de la clause `implements` consiste à déterminer quelles sont les interfaces *listener* étendues par la classe. Un nouvel objet est inséré dans la table `TabListeners` si l'interface appartient à l'ensemble `ListenerSet` (ensemble des interfaces *listeners*).

Analyse de la clause `extends`. Le traitement de la clause `extends` consiste à déterminer quelles sont les classes étendues par la classe en cours d'analyse. Une nouvelle entrée est créée dans la table `TabWidgets` si la classe étendue est une classe appartenant à l'ensemble des classes de widgets (`WidgetsSet`). Dans le cas contraire, l'attribut `Type` de l'objet en cours d'analyse (`this`) est modifié en ajoutant le type de la classe étendue (méthode `addType`) puis le constructeur de la classe étendue est analysé en réalisant un appel récursif à la méthode `analyze_constructor`.

Analyse des déclarations de variables globales. Une fois les clauses de la classe analysées, une analyse des variables globales de la classe (attributs de la classe) est effectuée. Pour chaque variable globale, une nouvelle entrée dans la table des symboles courante est créée. Enfin les instructions de la méthode constructrice sont analysées en appelant `analyze_node` sur le noeud μ(`nc`) qui correspond au premier noeud de la méthode (noeud `START`).

Remarques. Il faut re-préciser ici que l'algorithme présenté est simplifié par souci de concision. Notamment, l'analyse d'un appel de méthode est plus complexe puisqu'elle nécessite la gestion des paramètres formels et réels de la méthode. La gestion de ces appels de méthode dans un graphe de dépendances système a été présentée en section 2.3 de l'état de l'art.

5.5 Analyse des méthodes d'écouteurs et construction du modèle B_{Appl}

Les deux dernières étapes de l'analyse statique (cf. figure 5.1) consistent à analyser chacune des méthodes d'écouteurs présente dans l'application et à construire le modèle B_{Appl}.

L'analyse des méthodes d'écouteurs conduit à la construction d'un modèle intermédiaire. Ce modèle intermédiaire correspond à un canevas de modèle B événementiel dans lequel les instructions ou expressions de ce modèle sont encore représentées par un Arbre de Syntaxe Abstraite Java.

La construction du modèle B_{Appl} consiste à construire un AST B événementiel à partir du modèle intermédiaire et de la table des symboles construite au cours de l'analyse de la méthode `main()`.

5.5.1 Analyse des méthodes d'écouteurs

L'analyse d'une méthode d'écouteur est divisée en 4 sous-étapes :

– *Première étape* : les tables des symboles construites au cours de l'analyse de la méthode `main()` sont utilisées afin de déterminer l'ensemble des widgets associés à la méthode d'écouteur en cours d'analyse. Cette information est nécessaire afin de déterminer la garde du premier événement B représentant la méthode d'écouteur (conditions d'activation) ;

- *Deuxième étape* : le graphe de dépendances de la méthode d'écouteur est parcouru afin de regrouper les instructions de la méthode sous la forme d'un ensemble de blocs d'instructions. Concrètement chaque bloc correspond à une branche de contrôle de la méthode. Dans ces blocs, les instructions non pertinentes sont supprimées (abstraction) ;
- *Troisième étape* : les blocs d'instructions sont analysés afin de construire le modèle intermédiaire représentant les événements du modèle B_{Appl}. Le regroupement des instructions par blocs permet la gestion des variables de contrôle du modèle qui assurent le bon séquencement des événements. Pendant cette analyse, les dépendances entre les instructions d'un bloc sont calculées afin de déterminer le type d'événement B à introduire ;
- *Quatrième étape* : enfin, de nouvelles variables de contrôle sont introduites et les gardes des événements sont éventuellement modifiées afin d'assurer que les événements représentant une méthode d'écouteur ne soient pas entrelacés avec des événements représentant une autre méthode d'écouteur.

La première étape est évidente et il n'est pas nécessaire de détailler sa réalisation. La deuxième étape ressemble dans les grandes lignes à l'algorithme permettant l'analyse de la méthode `main()` (utilisation de pattern pour la reconnaissance des structures de contrôle et des instructions pertinentes). Cette étape n'est pas présentée dans la suite. Enfin, la quatrième étape est très proche de l'analyse effectuée lors de la troisième étape dont l'algorithme est présenté dans ce qui suit.

Représentations intermédiaires

Deux structures de données sont définies.

Représentation des blocs d'instructions de la méthode. Une méthode d'écouteur est constituée d'un ensemble de blocs d'instructions ordonnées par un ensemble de structures de contrôle.

Un bloc d'instructions `b` sera représenté par un objet possédant un type (`b.type`). On distingue seulement quatre types distincts dans cette présentation : `BLOCS`, `SEQ_BLOC`, `IF_BLOC` et `WHILE_BLOC`.

- Un objet `B` de type `BLOCS` permet de représenter un bloc de haut niveau composé d'un ensemble de blocs d'un des quatre types définis. Cet ensemble de blocs est implémenté par une pile dont le sommet de pile représente le premier bloc apparaissant dans `B`. Un arbre hiérarchique de blocs est constitué. Les feuilles de cet arbre sont des blocs élémentaires du type `SEQ_BLOC`. Du point de vue de l'implémentation, l'accès à cette pile de blocs sera représenté par l'attribut `B.blocs`.
- Un objet `b` de type `SEQ_BLOC` est un bloc constitué uniquement d'instructions séquentielles. Ces instructions, sous la forme d'AST Java, sont empilées dans une pile `b.pile` où l'instruction du sommet de pile correspond à la dernière instruction du bloc.
- Un objet `BB` de type `IF_BLOC` est constitué de deux blocs de type `BLOCS` : `b.blocsIf1` et `b.blocsIf2`. Ces deux blocs représentent les deux branches d'instructions sous le contrôle de la conditionnelle. L'accès à l'expression de la condition de contrôle est obtenu par l'attribut `BB.cond` de l'objet.

– Enfin, un objet `B` de type `WHILE_BLOC` possède un attribut de type `BLOCS` accessible par `B.blocs`. La condition de contrôle de la boucle est donnée par l'attribut `B.cond` de l'objet.

Représentation : modèle intermédiaire. On doit disposer d'une structure de données permettant de représenter le modèle B événementiel. Ce même modèle pourra être utilisé pour construire un modèle équivalent au modèle B événementiel dans un autre langage d'entrée (ex : un modèle NuSMV tel que présenté en section 4.4). Cette structure de données est calquée sur la représentation d'un modèle B événementiel. Ainsi, un objet de type `model` dispose des attributs `setsClause`, `variablesClause`, `invariantsClause`, `eventsClause`, (...), correspondant à l'ensemble des clauses d'un modèle B.

Les types des attributs d'un modèle utilisés dans l'algorithme sont les suivants :

– `variablesClause` est une liste de chaînes de caractères. Chaque chaîne de caractères représente l'identifiant d'une variable du modèle ;
– `invariantsClause` est une liste d'expressions représentées par des AST B ;
– `initialisationClause` est une liste de substitutions B représentées par un AST B événementiel ;
– l'attribut `eventsClause` est représenté par une liste ordonnée d'événements. Un événement `evt` est un objet possédant un attribut `type` permettant de définir la nature de l'événement considéré : `evt.type` $\in \{anyTyp, selectTyp\}$, où `anyTyp` représente un événement de type indéterministe, et `selectTyp` un événement de type déterministe. Enfin, un événement possède 4 attributs représentant les clauses possibles de l'événement : `anyClause`, `whereClause`, `selectClause`, `thenClause`. Chacun de ces attributs est représenté par une liste d'expressions Java représentées sous forme d'AST. Lors de la construction du modèle final, ces AST élémentaires devront être transformés en AST B événementiels.

5.5.2 Algorithme d'analyse des blocs d'instructions composant une méthode d'écouteur

La figure 5.12 expose le corps simplifié d'un algorithme réalisant l'analyse de blocs d'instructions. Cet algorithme est un algorithme récursif travaillant sur les blocs contenus dans une méthode d'écouteur. À ce stade de l'analyse, les instructions non pertinentes (modifications d'attributs de widgets non pertinentes) ont été abstraites.

```
1  analyze_blocs(BB,VI,model)=
2     /* VI : ensembles des variables d'intérêt */
3     /* BB : objet de type BLOCS */
4     /* model : modèle en cours de construction */
5  begin_algo
6     if (BB.blocs.size()>1)
7     then
8       int initSize := BB.blocs.size();
9       int indexVC := BB.blocs.size()-1;
10      string VC := generateNewId();
11      model.variablesClause.add(VC);
12      model.invariantsClause.add([VC∈1..indexVC])
13      model.initialisationClause.add([VC:=indexVC])
14      while(BB ≠ ∅)
15        b:=BB.blocs.pop();
16        if (b.type=SEQ_BLOC) then
17            evt:=analyze_seq_bloc(b.pile,VI);
18            if (evt.type=anyTyp)
19              then evt.anyClause.add([VC=index]);
20              else evt.selectClause.add([VC=index]);
21            end_if;
22            if(BB.size()=0)
23              then evt.thenClause.add([VC=initSize-1]);
24              else evt.thenClause.add([VC=VC-1]);
25            end_if;
26            model.eventsClause.add(evt);
27         else if (b.type=IF_BLOC) then
28            BB1:=b.blocsIF1; BB2:=b.blocsIF2;
29            modelIf1:= new Model(); modelIf2:= new Model();
30            analyze_listener(BB1,VI,modelIf1);
31            analyze_listener(BB2,VI,modelIf2);
32            for each evt_i ∈ modelIf1.eventsClause ⋃ modelIf1.eventsClause
33              if (evt.type=anyTyp)
34                then evt_i.anyClause.add([VC=index]);
35                else evt_i.selectClause.add([VC=index]);
36              end_if;
37            end_for;
38            if(BB.size()=0)
39             then lastEvt(modelIf1.eventsClause).thenClause.add([VC=initSize-1]);
40                  lastEvt(modelIf2.eventsClause).thenClause.add([VC=initSize-1]);
41             else lastEvt(modelIf1.eventsClause).thenClause.add([VC=VC-1]);
42                  lastEvt(modelIf2.eventsClause).thenClause.add([VC=VC-1]);
43            end_if;
44            firstEvt(modelIf1.eventsClause).selectClause.add([b.cond]);
45            firstEvt(modelIf1.eventsClause).selectClause.add([¬(b.cond)]);
46            model:=merge(model,modelIf1);
47            model:=merge(model,modelIf2);
48        else if (b.type=WHILE_BLOC) then
49            [...]
50        else if (b.type=BLOCS) then
51            [...]
52        end_if;
53        indexVC:=indexVC-1;
54      end_while;
55    else /* ¬(BB.blocs.size()>1) */
56      /* idem mais sans gestion de variables de contrôle */
57      /* appel récursif à analyze_bloc dans le cas où le type du bloc est
58         BLOCS, WHILE_BLOC ou IF_BLOC */
59      /* création d'un événement déterministe dans le cas où le type du
60         bloc est SEQ_BLOC */
61      [...]
62    end_if;
63      [...]
64 end_algo;
```

FIG. 5.12 – Algorithme : Analyse d'une méthode d'écouteur.

Algorithme `analyze_blocs`

Paramètres d'entrée de l'algorithme. L'algorithme prend en entrée trois paramètres :

1. un objet de type BLOCS `BB`. Au début de l'analyse la pile associée à cet objet (`BB.blocs`) représente l'ensemble des blocs composant la méthode d'écouteur ;
2. l'ensemble des variables d'intérêt du système $\mathcal{VI}$. Comme défini au chapitre précédent, cet ensemble est constitué des variables représentant les widgets de l'application et des variables globales appartenant au contrôleur de dialogue. Les variables du contrôleur de dialogue permettent de garder en mémoire une partie de l'état du système nécessaire au contrôle de l'application ;
3. enfin, l'entrée `model` correspond au modèle en cours de construction. Au début de l'analyse, le modèle est vide (l'ensemble des clauses est vide).

Description de l'algorithme. Dans le cas où la pile `BB.blocs` possède plus d'un élément, plusieurs événements seront nécessaires à la représentation de la méthode d'écouteur. L'ordre de déclenchement de ces événements doit correspondre à l'ordre d'exécution des instructions de la méthode. Une variable de contrôle est nécessaire pour satisfaire cette condition : un nouvel identifiant de variable `VC` est créé. Cette variable est ajoutée à la clause VARIABLES du modèle. L'expression définissant le type de `VC` est ajoutée à la clause INVARIANT et la substitution définissant sa valeur initiale est ajoutée à la clause INITIALISATION.

Une expression encadrée de crochets `[Exp]` représente la transformation de cette expression en AST après avoir évalué la valeur des identifiants apparaissant dans l'expression. Par exemple, `[VC` $\in$ `1..indexVC]` correspond à l'AST B événementiel de l'expression `vc0` $\in$ `1..5` si VC="vc0" et `indexVC=5`.

Une fois la variable de contrôle introduite, les blocs d'instructions de la pile `BB` sont analysés l'un après l'autre au sein d'une boucle `while`. Le bloc `b` situé en sommet de pile est retiré de la pile. Le traitement de ce bloc dépend alors du type du bloc `b`. Seulement deux cas représentatifs sont détaillés dans la figure 5.12 :

1. si `b.type=SEQ_BLOC` alors le bloc associé à `b` est un bloc composé d'instructions en séquence. Dans ce cas, un événement est construit en analysant ces instructions par le biais de la méthode `analyze_seq_bloc` dont l'algorithme est détaillé par la suite. La garde et le corps de l'événement sont modifiés en ajoutant une condition et une substitution relative à la variable de contrôle. Le dernier événement (`BB.size()=0`) est déclenché lorsque la variable de contrôle `VC` est égale à `0`, et le corps de cet événement réinitialise la variable à `initSize-1`. Enfin, l'événement construit est ajouté au modèle.
2. si `b.type=IF_BLOC`, deux nouveaux modèles `modelIf2` et `modelIf1` sont créés. Les deux branches `b.blocsIF1` et `b.blocsIF2` associées à la structure de contrôle conditionnelle sont analysées en appelant récursivement la méthode `analyze_blocs`. Les gardes de l'ensemble des événements appartenant au modèles `b.blocsIF1` et `b.blocsIF2` sont alors modifiées en ajoutant une condition liée à la variable de contrôle dont les événements dépendent. Les corps des deux derniers événements

représentant les blocs `b.blocsIF1` et `b.blocsIF2` sont également modifiés en ajoutant une instruction décrémentant la valeur de la variable de contrôle. Enfin, les gardes des premiers événements représentant les blocs `b.blocsIF1` et `b.blocsIF2` sont modifiées en ajoutant l'expression booléenne contrôlant la structure `if then else`. Les gardes introduites respectent les règles de traduction présentées en section 4.3. Une fois ces modifications effectuées, les modèles `modelIf2` et `modelIf1` sont fusionnés dans le modèle principal. Cette fusion correspond simplement à une union des différentes clauses des modèles.

En fin de boucle la valeur de la variable de contrôle est décrémentée.

Dans le cas où la pile de blocs analysée est constituée d'un seul bloc, le traitement est semblable au cas précédent mis à part qu'il ne nécessite pas l'introduction d'une variable de contrôle.

Enfin, il convient de rappeler que la dernière étape d'une analyse de méthode d'écouteur (absente du listing présenté) consiste à ajouter les conditions d'activations liées à la méthode analysée. Ces conditions d'activations sont ajoutées à la garde des premiers événements modélisant la méthode d'écouteur.

```
1  analyze_seq_bloc(b,VI)=
2  /* Précondition :
3     - b est une pile d'instructions
4     - l'instruction en haut de pile correspond à la dernière instruction d'un
5       bloc séquentiel */
6  begin_algo
7   evt:= new EVT();
8   evt.type:=selectTyp; /* par défaut */
9   W:=∅;
10  R:=∅;
11  while (b ≠ ∅)
12    instr:=b.pop();
13    r:=read_set(instr);
14    w:=write_var(instr);
15    if ( {w} ∩ R ≠ ∅)
16      /* Dépendance de données */
17      evt.type:=anyTyp;
18      String IdVar:=generateNewId();
19      evt.anyClause.add(IdVar);
20      evt.whereClause.add([IdVar=exp(instr)]);
21      evt.whereClause.add([IdVar ∈ typeOf(w)]);
22      for each op ∈ evt.thenClause
23        if (w ∈ read_var(op))
24        then replace(op,w,IdVar);
25        end_if;
26      end_for;
27      if (w ∈ VI)
28      then evt.thenClause.add(w=IdVar);
29      else skip;
30      end_if;
31    else /* w ∩ R = ∅ */
32     if ( write_var(instr) ∈ VI)
33       then evt.thenClause.add(instr);
34       else skip;
35    end_if;
36    W:=W∪w;
37    R:=R∪r;
38  end_while;
39    return evt;
40 end_algo
```

FIG. 5.13 – Algorithme : analyse de bloc. Traitement des dépendances.

Algorithme `analyze_seq_bloc`

L'algorithme `analyze_seq_bloc`, présenté par le listing de la figure 5.13, assure la construction d'un événement à partir d'un bloc composé d'un ensemble d'instructions séquentielles. Ces instructions séquentielles sont groupées dans le corps d'un événement B par la substitution ||. Ce regroupement des instructions nécessite une analyse des dépendances de données telle que présentée en section 4.3.4.

Paramètres de l'algorithme. L'algorithme `analyze_seq_bloc` accepte deux entrées : b une pile d'instructions et $\mathcal{VI}$ l'ensemble des variables d'intérêt de l'application. Cet algorithme renvoie un événement `evt`.

Présentation de l'algorithme. Dans un premier temps, un nouvel événement vide `evt` est créé. Par défaut le type de cet événement est fixé à `selectTyp` (événement déterministe). Deux ensembles initialement vides sont également créés : `W` et `R`. L'ensemble `W` correspond à l'ensemble des variables modifiées (*write-set*) et l'ensemble `R` à l'ensemble des variables lues (*read-set*) par les instructions précédemment analysées.

La suite de l'algorithme est constituée d'une boucle permettant d'analyser une à une les instructions présentes dans la pile. Tant que la pile n'est pas vide, l'instruction `instr` en sommet de pile est retirée. L'ensemble des variables lues et la variable modifiée par `instr` sont déterminés en explorant l'AST associé à cette instruction : cette opération est réalisée par les fonctions `read_set` et `write_var`[1].

Dépendance de données. Dans le cas d'une dépendance de données entre l'instruction `instr` et les instructions précédemment analysées, il est nécessaire de représenter la séquence d'instructions par un événement indéterministe. Le type de l'événement `evt` est par conséquent modifié et fixé à `anyTyp` (cf. section 4.3). Une nouvelle variable locale possédant un identifiant unique est introduite dans la clause ANY de l'événement.

Afin de typer cette variable dans la clause WHERE de l'événement, le type de la variable `w` est calculé par l'intermédiaire de la fonction `typeOf()`. Cette fonction détermine le type de `w` soit en explorant la table des symboles créée si `w` est une variable d'intérêt de l'application, soit en explorant le graphe de dépendances de la méthode d'écouteur en cours d'analyse. En effet, si la variable `w` n'est pas une variable d'intérêt, il s'agit nécessairement d'une variable locale de la méthode[2] : l'analyse du graphe de dépendances permet de retrouver la déclaration de cette variable et par conséquent de déterminer son type.

L'étape suivante consiste à modifier toute occurrence de `w` dans l'expression des substitutions `op` de l'événement `evt` par `IdVar`. La fonction `replace()` permet cette modification par manipulation de l'AST de `op`.

Si `w` est une variable pertinente, l'instruction `instr` est ajoutée à la clause THEN de l'événement. L'instruction est supprimée dans le cas contraire (variable locale).

[1] L'instruction `instr` analysée de la forme `Var=Exp(v1,...,vk)`. Dans ce cas, l'ensemble des variables modifiées est le singleton `{var}`.

[2] Les instructions modifiant les variables "non pertinentes" qui ne sont pas des variables locales ont été abstraites au début de l'analyse de la méthode d'écouteur.

Aucune dépendance de données. Dans le cas où les instructions précédemment analysées ne dépendent pas de l'instruction `instr` en cours d'analyse, et que la variable modifiée par `instr` est une variable d'intérêt, alors l'instruction est simplement ajoutée à la clause THEN de l'événement. Dans le cas contraire, l'instruction n'est pas prise en compte[3].

Enfin, les ensembles `W` et `R` sont mis à jour avant de traiter l'instruction suivante.

Les algorithmes présentés ont été simplifiés et ne présentent pas l'ensemble des cas possibles. Une variable appartenant au *read-set* d'une instruction peut par exemple ne pas être définie dans le bloc en cours d'analyse, mais dans un autre bloc de la méthode d'écouteur : il est par conséquent nécessaire de réaliser une étude des dépendances entre les différents blocs d'instructions.

5.5.3 Construction du modèle B_{Appl}

Le modèle intermédiaire correspond quasiment au modèle B événementiel B_{Appl} que l'on souhaite obtenir. La construction du modèle B_{Appl} consiste à construire l'AST du modèle puis à générer le texte correspondant à cet arbre (opération de "*pretty-print*"). L'AST est construit en explorant les tables de symboles construites lors de l'analyse de la méthode `main()` et du modèle intermédiaire construit au cours de l'analyse des méthodes d'écouteurs.

5.6 Conclusions

Ce chapitre a permis de mettre en évidence la possibilité de construire un modèle de dialogue B événementiel par analyse statique de code source Java-Swing. Cette analyse consiste à explorer le graphe de dépendances de l'application Java et à effectuer un ensemble d'abstractions et d'interprétations en fonction des instructions rencontrées.

L'analyse statique d'un code source Java-Swing n'est pas complètement automatique : elle nécessite de la part du développeur d'abstraire une partie de la bibliothèque Swing du langage. Cette étape préalable peut être coûteuse en temps, mais le travail réalisé pour une analyse particulière est réutilisable pour les analyses suivantes : l'investissement diminue ainsi au fur et à mesure des analyses. Enfin, les abstractions sont directement réalisées dans le langage Java ce qui facilite le travail du développeur qui peut travailler en utilisant un langage qu'il connaît.

Enfin, il est à remarquer que la correction de l'abstraction réalisée dépend principalement de la correction des abstractions effectuées par le développeur en Java. Si ces abstractions sont correctes, la correction de l'abstraction finale (modèles B_{Appl} et Nu_{Appl}) découle quasi directement des techniques utilisées pour l'analyse. En effet, toutes les opérations d'analyse statique se formalisent dans la théorie de l'Interprétation Abstraite [Cousot & Cousot, 1976], [Cousot & Cousot, 1977], [Cousot & Cousot, 1992], [Cousot & Cousot, 1999], [Cousot, 2002].

Un outil prototype basé sur les principes et algorithmes présentés dans ce chapitre a été élaboré et est en cours de développement. Cet outil, codé en Java, exploite les outils

[3] Il s'agit éventuellement d'un code mort.

JavaCC et JJTree permettant la construction de d'analyseur syntaxique et la manipulation d'AST[4].

[4]Renseignements concernant JavaCC et JJTree : https://javacc.dev.java.net/doc/JJTree.html

Chapitre 6

Contribution à la validation de l'utilisabilité d'une application Java/Swing

"Les ordinateurs sont inutiles. Ils ne savent que donner des réponses."
Pablo Picasso

6.1 Introduction

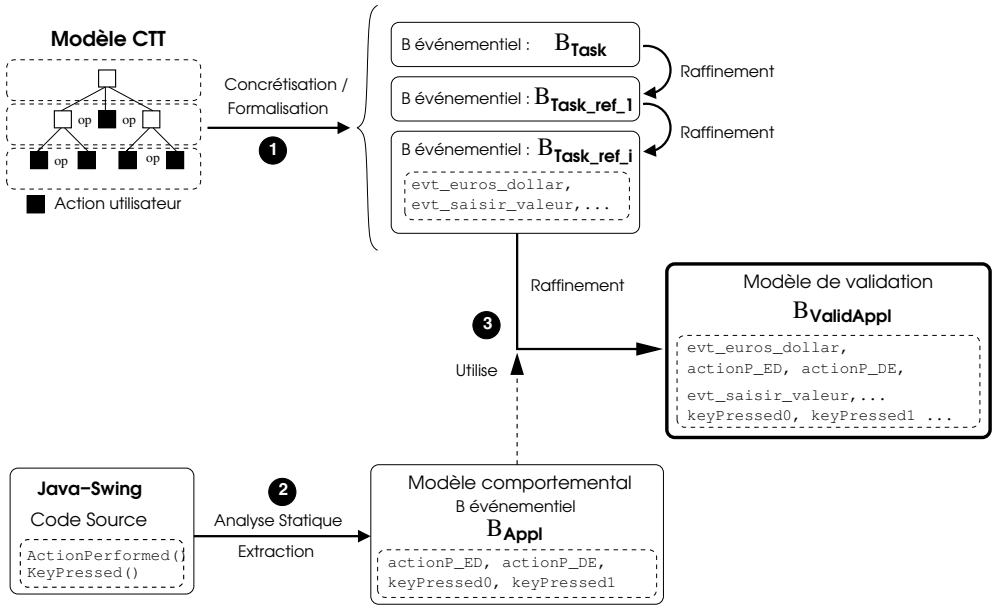


Fig. 6.1 – Principe du processus de validation d'une application interactive.

Le dernier chapitre de ce manuscrit propose une approche permettant la validation d'une partie de l'utilisabilité d'une application Java-Swing.

6.1.1 Principes du processus de validation B événementiel

La figure 6.1 présente les principes de l'approche de validation. L'objectif est de vérifier que l'IHM construite respecte les spécifications d'utilisabilité du système. Ces spécifications sont représentées par un modèle de tâches CTT et décrivent en compréhension les scénarii d'usage attendus de l'application. L'approche de validation consiste à montrer que les structures d'interaction encodées dans le programme s'inscrivent bien dans ces scénarii d'usage.

Les structures d'interaction de l'application ont été préalablement extraites par analyse du code source de l'application et encodées dans un modèle B événementiel B_{Appl} (Fig.6.1,❷).

Pour vérifier que l'application se conforme au modèle de tâches CTT, l'approche consiste à montrer que les scénarii d'interaction de l'application sont inclus dans les scénarii décrits par le modèle CTT. Cette preuve d'inclusion est délicate du fait de l'hétérogénéité des modèles utilisés et du manque de formalisation des modèles CTT. Une solution consiste à formaliser le modèle de tâches CTT en exploitant les règles de traduction proposées par [Aït-Ameur *et al.*, 2005b] (Fig.6.1,❶).

Prouver que l'application se conforme au modèle de tâches CTT revient alors à prouver que le modèle concret B_{Appl} est un raffinement correct du modèle de tâches formalisé B_{Task} (Fig.6.1,❸). Ce dernier raffinement du modèle B_{Task} introduit de nouveaux événements issus du modèle B_{Appl} qui modélisent, lors de l'exécution, la réaction du système en réponse à une action utilisateur. La construction automatique de ce raffinement nécessite une étape préalable de concrétisation du modèle de tâches (Fig.6.1,❶) afin d'assurer une jonction correcte au sein du modèle de validation $B_{ValidAppl}$ entre :

- les événements issus du modèle de tâches et traduisant les actions de l'utilisateur d'une part,
- et les événements issus de B_{Appl} représentant la réaction du système d'autre part.

Les différentes étapes de validation (concrétisation/formalisation et construction du modèle de validation $B_{ValidAppl}$) sont présentées dans les sections 6.2 et 6.2 respectivement.

La présentation du langage CTT est présentée en détail en annexe A.

La section 6.4 montrera qu'il est également possible de valider l'application vis-à-vis de modèles CTT en utilisant la méthode NuSMV. Cette formalisation met en oeuvre la technique de vérification exhaustive de modèles. Elle permet d'opérer une comparaison des deux techniques de preuve (*theorem-proving* et *model-checking*).

6.1.2 Présentation du modèle de tâches CTT de l'étude de cas

La figure 6.2 présente le modèle de tâches CTT considéré comme une spécification du convertisseur Euros/Dollars.

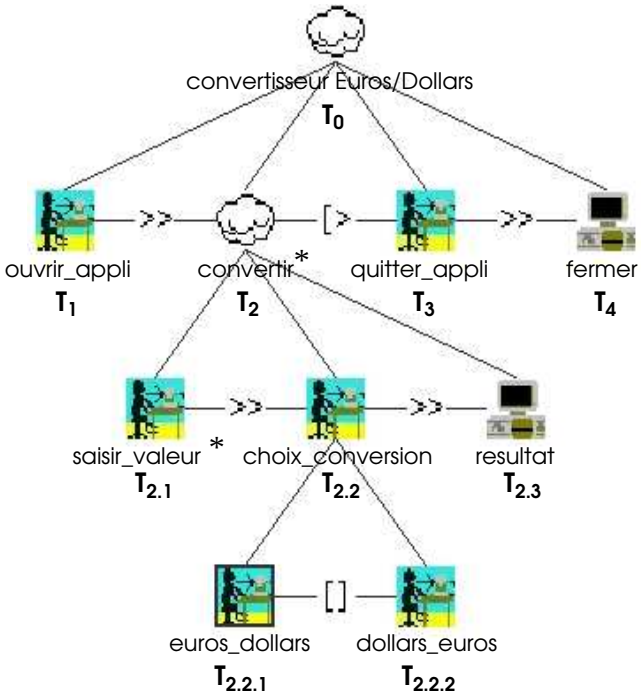


FIG. 6.2 – Modèle de tâche CTT du convertisseur Euros/Dollars.

