Acronymes

ARINC	Aeronautical Radio Incorporated
CDS	Control and Display System
DAL	Development Assurance Level
DU	Display Unit
EASA	Agence Européenne de la sécurité Aérienne
FAA	Federal Aviation Agency
FCU	Flight Control Unit
MCDU	Multipurpose Control and Display Unit
MFD	Multi Function Display
PetShop	Petri nets workShop
UA	User Application

Table des matières

Table des figures

Table des tableaux

Introduction

Dans tous les domaines, à la fois techniques et grand public, l'interaction homme système a beaucoup évoluée ces dernières années. Les écrans tactiles et autres tablettes tactiles sont des exemples de cette révolution. Dans cette mouvance, le cockpit d'un avion est passé d'un environnement principalement analogique à un environnement de plus en plus numérique. On rencontre cette évolution notamment au sein du système d'affichage et de contrôle (CDS) des cockpits, qui est passé d'un simple afficheur de données, à un système interactif offrant la possibilité à l'équipage de vol d'interagir avec les systèmes avions sur les écrans via l'utilisation du clavier et d'un dispositif de pointage. Cette évolution a aussi permis une meilleure représentation et intégration des informations.

La capacité d'interagir avec les écrans au travers du clavier et d'un dispositif de pointage n'a été introduite dans le cockpit qu'avec l'adoption du standard ARINC 661. Au sein d'Airbus, c'est la famille d'avion A380 qui a été la première à l'utiliser. Néanmoins, cette nouvelle capacité d'interaction n'est pas aussi simple à intégrer. Le système d'affichage et de contrôle est un système critique, son développement doit répondre à des exigences de sûreté de fonctionnement fixées par les autorités de certification. Ces exigences impliquent notamment d'assurer un niveau qualitatif dans le développement du logiciel et d'intégrer des mécanismes de tolérance aux fautes pour traiter d'éventuelles erreurs lors de l'exécution du système. L'application de ces exigences n'étant pas encore faite à ce jour, l'interaction sur les écrans via l'utilisation du clavier et d'un dispositif de pointage est limitée au sein du cockpit à des fonctions dites non critiques. On retrouve ainsi sur les écrans des zones accessibles par le curseur et d'autres ou l'utilisation du curseur est interdite.

Par ailleurs l'interaction homme machine qui est une discipline qui s'intéresse à la conception, l'évaluation et l'implémentation de système interactif a parmi ses concepts fondamentaux de mettre l'utilisateur au centre du processus de conception. Ceci implique de produire un système qui n'est pas complexe à utiliser et à apprendre pour les utilisateurs supposés, et de produire un système à la fois sans danger et efficace pour les autres. Le système d'affichage et de contrôle du cockpit est un système interactif, dont les principaux utilisateurs sont les pilotes. Sa conception doit donc aussi intégrer les principes liés à l'interaction homme machine. L'introduction d'une nouvelle capacité d'interaction au travers des écrans conduit à un dialogue plus élevé entre le système et l'utilisateur, il devient donc important d'avoir une analyse dès la conception du système interactif de l'impact que peut avoir l'introduction de mécanismes de sûreté sur l'utilisation du système. En effet un système sûr peut être inutile ou dangereux si aucun utilisateur ne peut s'en servir, ou s'il est trop complexe. Bien que pendant longtemps, les études sur la sûreté de fonctionnement du système étaient menées de façon indépendante à son utilisabilité, on commence à voir apparaitre dans les documents de réglementation des systèmes d'affichage et de contrôle des recommandations demandant de prendre en compte la charge de travail de l'équipage dans le développement du système.

Etendre la capacité d'interaction sur les écrans via le clavier et le dispositif de pointage aux fonctions critiques demandent donc de prendre en compte deux aspects, la sûreté de fonctionnement et une conception adapté à l'utilisateur. C'est dans ce contexte que s'intègre nos travaux.

Nous proposons différentes approches pour contribuer au développement d'un système interactif intégrant les aspects sûreté de fonctionnement et utilisabilité.

Pour assurer la qualité du logiciel, nous proposons d'avoir une conception allant vers du zéro défaut. Ceci en ayant une description précise et non ambigüe du système à travers l'utilisation de formalismes.

Pour traiter les fautes résiduelles de conception, les fautes matérielles ou de l'environnement, nous proposons trois options d'intégration de mécanismes de tolérance aux fautes : la première est d'avoir une tolérance aux fautes reposant sur l'utilisateur, la seconde est d'avoir une tolérance aux fautes reposant sur le système et la troisième est une approche mixte de la tolérance aux fautes reposant à la fois sur l'utilisateur et sur le système.

Enfin pour intégrer des notions d'utilisabilité, nous proposons d'expliciter l'impact des différentes approches de conception zéro défaut et de tolérance aux fautes sur l'utilisation du système. Ceci sera fait au travers de la réalisation de modèle de tâche pour analyser l'activité de l'utilisateur.

Nous présentons nos approches dans ce mémoire au travers de 5 chapitres. Le chapitre I présente la problématique du sujet de thèse. Il présente l'évolution des cockpits et du CDS en particulier, donne une description du Standard ARINC 661 et présente les enjeux en termes de sûreté et d'utilisabilité dans le développement de système interactif.

Le chapitre II dresse un état de l'art sur les domaines scientifiques pertinents pour nos travaux. On y présente les architectures des systèmes interactifs et leurs techniques de description allant de la description de l'utilisateur jusqu'au cœur de calcul du système. Les principes de bases de la sureté de fonctionnement sont ensuite présentés avec une focalisation sur la tolérance aux fautes.

Le chapitre III décrit de façon détaillée les différentes approches que nous proposons pour le développement d'un système interactif intégrant les aspects de sûreté de fonctionnement et d'utilisabilité. Nous présentons tout d'abord le contexte applicatif de nos travaux, avant de décrire successivement l'approche zéro défaut, l'approche tolérance aux fautes et les modèles de tâche.

Le Chapitre IV montre l'application de l'approche zéro défaut au travers d'une étude de cas basée sur une application avionique.

Le chapitre V est dédié à l'application de l'approche tolérance aux fautes. Nous décrivons ici les 3 options de l'approche de tolérance aux fautes que sont la tolérance aux fautes reposant sur l'utilisateur, la tolérance aux fautes reposant sur le système et la tolérance aux fautes mixte. Les modèles de tâche sont réalisés pour chaque application et le chapitre se conclut avec une synthèse des différentes approches.

Nous concluons enfin ce mémoire en présentant les apports de nos travaux et les différentes perspectives.

Chapitre I. Problématique

Résumé du chapitre

Afin de délimiter notre cadre de travail, nous présentons dans ce chapitre les différents points constituant la problématique de notre étude.

Celle-ci s'inscrit dans le développement d'un système interactif critique de cockpit d'avion. L'évolution des systèmes de visualisation des cockpits, le standard ARINC 661, les normes de sûreté pour le développement des systèmes avioniques critiques, ainsi que les enjeux en termes d'utilisabilité y sont présentés.

I.1 DES SYSTEMES REACTIFS AUX SYSTEMES INTERACTIFS DANS LES COCKPITS

Les cockpits d'avion ont beaucoup évolué ces dernières années (Figure I.1). Dans les années 70 ils étaient principalement équipés d'instruments électromécaniques, on pouvait compter une centaine de cadrans mélangeant chiffres, aiguilles et symboles. Le besoin d'une meilleure présentation et intégration des informations nécessaires au pilotage et à la navigation a conduit à la planche de bord avec écrans. Ces évolutions permettent notamment la réduction de l'équipage de vol sur les avions commerciaux passant ainsi de trois personnes à deux.

Cockpit A300 (années 70)

Cockpit A320 (années 80)

Cockpit A380 (années 2000)

Cockpit A350 (Prévu en 2013)

Figure I.1 Evolution des cockpits d'avion Airbus

Au niveau technologique, les premiers écrans sont de type cathodique comme sur les premiers cockpits de l'A320. Ils seront ensuite remplacés par des écrans à cristaux liquides (LCD).

Pendant longtemps la planche de bord reste assez simple dans son utilisation, les écrans sont de simples systèmes de visualisation ne permettant aucune entrée d'information par l'équipage. Toutes les

actions de commandes et/ou contrôle sont réalisés sur les boutons physiques situés de part et d'autre du cockpit, notamment sur le panneau plafond.

Ce n'est qu'avec l'introduction de l'ARINC 661 et dès le programme A380 chez Airbus, que l'on voit apparaître des planches de bord ressemblant plus à des PC avec des notions de fenêtrage et l'utilisation du clavier et d'un dispositif de pointage pour interagir avec les écrans. L'équipage a maintenant la capacité de réaliser des contrôles sur les systèmes avions directement sur les écrans.

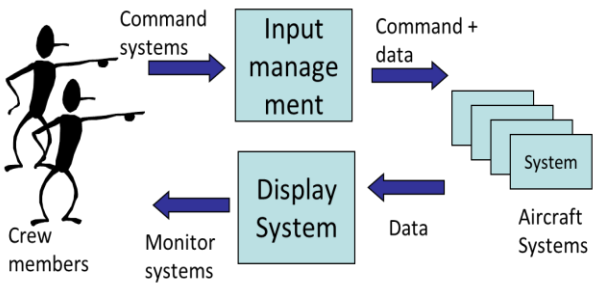
Un exemple assez parlant est présenté à la Figure I.2, où à gauche le MCDU (Multipurpose Control and Display Unit) qui possède plusieurs fonctions pour la gestion du vol, a une zone d'affichage et des boutons poussoirs physiques tout autour. Il est aujourd'hui remplacé par la MFD (Multi Function Display) qui a une interface entièrement logicielle avec une capacité d'interaction avec le clavier et un dispositif de pointage.

MCDU sur A320

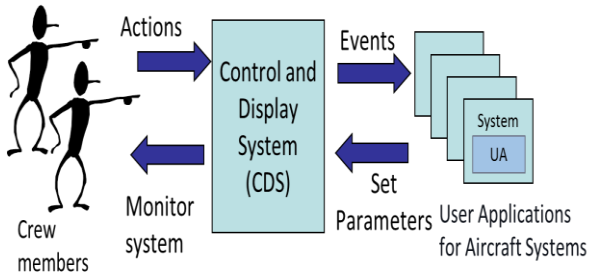
MFD sur A380

Figure I.2 MCDU/FMS

L'architecture du système, ainsi que la communication avec les systèmes avions ont aussi évolué. La Figure I.3 présente cette évolution.



System communication on A320



System communication on A380

Figure I.3 Communication entre les systèmes avions et l'utilisateur

Sur les Airbus A320, la gestion des entrées (commande sur les boutons poussoirs physiques) est entièrement ségréguée de la gestion des sorties (affichage des informations sur les écrans). A partir de l'A380, il y a une fusion de la gestion des entrées et des sorties, on parle maintenant de système d'affichage et de contrôle encore appelé CDS (Control and Display System), par rapport à l'ancienne architecture ce

sont les systèmes avions qui sont maintenant responsable de l'animation de leur page affichée, ils intègrent notamment une nouvelle application logicielle appelée UA (User Applications) qui a en charge l'animation de leurs pages.

Cette capacité d'interagir avec une interface entièrement logicielle et au travers des périphériques d'entrées tels que le clavier et un dispositif de pointage est encore aujourd'hui limitée au sein du cockpit à des fonctions dites non critiques. On retrouve ainsi une interface avec des zones accessibles par le curseur et des zones où l'utilisation du curseur est interdite. Cette limitation est due au fait que le système d'affichage et de contrôle est un système critique et il doit respecter des exigences de sûreté de fonctionnement et des règles de certification strictes (ARP4754, 1996) (DO-178B, 1992).

Pourtant les besoins d'évolution existent, de nouvelles fonctions sont ajoutées au sein du cockpit, l'augmentation du trafic aérien nécessite une représentation de l'information encore plus intégrée pour une meilleure gestion par les pilotes. Au niveau des équipes de recherche au sein d'Airbus, des propositions sont depuis faites pour avoir des interactions tactiles et des représentations 3D donc pour avoir un niveau d'interaction logicielle encore plus large.

Des réflexions sont menées pour réduire l'interface de contrôle physique et avoir plus de contrôle logiciel via l'écran. En effet l'utilisation du logiciel apporte plusieurs avantages par rapport au matériel que sont : la flexibilité de conception, par la facilité qu'on a avec le logiciel de redéfinir la position des objets sur une interface, de changer des couleurs etc. La configuration et la reconfiguration de l'information qu'il est difficile d'avoir sur les interfaces physiques, ceux-ci nécessitant un câblage assez lourd. Mais ces évolutions sont limitées dans leur intégration à cause des contraintes de sûreté.

C'est dans ce contexte que s'inscrit ce travail de thèse dont l'objectif est de proposer des approches outillées pour avoir une interaction logicielle critique au sein des cockpits. Nous présentons dans les sections suivantes le standard ARINC 661 qui a cette nouvelle capacité d'interaction dans le cockpit, les normes de développement des systèmes avioniques critiques et les enjeux en termes d'utilisabilité.

I.2 LE STANDARD ARINC 661

L'ARINC 661 est un standard qui a été adopté par l'AEEC (Airlines Electronic Engineering commitee) en 2001 (ARINC661, 2010). Son objectif est de définir l'interface entre le système d'affichage et de contrôle (CDS) du cockpit et les autres applications des systèmes de l'avion. Il est rédigé par les principaux acteurs de l'aéronautique : Airbus, Boeing, Thales, Rockwell Collins, Honeywell, etc.…

Les objectifs principaux du standard ARINC 661 sont :

- La minimisation des coûts d'ajout ou de modification d'une nouvelle fonction d'affichage pendant la durée de vie d'un avion (qui est d'une trentaine d'années pour les avions commerciaux) ;
- L'introduction de l'interactivité dans le cockpit, en mettant en place une base pour standardiser l'interface homme machine dans le cockpit ;
- de rendre les dispositifs plus faciles à utiliser, plus efficaces (en permettant l'affichage d'informations complexes dans les systèmes du bord).
- Minimiser le coût de gestion de l'obsolescence des composants matériel dans un environnement ou les évolutions technologiques sont rapides.

Le CDS fournit des services graphiques et interactifs aux applications utilisateurs (UA) au sein du cockpit. Les applications interactives sont exécutées sur des unités d'affichage appelées Display Unit (DU) et les interactions avec les pilotes se font par le biais de l'utilisation de périphériques physiques ou graphiques tel le Keyboard Cursor Control Unit (KCCU) qui est un ensemble clavier et dispositif de pointage (Figure I.4).

Figure I.4 KCCU (Keyboard Cursor Control Unit)

L'ARINC 661 est basé sur une architecture Client-serveur avec une ségrégation entre la partie fonctionnelle gérée par les applications avions appelées UA et la partie graphique gérée par le CDS (tel qu'on peut le voir sur la partie droite de la Figure I.3).

L'ARINC 661 utilise un concept de fenêtrage qui est comparable à celui des PC mais avec des restrictions dues à l'environnement de l'avion (Figure I.5).

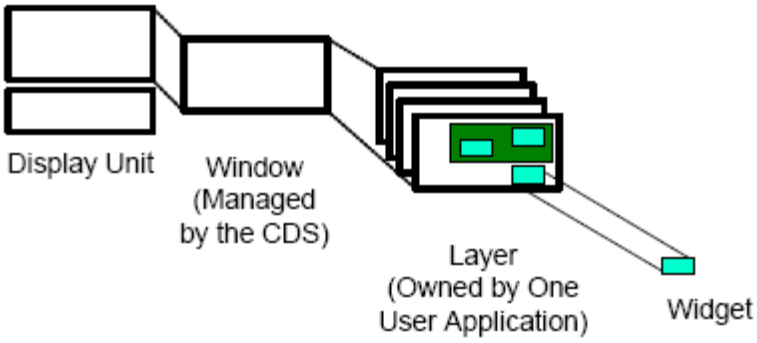


Figure I.5 Hiérarchie des niveaux de visualisation (ARINC661, 2010)

- La base physique est l'écran encore appelé *Display Unit*
- L'écran est composé d'un ensemble de fenêtres (*windows*).
- Les fenêtres sont subdivisées en couches (*layer*).
- Les *layers* sont connectées aux UA et fournies un espace d'affichage des widgets
- Les widgets[1] sont des objets pouvant avoir une représentation graphique sur lesquels les pilotes peuvent interagir. (exemple boutons, champ de texte …)

Les UA spécifient dans un fichier appelé UADF (User Application Definition File) leur interface, en décrivant les caractéristiques des layers et des widgets ainsi que leur structure hiérarchique.

Le protocole de communication entre le CDS et l'UA est définit en deux phases :

- Une phase de définition où l'UA crée son fichier UADF décrivant son interface graphique. Le fichier UADF est ensuite chargé dans le CDS afin d'être instancié et les paramètres à l'initialisation sont activés.
- Une phase d'exécution qui consiste en un transfert dynamique de données entre l'UA et le CDS. L'UA peut mettre à jour les paramètres des widgets au travers des « *setparameters* », et le CDS notifie à l'UA des actions de l'utilisateur sur son interface au travers des « Event ».

L'ARINC 661 ne définit pas le « look and feel » des widgets, c'est-à-dire le rendu graphique des widgets (« look ») et le comportement des widgets (« feel »). Le document du standard définit une bibliothèque de widgets, leurs caractéristiques et le protocole de communication entre le CDS et les UAs. Les utilisateurs du standard doivent ensuite spécifier le comportement de leur widgets et de l'interface vis-à-vis des caractéristiques listées dans le standard.

Le standard ARINC 661 comprenait dans sa première version sortie en novembre 2001 près 42 widgets. Il a beaucoup évolué et compte aujourd'hui à son supplément 4 plus de 60 widgets.

[1] Les widgets sont plus que des objets graphiques et correspondent mieux à la notion d'interacteur. Dans le modèle d'York (Duke, et al., 1993) un interacteur est défini comme un *composant dans la description d'un système interactif qui encapsule un état, les évènements qui manipulent cet état, et les moyens par lesquels cet état est rendu perceptible par l'utilisateur*.

I.3 DEVELOPPEMENT DES SYSTEMES AVIONIQUES CRITIQUES

Avant de pouvoir voler, chaque avion doit obtenir son certificat de navigabilité qui garantit la conformité de l'appareil aux normes de sécurité internationales. Ces normes de sécurité sont définies par les autorités de certification qui se chargent aussi de vérifier leur application. Les deux principales agences de certification sont l'EASA (Agence Européenne de la sécurité Aérienne) et la FAA (Federal Aviation Agency) aux Etats-Unis. L'EASA et la FAA définissent des exigences concernant la conception de l'avion, leur exploitation commerciale ou les licences de pilotage.

Parmi les normes établies au travers de groupes de travail on distingue :

- L'ARP 4754 (Guidelines for Development of Civil Aircraft and Systems) (ARP4754, 1996) fournit des recommandations pour le développement des systèmes avions en prenant en considération l'ensemble de l'environnement opérationnel de l'avion et ses fonctions. Ceci inclut la validation des exigences et la vérification de l'implémentation pour la certification et l'assurance du produit.
- L'ARP 4761 (Guidelines and Methods for conducting the safety assessment process on civil airbone systems and equipment) (ARP4761) propose des méthodes possibles pour évaluer la sûreté de fonctionnement des systèmes d'un avion en vue de sa certification.

De ces normes découlent d'autres recommandations telles que la DO-178B (DO-178B, 1992) qui propose des recommandations pour le développement des logiciels, la DO-254 (DO-254, 2000) pour la partie matérielle, l'AMC 25-11 pour les systèmes d'affichage électronique tel que le CDS, et des CRI (*Certification Review Item*) spécifiques pour la partie applicative du CDS réalisée en langage formel, par exemple le CRI F15 sur le CDS A350 (CRIF15, 2010).

I.3.1 La DO-178B

Dans l'avionique, la certification des systèmes embarqués met principalement l'accent, d'un point de vue de la sûreté de fonctionnement, sur la sécurité-innocuité (safety). La démarche générale repose sur un processus d'analyse de la sécurité-innocuité (intégrant des études de risques préliminaires, fonctionnelles et architecturales) qui consiste à analyser les fonctions et l'architecture des systèmes avioniques et à identifier leurs modes de défaillance. Ceux-ci sont classés en fonction d'une échelle de sévérité (vis-à-vis des conséquences de ces défaillances sur la sécurité des vols et des passagers) qui comprend cinq niveaux: catastrophique, dangereux, majeur, mineur et sans effet. La criticité d'un système est alors déterminée par la plus forte sévérité de ses modes de défaillance. On définit alors cinq niveaux d'assurance de développement (Development Assurance Level/ DAL) notés DAL-A, DAL-B, DAL-C, DAL-D et DAL-E par ordre décroissant. Ainsi, les logiciels critiques sont développés au niveau DAL-A alors que les logiciels qui n'ont aucun impact sur la sécurité de l'appareil sont développés au niveau DAL-E. Cette classification détermine ainsi le niveau d'assurance que le système doit satisfaire. Pour chacun de ces niveaux, un ensemble de critères et de preuves est requis.

Le Tableau I.1 résume le lien entre les différents niveaux de DAL présents dans la DO-178B et les objectifs en termes de probabilité et de sévérité de défaillance.