Une fois l'application lancée (tâche T_1), l'utilisateur peut effectuer un nombre indéfini ($*$) de conversions (tâche T_2). La tâche de conversion peut être désactivée ($[>$) à tout moment par l'utilisateur en quittant l'application (tâche T_3) : cette action active alors ($>>$) la fermeture du système (tâche T_4).

La tâche abstraite de conversion est décomposée en trois sous-tâches. L'utilisateur doit tout d'abord saisir une valeur (tâche $T_{2.1}$). Cette opération de saisie peut être effectuée un nombre indéfini de fois comme l'indique l'opérateur d'itération ($*$). Suite à cette saisie ($>>$), l'utilisateur doit choisir un sens de conversion : il s'agit de la décomposition de la tâche $T_{2.2}$ en deux sous-tâches $T_{2.2.1}$ et $T_{2.2.2}$ reliées par l'opérateur de choix ($[]$). Une fois ce choix réalisé, le système doit afficher le résultat de la conversion (tâche $T_{2.3}$).

6.2 Concrétisation et formalisation B du modèle CTT

Afin d'établir de manière automatique le modèle de validation $B_{ValidAppl}$ par raffinement du modèle de tâches, une étape de concrétisation et de formalisation du modèle CTT est réalisée.

6.2.1 Concrétisation

Principe de la concrétisation. La concrétisation du modèle de tâches est à la charge du développeur et consiste à enrichir les feuilles de l'arbre CTT qui représentent des actions de l'utilisateur par des informations concrètes issues de l'application. Trois informations sont ajoutées : le type et le nom du *widget* sur lequel l'utilisateur agit (`wid_type` et `wid_name`) et le type d'action effectuée (`action`). Cette étape permet par la suite d'automatiser le couplage entre les modèles B_{ApplM} et B_{Task}. Il convient ici de préciser que les

informations de concrétisation ne sont pas représentées dans le modèle de tâches formalisé B_{Task}. La figure 6.3 présente un exemple de concrétisation des tâches utilisateur $T_{2.1}$, $T_{2.2.1}$ et $T_{2.2.2}$ du modèle CTT.

FIG. 6.3 – Concrétisation des actions utilisateur du modèle CTT.

6.2.2 Formalisation B du modèle CTT

Formalisation des opérateurs temporels de CTT.

La formalisation du modèle CTT en B événementiel repose sur les règles de traduction définies par [Aït-Ameur *et al.*, 2005b]. Ces travaux proposent des règles de traduction pour chaque opérateur de l'algèbre de processus associée à CTT. Cette traduction peut être effectuée de manière automatique au sein d'un outil.

Ce processus de formalisation utilise le principe de raffinement de la méthode B événementiel. Les figures 6.4 et 6.5 présentent la description de deux de ces règles de transformation des opérateurs CTT : celle de l'opérateur d'activation ">>" et celle de l'opérateur de choix "[]". La tâche T_0 désigne la tâche mère décomposée en une expression correspondant à la grammaire CTT. L'ensemble var_i désigne l'ensemble des variables caractérisant l'événement de haut niveau T_0. De nouvelles variables peuvent être introduites dans un raffinement s'il est nécessaire d'observer de nouveaux éléments lors de la décomposition de la tâche T_0 de l'arbre de tâches. S_0 est la substitution qui exprime le changement d'état des variables du processus T_0.

Opérateur d'activation ">>". Soit $T_0 ::= T_1 >> T_2$ pour l'activation de T_1 suivie de T_2 (séquence). La traduction en B événementiel (Fig. 6.4) est donnée par un modèle possédant deux nouveaux événements $EvtT_1$ et $EvtT_2$ correspondant aux tâches T_1 et T_2.

La traduction utilise un variant V_{Etat} qui est initialisé à la valeur 2. Ce variant est strictement décroissant. $EvtT_1$ est déclenché lorsque sa garde G_1 est vraie. La substitution S_1 est alors réalisée et le variant est décrémenté.

La condition de non-blocage représentée par la disjonction des gardes est donnée dans la clause ASSERTIONS du modèle. L'ensemble var_j définit l'ensemble des variables du raffinement. Elles sont liées aux variables du modèle abstrait par l'intermédiaire de l'invariant de collage $\mathtt{J}(var_i, var_j)$. L'événement $EvtT_0$ termine l'activation des deux processus : les événements concrets redonnent le contrôle à l'abstraction.

```
1  REFINEMENT Enabling_{T0}_ref
2  REFINES  T0
3  INVARIANT
4      J(var_i, var_j)  ∧ V_Etat  ∈ {0,1,2}
5  ASSERTIONS
6      G0  ⇒ ((V_Etat=2  ∧ G1)  ∨
7                    (V_Etat=1  ∧ G2)  ∨
8                    (V_Etat=0  ∧ G'0))
9  VARIANT
10   V_Etat
11 INITIALISATION
12   V_Etat  :=2  ||
13 EVENTS
14  EvtT1=            EvtT2=            EvtT0=
15  SELECT            SELECT            SELECT
16   V_Etat=2  ∧ G1     V_Etat=1  ∧ G2     V_Etat=0  ∧ G'0
17  THEN              THEN              THEN
18   V_Etat:=1  || S1   V_Etat:=0  || S2   S'0
19  END;              END;              END;
```

FIG. 6.4 – Traduction de l'opérateur CTT ">>" en B événementiel.

Opérateur de choix "[]". L'expression $T_0 ::= T_1 \; [] \; T_2$ définit un choix non déterministe entre les tâches T_1 et T_2. La traduction est obtenue par un modèle B événementiel composé de trois nouveaux événements gardés : $EvtChoixT_1$ et $EvtChoixT_2$ et $Evt_{InitChoix}$ (Fig.6.5).

```
1  REFINEMENT Choix_{T0}_ref
2  REFINES  T0
3  INVARIANT
4      J(var_i, var_j)  ∧ V_EtatChoix  ∈ {0,1,2,3}
5  ASSERTIONS
6      G0  ⇒ ((∃(p).(p  ∈ {1,2}  ∧  V_EtatChoix=3))  ∨
7                    (V_EtatChoix=1  ∧ G1)  ∨
8                    (V_EtatChoix=2  ∧ G2)  ∨
9                    (V_EtatChoix=0  ∧ G'0))
10 VARIANT
11   V_EtatChoix
12 INITIALISATION
13   V_EtatChoix  :=3  ||
14 EVENTS
15  Evt_InitChoix=          EvtChoixT1=          EvtChoixT2=          EvtT0=
16  ANY p WHERE             SELECT               SELECT               SELECT
17   V_EtatChoix=3  ∧ G_init   V_EtatChoix=1  ∧ G1   V_EtatChoix=2  ∧ G2   V_Etat=0  ∧ G'0
18    ∧  p  ∈  {1,2}         THEN                 THEN                 THEN
19  THEN                      V_EtatChoix:=0  || S1   V_EtatChoix:=0  || S2   S'0
20     V_EtatChoix:=p         END;                 END;                 END;
21  END;
```

FIG. 6.5 – Traduction de l'opérateur CTT "[]" en B événementiel.

Un variant $V_{EtatChoix}$ est introduit et initialisé à 3. L'événement $Evt_{InitChoix}$ est le premier événement déclenché parmi les trois événements introduits. La substitution de cet événement assigne de manière indéterministe la valeur 1 ou 2 à $V_{EtatChoix}$. Suivant la valeur de la garde de chacun des événements, l'un des deux événements $EvtChoixT_1$ et $EvtChoixT_2$ est alors déclenché. Ces deux événements font décroître immédiatement la valeur du variant à 0, interdisant ainsi à l'événement concurrent de prendre le contrôle. Enfin l'événement raffiné $EvtChoixT_0$ termine le processus T_0.

Les règles de transformation définies dans [Aït-Ameur *et al.*, 2005b] couvrent l'ensemble des constructions (opérateur et informations de tâches) du langage CTT pour la modélisation de tâches utilisateurs. Ces règles définissent une sémantique formelle qui utilise la sémantique à base de traces du B événementiel. Cette sémantique correspond au point de vue des auteurs : ils donnent une interprétation formelle à ces opérateurs.

La décomposition en B événementiel d'un modèle de tâche CTT exploite ces règles de transformation pour construire l'arbre de tâches. À l'arbre de décomposition CTT correspondent alors un modèle et des raffinements B. Le processus de raffinement est effectué jusqu'à ce que les sous-tâches (tâches terminales) correspondent à des événements utilisateurs. Le nombre de raffinements dépend donc de la profondeur de l'arbre de tâches.

Modification du modèle CTT : formalisation des tâches "application".

Le modèle de tâches de l'étude de cas (Fig. 6.1) est constitué de tâches de type "application" () : il s'agit "*de tâches effectuées complètement par le système et qui sont utilisées pour rendre compte de l'état du système*". Concrètement, toute action utilisateur sur l'interface d'une application active une réaction du système qui modifie potentiellement l'état d'activation du système. Dans le cas d'une application Java-Swing, ces réactions sont encodées par les méthodes d'écouteurs d'événements. Par conséquent, il semblerait juste que toute tâche utilisateur élémentaire ()(feuille de l'arbre de tâches) soit immédiatement suivie d'une tâche "application" par l'intermédiaire d'un opérateur d'activation (>>). Pourtant, la plupart des modèles CTT rencontrés ne présentent pas une telle spécificité.

Dans le cadre de cette étude, la présence d'une tâche "application" dans un modèle de tâches CTT sera interprétée comme l'*attente d'un état observable particulier de l'IHM suite à une action utilisateur*. Suivant cette acception, une tâche "application" dénote une post-condition que l'on souhaite vérifier. La prise en compte de cette post-condition est effectuée en modifiant le modèle de tâches initial. Dans le modèle obtenu, l'ensemble des tâches "application" sont supprimées et les tâches utilisateur précédant une tâche "application" sont étiquetées par l'expression d'une post-condition. La figure 6.6 présente le modèle de tâches modifié. Ces post-conditions seront formulées suivant la syntaxe B événementiel. Toute variable intervenant dans cette post-condition doit être une variable du modèle B_{Appl} extrait de l'application par analyse statique.

Le modèle modifié (Fig.6.6) présente deux post-conditions :

- $\forall w.(w \in$ `WIDGETS` $\Rightarrow$ `enabled(w)=false` $\wedge$ `visible(w)=false`) : cette post-condition exprime que la fermeture de l'application rend tous les widgets non visibles et inactifs ;
- `visible(output)=true` $\wedge$ `setText(BW_output)=full`: cette deuxième post-condition exprime que le système doit afficher le résultat, c'est à dire que le champ de saisie `output` est visible et non vide.

Ces deux post-conditions sont ajoutées au modèle B événementiel lors du couplage entre le modèle CTT formalisé B_{Task} et le modèle de l'application B_{Appl} (cf. sous-section 6.3). Ce couplage est réalisée lors de la construction du modèle de validation $B_{ValidAppl}$ obtenu par raffinement du modèle B_{Task}.

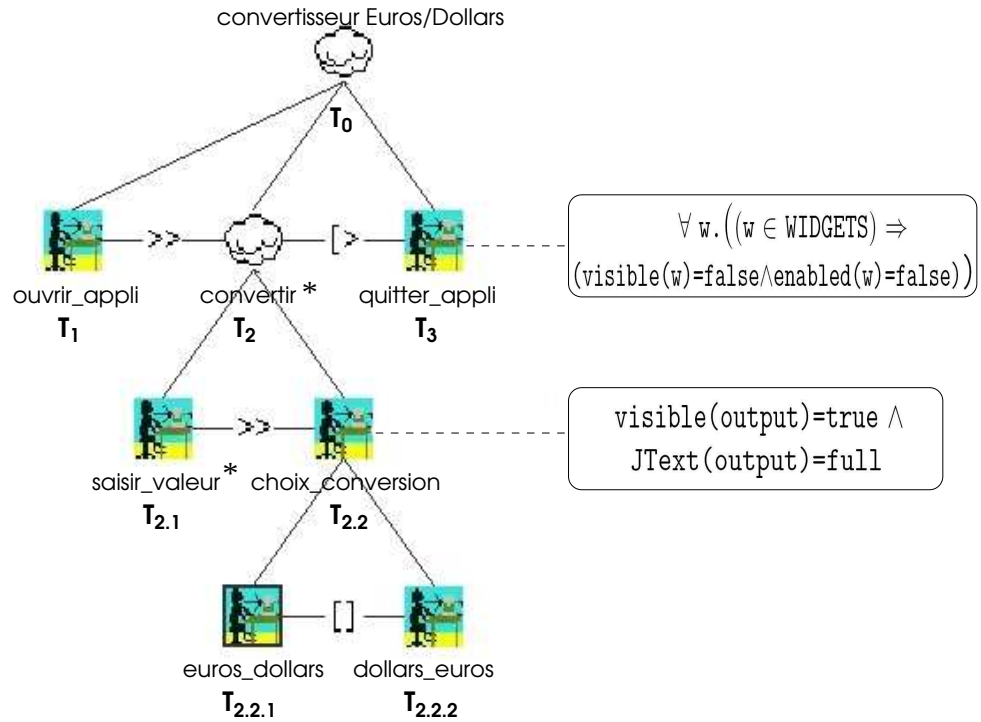


FIG. 6.6 – Modèle de tâche CTT modifié : "formalisation" des tâches de type application.

6.2.3 Application à l'étude de cas

Cette section présente la formalisation du modèle de tâches CTT de l'étude de cas. Dans cet exemple, les règles de traduction définies par [Aït-Ameur *et al.*, 2005b] ont été légèrement modifiées afin de réduire le nombre d'événements du modèle. La formalisation du modèle de tâches présentée en figure 6.6 est obtenue à partir d'un modèle B abstrait B_{Task} et de deux raffinements successifs de ce modèle : B_{Task_ref1},B_{Task_ref2}.

Premier modèle : B_{Task}. Le premier modèle correspond à la formalisation de la première décomposition de l'arbre de tâches donnée par l'expression $T_0 = T_1 >> T_2 * [> T_3$. Ce modèle (figure 6.7) est composé de trois variables :

- `run` indique si l'application est en cours d'exécution (`true`) ou non (`false`) ;
- $\mathtt{V}_{open}$ est une variable de contrôle permettant de traduire le séquencement des tâches T_1 et T_2 ;
- $\mathtt{V}_{star1}$ est une variable de contrôle qui représente l'itération de la tâche T_2.

Dans l'état initial l'application est fermée, la variable $\mathtt{V}_{open}$ prend la valeur `1` et $\mathtt{V}_{star1}$ prend sa valeur ($:\in$) dans l'ensemble $\mathbb{N}$ de telle sorte que $\mathtt{V}_{star1} \geq 1$. $\mathtt{V}_{open}+\mathtt{V}_{star1}$ est un variant du modèle.

Le modèle B_{Task} est constitué de trois événements :

1. Evt_T_1 (ouverture de l'application) est déclenché lorsque l'application n'est pas en cours d'exécution. La substitution de cet événement fait décroître la variable $\mathtt{V}_{open}$ et modifie la valeur de la variable `run` donnant ainsi le contrôle aux autres événements ;
2. Evt_T_2 représente une tâche itérative ($*$). Cet événement peut être déclenché tant que $\mathtt{V}_{star1} > 1$. Evt_T_2 peut être interrompu à n'importe quel moment de son exécution

par Evt_T_3. Tant que V_{star1}>1 ces deux événements sont en concurrence puisque la garde de Evt_T_2 implique la garde de Evt_T_3 : $G_{Evt\_T_2} \Rightarrow G_{Evt\_T_3}$. La substitution de cet événement fait décroître la variable V_{star1} et donc le variant ;

3. Evt_T_3 peut interrompre l'itération de l'événement Evt_T_2 à tout moment de son exécution. Si la tâche T_2 n'est pas interrompue, la valeur de V_{star1} finira pas prendre la valeur 1 : dans ce cas la tâche T_3 est exécutée. L'événement Evt_T_3 correspond à la fermeture de l'application : la variable run prend alors la valeur false et V_{star1} la valeur 0 marquant ainsi la fin des scénarii d'interaction.

Enfin, une assertion qui exprime le non-blocage de l'application[1] est ajoutée au modèle. Cette condition est exprimée par la disjonction des gardes des événements à laquelle s'ajoute une formule caractérisant l'arrêt du système : (run=false $\wedge$ V_{open}=0 $\wedge$ V_{star1}=0). Cette assertion assure que tous les événements du système peuvent être déclenchés.

La preuve de cette assertion nécessite l'ajout d'un invariant exprimant que si la variable V_{open} vaut 1 alors la variable de contrôle V_{star1} est nécessairement supérieure ou égale à 1. Il s'agit d'une contrainte structurelle du modèle.

```
MODEL B_Task

VARIABLES
run, /* run -> indique si l'application est en cours d'exécution */
V_open, /* V_open -> variable de contrôle (séquencement des événements) */
V_star1 /* V_star1 -> variable de contrôle (tâche itérative) */
INVARIANT
 /* invariants de typage et de collage */
  run ∈ BOOL  ∧  V_open ∈ 0..1  ∧ V_star1  ∈ ℕ ∧
 /* autre invariant (invariant structurel)  */
  (V_open=1  ⇒ V_star1 ≥ 1)
ASSERTIONS
 /* propriété de non-blocage -> disjonction des gardes des événements */
   (run=false  ∧  V_open=1)  ∨ (run=true  ∧ V_open=0  ∧ V_star1>1)  ∨
   (run=true  ∧  V_open=0  ∧ V_star1 ≥ 1)  ∨ (run=false  ∧ V_open=0  ∧ V_star1=0)
VARIANT
  V_open  + V_star1
INITIALISATION
/* V_star1 prend sa valeur de manière indéterministe dans ℕ tel que V_star1 ≥ 1 */
  V_open:=1  || run:=false  || V_star1:∈ (V_star1  ∈ ℕ ∧ V_star1 ≥ 1)
EVENTS

/* Evt_T1 -> ouverture de l'application */
/* Evt_T2 -> conversion */
/* Evt_T3 -> fermeture de l'application */

 Evt_T1=                Evt_T2=                Evt_T3=
 SELECT                 SELECT                 SELECT
  run=false ∧            V_open=0 ∧             V_open=0  ∧ run=true  ∧
  V_open=1               run=true ∧             V_star1 ≥ 1
 THEN                    V_star1>1              THEN
  V_open:=0  ||          THEN                    run:=false  ||
  run:=true              V_star1:=V_star1-1      V_star1:=0
 END;                   END;                   END;
```

FIG. 6.7 – Formalisation du modèle de tâche : premier modèle B_{Task}.

Premier raffinement : B_{Task_ref1}. Le premier raffinement correspond à la décomposition de la tâche T_2 : $T_2 = T_{2.1} *>> T_{2.2}$. La tâche $T_{2.1}$ est effectuée un nombre indéfini de

[1] cf. Chap.3, section 3.2

fois (itération $*$) et est suivie en séquence par la tâche $T_{2.2}$. La règle de traduction proposée par [Aït-Ameur *et al.*, 2005b] introduit quatre nouveaux événements pour ce type de décomposition : trois pour la représentation de la tâche itérative $T_{2.1}$ (initialisation de la boucle, boucle et sortie de boucle) et un événement pour la représentation de $T_{2.2}$. Le bon séquencement est alors assuré par deux variables de contrôle, l'une assurant l'itération de l'événement $T_{2.1}$ et l'autre assurant le séquencement des deux tâches.

La traduction présentée par la figure 6.8 propose d'introduire seulement deux événements ($Evt\_T_{2.1}$ et $Evt\_T_{2.2}$) et une seule variable de contrôle $\mathtt{V}_{star2}$. Ce type de traduction permet de diminuer le nombre d'événements et de variables utilisées et par conséquent de diminuer le nombre d'obligations de preuve à décharger et d'assurer une meilleure lisibilité du modèle. $\mathtt{vars}_{B_{Task}}$ désigne dans ce raffinement l'ensemble des variables déclarées dans le modèle abstrait B_{Task}.

MODEL B_{Task_ref1} **REFINES** B_{Task}
VARIABLES
$\mathtt{vars}_{B_{Task}}$, $\mathtt{V}_{star2}$ /* V_{star2} -> variable de contrôle (tâche itérative) */
INVARIANT
/* invariants de typage et de collage */
[...] $\wedge$ $\mathtt{V}_{star2} \in \mathbb{N} \wedge$
ASSERTIONS
/* propriété de non-blocage */
$(\mathtt{G'}_{Evt\_T_1} \vee \mathtt{G'}_{Evt\_T_2} \vee \mathtt{G'}_{Evt\_T_3} \vee (\mathtt{run=false} \wedge \mathtt{V}_{open}\mathtt{=0} \wedge \mathtt{V}_{star1}\mathtt{=0}))$
$\Rightarrow$
$(\mathtt{G}_{Evt\_T_1} \vee \mathtt{G}_{Evt\_T_2} \vee \mathtt{G}_{Evt\_T_{2.1}} \vee \mathtt{G}_{Evt\_T_{2.2}} \vee \mathtt{G}_{Evt\_T_3} \vee (\mathtt{run=false} \wedge \mathtt{V}_{open}\mathtt{=0} \wedge \mathtt{V}_{star1}\mathtt{=0}))$
VARIANT
$\mathtt{V}_{star2}$ /* Evénements convergents = $Evt\_T_{2.1}$ et $Evt\_T_{2.2}$ */
INITIALISATION
/* V_{star2} prend sa valeur de manière indéterministe dans $\mathbb{N}$ tel que $V_{star2} \geq 2$ */
[...] $\parallel$ $\mathtt{V}_{star2} :\in (\mathtt{V}_{star2} \in \mathbb{N} \wedge \mathtt{V}_{star2} \geq \mathtt{2})$
EVENTS

/* $Evt\_T_{2.1}$ -> saisie d'une valeur dans le champ de texte input (tâche itérative) */
/* $Evt\_T_{2.2}$ -> opération de conversion */

$Evt\_T_{2.1}$=
SELECT
$\mathtt{G'}_{Evt\_T_2} \wedge$
$\mathtt{V}_{star2} \geq \mathtt{2}$
THEN
$\mathtt{V}_{star2} := \mathtt{V}_{star2}\mathtt{-1}$
END;

$Evt\_T_{2.2}$=
SELECT
$\mathtt{G'}_{Evt\_T_2} \wedge$
$\mathtt{V}_{star2}\mathtt{=1}$
THEN
$\mathtt{V}_{star2} := \mathtt{0}$
END;

Evt_T_2=
SELECT
$\mathtt{G'}_{Evt\_T_2} \wedge$
$\mathtt{V}_{star2}\mathtt{=0}$
THEN
$\mathtt{V}_{star1} := \mathtt{V}_{star1}\mathtt{-1} \parallel$
$\mathtt{V}_{star2} :\in (\mathtt{V}_{star2} \in \mathbb{N} \wedge \mathtt{V}_{star2} \geq \mathtt{2})$
END;

Evt_T_3=
SELECT
$\mathtt{G'}_{Evt\_T_3} \wedge \mathtt{V}_{star2} \neq \mathtt{0}$
THEN
$\mathtt{S'}_{Evt\_T_3}$
END;

FIG. 6.8 – Formalisation du modèle de tâche : premier raffinement B_{Task_ref1}.

La variable $\mathtt{V}_{star2}$ est un variant du système pour les deux événements introduits (appelés événements convergents au sein de la plate-forme Rodin). Sa valeur initiale est un entier naturel supérieur ou égal à 2. Dans ce qui suit, $\mathtt{G'}_{Nom\_Evt}$ désigne la garde de l'événement Nom_Evt dans l'abstraction du raffinement en cours : dans la figure 6.8, par exemple, $\mathtt{G'}_{Evt\_T_2}$ désigne la garde de l'événement Evt_T_2 du modèle B_{Task}. De la même

manière $\mathtt{S'}_{Nom\_Evt}$ désigne la substitution de l'événement Nom_Evt dans l'abstraction du raffinement en cours.

- L'événement $Evt\_T_{2.1}$ est déclenché lorsque la garde $\mathtt{G'}_{Evt\_T_2} \wedge \mathtt{V}_{star2} \geq 2$ est vraie. La substitution associée à l'événement fait décroître le variant $\mathtt{V}_{star2}$.
- L'événement $Evt\_T_{2.2}$ est déclenché lorsque la garde $\mathtt{G'}_{Evt\_T_2} \wedge \mathtt{V}_{star2}=1$ est vraie. Dans ce cas le variant est décrémenté et prend la valeur 0 ce qui rend le contrôle à l'événement abstrait Evt_T_2.
- Lorsque l'événement Evt_T_2 est déclenché, il réinitialise la valeur du variant (entier naturel supérieur ou égal à 2). Cette réinitialisation est nécessaire du fait que l'événement abstrait Evt_T_2 représente également une tâche itérative : la séquence de tâches $T_{2.1} *>> T_{2.2}$ doit par conséquent pouvoir être déclenchée un nombre indéfini de fois.

Comme pour tous les raffinements qui suivent, une assertion assure le non-blocage du système.

La garde de l'événement Evt_T_3 est modifiée. T_3 doit pouvoir désactiver la tâche T_2 quel que soit son état. La concrétisation de la tâche abstraite T_2 est donnée par les événements $Evt\_T_{2.1}$ et $Evt\_T_{2.2}$: suivant les gardes de ces événements, $Evt\_T_{2.1}$ et $Evt\_T_{2.2}$ sont bien en concurrence avec la tâche de désactivation T_3. Si T_3 ne prend pas le contrôle avant l'exécution de l'événement $Evt\_T_{2.2}$, ceci signifie que la tâche abstraite n'a pas été désactivée puisque toutes les tâches concrétisant la tâche abstraite sont réalisées. Dans ce cas, l'événement de désactivation T_3 doit laisser l'abstraction Evt_T_2 reprendre le contrôle. Afin d'assurer ce cas de figure la condition $\mathrm{V}_{star2} \neq 0$ est ajoutée à la garde de Evt_T_3.

La garde de l'événement Evt_T_3 sera modifiée de manière analogue lors de chaque nouvelle décomposition du modèle.

```
1  MODEL B_Task_ref2 REFINES B_Task_ref1
2  VARIABLES
3   vars_B_Task_ref1 , /* Ensemble des variables de B_Task_ref1 */
4   V_choice /* V_choice -> variable de contrôle */
5  INVARIANT
6   /* invariants de typage et de collage */
7     [...]  ∧  V_choice  ∈ 0..3  ∧
8  ASSERTIONS
9   /* propriété de non-blocage */
10    (G'_Evt_T1 ∨ G'_Evt_T2 ∨ G'_Evt_T2.1 ∨ G'_Evt_T2.2 ∨  ...)
11     ⇒
12    (  ...  ∨ G_Evt_T2.1 ∨ G_Evt_InitChoice ∨ G_Evt_T2.2.1 ∨ G_Evt_T2.2.2 ∨ G_Evt_T2.2 ∨  ...)
13
14 VARIANT
15    V_choice /* Evénements convergents = Evt_InitChoice , Evt_T2.2.1 , Evt_T2.2.2 */
16 INITIALISATION
17    [...]  ||   V_choice:=3
18 EVENTS
19
20  Evt_InitChoice=          Evt_T2.2.1=          Evt_T2.2.2=          Evt_T2.2=
21  ANY var1                 SELECT               SELECT               SELECT
22  WHERE                     G'_Evt_T2.2 ∧        G'_Evt_T2.2 ∧        G'_Evt_T2.2 ∧
23   G'_Evt_T2.2 ∧            V_choice=1          V_choice=2           V_choice=0
24   var1 ∈ {1,2} ∧          THEN                 THEN                 THEN
25   V_choice=3               V_choice:=0          V_choice:=0          V_choice:=3  || S'_Evt_T2.2
26  THEN                     END;                 END;                 END;
27   V_choice:=var1
28  END;
```

FIG. 6.9 – Formalisation du modèle de tâche : second raffinement B_{Task_ref2}.

Second raffinement : B_{Task_ref2}. Le second raffinement réalise la décomposition de la tâche abstraite $T_{2.2}$ qui correspond à un choix ([]) entre les tâches concrètes $T_{2.2.1}$ et $T_{2.2.2}$: $T_{2.2} = T_{2.2.1}$ [] $T_{2.2.2}$. La représentation de ce type de décomposition a été présentée dans la sous-section précédente. Le raffinement obtenu en appliquant la règle de traduction proposée par [Aït-Ameur *et al.*, 2005b] est présenté par la figure 6.9.

Notons que lorsque l'événement abstrait $Evt\_T_{2.2}$ reprend le contrôle, celui-ci réinitialise le variant V_{choice} : la raison de cette réinitialisation est identique au cas vu précédemment. En effet, la tâche T_2 est une tâche itérative : il est donc nécessaire de réinitialiser la variable de contrôle V_{choice}. Cette réinitialisation est effectuée par l'événement abstrait, i.e. par l'événement que $Evt_{InitChoice}$, $Evt\_T_{2.2.1}$ et $Evt\_T_{2.2.2}$ raffinent (c.a.d $Evt\_T_{2.2}$).