Classification des FC	Niveau de DAL	Objectif quantitatif
Catastrophique	A + *Fail Safe*	<10-9
Dangereux	B	<10-7
Majeur	C	<10-5
Mineur	D	<10-3
Sans Effet	E	aucun

Tableau I.1 Les niveaux de criticité, DAL et objectif quantitatifs

La DO-178B est consacrée au processus de développement des logiciels embarqués. Elle a pour objectif de s'assurer que le logiciel exécute sa fonction avec un niveau de sécurité suffisant. Elle est basée sur une approche orientée processus. On distingue trois types de processus dans le cycle de vie du logiciel :

- le processus de développement, qui comprend la spécification, la conception, le codage et l'intégration.
- les processus intégraux, qui comprennent la vérification, la gestion de configuration, l'assurance qualité et la coordination pour la certification ;
- le processus de planification qui coordonne le processus de développement et les processus intégraux.

Pour chaque processus et chaque niveau d'assurance, les objectifs sont définis et une description des données du cycle de vie permettant de démontrer que les objectifs sont satisfaits est également fournie. La norme n'impose aucune méthode pour atteindre les objectifs fixés. C'est à la charge du postulant à la certification (donc les constructeurs avioniques) d'apporter les preuves que les méthodes employées permettent de répondre aux objectifs. De même, bien que la norme ne l'impose pas, elle recommande d'utiliser des stratégies de tolérance aux fautes qui peuvent être employées pour la détection et le recouvrement d'erreurs.

Pour couvrir des défaillances catastrophiques, il faut atteindre l'objectif de probabilité d'erreur inférieure à 10^{-9}, mais aussi respecter le critère « *Fail Safe* » qui veut qu'aucune panne simple ne doit conduire à une défaillance catastrophique. Pour atteindre ses objectifs, les systèmes avioniques se basent habituellement sur la redondance pour intégrer des systèmes tolérants aux fautes.

Certains CRI (*Certification Review Item)* adressent une demande de démonstration de la complétude et de la pertinence de la validation et de la vérification des exigences écrites en langage formel et utilisé pour le développement du système matériel/logiciel sous les normes telle que la DO178B. Ces CRI couvrent toutes les étapes de développement du système allant de la spécification à l'implémentation.

L'application des critères de validation et vérification dans le cas d'une spécification formelle est utilisée comme point d'entrée pour la clarification des besoins de développement du logiciel.

Les définitions suivantes sont proposées dans le contexte d'un CRI et sont directement extrait de l'ARP4754.

- Validation : la détermination que les exigences pour un produit sont suffisamment correctes et complètes.
- Vérification : l'évaluation de l'intégration des exigences pour déterminer qu'elles ont été atteintes

Les premiers objectifs liés à la validation des spécifications formelles pour être en conformité avec l'ARP4754 sont :

- Des vérifications de complétude et d'exactitude seront effectuées en utilisant des méthodes telles que les revues, des analyses, des simulations, et des tests. Des simulations peuvent être utilisées afin de détecter des problèmes le plus tôt possible. Des dispositions doivent être prises pour s'assurer que les simulations sont suffisamment représentatives du produit.
- La traçabilité des spécifications de haut niveau
- Le process et les résultats doivent être documentés.

Alors que la DO178B est appliquée par le fournisseur du logiciel, l'avionneur en l'occurrence Airbus doit appliquer les CRI.

I.3.2 L'AMC 25-11

L'AMC 25-11 est un document qui contient des moyens acceptables de conformité pour l'approbation de navigabilité des systèmes d'affichage électronique tel que le CDS. L'AMC donne des recommandations sur la présentation de l'information, l'utilisation des couleurs, la gestion de l'information, les moyens d'interaction avec l'équipage et les exigences de sûreté sur le système. Au niveau de la sûreté du système, différents points sont proposés en vue de démontrer une conformité à la réglementation.

Le premier point est l'identification des conditions de défaillances du système d'affichage. Ces défaillances concernent la perte des fonctions du système et l'affichage et/ou le contrôle erroné de l'information.

Le deuxième point est la description des effets de ces défaillances sur le pilotage de l'avion. La description des effets permet de classer le niveau de criticité de la fonction.

Le troisième point est la proposition des moyens de mitigations face aux défaillances définies. Il s'agit ici de décrire les différents mécanismes de sûreté qui permettront de traiter ou réduire les effets des défaillances.

Le CDS étant utilisé principalement par les pilotes, l'AMC demande depuis peu d'établir et documenter certains aspects de l'activité humaine lors du développement d'un système d'affichage électronique. Il s'agit de la charge de travail de l'équipage, du temps de formation de l'équipage pour se familiariser avec le système et des potentielles erreurs de l'équipage.

I.4 L'UTILISABILITE

Avec l'introduction du standard ARINC 661, le système d'affichage et de contrôle des cockpits devient ainsi un système interactif. Un système interactif est avant tout conçu pour permettre à des utilisateurs d'atteindre un but défini dans un contexte d'utilisation spécifié. C'est pourquoi l'Interaction Homme Machine qui est une discipline qui s'intéresse à la conception, l'évaluation et l'implémentation de système interactif (ACM SIGHI, 1992) a parmi ses concepts fondamentaux « de mettre l'utilisateur au centre du processus de conception ».

L'objectif principal d'une conception centrée sur l'utilisateur est de produire un système qui n'est pas complexe à utiliser et à apprendre pour les utilisateurs supposés, et de produire un système à la fois sans danger et efficace pour les autres (Preece, et al., 1994). Ceci se traduit encore par la notion d'utilisabilité qui est définit par la norme ISO 9241-11 (ISO9241-11, 1998) comme le degré selon lequel un produit peut être utilisé, par des utilisateurs identifiés, pour atteindre des buts définis avec efficacité, efficience et satisfaction, dans un contexte d'utilisation spécifié.

- **L'efficacité** est la capacité d'arriver à ses buts. Être efficace, c'est produire les résultats escomptés et réaliser les objectifs fixés dans les domaines de la qualité. Est-ce que le pilote a pu entrer son plan de vol ?
- **L'efficience** est un terme utilisé en économie, qui désigne le fait de réaliser un objectif avec le minimum de moyens engagés possibles. Est-ce que le pilote a entré son plan de vol dans un temps raisonnable et sans faire d'erreur?
- **La satisfaction** est le nom donné à l'état d'âme et/ou du corps qui accompagne l'assouvissement d'un objectif. La satisfaction découle directement du degré d'efficacité et d'efficience atteint.

Il existe différentes méthodes pour évaluer l'utilisabilité d'un système interactif, réaliser des tests avec les utilisateurs afin de s'assurer que le système est adapté à leur besoins, l'évaluation heuristique, ou des méthodes basées sur les modèles.

L'étude du profil humain, de son activité, de l'environnement du système interactif sont des points importants à évaluation de l'utilisabilité du système. Des études au niveau facteur humain contribuent notamment à cela.

Réaliser des tests utilisateurs permet notamment de détecter assez tôt les potentielles erreurs humaines et de modifier le système pour y remédier.

Dans le cadre de notre étude, nous ne nous intéressons pas aux erreurs humaines, mais soulignons le fait qu'il est important de prendre en compte des notions d'utilisabilité dans le développement d'un système interactif critique. En effet il est inutile de faire un système fiable si son utilisateur ne peut s'en servir correctement.

I.5 CONCLUSION

Nous avons abordé dans ce chapitre les différents points constituant la problématique de notre étude. Celle-ci s'inscrit dans le développement d'un système interactif critique de cockpit d'avion. Nous avons ainsi présenté l'évolution des cockpits d'avion civil et plus particulièrement le système d'affichage et de contrôle. Celui-ci est passé d'un simple système d'affichage à un système interactif, qui offre la capacité à l'équipage d'interagir avec les systèmes avions sur une interface logicielle au travers d'un clavier et d'un dispositif de pointage. Comme pour tout développement des systèmes avioniques critiques, le système d'affichage et de contrôle des cockpits doit respecter certaines exigences de sûreté provenant des autorités de réglementation. C'est aussi un système qui fonctionne principalement avec des utilisateurs, son développement doit donc aussi intégrer des notions d'utilisabilité, tels que l'efficacité, l'efficience et la satisfaction.

Le développement d'un système interactif critique nécessite donc de relever certains défis, telles que la prise en compte des aspects de sûreté de fonctionnement provenant des autorités de certification, et la prise en compte des aspects d'utilisabilité afin que le système soit adapter à ses utilisateurs. Ces défis doivent être abordés de façon synergique et non indépendante. La problématique de cette thèse est donc d'essayer de proposer une étude conjointe de ces aspects afin de permettre la conception et la certification d'un système interactif de cockpit toujours sûr tout en étant utilisable.

Chapitre II. Etat de l'Art

Résumé du chapitre

L'objectif de cette partie est de dresser un état de l'art des domaines scientifiques dans lesquels s'inscrit cette thèse.

La thèse se situe dans un cadre pluridisciplinaire. Elle touche à la fois le domaine des systèmes interactifs, à leurs techniques de description et la sûreté de fonctionnement. Les travaux de recherche menés dans ces domaines scientifiques étant très vaste, nous ne dresserons pas un état de l'art intégral, mais présenterons que ce qui est pertinent pour notre travail.

La première section du chapitre décrit un système interactif, et présente ensuite deux modèles d'architecture des systèmes interactifs que sont Seeheim et Arch.

La seconde section du chapitre présente les différentes techniques de description des systèmes interactifs. Les techniques de description de l'opérateur, des éléments d'entrées et de sorties et de la partie applicative seront présentés successivement.

La troisième section du chapitre présente plus en détail une technique de description des systèmes interactifs basés sur les réseaux de Petri à Objet et appelé ICO (Objet Coopératif Interactif). L'environnement d'édition des ICO est ensuite présenté, ainsi que leur représentation en composant qui permet d'avoir une vision macroscopique des éléments du système interactif. Parmi les techniques de description vues dans la seconde section, ICO est celui qui permet une description de l'ensemble du système interactif et dont les travaux ont porté sur les systèmes critiques. La compréhension d'ICO est importante pour nos travaux, c'est pourquoi nous lui avons dédié une section.

La quatrième section du chapitre présente les concepts de base de la sûreté de fonctionnement. Elle se concentre ensuite sur les techniques de tolérance aux fautes, très utilisés sur les systèmes critiques tels que les commandes de vol avioniques.

II.1 LES SYSTEMES INTERACTIFS

Nous donnerons la définition d'un système interactif ici en comparaison d'un système réactif.

Dans (Berry, 2000), un système réactif est défini comme un système qui réagit constamment aux sollicitations de son environnement en produisant des actions sur celui-ci. Dans ces systèmes, c'est l'environnement qui est maître de l'interaction, le rôle de l'ordinateur est de réagir de façon continue aux stimuli externes en produisant des réponses instantanées. On retrouve généralement ces systèmes dans le contrôle de processus industriel, les systèmes embarqués …

Un système interactif (Figure II.1) est avant tout un système réactif avec en plus un utilisateur humain dans la boucle. Le système doit ici prendre en compte les entrées de l'utilisateur qui peuvent arriver à n'importe quel moment, et fournir à cet utilisateur les informations nécessaires à la réalisation de ces tâches. Le dialogue entre le cœur de calcul du système et l'utilisateur se fait au travers d'une interface qui est un ensemble des moyens d'entrée et de sortie (exemple écran, clavier, dispositif de pointage, joystick, trackball, etc.)

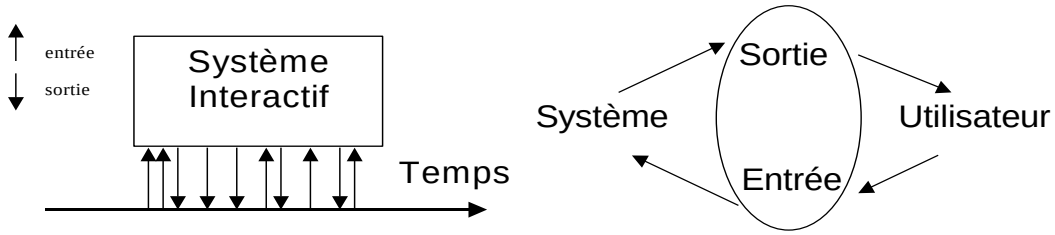


Figure II.1 présentation d'un système interactif

Le développement d'un système interactif est complexe. Il nécessite de prendre en compte à la fois l'utilisation des techniques de conception centrées sur l'utilisateur et des techniques liées à l'ingénierie logicielle et matérielle (Wasserman, 1981). Trouver une cohérence entre ces deux domaines n'est pas évident, une étude de Brad Myers (Myers, et al., 1992) a montré que 48% du code des applications était dédié à l'interface utilisateur et environ 50% du temps d'implémentation à la portion de l'interface utilisateur.

Afin de minimiser ce temps de développement important, les acteurs des systèmes interactifs ont proposé différents modèles d'architecture. L'approche commune dans la proposition de ses modèles est d'examiner le fonctionnement des systèmes interactifs, de décider ce qui sépare les fonctionnalités de l'interface des autres et de décrire une architecture supportant cette séparation.

II.1.1 Modèle d'architecture seeheim

Il existe plusieurs modèles d'architecture des systèmes interactifs telles que Seeheim (Green, 1986), ARCH (UIMS, 1992), MVC (Krasner, et al., 1988), PAC (Coutaz, 1987), PAC-Amodeus (Nigay, et al., 1993). Nous ne présentons ici que deux architectures pertinentes pour notre étude, l'architecture seeheim et son modèle d'extension ARCH. les synthèses sur les modèles d'architectures peuvent être retrouvé dans (COUTAZ, 1990) (DRAGICEVIC, 2004).

Le modèle seeheim (Green, 1986), est issu d'un groupe de travail sur les systèmes interactifs ayant eu lieu dans la ville de seeheim en 1983. C'est l'un des premiers modèles d'architecture à avoir proposé clairement une division de l'interface en trois couches logicielles (Figure II.2).

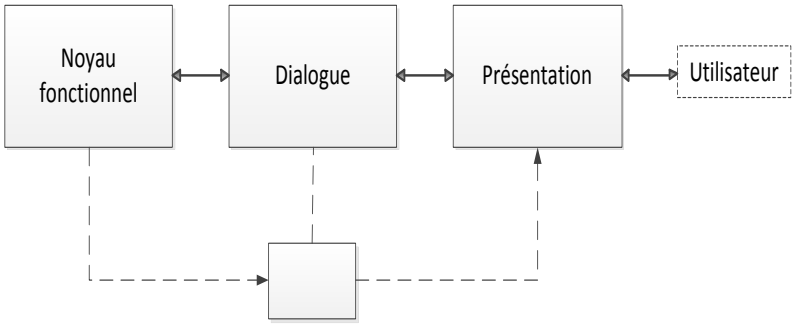


Figure II.2 Modèle d'architecture de seeheim

- La présentation est la partie responsable des entrées et sorties utilisateur. Elle interprète les actions utilisateurs sur les périphériques d'entrées (clavier, dispositif de pointage) et génère-les sorties que perçoivent les utilisateurs sur les périphériques de sortie (écran...).
- Le contrôleur de dialogue est la partie qui gère le dialogue entre l'utilisateur et l'application. Il garde un état lui permettant de gérer les enchainements des écrans et les modes d'interactions.
- L'interface au noyau fonctionnel est la partie en charge de convertir les entrées de l'utilisateur en terme d'appels au noyau fonctionnel et des données abstraites de l'application en éléments présentables à l'utilisateur.

Il existe un dernier élément visible sur la Figure II.2 qui permet de court-circuiter la partie dialogue ceci afin de permettre un feedback rapide à l'utilisateur, l'exemple courant est celui de la modification de l'affichage de l'icône cible lors d'un glisser déposer afin d'indiquer si celui-ci est valide ou pas.

II.1.2 Modèle d'architecture Arch/ Slinky

Le modèle Arch (UIMS, 1992) illustré Figure II.3 est une extension du modèle de Seeheim. L'architecture précise le contenu de certains modules et prend en compte l'existence des boîtes à outils graphiques. On distingue cinq composants dans le modèle Arch.

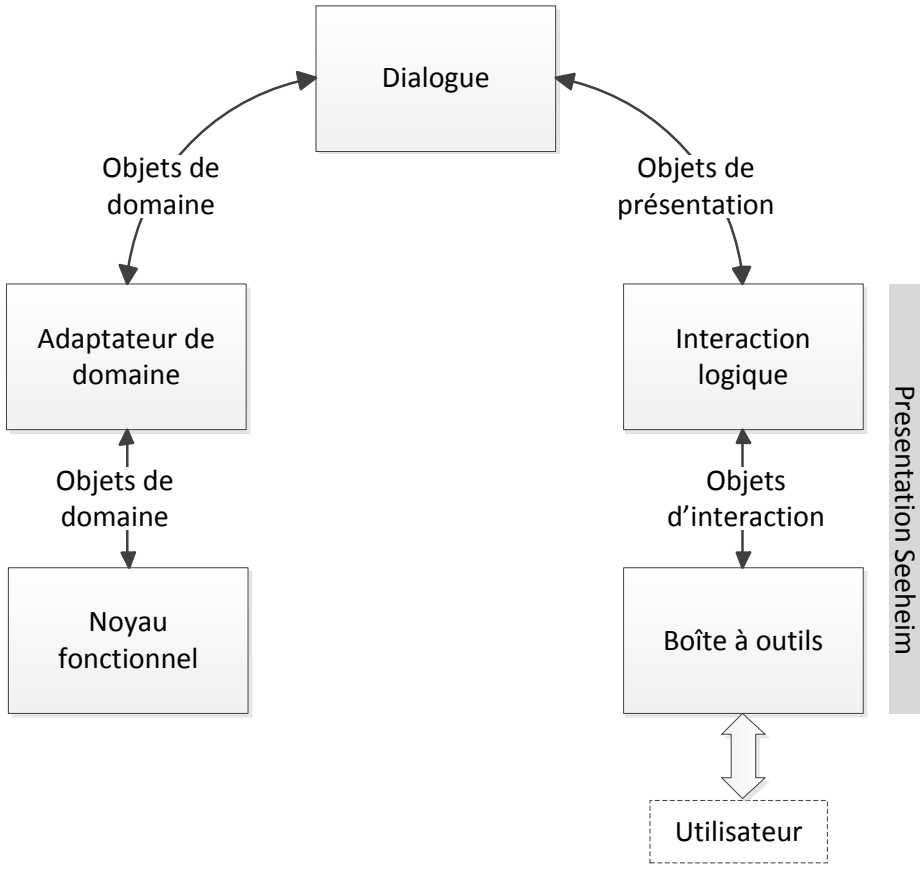


Figure II.3 modèle ARCH/Slinky

- Le composant *noyau fonctionnel* permet la manipulation de concepts à travers l'exécution de fonctions spécifiques au domaine, il ne se préoccupe pas de comment ces fonctions sont rendus perceptibles à l'utilisateur.
- Le composant *adaptateur du noyau fonctionnel* est un composant médiateur entre le contrôleur de dialogue et le noyau fonctionnel. Il est responsable des tâches dépendantes du domaine qui ne font pas partie du noyau fonctionnel mais qui sont nécessaires à sa manipulation par l'utilisateur. Ces tâches comprennent la réorganisation des données du domaine dans des buts de manipulation interactive, ou la détection des erreurs sémantiques.
- le composant de *dialogue* joue le même rôle que celui du modèle seeheim. Il assure le contrôle du séquencement des tâches. Il décrit précisément en fonction de l'état du système interactif l'ensemble des tâches autorisées ainsi que l'effet de l'accomplissement d'une tâche sur l'état du système interactif.
- le composant *interaction logique* appelé dans le modèle initial présentation joue le rôle de médiateur entre le composant boîte à outils et le dialogue. Il inclut les données à présenter à l'utilisateur et les évènements à générer à l'utilisateur. Il maintient une représentation logique des widgets qui est indépendante de la plate-forme, ceci pour résoudre les différences de modélisation entre le contrôleur du dialogue et la présentation physique du système interactif.
- Le composant *boîte à outils*, implémente la partie physique de l'interaction avec l'utilisateur. Il représente les périphériques d'entrée-sortie ainsi que l'ensemble des widgets.

Les composants boîte à outils et interaction logique forment ensemble le bloc présentation de l'architecture seeheim.

Le modèle Arch introduit également trois types d'objets décrivant la nature des informations qui transitent entre les composants (Figure II.3)

- *Les objets du domaine* contiennent des données provenant directement ou indirectement du noyau fonctionnel (par exemple, le résultat d'une requête dans une base de données).
- *Les objets de présentation* sont des objets d'interaction virtuels qui décrivent les évènements produits par l'utilisateur et les données qui lui sont présentées, indépendamment des méthodes d'interaction et de visualisation.
- *Les objets d'interaction* sont des instances propres à une boîte à outils, et qui implémentent des techniques d'interaction et de visualisation spécifiques.

Selon les types de système interactif et les choix fixés par les développeurs, certains composants du modèle Arch peuvent demander un effort plus important que d'autres. Le métamodèle Slinky a été (UIMS, 1992) a été conçu au-dessus de Arch pour mieux présenter la variante de poids dans les composants. Le métamodèle Slinky inspiré du jouet du même nom, est présenté à la Figure II.4.

Figure II.4 Le jouet Slinky ayant inspiré le modèle du même nom

Le modèle Arch a été conçu afin de protéger les systèmes interactifs des changements technologiques (modification de la boîte à outils, du matériel, du média de transmission etc.). Le terme Arch suggère notamment un environnement de développement stable lorsque les objectifs et les choix ont été définis. Par contre Arch ne répond pas forcément aux autres critères qui avaient été estimé tout aussi important pour les systèmes interactifs que sont l'amélioration des performances du système, la simplicité de conception, la réutilisation du code etc. (UIMS, 1992)

Nous noterons ici que de même que Arch, un des objectifs premiers du standard ARINC 661 était de limiter l'impact des modifications d'un système avionique sur l'ensemble du système d'affichage et de contrôle des cockpits.

II.2 LES TECHNIQUES DE DESCRIPTION DES SYSTEMES INTERACTIFS

Une variété de techniques décrivant le comportement des systèmes interactifs ont été développés. Ces techniques formelles ou non formelles permettent d'avoir une approche plus rigoureuse dans le développement des systèmes interactifs devenus de plus en plus complexes.

Pour les systèmes critiques, l'utilisation de technique de description formelle présente l'avantage de fournir une description complète, précise et non ambiguë du système. Elle permet aussi la vérification et la validation des propriétés du système lors des phases de conception et facilite ainsi son implémentation.

Nous allons présenter dans cette section quelques techniques de description des systèmes interactifs, ceci en fonction des composants du modèle d'architecture Arch et en y incluant l'utilisateur.

II.2.1 Technique de description de l'utilisateur

Le fonctionnement du système interactif dépend fortement des entrées de l'utilisateur sur des périphériques d'entrées du système et de l'interprétation et des actions qu'il entreprendra grâce aux informations fournies sur les périphériques de sorties. C'est pourquoi l'Interaction Homme Machine qui est une discipline qui s'intéresse à la conception, l'évaluation et l'implémentation de système interactif (ACM SIGHI, 1992) a parmi ces concepts fondamentaux « de mettre l'utilisateur au centre du processus de conception ».

Différentes méthodes sont ainsi proposées pour avoir un développement centré sur l'utilisateur. On distingue parmi ceux-ci la réalisation de modèles de tâche qui permet de décrire les intentions des utilisateurs et les activités dans lesquelles ces utilisateurs doivent s'engager pour accomplir leurs buts.

L'analyse des tâches consiste à relier objectifs, tâches et actions. Ces termes ont des significations différentes selon les auteurs, nous utiliserons ici les définitions utilisées dans (Preece, et al., 1994).

- L'*objectif* est l'état du système que l'utilisateur souhaite atteindre. (Exemple écrire une lettre, prendre un avion etc….). Il y a différent moyens pour atteindre l'objectif. La sélection du moyen détermine la tâche à entreprendre. Un objectif doit avoir une certaine stabilité dans le temps
- Une *tâche* est une activité nécessaire, ou utilisé pour atteindre un objectif en utilisant un moyen donné. Une tâche est en général décomposable en sous-tâches, jusqu'au niveau des actions.
- Une *action* est une tâche n'impliquant pas de résolution de problème ou de structure de contrôle.

De nombreux modèles de tâche ont été développés HTA (Annett, et al., 1967), GOMS (Baumeister, et al., 2000), MAD (Scapin, et al., 1989), GTA (Van der Veer, et al., 1996), TKS (Johnson, et al., 1989), CTT (Paternò, et al., 1999), HAMSTERS (Navarre, et al., 2010) .

II.2.1.1 *HTA*

Le plus connu et le plus ancien est le modèle HTA (Hierarchical Task Analysis) (Annett, et al., 1967). HTA est une méthode orientée tâches qui propose de décomposer hiérarchiquement une tâche en sous

tâches et en opérations. Cette décomposition est logique plutôt que psychologique/cognitive, i.e. elle n'a pas l'ambition de refléter la manière dont l'opérateur se représente mentalement sa tâche.

II.2.1.2 *GOMS*

GOMS (Goals, Operators, Methods and selection rules) (Baumeister, et al., 2000) est un modèle de description du comportement, qui prend comme hypothèse le caractère adaptatif du sujet humain. Il permet de modéliser le comportement à différents niveaux d'abstraction. Les ingrédients de GOMS sont les buts (Goal), les opérateurs (Operator), les méthodes (Method) et les sélections (Selection).

- un but définit un état recherché. Il est réalisé par l'exécution d'une suite d'opérateurs et est décomposé de manière hiérarchique ;
- un opérateur est une action atomique dans le niveau de modélisation considéré. Son exécution provoque un changement d'état du modèle mental de l'usager mais aussi de l'environnement et notamment du système. Tous les opérateurs d'un niveau de modélisation GOMS ont une durée d'exécution de même ordre de grandeur. Cette métrique est une façon de définir le niveau d'abstraction d'un modèle GOMS.
- une méthode décrit le procédé qui permet d'atteindre un but. Elle représente le savoir-faire et s'exprime par composition de sous-buts et d'opérateurs.
- une règle de sélection est utilisée lorsque plusieurs méthodes permettent d'atteindre un but donné. Elle offre le moyen de choisir une méthode parmi les candidates possibles.

GOMS s'utilise ainsi : ayant une décomposition de buts et de sous-buts, GOMS permet de prédire le temps mis par l'utilisateur (expert) pour atteindre un but donné. Cette prédiction s'obtient simplement en additionnant les temps d'exécution des opérateurs impliqués pour atteindre le but. La valeur obtenue peut être comparée, par exemple, avec celle des seuils fixés dans le plan qualité. La méthode peut également servir à guider un choix de conception : il suffit de pratiquer une évaluation comparative des performances attendues pour chaque solution envisageable.

II.2.1.3 *MAD*

MAD (Méthode Analytique de Description de tâches) (Scapin, et al., 1989), les principaux concepts sont ceux d'objet-tâche, d'action, de structure. Contrairement à HTA, et de façon analogue à GTA, la première caractéristique de MAD est de permettre de représenter la dimension hiérarchique de la planification dans l'activité humaine. En ce sens, la décomposition des éléments de la tâche en sous-tâches n'est pas une simple décomposition logique mais a pour ambition de refléter la structure des représentations mentales de l'opérateur expert de la tâche. Toutefois, ce reflet est essentiellement lié à la méthode d'analyse et de recueil des données permettant la construction du modèle de la tâche, et non pas à l'utilisation du seul formalisme. Les objectifs initiaux de MAD* sont :

« Considérer la façon dont l'utilisateur se représente sa tâche et non pas la logique du traitement informatique (si informatisation il y a) ou la tâche prescrite ; prendre en compte les aspects conceptuels et sémantiques, et pas seulement syntaxiques et lexicaux ; arriver à une décomposition structurelle des tâches de façon uniforme ; effectuer une description aussi bien du point de vue déclaratif (état des choses)

que procédural (façon d'arriver à ces états) ; autoriser la prise en compte du parallélisme autant et pas seulement du séquentiel (synchronisation des tâches) ; viser un caractère implémentable. »

II.2.1.4 *GTA*

GTA (Groupware Task Analysis) (Van der Veer, et al., 1996) est à l'origine une méthode de conception de logiciels de travail collaboratif. Contrairement aux autres modèles hiérarchiques, GTA n'est pas centré uniquement sur la tâche. En effet, un des préceptes de GTA est de considérer le monde selon 3 points de vue différents :

- les agents, i.e. les entités (individus, organisations, systèmes informatiques) qui agissent à l'intérieur du monde en le modifiant ;
- le travail (work), i.e. les tâches réalisées par les agents ;
- la situation, i.e. l'environnement dans lequel les tâches sont réalisées

II.2.1.5 *TKS*

TKS (Task Knowledge structure) (Johnson, et al., 1989) est fondée sur l'hypothèse que les gens possèdent une structure de connaissance dans leur mémoire qui se rapporte aux tâches. La méthode TKS représente cette connaissance conceptuelle pour décrire les rôles, les buts, les plans et les procédures qui consistent en des actions et des objets.

II.2.1.6 *CTT*

CTT (ConcurrentTaskTrees) (Paternò, et al., 1999) a été conçu pour décrire l'activité de l'utilisateur jusqu'au niveau de l'interaction. Il est résolument orienté vers la description de systèmes interactifs. Il permet de décrire des tâches utilisateurs en les combinant avec des opérateurs temporels. Un modèle de tâches CTT est basé sur une hiérarchie de tâches semblable à une structure d'arbre. Chaque tâche d'un même niveau est composée par un opérateur temporel qui détermine l'ordonnancement de ce niveau. CTT a une notation graphique et un outil dédié appelé CCT environment (CTTe)

II.2.1.7 *HAMSTERS*

Plus récemment HAMSTERS (Human-centered Assessment and Modeling to Support Task Engineering for Resilient Systems) (Navarre, et al., 2010) est un modèle de tâche avec un outil dédié. Il est largement inspiré des formalismes listés ci-dessus. Comme MAD et CTT le temps qualitatif est exprimé en utilisant les opérateurs temporels attachés au nœud de parent. Le temps quantitatif est représenté en exprimant la durée de tâche et le retard avant la disponibilité de la tâche.

La représentation globale du modèle de tâches Hamsters est un arbre hiérarchique. La racine de l'arbre est une tâche abstraite. Les feuilles sont les tâches d'interactions, systèmes et/ou utilisateurs indivisibles (pas de tâches abstraites). Une tâche abstraite se décompose en plusieurs sous-tâches de types différents. Le Tableau II.1 liste les tâches de Hamsters.

Pictogramme	Explication
	Les tâches abstraites sont des tâches qui généralement n'ont pas de type défini, car il s'agit de tâches composées d'autres sous-tâches de types différents. Ce type de tâche est représenté par une tache de peinture.
	Les tâches *sub-routine* représentent un groupe de tâches qui vont être défini dans un autre modèle de tâche.
	Les tâches systèmes sont des tâches effectuées par le système en interne. Ce type de tâche est représenté par un circuit imprimé.
	Les tâches interactives illustrent l'action de l'utilisateur sur le système (input), l'envoi d'information du système vers l'utilisateur (output), voir les deux (in/out) en parallèle. Ce type de tâches est représenté par un personnage face à un écran. Le sens de la flèche entre le personnage et l'écran représente le sens du flux d'information
	Les tâches utilisateurs correspondent à des actions qu'effectue l'utilisateur indépendamment du système. Il peut s'agir d'une tâche motrice, d'une tâche cognitive ou d'une tâche perceptive. Ce type de tâche est représenté par un personnage dont la main (tâche motrice), les oreilles (tâche perceptrice) ou la pensée (tâche cognitive) est mise en évidence selon le type de tâche.

Tableau II.1 Types de tâches de Hamsters

Le Tableau II.2 liste les opérateurs disponibles dans la notation Hamsters. Les opérateurs temporels définissent qualitativement les entrelacements, la séquence, ou le choix entre deux tâches. Tout comme pour CTT (Paternò, et al., 1999), les opérateurs sont issus de la notation LOTOS (ISO/IS8807, 1988).

Symbole	Nom	signification
T1 >> T2	ACTIVATION	T2 démarre après la fin de T1
T1 [> T2	DESACTIVATION	l'activation de T2 stoppe T1
T1 ||| T2	CONCURRENCE	T1 et T2 peuvent être exécutées dans n'importe quel ordre et éventuellement en parallèle
T1 [] T2	CHOIX	T1 ou T2 est exécuté
T1 |> T2	SUSPENSION /RESUME	L'activation de T2 stoppe T1, T1 redémarre lorsque T2 est terminée
T1 |=| T2	INDEPENDANCE de L'ORDRE	T1 et T2 peuvent être exécutées dans n'importe quel ordre

Tableau II.2 Opérateurs de Hamsters

II.2.2 Technique de description de l'interface utilisateur

Par rapport au modèle Arch l'interface utilisateur représente ici les données que le système présente à l'utilisateur sur les périphériques de sortie. Elle correspond à une partie de l'ensemble du bloc présentation et représente les flux allant du dialogue à l'utilisateur.

Le développement des interfaces utilisateur (UI) des applications interactives est très difficile à cause de la complexité et de la diversité des environnements de développement, et d'un nombre important de compétences demandé aux développeurs afin d'avoir une interface utilisable : langage de balisage (exemple HTML), langage de programmation (exemple C++, java), protocole de communication et compétence en utilisabilité.

Cette difficulté est accrue lorsque la même interface utilisateur doit être développée pour des contextes différents d'utilisation qui peuvent être des catégories différents d'utilisateurs (ayant des préférences différentes, des langues différentes..), des plates-formes de calcul différentes (téléphone mobile, PC portable etc.…) ou encore des environnements de travail variés (statique ou mobile...)

Pour traiter ces difficultés, un nombre de langages de description des interfaces utilisateurs dit UIDL (User Interface description Language) pour le développement des interfaces utilisateur multi-supports ont été proposés. Les efforts dans ce domaine se concentrent essentiellement autour des langages basés sur XML. UIML (Abrams, et al., 1999), AUIML (Azevedo, et al., 2000), SISL (Ball, et al., 2000), Seescoa (Luyten, et al., 2001), TADEUS XML (Müller, et al., 2001), XIML (Puerta, et al., 2002) et UsiXML (Limbourg, et al., 2004). Une étude comparative des notations basées sur XML peut être trouvée dans (Souchon, et al., 2003). Nous présentons ici UsiXML car il tend à devenir un standard (Standardisation W3C (World Wide Web Consortium) soumisse).

II.2.2.1 *UsiXML*

UsiXML (USer Interface eXtensible Markup Language) (Limbourg, et al., 2004) est un langage XML qui décrit l'interface utilisateur pour des contextes multiples (différentes techniques d'interaction, modalités d'utilisation et plates-formes d'utilisation différentes).

UsiXML permet la spécification de l'interface utilisateur indépendamment des caractéristiques physiques et la description à un haut niveau d'abstraction des éléments constitutifs de l'interface d'une application: les widgets, les contrôles, les conteneurs, les modalités, les techniques d'interaction, ...

Ci-dessous un exemple de fichier UsiXML décrivant l'affichage du fameux « hello world »

```
<?xml version="1.0" encoding="UTF-8"?>

<uiModel xmlns="http://www.usixml.org"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://www.usixml.org/ http://www.usixml.org/spec/UsiXML-
ui_model.xsd"

    id="hello_world_12" name="hello world"

    creationDate="2007-02-12T14:31:45.143+01:00" schemaVersion="1.8.0">
```

Début de tout fichier UsiXML

```
-------------------------------------------------------------------------------------
    <head>
        <version modifDate="2007-02-12T14:31:45.143+01:00">1</version>
        <authorName>Benjamin Michotte</authorName>
        <comment>Hello World example</comment>
    </head>
-------------------------------------------------------------------------------------
    <cuiModel id="hello_world-cui_12" name="hello world-cui">
        <window id="window_component_0" name="window_component_0"
content="/uiModel/resourceModel/cioRef[@cioId='window_component_0']/resource/@content"
            defaultContent="Hello world" width="470" height="255">
            <gridBagBox id="grid_bag_box_1" name="grid_bag_box_1"
                gridHeight="12" gridWidth="23"/>
        </window>
    </cuiModel>
-------------------------------------------------------------------------------------
    <contextModel id="hello_world-contextModel_12" name="hello world-contextModel">
        <context id="hello_world-context-en_US_12" name="hello world-context-en_US">
            <userStereotype id="hello_world-sten_US_12" language="en_US"
                  stereotypeName="hello world-sten_US"/>
            <platform id="hello_world-platform_12" name="hello world-platform"/>
            <environment id="hello_world-env_12" name="hello world-env"/>
        </context>
        <context id="hello_world-context-fr_FR_12" name="hello world-context-fr_FR">
            <userStereotype id="hello_world-stfr_FR_12" language="fr_FR"
                  stereotypeName="hello world-stfr_FR"/>
            <platform id="hello_world-platform_12" name="hello world-platform"/>
            <environment id="hello_world-env_12" name="hello world-env"/>
        </context>
    </contextModel>
    <resourceModel id="hello_world-res_12" name="hello world-res">
        <cioRef cioId="window_component_0">
            <resource content="Bonjour le monde" contextId="hello_world-context-
fr_FR_12"/>
            <resource content="Hello world" contextId="hello_world-context-en_US_12"/>
        </cioRef>
    </resourceModel>
</uiModel>
```

En tête du fichier, ou est écrit la version du fichier, le nom des auteurs

Description du modèle concret de hello world

CuiModel (Concrete user interface model)

Description du contexte et des ressources

Figure II.5 Exemple d'un fichier UsiXML de Hello World (hhttp://www.usixml.org)

Dans le standard ARINC 661, il existe un fichier de description de l'interface utilisateur en XML appelé UADF (User Application definition file), il comporte la même structure que le fichier de description UsiXML ci-dessus. Le fichier UADF décrit l'interface graphique d'une application avion (la position des widgets, leur taille, le texte affiché etc…)

Bien qu'une étude sur l'utilisation des réseaux de Petri afin de vérifier certaines propriétés des interfaces utilisateurs décrit en UsiXML a été réalisée dans (Romero, et al., 2006). UsiXML comporte quelques limites :

- Il ne décrit pas les détails de bas niveau des éléments impliqués dans les diverses modalités, tels que des évènements et primitives (ex : grisé un bouton)
- Il ne peut pas être rendu ni exécuté par lui-même.
- Il ne peut pas assurer tous les attributs, évènements des primitives de tous les objets interactifs existant dans presque tous les toolkits.

II.2.3 Technique de description de l'interaction en entrée

Par analogie avec la section précédente qui présente les techniques de description pour les données en sorties présentées à l'utilisateur, cette sous-section présente les techniques de description pour la gestion des entrées par l'utilisateur.

La distinction entre les entrées représentant les flux d'informations allant de l'homme à la machine et les sorties flux d'information allant de la machine vers l'homme est couramment faite en interaction homme machine, bien que les deux aspects sont étroitement liés.

L'interaction en entrée englobe l'ensemble des dispositifs physiques contrôlé par l'utilisateur, les techniques d'interaction décrivant la manière avec laquelle l'utilisateur va se servir des dispositifs d'entrée et les paradigmes d'interaction (WIMP, Post-WIMP, multimodale …)

Les micro-ordinateurs grand public utilisent des dispositifs d'entrée standard que sont le clavier et un dispositif de pointage. Il existe aussi un grand nombre de dispositifs d'entrée non standard qui sont adaptés pour certaines tâches à accomplir, pour certains utilisateurs ou encore dans les environnements de travail particulier. On retrouve l'utilisation des trackballs dans les cockpits d'avion estimé plus stable dans des moments de turbulence, des dispositifs d'entrées adaptés pour des personnes handicapés, les bornes interactives très utilisé dans les lieux publics pour un accès à des informations, achat de tickets etc.

Le nombre important et varié de dispositifs d'entrée a conduit au besoin de construire des modèles pour faire face à leur complexité de gestion. On distingue parmi ceux-ci Squidy (König, et al., 2010) et leurs ancêtres Marigold (Willans, et al., 2001) et Wizzed (Esteban, et al., 1995) et Icon (Dragicevic, et al., 2004).

Nous présenterons ici Icon dont les outils sont disponibles en téléchargement et dont les travaux de Dragicevic ont beaucoup inspirés les réflexions de cette sous-section.

II.2.3.1 *ICON*

ICON (Input Configurator) (Dragicevic, et al., 2004), est une notation liée à une boite à outils qui permet de modéliser les entrées d'applications interactives. Cette notation est capable de décrire des interactions basées sur des dispositifs non-standards. Son éditeur visuel permet au développeur de créer rapidement des configurations pour des entrées appauvries ou enrichies, que les utilisateurs avancés peuvent ensuite adapter à leurs besoins.

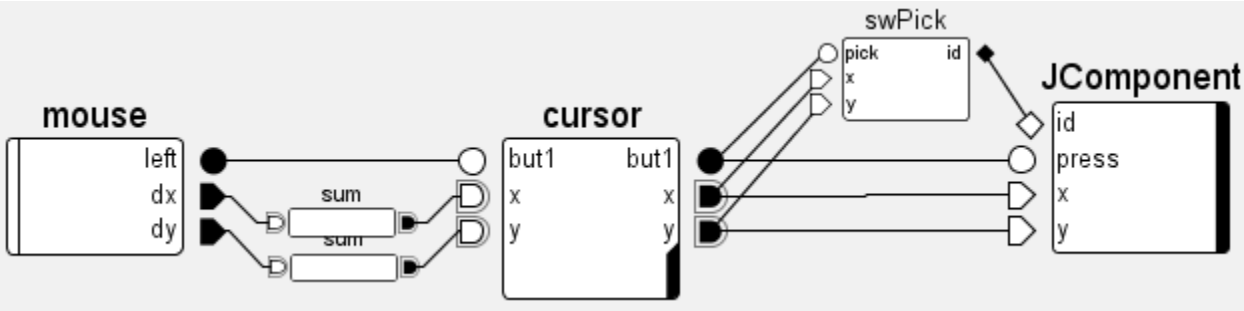


Figure II.6 Contrôle positionnel standard des composants Swing

ICON s'appuie sur une classification des dispositifs parmi lesquels on trouve :

- Les dispositifs systèmes (mouse dans la Figure II.6) qui représentent en général un dispositif d'entrée physique.
- Les dispositifs utilitaires sont les dispositifs de la bibliothèque d'ICON qui ne sont pas des dispositifs systèmes. Ces dispositifs peuvent être des dispositifs de traitement (sum dans Figure II.6) qui effectuent essentiellement des transformations de données ou des dispositifs à retour graphique (cursor et JComponent dans Figure II.6) qui produisent un affichage graphique afin d'émettre des informations vers l'utilisateur.