6.3 Validation de l'application vis-à-vis du modèle CTT : B événementiel

Prouver que l'application se conforme au modèle de tâches CTT revient à prouver que le modèle concret B_{Appl} est un raffinement correct du modèle de tâches formalisé B_{Task}. Il s'agit donc de construire un raffinement $B_{ValidAppl}$ du modèle B_{Task}[2] qui concrétise les événements correspondant aux tâches utilisateurs en introduisant les réactions de l'application encodées dans le modèle B_{Appl}.

Le raffinement d'un événement B caractérisant une action utilisateur élémentaire exploite les informations issues de la concrétisation du modèle de tâches. Ces informations permettent d'établir le lien entre un événement appartenant au modèle B_{Task} et caractérisant une tâche utilisateur et les événements B du modèle B_{Appl} qui correspondent à la réaction de l'application à cette action utilisateur.

Les post-conditions introduites lors de la suppression des tâches de type "application" du modèle sont également utilisées dans le modèle de validation $B_{ValidAppl}$ (clause INVARIANT).

La figure 6.10 présente le modèle de tâches CTT de l'étude de cas et la concrétisation des tâches élémentaires de ce modèle par des tâches correspondant aux réactions de l'interface. Les tâches de concrétisation correspondent aux événements B issus du modèle B_{Appl} qui modélisent l'exécution des méthodes de d'écouteurs (méthodes déclenchées lors d'une action utilisateur).

[2]En réalité il s'agit d'un raffinement du dernier modèle obtenu lors de la formalisation du modèle de tâches : dans notre cas il s'agit du modèle B_{Task_ref2}.

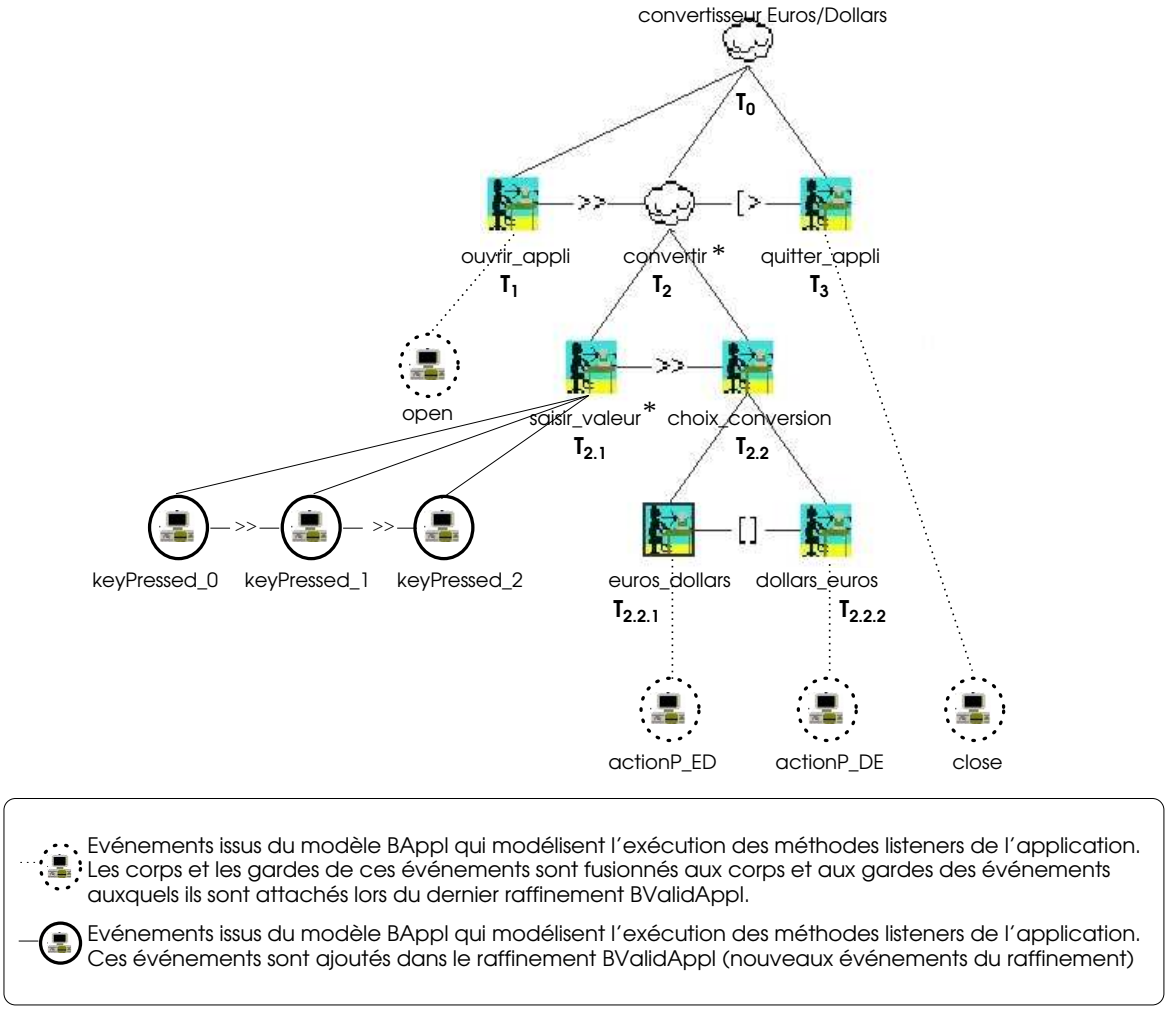


FIG. 6.10 – Modèle de tâches CTT de l'étude de cas et concrétisation des tâches élémentaires par des événements issus du modèle B_{Appl}.

6.3.1 Construction du modèle de validation $B_{ValidAppl}$: application à l'étude de cas

La figure 6.11 présente une partie du modèle de validation $B_{ValidAppl}$ obtenu par raffinement du modèle B_{Task_ref2}. Seule la concrétisation des événements $Evt\_T_{2.1}$, $Evt\_T_{2.2.1}$ et $Evt\_T_{2.2.2}$ est ici présentée.

Le modèle $B_{ValidAppl}$ possède une visibilité sur les ensembles et constantes du modèle B_{Swing} (modélisation de la bibliothèque Java/Swing).

Concrétisation de $Evt\_T_{2.1}$. L'événement $Evt\_T_{2.1}$ correspond à la tâche `saisir_valeur` du modèle de tâche. Cette tâche élémentaire est concrétisée par une action de type `KeyPressed` sur le champ de saisie `input` de l'interface. La méthode d'écouteur `KeyPressed` est modélisée au sein du modèle B_{Appl} par les événements concrets $keyPressed\_0$, $keyPressed\_1$ et $keyPressed\_2$: ces événements apparaissent donc dans ce nouveau raffinement afin de représenter la réaction du système. La variable V_{kp} (déjà

présente dans le modèle B_{Appl}[3]) devient le nouveau variant du système pour ces trois événements. Lorsque ce variant prend la valeur `0`, l'événement abstrait $Evt\_T_{2.1}$ récupère le contrôle et réinitialise ce variant à la valeur `2`.

Les gardes des événements $keyPressed\_0$, $keyPressed\_1$ et $keyPressed\_2$ du modèle B_{Appl} sont modifiées dans ce raffinement par l'ajout de la condition G'$_{Evt\_T_{2.1}}$[4] (garde de contexte).

Concrétisation des événements $Evt\_T_{2.2.1}$ et $Evt\_T_{2.2.2}$. Les événements $Evt\_T_{2.2.1}$ et $Evt\_T_{2.2.2}$ correspondent au choix de l'utilisateur d'effectuer une conversion d'Euros en Dollars (tâche `euros_dollars`) ou de Dollars en Euros (tâche `dollars_euros`). Ces tâches élémentaires sont concrétisées par une action de type `ActionPerfomed` sur les boutons `ED` et `DE` de l'interface. La réaction de l'application suite à l'une de ces deux opérations de conversion est modélisée par les événements $actionP\_ED$ et $actionP\_DE$ du modèle B_{Appl}.

Dans ce cas il n'est pas nécessaire d'introduire de nouveaux événements dans le raffinement : les événements concrets $actionP\_ED$ et $actionP\_DE$ sont directement fusionnés avec les événements $Evt\_T_{2.2.1}$ et $Evt\_T_{2.2.2}$. Les gardes de ces deux derniers événements sont modifiées par l'ajout des conditions G'$_{actionP\_ED}$ et G'$_{actionP\_DE}$ respectivement. Ces gardes correspondent à la formalisation des conditions d'activation des méthodes d'écouteurs que les événements $actionP\_ED$ et $actionP\_DE$ modélisent. La fusion consiste également à mettre en parallèle les jeux d'instructions issus des deux événements fusionnés.

Enfin, la garde de l'événement $Evt_{InitChoice}$ est également modifiée en introduisant la conjonction des deux gardes G'$_{actionP\_ED}$ et G'$_{actionP\_DE}$.

"Post-conditions" : prise en compte des tâches de type application. Les tâches de type "application" du modèle CTT sont prises en compte au sein du modèle $B_{ValidAppl}$ par l'ajout d'invariants de collage dans la clause INVARIANT du modèle. Ces invariants de collage permettent d'établir un lien entre les variables abstraites et concrètes du système. Dans l'exemple présenté, deux invariants de collage sont ajoutés au modèle pour exprimer les post-conditions portant sur les tâches T_3 et $T_{2.2}$.

La post-condition portant sur la tâche T_3 est exprimée de la manière suivante : l'état obtenu après l'exécution la substitution de l'événement Evt_T_3 implique que les widgets de l'application sont invisibles et inactifs. Ceci est effectivement assuré par la nouvelle substitution de l'événement T_3.

La post-condition portant sur la tâche $T_{2.2}$ exprime que si la garde de l'événement abstrait $Evt\_T_{2.2}$ est vraie (G'$Evt\_T_{2.2}$) alors le champ de saisie `output` est visible et non vide (un résultat est affiché). On utilise ici la garde G'$Evt\_T_{2.2}$ pour représenter l'état caractérisant la fin de la tâche $T_{2.2}$. Ceci est justifié par le fait que cet événement prend nécessairement le contrôle si tous les événements le concrétisant sont exécutés. La garde de cet événement caractérise donc bien l'état correspondant à la fin de la réalisation de la tâche $T_{2.2}$ (post-condition de la tâche).

[3]Voir la modélisation de la méthode d'écouteur d'événement *KeyPressed* de l'étude de cas en sous-section 4.3.5, figure 4.7.

[4]Rappel : G'$_{Evt\_T_{2.1}}$ correspond ici à la garde de l'événement $Evt\_T_{2.1}$ dans le précédent raffinement, en l'occurrence dans B_{Task_ref2}.

```
MODEL B_ValidAppl REFINES B_Task_ref2
SEES B_Swing /* Introduction de la modélisation de la bibliothèque Swing */
                /* Contexte de l'application */
VARIABLES
/* Union des variables des modèles B_Task_ref2 et B_Appl */
 Vars_B_Task_ref2 , Vars_B_Appl , V_kp
INVARIANT
 /* invariants de typage */
  Inv(B_Appl) ∧ Inv(B_Task_ref2) ∧ V_kp ∈ 0..2 ∧
 /* invariants de collage */
 ((run=false) ⇒ (visible(pc)=false)) ∧ ((run=true) ⇒ (visible(pc)=true)) ∧
 ((V_open=0 ∧ run=true) ⇒(enabled(input)=true ∧ visible(input)=true)) ∧ [...]
 /* invariant de collage -> ''post-condition'' de l'événement T_2.1 */
  ∧ ((run=false) ⇒ (∃w.(w ∈ WIDGETS ⇒ visible(w)=false ∧ enabled(w)=false)))
 /* invariant de collage -> ''post-condition'' de l'événement T_2.2 */
  ∧ (G'Evt_T_2.2 ⇒(visible(output)=true ∧ Jtext(output)=full))
ASSERTIONS
 /* propriété de non-blocage */
   (... ∨ G'Evt_T_2.2.1 ∨ G'Evt_T_2.2.2 ∨ G'Evt_T_2.2 ∨ ...)
    ⇒
   ( ... ∨ G_keyPressed_0 ∨ G_keyPressed_1 ∨ G_keyPressed_2 ∨ ...)
VARIANT
  V_kp /* Evénements convergents = keyPressed_0, keyPressed_1, keyPressed_2 */
INITIALISATION
  [...] ‖ V_kp:=2
EVENTS
  [...]
 keyPressed_0=
 ANY tt WHERE
  G'Evt_T_2.1 ∧
  G'keyPressed_0 ∧
  tt ∈ STRING ∧
  V_kp=2
 THEN
  V_kp:=1 ‖
  JText(input):=tt
 END;

 keyPressed_1=
 SELECT
  G'Evt_T_2.1 ∧
  G'keyPressed_1 ∧
  V_kp=1
 THEN
  S'keyPressed_1 ‖
  V_kp:=0
 END;

 keyPressed_2=
 SELECT
  G'Evt_T_2.1 ∧
  G'keyPressed_2 ∧
  V_kp=1
 THEN
  S'keyPressed_2 ‖
  V_kp:=0
 END;

 Evt_T_2.1=
 SELECT
  G'Evt_T_2.2 ∧
  V_kp=0 ∧
 THEN
  S'Evt_T_2.1 ‖
  V_kp:=2
 END;

 Evt_InitChoice=
 ANY var1
 WHERE
  G'InitChoice ∧
  G'actionP_DE ∧
  G'actionP_ED ∧
 THEN
  S'InitChoice
 END;

 Evt_T_2.2.1=
 SELECT
  G'Evt_T_2.2.1 ∧
  G'actionP_ED
 THEN
  S'Evt_T_2.2.1 ‖
  S'actionP_ED
 END;

 Evt_T_2.2.2=
 SELECT
  G'Evt_T_2.2.2 ∧
  G'actionP_DE
 THEN
  S'Evt_T_2.2.2 ‖
  S'actionP_DE
 END;

 Evt_T_2.2=
 SELECT
  G'Evt_T_2.2
 THEN
  S'Evt_T_2.2
 END;

[...]

 Evt_T_3=
 SELECT
  /* G'Evt_T_3 */
  run=TRUE ∧ visible(pc)=true ∧ V_open=0 ∧ V_star2≠0 ∧
  V_choice/ = 0 ∧ V_kp/ = 0
  /* G'close -> garde de l'événement close du modèle B_Appl */
  visible(pc)=true ∧ enabled(pc)=true
 THEN
  run:=false ‖ V_star1:=0 /* S'Evt_T_3 */
  enabled:=enabled⩤{pc ↦ false,input ↦ false,output ↦ false,ED ↦ false,
                    DE ↦ false,...} ‖
  visible:=enabled⩤{pc ↦ false,input ↦ false,output ↦ false,ED ↦ false,
                    DE ↦ false,...}
 END;
```

FIG. 6.11 – Validation : raffinement $B_{ValidAppl}$.

L'annexe 2 de ce mémoire présente le détail des modèles B_{Swing}, B_{Appl} et $B_{ValidAppl}$.

6.3.2 Résultats obtenus et preuves de propriétés

Le tableau 6.1 présente un récapitulatif du nombre d'obligations de preuve générées et prouvées (automatiquement ou interactivement) pour l'ensemble des modèles et raffinements construits pour la validation de l'application.

Nom du modèle/ raffinement	**Nombre d'OP**	**Preuves automatiques**	**Preuves interactives**	**Nombre d'OP non-prouvées**
B_{Swing}	19	19	0	0
B_{Task}	25	25	0	0
B_{Task_ref1}	40	39	1	0
B_{Task_ref2}	49	46	3	0
$B_{ValidAppl}$	125	64	60	1
$B_{ValidAppl\_2}$	157	82	75	0

Totaux des nombres d'obligations de preuve pour la validation complète du système (B_{Swing}, B_{Task}, B_{Task_ref1}, B_{Task_ref2}, $B_{ValidAppl\_2}$)		
Nombre total d'OP	$\Rightarrow$	**290**
Preuves automatiques	$\Rightarrow$	**211**
Preuves interactives	$\Rightarrow$	**79**

TAB. 6.1 – Nombre d'obligations de preuve pour chacun des modèles et raffinements B présentés.

Obligations de preuve du modèle $B_{ValidAppl}$. Comme l'indique le tableau 6.1, une obligation de preuve n'est pas prouvée sur le raffinement $B_{ValidAppl}$. Cette obligation de preuve concerne le non-blocage du système (assertion du raffinement). En effet, la concrétisation de l'événement $Evt\_T_{2.1}$ correspondant à la tâche `saisir_valeur` ($T_{2.1}$) par les événements $keyPressed\_0$, $keyPressed\_1$ et $keyPressed\_2$ du modèle B_{Appl} n'assure pas l'exécution en séquence de la tâche $T_{2.2}$ (tâche de conversion). Plus précisément, dans le cas où l'utilisateur saisit une valeur vide dans le champ de texte `input`, l'événement $keyPressed\_1$ est alors exécuté. La substitution de cet événement assigne la valeur `false` à l'attribut `enabled` des boutons `ED` et `DE` (`enabled(ED):=false enabled(DE):=false`)[5]. Dans ces conditions, les événements $Evt_{InitChoice}$, $Evt\_T_{2.2.1}$ et $Evt\_T_{2.2.1}$ ne peuvent pas être exécutés en séquence puisque leurs gardes ne sont pas vérifiées. Rappelons que ces gardes sont constituées en partie par les conditions d'activation des widgets `ED` et `DE` qui expriment notamment que ces boutons doivent être actifs et visibles pour permettre l'activation de ces événements.

[5]Voir le modèle de la méthode keyPressed extrait de l'application (Chapitre 4, section 4.3, figure 4.7)

Le modèle de tâches manque d'informations concernant la tâche `saisir_valeur` : il est en effet impossible de savoir si cette tâche présuppose ou non la saisie d'une valeur non vide de la part de l'utilisateur. L'interprétation de l'action utilisateur attendue par cette tâche dépend donc d'un choix de la personne en charge de la validation.

Cette situation pourrait être évitée en ajoutant certaines informations à l'arbre de tâches CTT (pré-condition/post-condition portant sur les tâches) ou bien en décomposant de manière plus fine la tâche T_2 dans le modèle de tâches.

Modification du modèle $B_{ValidAppl}$: $B_{ValidAppl\_2}$. Le modèle $B_{ValidAppl\_2}$ est identique au modèle $B_{ValidAppl}$ sauf dans sa définition de l'événement $keyPressed\_0$. La figure 6.12 présente les modifications effectuées sur cet événement : la garde de l'événement est modifiée par l'ajout de la condition `xx=full`. Cette condition assure que l'utilisateur saisit une valeur non vide.

```
1 keyPressed_0=
2 ANY xx WHERE
3   xx ∈ TEXT  ∧
4   [...]
5 THEN
6   Jtext(input):=xx  ||
7   Vkp:=0
8 END;
```

(A) $B_{ValidAppl}$: événement $keyPressed\_0$.

```
1 keyPressed_0=
2 ANY xx WHERE
3   xx ∈ TEXT  ∧  xx=full  ∧
4   [...]
5 THEN
6   Jtext(input):=xx  ||
7   Vkp:=0
8 END;
```

(B) $B_{ValidAppl\_2}$: modification de l'événement.

FIG. 6.12 – Différence entre les modèles $B_{ValidAppl}$ et $B_{ValidAppl\_2}$.

La modification de cet événement permet la preuve de toutes les OP du raffinement.

Obligations de preuve. La preuve de correction du raffinement $B_{ValidAppl\_2}$ a été obtenue en déchargeant 157 OP (dont 82 OP sont déchargées automatiquement). La plateforme Rodin a été utilisée comme assistant de preuve pour établir cette correction. La figure 6.13 présente une capture d'écran de cet outil.

Les OP à prouver interactivement (82) sont nombreuses compte tenu de la simplicité du système. Cependant toutes les OP rencontrées, mise à part la preuve de l'assertion de non-blocage du système, sont très simples à décharger. Le tableau 6.3 présente les formes d'OP les plus souvent rencontrées.

De manière générale, le choix des hypothèses appropriées et la reformulation des OP à l'aide d'axiomes simples ajoutés au modèle permettent la preuve de ces OP. En outre, quel que soit le système étudié, la forme des OP à décharger sera identique[6]. Ceci laisse penser que les OP prouvées interactivement à l'heure actuelle pourraient être prouvées automatiquement. Ceci nécessiterait de regrouper les OP rencontrées suivant différentes classes en fonction de leur forme et d'introduire des stratégies de preuve adaptées à chacune de ces classes.

[6]La modélisation repose sur formalisation de la boîte à outils Java-Swing B_{Swing} qui ne diffère pas d'une étude à l'autre. En outre, la formalisation du modèle CTT utilise des règles de décomposition strictement définies : la "forme" du modèle $B_{ValidAppl}$ est par conséquent identique d'un système à un autre.

Au vu des preuves interactives effectuées sur l'étude de cas, il parait tout à fait vraisemblable que 90% à 95% des OP pourraient être ainsi déchargées automatiquement.

La preuve des 5-10% d'OP restantes nécessite l'introduction d'invariants de collage. À titre d'exemple, la preuve de l'assertion de non-blocage à nécessité l'introduction de trois invariants de collage. L'introduction de ces invariants de collage est plus délicate à automatiser. L'étude de cette automatisation constituerait un travail de recherche en soi.

Obligations de Preuve	**Hypothèses utilisées pour la preuve**
$\texttt{input} \in \texttt{dom(enabled)}$	$\mathbf{H}_1 : \texttt{enabled} \in \texttt{WIDGETS} \rightarrow \texttt{BOOL}$ $\mathbf{H}_2 : \texttt{input} \in \texttt{WIDGETS}$
$\texttt{enabled} \lhd \texttt{\{ED} \mapsto \texttt{true,DE} \mapsto \texttt{true\}} \in \texttt{WIDGETS} \rightarrow \texttt{BOOL}$	$\mathbf{A}_1 : \forall \texttt{f,g.((f} \in \texttt{WIDGETS} \rightarrow \texttt{BOOL} \wedge \texttt{g} \in \texttt{WIDGETS} \leftrightarrow \texttt{BOOL)} \Rightarrow \texttt{(f} \lhd \texttt{g} \in \texttt{WIDGETS} \rightarrow \texttt{BOOL))}$ (axiome ajoutée au modèle B_{Swing})
$\texttt{(enabled} \lhd \texttt{\{ED} \mapsto \texttt{true,...)(ED)=true}$	$\mathbf{A}_2 : \forall \texttt{f,g,w,b.}\big(\texttt{(b} \in \texttt{BOOL} \wedge \texttt{f} \in \texttt{WIDGETS} \rightarrow \texttt{BOOL} \wedge \texttt{g} \in \texttt{WIDGETS} \leftrightarrow \texttt{BOOL} \wedge \texttt{w} \in \texttt{dom(g))} \Rightarrow \texttt{((f} \lhd \texttt{g)(w)=g(w))}\big)$ (axiome ajoutée au modèle B_{Swing})

TAB. 6.2 – Exemple d'OP le plus fréquemment rencontrées lors de la preuve de correction du raffinement $B_{ValidAppl\_2}$.

Simulation des modèles B événementiel. Il existe également plusieurs outils permettant l'animation de modèles B événementiel et la génération de cas de tests (GeneSyst [Bert *et al.*, 2005], BZ-Testing-Tools [Ambert *et al.*, 2002], B2EXPRESS [Aït-Sadoune, 2007]) ou le *model-checking* de spécifications (ProB [Leuschel & Turner, 2005]). Parmi ces outils, l'animateur B2EXPRESS [Ait-Sadoune & Aït-Ameur, 2007] a été utilisé pour animer le modèle B_{ApplM}. Cet outil permet entre autres d'animer un modèle B pas à pas. À chaque pas de l'animation, les événements B pouvant être déclenchés (c'est-à-dire les événements dont la garde est vraie) sont mis en évidence sur l'interface.

Une autre possibilité très intéressante de l'outil est de pouvoir animer un modèle suivant des scénarios décrit dans le langage CTT. Dans le cas où l'animation échoue, la garde de l'événement responsable de cet échec est exhibée au niveau du modèle.

L'utilisation de cet outil peut considérablement aider la personne en charge de la validation. En effet cet outil permet au concepteur de mieux comprendre les modèles B

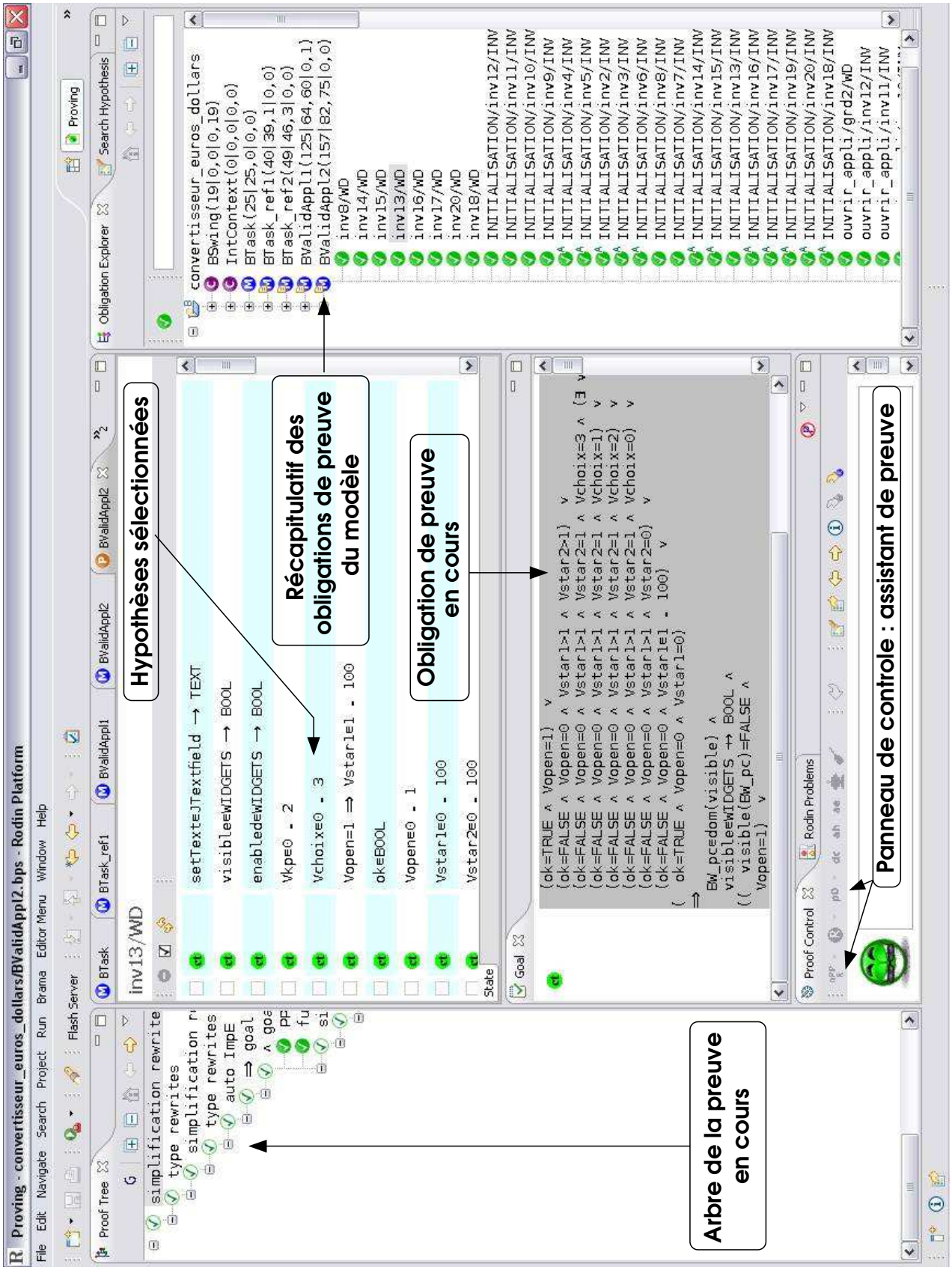


FIG. 6.13 – Capture d'écran : OP sous la plateforme Rodin.

extraits et peut l'assister dans sa démarche de preuve en l'aidant à identifier certains problèmes de conception.

6.4 Validation de l'application vis-à-vis du modèle CTT : NuSMV

Cette section présente un processus de validation exploitant la technique formelle NuSMV. Cette section vise à mettre en évidence la faisabilité d'un processus de validation de l'utilisabilité d'une application Java-Swing exploitant une technique formelle basée sur la vérification exhaustive de modèle. Les résultats présentés dans cette section permettent d'établir quelques constats et quelques comparaisons quant à l'utilisation de deux systèmes de preuve distincts (*theorem-proving* et *model-checking*).

Les grandes lignes de ce processus sont similaires à celles présentées pour le processus de validation exploitant la méthode B événementiel. Par conséquent, ce processus de validation est présenté de manière plus concise.

Un exemple tiré de l'étude de cas est également présenté dans cette section pour illustrer la démarche proposée.