Grâce à son aspect visuel et sa simplicité d'utilisation (ajout de boîte et lien entre les ports de ces boîtes), ICON permet de rapidement modéliser différentes interactions ou de modifier des modèles d'interaction existants. Par contre il ne permet pas une représentation des états et ne peut être utilisé pour modéliser tous les aspects du dialogue.

II.2.4 Technique de description du dialogue et du noyau fonctionnel

Le composant dialogue de l'architecture Arch gère le dialogue entre l'utilisateur et le noyau fonctionnel. Depuis de nombreuses années la modélisation de son comportement est au cœur de nombreux travaux de recherche en interaction homme machine (IHM).

Nous présentons dans cette sous-section deux classes de techniques de description du dialogue que sont les techniques basées sur les automates à états finis et les réseaux de Petri bien que ces techniques ne pas originaires de l'IHM mais ont été importés d'autres domaines de l'informatique ou de l'automatique.

Les techniques de description présentées ci-dessous sont aussi applicables au composant noyau fonctionnel de l'architecture Arch.

II.2.4.1 *Les techniques de description basées sur les automates à états finis*

Parmi les différentes notations basées sur les automates à états finis (ou FSM : Finite state machines), nous pouvons citer les réseaux de transitions de (Kieras, et al., 1983), le 3StateModel de (Buxton, 1990), les automates à états finis réalisés dans IMBuilder (Bourget, 2002), les Hierarchical States Machine (Blanch, et al., 2006), les SwingStates (Appert, et al., 2006) et HephaisTK (Dumas, et al., 2008).

Nous présentons ici les Hierarchical States Machines. Les autres techniques ont soit sensiblement les mêmes propriétés (comme les SwingStates par rapport aux HSM), soit n'ont été utilisées que pour modéliser de très petites parties du système interactif.

II.2.4.1.1 Hierarchical States Machine

Les machines à états hiérarchiques (Blanch, et al., 2006) ou HSM (Hierarchical state machine) sont une notation basée sur les automates à états finis. Tout comme dans le cas des SwingStates (Appert, et al., 2006). La Figure II.7 montre un exemple du comportement d'un bouton modélisé en automates à états finis. .

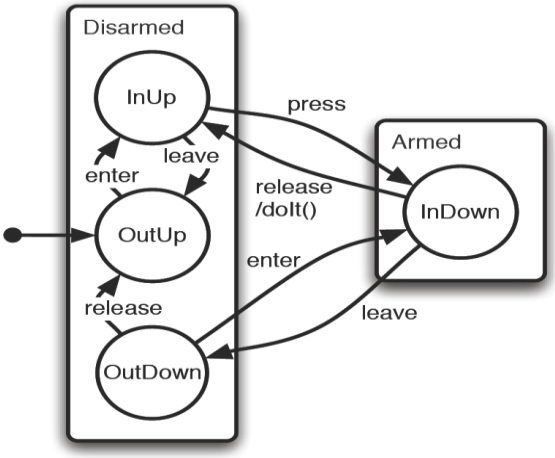


Figure II.7 modélisation du comportement d'un bouton en Hierarchical state machine

La Figure II.8 présente la version textuelle de l'automate présenté à la Figure II.7. Cette version textuelle est structurée comme un code C++.

```
01 hsm Button {
02    hsm Disarmed {
03       // variables
04       var svg::SVGElement *elem;
05       var svg::SVGGElement *armed = 0;
06       var svg::SVGGElement *disarmed = 0;
07
08       // initialization
09       init {
10          hsm::svg::Childs(elem).assert(armed)
11                                .assert(disarmed);
12          elem->removeChild(armed);
13       }
14
15       // visual aspect coherence
16       enter { elem->replaceChild(armed, disarmed); }
17       leave { elem->replaceChild(disarmed, armed); }
18
19       hsm OutUp {
20          - enter() > InUp
21       }
22
23       hsm InUp {
24          - leave() > OutUp
25          - press() > Armed::InDown
26       }
27
28       hsm OutDown {
29          - enter()   > Armed::InDown
30          - release() > OutUp
31       }
32    }
33
34    hsm Armed {
35       hsm InDown {
36          - leave() > Disarmed::OutDown
37
38          // action invocation
39          - release() { doIt(); } > Disarmed::InUp
40       }
41    }
42 }
```

Figure II.8 version textuelle du modèle en Hierarchical state machine (Blanch, et al., 2006)

Les Hierarchical state machines et les SwingStates ont comme avantages la facilité d'intégration dans une application existante et la modélisation de certains aspects d'un système.

Néanmoins, les automates à états finis posent certains problèmes. Il est par exemple impossible de modéliser les systèmes avec un grand nombre d'état car il mène rapidement à une explosion du nombre d'états et de transitions. La description par automate n'est de plus pas modulable pour la concurrence.

Dans (Appert, et al., 2009) une intégration entre les automates à états (swingstates) dans Icon est proposée. Cette intégration permet un pouvoir d'expression plus grand et une grande flexibilité pour le prototypage de l'interaction, mais ne résout pas les problèmes de taille de modèle.

II.2.4.2 *Les techniques de description basées sur les réseaux de pétri*

Les réseaux de Petri constituent une technique de description formelle de la dynamique des systèmes. Ils permettent de modéliser les états d'un système et ses changements d'états. La technique de réseaux de Petri constitue actuellement l'outil le plus avancé et le plus complet pour la spécification et la description des systèmes de processus fonctionnant en parallèle et avec des contraintes de

synchronisation. Il s'attache en particulier à décrire deux aspects de ces systèmes : des évènements (ou actions) et des conditions, ainsi que les relations entre évènements et conditions. (Fichefet, 1995)

Les réseaux de Petri sont utilisés pour la description des systèmes interactifs, on y retrouve notamment, les réseaux de Petri colorés utilisés par (Rieder, et al., 2010), les réseaux de Petri à Objet utilisé par (Bastide, et al., 1990) et les ICO (Palanque, 1992) une extension des réseaux de Petri à Objet.

II.2.4.2.1 Les Réseaux de Petri colorés

(Rieder, et al., 2010) Présente une notation basée sur les réseaux de Petri colorés pour la description d'interaction en environnement virtuel (EV). Les couleurs du réseau de Petri sont remplacées par des icônes représentant les objets et les actions de l'interaction. Cette transformation permet de plus facilement différencier les divers types de données qui sont manipulées dans l'application de réalité virtuelle.

Les modèles peuvent être simulés permettant à l'utilisateur de tester les différentes techniques d'interaction et de vérifier si le modèle fonctionne comme prévu.

Dans l'exemple présenté en Figure II.9, nous pouvons voir au niveau de la légende les différentes icônes représentant les objets de l'interaction tels que le pointeur, la position tridimensionnelle, ou l'objet sélectionné.

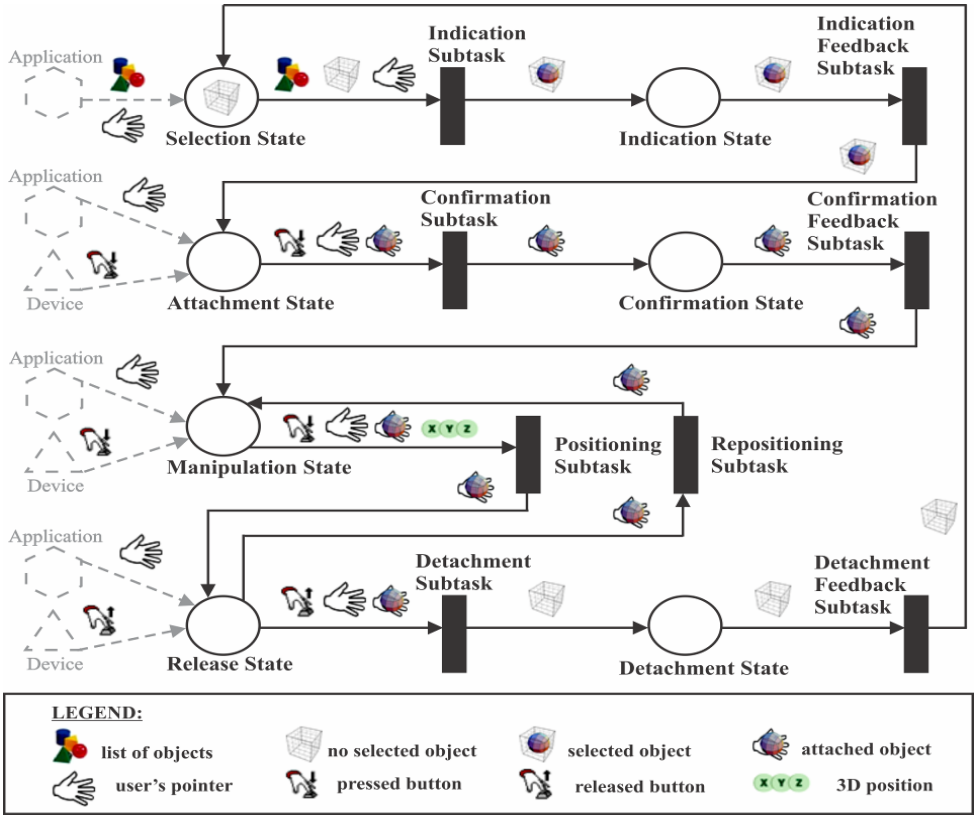


Figure II.9 Réseau de Petri coloré d'un processus d'interaction (Rieder, et al., 2010)

Ces réseaux de Petri colorés ont deux principaux avantages. L'aspect visuel des données échangées permet de faciliter la lecture et cette notation peut tirer parti du formalisme des réseaux de Petri pour pouvoir réaliser différentes analyses sur ces réseaux.

Par contre, l'outil lié à cette notation nécessite une compilation et ne permet donc pas de modifier les modèles durant l'exécution, comme par exemple ICON (Dragicevic, et al., 2004) et le formalisme ne prévoit pas la possibilité de spécifier certains aspects d'un système interactif comme son rendu.

II.2.4.2.2 Les réseaux de Petri à objet

(Bastide, 1992) Présente dans ces travaux une spécification du composant dialogue du modèle d'architecture seeheim en réseau de Petri à objet plus précisément en Objet Coopératif, dont la classe éditeur est présenté à la Figure II.10.

```
Class implémentation Editeur
Attributes
       f : Fenêtre;  -- Fenêtre d'édition des attributs du n-uplet.
Services
       Editer; -- Effectuer une opération d'édition quelconque sur
                    -- Le n-uplet sélectionné.
       Remplacer;    -- Remplacer le n-uplet courant par celui en cours
                    -- d'édition.
       Ajouter;      -- Ajouter le n-uplet en cours d'édition
                    -- et le sélectionner.
       Détruire;     -- Détruire le n-uplet en cours d'édition.
                    -- En sélectionner un nouveau s'il en reste, sinon
                    -- en créer un avec des valeurs par défaut.
       Rétablir;     -- Annuler toute édition effectuée sur le n-uplet, et
                    -- restaurer ses valeurs initiales.
       Sélectionner<clé : Identifiant>;
                    -- Sélectionner le n-uplet ayant la clé spécifiée,
                    -- et afficher la valeur de ses attributs.
       Quitter;      -- Terminer le dialogue, fermer la fenêtre.
       Create(fen : Fenêtre) is
              f := fen; Défaut := n_uplet.Create;
       end; -- Create
ObCS
       Edité :<o, dup : n_uplet>;
       Sélectionné, Liste, Défaut : <o : n_uplet>;
```

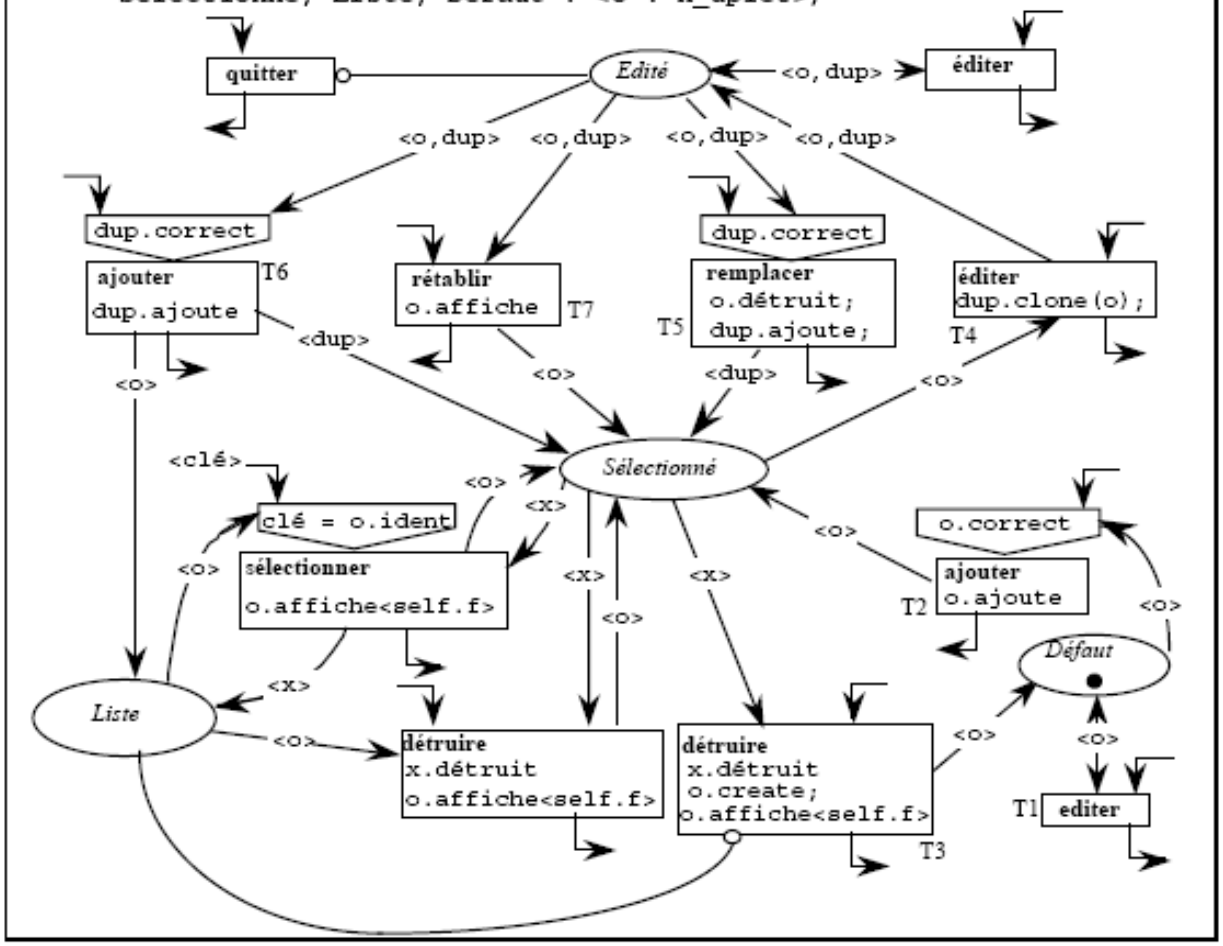


Figure II.10 Implémentation classe Editeur du composant dialogue (Bastide, 1992)

Une extension à ces travaux est proposé dans (Palanque, 1992) au travers des ICO (Objet coopératif Interactif). ICO couvre à la fois les interactions en entrée, i.e. comment les actions de l'utilisateur impacte l'état interne du système et quelles actions sont disponibles à un instant donné et les sorties i.e. quand les informations sont affichés à l'utilisateur.

ICO a été utilisé pour les applications interactives du control aérien (Navarre, et al., 2001), des cockpits d'avion civil (Barboni, et al., 2006), et pour les systèmes interactifs multimodaux dans la thèse de (Ladry, 2010). Un environnement dédié à l'édition et l'exécution des ICO appelé PetShop a également été développé (Navarre, et al., 2001).

La couverture de modélisation d'ICO (dialogue, entrée et sortie), et les travaux réalisés dessus pour la description de systèmes interactifs critiques, nous ont conduits à faire le choix de cette technique de description pour notre étude. La section suivante est consacrée à une explication plus détaillé d'ICO et de son environnement d'édition PetShop.

II.3 UN FORMALISME ET UN OUTIL POUR LA DESCRIPTION DES SYSTEMES INTERACTIFS

L'objectif de cette partie est de présenter les ICO (Objet Coopératif Interactif) et son environnement d'édition PetShop. Pour mieux comprendre le fonctionnement des ICO nous expliquerons premièrement les bases des réseaux de Petri et des réseaux de Petri haut niveau. Nous présenterons aussi la notion des CompoNet offrant une représentation abstraite des ICO et facilitant la présentation des architectures des systèmes interactifs.

II.3.1 Les réseaux de Petri et Réseaux de Petri haut niveau

Les réseaux de Petri (Petri, 1962) Figure II.11 sont une technique de description formelle de la dynamique des systèmes à évènements discrets. Ils s'attachent en particulier à décrire deux aspects de ces systèmes : des évènements (ou actions) et des conditions, ainsi que les relations entre évènements et conditions.

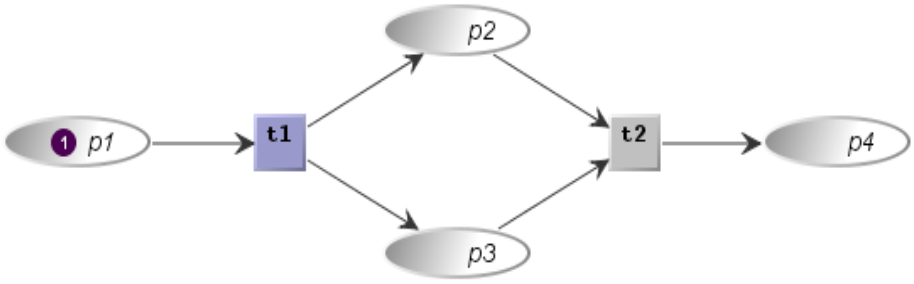


Figure II.11 Exemple d'un réseau de Petri

Michel Diaz dans (Diaz, 2001) page 21 liste comme intérêts aux réseaux de Petri :

- « Ils ont fourni toutes les premières approches de modélisation, ainsi que leurs sémantiques, utilisées pour modéliser et maitriser les comportements des systèmes parallèles et distribués, synchronisés et communicants ;
- Ils définissent un support graphique aisé pour l'expression et la compréhension des mécanismes de base de ces comportements;
- Ils constituent, par leur relation avec les machines à états, des représentations faciles à comprendre et donc à manipuler, à la fois pour la création des modèles et pour leur analyse ;
- Ils permettent d'exprimer très simplement les concepts premiers des fonctionnements communicants, en incluant les phénomènes d'attente et de synchronisation, en prenant en considération leurs caractéristiques et paramètres temporels et stochastiques ;
- Etant non liés à un langage particulier de réalisation, ils assurent l'indépendance de la modélisation et de sa représentation vis-à-vis des implantations ;

- Ils possèdent, pour les étapes du processus de conception, des méthodes de validation basées sur un grand nombre de résultats théoriques et d'outils de support, utilisables pour étudier les comportements fonctionnels, temporels et stochastiques. »

Les approches basées sur les réseaux de Petri présentent un grand nombre d'avantages. En effet, les réseaux de Petri permettent de modéliser explicitement les notions d'état et de changement d'état. Ils permettent également de prendre en compte un nombre infini d'états et ils supportent l'entrelacement et la concurrence.

Les réseaux de Petri sont constitués de trois types d'éléments (Figure II.11) :

- **Les places** : représentées par des ellipses, permettent de modéliser les variables d'état du système.
- **Les transitions** : représentées par un rectangle, correspondent à un opérateur de changement d'état.
- **Les arcs** : permettent de relier les places et les transitions. Un arc allant d'une place à une transition indique une condition nécessaire au changement d'état du système. Un arc allant d'une transition à une place précise l'impact du changement d'état du système.

Les réseaux de Petri à objets reposent sur l'introduction des nouveautés suivantes, les jetons circulant dans un réseau de Petri à objets sont des références à d'autres objets du système, ajoutant aux réseaux de Petri la notion de dynamicité qui permet d'utiliser des instances puisque les objets font référence à des classes d'objets et non plus à un objet.

- les arcs sont étiquetés par des noms de variables ;
- les transitions peuvent contenir des actions ;
- la franchissabilité d'une transition est conditionnée non seulement par la présence de jetons dans ses places d'entrée, mais aussi par une précondition portant sur la valeur de ces jetons. »

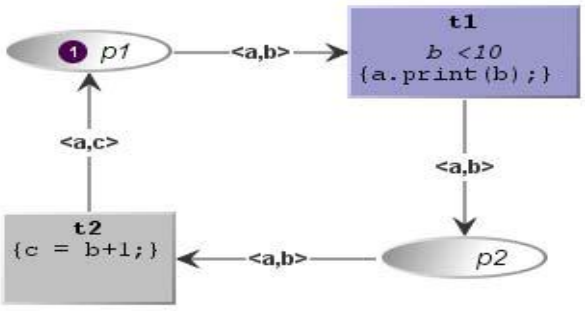


Figure II.12 Réseau de Petri à objets

Sur la Figure II.12, par exemple, les jetons circulant dans le réseau sont des couples constitués de :

- un objet a de type chaîne de caractères ;
- un entier b.

Supposons, par exemple, que dans l'état initial, le jeton de la place p1 contienne un objet a du bon type et de l'entier 0.

La transition t1 possède :

- Une précondition b < 10, portant sur l'entier contenu par le jeton (ici, b vaut 0, la transition t1 est donc franchissable) ;
- Une action print(a) qui affiche le contenu de l'objet a sur l'écran.

La transition t2 ne possède pas de précondition, mais possède une action c = b +1.

Lorsqu'elle est franchie, elle prend le jeton de la place p2 (ici, <a, 0>) et dépose un jeton dans la place p1 (ici, <a, 1>).

La transition t1 est de nouveau franchissable, tant que la valeur de b est inférieure à 10 (c'est à-dire jusqu'à ce que la transition t2 ait été franchie dix fois).

Contrairement aux réseaux de Petri classiques, les jetons consommés lors du franchissement d'une transition transportent des valeurs qui peuvent être utilisées dans l'exécution de cette transition.

Les réseaux de Petri à objets définissent la notion de substitution comme l'ensemble des couples (nom de variables de l'arc, valeur correspondante du jeton). Dans l'exemple de la Figure II.12, la substitution pour laquelle t1 est franchissable est {{a => x}, {b => 0}}.

II.3.1.1 *Les Objets Coopératifs*

Le formalisme des Objets Coopératifs est constitué d'un ensemble de classes d'objets coopératifs (les CO-classes) qui coopèrent au moyen d'un protocole de type client-serveur. Une CO-classe spécifie une classe d'objets qui se conforme à une interface logicielle (qui décrit des services et leurs signatures) et utilise les réseaux de Petri à objets pour décrire le comportement des objets. L'interface logicielle est décrite à l'aide du langage Java.

Le réseau de Petri décrivant le comportement d'un objet sera appelé ObCS (pour Structure de Contrôle de l'Objet, Object Control Structure). Ceci nous donne la définition suivante :

CO-classe = interface Java + ObCS (décrit par un réseau de Petri de haut-niveau).

Le formalisme des Objets Coopératifs se distingue par quatre caractéristiques :

- des objets dans des réseaux de Petri. Comme tout réseau de Petri à objets (OPN), l'ObCS d'une CO-classe contient des jetons qui font référence à des objets. Cette caractéristique permet d'utiliser les techniques de composition et de décomposition de l'approche à objets afin de construire le modèle d'un système.
- des réseaux de Petri dans des objets. Le comportement d'une classe d'objets peut être représenté par un réseau de Petri à objets, permettant ainsi de décrire précisément des comportements pouvant être concurrents.
- un protocole client-serveur. Le formalisme des Objets Coopératifs définit une communication entre objets en termes de réseaux de Petri. Ainsi, un système est modélisé comme une collection d'objets qui interagissent en appelant mutuellement leurs services.
- la communication par évènements. Le formalisme des Objets Coopératifs permet la communication par évènements entre les objets.

II.3.1.1.1 Notion de services dans les Objets Coopératifs

L'interface logicielle associée à chaque CO-classe permet de mettre en avant l'ensemble des services offerts par cette classe.

```
Public interface MouseListener extends EventListener {
Public void mouseClicked (MouseEvent e);
Public void mousePressed (MouseEvent e);
Public void mouseReleased (MouseEvent e);
}
```

Figure II.13 Interface java java.awt.event.MouseListener

La Figure II.13 présente l'interface Java java.awt.event.MouseListener. Nous pouvons voir qu'il offre trois services MouseClicked (MouseEvent), mousePressed (MouseEvent e), mouseReleased (MouseEvent e) mais comme toute interface Java, elle ne définit pas la manière dont fonctionneront ces services. Cette ambiguïté est levée par la CO-Classe qui implémente ces fonctions.

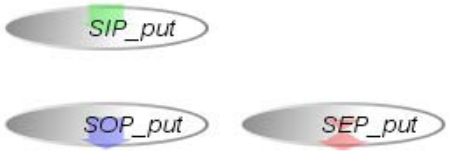


Figure II.14 Triplet (SIP, SOP, SEP) de places générées à partir de l'interface java

Pour chacune des fonctions de l'interface, trois places sont créées. Une place d'entrée (SIP), une place de sortie (SOP) et une place exception (SEP). Lors de l'exécution, ce triplet (SIP, SOP et SEP) Figure II.14 fonctionne de la manière suivante :

- L'appel du service consiste à déposer un jeton contenant les paramètres de l'appel dans la place d'entrée du service (SIP).
- Rendre le service revient à déposer un jeton contenant le résultat de la requête dans la place de sortie du service (SOP).
- La notification d'une erreur d'exécution du service revient à déposer un jeton décrivant l'erreur de la requête dans la place exception (SEP).

Figure II.15 Modélisation en CO de l'interface java.awt.event.MouseListener

Figure II.15 présente l'instanciation des méthodes de l'interface java.awt.event.MouseListener. En prenant l'exemple de la méthode MouseClicked à gauche sur la Figure, nous trouvons donc la place d'entrée SIP_mouseClicked, la place de sortie SOP_mouseClicked et la place exception SEP_mouseClicked. Lors d'un appel de méthode mouseClicked, un jeton est déposé dans la place d'entrée avec comme valeurs, les différents paramètres de la fonction. Nous pouvons ensuite utiliser ces différents paramètres pour, par exemple, ajouter des nouveaux objets dans le réseau ou modifier les paramètres de ces objets. Pour rendre la main à l'application, un jeton est placé dans la place de sortie avec comme valeurs les paramètres de sorties de la fonction. Si nous désirons envoyer une exception, un jeton est déposé dans la place exception. L'exécution de ces fonctions est synchrone.

II.3.1.1.2 Notion d'évènement dans les objets coopératifs

En plus de la notion de services permettant la communication synchrone et unicast (un à un) entre modèles, la notion d'évènements a été intégrée dans les Objets Coopératifs afin de permettre la communication asynchrone et multicast (à plusieurs autres modèles).

Les évènements sont échangés par les objets coopératifs à l'aide de transitions particulières appelées transitions synchronisées et de gestionnaires d'évènements. Ces transitions synchronisées sont liées à un gestionnaire d'évènement (ou event Handler) correspondant à un évènement.

Pour pouvoir être franchies, ces transitions synchronisées doivent non seulement être franchissables comme toute autre transition mais également recevoir l'évènement de l'event handler correspondant qui sert de déclenchement au tir de la transition.

Pour réaliser cette communication par évènements, trois rôles sont nécessaires : l'Emetteur, le Récepteur et l'Abonnement.

II.3.1.1.2.1 L'Emetteur d'évènements

Un modèle capable d'émettre un évènement implémente une interface permettant l'abonnement par d'autres modèles. Cet abonnement est réalisé en appelant une méthode addEventListener (nom, récepteur) où le premier paramètre représente le nom de l'évènement et le second le nom du modèle récepteur.

L'interface contient également d'autres méthodes permettant l'abonnement aux évènements relatifs aux changements d'état du modèle émetteur. Ces évènements sont présentés dans le Tableau II.3.

Nom	Evénement
TokenAdded	Un jeton entre dans une place.
TokenRemoved	Un jeton sort d'une place.
MarkingReset	Le marquage d'une place est réinitialisé.
TransitionCompleted	Une transition est franchie.
Enabled	Un event handler devient activé.
Disabled	Un event handler devient désactivé.

Tableau II.3 Evènements de changement d'état des réseaux de Petri

La Figure II.16 présente un modèle émettant deux évènements : event1 et event2. L'émission d'évènements est déclenchée par la commande trigger (nom, paramètre) qui envoie l'évènement nom ayant comme valeur complémentaire paramètre.

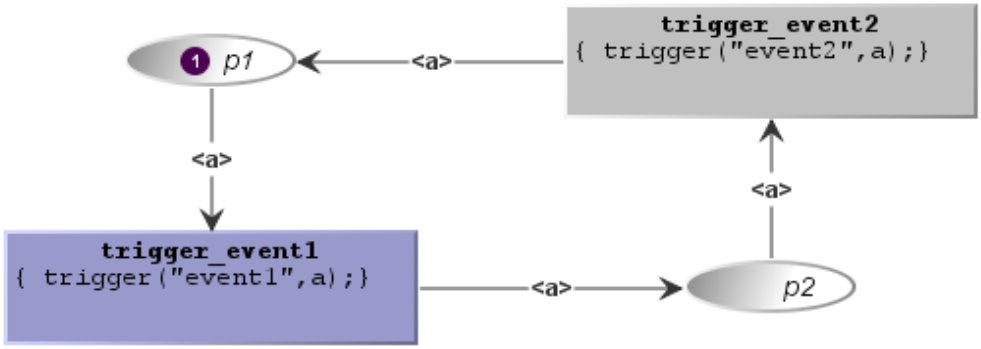


Figure II.16 Exemple de modèle Emetteur

Le comportement de ce modèle est le suivant : Au départ, la place p1 contient un jeton ;

- La transition trigger_event1 est donc franchissable. Un jeton est retiré de la place p1. Lors du franchissement de t1, un évènement nommé event1 est levé contenant a en paramètre. Un jeton est ensuite déposé dans p2.
- La transition trigger_event2 est alors franchissable. Un jeton est retiré de la place p2. Lors du franchissement de trigger_event2, un évènement nommé event2 est levé contenant a en paramètre. Un jeton est ensuite déposé dans p1.

II.3.1.1.2.2 Le Récepteur d'évènements

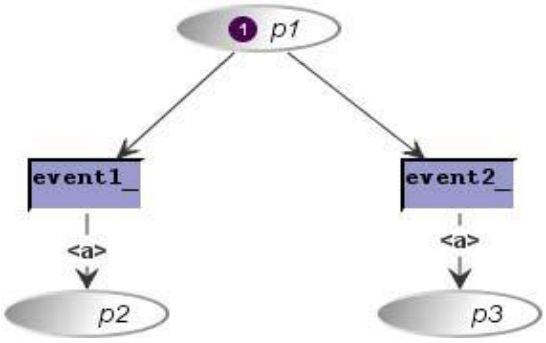


Figure II.17 Exemple de modèle récepteur

Un modèle récepteur implémente une interface ayant une méthode appelée acceptEvent permettant de recevoir les évènements auxquels ce modèle est abonné. La Figure II.17 présente un modèle écoutant deux évènements event1 et event2.

Le comportement de ce modèle est le suivant : Au départ, la place p1 contient un jeton. Les transitions event1_ et event2_ sont franchissables. Il s'agit de transitions synchronisées. Ces transitions attendent l'arrivée d'un évènement pour pouvoir être franchies. Si l'évènement event1 est reçu, alors un jeton contenant le paramètre est déposé dans la place p2. Il en va de même pour event2.

II.3.1.1.2.3 L'Abonnement aux évènements

Chaque instance Récepteur doit être abonnée à l'Emetteur afin de pouvoir traiter l'évènement. Cet abonnement est réalisé par un modèle tiers. La Figure II.18 représente un exemple de modèle tiers responsable de l'abonnement.

Le comportement de ce modèle est le suivant : la place Init_Emetteur et la place Init_Receveur contiennent chacune un jeton. La transition creationEmetteur va prendre un jeton de la place Init_Emetteur, exécuter l'action emetteur = create Emetteur () qui permettra la création d'une instance de la classe Emetteur. Un jeton contenant la référence de l'émetteur sera déposé dans la place EmetteurCréé. La transition créationReceveur va prendre un jeton de la place Init_Receveur, exécuter l'action receveur = create Recepteur () qui permettra la création d'une instance de la classe Récepteur. Un jeton contenant la référence du récepteur sera déposé dans la place ReceveurCréé.

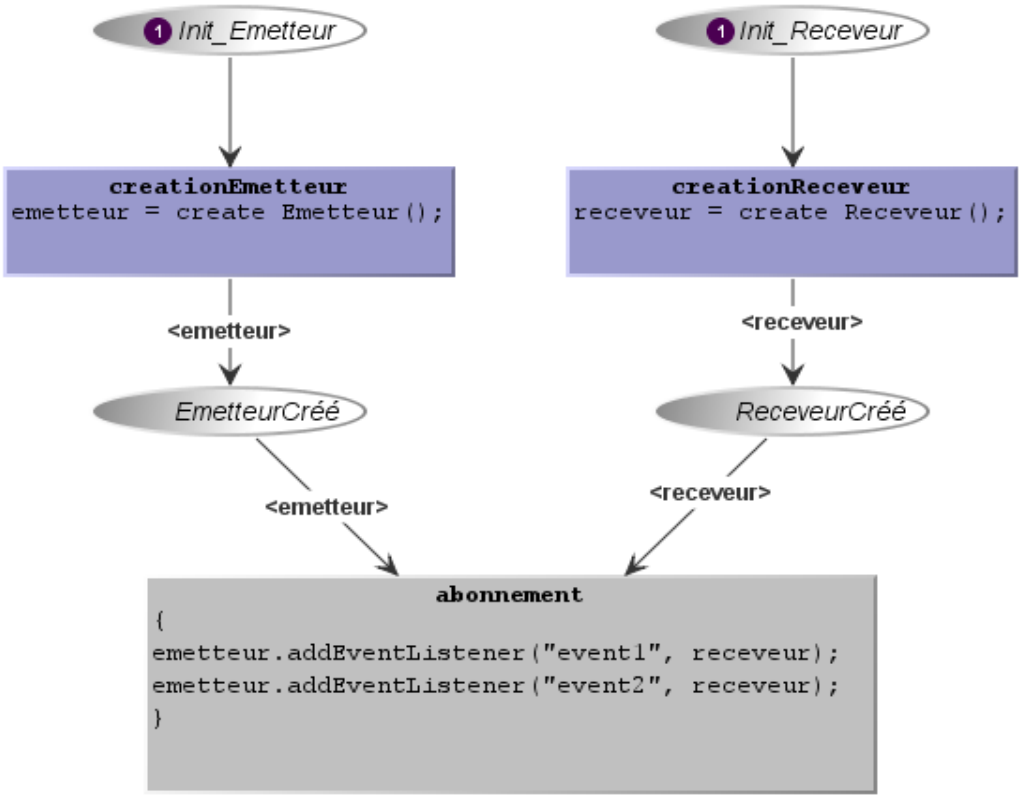


Figure II.18 Exemple de modèle abonnement

Lorsque qu'un jeton est dans la place EmetteurCréé et un autre dans la place ReceveurCréé, la transition abonnement devient franchissable. Lorsque la transition est franchie, un jeton est retiré de la place EmetteurCréé et un autre de la place ReceveurCréé.

L'action de la transition abonnement est double :

- la première ligne emetteur.addEventListener ("event1", receveur) indique que l'instance receveur va écouter l'évènement event1 émis par l'instance émetteur de la classe Emetteur.
- la seconde ligne emetteur.addEventListener ("event2", receveur) abonne de la même manière l'instance receveur à l'évènement event2.

II.3.2 Les objets coopératifs interactifs

Les objets coopératifs interactifs (ICO) (Palanque, 1992) sont une extension des CO (objets coopératifs) permettant de prendre en compte les systèmes interactifs.

Les ICO permettent de décrire :

- Le comportement du système et le dialogue grâce à un ensemble de CO-Classe que nous avons présenté dans les sous-sections précédentes.
- L'apparence graphique du système interactif grâce à la partie présentation composée éventuellement des objets d'interaction pour présenter l'information à l'utilisateur.
- Le lien entre l'aspect comportement du système et l'aspect graphique se fait grâce à:
    - La fonction de rendu qui maintient la consistance entre l'état de la CO-Classe et son apparence.
    - La fonction d'activation qui indique comment les évènements sur les systèmes d'entrée seront reçus par des transitions de la CO-Classe associées aux évènements ainsi que le lien entre la disponibilité des évènements entrées de l'utilisateur et des transitions de la CO-Classe.

Ces différents éléments sont représentés sur ARCH dans la Figure II.19.

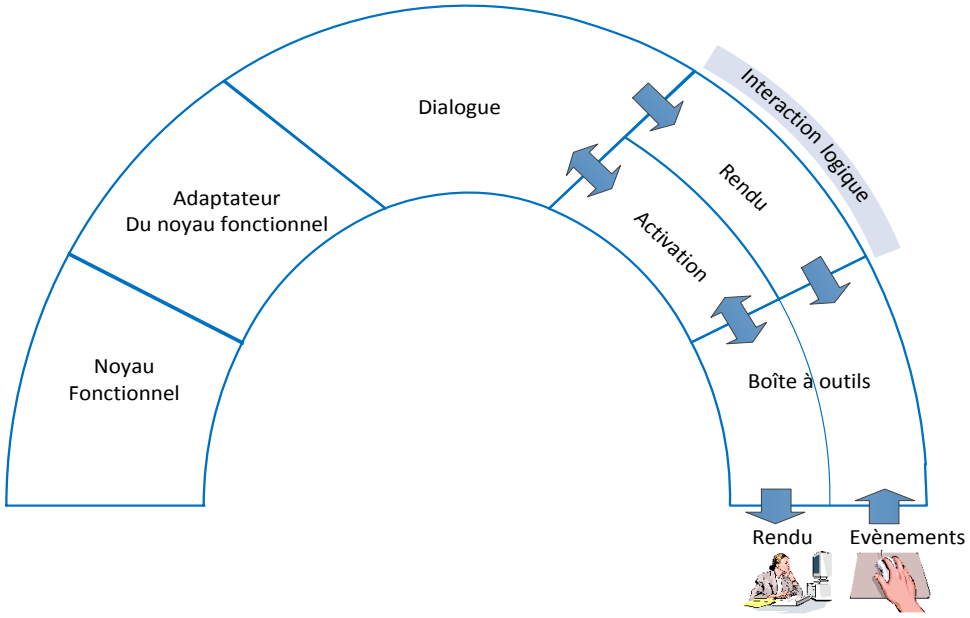


Figure II.19 Représentation sur ARCH des éléments d'ICO

❖ La boîte à outils

La partie boîte à outils représente les périphériques d'entrées-sorties ainsi que les widgets. Nous nous intéressons ici aux widgets en particulier. Chaque widget peut être un moyen d'interagir avec le système interactif (interaction de l'utilisateur sur le système) et / ou un moyen pour afficher les informations de ce système interactif (système d'interaction vers l'utilisateur). Même si les méthodes utilisées pour le rendu (description et / ou code) sont hors de la portée d'une spécification ICO, il est possible qu'elles soient gérées par un ICO, décrivant la partie de présentation comme un ensemble de méthodes de rendu (en vue d'afficher les changements d'état et la disponibilité des gestionnaires d'évènements) et un ensemble d'évènements utilisateur.

Ci-dessous l'exemple d'une représentation graphique d'une gestion de 4 saisons. Cette partie graphique possède comme widgets un label et quatre boutons.

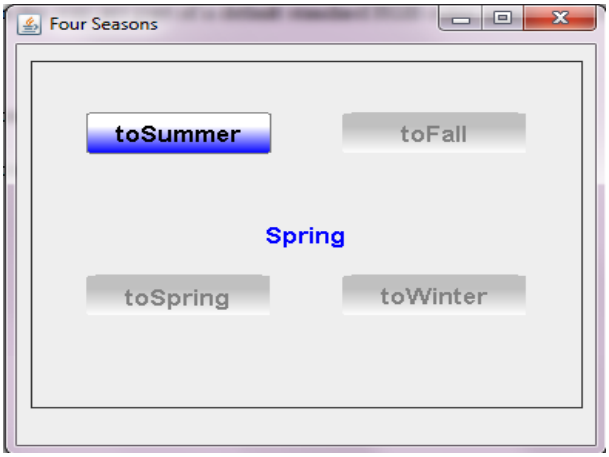


Figure II.20 Représentation graphique d'une gestion des 4 saisons

❖ Le Dialogue

Le modèle de dialogue définit le comportement du système interactif à l'aide des objets coopératifs. A la Figure II.21 est représenté le comportement du système interactif de l'interface graphique de l'exemple des 4 saisons.