6.4.1 Principes du processus de validation NuSMV

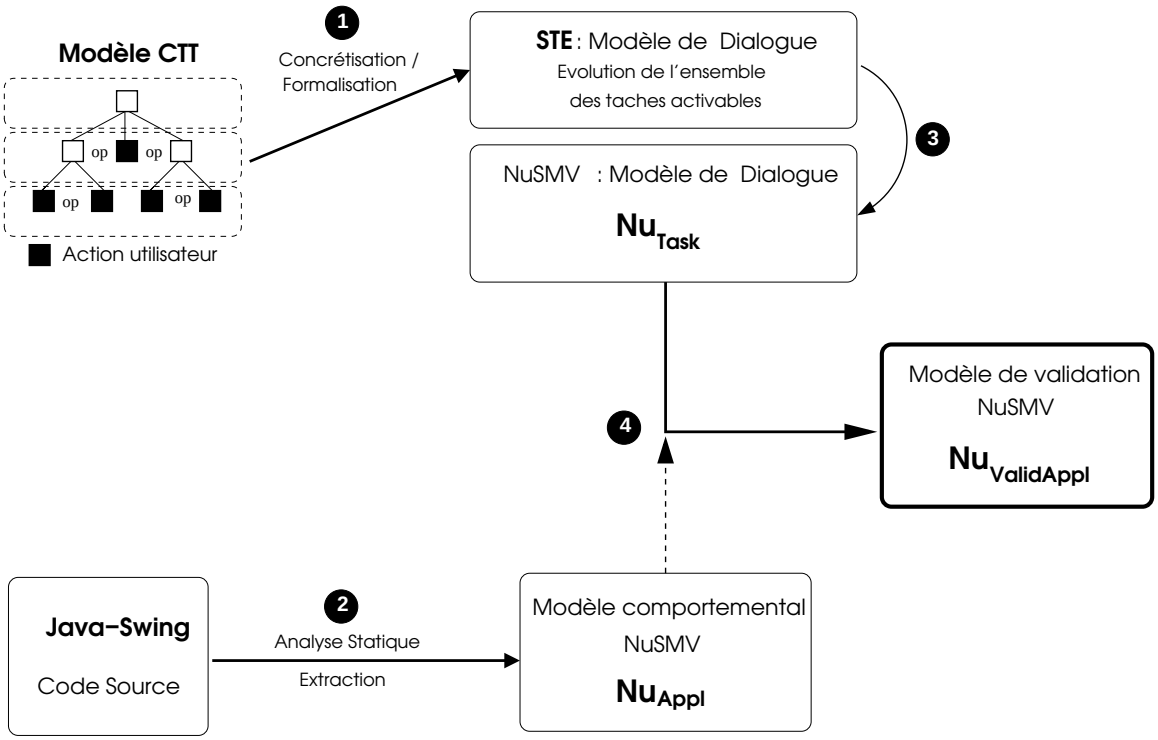


Fig. 6.14 – Processus de validation NuSMV.

Quatre étapes distinctes structurent ce processus :

1. **Première étape** (Fig.6.14,❶) : elle consiste à extraire un modèle NuSMV Nu_{Appl} de l'application Java-Swing étudiée ;
2. **Deuxième étape** (Fig.6.14,❷) : elle consiste à concrétiser et à formaliser le modèle de tâche en un système de transitions étiquetées (STE). Pour ce faire, on exploite les travaux de [Luyten *et al.*, 2003] qui présente un algorithme permettant d'extraire un modèle de dialogue sous la forme d'un STE à partir d'un modèle CTT ;
3. **Troisième étape** (Fig.6.14,❸) : la troisième étape consiste à représenter ce système de transitions dans le formalisme NuSMV. Le modèle obtenu est appelé Nu_{Task}. Cette étape, comme la précédente, est automatisable ;
4. **Quatrième étape** (Fig.6.14,❹) : enfin, la dernière étape consiste à fusionner les modèles Nu_{Task} et Nu_{Appl} au sein d'un modèle de validation $Nu_{ValidAppl}$. Ce modèle est alors utilisable pour prouver que tous les scénarios d'interaction décrits par le modèle CTT sont réalisables sur l'interface.

6.4.2 Concrétisation et formalisation du modèle CTT en NuSMV

La concrétisation du modèle CTT est identique à celle présentée dans la section 6.2. La suite de cette section présente donc uniquement le principe d'extraction d'un modèle de dialogue à partir d'un modèle CTT sous la forme d'un STE, et l'implémentation de ce STE dans le formalisme NuSMV.

Construction d'un modèle de dialogue : STE

Le principe de construction d'un modèle de dialogue à partir d'une spécification CTT sous la forme d'un STE repose sur la génération des Ensembles de Tâches Activables (*Enabled Task Sets* ou ETS en anglais). [Paternò, 2001] définit un ETS comme "un ensemble de tâches logiquement autorisées à être déclenchées au même instant".

[Paternò, 2001] propose un algorithme pour calculer les ETS d'un modèle de tâches. Les ETS calculés à partir du modèle CTT présenté dans la première section de ce chapitre (figure 6.2) sont présentés dans le tableau suivant :

$$
\begin{aligned}
ETS_{Start} &= \{\mathtt{T}_1\} \\
ETS_1 &= \{\mathtt{T}_{2.1},\mathtt{T}_3\} \\
ETS_2 &= \{\mathtt{T}_4\} \\
ETS_3 &= \{\mathtt{T}_{2.1},\mathtt{T}_{2.2.1},\mathtt{T}_{2.2.2},\mathtt{T}_3\} \\
ETS_{Stop} &= \{\ \}
\end{aligned}
$$

Tab. 6.3 – ETS calculés à partir du modèle CTT de la figure 6.2.

[Luyten *et al.*, 2003] propose un algorithme permettant de traduire un modèle de tâches CTT en un STE en utilisant ces ETS. Cette traduction consiste dans un premier temps à repérer l'état initial puis à détecter les transitions entre les différents ETS en analysant les opérateurs temporels qui lient les différentes tâches.

L'utilisation de cet algorithme permet d'obtenir le STE présenté en figure 6.15. Il s'agit du STE obtenu à partir du modèle de tâches de la figure 6.2 (section 6.1.2).

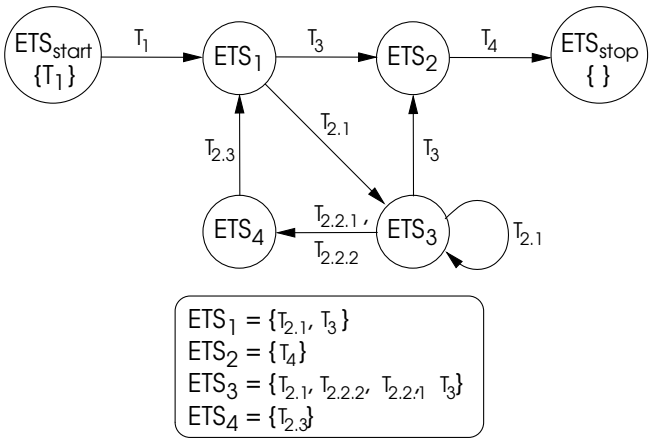


FIG. 6.15 – Traduction du modèle de tâches en STE.

Suppression des tâches "application". Comme dans le cas de la formalisation d'un modèle CTT en B événementiel, les tâches de type "application" du modèle CTT doivent être supprimées. Ces tâches sont prises en compte en ajoutant des propriétés de sûreté devant être vérifiées ("post-conditions" de tâches). La figure 6.16 présente le STE obtenu suite à ces modifications (cf. modèle CTT présenté en figure 6.6, section 6.2.2).

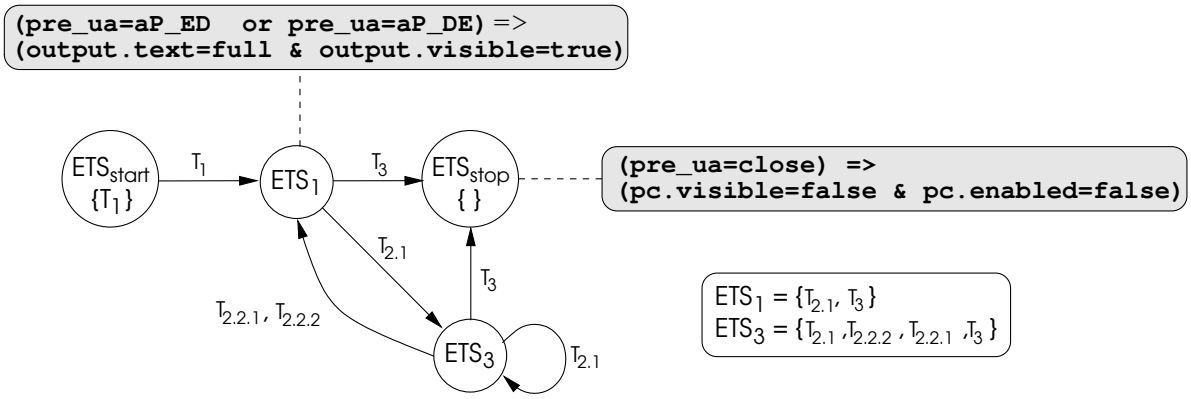


FIG. 6.16 – STE obtenu après suppression des tâches "application".

Concrétisation du STE. La dernière étape consiste à prendre en compte la concrétisation du modèle CTT. Cette étape consiste à remplacer les tâches CTT par les opérations concrètes du système. La figure 6.17 présente le STE finalement obtenu. Cette étape de concrétisation nécessite l'introduction d'un nouvel état ($\mathbf{ETS}_1'$) et de nouvelles transitions. Ces ajouts s'expliquent par le fait que la tâche $T_{2.1}$ (`saisir_valeur`) du modèle de tâches est concrétisée par trois événements (`kp0`, `kp1` et `kp2`) décrivant l'exécution de la méthode `keyPressed` du système (cf. le modèle NuSMV extrait à partir du code source Java-Swing au chapitre 4, section 4.4).

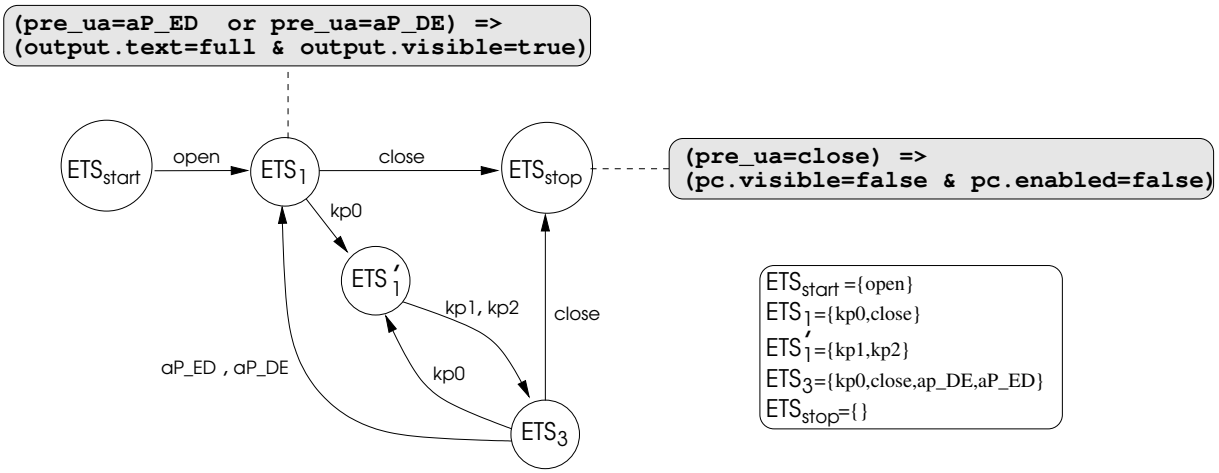


FIG. 6.17 – Concrétisation du STE.

Représentation du STE en NuSMV

Dans le modèle Nu_{Appl} (cf. chapitre 4, section 4.4) obtenu par analyse statique, la variable `ua` correspond à la prochaine action exécutée et prend sa valeur dans l'ensemble des étiquettes désignant les actions du système : `ua∈{open,kp0,kp1,kp2,aP_ED,aP_DE, close,nil,LM_action}`. Dans le modèle Nu_{Appl}, le choix de la prochaine action exécutée est indéterministe : toute action dont la garde est vérifiée peut être déclenchée.

```
1  ASSIGN
2   next(ua):=
3     case
4       ua in  {open,kp1, kp2, aP_ED, aP_DE, close, kp0,nil} : LM_action;
5       ua=LM_action ∧ preua=open ∧ G_kp0 ∧ G_close : {kp0, close};
6       ua=LM_action ∧ preua=kp1 ∧ G_aP_ED ∧ G_aP_DE ∧ G_kp0 ∧ G_close : {aP_ED,aP_DE,kp0,close};
7       ua=LM_action ∧ preua=kp2 ∧ G_aP_ED ∧ G_aP_DE ∧ G_kp0 ∧ G_close : {aP_ED,aP_DE,kp0,close };
8       ua=LM_action ∧ preua=aP_ED ∧ G_kp0 ∧ G_close : {kp0, close };
9       ua=LM_action ∧ preua=aP_DE ∧ G_kp0 ∧ G_close : {kp0, close };
10      ua=LM_action ∧ preua=close ∧ G_open : {open};
11      ua=LM_action ∧ preua=kp0 ∧ input.text=1 /* 1 ≡ full */ : {kp2};
12      ua=LM_action ∧ preua=kp0 ∧ input.text=0 /* 0 ≡ empty */ : {kp1};
13      1 : nil;
14    esac;
15 SPEC
16    AG((preua=close)⇒ (pc.visible=true ∧ pc.enabled=true))
17    AG((preua=aP_ED ∨ preua=aP_DE) ⇒ (output.text=full ∧ output.visible=true))
```

FIG. 6.18 – Extrait du modèle de validation $Nu_{ValidAppl}$.

Le système de transitions de la variable `ua` est ici modifié afin d'accepter uniquement les actions prévues par le modèle de tâches. La représentation NuSMV du STE de la figure 6.17 est donnée par la figure 6.18. Le choix de la prochaine valeur de `ua` est effectué en fonction de la dernière action réalisée (`preua`). Lorsque plusieurs actions appartenant au même ETS sont activables, le choix de la prochaine action est effectué de manière indéterministe. Enfin, le choix de la prochaine action réalisable est conditionné par les gardes des actions possibles. Par exemple, dans le cas où `ua=LM_action ∧ preua=open` (Fig.6.18,ligne 5), la prochaine action peut être `kp0` ou `close` (Fig. 6.17). Dans ce cas, `ua`

peut prendre la valeur `kp0` ou `close` si et seulement si $\mathtt{G}_{kp0} \wedge \mathtt{G}_{close}$ est vrai. Ces gardes correspondent aux conditions d'activation des méthodes Java-Swing. Si aucune action n'est réalisable, la variable `ua` prend alors la valeur `nil`.

Dans ce modèle, les propriétés de sûreté permettant de prendre en compte les tâches "application" sont représentées dans la clause SPEC du modèle.

Validation

Le modèle de validation $\mathrm{Nu}_{ValipAppl}$ est identique au modèle extrait Nu_{Appl} (cf. chapitre 4, section 4.4) : seul le STE de la variable `ua` est remplacé par celui présenté par la figure 6.18.

Montrer que le système Java-Swing se conforme au modèle de tâche CTT revient alors à prouver la propriété suivante :

$$\mathbf{Prop}_{NuValid} : AG(\neg(ua = nil))$$

En effet, la variable `ua` prend la valeur `nil` uniquement dans le cas où le choix d'une action prévue par le modèle de tâche n'est pas activable (au moins l'une des gardes des actions possibles est fausse).

6.4.3 Résultats obtenus

Les propriétés liées aux tâches "application" du modèle ont été vérifiées par le *model-checker* NuSMV comme en témoigne la figure 6.19.

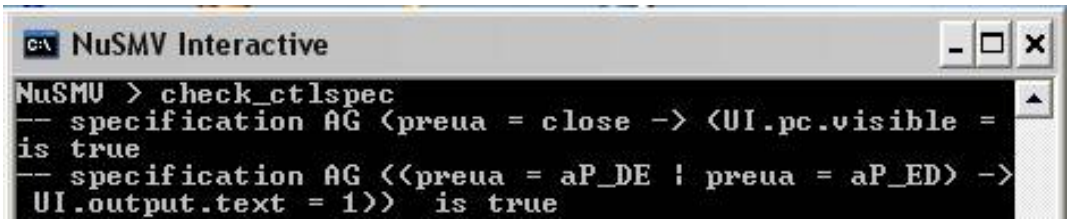


FIG. 6.19 – Environnement NuSMV : Preuves des propriétés liées aux tâches "application" du modèle CTT.

La propriété $\mathbf{Prop}_{NuValid}$ du modèle $\mathrm{Nu}_{ValidAppl}$ n'a pas pu être vérifiée. Le *model-checker* fournit dans ce cas un contre-exemple sous la forme d'une trace. La figure 6.20 présente la trace donnée en sortie du *model-checker*.

Les raisons de cet échec sont identiques à celles données dans le cadre de la validation B événementiel. Une modification simple du modèle, qui assure que l'utilisateur ne peut pas entrer une valeur à convertir vide, permet d'obtenir un modèle $\mathrm{Nu}_{ValidAppl\_2}$. Ce dernier modèle est validé sans aucune difficulté. Il est alors possible de conclure que les scénarii d'interaction de l'application Java-Swing sont conformes aux scénarii d'usage décrits par la spécification CTT.

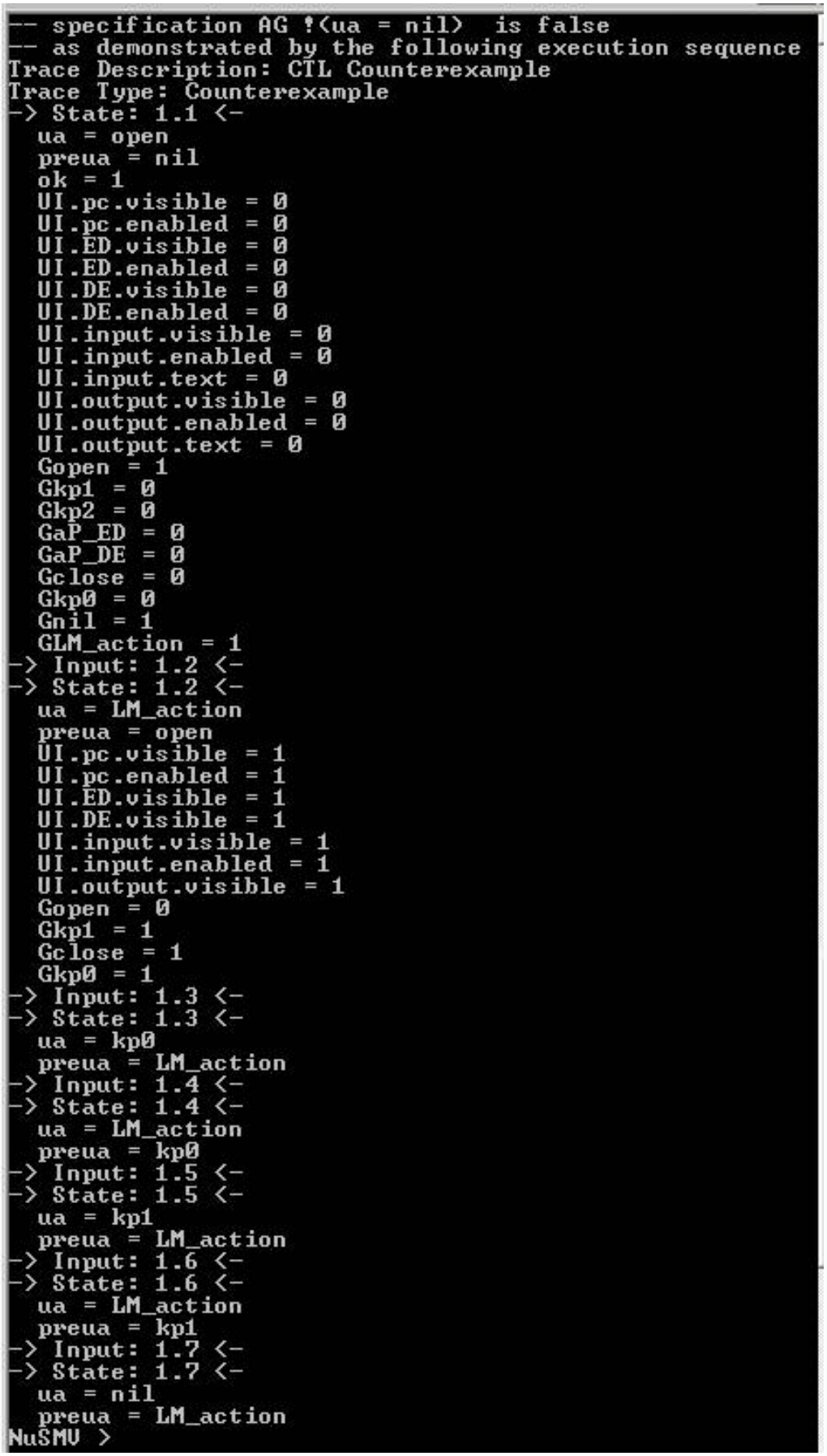


 FIG. 6.20 – Environnement NuSMV : trace d'une propriété non vérifiée.

6.5 Conclusions sur le processus de validation

Ce chapitre a présenté une méthodologie permettant de valider une application Java-Swing vis-à-vis d'une spécification CTT suivant deux techniques formelles : la méthode B événementielle et la méthode NuSMV.

Validation : utilisation du B événementiel. Le processus de validation B événementiel exploite le modèle événementiel B_{Appl} qui modélise l'exécution des méthodes d'écouteurs d'événements, et le modèle B_{Task} qui est obtenu par formalisation du modèle de tâches CTT. La validation consiste à montrer que le modèle concret B_{Appl} est un raffinement correct du modèle abstrait B_{Task} : un raffinement $B_{ValidAppl}$ de B_{Task} est construit à cet effet. Ce modèle de validation lie chacun des événements du modèle B_{Task} correspondant à des actions utilisateurs élémentaires aux événements de B_{Appl} décrivant la réaction de l'application suite à cette action. La preuve de correction du raffinement (preuves sur les variants du système, preuve de non-blocage du système, preuves des invariants...) permet alors de valider l'application.

Validation : utilisation de NuSMV. La validation de l'application exploitant la technique NuSMV est analogue à la précédente mise à part la notion de raffinement qui est absente du processus.

Automatisation de la démarche. La démarche de validation nécessite une participation active du concepteur/développeur quelle que soit la technique formelle utilisée (B ou NuSMV). Notamment, la concrétisation du modèle de tâches ne peut pas être automatisée et nécessite de la part du développeur une bonne connaissance du système (quel *widget* permet de réaliser telle action ?) et une bonne compréhension du modèle extrait par analyse statique (comment définir suivant le formalisme B ou NuSMV un invariant formalisant une tâche CTT de type application ?). Cette concrétisation peut être effectuée par le développeur au cours de l'implémentation du logiciel.

La phase de formalisation du modèle de tâches concrétisé peut quant à elle être effectuée de manière automatique en utilisant les règles de traduction de CTT vers B dans le cas du B événementiel (règles initialement proposées par [Aït-Ameur *et al.*, 2005b] et dont certaines ont été modifiées pour réduire le nombre d'événements générés), ou en utilisant l'algorithme de [Luyten *et al.*, 2003] dans le cas de NuSMV.

La construction des modèles de validation $B_{ValidAppl}$ et $Nu_{ValidAppl}$ peut également être automatisée : la concrétisation du modèle de tâches fournit les informations qui permettent de lier la formalisation du modèle de tâches et le modèle formel extrait.

Les modèles de validation obtenus ne correspondent pas forcément à notre attente. En effet, dans l'exemple de l'étude de cas, il est impossible de conclure que l'implémentation respecte sa spécification (une OP non prouvable dans la technique B événementiel et contre-exemple dans le cas de NuSMV). Dans un tel cas de figure, le concepteur doit donc être capable d'interpréter les résultats obtenus. En l'occurrence, dans l'étude de cas, le problème vient d'un manque de précision du modèle de tâches (l'utilisateur peut-il saisir une valeur vide ?). Une modification très simple des modèles permet alors de conclure que l'application étudiée respecte ses spécifications.

Comparaison des deux approches : B événementiel, NuSMV.

Il est évident que la preuve de correction du raffinement en B événementiel est plus complexe que la validation automatique par NuSMV. Sur l'exemple du convertisseur Euros/Dollars, un nombre très important d'OP doivent être déchargées interactivement. Cependant, le nombre de preuves à effectuer interactivement peut aisément être diminué en introduisant des stratégies simples à mettre en place. Cette affirmation s'appuie sur la similitude de la plupart des OP générées par le prouveur B et la simplicité à prouver celles-ci après adjonction de quelques axiomes. La seule difficulté rencontrée peut concerner la preuve de non-blocage du système [7]. Cette dernière preuve nécessite l'introduction d'invariants "structurels" (invariants sur les variables de contrôle du modèle permettant de lier certains états du système). S'il est tout à fait envisageable de générer automatiquement de tels invariants, il est impossible d'affirmer à ce stade de l'étude que ces invariants permettront effectivement la preuve de non-blocage du système. Afin de répondre à cette question, il sera nécessaire d'appliquer la méthodologie proposée sur des exemples moins triviaux (systèmes réels).

Malgré les possibles difficultés rencontrées lors de la preuve de propriétés, cette étape de preuve interactive possède un atout : elle oblige en effet la personne en charge de la validation à réfléchir sur le système et par conséquent peut l'aider à détecter des problèmes d'implémentation.

Il est également à rappeler une différence notable entre le B événementiel et NuSMV : B événementiel permet de traiter des systèmes infinis alors que NuSMV ne prend en charge que des systèmes finis.

Enfin, il est intéressant de remarquer la complémentarité des deux méthodes du point de vue de la vision du système qu'elles imposent au concepteur. En B événementiel, le modèle obtenu par analyse statique met l'accent sur les événements : il s'agit d'une vue globale du système très proche du programme d'origine puisque les événements modélisent directement l'exécution des méthodes d'écouteurs d'événements (réactions du système). En NuSMV, le modèle met l'accent sur les variables du système : il est possible de suivre l'évolution d'une variable en fonction des actions de l'utilisateur en regardant le système de transitions associé à cette variable. Contrairement au cas précédent, il s'agit d'une vue locale du système. Ces deux visions différentes d'un même système sont complémentaires dans une approche s'intéressant à la compréhension de programmes.

Conception itérative. Ce dernier point est très intéressant : au-delà des preuves effectuées lors du processus de validation, la formalisation oblige le concepteur à s'interroger sur la spécification. En ce sens, le processus de validation présenté trouverait toute son efficacité dans un cycle de développement itératif. Dans un tel cycle de développement, le concepteur exploiterait la démarche de validation (extraction d'un modèle partiel de l'application et formalisation du modèle de tâches) au fur et à mesure de son implémentation afin de s'interroger sur la spécification et le système en cours de construction. Ce va-et-vient du code vers les modèles formels permettrait de détecter au plus tôt les éventuelles erreurs de conception ou d'implémentation.

[7] Propriété de non-blocage : disjonction des gardes des événements abstraits ⇒ disjonction des gardes des événements concrets

L'utilisation conjointe des deux techniques pourrait également être envisagée dans un développement itératif.

En résumé :

1. *Méthode B événementiel* :
   - preuves interactives ;
   - prise en charge de systèmes infinis ;
   - vision par événements proche des programmes (vision globale : les événements reflètent directement l'exécution des méthodes d'écouteurs d'événements) ;
2. *Méthode NuSMV* :
   - preuves automatiques ;
   - prise en charge uniquement de systèmes finis ;
   - vision du système par variables (vision locale) ;

Conclusion générale et perspectives

"Le secret d'un bon discours, c'est d'avoir une bonne introduction et une bonne conclusion. Ensuite, il faut s'arranger pour ces deux parties ne soient pas trop éloignées l'une de l'autre."
George Burns

1/ Conclusion

Le travail présenté dans ce mémoire a été motivé par la complexité de la validation des systèmes interactifs dits "critiques" et réalisé dans le souci de proposer une solution à la validation de ces systèmes pouvant trouver sa place dans un cycle de développement traditionnel. L'étude bibliographique sur le domaine de l'Interaction Homme-Machine (Première partie) a permis d'établir un certain nombre de constats. Ces constats ont amené à proposer une approche de validation des IHM exploitant les techniques formelles uniquement dans la phase de validation du système déja réalisé et non dans une démarche de conception. Ce choix évite au concepteur une remise en cause complète de ses méthodes de travail tout en améliorant la démarche de validation et, potentiellement, diminue les tests nécessaires à la validation de tels systèmes. Ce positionnement vis-à-vis du cycle de développement d'une application tend naturellement à l'utilisation du seul matériel dont on dispose dans une telle démarche : le code source de l'application. Parmi les nombreux langages de programmation existants, le langage Java-Swing a été choisi du fait de son utilisation croissante et de sa portabilité intrinsèque.

Méthodologie

Les travaux consignés dans ce mémoire n'introduisent aucune nouvelle notation. Le choix a été fait de proposer une approche de validation utilisant des techniques, types de modèles ou notations existantes. Le principal apport de ces travaux est avant tout de proposer une méthodologie réunissant les capacités et atouts de différentes techniques

issues d'horizons divers dans une approche cohérente de validation des systèmes interactifs. Cependant, la méthodologie proposée reste valable pour tout autre langage construit sur des concepts analogues (utilisation de boîte à outils graphiques).

Bien entendu, la méthodologie proposée n'a pas la prétention de répondre complètement au problème de la validation des IHM. Elle se focalise sur la validation de l'application vis-à-vis de spécifications de tâches utilisateur exprimées dans la notation CTT qui est apparue comme la notation la plus apte à représenter le comportement attendu d'une application sous forme d'arbres de tâches.

L'approche proposée est constituée de deux grandes étapes présentées dans la seconde partie de ce manuscrit :

1. une première étape d'extraction de modèles formels de l'application par analyse de son code source ;
2. une seconde étape dédiée à la validation de l'application et exploitant entre autres les modèles formels extraits.

Extraction de modèles formels

Les chapitres 4 et 5 ont présenté d'une part les principes de la modélisation d'une application Java-Swing suivant deux techniques formelles (le B événementiel et NuSMV), et d'autre part les techniques d'analyse statique permettant de construire explicitement ces modélisations.

Modélisation formelle. Les modèles extraits représentent la dynamique du dialogue de l'application. La construction de ces modèles de dialogue repose sur l'analyse des méthodes d'écouteurs d'événements et de la méthode principale de l'application. Plusieurs travaux ont déjà proposé la modélisation du dialogue d'une application suivant diverses techniques formelles, principalement dans une démarche descendante de conception. Cependant, les travaux présentés ici correspondent sans doute à la première tentative d'extraction de modèles de dialogue par analyse statique de code source Java-Swing et exploitant une technique formelle (le B événementiel) fondée sur la preuve (*theorem-proving*) : il s'agit de l'une des principales contributions de ce travail.

Afin de mettre en évidence la possibilité d'extraire un modèle de dialogue en utilisant une technique formelle basée sur la vérification exhaustive de modèles (*model-checking*), et de comparer deux techniques de preuve, la possibilité d'extraire un modèle de dialogue NuSMV à partir de l'implémentation du système a été présentée. L'extraction de ce modèle NuSMV peut être réalisée avec un surcoût négligeable puisqu'il est possible de traduire les modèles B événementiel obtenus par l'analyse effectuée en un modèle NuSMV de manière ad hoc et automatique. Concrètement, la projection d'un modèle de dialogue suivant différents langages et techniques formels peut s'appuyer sur un modèle central. Ce modèle central correspond à une représentation intermédiaire de l'analyse statique et peut être utilisé pour dériver un ensemble de modèles basés sur différentes notations formelles.

La possibilité d'extraire plusieurs modèles qui ne diffèrent que par le choix du modèle formel utilisé constitue un atout considérable. En effet, ceci assure une adaptabilité de la méthodologie : la personne en charge de la validation peut exploiter le langage et la technique de preuve les plus adaptés.

Analyse statique. L'extraction de ces modèles formels est réalisée de manière automatique. Par souci de concision, seuls certains algorithmes permettant cette dérivation ont été présentés au chapitre 5. La seule intervention de l'utilisateur dans la démarche méthodologique proposée consiste à définir une abstraction des widgets utilisés dans le code source Java. Ces abstractions sont définies dans le langage Java et sont exploitées au cours de l'analyse. Ainsi, le concepteur n'utilise pas directement une syntaxe formelle pour ses abstractions, mais conserve le langage qu'il est habitué à manipuler.

Le modèle obtenu en sortie de l'analyse statique dépend de l'abstraction effectuée. Dans les travaux présentés, seuls quelques attributs des *widgets* ont été modélisés (visibilité et activité du widget principalement). Cependant, il est tout à fait envisageable d'abstraire d'autres attributs afin de prouver que le système possède des propriétés spécifiques. Par exemple, la couleur peut être ajoutée à l'abstraction des *widgets* pour prouver certaines propriétés d'insistance ou d'observabilité.

L'analyse statique d'un code source Java-Swing nécessite une phase d'abstraction du système et notamment du noyau fonctionnel de l'application (interprétation abstraite, *slicing*). Ces étapes reposent sur une connaissance préalable du système analysé : il est délicat d'analyser un système quelconque. L'utilisation de certaines règles d'implémentation et l'utilisation d'un modèle d'architecture permettent d'effectuer ces étapes d'abstraction de manière automatique. Tout type de modèle d'architecture peut être utilisé s'il est possible d'isoler de manière conceptuelle le noyau fonctionnel de l'application de sa partie purement interactive. Ainsi, il est tout à fait opportun d'utiliser un tel outil d'extraction de modèles formels sur un code source en partie généré par un outil de conception.

Enfin, il est à remarquer que la correction de l'abstraction réalisée découle des techniques utilisées pour l'analyse. Rappelons que toutes les analyses statiques se formalisent en fin de compte dans la théorie de l'Interprétation Abstraite [Cousot & Cousot, 1976], [Cousot & Cousot, 1977], [Cousot & Cousot, 1992], [Cousot & Cousot, 1999], [Cousot, 2002].

Validation de l'application

Vérification de propriétés. Les modèles formels extraits par analyse du code source d'une application Java-Swing peuvent être utilisés directement pour la vérification de propriétés du système. Dans le cas de B événementiel, les propriétés qu'il est possible de vérifier sont des propriétés de sûreté. Les propriétés de sûreté sont exprimées en tant qu'invariants du système. Quel que soit la technique utilisée, l'expression des propriétés (en logique du premier ordre dans le cas de la méthode B événementiel et dans la logique temporelle CTL dans le cas de la méthode NuSMV) relève du même niveau de difficulté. L'utilisation de la logique CTL se révèle toutefois plus souple. Notamment, CTL permet l'expression de propriétés de vivacité et d'équité. L'utilisation de NuSMV sur l'étude de cas a permis de mettre en évidence l'existence de quelques comportements dynamiques du système, par exemple : `(conversion_ED >>conversion_DE)`*`[](conversion_DE >> conversion_ED)`*.

Validation de l'utilisabilité du système. Le chapitre 6 présente comment il est possible d'utiliser les modèles formels extraits pour valider l'application vis-à-vis de sa spécification CTT. Malgré l'hétérogénéité des modèles en présence (d'un côté des modèles

formels et de l'autre un modèle semi-formel qui privilégie une notation graphique), il est possible de mener à bien cette validation en se ramenant à l'utilisation d'un unique langage de modélisation (B ou NuSMV). Quel que soit le langage utilisé, il est possible de formaliser le modèle de tâches CTT de manière automatique. Une fois cette formalisation effectuée un nouveau modèle est construit. Ce dernier modèle permet de lier la modélisation des actions de l'utilisateur à la modélisation correspondante de l'exécution des réactions du système. Ce modèle de validation permet alors de conclure quant à la validité du système vis-à-vis du modèle de tâches.

Ce couplage entre les actions de l'utilisateur et les réactions du système peut être réalisé de manière automatique si la concrétisation du modèle de tâches, qui est à la charge du développeur, est correctement menée.

Dans le cas de l'utilisation du B événementiel, la construction du modèle de validation repose sur le principe de raffinement. La construction de la formalisation du modèle CTT est obtenue de manière automatique en plusieurs étapes de raffinement. Chaque nouveau raffinement correspond à la décomposition d'une tâche utilisateur abstraite. Cette étape exploite des règles de traductions définies par [Aït-Ameur *et al.*, 2005b] : certaines de ces règles de traductions ont été légèrement modifiées pour l'occasion afin de diminuer le nombre d'événements générés.

Comparaison des modèles formels. Si l'on considère la combinatoire très forte du dialogue des systèmes interactifs, l'utilisation de la méthode B événementiel semble mieux adaptée. En effet, la technique de *model-checking* avec NuSMV est limitée théoriquement par le problème de l'explosion combinatoire puisqu'un système de transitions est explicitement généré lors du parcours exhaustif des états. Cette limite est loin d'avoir été atteinte sur l'étude de cas présentée et il serait nécessaire d'exploiter la méthodologie sur une application à l'échelle réelle pour constater si cette limite est réellement atteinte. Il est à rappeler que l'avantage de la méthode B événementiel vis-à-vis de NuSMV réside en sa capacité à pouvoir modéliser des systèmes infinis.

Contrairement à NuSMV, où la preuve de propriétés est complètement automatisée, l'utilisation du B événementiel demande une implication plus forte de la part de la personne en charge de la vérification : certaines propriétés du système doivent être démontrées interactivement. Toutefois la plupart des OP déchargées interactivement sur l'étude de cas sont très simples et pourraient être automatisées. En outre, les interfaces graphiques de la dernière génération d'assistants de preuve B (la Balbulette, plate-forme Rodin) facilitent grandement la démonstration de propriétés : visualisation de l'arbre de la preuve, utilisation simple de stratégies de preuve par simple clic, réutilisation des invariants préalablement prouvés comme hypothèses, gestion interactive des hypothèses à utiliser...

Coopération des techniques formelles

Chacune des techniques formelles présentées possède ses avantages et ses inconvénients. Cependant, loin de s'exclure l'une de l'autre, celles-ci semblent bien au contraire se compléter habilement : une utilisation conjointe de ces deux techniques peut se révéler utile dans une méthodologie de validation des systèmes.

Complémentarité des approches : compréhension de programmes. L'utilisation des modèles B événementiel et NuSMV permet d'obtenir deux visions complémentaires d'un même système interactif. Un modèle B événementiel donne une vision du système très proche des programmes. Cette vision du système, qui peut être qualifiée de "globale", reflète directement l'exécution des méthodes d'écouteurs sous la forme d'événements : il est alors aisé de suivre l'évolution des variables (réaction de l'interface) correspondante à une action utilisateur particulière. À l'opposé, un modèle NuSMV offre une vision du système par variables. Cette vision "locale" du système permet de suivre aisément l'évolution d'une variable en fonction des actions de l'utilisateur : il suffit pour cela d'étudier le système de transitions associé à cette variable.

Une compréhension fine du système peut devenir indispensable au cours du processus de validation, notamment dans le cas où une propriété souhaitée du système se révèle être fausse. La complémentarité des deux modèles est alors très appréciable puisqu'elle permet d'aborder le problème posé sous différents points de vue.

Complémentarité des approches : coopération des modèles pour la validation du système. Les méthodes sont également fortement complémentaires du point de vue de la preuve de propriétés.

D'un côté, le B événementiel permet la représentation de systèmes infinis, mais est limité quant aux types de propriétés qu'il est capable d'exprimer (uniquement des propriétés de sûreté). En outre, il se peut que des propriétés doivent être prouvées interactivement : ceci n'est pas nécessairement une faiblesse si l'on se place du point de vue de la compréhension de programmes.

De l'autre côté, la méthode NuSMV permet uniquement la représentation de systèmes finis, mais dispose d'une plus grande souplesse du point de vue de l'expression de propriétés. La logique temporelle CTL, utilisée au sein de NuSMV permet d'exprimer non seulement des propriétés de sûreté, mais également des propriétés de vivacité et quelques formes d'équité. De surcroit, la preuve de ces propriétés est automatique.

Cette complémentarité assure une flexibilité pour la personne en charge de la validation : utilisation des deux techniques en parallèle ou bien de l'une des deux. Cette flexibité permet au développeur de répondre à la fois à des contraintes externes (le système étudié est infini), ou bien à une simple convenance personnelle (préférence pour la preuve ou pour la vérification sur modèles).

2/ Perspectives

La réalisation des travaux présentés dans ce manuscrit permet d'entrevoir plusieurs perspectives.

Outillage et passage à l'échelle. La première de ces perspectives est la finalisation d'un outil prototype permettant d'une part l'extraction des modèles formels et d'autre part la construction du modèle de validation (formalisation CTT et couplage des modèles). Cet outil est partiellement implémenté et est codé en Java. Cet outil exploite les technologies JavaCC et JJTree pour la construction d'un analyseur syntaxique Java et la manipulation des arbres de syntaxe abstraite obtenus.

L'existence d'un tel outil permettrait de tester le passage à l'échelle de la méthodologie présentée par l'étude de systèmes réels. Cette étude permettrait :

- de vérifier si la validation de propriétés utilisant la technique formelle NuSMV s'abstrait du problème d'explosion combinatoire ;
- d'analyser d'une part les OP générées par un assistant de preuve B afin de proposer des stratégies de preuve permettant de réduire considérablement le nombre d'OP à décharger interactivement, et d'autre part d'analyser la faisabilité d'une génération automatique d'invariants de collage permettant une preuve plus aisée de la propriété de non-blocage du système.

Il serait également intéressant de coupler un tel outil à un outil de génération d'interfaces et d'explorer son utilisation dans le cadre d'un développement de type itératif. L'utilisation de la méthodologie présentée dans un cycle de développement itératif permettrait d'obtenir une garantie sur le respect des spécifications au fur et à mesure de l'implémentation ou bien de détecter au plus tôt des erreurs. La détection des erreurs nécessiterait de conserver un lien entre le modèle extrait et le code du système étudié afin de cibler rapidement la partie du code source pouvant être responsable d'une anomalie détectée. Concernant la détection au plus tôt des erreurs, l'utilisation parallèle d'un animateur de modèle B tel que B2EXPRESS [Aït-Sadoune, 2007] serait un plus non négligeable afin d'accélerer le processus itératif.

Extension de l'approche. Une seconde perspective est d'étendre l'approche présentée afin d'élargir le spectre de la validation : prise en compte d'autre propriétés liées à l'utilisabilité ou la robustesse du système et prise en compte de spécifications autres que des modèles de tâches de type CTT.

Prise en compte des IHM multimodales. Une perspective de recherche naturelle est d'étendre la méthodologie afin de pouvoir étudier des applications Java-Swing multimodales (*multi-threading*). Le cadre formel générique pour la formalisation d'IHM multimodales proposé par [Kamel, 2006] et plus généralement l'ensemble des travaux menés au sein du projet RNRT-VERBATIM [VERBATIM, 2003 2007] constituent une base solide qui laisse présager la faisabilité d'une telle entreprise.

Bibliographie

[Abowd *et al.*, 1995] Abowd G. D., Wang H.-M. & Monk A. F. (1995). A formal technique for automated dialogue development. In *DIS '95: Proceedings of the conference on Designing interactive systems*, p. 219–226, New York, NY, USA: ACM Press.

[Abrial, 2000] Abrial J. (2000). Guidelines to Formal System Studies. Draft Version 2. url : http://www.atelierb.eu/ressources/articles/gls.V2.1.pdf.

[Abrial, 1996] Abrial J.-R. (1996). *The B-book: assigning programs to meanings*. New York, NY, USA: Cambridge University Press. ISBN : 0-521-49619-5.

[Abrial, 2003a] Abrial J.-R. (2003a). B : passé, présent, futur. *TSI*, **22**(1), p. 89–118.

[Abrial, 2003b] Abrial J.-R. (2003b). B#: Towards a Synthesis between Z and B. In D. e. a. Bert, Ed., *Third International Conference of B and Z Users (ZB2003)*, volume 2651 of LNCS, p. 168–177, Turku. Finland: Springer-Verlag.

[Abrial & Cansell, 2003] Abrial J.-R. & Cansell D. (2003). Click'n Prove: Interactive Proofs within Set Theory. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, p. 1–24, Rom, Italy.

[Abrial *et al.*, 2005] Abrial J.-R., Cansell D. & Méry D. (2005). Refinement and reachability in Event B. In *Conférence ZB'2005*.

[Abrial & Mussat, 1998] Abrial J.-R. & Mussat L. (1998). Introducing dynamic constraints in B. In *B'98 : The 2nd International B Conference*, p. 83–128.

[Aho *et al.*, 2006] Aho A., Lam M., Sethi R. & Ullman J. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley. ISBN : 0321486811.

[Ait-Sadoune, 2005] Ait-Sadoune I. (2005). Vérification et validation formelle d'IHM multimodales fondées sur la preuve. Utilisation de la méthode B. Master's thesis, Mémoire d'ingénieur d'état en informatique, INI, Alger.

[Ait-Sadoune & Aït-Ameur, 2007] Ait-Sadoune I. & Aït-Ameur Y. (2007). B2EXPRESS : Un animateur de modèles B événementiel. In *Approches Formelles*

dans l'Assistance au Développement de Logiciel (AFADL 2007), Université de Namur, Belgique.

[Ambert *et al.*, 2002] Ambert F., Bouquet F., Chemin S., Guenaud S., Legeard B., Peureux F. & Vacelet N. (2002). *Projet BZ-Testing-Tools - Génération de tests aux limites à partir d'un modèle formel B ou Z - Annexes Techniques*. Compte rendu d'avancement au 30 avril 2002, ANVAR. 150 pages.

[Annett & Duncan, 1967] Annett & Duncan (1967). Task analysis and training design. *Occupational Psychology*, **41**, p. 211–221.

[Appel, 1998] Appel A. W. (1998). *Modern compiler implementation in Java*. New York, NY, USA: Cambridge University Press. ISBN : 0-521-58388-8.

[Atelier-B,] Atelier-B. Outils pour l'utilisation opérationnelle de la méthode B, http://www.atelierb.societe.com/produit.html.

[Aït-Ameur, 2000] Aït-Ameur Y. (2000). Cooperation of Formal Methods in an Engineering Based Software Development Process. In *Integrated Formal Methods : Second International Conference*, volume 1945/2000, p. 136–155, Dagsthul Castler, Germany: Springer Verlag.

[Aït-Ameur *et al.*, 2006a] Aït-Ameur Y., Ait-Sadoune I. & Baron M. (2006a). Etude et comparaison de scénarios de développements formels d'interfaces multi-modales fondés sur la preuve et le raffinement. In *MOSIM 2006 - 6ème Conférence Francophone de Modélisation et Simulation. Modélisation, Optimisation et Simulation des Systèmes : Défis et Opportunités*, Rabat, Maroc.

[Aït-Ameur *et al.*, 2006b] Aït-Ameur Y., Ait-Sadoune I., Baron M. & Mota J.-M. (2006b). Validation et Vérification Formelles de Systèmes Interactifs Multi-Modaux Fondées sur la Preuve. In *18° Conférence Francophone sur l'Interaction Homme-Machine (IHM)*, p. 123–130, Montréal.

[Aït-Ameur *et al.*, 2005a] Aït-Ameur Y., Aït-Sadoune I. & Baron M. (2005a). *Modélisation et Validation formelles d'IHM : LOT 1 (LISI/ENSMA)*. Rapport interne 73, Projet RNRT Verbatim, LISI/ENSMA.

[Aït-Ameur & Baron, 2004] Aït-Ameur Y. & Baron M. (2004). Bridging the gap between formal and experimental validation approaches in HCI systems design : use of the event B proof based technique. In Cyprus, Ed., *ISOLA 2004 - 1st International Symposium on Leveraging Applications of Formal Methods*, p. 74–81, Department of Computer Science University of Paphos.

[Aït-Ameur *et al.*, 2003] Aït-Ameur Y., Baron M. & Kamel N. (2003). 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, Algérie*, p. 57–66.

[Aït-Ameur *et al.*, 2005b] Aït-Ameur Y., Baron M. & Kamel N. (2005b). Encoding a Process Algebra Using the Event B Method. In L. College, Ed., *ISOLA 2005, 2nd IEEE International Symposium on Leveraging Applications of Formal Methods*, Department of Computer Science University of, Columbia, Maryland USA.

[Aït-Ameur *et al.*, 1998a] Aït-Ameur Y., Girard P. & Jambon F. (1998a). A Uniform approach for the Specification and Design of Interactive Systems: the B

method. In J. Markopoulos, Panos & Peter, Eds., *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'98)*, p. 333–352, Abingdon, UK.

[Aït-Ameur *et al.*, 1998b] Aït-Ameur Y., Girard P. & Jambon F. (1998b). Using the B formal approach for incremental specification design of interactive systems. In S. Chatty & K. A. P. Dewan, Prasun, Eds., *Engineering for Human-Computer Interaction*, volume 22, p. 91–108.

[Aït-Sadoune, 2007] Aït-Sadoune I. (2007). *Animation et Test de modèles B événementiel. Une approche fondée sur les modèles de données.* Rapport interne, Mémoire de Master Recherche, Université de Poitiers, Poitiers.

[Balbo, 1994] Balbo S. (1994). *Evaluation ergonomique des interfaces utilisateur : un pas vers l'automatisation.* PhD thesis, Université Josef Fourier.

[Ball & Horwitz, 1993] Ball T. & Horwitz S. (1993). Slicing Programs with Arbitrary Control-flow. In *Automated and Algorithmic Debugging*, p. 206–222.

[Balzert *et al.*, 1996] Balzert H., Hofmann F., Kruschinski V. & Niemann C. (1996). The JANUS Application Development Environment - Generating More than the User Interface. In J. Vanderdonckt, Ed., *CADUI'96 : Computer-Aided Design of User Interfaces*, p. 183–208: Presses Universitaires de Namur.

[Baron, 2003] Baron M. (2003). *Vers une approche sûre du développement des Interfaces Homme-Machine.* PhD thesis, Université de Poitiers.

[Bass *et al.*, 1991] Bass L., Pellegrino R., Reed S., Sheppard S. & Szczur M. (1991). The Arch Model: Seeheim Revisited. In *CHI 91 User Interface Developper's Workshop.*

[Bastide, 1992] Bastide R. (1992). *Objets Coopératifs : Un formalisme pour la modélisation des systèmes concurrents.* PhD thesis, Université de Toulouse.

[Bergeretti & Carré, 1985] Bergeretti J.-F. & Carré B. A. (1985). Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, **7**(1), p. 37–61.

[Bert & Cave, 2000] Bert D. & Cave F. (2000). Construction of Finite Labelled Transition Systems from B Abstract Systems. In *Proc. of the Second International Conference Integrated Formal Methods, IFM'2000*, number 1945 in LNCS, p. 235–254, Dagstuhl Castle. Springer Verlag.

[Bert *et al.*, 2005] Bert D., Potet M.-L. & Stouls N. (2005). GeneSyst: A tool to reason about behavioral aspects of B Event specifications. application to security properties. In *ZB'05 conference*, p. 299–318.

[Boehm, 1981] Boehm B. (1981). Structuring the design space. Pentice-Hall.

[Boehm, 1988] Boehm B. W. (1988). A Spiral Model of Software Development and Enhancement. *Computer*, **21**(5), p. 61–72.

[Bouchet *et al.*, 2006] Bouchet J., Madani L., Nigay L., Oriat C. & Parissis I. (2006). Formal Testing of Multimodal Interactive Systems. In *Proceedings of DSV-IS 2006, The XIII International Workshop on Design, Specification and Verification of Interactive Systems (Dublin, Ireland, 26-28 july 2006).*

[Bouchet *et al.*, 2004] Bouchet J., Nigay L. & Ganille T. (2004). ICARE software components for rapidly developing multimodal interfaces. In *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*, p. 251–258, New York, NY, USA: ACM Press.

[Bouillon *et al.*, 2005] Bouillon L., Limbourg Q., Vanderdonckt J. & Michotte B. (2005). Reverse Engineering of Web Pages Based on Derivations and Transformations. In *LA-WEB '05: Proceedings of the Third Latin American Web Congress*, p.3̃, Washington, DC, USA: IEEE Computer Society.

[Browne *et al.*, 1990] D. Browne, P. Totterdell & M. Norman, Eds. (1990). *Adaptive user interfaces*. London, UK, UK: Academic Press Ltd. ISBN : 0-12-137755-5.

[Brun, 1998] Brun P. (1998). *XTL : une logique temporelle pour la spécification formelle des systèmes interactifs*. PhD thesis, Université Paris-Sud.

[Burch *et al.*, 1994] Burch J., Clarke E., Long D., MacMillan K. & Dill D. (1994). Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **13**(4), p. 401–424.

[Burch *et al.*, 1990] Burch J., Clarke E., McMillan K., Dill D. & Hwang L. (1990). Symbolic Model Checking: States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, p. 1–33, Washington, D.C.: IEEE Computer Society Press.

[Cadence Berkeley Labs, 1998] Cadence Berkeley Labs (1998). Getting Started with SMV. Tutorial.

[Calvary, 1998] Calvary G. (1998). *Proactivité et réactivité : de l'assignation à la complémentarité en conception et évaluation d'interfaces homme-machine*. PhD thesis, Université Joseph Fourier.

[Calvary *et al.*, 2005] Calvary G., Daassi O., Coutaz J. & Demeure A. (2005). Des widgets aux comets pour la Plasticité des Systèmes Interactifs. *Revue d'Interaction Homme-Machine (RIHM)*, p. p. 33–53. Volume 6, n°1, ISSN 1289-2963.

[Campos & Harrison, 1999] Campos J. & Harrison M. (1999). From Interactors to SMV: A Case Study in the Automated Analysis of Interactive Systems.

[Campos & Harrison, 2001] Campos J. C. & Harrison M. D. (2001). Model Checking Interactor Specifications. *Automated Software Engineering*, **8**(3-4), p. 275–310.

[Cansell, 2003] Cansell D. (2003). Assistance au développement incrémental et à sa preuve. Habilitation à diriger les recherches (HDR), Université Henri Poincaré.

[Cansell & Méry, 2006] Cansell D. & Méry D. (2006). *Software Specification Methods*, chapter Event B, p. 157–175. Hermes-Science Lavoisier, Henri Habrias and Marc Frappier.

[Card *et al.*, 1983] Card S., Moran T. & Newell A. (1983). *The psychology of Human-Computer Interaction*. Mahwah, NJ, USA: Lawrence Erlbaum Associates, Inc. ISBN : 0898592437.

[Cavada *et al.*, 2005] Cavada R., Cimatti A., Jochim C. A., Keighren G., Olivietti E., Pistore M., Roveri M. & Tchaltsey A. (2005). NuSMV 2.4

User Manual. This document is part of the distribution package of the NuSMV model checker : http://nusmv.irst.itc.it/NuSMV/userman/v24/nusmv.pdf.

[Cazin *et al.*, 1996] Cazin J., Seguin C., Osnowycz W., Ciapessoni E. & Ratto E. (1996). An experience in the specification and test of an Electrical Flight Control System using a temporal logic framework. In *DASIA'96 : DAta Systems In Aerospace conference*, Rome.

[Chikofsky *et al.*, 1990] Chikofsky E., , Cross J. & James H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, **7(1)**, p. 13–18.

[Cimatti *et al.*, 2000] Cimatti A., Clarke E. M., Giunchiglia F. & Roveri M. (2000). NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, **2**(4), p. 410–425.

[Clarke & Emerson, 1982] Clarke E. M. & Emerson E. A. (1982). Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, p. 52–71, London, UK: Springer-Verlag.

[ClearSy, 2001] ClearSy (2001). Event B reference manual. http://. www.atelierb.societe.com/ressources/evt2b/.

[Cleaveland *et al.*, 1993] Cleaveland R., Parrow J. & Steffen B. (1993). The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, **15**(1), p. 36–72.

[Corbett *et al.*, 2000] Corbett J. C., Dwyer M. B., Hatcliff J., Laubach S., Păsăreanu C. S., Robby & Zheng H. (2000). Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, p. 439–448.

[Cortier *et al.*, 2007a] Cortier A., d'Ausbourg B. & Aït-Ameur Y. (2007a). Contribution à la validation formelle des systèmes interactifs. In *Approches Formelles dans l'Assistance au Développements de Logiciels (AFADL'2007)*, p. 27–35, Namur (Belgique): FUNDP Namur, Faculté Universitaire Notre-Dame de la Paix.

[Cortier *et al.*, 2007b] Cortier A., d'Ausbourg B. & Aït-Ameur Y. (2007b). Formal Validation of Java/Swing User Interfaces with the Event-B Method. In J. A. Jacko, Ed., *12th International Conference on Human-Computer Interaction (HCI'07)*, volume volume 1 of *Springer, Lecture Notes in Computer Science*, p. 1062–1071, Beijing (China).

[Cousot, 1997] Cousot P. (1997). Types as abstract interpretations. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 316–331, New York, NY, USA: ACM.

[Cousot, 2000] Cousot P. (2000). Partial Completeness of Abstract Fixpoint Checking. In *SARA '02: Proceedings of the 4th International Symposium on Abstraction, Reformulation, and Approximation*, p. 1–25, London, UK: Springer-Verlag.

[Cousot, 2002] Cousot P. (2002). Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, **277**(1-2), p. 47–103.

[Cousot & Cousot, 1976] Cousot P. & Cousot R. (1976). Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, p. 106–130: Dunod, Paris, France.

[Cousot & Cousot, 1977] Cousot P. & Cousot R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints (POPL). In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 238–252, Los Angeles, California: ACM Press, New York, NY.

[Cousot & Cousot, 1992] Cousot P. & Cousot R. (1992). Inductive definitions, semantics and abstract interpretations. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 83–94, New York, NY, USA: ACM.

[Cousot & Cousot, 1999] Cousot P. & Cousot R. (1999). Refining Model Checking by Abstract Interpretation. *Automated Software Engineering: An International Journal*, **6**(1), p. 69–95.

[Cousot & Cousot, 2000] Cousot P. & Cousot R. (2000). Temporal Abstract Interpretation. In *Conference Record of the 27th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming (POPL)*, p. 12–25, Boston, MA.