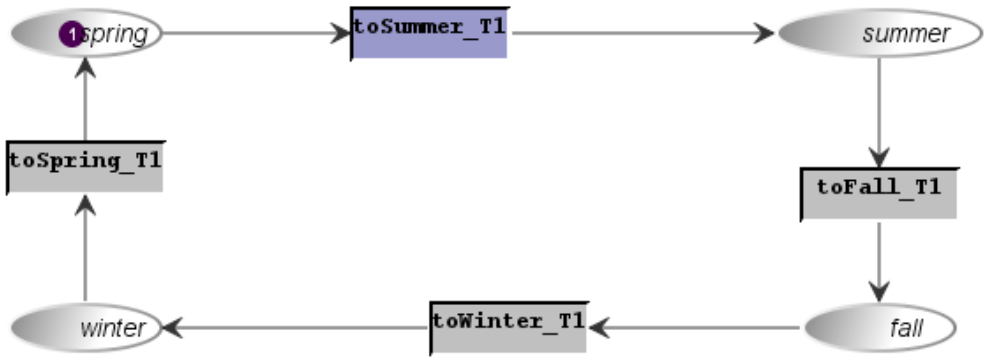


Figure II.21 Réseau de Petri représentant le cycle des saisons

❖ L'interaction logique

L'interaction logique qui joue le rôle de médiateur entre le dialogue et la boîte à outils est composée des fonctions d'activation et de rendu. Elle offre une représentation logique des widgets qui est indépendante de la plate-forme.

○ Fonction de rendu

Dans un réseau de Petri, les places sont les variables d'état du système (l'état du système étant modélisé par la distribution de jetons dans les différentes places) et les transitions sont les opérateurs de changement d'état. La spécification du rendu peut donc être liée aux places des réseaux de Petri. Un changement d'état ne peut être représenté que par l'entrée ou la sortie d'un jeton d'une place.

Si nous ajoutons la description de l'état initial, à chaque place d'un ObCS peuvent être associées trois méthodes de rendu pour trois cas bien distincts :

- à chaque fois qu'un jeton entre dans la place (par le franchissement d'une transition),
- à chaque fois qu'un jeton est retiré de la place,
- à chaque fois que l'on remet cette place dans son état initial

La présentation d'informations auprès de l'utilisateur se décrit à travers la fonction de rendu qui met en relation les places et les transitions du dialogue et un ensemble d'objets d'interaction, qui peuvent être utilisés pour présenter (rendre) de l'information à l'utilisateur.

Nous pouvons représenter un lien entre le couple place et évènement à une méthode de rendu de l'exemple des 4 saisons dans le Tableau II.4 :

Place	Evènement	Méthode de rendu
Printemps	Réinitialisation	renderStateSpring ()
Printemps	TokenAdded	renderStateSpring ()
Eté	TokenAdded	renderStateSummer ()
Automne	TokenAdded	renderStateFall ()
Hiver	TokenAdded	renderStateWinter ()

Tableau II.4 lien entre les couples Place-Evènement et la méthode de rendu

Par exemple, la dernière ligne lie l'évènement TokenAdded (Jeton_entré) dans la place Hiver à la méthode renderStateWinter () sur la partie présentation. Lorsqu'un jeton est déposé dans la place Hiver du modèle de dialogue, la méthode renderStateWinter () est exécutée sur la partie présentation.

Le réseau de Petri correspondant de la fonction de rendu est représenté à la Figure II.22.

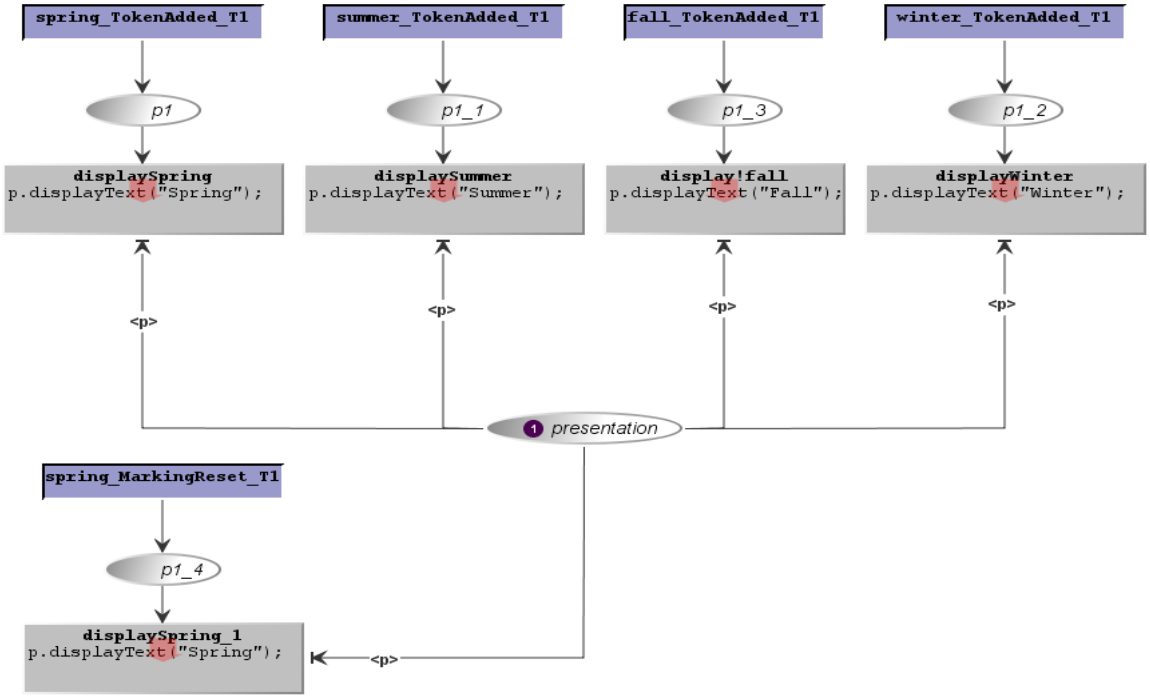


Figure II.22 Réseau de Petri de la fonction de rendu

o La fonction d'activation

L'activation des objets de l'interaction représente en partie l'espace d'interaction de l'utilisateur, en permettant de déterminer l'ensemble des actions utilisateur (ou évènements) qui auront un impact sur le système interactif. Plus précisément, la fonction d'activation décrit le lien qui existe entre la disponibilité de services utilisateur et la possibilité d'accomplir une action sur un objet de l'espace d'interaction. Cet espace d'interaction est réduit aux états possibles du dialogue et à l'ensemble des évènements jugés pertinents pour l'interaction avec le système modélisé.

Dans une spécification par ICO, l'ensemble des interactions possibles peut être déduit du marquage courant de l'ObCS des différentes CO-classes qui modélisent le dialogue du système interactif.

La fonction d'activation permet:

- l'envoi d'évènements vers le réseau de Petri lors d'actions sur les objets de l'interface,
- le lien entre le dialogue et la présentation active les comportements possibles des objets graphiques

Toujours par rapport à l'exemple des 4 saisons, nous pouvons représenter cette interaction par un tableau liant les actions sur les objets aux évènements qui seront envoyés au réseau du dialogue. Tableau II.5

Action	**Evènement envoyé**	**Méthode d'activation**
Click sur le bouton toSpring	toSummer	setSpringEnabled ()
Click sur le bouton toSummer	toFall	setSummerEnabled ()
Click sur le bouton toFall	toWinter	setFallEnabled()
Click sur le bouton toWinter	versPrintemps	setWinterEnabled ()

Tableau II.5 Liens entre les actions sur les objets et les évènements envoyés au réseau

Lorsque un gestionnaire d'évènements change d'état (d'activé à désactivé ou de désactivé à activé), la fonction d'activation exécute le code d'activation/désactivation sur l'objet graphique correspondant. Dans notre exemple, à la suite du franchissement de la transition *toSummer*, l'évènement *toSummer* est désactivé et l'évènement *toFall* est activé. La fonction d'activation exécute alors le code (setSummerEnabled()) avec comme variable le booléen false (étant donné que l'évènement *toSummer* passe à désactivé) ce qui a pour résultat de désactiver le bouton *toSummer* et exécute aussi le code (*setFallEnabled*()) avec comme variable le booléen true (étant donné que l'event handler *toFall* passe à activé) ce qui a pour résultat d'activer le bouton *toFall*. Le réseau de Petri correspondant de la fonction d'activation est représenté à la Figure II.23.

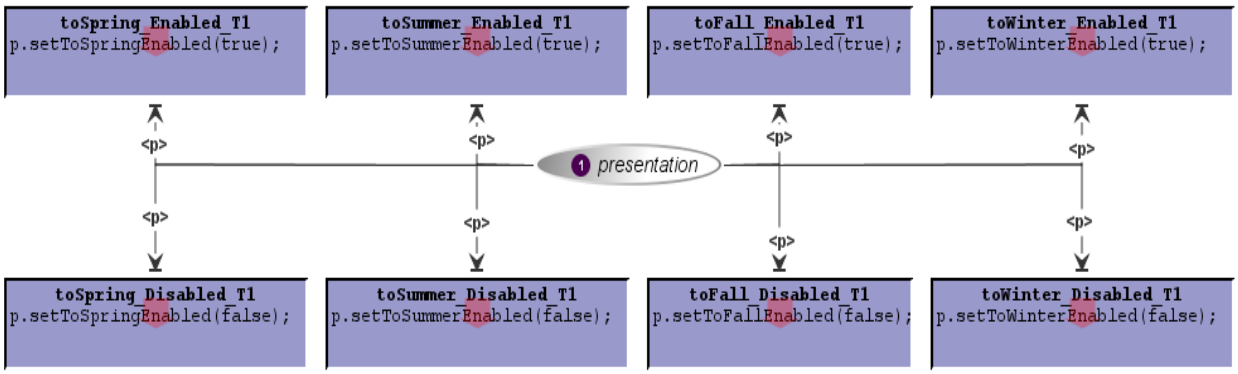


Figure II.23 Réseau de Petri de la fonction d'activation

II.3.3 Les ICompoNets

Il est assez difficile de se faire une représentation visuelle de l'architecture d'un système uniquement avec les réseaux de Petri. Il parfois nécessaire d'avoir une vision abstraite des composants constituant l'architecture du système, comment ces composants dont interconnectés, les services fournis et reçus par chaque composant et la source de ces services.

Durant sa thèse Eric Barboni (Barboni, 2006) a proposé des modèles de composant appelés ICompoNets permettant de modéliser les architectures de systèmes interactifs.

Les ICompoNets sont inspirés de l'enveloppe de composant CCM (Corba Component Model) (OMG, 2008), ceci en partant de la constatation que l'interaction homme machine est dirigée par des évènements et que le modèle CCM explicitait les différents type de connections à savoir la communication par évènements ainsi que par appel de méthode.

Un ICompoNet est constitué d'un comportement décrivant l'enchaînement de ses états internes et d'un ensemble de ports décrivant les services offerts ou requis par le composant. La syntaxe graphique des ICompoNet est proche de CCM (Figure II.24).

Les ports décrivent la manière dont le composant peut être mis en relation avec ses pairs. Les ports sont de quatre types (Facet, Receptacles, Source, Sink).

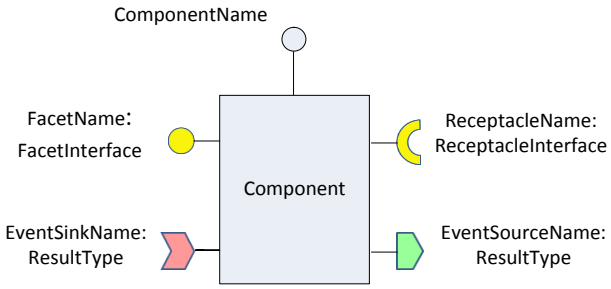


Figure II.24 Syntaxe graphique des CompoNets

- Les facettes (*Facet*) : représentent l'ensemble des fonctionnalités fournies aux autres composants.
- Les réceptacles (*Receptable*) : représentent l'ensemble des fonctionnalités nécessaires au fonctionnement du composant.
- Les sources d'évènement (*EventSource*) : représentent les évènements produits par un composant.
- Les puits d'évènement (*EventSink*): représentent les évènements que le composant est susceptible de recevoir.

L'assemblage est composé d'un ensemble de connections permettant de mettre en relation des ports. Les connections étant de deux types :

Des connections sources/puits : responsables de la communication via évènements, qualifié d'asynchrone multicast. Sur le premier schéma de la Figure II.25, le composant A va signaler par évènement

une information dont B a besoin pour continuer son traitement, A continue son déroulement sans être bloqué.

Des connections facette/receptacles : responsables de la communication via appel de méthode, qualifié de synchrone unicast. Sur le second schéma de la Figure II.25, le composant A est demandeur d'une opération dans B, A va rester en attente jusqu'à la fin de l'appel de cette méthode.

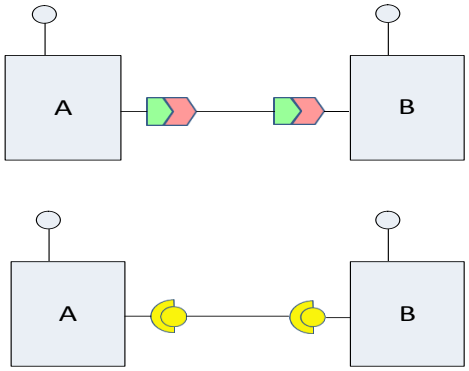


Figure II.25 Les connections

Le comportement d'un ICompoNet est décrit sous forme d'ICO ou d'un assemblage d'ICompoNets.

Si l'on reprend l'interface des quatre saisons de la Figure II.20, il est composé d'un label et de quatre boutons. L'assemblage ICompoNet de cette application est représenté à la Figure II.26.

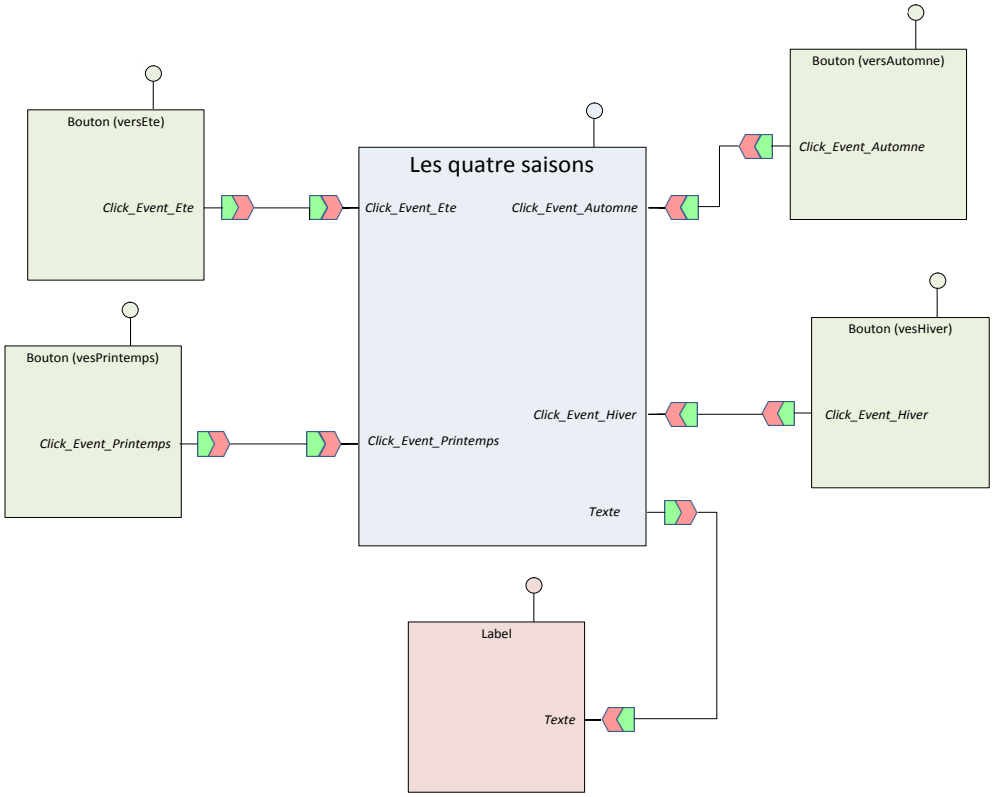


Figure II.26 Assemblage ICompoNet décrivant les 4 saisons

Le principe de fonctionnement des 4 saisons est le suivant : le label affiche la saison courante et seul le bouton de la saison suivante est actif donc disponible pour recevoir un click de l'utilisateur. Pour gérer ce comportement, l'application quatre saisons a besoin de recevoir les évènements de click des boutons pour rendre actif le bouton correspondant et non actif les autres. L'application doit aussi fournir au label le texte à afficher en fonction du click qu'il a reçu.

Nous avons vu dans la section précédente le comportement ICO de l'ICompoNet « les quatre saisons » de la Figure II.26, les autres ICompoNets (les widgets bouton et label), peuvent aussi avoir leur comportement décrit en ICO. La représentation en ICompoNet de la Figure II.26 donne ainsi une vision globale de l'architecture des composants et des services échangés.

II.3.4 L'environnement de travail PetShop

L'outil PetShop (Petri nets workShop) (Navarre, 2001) est un environnement d'édition, de vérification et d'exécution de réseaux de Petri de haut-niveau (ICO) écrit en JAVA (Figure II.27).

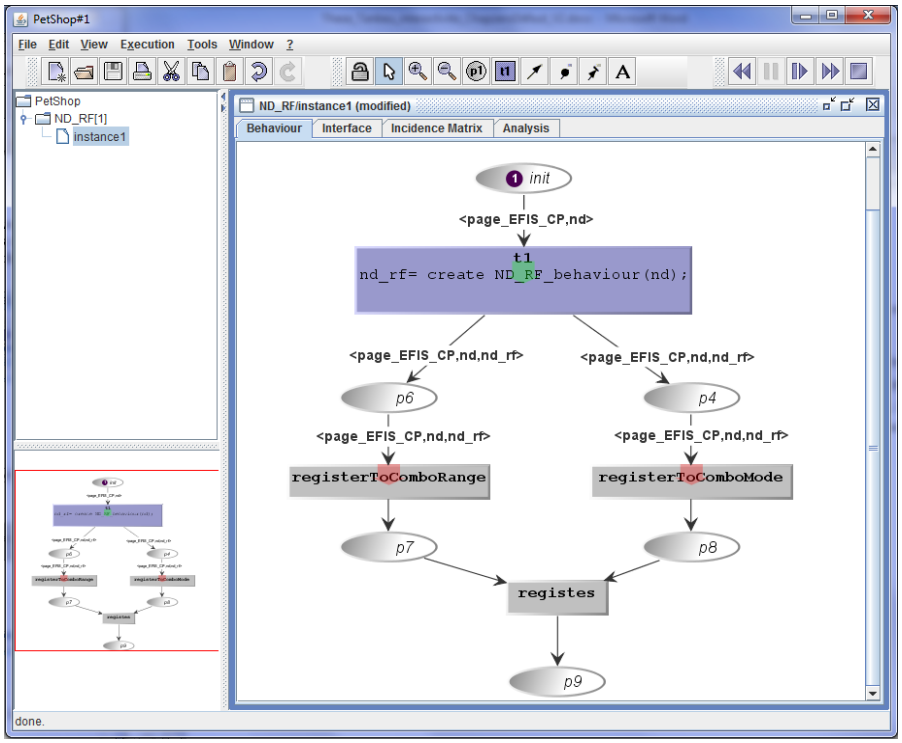


Figure II.27 Environnement PetShop

PetShop a pour objectif la modélisation de systèmes interactifs et permet :

- **L'Edition de la spécification comportementale** : l'environnement comprend un éditeur syntaxique des ObCS (réseau de Petri décrivant le comportement d'un objet).
- **La Génération à partir d'une interface Java** : l'objectif principal de cet environnement est la construction de la spécification comportementale du système à partir d'une interface Java (fournie par les concepteurs du système). En utilisant le « mapping » Interface-CO, un squelette de spécification comportementale est généré automatiquement.

- **L'analyse mathématique des modèles** : un module d'analyse permet de vérifier des propriétés sur les ObCS, aussi bien structurelles (les invariants, les conflits…) que comportementales (la vivacité, le caractère borné des places, la présence d'un état d'accueil…). PetShop inclut ainsi des algorithmes d'analyse classiques qui vérifient des propriétés sur le réseau de Petri Place/Transition sous-jacent à chaque ObCS.
- **L'interprétation et débogage des modèles** : un avantage reconnu des réseaux de Petri est leur caractère exécutable. Cette fonctionnalité est exploitée dans PetShop qui fournit un environnement interprété, avec certains avantages en termes de flexibilité, d'interactivité, de facilité d'utilisation et de productivité
- **La génération de fichiers log de la simulation des modèles** : PetShop permet également d'enregistrer sous forme de fichier XML, l'exécution des modèles. Nous présentons cette fonction dans la partie évaluation de l'utilisabilité.

II.3.4.1.1 Edition de la spécification ICO

L'outil PetShop permet l'édition des ObCS en générant les places d'entrée, de sortie et d'erreur des modèles ayant une interface java , en éditant dans les ObCS les échanges d'évènement (Figure II.28), les appels de méthodes et les fonctions de rendu et d'activation.