[Cousot & Cousot, 2002] Cousot P. & Cousot R. (2002). On Abstraction in Software Verification. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, p. 37–56, London, UK: Springer-Verlag.

[Coutaz, 1987] Coutaz J. (1987). PAC, an Implementation Model for Dialogue Design. In *INTERACT'87*, p. 431–437, North Holland.

[Coutaz, 1990] Coutaz J. (1990). *Interfaces Homme-Ordinateur, Conception et Réalisation*. Paris: Dunod Informatique.

[Coutaz *et al.*, 1993a] Coutaz J., G.Faconti, F.Paterno, L.Nigay & D.Salber (1993a). *MATIS: a UAN Description and Lesson Learned*. Rapport interne, SM/WP14, ESPRIT BRA 7040 Amodeus-2.

[Coutaz *et al.*, 1995] Coutaz J., Nigay L., Salber D., Blandford A., May J. & Young R. (1995). Four Easy Pieces for Assessing the Usability of Multimodal Interaction: The CARE properties. In S. A. A. & D. Gilmore, Ed., *INTERACT'95 conference*, p. 115–120, Lillehammer, Norway: Hall.

[Coutaz *et al.*, 1993b] Coutaz J., Paterno F., Faconti G. & Nigay L. (1993b). A comparison of Approaches for Specifying Multimodal Interactive Systems. In *ERCIM (European Research Consortium for Informatics and Mathematics) Workshop on Multimodal Human-Computer Interaction*, Nancy.

[Curtis & Hefley, 1994] Curtis B. & Hefley B. (1994). A wimp no more : The maturing of user interface engineering. *Interactions*, **1**, p. 23–34.

[d'Ausbourg, 1998] d'Ausbourg B. (1998). Using Model Checking for the Automatic Validation of User Interface Systems. In *Design, Specification and Verification of Interactive Systems'98, Proceedings of the Fifth International Eurographics Workshop (DSV-IS)*, volume 1, p. 242–260, Abingdon, United Kingdom: Springer. ISBN : 3-211-82900-8.

[d'Ausbourg, 2001] d'Ausbourg B. (2001). *Rapport de synthèse en vue de candidater à l'Habilitation à Diriger des Recherches*. Rapport interne, ONERA-CERT.

[d'Ausbourg & Cazin, 1999] d'Ausbourg B. & Cazin J. (1999). Using TRIO Specifications to Generate Test Cases for an Interactive System. In *Design, Specification and Verification of Interactive Systems'99, Proceedings of the Eurographics Workshop in Braga (DSV-IS)*, p. 148–166: Springer. ISBN : 3-211-83405-2.

[d'Ausbourg & Durrieu, 2006] d'Ausbourg B. & Durrieu G. (2006). *Sous projet SP4 : Analyse statiques pour la validation de codes, Lot3-Elaboration de modèle par abstraction de données et d'opérateurs, Lot4-Vérification de modèles et Génération de scénarios de test de programmes*. Rapport interne, Projet exploratoire RNRT 2005-2006, VERBATIM.

[d'Ausbourg *et al.*, 1996a] d'Ausbourg B., Durrieu G. & Roché P. (1996a). Deriving a Formal Model of an Interactive System from its UIL Description in order to Verify and Test its Behaviour. In *Design, Specification and Verification of Interactive Systems'96, Proceedings of the Third International Eurographics Workshop (DSV-IS)*, p. 105–122, Namur, Belgium: Springer.

[d'Ausbourg *et al.*, 1996b] d'Ausbourg B., Durrieu G. & Roché P. (1996b). Une approche pour la validation et la vérification formelles des systèmes IHM. In *AFIHM, Actes du Colloque Francophone sur l'IHM*, Grenoble.

[d'Ausbourg *et al.*, 1996c] d'Ausbourg B., Durrieu G. & Roché P. (1996c). Using Flows for Describing and Verify the Behaviour of Interactive Systems. In *Annals of Telecommunications*, volume 51, p. 9–10.

[d'Ausbourg *et al.*, 1997] d'Ausbourg B., Durrieu G. & Roché P. (1997). *Application des techniques formelles au génie logiciel*, chapter Approche pour la validation et la vérification formelles de systèmes d'Interaction Homme-Machine. Observatoire Français des Techniques Avançées (ARAGO 20).

[d'Ausbourg & Roche, 1995] d'Ausbourg B. & Roche P. (1995). Specifying formally or deriving a formal model from an informal description of user interfaces? In C.Rouff, Ed., *CHZ 95 Workshop on Formal Specification of User Interfaces*, Denver, Colorado.

[Depaulis *et al.*, 2006] Depaulis F., Jambon F., P. G. & Guittet L. (2006). Le modèle d'architecture logicielle H4 : Principes, usages, outils et retours d'expérience dans les applications de conception technique. *Revue d'Interaction Homme-Machine, EuropIA*, **7 (1)**, p. 93–129.

[Detlefs *et al.*, 1998] Detlefs D., Leino K. R., Nelson G. & Saxe J. (1998). *Extended static checking*. Rapport interne 159, Compaq Research Center.

[Dijkstra, 1976] Dijkstra E. W. (1976). *A Discipline of Programming*. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN : 013215871X.

[Dix *et al.*, 1993] Dix A., Finlay J., Abowd G. & Beale R. (1993). *Human-Computer Interaction*. Pentice-Hall.

[du Bousquet *et al.*, 1999] du Bousquet L., Ouabdesselam F., Richier J.-L. & Zuanon N. (1999). Lutess: a Specification-driven Testing Environment for

Synchronous Software. In *21st International Conference on Software Engineering*, p. 267–276: ACM Press.

[Duke & Harisson, 1997] Duke D. & Harisson M. (1997). Mapping user requirements to implementations. *Software Engineering Journal*, **10**, p. 54–75.

[Duke & Harrison, 1993] Duke D. J. & Harrison M. D. (1993). Abstract Interaction Objects. *Comput. Graph. Forum*, **12**(3), p. 25–36.

[Duke & Harrison, 1995] Duke J. & Harrison M. (1995). Event model of human system interaction. *Software Engineering*, **-**, p. –.

[Emerson & Halpern, 1986] Emerson E. A. & Halpern J. Y. (1986). "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *J. ACM*, **33**(1), p. 151–178.

[Ezzedine & Kolski, 2005] Ezzedine H. & Kolski C. (2005). Modelling of cognitive activity during normal and abnormal situations using Object Petri Nets, application to a supervision system. *Cognition, Technology and Work*, **7**, p. 167–181.

[Faconti & Paternó, 1990] Faconti G. & Paternó F. (1990). An Approach to the Formal Specification of the components of an interaction. In *Eurographics*.

[Fekete & Girard, 2001] Fekete D. & Girard P. (2001). *Environnements de développement des systèmes interactifs*, chapter 1, p. 23–52. Hermès Science, Interaction homme-machine pour les SI.

[Fekete, 1996] Fekete J. (1996). *Un modèle multicouche pour la construction d'application graphique interactive*. PhD thesis, Université Paris XI.

[Fernandez *et al.*, 1992] Fernandez J.-C., Garavel H., Mounier L., Rasse A., Rodríguez C. & Sifakis J. (1992). A toolbox for the verification of lotos programs. In *ICSE'92, 14th International conference on software engineering*, Malbourne.

[Ferrante *et al.*, 1987] Ferrante J., Ottenstein K. J. & Warren J. D. (1987). The program dependence graph and its use in optimization. *ACM TOPL*, **9**, p. 319–349.

[Flanagan *et al.*, 2002] Flanagan C., Leino K. R. M., Lillibridge M., Nelson G., Saxe J. B. & Stata R. (2002). Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37-5, p. 234–245.

[Foundation, 1990] Foundation C. O. S. (1990). *OSF/Motif programmer's guide: revision 1.0*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN : 0-13-640525-8.

[Gamboa & Scapin, 1997] Gamboa R. F. & Scapin D. (1997). Editing MAD* Task Description for Specifying User Interfaces at both Semantic and Presentation Levels. In S. Verlag, Ed., *Proceedings DSV-IS'97*, p. 193–208.

[Gannod & Cheng, 1999] Gannod G. C. & Cheng B. H. C. (1999). A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques. In *Working Conference on Reverse Engineering*, p. 77–88.

[Ghezzi *et al.*, 1990] Ghezzi C., Madrioli D. & . A. M. (1990). TRIO: A Logic Language for Executable Specifications of Real- Time Systems. *Journal of Systems Software*, **12(2)**, p. 107–123.

[GKS, 1985] GKS (1985). *Graphical Kernel System - functional description*. Rapport interne, IS 7942, ISO.

[Goldberg, 1984] Goldberg A. (1984). *SMALLTALK-80: the interactive programming environment*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN : 0-201-11372-4.

[Gopal & Schach, 1989] Gopal R. & Schach S. R. (1989). Using automatic program decomposition techniques in software maintenance tools. In *Proceedings of ICSM'89, International Conference on Software Maintenance*, p. 132–141.

[Gram & Cockton, 1997] Gram C. & Cockton G. (1997). *Design principles for interactive software*. London, UK: Chapman & Hall, Ltd. ISBN : 0-412-72470-7.

[Gray *et al.*, 1994] Gray P., England D. & McGowan S. (1994). *XUAN : Enhancing the UAN to capture temporal relation among actions*. Rapport interne, Departement Research Report IS-94-02, Departement of Computing Science, University of Glasgow.

[Grothoff *et al.*, 2001] Grothoff C., Palsberg J. & Vitek J. (2001). Encapsulating Objects with Confined Types. In *Conference on Object-Oriented*, p. 241–253.

[Halbwachs *et al.*, 1991] Halbwachs N., Caspi P., Raymond P. & Pilaud D. (1991). The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, **79**(9), p. 1305–1320.

[Hall, 1991] Hall P. (1991). ASF/Motif Programmer's Guide. Open source foundation.

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

[Harston & Gray, 1992] Harston H. & Gray P. (1992). Temporal aspects of tasks in the user action notation. *Human-Computer interaction*, **7**, p. 1–45.

[Hartson & Hix, 1989] Hartson H. & Hix D. (1989). Towards emprically derived methodologies and tools for human-computer interface development. *International Journal of Man-Machine Studies*, **31**, p. 477–494.

[Hatcliff *et al.*, 1999] Hatcliff J., Corbett J. C., Dwyer M. B., Sokolowski S. & Zheng H. (1999). A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, p. 1–18, London, UK: Springer-Verlag.

[Hatcliff *et al.*, 2000a] Hatcliff J., Dwyer M. B. & Zheng H. (2000a). Slicing software for model construction. *Higher-Order and Symbolic Computation*, **13**(4), p. 315–353.

[Hatcliff *et al.*, 2000b] Hatcliff J., Dwyer M. B. & Zheng H. (2000b). Slicing software for model construction. *Higher-Order and Symbolic Computation*, **13**(4), p. 315–353.

[Hausler, 1989] Hausler P. A. (1989). Denotational program slicing. In *22 Annual Hawaii International Conference on System Sciences*, volume II, p. 486–495.

[Hill, 1992] Hill R. D. (1992). The Abstraction-Link-View Paradigm: Using constraints to connect user interfaces to applications. In *SIGCHI Conference on Human Factors and Computing Systems*, p. 335–342, Monterey, California, United States.

[Hoare, 1985] Hoare C. (1985). *Communicating Sequential Processes*. Prentice Hall International.

[Hoare, 1978] Hoare C. A. R. (1978). Communicating sequential processes. *Communication of ACM*, **21**(8), p. 666–677.

[Horwitz *et al.*, 1988] Horwitz S., Reps T. & Binkley D. (1988). Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, p. 35–46, Atlanta, GA.

[ISO84, 1984] ISO84 (1984). ISO - Information Processing Systems - Definition of the temporal ordering specification language LOTOS. Rapport Interne, TC 97/16 N1987, ISO.

[Jacko & Sears, 2003] J. A. Jacko & A. Sears, Eds. (2003). *The human-computer interaction handbook: fundamentals, evolving technologies and emerging applications*. Mahwah, NJ, USA: Lawrence Erlbaum Associates, Inc. ISBN : 0-8058-3838-4.

[Jacob, 1983] Jacob R. J. K. (1983). Using Formal Specifications in the Design of a Human-Computer Interface. *Communications of the ACM*, **26**(4), p. 259–264.

[Jambon, 1996] Jambon F. (1996). *Erreurs et interruptions du point de vue de l'ingénierie de l'interaction homme-machine*. PhD thesis, Université Joseph Fourier (Grenoble 1).

[Jambon, 2002] Jambon F. (2002). From Formal Specifications to Secure Implementations. In *Proceedings of Computer-Aided Design of User Interfaces (CADUI'2002)*, p. 43–54, Valenciennes, France: Kluwer Academics.

[Jambon *et al.*, 2001] Jambon F., Girard P. & Aït-Ameur Y. (2001). Spécifications des systèmes interactifs. In C. Kolski, Ed., *Analyse et Conception de l'IHM pour les S.I.*, volume 1, p. 175–206: Hermès Science.

[Javahery *et al.*, 2003] Javahery H., Seffah A., Engelberg D. & Sinnig D. (2003). *Multiple User Interfaces: Multiple-Devices, Cross-Platform and Context-Awareness*, chapter Migrating User Interfaces between Platforms Using HCI Patterns, p. 241–259. Wiley.

[Jones, 1990] Jones C. B. (1990). *Systematic Software Development using VDM*. Upper Saddle River, NJ 07458, USA: Prentice-Hall. ISBN : 0-13-880733-7.

[Jourde *et al.*, 2006] Jourde F., Nigay L. & Parissis I. (2006). Test formel de systèmes interactifs multimodaux : couplage ICARE-Lutess. In *ICSSEA 2006, The 19th International Conference on Software Systems Engineering and their Applications*, Paris, France.

[Kamel, 2006] Kamel N. (2006). *Un cadre formel générique pour la modélisation d'IHM Multi-modales. Cas de la multi-modalité en entrée.* PhD thesis, Ecole Nationale Supérieure de Mécanique et d'Aérotechnique (ENSMA) et Faculté des Sciences Fondmentales et Appliquées de Poitiers.

[Kolski, 1993] Kolski C. (1993). *Ingénierie des interfaces homme-machine, conception et évaluation.* Paris: Hermès. ISBN : 2-86601-377-8.

[Kolski, 1998] Kolski C. (1998). A call for answers around the proposition of an HCI-enriched model. *ACM SIGSOFT Software Engineering Notes*, **3**, p. 93–96.

[Kovács *et al.*, 1996] Kovács G., Magyar F. & Gyimóthy T. (1996). *Static Slicing of JAVA Programs*. Rapport interne TR-96-108, Hungarian Academy of Sciences, József Attila University, Hungary.

[Kurshan, 1994] Kurshan R. P. (1994). *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton, NJ, USA: Princeton University Press. ISBN : 0-691-03436-2.

[Larsen & Harrold, 1996] Larsen L. & Harrold M. J. (1996). Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, p. 495–505, Washington, DC, USA: IEEE Computer Society.

[Leuschel & Turner, 2005] Leuschel M. & Turner E. (2005). Visualising larger state spaces in ProB. In *ZB05*, p. 6–23.

[Liang & Harrold, 1998] Liang D. & Harrold M. J. (1998). Slicing Objects Using System Dependence Graphs. In *ICSM : IEEE International Conference on Software Maintenance*, p. 358–367.

[Limbourg & Vanderdonckt, 2003] Limbourg Q. & Vanderdonckt J. (2003). *Comparing Task Models for User Interface Design*, chapter 6, p.̃-. Lawrence Erlbaum Associates.

[Loer & Harrison, 2000] Loer K. & Harrison M. D. (2000). Formal Interactive Systems Analysis and Usability Inspection Methods: Two Incompatible Worlds ? In *DSV-IS : International Workshop on Design, Specification and Verification of Interactive Systems*, p. 169–190.

[Luyten *et al.*, 2003] Luyten K., Clerckx T., Coninx K. & Vanderdonckt J. (2003). Derivation of a Dialog Model from a Task Model by Activity Chain Extraction. In *Interactive Systems: Design, Specification, and Verification, 10th International Workshop DSV-IS*, Funchal, Madeira Island, Portugal.

[Lyle, 1984] Lyle J. R. (1984). *Evaluating variations on program slicing for debugging (data-flow, ada)*. PhD thesis, College Park, MD, USA.

[MacColl & D. Carrington, 1998] MacColl I. & D. Carrington D. (1998). Testing MATIS : a case study on specification-based testing of interactive systems. In *FAHCI : Formal Aspects of the Human Computer Interface (Conference)*, p. 57–69. ISBN : 0-86339-7948.

[Mahajan & Shneiderman, 1997] Mahajan R. & Shneiderman B. (1997). Visual and Textual Consistency Checking Tools for Graphical User Interfaces. *IEEE Trans. Softw. Eng., IEEE Press*, **23**(11), p. 722–735. issn : 0098-5589, Piscataway, NJ, USA.

[Manna & Pnueli, 1992] Manna Z. & Pnueli A. (1992). *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer-Verlag.

[Markopoulos, 1995] Markopoulos P. (1995). On the Expression of Interaction Properties within an Interactor Model. In B. R. Palanque P, Ed., *Design, Specification, Verification of Interactive Systems (DS-VIS'95)*, p. 294–311: Springer Wien.

[Markopoulos, 1997] Markopoulos P. (1997). *A Compositional Model for the Formal Specification of User Interface Software*. PhD thesis, University of London.

[Markopoulos *et al.*, 1996] Markopoulos P., Rowson J. & Johnson P. (1996). Dialogue Modelling in the Framework of an Interactor Model. In *DSV-IS'96 : International Workshop on Design, Specification and Verification of Interactive Systems*.

[Martin, 1998] Martin J. (1998). TYCOON : Theoretical Framework and Software Tools for Multimodal Interfaces. In *Intelligence and Multimodality in Multimedia interfaces. (ed.) John Lee, AAAI Press*.

[Martin & Broule, 1993] Martin J. & Broule D. (1993). Types and goals of cooperation between modalities. In *Proceedings of the 5th Conf. on Human-Computer Interaction (IHM'93)*, p. 17–22, Lyon, France.

[McDermid & Ripkin, 1984] McDermid J. & Ripkin K. (1984). *Life Cycle Support in the ADA environnement*. Cambridge University Press.

[Memon, 2001] Memon A. (2001). *A Comprehensive Framework for Testing Graphical User Interfaces*. PhD thesis, Department of Computer Science, University of Pittsburgh.

[Memon *et al.*, 2003] Memon A., Banerjee I. & Nagarajan A. (2003). GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, p. 260–275, Washington, DC, USA: IEEE Computer Society. ISBN : 0-7695-2027-8.

[Memon, 2002] Memon A. M. (2002,). GUI Testing: Pitfalls and Process. *Computer, IEEE Computer Society Press*, **35**(8), p. 87–88. ISSN : 0018-9162, Los Alamitos, CA, USA.

[Metayer *et al.*, 2005] Metayer C., Abrial J.-R. & Voisin L. (2005). Event-B Language. RODIN Project Deliverable D7.

[Milner, 1980] Milner R. (1980). A Calculus of Communicating Systems. Lect. Notes in Computer Science vol.94, Springer-Verlag.

[Morasca *et al.*, 1996] Morasca S., Morzenti A. & SanPietro P. (1996). Generating functional test cases in-the-large for time-critical systems from logic-based specifications. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, p. 39–52, New York, NY, USA: ACM Press.

[Mori *et al.*, 2002] Mori G., Paternó F. & Santoro C. (2002). CTTE: support for developing and analyzing task models for interactive system design. *IEEE Trans. Softw. Eng.*, **28**(8), p. 797–813.

[Morzenti *et al.*, 1992] Morzenti A., Mandrioli D. & Ghezzi C. (1992). A Model Parametric Real-Time Logic. *ACM Transactions on Programming Languages and Systems*, **14**(4), p. 521–573.

[Muchnick, 1997] Muchnick S. S. (1997). *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN : 1-55860-320-4.

[Myers, 1995a] Myers B. A. (1995a). State of the art in user interface software tools. *Human-computer interaction: toward the year 2000*, p. 323–343.

[Myers, 1995b] Myers B. A. (1995b). User interface software tools. *ACM Trans. Comput.-Hum. Interact.*, **2**(1), p. 64–103.

[Myers *et al.*, 1996] Myers B. A., Hollan J. D. & Cruz I. F. (1996). Strategic Directions in Human-Computer Interaction. *ACM Comput. Surv.*, **28**(4), p. 794–809.

[Myers *et al.*, 1997] Myers B. A., McDaniel R. G., Miller R. C., Ferrency A. S., Faulring A., Kyle B. D., Mickish A., Klimovitski A. & Doane P. (1997). The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Trans. Softw. Eng.*, **23**(6), p. 347–365.

[Myers & Rosson, 1992] Myers B. A. & Rosson M. B. (1992). Survey on User Interface Programming. In *Proceedings of the Conference on Human Factors in Computing Systems*, p. 195–202, Monterey, CA, USA.

[Méry & Cansell, 2004] Méry D. & Cansell D. (2004). Tutorial on the Event-based B method : Concepts and case studies. In *Logics of Formal Software Specification Languages - LFSL'2004*, The High Tatras, Slovakia: Dines Bjoerner and Martin Henso.

[Müller *et al.*, 2000] Müller H. A., Jahnke J. H., Smith D. B., Storey M.-A. D., Tilley S. R. & Wong K. (2000). Reverse engineering: a roadmap. In *ICSE - Future of SE Track*, p. 47–60.

[Navarre, 2001] Navarre D. (2001). *Contribution à l'ingénierie en Interaction Homme-Machine*. PhD thesis, UYniversité de Toulouse.

[Navarre *et al.*, 2005] Navarre D., Palanque P., Bastide R., Schyn A., Winckler M. A., Nedel L. & Freitas C. (2005). A Formal Description of Multimodal Interaction Techniques for Immersive Virtual Reality Applications. In M. F. Costabile & F. Paternò, Eds., *Human-Computer Interaction - INTERACT 2005: IFIP TC13 International Conference, Rome, Italy, 12/09/05-16/09/05*, p. 170: Springer-Verlag GmbH.

[Nielson *et al.*, 1999] Nielson F., Nielson H. R. & Hankin C. (1999). *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN : 3540654100.

[Nigay & Coutaz, 1997] Nigay L. & Coutaz J. (1997). *Multifeature Systems: The CARE Properties and Their Impact on Software Design*, In *Intelligence and Multimodality in Multimedia Interfaces: Research and Applications*, chapter Nine, p.˜-. -. This book is available in ISO 9660 CD format only 16 pages.

[Norman, 1986] Norman D. (1986). *User Centered System Design*. Lawrence Erlbaum Associates.

[Normand, 1992] Normand V. (1992). *Le modèle SIROPO : de la spécification conceptuelle des interfaces à leur réalisation*. PhD thesis, Université Joseph Fourier, Grenoble.

[Olsen, 1990] Olsen D. (1990). Propositional production systems for dialog description. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, p. 57–64, New York, NY, USA: ACM Press.

[Ottenstein & Ottenstein, 1984] Ottenstein K. J. & Ottenstein L. M. (1984). The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, p. 177–184, New York, NY, USA: ACM Press.

[Ousterhout, 1994] Ousterhout J. K. (1994). *Tcl and the Tk Toolkit*. Addison Wesley. ISBN : 0-201-63337-X.

[Paganelli & Paternò, 2003] Paganelli L. & Paternò F. (2003). A Tool for Creating Design Models from Web Site Code. *International Journal of Software Engineering and Knowledge Engineering*, **13**(2), p. 169–189.

[Palanque, 1992] Palanque P. (1992). *Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigés par l'utilisateur*. PhD thesis, Université de Toulouse.

[Palanque *et al.*, 1995] Palanque P., Bastide R. & Sengès V. (1995). Validating interactive system design through the verification of formal task and system models. In *Engineering for Human-Computer Interaction*: Chapman & Hall.

[Palanque & Schyn, 2003] Palanque P. & Schyn A. (2003). A Model-Based Approach for Engineering Multimodal Interactive Systems. In M. Rauterberg, M. Menozzi & J. Wesson, Eds., *INTERACT'2003 : Ninth IFIP TC13 International Conference on Human-Computer Interaction , Zurich, Switzerland, 01/09/03-05/09/03*, p. 543–550: IOS Press.

[Party, 1999] Party G. (1999). *Contribution à la conception du dialogue Homme-Machine dans les applications graphiques interactives de conception technique : le système GIPSE*. PhD thesis, Université de Poitiers.

[Paternò, 1993] Paternò F. (1993). A formal specification of appearance and behaviour of visual environments. *Software Engineering Journal*, **Volume 8, Issue 3**, p. 154–164.

[Paternò, 1995] Paternò F. (1995). *A Method for formal specification and verification of interactive system*. PhD thesis, Departement of Computer System, York University.

[Paternò, 2001] Paternò F. (2001). *Model-Based Design and Evaluation of Interactive Applications*. Springer.

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

[Paternò & Mezzanotte, 1994] Paternò F. & Mezzanotte M. (1994). *Analysing MATIS by interactors and ACTL*. Rapport interne, Rapport Interne, Amodeus Esprit Basic Research Project 7040, System Modelling/WP36.

[Paternò *et al.*, 2001] Paternò F., Mori G. & Galiberti R. (2001). CTTE: an environment for analysis and development of task models of cooperative applications. In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, p. 21–22, New York, NY, USA: ACM Press.

[Perry, 1995] Perry W. (1995). *Effective methods for software testing*. Somerset, NJ, USA: Wiley-QED Publishing. isbn : 0-471-06097-6.

[Peterson, 1981] Peterson J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN : 0136619835.

[Pfaff, 1985] Pfaff G. E. (1985). *User Interface Management Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN : 038713803X.

[Pinheiro da Silva, 2000] Pinheiro da Silva P. (2000). User Interface Declarative Models and Development Environments: A Survey. In P. Palanque & F. Paternò, Eds., *Proceedings of DSV-IS2000*, volume 1946 of *LNCS*, p. 207–226, Limerick, Ireland: Springer-Verlag.

[Pnueli, 1977] Pnueli A. (1977). The Temporal Logic of Programs. In *FOCS : Annual IEEE Symposium on Foundations of Computer Science*, p. 46–57.

[Puerta, 1996] Puerta A. R. (1996). The MECANO Project: Comprehensive and Integrated Support for Model-Based Interface Development. In *Computer-Aided Design of User Interfaces (CADUI)*, p. 19–36, Namur, Belgium: Presses Universitaires de Namur.

[Queille & Sifakis, 1981] Queille J. & Sifakis J. (1981). Specification and verification of concurent systems in Caesar. In *5th International Symposium on Programming*.

[Quemada *et al.*, 1993] Quemada J., Pires J., Marias J. & Azcorra A. (1993). *Using formal description techniques - an introduction to Estelle, Lotos and SDL*, chapter Introduction to LOTOS, p. 47–83. K. Turner, Wiley.

[Reps & Bricker, 1989] Reps T. & Bricker T. (1989). Illustrating interference in interfering versions of programs. In *Proceedings of the 2nd International Workshop on Software configuration management*, p. 46–55, New York, NY, USA: ACM.

[Richardson, 1994] Richardson D. J. (1994). TAOS: Testing with Analysis and Oracle Support. In *International Symposium on Software Testing and Analysis*, p. 138–153.

[Robert, 2008] L. Robert, Ed. (2008). *Le Nouveau Petit Robert de la langue française.* S.C.C, Les dictionnaires Robert-Canada.

[Roche, 1998] Roche P. (1998). *Modélisation et validation d'interface homme-machine*. PhD thesis, École Nationale Supérieure de l'Aéronautique et de l'Espace.

[Rousseau *et al.*, 2004] Rousseau C., Bellik Y., Vernier F. & Bazalgette D. (2004). Architecture framework for output multimodal systems design. In *OZCHI'04*, Wollongong, Australia.

[Royce, 1987] Royce W. W. (1987). Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering, Monterey, California, United States*, p. 328–338, Los Alamitos, CA, USA: IEEE Computer Society Press. isbn : 0-89791-216-0.

[Ryan *et al.*, 1991] Ryan M., Fiadeiro J. & Maibaum T. (1991). Sharing actions and attributes in. modal action logic. In *Theoretical Aspects of Computer Software*, volume 526, p. 569–593: LNCS, Springer Verlag.

[Scapin & Pierret-Golbreich, 1989] Scapin D. & Pierret-Golbreich C. (1989). Mad : Une méthode analytique de description de tâches. In *Colloque sur l'Ingénierie des Interfaces Homme-Machine (IHM'89)*, p. 131–148, Sophia-Antipolis, France.

[Schmidt & Steffen, 1998] Schmidt D. A. & Steffen B. (1998). Program Analysis as Model Checking of Abstract Interpretations. In *Static Analysis Symposium*, p. 351–380.

[Schmucker, 1987] Schmucker K. J. (1987). MacApp: An application framework. *Human-Computer Interaction: a multidisciplinary approach*, **11**(8), 591–594.

[Schneider-Hufschmidt *et al.*, 1993] M. Schneider-Hufschmidt, U. Malinowski & T. Kuhme, Eds. (1993). *Adaptive User Interfaces: Principles and Practice*. New York, NY, USA: Elsevier Science Inc. ISBN : 0444815457.