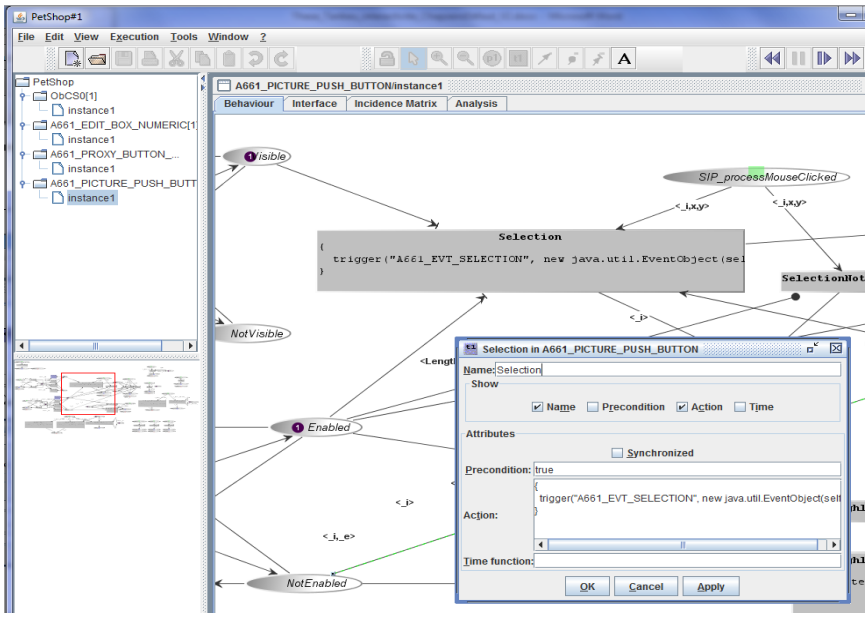


Figure II.28 Edition du code contenu dans une transition

L'édition des ICO se fait graphiquement en utilisant la palette de la barre d'outils (Figure II.29). La partie gauche de cette barre d'outils est utilisée pour les fonctions classiques telles que l'ouverture ou la sauvegarde de fichiers ou les fonctions copier, couper, coller. La partie droite permet la sélection des éléments, la visualisation (zoom) et l'ajout des places, des transitions et des différents types d'arcs

Figure II.29 Palette d'édition de modèle dans PetShop

II.3.4.1.2 Analyse formelle

Un module d'analyse des réseaux de Petri (Figure II.30) est implémenté dans l'outil PetShop permettant de vérifier certaines propriétés des modèles telles que les invariants ou les matrices d'incidence. Tout comme pour la simulation, cette analyse formelle est accessible directement pendant la conception (sans devoir lancer un outil d'analyse extérieur ou compiler notre modèle) et est mise à jour à chaque modification de l'ObCS.

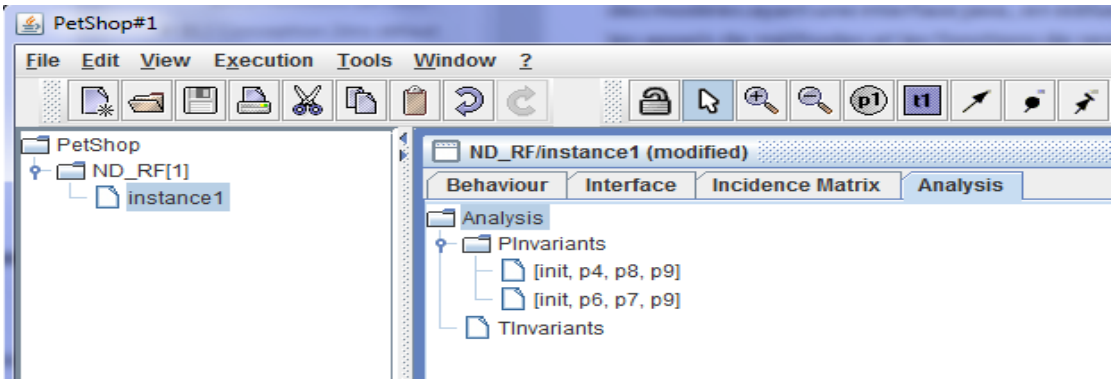


Figure II.30 Analyse d'invariants dans PetShop

II.3.4.1.3 La simulation

La simulation de modèles est fortement couplée à l'édition afin de pouvoir tester de façon interactive le comportement du modèle en cours de conception (sans besoin de phase de compilation préalable ou autre changement de mode).

Un interprète d'ICO est associé à chacune des instances d'ICO dans PetShop. L'exécution d'une instance est pilotée au moyen d'un contrôleur d'instances, semblable à une commande de magnétoscope.

Figure II.31 Palette de simulation de modèle dans PetShop

PetShop offre la possibilité d'exécuter une spécification à la manière d'un débuggeur c'est-à-dire qu'il est possible d'exécuter la spécification courante en mode automatique ou en mode manuel. Le mode automatique permet de laisser à l'interprète le libre choix dans les substitutions à employer pour le franchissement. Ce mode automatique peut être réglé pour le pas à pas ou pour une exécution illimitée. Il permet également de revenir au marquage initial et de mettre en pause. Le mode manuel permet au concepteur de choisir la transition à franchir et éventuellement la substitution à effectuer.

Son API permet aux spécifications ICO de recevoir les évènements provenant de l'utilisateur (comme un clic) ou les évènements de changements d'état du système ce qui permet d'animer le réseau. Lors de la réception d'un évènement, la transition synchronisée correspondant à cet évènement peut être franchie. Pour être franchie, cette transition synchronisée doit être franchissable au sens classique du terme et elle doit recevoir l'évènement qui sert de déclencheur à son franchissement.

De plus, lors de l'exécution d'un modèle, l'API PetShop permet de lever des évènements suite aux changements d'états du modèle tels que le franchissement d'une transition et permettre à d'autres modèles d'être abonnés à ces évènements. Cette communication par évènements permet de réaliser le rendu et l'activation des notations ICO.

II.4 LES PRINCIPES DE BASE DE LA SURETE DE FONCTIONNEMENT

Nos travaux portent sur la sûreté de fonctionnement des systèmes interactifs des Cockpits. Pour mieux comprendre la notion de sureté de fonctionnement, nous ferons dans cette section, une synthèse des principaux concepts de base de la sûreté de fonctionnement définies dans (Laprie, et al., 1996) et mis à jour dans (Laprie, et al., 2004).

La sûreté de fonctionnement d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre. Le service délivré par un système est son comportement tel qu'il est perçu par ses utilisateurs. Ces derniers peuvent êtres humains ou physiques. La sûreté de fonctionnement informatique s'articule autour de trois principaux axes : les attributs qui la caractérisent, les entraves qui empêchent sa réalisation et les moyens de l'atteindre (Figure II.32).

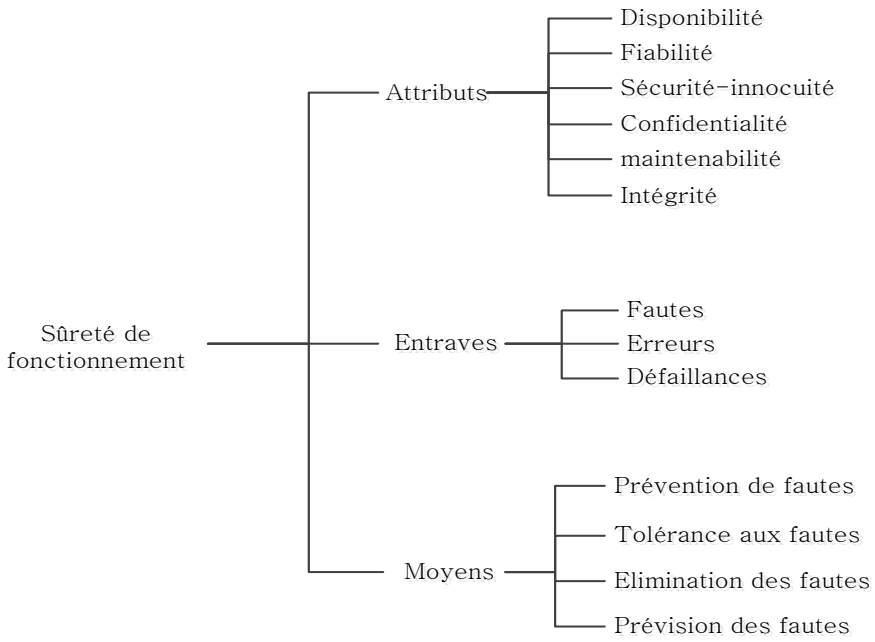


Figure II.32 Arbre de sûreté de fonctionnement

II.4.1 Les attributs

La sûreté de fonctionnement d'un système peut être vue selon des propriétés différentes mais complémentaires. Ces propriétés sont appelées les attributs de la sûreté de fonctionnement. On distingue six attributs :

- La disponibilité : Qui est l'aptitude du système à être prêt à l'utilisation. On s'intéresse ici au fait que le service fournit est correct au moment où l'utilisateur en a besoin.
- La fiabilité : exprime la continuité du service. On ne prend pas compte ici la sévérité des défaillances.

- La sécurité-innocuité (safety) : absence de conséquences catastrophiques pour l'environnement, c'est-à-dire celles pour lesquelles les conséquences sont inacceptables vis-à-vis du risque encouru par les utilisateurs du système.
- La confidentialité : absence de divulgations non autorisées de l'information.
- L'intégrité : absence d'altérations inappropriées de l'information
- La maintenabilité : aptitude aux réparations et aux évolutions.

Les entraves couvrent de la non-sûreté de fonctionnement.

II.4.2 Les entraves à la sûreté de fonctionnement

Une entrave est la cause ou le résultat de la non-sûreté de fonctionnement. On en distingue trois: les fautes, les erreurs et les défaillances (Figure II.33).

Une défaillance du système survient lorsque le service délivré dévie de l'accomplissement de la fonction du système. Une erreur est la partie de l'état du système qui est susceptible d'entrainer une défaillance. La cause adjugée ou supposée d'une erreur est une faute.

Figure II.33 Faute, Erreur, Défaillance

Les fautes et leurs sources sont extrêmement diverses, elles peuvent être classées selon différents critères : le moment où elles apparaissent dans le cycle de vie du système (faute de conception, faute opérationnelle), leur origine par rapport aux frontières du système (faute interne, faute externe), leur origine phénoménologique (faute naturelle, faute humaine), de leur dimension (faute logicielle, faute matérielle), de leur objectif (faute malicieuse, faute non-malicieuse), de leur intention (faute intentionnelle, faute non-intentionnelle), de la compétence des utilisateurs (faute d'incompétence, faute accidentelle) ou de leur persistance (faute persistante, faute transitoire).

Une erreur est la deuxième entrave à la sureté de fonctionnement, c'est la conséquence sur l'état du système de l'activation d'une faute. Une erreur affectant le service est une indication qu'une défaillance peut survenir si elle n'est pas traitée par des mécanismes appropriés. Une erreur peut être latente ou détectée ; une erreur est latente tant qu'elle n'a pas été reconnue en tant que telle ; une erreur est détectée par un algorithme ou un mécanisme de détection. Une erreur peut disparaître sans être détectée. Par propagation d'un élément à un autre, une erreur crée de nouvelles erreurs. Une défaillance survient lorsque, par propagation, une erreur affecte le service délivré par le système. Cette défaillance peut alors apparaître comme une faute du point de vue d'un autre composant. On obtient ainsi la chaîne fondamentale suivante :

Défaillance →faute→erreur→défaillance→faute …

Pour minimiser l'impact de ces entraves sur les attributs retenus d'un système, la sûreté de fonctionnement dispose de moyens.

II.4.3 **Moyens de la sûreté de fonctionnement**

Ces moyens sont les méthodes et techniques qui permettent de conforter les utilisateurs quant au bon accomplissement de la fonction du système. Ces moyens peuvent être utilisés simultanément lors de la phase de conception et de développement d'un système sûr de fonctionnement. Ils sont classés en quatre moyens suivant l'objectif visé:

- La prévention : empêcher l'occurrence ou l'introduction de fautes.
- La tolérance : fournir un service qui remplit la fonction du système en dépit des fautes.
- L'élimination : réduire la présence (nombre, sévérité) des fautes.
- La prévision : estimer la présence, la création et les conséquences des fautes.

Ces moyens sont complémentaires et dépendants et doivent être utilisés de façon combinée. En effet, en dépit de la prévention des fautes grâce à des méthodes de conception et à des règles de construction rigoureuses, des erreurs surviennent, résultant en des fautes. D'où le rôle de l'élimination des fautes : lorsqu'une erreur est générée durant la vérification, un diagnostic est entrepris afin de déterminer la, ou les fautes causes de l'erreur, en vue de les éliminer. Il est également nécessaire d'effectuer de la prévision de fautes.

Nous partons de l'hypothèse que les fautes peuvent toujours exister, puisqu'il est difficile de s'assurer qu'on a traité toutes les fautes possibles d'un système. Nous nous intéressons ci-dessous à la tolérance aux fautes.

II.4.4 **La tolérance aux fautes**

La tolérance aux fautes a pour but d'empêcher les défaillances malgré la présence ou l'occurrence de fautes. La tolérance aux fautes est mise en œuvre par le traitement des erreurs et par le traitement des fautes (Lee, et al., 1990). Le traitement d'erreur est destiné à éliminer les erreurs, de préférence avant qu'une défaillance ne survienne. Le traitement de faute est destiné à éviter qu'une ou des fautes ne soient activées de nouveau.

Nous présentons très brièvement dans les paragraphes suivants, les principes de tolérance aux fautes physiques et de conception qui sont considérées dans cette thèse, ceci après avoir expliqué les modèles de fautes auxquels nous nous intéressons

II.4.4.1 *Les modèles de fautes*

Parmi le large spectre de fautes recensées dans les travaux sur la sûreté de fonctionnement, nous nous intéressons aux fautes physiques et de conception.

Les fautes physiques rassemblent les fautes affectant le matériel. Elles peuvent provenir soit d'un défaut de production interne au composant matériel, soit à l'environnement (perturbation électromagnétique). Les systèmes avioniques comme la plupart des systèmes embarqués sont aussi très exposé à ce type de fautes. La plupart de calculateur avioniques doivent par exemple implémenter des mécanismes de protection contre des phénomènes de radiation atmosphérique qui produisent des changements anormaux de bits dans les dispositifs de semi-conducteur.

Les fautes de conception concernent à la fois le matériel et le logiciel. Bien que la problématique ait été identifiée de longue date, les fautes de conception du logiciel posent toujours un défi. Un des défis est l'accroissement de la complexité des fonctionnalités et l'inflation du logiciel. On note par exemple la grande croissance du nombre de lignes de code dans les systèmes embarqués avioniques (douze millions de lignes pour l'A320, plus de vingt pour l'A340 et, soixante-cinq pour l'A380 !). Au niveau matériel, le nombre de composants explose également (le chiffre de 170 millions de transistors est affiché pour la dernière version (le Pentium 4), alors que la version de 1993 n'en comportait que 3,3 millions !) (Arlat, et al., 2006)

La connaissance de notre modèle de fautes permet de mieux cibler les mécanismes permettant de les tolérer.

Nous ne nous intéressons pas vraiment à l'origine de la faute, mais à la défaillance qu'elle peut causer sur le service délivré par le système : défaillance conduisant soit à un arrêt du système, ou produisant des valeurs erronées pouvant conduire le système à une situation jugée critique. L'objectif est donc de détecter la défaillance d'une tâche, et d'éviter que cette défaillance provoque la défaillance du système.

II.4.4.2 *Tolérance aux fautes physiques:*

Les différentes techniques développées pour tolérer les erreurs physiques sont basées sur la redondance. La redondance consiste à faire accomplir la même tâche par plusieurs unités qui doivent être indépendantes par rapport au processus de création et d'activation des fautes. Il faut donc s'assurer que soit les fautes sont créées ou activés indépendamment dans les unités, soit que si une même faute provoque des erreurs dans les unités, ces erreurs sont différentes.

II.4.4.2.1 Principe général de détection de fautes physiques

Le principe général de détection des fautes physiques est basé sur la notion de composant autotestable.

Un composant autotestable est obtenu en ajoutant à un composant purement fonctionnel, un module de contrôle observant si certaines propriétés entre les entrées et les sorties du composant fonctionnel sont vérifiées (Laprie, et al., 1996) (Figure II.34).

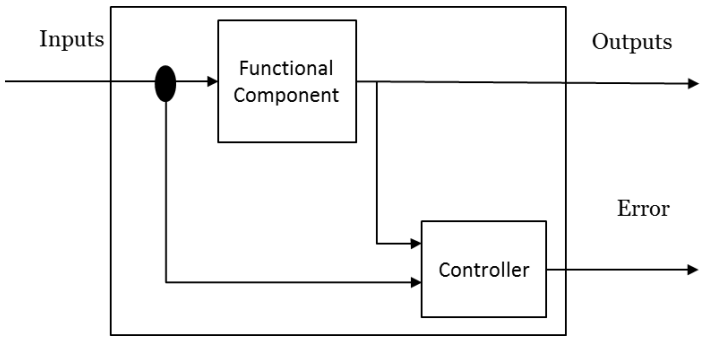


Figure II.34 Structure générale d'un composant autotestable

Un composant autotestable doit présenter deux propriétés :

- Sûreté en présence de faute : un composant est dit sûr en présence de fautes vis-à-vis d'un ensemble de fautes F, si quelle que soit la faute appartenant à F, toutes les configurations d'entrée conduisent soit à une sortie correcte, soit à des erreurs détectées.
- Autotestabilité : un composant est dit autotestable vis-à-vis d'un ensemble de fautes F, si pour toute faute appartement à F, il existe au moins une configuration d'entrée permettant de révéler la faute

Le module de contrôle du composant autotestable peut être soit un test d'acceptation qui vérifie la validité des sorties du fonctionnel par rapport à ses entrées. Soit une variante du fonctionnel plus un bloc de comparaison. Après la détection des fautes, pour assurer la continuité de service, des techniques de recouvrement sont mis en place.

II.4.4.2.2 Techniques de recouvrement des fautes physiques

Les techniques de recouvrement relatives aux fautes physiques sont:

- Recouvrement par point de reprise : elle consiste en la sauvegarde régulière de l'état du système de façon à pouvoir, après avoir détecté une erreur ramener le système dans un état antérieur. Elle nécessite donc un espace de stockage protégé et qui peut être important et de plus la réexécution à partir d'un état antérieur induit des retards dans l'exécution globale des tâches.
- Recouvrement d'erreur par poursuite, repose sur le même principe que la reprise dans le sens où l'on essaie de mettre le système dans un état correct, mais en acceptant cette fois de perdre certaines données, voire certaines étapes de traitement, quitte à dégrader temporairement les fonctionnalités du système. Ainsi, la poursuite consiste souvent à réinitialiser le système en essayant d'acquérir de nouvelles données depuis l'environnement externe d'exécution.
- Recouvrement d'erreur par compensation : elle consiste à avoir assez de redondance pour permettre en dépit des erreurs, de reconstituer un état exempt d'erreur. Un exemple est celui de l'utilisation en parallèle de deux composants autotestables : un composant est considéré comme unité principale le second est une copie à laquelle on fait appel en cas de détection d'erreurs. La compensation consiste alors à commuter d'une unité à une autre.

II.4.4.3 *Tolérance aux fautes de conception*

Il existe deux approches pour la tolérance aux fautes de conception :

1. L'utilisation de la programmation défensive (Rabéjac, 1995) consiste à ajouter des contrôles sur les entrées et les sorties des modules sous la forme d'assertions, servant à détecter s'il y a erreur et éventuellement déclencher un traitement d'exceptions. Le traitement d'exceptions constitue alors une manière d'éviter qu'une défaillance d'une tâche n'entraîne la défaillance de tout le système.
2. Diversification fonctionnelle, qui consiste à disposer d'au moins un autre composant à même d'assurer la tâche, conçu et réalisé séparément à partir de la même spécification. Les exemplaires d'un système produits selon l'approche de diversification fonctionnelle sont des variantes. L'objectif ici étant d'assurer la continuité de service. Il existe trois approches de bases dans la diversification fonctionnelle :

a. Les blocs de recouvrement (Randel, 1975): les variantes sont appelées des alternants et le décideur est un test d'acceptation, qui est appliqué séquentiellement aux résultats fournis par les alternants : si les résultats fournis par l'alternant primaire ne satisfont pas le test d'acceptation, l'alternant secondaire est exécuté, et ainsi de suite jusqu'à la satisfaction du test d'acceptation ou l'épuisement des alternants disponibles, auquel cas le bloc de recouvrement est globalement défaillant.
b. La programmation en N-versions (Avizienis, 1985). Les variantes sont appelées des versions, et le décideur effectue un vote sur les résultats de toutes les versions.
c. La programmation N-autotestable (Laprie, et al., 1990), au moins deux composants autotestables diversifiés sont exécutés en parallèle, chaque composant autotestable étant constitué soit de l'association d'une variante et d'un test d'acceptation, soit de deux variantes et d'un algorithme de comparaison.

Pour résumer, les techniques de tolérance aux fautes sont basées sur 3 principes :

- La redondance qui est la base de la tolérance aux fautes, elle consiste à avoir plusieurs exemplaires d'un même élément (matériel ou logiciel).
- La diversification : elle consiste à utiliser des éléments matériels et/ou logiciels qui ne sont pas issus d'un même processus de développement. La diversification logicielle consisterait à avoir par exemple d'une seule spécification fonctionnelle, plusieurs variantes de la fonction ; ceci afin d'éviter les fautes de conception.
- La ségrégation qui consiste à une séparation et isolation des exemplaires redondants. Cette ségrégation peut être spatiale et temporelle conformément à l'ARINC 653 (ARINC653, 1997) . Si l'on veut pousser plus loin, elle consisterait à faire que les exemplaire ne soient pas localisés au même endroit et ne soient pas alimentés par une même source. Les systèmes de commandes de vol sont particulièrement basés sur ces principes, de même que la plupart des systèmes critiques avioniques. (Traverse, et al., 2004)

Nous allons maintenant présenter quelques exemples de modèles de tolérance aux fautes dans l'avionique.