[Schnoebelen *et al.*, 1999] Schnoebelen P., Bérard B., Bidoit M., Laroussinie F. & Petit A. (1999). *Vérification de logiciels : Techniques et outils du Model-checking*. Vuibert.

[Schyn, 2005] Schyn A. (2005). *Une approche fondée sur les modèles pour l'ingénierie des systèmes interactifs multimodaux*. PhD thesis, Toulouse III.

[Schyn *et al.*, 2003] Schyn A., Navarre D., Palanque P. & Porcher Nedel L. (2003). Description Formelle d'une Technique d'Interaction Multimodale dans une Application de Réalité Virtuelle Immersive. In *IHM'2003 : 15th French Speaking conference on human-computer interaction, Caen, France, 24/11/03-28/11/03*, p. 150–157: ACM Press.

[Shepherd, 1989] Shepherd A. (1989). *Analysis and training in information technology tasks*. Chichester, USA: Ellis Horwood, Books in Information Technology.

[Shneiderman, 1987] Shneiderman B. (1987). Direct manipulation: A step beyond programming languages. *Human-computer interaction: a multidisciplinary approach, Morgan Kaufmann Publishers Inc.*, **-**, p. 461–467. isbn : 0-934613-24-9, San Francisco, CA, USA.

[Sibertin-Blanc, 1985] Sibertin-Blanc C. (1985). High Level Petri Nets with Data Structure. In *6th European Workshop on Petri Nets and Applications*, Espoo, Finland.

[Silva *et al.*, 2006] Silva J. C., Campos J. C. & Saraiva J. (2006). Models for the Reverse Engineering of Java/Swing Applications. In J. M. Favre, D. Gasevic, R. Lämmel & A. Winter, Eds., *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies (ateM 2006) for Reverse Engineering*, number 1/2006 in Informatik-Bericht series: Johannes Gutenberg-Universität Mainz, Institut für Informatik - FB 8. ISSN: 0931-9972.

[Spivey, 1988] Spivey J. M. (1988). *Understanding Z: a specification language and its formal semantics*. New York, NY, USA: Cambridge University Press. ISBN : 0-521-33429-2.

[Stouls & Potet, 2004] Stouls N. & Potet M.-L. (2004). Explicitation du contrôle de développements B évènementiels. In *AFADL'2004 : Approches Formelles dans l'Assistance au Développement de Logiciels*, p. 13–28.

[Suberri, 2003] Suberri G. (2003). *Automated Reverse Engineering of Graphical User Interfaces*. Rapport interne, University of Maryland, Departement of Computer Science. Advisor : Memon.

[Systems, 1984] Systems I. I. P. (1984). *Definition of the Temporal Ordering Specification Language LOTOS*. Rapport interne, TC 97/16 N1987, ISO.

[Szekely, 1996] Szekely P. (1996). Retrospective and Challenges for Model-Based Interface Development. In F. Bodart & J. Vanderdonckt, Eds., *Design, Specification and Verification of Interactive Systems '96*, p. 1–27, Wien: Springer-Verlag.

[Szekely *et al.*, 1995] Szekely P. A., Sukaviriya P. N., Castells P., Muthukumarasamy J. & Salcher E. (1995). Declarative interface models for user interface construction tools: the MASTERMIND approach. In *EHCI : Engineering for Human-Computer Interaction, Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, p. 120–150.

[Tabary, 2001] Tabary D. (2001). *Contribution à TOOD, Une méthode à base de modèles pour la spécification et la conception des systèmes interactifs*. PhD thesis, Université de Valenciennes.

[Tarby, 1993] Tarby J.-C. (1993). *Gestion Automatique de Dialogue Homme-Machine à partir de spécifications conceptuelles*. PhD thesis, Université Toulouse I.

[Texier, 2000] Texier G. (2000). *Contribuiton à l'ingénierie des systèmes interactifs : un environnement de conception graphique d'applications spécialisées de conception*. PhD thesis, Université de Poitiers.

[Tip, 1995] Tip F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, **3**, p. 121–189.

[Vanderdonckt *et al.*, 2001] Vanderdonckt J., Bouillon L. & Souchon N. (2001). Flexible Reverse Engineering of Web Pages with VAQUITA, IEEE Computer Society Press. In *IEEE 8th Working Conference on Reverse Engineering WCRE'2001*, p. 241–248, Stuttgart.

[VERBATIM, 2003 2007] VERBATIM (2003-2007). Projet Exploratoire RNRT-ANR, VERification Biformelle et Automatisation du Test d'Interfaces Multimodales, Partenaires : CLEARSY - CLIPS/IHM - FT R&D - LISI/ENSMA - LSR/VASCO - ONERA - SILICOM/AQL. Publications accessible sur : http://iihm.imag.fr/nigay/VERBATIM/.

[Walkinshaw & Roper, 2003] Walkinshaw M. & Roper M. (2003). The Java system dependence graph. In *Third IEEE International Workshop on Source Code Analysis and Manipulation*, p. p. 55.

[Wasserman, 1981] Wasserman A. I. (1981). User Software Engineering and the design of interactive systems. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, p. 387–393, Piscataway, NJ, USA: IEEE Press.

[Weiser, 1979] Weiser M. (1979). *Program Slices : Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, MI.

[Weiser, 1981] Weiser M. (1981). Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, p. 439–449: IEEE Computer Society Press.

[Wick *et al.*, 1993] Wick D. T., Shehad N. M. & Hajare A. R. (1993). Testing the Human Computer Interface for the Telerobotic Assembly of the Space Station. In *HCI (1)*, p. 213–218.

[Wiecha *et al.*, 1990] Wiecha C., Bennett W., Boies S., Gould J. & Greene S. (1990). ITS: a tool for rapidly developing interactive applications. *ACM Trans. Inf. Syst.*, **8**(3), p. 204–236.

[Woods, 1986] Woods W. A. (1986). Transition network grammars for natural language analysis. *Readings in natural language processing*, p. p. 71–87.

[Wright, 1991] Wright D. A. (1991). A new technique for strictness analysis. In *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Advances in distributed computing (ADC) and colloquium on combining paradigms for software development (CCPSD): Vol. 2*, p. 235–258, New York, NY, USA: Springer-Verlag New York, Inc.

[Zhao *et al.*, 1996] Zhao J., Cheng J. & Ushijim K. (1996). Static Slicing of Concurrent Object-Oriented Programs. In *COMPSAC '96: Proceedings of the 20th Conference on Computer Software and Applications*, p. 312, Washington, DC, USA: IEEE Computer Society.

Chapitre A

Présentation de la notation ConcurTaskTrees

Cette annexe présente la notation ConcurTaskTrees (CTT) définie par [Paternò, 2001].

A.1 La sémantique de la notation CTT

CTT est une notation de spécification de tâches fondée sur une structuration hiérarchique. Une tâche CTT définit comment un utilisateur peut atteindre un but en utilisant une application interactive. Une tâche CTT est définie par différentes caractéristiques :

- un *nom* qui identifie la tâche ;
- un type de tâche ;
- un ensemble d'objets ;
- des caractéristiques associées aux opérateurs temporels.

Les types de tâches

CTT distingue quatre types de tâches qui permettent de construire l'arbre de tâches.

1. Les **tâches utilisateur** () correspondent aux tâches accomplies par l'utilisateur indépendamment de toute interaction sur le système : il s'agit majoritairement de tâches cognitives qui témoignent d'une réflexion de l'utilisateur.
2. Les **tâches application** () sont des tâches effectuées par le système. Ces tâches sont utilisées pour rendre compte de l'état du système.
3. Les **tâches interaction** () correspondent aux tâches qui caractérisent une interaction de l'utilisateur avec le système (clic souris, saisie clavier, etc.).
4. Les **tâches abstraites** () correspondent à des tâches dont la décomposition (ou le raffinement) est constituée par des tâches appartenant à plusieurs des types pré-

cédemment décrits. Il s'agit de tâches de haut niveau dont la portée n'est pas exactement définie.

Les objets de tâches

Les objets de tâches sont des entités manipulés pour accomplir une tâche. Elles sont regroupées en deux catégories :

- les objets dits perçus qui peuvent représenter des objets pour le retour d'information (champs de texte, arbre, etc.) ou des techniques d'interaction pour agir sur le système (bouton, zone de saisie) ;
- et les objets application qui représentent des entités du noyau fonctionnel pouvant être associées à des objets perçus afin de refléter le plus fidèlement possible l'état du noyau sur l'interface.

Cependant, la sémantique de ces objets n'est pas précisément définie au sein de CTT.

Les opérateurs temporels

Les opérateurs temporels définis par la notation CTT et issus du langage Lotos [ISO84, 1984] permettent d'établir des relations temporelles entre deux tâches de l'arbre CTT. Dans ce qui suit, T_1 et T_2 dénotent des tâches de l'un des types définis par CTT (utilisateur, application, interaction ou abstraite). Les différents opérateurs de la notation CTT sont les suivants :

- Opérateur d'**Activation** : $T_1 >> T_2$. 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.
- Opérateur de **Choix** : $T_1[]T_2$. Cet opérateur signifie que deux tâches sont disponibles, mais une seule tâche est activable. Si T_1 est activée alors T_2 n'est plus activable et réciproquement.
- Opérateur **Ordre indépendant** : $T_1 \models |T_2$. T_1 et T_2 sont activables dans n'importe quel ordre. Cet opérateur peut être exprimé à partir des opérateurs précédents : $(T_1 >> T_2)[](T_2 >> T_1)$.
- Opérateur d'**Interruption** : $T_1| > T_2$. Cet opérateur signifie que si T_1 est en cours d'exécution, la tâche T_2 peut l'interrompre. Une fois T_2 terminée, T_1 reprend son traitement.
- Opérateur de **Désactivation** : $T_1[> T_2$. Cet opérateur signifie que si T_1 est en cours d'exécution, la tâche T_2 peut la désactiver.
- Opérateur d'**Activation avec passage d'information** : $T_1[] >> T_2$. Cet opérateur est identique à l'opérateur d'activation avec un passage d'information de T_1 vers T_2 au moment de l'activation.
- Opérateur de **Synchronisation** : $T_1|[]|T_2$. Cet opérateur désigne une synchronisation entre des objets des tâches T_1 et T_2.

Caractéristiques de tâches

En plus des opérateurs temporels, il est possible d'associer des caractéristiques temporelles aux tâches. Considérons une tâche $T_{1]}$ de type quelconque.

- **Itération** : T_1^*. La tâche T_1 se répète indéfiniment tant qu'une autre tâche ne la désactive pas. L'expression $T_1* >> T_2$ n'est donc pas autorisée, car T_2 n'est jamais atteignable ;
- Itération finie : T_1^N. La tâche T_1 se répète N fois à moins qu'une autre tâche ne la désactive ;
- Optionnelle : $[T_1]$. Cette caractéristique indique que la tâche T_1 n'est pas obligatoirement exécutable. Les tâches à gauche ou à droite des opérateurs $|>$, $[>$ et $[]$ ne peuvent pas être optionnelles.

A.2 L'outil CTTE

L'outil CTTE[1] est un environnement permettant la description de modèles de tâches suivant la notation CTT. Cet environnement permet de vérifier automatiquement la syntaxe d'un arbre CTT (combinaison de tâches avec les opérateurs temporels), d'ajouter des informations sur les modèles de tâches, de vérifier le comportement décrit par un modèle, de construire des scénarii, etc.

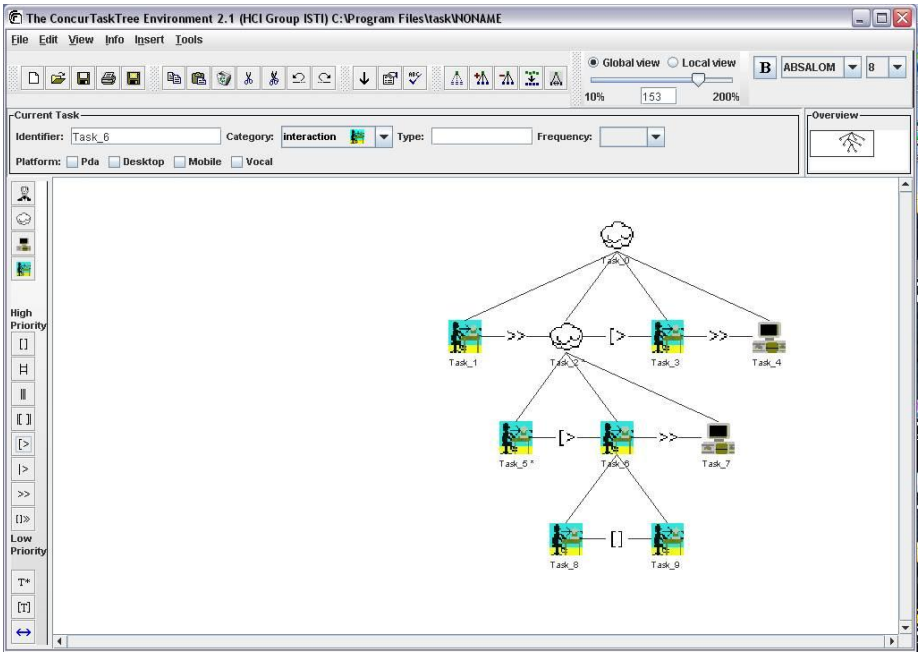


FIG. A.1 – Capture d'écran de l'éditeur de modèles de tâches CTT.

L'éditeur de modèle de tâches

La figure A.1 présente l'éditeur de modèles de tâches CTT. Cette capture d'écran présente le modèle de tâches traité dans ce mémoire. A gauche sur l'interface se trouve une palette d'outils permettant d'ajouter une tâche (Fig.A.1,❶) au modèle, d'introduire un opérateur temporel (Fig.A.1,❷) ou de modifier la caractéristique d'une tâche (Fig.A.1,❸). En haut de l'interface, une palette d'outils (Fig.A.1,❹) permet de paramétrer le modèle de

[1]Téléchargement de l'outil : http://giove.cnuce.cnr.it/ctte.html

tâches en cours et enfin la partie centrale permet d'éditer le modèle en cours de construction (Fig.A.1,❺).

L'outil CTTE n'exploite pas complètement la sémantique de la notation CTT. Notamment, l'itération finie n'est pas prise en charge.

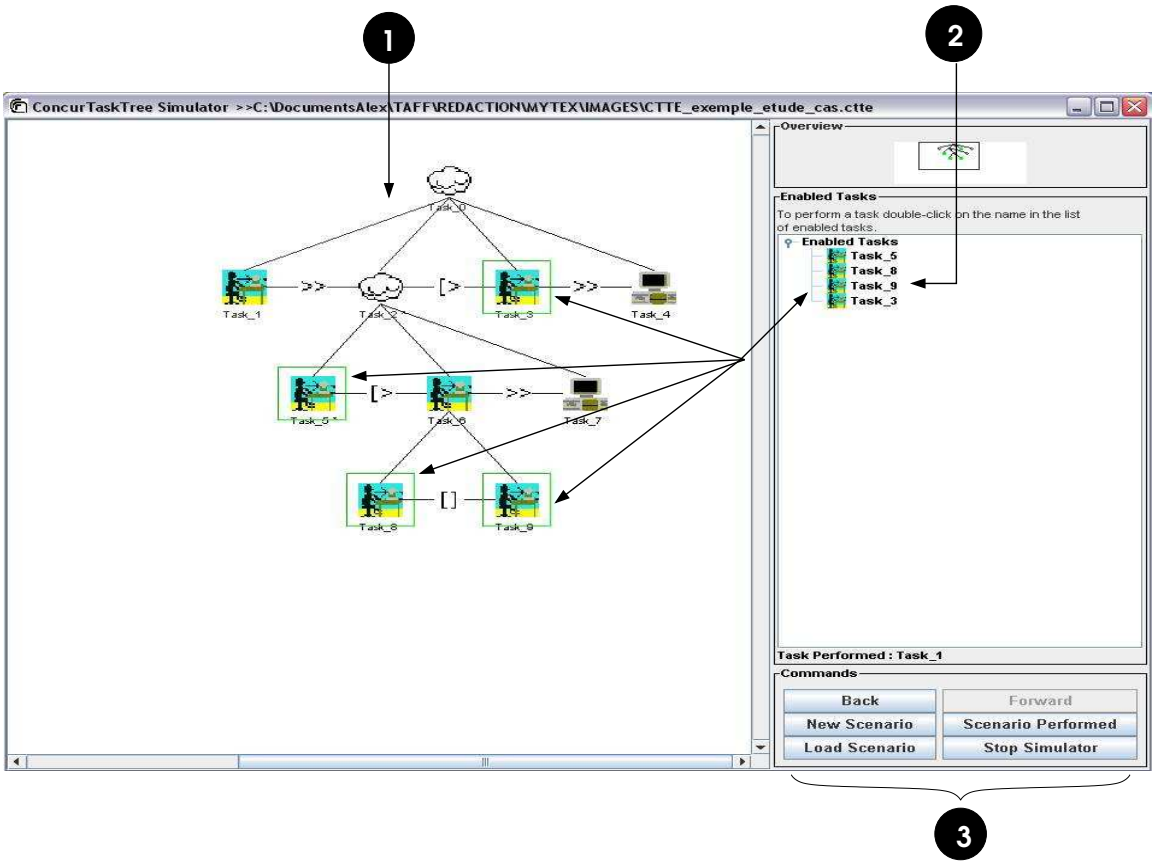


FIG. A.2 – Capture d'écran d'un simulateur de modèles de tâches CTT.

Le simulateur de modèles de tâches

CTTE propose un outil de simulation permettant de vérifier le comportement d'un modèle. Cette simulation se base sur le calcul des ETS (Enabled Task Sets ou Ensemble des Tâches Activables) à partir des contraintes de précédences introduites par les opérateurs temporels. La figure A.2 présente succinctement cet outil. La partie centrale (Fig.A.2,❶) permet la visualisation du modèle de tâches en cours de simulation. La partie droite donne la liste des tâches activables à l'instant donné (Fig.A.2,❷). Ces tâches activables sont mises en évidence sur la visualisation du modèle par un cadre de couleur. Enfin, un ensemble d'options (Fig.A.2,❸) sont accessibles sur l'interface : sauvegarde d'un scénario, retour à l'état précédent, etc.

Chapitre B

Modèles B événementiel

Cette annexe présente le code B événementiel complet des modèles utilisés dans l'approche de validation proposée dans ce mémoire.

Modèle B_{Swing}

Les figures B.1 et B.2 présentent le code B événementiel du modèle B_{Swing} qui modélise la boîte à outils Swing du langage Java. Cette abstraction permet de représenter les instances de widgets et la structure hiérarchique de widgets qui composent l'interface (`WIDGETS_parents`).

```
CONTEXT BSwing /* Modélisation de la boîte à outils Swing */

SETS

WIDGETS, TEXT
/* Définition d'ensembles abstraits */
CONSTANTS

empty, full, pc, gauche, boutonsPanel, inputPanel, input, ED, DE, droit, outputPanel, output,
WIDGETS_parents, JFrame, JTextfiels, JButton, JPanel

PROPERTIES

/* Propriétés de typages */
empty ∈ TEXT ∧ full ∈ TEXT ∧TEXT={empty,full} ∧ empty≠full ∧

pc ∈ WIDGETS ∧ gauche ∈ WIDGETS ∧ boutonsPanel ∈ WIDGETS ∧ inputPanel ∈ WIDGETS
∧ input ∈ WIDGETS ∧ ED ∈ WIDGETS ∧ DE ∈ WIDGETS ∧ droit ∈ WIDGETS
∧ outputPanel ∈ WIDGETS ∧ output ∈ WIDGETS ∧

WIDGETS={pc,gauche,boutonsPanel,droit,inputPanel,input,ED,DE,outputPanel,output} ∧

JButton ⊆ WIDGETS ∧ ⊆ WIDGETS ∧ JTextfield ⊆ WIDGETS ∧ JFrame ⊆ WIDGETS ∧ JFrame={pc} ∧
JTextfield={input,output} ∧ JButton={ED,DE} ∧ JPanel={gauche,inputPanel,droit,outputPanel, boutonsPanel} ∧

WIDGETS_parents ∈ WIDGETS → ℙ(WIDGETS) ∧
WIDGETS_parents={pc ↦ ∅,gauche ↦ {pc},boutonsPanel ↦ {pc},droit ↦ {pc}, inputPanel ↦ {pc,gauche},
               input ↦ {pc,gauche,inputPanel}, ED ↦ {pc,boutonsPanel}, DE ↦ {pc,boutonsPanel},
               outputPanel ↦ {pc,droit}, output ↦ {pc,droit,outputPanel}} ∧
```

FIG. B.1 – Modèle B_{Swing} (1)

```
PROPERTIES /* Suite du modèle B_Swing */                /* Contraintes d'intégrité */
∀w.(w ∈ {gauche, boutonsPanel, inputPanel, input, ED, DE, droit, outputPanel, output} ⇒ pc ≠ w) ∧
∀w.(w ∈ {boutonsPanel, inputPanel, input, ED, DE, droit, outputPanel, output} ⇒ gauche ≠ w) ∧
∀w.(w ∈ {inputPanel, input, ED, DE, droit, outputPanel, output} ⇒ boutonsPanel ≠ w) ∧
∀w.(w ∈ {input, ED, DE, droit, outputPanel, output} ⇒ inputPanel ≠ w) ∧
∀w.(w ∈ {ED, DE, droit, outputPanel, output} ⇒ input ≠ w) ∧
∀w.(w ∈ {DE, droit, outputPanel, output} ⇒ ED ≠ w) ∧
∀w.(w ∈ {droit, outputPanel, output} ⇒ DE ≠ w) ∧
∀w.(w ∈ {outputPanel, output} ⇒ droit ≠ w) ∧
outputPanel ≠ output ∧

WIDGETS= JFrame ∪ JButton ∪ JTextfield ∪ JPanel ∧

 /* Axiomes ajoutées afin de faciliter la preuve interactive de certaines OP */
∀ff,gg,ww,bb·( (ff ∈ WIDGETS → BOOL ∧ gg ∈ WIDGETS ⇸ BOOL ∧ bb ∈ BOOL
∧ ww ∈ WIDGETS ∧ gg(ww)=bb ∧ ww ∈ dom(gg))
⇒
((ff⩤gg)(ww)=bb)) ∧

∀ff,gg,ww,bb·((ff ∈ WIDGETS → BOOL ∧ gg ∈ WIDGETS ⇸ BOOL ∧ bb ∈ BOOL
∧ ww ∈ WIDGETS ∧ ff(ww)=bb ∧ ww /∈ dom(gg))
⇒
((ff⩤gg)(ww)=bb)) ∧

END
```

FIG. B.2 – Modèle B_{Swing}(2)

Modèle B_{Appl}

Les figures B.3 et B.4 présentent le modèle B_{Appl} extrait de l'application par analyse statique du code source de l'application "convertisseur Euros/Dollars"' (étude de cas). Il s'agit d'un modèle de dialogue de l'application. Ce modèle est exploitable pour vérifier certaines propriétés de sûreté du système.

```
MODEL BAppl
/* Modèle extrait de l'application "convertisseur Euros/Dollars" */
SEES BSwing /* Utilise la modélisation B_Swing */

VARIABLES

V_kp, run, enabled, visible, JText
/* Les définitions de 'enabled', 'visible' et 'JText' sont ré-utilisable pour chaque nouvelle extraction */

INVARIANTS

V_kp ∈ 0,1 ∧
run ∈ BOOL ∧
JText ∈ JTextfield → TEXT ∧
visible ∈ WIDGETS → BOOL ∧
enabled ∈ WIDGETS → BOOL

INITIALISATION

V_kp:=1 || run:=false || JText:={input ↦ empty, output ↦ empty} ||
visible:={pc  ↦  false,gauche ↦ false,boutonsPanel ↦ false,droit↦false,inputPanel ↦ false,
          input ↦ false,ED ↦ false,DE ↦ false, outputPanel ↦ false, output ↦ false}||
enabled:={pc ↦ false,gauche ↦ false,boutonsPanel ↦ false,droit ↦ false,inputPanel ↦ false,
           input ↦ false,ED ↦ false,DE ↦ false,outputPanel ↦ false,output ↦ false}

EVENTS

/* Definitions des evenements du modèle B_Appl */

open = /* ouverture de l'application */
SELECT ∀w.(w ∈ WIDGETS ∧ w.visible=false ∧ w.enabled=false) ∧ run=false
THEN
visible:={pc ↦ true, gauche ↦ true,boutonsPanel ↦ true,droit ↦ true, inputPanel ↦ true,
           input ↦ true,ED ↦ true,DE ↦ true,outputPanel ↦ true,output ↦ true} ||
enabled:={pc ↦ true,gauche ↦ false,boutonsPanel ↦ false,droit ↦ false,inputPanel ↦ false,
           input ↦ true,ED ↦ false,DE ↦ false,outputPanel ↦ false,output ↦ false} ||
JText:={input ↦ empty, output ↦ empty} ||
run:=true || V_kp:=1
END ;

close = /* fermeture de l'application */
SELECT run=true ∧ visible(pc)=true ∧ enabled(pc)=true ∧ V_kp≠0
THEN
visible:={pc ↦ false,gauche ↦ false,boutonsPanel ↦ false,droit ↦ false,inputPanel ↦ false,
           input ↦ false,ED ↦ false,DE ↦ false,outputPanel ↦ false,output ↦ false} ||
enabled:={pc ↦ false,gauche ↦ false,boutonsPanel ↦ false,droit ↦ false,inputPanel ↦ false,
           input ↦ false,ED ↦ false,DE ↦ false,outputPanel ↦ false,output ↦ false} ||
JText:={input ↦ empty, output ↦ empty} ||
run:=false || V_kp:=1
END ;
```

FIG. B.3 – Modèle B_{Appl} (1)

```
/* Suite du modèle BAppl (2) */
- EVENTS

actionP_DE = /* Clic sur le bouton de conversion "Dollar -> Euros" */
SELECT run=true ∧ visible(DE)=true ∧ enabled(DE)=true ∧
∀w.((w ∈ WIDGETS ∧ w ∈ WIDGETS_parents(DE)) ⇒ visible(w)=true)
THEN
enabled:=enabled ⩤ {DE ↦ false, ED ↦ true}
END ;

actionP_ED = /* Clic sur le bouton de conversion "Euros -> Dollars" */
SELECT run=true ∧ visible(ED)=true ∧ enabled(ED)=true ∧
∀w.((w ∈ WIDGETS ∧ w ∈ WIDGETS_parents(ED)) ⇒ visible(w)=true)
THEN
enabled:=enabled ⩤ {ED ↦ false, DE ↦ true}
END ;

keyPressed_0 = /* Entrée d'une valeur dans le champ de texte input : indetrministe */
ANY tt WHERE tt ∈ TEXT ∧ run=true ∧ visible(input)=true ∧ enabled(input)=true ∧
∀w.((w ∈ WIDGETS ∧ w ∈ WIDGETS_parents(input)) ⇒ visible(w)=true) ∧
V_kp=1
THEN
V_kp:=0 ∥ JText:=JText⩤{input↦tt}
END ;

keyPressed_1 = /* Reaction de l'interface à une entrée dans le champ de texte input dans
                  le cas où le champ de texte est vide */
SELECT run=true ∧ visible(input)=true ∧ enabled(input)=true ∧
∀w.((w ∈ WIDGETS ∧ w ∈ WIDGETS_parents(input)) ⇒ visible(w)=true) ∧
V_kp=0 ∧ JText(input)=empty
THEN
V_kp:=1 ∥ enabled:=enabled⩤{ED↦false,DE↦false}
END ;

keyPressed_2 =/* Reaction de l'interface à une entrée dans le champ de texte input dans
                  le cas où le champ de texte est non vide */
SELECT run=true ∧ visible(input)=true ∧ enabled(input)=true ∧
∀w.((w ∈ WIDGETS ∧ w ∈ WIDGETS_parents(input)) ⇒ visible(w)=true) ∧
V_kp=0 ∧ JText(input)=full
THEN
V_kp:=1 ∥ enabled:=enabled⩤{ED↦true,DE↦true}
END ;

END
```

FIG. B.4 – Modèle B_{Appl} (2)

Modèle $B_{ValidAppl}$

Les figures B.6, B.7 et B.8 présentent le modèle de validation $B_{ValidAppl}$. Ce modèle permet notamment de vérifier que les structures d'interaction encodées dans le programme s'inscrivent bien dans les scénarii d'usage décrits par un modèle de tâches CTT. Le modèle CTT utilisé ici est le modèle présenté en figure B.5.

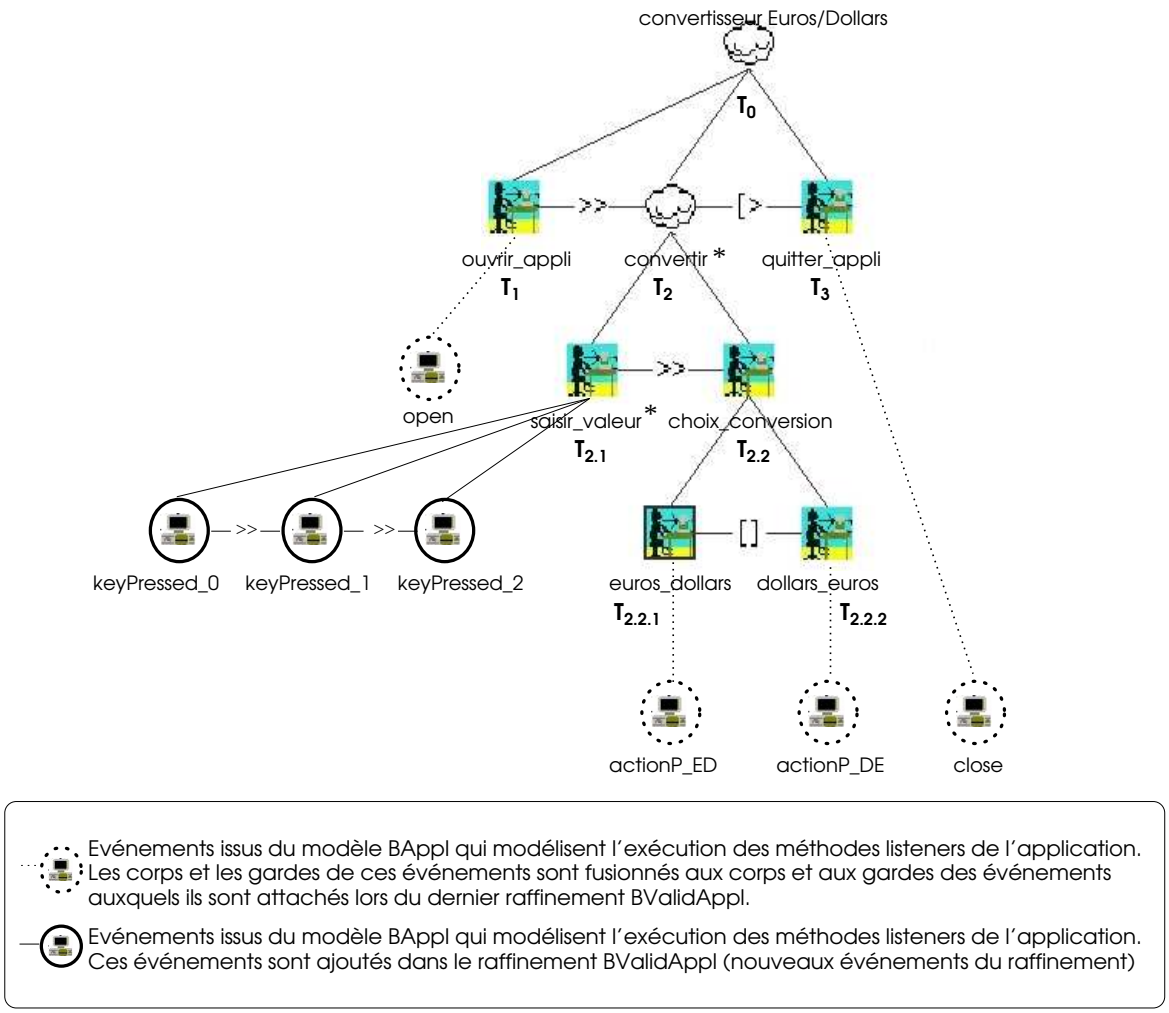


FIG. B.5 – Modèle de tâche CTT du convertisseur Euros/Dollars.

REFINEMENT *BValipAppl /* Modèle de validation de l'application vis-à-vis d'une spécification CTT */*
REFINES *BTask_ref2*
SEES *BSwing*

VARIABLES

run, V_{open}, V_{star1}, V_{star2}, V_{choix}, V_{kp}, enabled, visible, JText

VARIANT

V_{kp} */* Variant pour les événements introduits : keyPressed_0, keyPressed_1 et keyPressed_2*/*

INVARIANTS

/ Invariants de typages */*
JText $\in$ JTextfield $\rightarrow$ TEXT $\wedge$ visible $\in$ WIDGETS $\rightarrow$ BOOL $\wedge$ enabled $\in$ WIDGETS $\rightarrow$ BOOL $\wedge$
run $\in$ BOOL $\wedge$ $V_{kp}\in$ 0..2 $\wedge$ V_{choix} $\in$ 0..3 $\wedge$ V_{open} $\in$ 0..1 $\wedge$ V_{star1} $\in$ 0..100 $\wedge$ V_{star2} $\in$ 0..100 $\wedge$

/ Invariants de collage */*
((run=false) $\Rightarrow$ (visible(pc)=false)) $\wedge$ ((run=true) $\Rightarrow$ (visible(pc)=true))
$\wedge$ (V_{open}=1 $\Rightarrow$ V_{star2}>1) $\wedge$ (V_{open}=1 $\Rightarrow$ $V_{star1}\in$1..100)

/ invariant de collage : 'post-condition' de l'événement $T_{2.1}$ */*
$\wedge$ ((run=false) $\Rightarrow$ ($\forall$w.(w$\in$WIDGETS $\Rightarrow$ visible(w)=false $\wedge$ enabled(w)=false)))

/ invariant de collage : 'post-condition' de l'événement $T_{2.2}$ */*
$\wedge$ (G'$Evt\_T_{2.2}$ $\Rightarrow$ (visible(output)=true $\wedge$ Jtext(output)=full))

ASSERTIONS

/ Propriété de non-blocage */*
(...$\vee$ G'$_{Evt\_T_{2.2.1}}$ $\vee$ G'$_{Evt\_T_{2.2.2}}$ $\vee$ G'$_{Evt\_T_{2.2}}$ $\vee$...)
$\Rightarrow$
(...$\vee$ $G_{keyPressed\_0}$ $\vee$ $G_{keyPressed\_1}$ $\vee$ $G_{keyPressed\_2}$ $\vee$...)

INITIALISATION

V_{open}:=1 $\|$ V_{star1}:$\in$(V_{star1} $\in$ NATURAL $\wedge$ $V_{star1}\geq$1) $\|$ V_{star2}:$\in$(V_{star2} $\in$ NATURAL $\wedge$ $V_{star2}\geq$2) $\|$
V_{choix}:=3 $\|$ V_{kp}:=2 $\|$
JText:={input $\mapsto$ empty, output $\mapsto$ empty } $\|$
visible:={pc$\mapsto$false,gauche$\mapsto$false,boutonsPanel$\mapsto$ false,droit$\mapsto$false,inputPanel$\mapsto$false,
input$\mapsto$false,ED$\mapsto$false,DE$\mapsto$false,outputPanel$\mapsto$false,output$\mapsto$false} $\|$
enabled:={pc$\mapsto$false,gauche$\mapsto$false,boutonsPanel$\mapsto$false,droit$\mapsto$false,inputPanel$\mapsto$false,
input$\mapsto$false,ED$\mapsto$false,DE$\mapsto$false,outputPanel$\mapsto$false,output$\mapsto$false}$\|$
run:=false $\|$

FIG. B.6 – Modèle $B_{ValidAppl}$ (1)

EVENTS

ouvrir_appli (Evt_T_1)= /* Fusion avec l'événement "open" du modèle Bappl */
SELECT
V_{open}=1 ∧ visible(pc)=false ∧ enabled(pc)=false ∧ run=false ∧ V_{kp}=2 ∧ V_{choix}=3
THEN
V_{kp}:=2 ‖ V_{open}:=0 ‖ run:=true ‖ V_{choix}:=3
enabled:={pc↦true,gauche↦false,boutonsPanel↦false,droit↦ false,inputPanel↦ false,
input↦true,ED↦false,DE↦false,outputPanel↦false,output↦ false} ‖
visible:={pc↦true,gauche↦true,boutonsPanel↦true,droit↦true,inputPanel↦ true,
input↦true,ED↦true,DE↦ true, outputPanel↦ true, output↦ true} ‖
JText:={input↦empty,output↦empty} ‖
END ;

convertir (Evt_T_2)=
SELECT
run=true ∧ visible(pc)=true ∧ V_{open}=0 ∧ V_{star1} > 1 V_{star2}=0
THEN
V_{star1}:=V_{star1}-1 ‖
V_{star2}:∈(V_{star2} ∈ NATURAL ∧ V_{star2}≥2)
END ;

quitter_appli (Evt_T_3)=
/ Fusion des événements T_3 du modèle B_{Task_ref2} et close du modèle B_{Appl} */*
SELECT
run=true ∧ visible(pc)=true ∧ V_{open}=0 ∧ V_{star1}∈1..100
THEN
JText:={input ↦ empty, output ↦ empty} ‖
visible:={pc↦false,gauche↦false,boutonsPanel↦false,droit↦ false,inputPanel↦false,
input↦false,ED↦false,DE↦false,outputPanel↦false,output↦false} ‖
enabled:={pc↦false,gauche↦false,boutonsPanel↦false,droit↦false,inputPanel↦ false,
input↦false,ED↦false,DE↦false,outputPanel↦false,output↦false} ‖
V_{open}:=0 ‖ V_{star1}:=0 ‖ run:=false
END ;

saisir_valeur ($Evt\_T_{2.1}$)=
SELECT
run=true ∧ visible(pc)=true ∧ V_{open}=0 ∧ V_{star1} > 1 ∧ V_{star2} > 1 ∧ V_{kp}=0
THEN
V_{star2}:=V_{star2}-1 ‖ V_{kp}:=2
END ;

choix_conversion ($Evt\_T_{2.2}$) =
SELECT
run=true ∧ visible(pc)=true ∧ V_{open}=0 ∧ V_{star1} > 1 ∧ V_{star2} = 1 ∧ V_{choix}=0
THEN
V_{star2}:=0 ‖ V_{choix}:=3
END ;

initChoice = */* Evénement permettant de représenter l'opérateur choix "[]" de CTT */*
ANY var1 WHERE
run=true ∧ visible(pc)=true ∧ V_{open}=0 ∧ V_{star1} > 1 ∧ V_{star2} = 1 ∧ V_{choix}=3 ∧ var1∈1,2
∧ enabled(ED)=true ∧ enabled(DE)=true ∧ visible(ED)=true ∧ visible(DE)=true
∧ ∀w.(w ∈ WIDGETS_parents(ED) ⇒ visible(w)=true) ∧ ∀w.(w ∈ WIDGETS_parents(DE) ⇒ visible(w)=true)
THEN
V_{choix}:=var1
END

FIG. B.7 – Modèle $B_{ValidAppl}$ (2)

$euros\_dollars$ ($Evt\_T_{2.2.1}$) =
/* Fusion des événements $Evt\_T_{2.2.1}$ du modèle B_{Task_ref2} et $actionP\_ED$ du modèle B_{Appl} */
SELECT
run=true ∧ visible(pc)=true ∧ V_{open}=0 ∧ V_{star1} > 1 ∧ V_{star2} = 1 ∧ V_{choix}=1
∧ enabled(ED)=true ∧ visible(ED)=true ∧ ∀w.(w ∈ WIDGETS_parents(ED) ⇒ visible(w)=true)
THEN
V_{choix}:=0 ‖ enabled:=enabled ⩤ {ED↦false,DE↦true}
END ;

$dollars\_euros$ ($Evt\_T_{2.2.2}$) =
/* Fusion des événements $Evt\_T_{2.2.2}$ du modèle B_{Task_ref2} et $actionP\_DE$ du modèle B_{Appl} */
SELECT
run=true ∧ visible(pc)=true ∧ V_{open}=0 ∧ V_{star1} > 1 ∧ V_{star2} = 1 ∧ V_{choix}=2
∧ enabled(DE)=true ∧ visible(DE)=true ∀w.(w ∈ WIDGETS_parents(DE) ⇒ visible(w)=true)
THEN
V_{choix}:=0 ‖ enabled:=enabled ⩤ {ED↦true,DE↦false}
END ;

$keyPressed\_0$ =
textit/* Nouvel événement issu du modele B_{Appl} : raffine l'événement "saisir_valeur" */
ANY tt WHERE
run=true ∧ visible(pc)=true ∧ V_{open}=0
∧ V_{star1} > 1 ∧ V_{star2} > 1
∧ V_{kp}>1 ∧ V_{kp}=2
∧ enabled(input)=true ∧ visible(input)=true
∧ ∀w.(w ∈ WIDGETS_parents(input) ⇒ visible(w)=true)
∧ tt ∈ TEXT
THEN
V_{kp}:=1 ‖
JText(input):=tt
END ;

$keyPressed\_1$ =
/* Nouvel événement issu du modèle B_{Appl} : raffine l'événement "saisir_valeur" */
SELECT
run=true ∧ visible(pc)=true ∧ V_{open}=0 ∧ V_{star1} > 1
∧ V_{star2} > 1
∧ V_{kp}=1 ∧ V_{kp}>0
∧ enabled(input)=true ∧ visible(input)=true
∧ ∀w.(w ∈ WIDGETS_parents(input) ⇒ visible(w)=true)
∧ JText(input)=full
THEN
V_{kp}:=0
enabled:=enabled⩤{DE ↦ true ,ED ↦ true}
END ;

$keyPressed\_2$ =
/* Nouvel événement issu du modèle B_{Appl} : raffine l'événement "saisir_valeur" */
SELECT
run=false ∧ visible(pc)=true ∧ V_{open}=0 ∧ V_{star1} > 1
∧ V_{star2} > 1
∧ V_{kp}=1 ∧ V_{kp}>0
∧ enabled(input)=true ∧ visible(input)=true
∧ ∀w.(w ∈ WIDGETS_parents(input) ⇒ visible(w)=true)
∧ JText(input)=empty
THEN
V_{kp}:=0 ‖
enabled:=enabled⩤{DE ↦ false ,ED ↦ false}
END

END

FIG. B.8 – Modèle $B_{ValidAppl}$ (3)

Chapitre C

Modèles NuSMV

Cette annexe présente le modèle Nu_{Appl} extrait de l'application par analyse du code source de l'application "convertisseur Euros/Dollars" (étude de cas). Ce modèle est un modèle de dialogue de l'application interactive Java analysée. Ce modèle peut être construit avec un surcoût négligeable puisqu'il est possible de le dériver de manière "ad hoc" à partir du modèle B_{Appl}.

Le modèle Nu_{Appl} est constitué d'un ensemble de modules NuSMV. La figure C.2 présente les modules modélisant la bibliothèque de composants graphiques Swing du langage Java.

La figure C.3 présente le module appelé *ihm*. Ce module utilise les modèles précédents (bibliothèque Swing) et définit les instances de widgets qui composent l'application interactive. Il dispose de deux paramètres d'entrée : `ua` et `preua`. `ua` caractérise l'action de l'utilisateur sur l'interface et `preua` la valeur de `ua` à l'instant précédent. Les systèmes de transitions associées aux variables du modèle sont définis en fonction de l'action utilisateur `ua`. Il est possible de voir l'ensemble des valeurs `ua` comme les étiquettes associées aux transitions des variables modélisant l'application, c.a.d les attributs pertinents des widgets de l'application.

Enfin, les figures C.3, C.4 et C.5 présentent le module principal. Ce dernier module définit le système de transition de la variable `ua`. La prochaine action utilisateur est choisie de manière indéterministe parmi l'ensemble des actions réalisables à l'instant courant. Une action est réalisable si le widget utilisé pour l'action considérée est actif et visible. Le modèle Nu_{Appl} est exploitable en vue de vérifier certaines propriétés de sûreté, de vivacité et d'équité du système (B événementiel permet uniquement de vérifier des propriétés de sûreté).

Le modèle $Nu_{ValidAppl}$ n'est pas présenté dans cette annexe. La seule différence entre les modèles Nu_{Appl} et $Nu_{ValidAppl}$ est la définition du système de transitions associé à la variable `ua`.

```
MODULE widget
VAR
visible: boolean;
enabled : boolean;
```

```
MODULE jtextfield
VAR
visible: boolean;
enabled: boolean;
text : boolean;
```

```
MODULE jFrame
VAR
visible: boolean;
enabled : boolean;
```

FIG. C.1 – Modélisation de la boîte à outils Swing en NuSMV

```
MODULE ihm(ua,pre_ua)

VAR

pc : jFrame; ED : widget; DE : widget; input : jtextfield; output : jtextfield;

ASSIGN

init(ED.visible) := 0;
init(DE.visible) := 0;
init(input.visible) := 0;
init(output.visible) := 0;
init(ED.enabled) := 0;
init(DE.enabled) := 0;
init(input.enabled) :=0;
init(input.text) :=0;
init(output.enabled) :=0;
init(output.text) :=0;
init(pc.visible) :=0;
init(pc.enabled) :=0;

next(ED.visible) := case
ua=open : 1;
ua=close : 0;
1 : ED.visible;
esac;

next(DE.visible) := case
ua=open : 1;
ua=close :0;
1 : DE.visible ;
esac;

next(input.visible) :=case
ua=open : 1;
ua=close :0;
1 : input.visible;
esac;

next(output.visible) := case
ua=open : 1;
ua=close :0;
1 :output.visible ;
esac;

next(pc.visible) := case
ua=open : 1;
ua=close :0;
1 : pc.visible ;
esac;

next(input.enabled) :=case
ua=open :1 ;
ua=close :0;
1 :input.enabled ;
esac;

next(ED.enabled) := case
ua=open : 0;
ua=kp0 : 0;
ua=kp1 : 0;
ua=kp2 :1 ;
ua=ap_ED :0 ;
ua=ap_DE :1 ;
ua=close :0;
1 : ED.enabled;
esac;

next(DE.enabled) := case
ua=open : 0;
ua=kp0 : 0;
ua=kp1 : 0;
ua=kp2 :1 ;
ua=ap_ED :1 ;
ua=ap_DE :0 ;
ua=close :0;
1 : DE.enabled;
esac;

next(output.text) :=case
ua=open : 0;
ua=ap_ED : 1;
ua=ap_DE : 1;
ua=close :0;
ua=kp0 : 0;
ua=kp1 : 0;
ua=kp2 : 0;
1 :output.text ;
esac;

next(output.enabled) :=case
ua=open : 0;
1 : output.enabled;
esac;

next(pc.enabled) :=case
ua=open : 1;
ua=close : 0;
1 : pc.enabled;
esac;

next(input.text) :=case
ua=open : 0;
ua=kp0 : 0,1;
ua=close : 0;
1 : input.text;
esac;
```

FIG. C.2 – Modèle Nu_{Appl} (1)

```
MODULE NuAppl

VAR

ua : open, kp1, kp2, ap_ED, ap_DE, close, kp0,nil,LM_action;
preua : open,kp1, kp2, ap_ED, ap_DE, close, kp0,nil, LM_action;
ok : boolean;
UI : ihm(ua,preua);
Vkp : boolean;
Gopen : boolean; Gkp1 : boolean; Gkp2 : boolean; Gap_ED : boolean; Gap_DE : boolean;
Gclose : boolean; Gkp0 : boolean; Gnil : boolean; GLM_action : boolean;

ASSIGN

/* Les gardes sont ici simplifiées : elles ne tiennent pas compte de la hiérarchie de widgets.
cf. conditions d'activation */
Gopen:= (UI.pc.enabled=0) ∧ (UI.pc.visible=0) ∧ (UI.DE.enabled=0 ∧ UI.DE.visible=0)
           ∧ (UI.ED.enabled=0 ∧ UI.ED.visible=0)
           ∧ (UI.input.enabled=0 ∧ UI.input.visible=0)
           ∧ (UI.output.enabled=0 ∧ UI.output.visible=0)
           ∧ Vkp=1;
Gkp1:= (UI.input.enabled=1 ∧ UI.input.visible=1) ∧ UI.input.text=0 ∧ Vkp=0;
Gkp2:= (UI.input.enabled=1 ∧ UI.input.visible=1) ∧ UI.input.text=1 ∧ Vkp=0;
Gap_ED:= (UI.ED.enabled=1 ∧ UI.ED.visible=1)∧ Vkp=1;
Gap_DE:= (UI.DE.enabled=1 ∧ UI.DE.visible=1)∧ Vkp=1;
Gclose:= (UI.pc.enabled=1) ∧ (UI.pc.enabled=1) ∧ Vkp=1;
Gkp0:= (UI.input.enabled=1 ∧ UI.input.visible=1) ∧ Vkp=1;
Gnil:= 1;
GLM_action:= 1;

/* INITIALISATION des variables */

init(ua):=open;
init(preua):=nil;
init(ok):=1;
init(Vkp):=1;

/* DEFINITION des SYSTEMES de TRANSITIONS */

/* Définition du système de transitions associé à la variable Vkp. Cette variable est une variable
de contrôle assurant le bon enchaînement des transitions kp0,kp1 et kp2 */
next(Vkp):=case
(( (ua=kp1)∨(ua=kp2) ) ∧ Vkp=0) : 1;
(ua=kp0) : 0;
1 : Vkp;
esac;

/* Définition du système de transitions associé à la variable preua */
next(preua):= ua;

/* Définition du système de transitions associé à la variable ok. La variable ok permet de s'assurer
que les gardes des transitions réalisées sont valides (gardes des transitions respectées) */
next(ok):= case
ok=0 : 0;
ua=open ∧ preua=nil : (Gopen);
ua=LM_action ∧ preua=open : (Gkp0 ∧ Gclose);
ua=LM_action ∧ preua=kp0 ∧ UI.input.text=1: (Gkp2);
ua=LM_action ∧ preua=kp0 ∧ UI.input.text=0: (Gkp1);
ua=LM_action ∧ preua=kp1 : (Gkp0 ∧ Gclose);
ua=LM_action ∧ preua=kp2 : (Gap_ED∧ Gap_DE ∧ Gclose);
ua=LM_action ∧ preua=ap_ED : (Gkp0 ∧ Gclose);
ua=LM_action ∧ preua=ap_DE : (Gkp0 ∧ Gclose);
ua=LM_action ∧ preua=close : (Gopen);
1 : ok;
esac;
```

FIG. C.3 – Modèle Nu_{Appl} (2)

```
(Suite du modèle NuAppl) (3)

/* Définition du système de transitions associé à la variable ua. Une transition est réalisable
si sa garde est vraie. Le choix de la prochaine transition effectuée est indéterministe (choix
parmi l'ensemble des transitions réalisables).
Le seul moyen d'assurer cette condition est d'énumérer l'ensemble des cas possibles. */
next(ua):= case
(ua=nil ∨ ua=open ∨ ua=close ∨ ua=kp0 ∨ ua=kp1 ∨ua=kp2 ∨ ua=ap_ED ∨ ua=ap_DE) : LM_action;
(preua=close) ∧ Gopen : open ;
(ua=LM_action) ∧ Gopen ∧ ¬Gclose : open;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ Gap_DE :close,kp0,kp1,kp2,ap_ED,ap_DE;

(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : close,kp0,kp1,kp2,ap_ED;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : close,kp0,kp1,kp2,ap_DE;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ Gap_DE : close,kp0,kp1,ap_ED,ap_DE;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ Gap_DE : close,kp0,kp2,ap_ED,ap_DE;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ Gap_DE : close,kp1,kp2,ap_ED,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ Gap_DE : kp0,kp1,kp2,ap_ED,ap_DE;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : close,kp0,kp1,kp2;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : close,kp0,kp1,ap_ED;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : close,kp0,kp2,ap_ED;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : close,kp1,kp2,ap_ED;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : kp0,kp1,kp2,ap_ED;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : close,kp0,kp1,ap_DE;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : close,kp0,kp2,ap_DE;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : close,kp1,kp2,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : kp0,kp1,kp2,ap_DE;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ Gap_DE : close,kp0,ap_ED,ap_DE;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ Gap_DE : close,kp1,ap_ED,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ Gap_DE : kp0,kp1,ap_ED,ap_DE;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ Gap_DE : close,kp2,ap_ED,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ Gap_DE : kp0,kp2,ap_ED,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ Gap_DE : kp1,kp2,ap_ED,ap_DE;

(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : close,kp0,kp1;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : close,kp0,kp2;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : close,kp1,kp2;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : kp0,kp1,kp2;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : close,kp0,ap_ED;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : close,kp1,ap_ED;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : kp0,kp1,ap_ED;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : close,kp2,ap_ED;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : kp0,kp2,ap_ED;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : kp1,kp2,ap_ED;
(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : close,kp0,ap_DE;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : close,kp1,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : kp0,kp1,ap_DE;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : close,kp2,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : kp0,kp2,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : kp1,kp2,ap_DE;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ Gap_DE : close,ap_ED,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ Gap_DE : kp0,ap_ED,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ Gap_DE : kp1, ap_ED,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ Gap_DE : kp2, ap_ED,ap_DE;
```

FIG. C.4 – Modèle Nu_{Appl} (3)

```
(Suite du modèle NuAppl) (4)
- Fin de définition du système de transition associé à la variable ua -


(ua=LM_action) ∧ Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : close,kp0;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : close,kp1;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : close,kp2;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : close,ap_ED;
(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : close,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : kp0,kp1;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : kp0,kp2;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : kp0,ap_ED;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : kp0,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : kp1,kp2;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : kp1,ap_ED;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : kp1,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : kp2,ap_ED;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ Gap_DE : kp2,ap_DE;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ Gap_DE : ap_ED,ap_DE;

(ua=LM_action) ∧ Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : close;
(ua=LM_action) ∧ ¬Gclose ∧ Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : kp0;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ Gkp1 ∧ ¬Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : kp1;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ Gkp2 ∧ ¬Gap_ED ∧ ¬Gap_DE : kp2;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ ¬Gap_DE : ap_ED;
(ua=LM_action) ∧ ¬Gclose ∧ ¬Gkp0 ∧ ¬Gkp1 ∧ ¬Gkp2 ∧ Gap_ED ∧ Gap_DE : ap_DE;
1:nil;
esac;

SPEC AG( ok=0 ⇒ ((UI.input.enabled=1 ∧ UI.input.visible=1)
       ∧ (UI.ED.enabled=1 ∧ UI.DE.visible=1) ∧ (UI.output.visible=1)))
SPEC AG(ok=0 ∧ ¬(Vkp=0) ∧ UI.input.text=1 ⇒ (UI.ED.enabled=1 ∧ UI.DE.visible=1))
SPEC AG( (ok=0 ∧ Vkp=1 ∧ UI.input.text=1) ⇒ (UI.ED.enabled=0 ∧ UI.DE.enabled=0))
SPEC AG(ok=0 ∧ ((UI.ED.enabled=1 ∧ UI.DE.enabled=0) ∨ (UI.ED.enabled=0 ∧ UI.DE.enabled=1))
        ⇒ UI.output.text=1)
SPEC AG(preua=kp2 ⇒ (EG(ua=LM_action ∨ ua=ap_DE ∨ ua=ap_ED)))
SPEC AG(preua=kp2 ⇒ EG(Gap_ED=1 ∨ Gap_DE=1))
SPEC AG((preua=ap_DE ∨ preua=ap_ED) ⇒ (AF(UI.output.text=1)))
SPEC AG((preua=open) ⇒ AF(preua=close))
SPEC AG(EF(Gap_DE=1))
SPEC AF AG(Gkp0=1)
```

FIG. C.5 – Modèle Nu_{Appl} (4)

Contribution à la validation formelle d'applications interactives Java

Les travaux présentés dans ce manuscrit proposent une approche formelle pour la validation d'applications interactives Java-Swing vis-à-vis d'une spécification décrite par un modèle de tâches CTT. L'objectif de cette approche est de valider une partie de l'utilisabilité du système en s'appuyant sur l'extraction d'un modèle formel décrivant le comportement dynamique de l'application (modèle de dialogue). Cette extraction est obtenue par analyse statique du code source Java-Swing de l'application. La validation du système consiste alors à démontrer formellement que les structures d'interaction encodées dans le programme s'inscrivent bien dans les scénarii d'usage représentés en compréhension par le modèle de tâches CTT. Cette étape de validation exploite d'une part le modèle formel extrait par analyse statique et d'autre part une formalisation du modèle de tâches. La démarche d'extraction et de validation est abordée suivant deux techniques formelles distinctes : la méthode B événementielle basée sur la démonstration de théorèmes (theorem-proving), et la méthode NuSMV basée sur la vérification exhaustive de modèles (model-checking). Une étude de cas permet d'illustrer tout au long du mémoire la démarche de validation proposée suivant ces deux techniques formelles.

Mots clefs : validation, Vérification, Méthodes Formelles, Interaction Homme-Machine, IHM, Utilisabilité, Méthode B événementiel, NuSMV, Theorem Proving, Model-Checking, Analyse Statique, Abstraction, Modèle de dialogue, Java, Swing, Java-Swing, CTT, Modèle de Tâches

Contribution to the formal validation of Java interactive systems

User Interface (UI) systems are increasingly complex and nowadays assist critical activities. The development of UIs needs empowered validation methodologies in order to ensure the correctness of the developed UI-based applications. This thesis investigates the applicability of reverse engineering, static analysis and formal approaches to the validation and the verification of UIs correctness. The approach is the following. User interface's abstract models (NuSMV and B Event models) are derived starting from its Java/Swing source code. These formal execution models (dialog models) are then used to prove that the developed interactive system is in accordance with usability requirements expressed in CTT tasks models. A case study illustrates the proposed validation and verification process following these two formal techniques (Model-Checking with NuSMV and Theorem-Proving with Event B).

Keywords : validation, Verification, user interface, human computer interaction, HCI, usability, formal methods, Event B method, NuSMV, Theorem Proving, Model-Checking, Static Analysis, abstraction, dialog model, task model, CTT, Java, Swing, Java-Swing