II.4.4.4 *Exemple de modèle de tolérance aux fautes dans l'avionique*

Les techniques de tolérance aux fautes occupent une place privilégiée dans les systèmes de commande de vol aéronautique (CDVE).

Chez Airbus le système CDVE utilise deux types de calculateurs numériques autotestables : 3 primaires (PRIM) et 3 secondaires (SEC). Les calculateurs SEC utilise un matériel différent des PRIM et des lois de pilotage simples. (Traverse, et al., 2004)

Le processus de développement de l'applicatif de ces calculateurs est basé sur la programmation N-autotestable avec N=2. Chaque calculateur est composé de deux unités matérielles de calcul quasiment identiques, mais ségréguées (les cartes sont dans des boitiers distincts, mais accolés). Le mode de fonctionnement général est le suivant :

− Une voie COM (commande ou command) élabore les consignes de commande des actionneurs et les émet vers, les actionneurs et vers sa voie MON qui la surveille, mais aussi vers les autres calculateurs.

– Une voie MON (moniteur ou monitor) élabore les mêmes consignes que COM mais ne fait que surveiller COM, et lève des alarmes vers d'autres systèmes de l'avion

A la sortie, les valeurs obtenues par chaque unité sont comparées. S'il y a un écart qui dépasse le seuil de tolérance autorisé par les concepteurs système, le calculateur perd la fonction système concernée et passe la main à un autre calculateur pour traiter la fonction perdue.

La Figure II.35 présente l'architecture matérielle simplifiée de ce principe de COM/MON

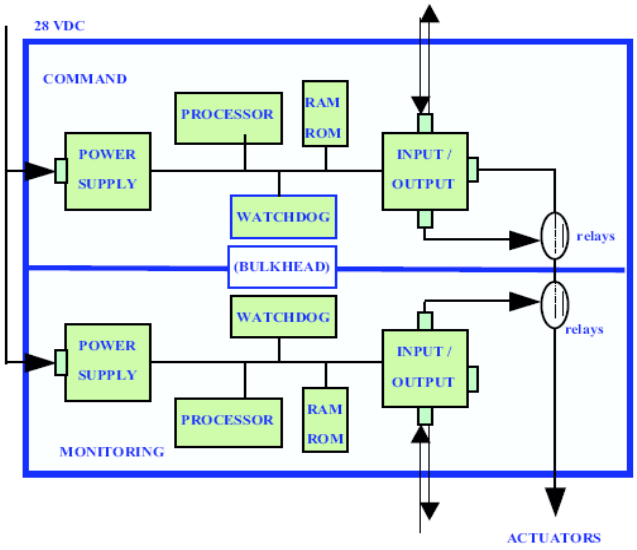


Figure II.35 Schéma simplifié du matériel d'un calculateur de commande de vol Airbus

Chez Boeing un des fondements de la tolérance aux fautes est l'utilisation de la solution de TMR (Triple Modular Redundancy) qui consiste à avoir 3 exemplaires pour chaque élément (Yeh Y.C, 1996). On distingue ainsi :

- Trois calculateurs numériques centraux PFC (Primary Flight Control), aussi appelés « chaîne », pour le calcul des lois.
- Quatre calculateurs analogiques ACE (Actuator Control Electronics) pour l'asservissement des actionneurs et la réalisation des conversions numériques-analogiques.

Les PFC sont identiques sur le plan matériel (tout comme les ACE), alors qu'au plan logiciel, de la diversification intervient au niveau des compilateurs pour la génération de l'applicatif pour les PFC, et elle est basée sur la programmation N-versions avec N=3.

La Figure II.36 présente l'architecture d'un PFC chez Boeing.

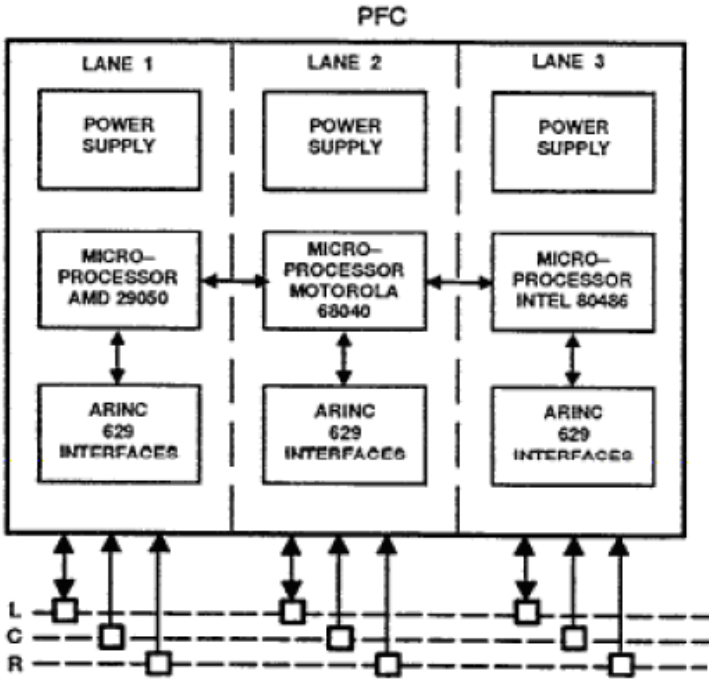


Figure II.36 Schéma simplifié du matériel d'un calculateur de commande de vol Boeing

II.5 SYNTHESE ET CONCLUSION DE L'ETAT DE L'ART

Nous avons présenté dans ce chapitre les caractéristiques des systèmes interactifs, leurs techniques de description et les principes de bases de la sûreté de fonctionnement.

Un système interactif est un système réactif avec en plus un utilisateur humain dans la boucle. Son développement est assez complexe, et pour y répondre les acteurs de l'interaction homme machine ont proposé des modèles d'architecture des systèmes interactifs afin de comprendre leur fonctionnement et distinguer ces différents composants. Parmi les modèles proposés, nous nous intéressons particulièrement au modèle d'architecture Arch. Ce modèle prend en compte les boîtes à outils et a été conçu dans l'objectif de faciliter le changement de technologie sur le système. C'est notamment un des objectifs visé dans le standard ARINC 661 qui fait l'objet de notre étude.

Différentes techniques de description des systèmes interactifs sont proposées dans la littérature. Nous les avons présentés dans ce chapitre en fonction des parties du système interactif qu'elle couvrait. Certaines se concentrent sur la description des interactions en entrée comme ICON, d'autres sur la description des interfaces utilisateurs comme UsiXML. Nous avons aussi présenté les automates à états finis et les réseaux de Petri qui sont utilisés pour la description du dialogue et du noyau fonctionnel. Les automates à état finis comme les Hierarchical state machines et les SwingStates ont comme avantages la facilité d'intégration dans une application existante et la modélisation de certains aspects d'un système. Cependant Ils ne permettent pas une modélisation d'un système interactif avec un grand nombre d'état. Une étude comparative des différentes techniques de description des systèmes interactifs peut être retrouvée dans (Navarre, et al., 2009).

Les ICO qui sont issus des réseaux de Petri haut niveau permettent une description plus large des différents composants des systèmes interactifs, à savoir la présentation (les entrées, les sorties) et le dialogue. Cette couverture dans la modélisation, l'aspect formel des réseaux de Petri point important pour les systèmes critiques et les différents travaux sur les systèmes interactifs critiques déjà effectués avec ICO nous ont conduits à choisir cette technique pour notre étude.

Nous ne pouvions parler de système interactif sans s'intéresser à l'utilisateur, c'est un composant important du système. Le bon fonctionnement de l'ensemble du système dépend de la capacité d'interaction entre lui et le cœur de calcul. Nous avons présenté ici les modèles de tâches de l'utilisateur, ceux-ci nous servirons pour l'évaluation de l'utilisabilité du système.

Nous nous intéressons dans nos travaux aux systèmes interactifs critiques. Le développement des systèmes critiques exige de suivre des processus rigoureux, en respectant des exigences de sûreté de fonctionnement. Nous avons présenté à la quatrième section de ce chapitre les principes de bases de la sûreté de fonctionnement en nous focalisant sur la tolérance aux fautes, dont le but est d'empêcher les défaillances malgré la présence ou l'occurrence de fautes.

L'interaction homme machine qui s'intéresse au développement des systèmes interactifs et la sûreté de fonctionnement sont étudiés généralement de façon indépendante. D'un côté l'interaction homme machine s'intéresse à développer des systèmes toujours plus innovants, avec des interactions nouvelles et qui sont adaptés à l'utilisateur. Et d'un autre côté des efforts croissants sont mis pour avoir des systèmes

plus robustes aux fautes au travers de nouvelles techniques de tolérance aux fautes. Ces mécanismes de tolérance aux fautes sont généralement implémentés qu'au cœur de calcul (noyau fonctionnel) comme sur les systèmes avioniques tels que nous l'avons vu dans les exemples ci-dessus. Pourtant la partie présentation est aussi sujette à des fautes.

Le chapitre suivant présente les approches que nous proposons pour aider au développement d'un système interactif intégrant les aspects sûreté de fonctionnement et utilisabilité.

Chapitre III. Approches proposées pour le développement de systèmes interactifs intégrant les aspects sûreté de fonctionnement et utilisabilité

Résumé du chapitre

Ce chapitre présente les approches que nous proposons pour le développement de systèmes interactifs intégrant les aspects sûreté de fonctionnement et utilisabilité.

La première section explique le contexte applicatif aux travaux de la thèse qui est le système d'affichage et de contrôle de cockpit d'avion civil. Nous présentons donc premièrement l'architecture matérielle et logicielle du système d'affichage et de contrôle, son principe de fonctionnement et les recommandations en termes de sûreté de fonctionnement et d'utilisabilité sur le système. Cette première section permet de situer le contexte de travail et de mieux appréhender les enjeux dans le développement d'un système interactif critique.

La seconde section décrit les approches proposées qui intègrent à la fois les principes de la sûreté de fonctionnement et de l'utilisabilité. Comme le titre de ce mémoire le précise, nous proposons une approche outillée. L'outil et la technique de description formelle ICO sur lesquels sont basés nos travaux ont été présentés dans le chapitre état de l'art. Dans ce chapitre nous nous concentrons à détailler les différentes approches et c'est dans les chapitres suivants que nous montrons leur application via l'utilisation de l'outil Petshop et de la technique de description formelle ICO.

Les approches proposées ne se limitent pas uniquement au domaine de l'aéronautique, mais peuvent être généralisables aux autres systèmes interactifs

III.1 CONTEXTE APPLICATIF AU TRAVAIL DE LA THESE

Comme nous l'avons vu dans le chapitre problématique, le cockpit désigne l'espace réservé aux pilotes, et contient toutes les commandes et tous les instruments nécessaires au pilotage de l'avion. Le système interactif auquel nous nous intéressons au sein du cockpit est le système d'affichage et de contrôle encore appelé CDS (*Control and Display System*). Le rôle du CDS est de fournir à l'équipage de vol les informations et les moyens d'interaction nécessaires pour assurer les tâches de pilotage et de navigation de l'avion, de gestion des systèmes à bord et de communication. Nous parlons ici de cockpit interactif en référence au CDS qui est le principal système interactif au sein du cockpit.

III.1.1 Architecture matérielle et logicielle du CDS

III.1.1.1 *Architecture générale*

L'architecture matérielle et logicielle du CDS diffère selon le type d'avion. Sur les récents avions gros porteur civil, il est constitué en général d'un ensemble d'unités d'affichages appelées Display Unit (DU) et d'un ensemble d'unité de contrôle appelé KCCU (Keyboard and Cursor Control Unit), que nous pouvons observer sur les Figure III.1 et Figure III.2.

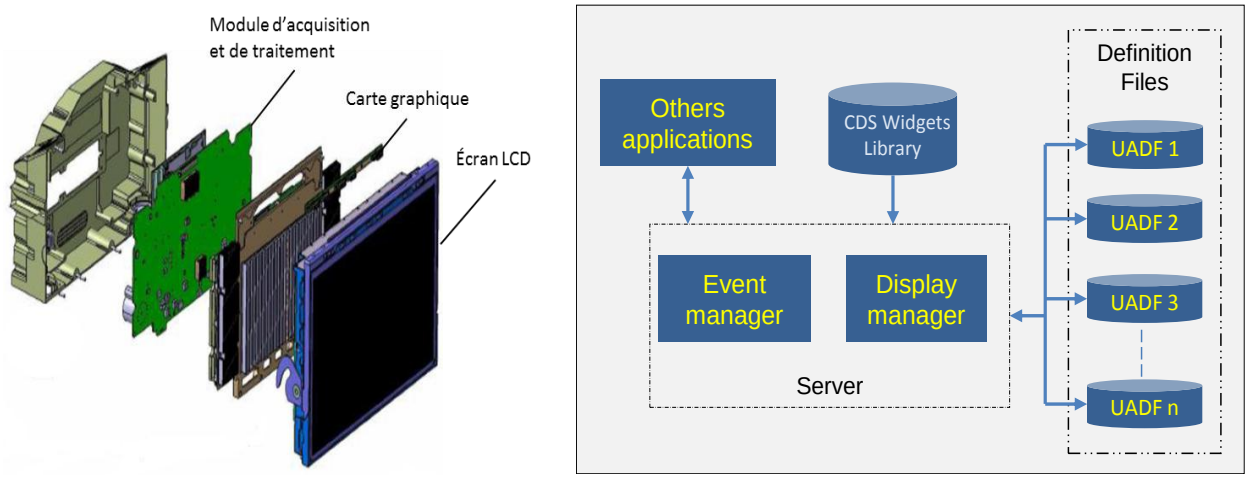
La Figure III.1 montre un exemple de KCCU d'un avion Airbus. Ce KCCU est composé d'un clavier et d'un dispositif de pointage de type trackball, sur le Boeing 787 c'est plutôt un touchpad qui est utilisé. Le KCCU permet à l'équipage d'interagir sur les écrans. Il est apparu au sein du CDS avec l'introduction du standard ARINC 661.

Figure III.1 Exemple d'un KCCU

La partie a) de la Figure III.2 présente l'architecture matérielle d'une DU. Une DU est un calculateur composé d'écran de type LCD, d'une carte graphique et d'un module d'acquisition et de traitement. La DU a pour rôle de traiter les informations reçues des calculateurs des systèmes avions, de les afficher à l'équipage, et aussi de gérer les actions de l'utilisateur sur le KCCU et de les transmettre aux systèmes avions.

La partie b) de la Figure III.2 présente l'architecture logicielle d'une DU traitant les données de type ARINC 661. Elle est composée de :

- Une bibliothèque de widgets qui est l'ensemble des widgets utilisés par toutes les UA qui communiquent avec le CDS et provenant du standard ARINC 661.
- Des UADF (User Application Definition File) des différentes applications clientes des systèmes avions (UA). Les UADF sont des fichiers dans lesquels chaque UA décrit les caractéristiques de leur interface graphique, c'est-à-dire leur layer et leur widgets ainsi que leur structure hiérarchique.
- D'un serveur qui regroupe les logiciels de gestion des évènements des périphériques d'entrées (*Event manager*) et de gestion de l'affichage (*Display manager*).
- D'autres applications pour la gestion du comportement interne au CDS (gestion des configurations, gestion des drivers etc.)



a) Architecture matérielle DU *b) Architecture logicielle DU*

Figure III.2 Architecture matérielle et logicielle d'une DU

La Figure III.3 présente l'architecture générale de communication pour l'échange de données de type ARINC 661 entre le CDS et l'UA.

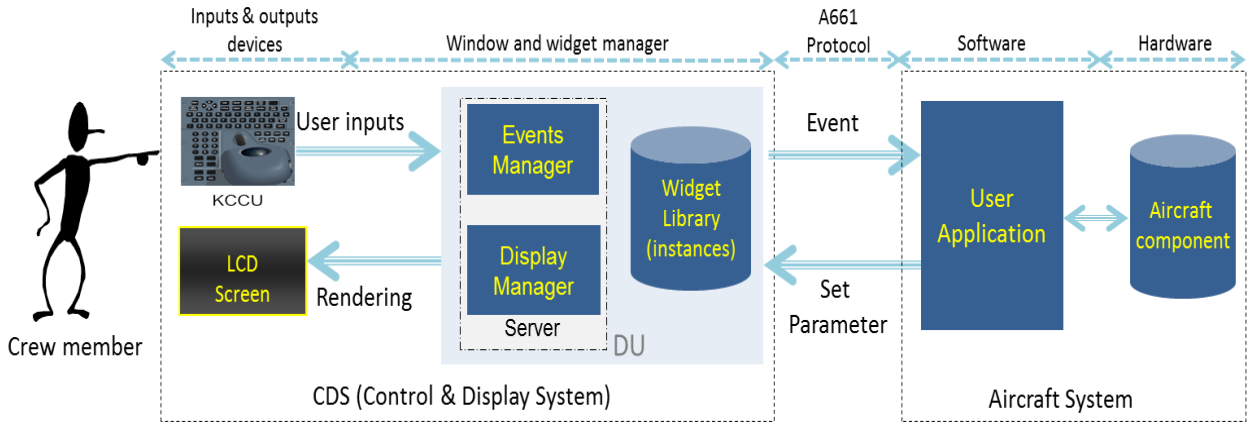


Figure III.3 Architecture de communication ARINC 661 DU/UA

L'équipage de vol (*Crew Member*) interagit avec le système avion via les périphériques d'entrées/ sorties que sont le KCCU et la DU.

Le diagramme de séquence à la Figure III.4 présente le flot d'évènement allant de l'utilisateur à l'UA. Lors d'une action de l'utilisateur (exemple un clic), le serveur réceptionne l'évènement de clic et identifie le widget sur lequel il y a eu l'action de clic par sa position sur l'interface. L'évènement est ensuite transmit au widget qui selon son état interne va traiter l'action, et envoyer une notification sous forme d'*Event* à l'UA. La réception de l'*Event* au sein de l'UA peut conduire à déclencher un traitement dans les calculateurs des systèmes avions ou à mettre à jour son interface graphique.

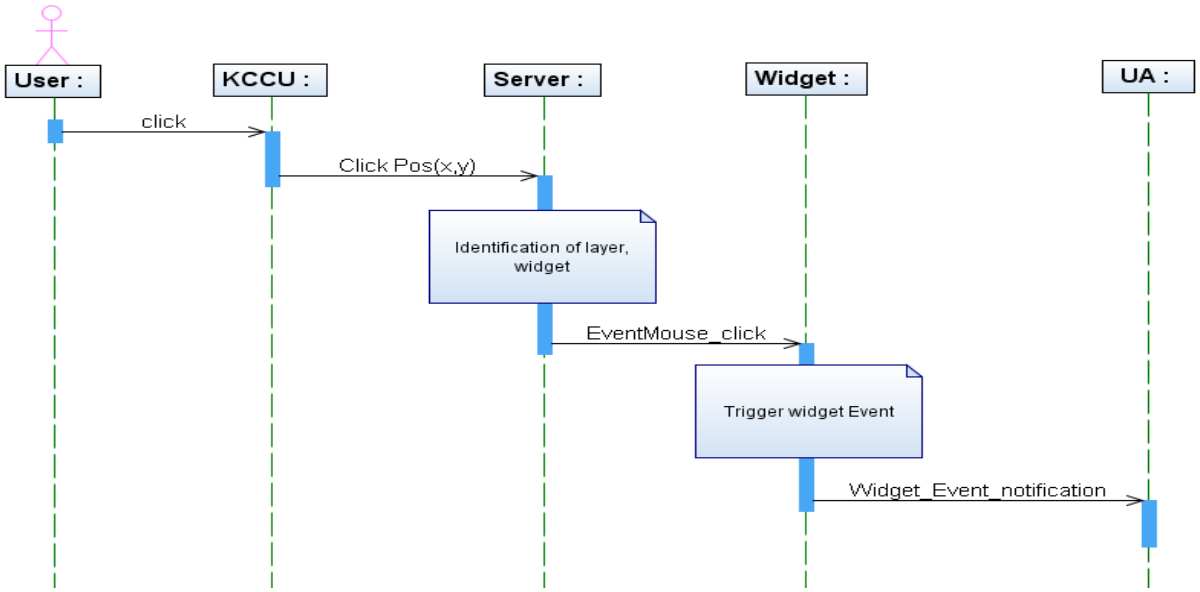


Figure III.4 Diagramme de séquence du flot d'évènements allant du CDS à UA

Le diagramme de séquence à la Figure III.5 présente le flot de données allant de l'UA à l'utilisateur. En réponse à une notification d'évènement ou suite à l'évolution de l'état interne du système avion, l'UA peut effectuer des mises à jour sur son interface graphique en changeant les paramètres de ses widgets (exemple visibilité, texte affiché etc.), ceci en envoyant des commandes de type s*etparameter* à la DU. Les mises à jour sont ensuite observables par l'utilisateur (pilote) sur les écrans.

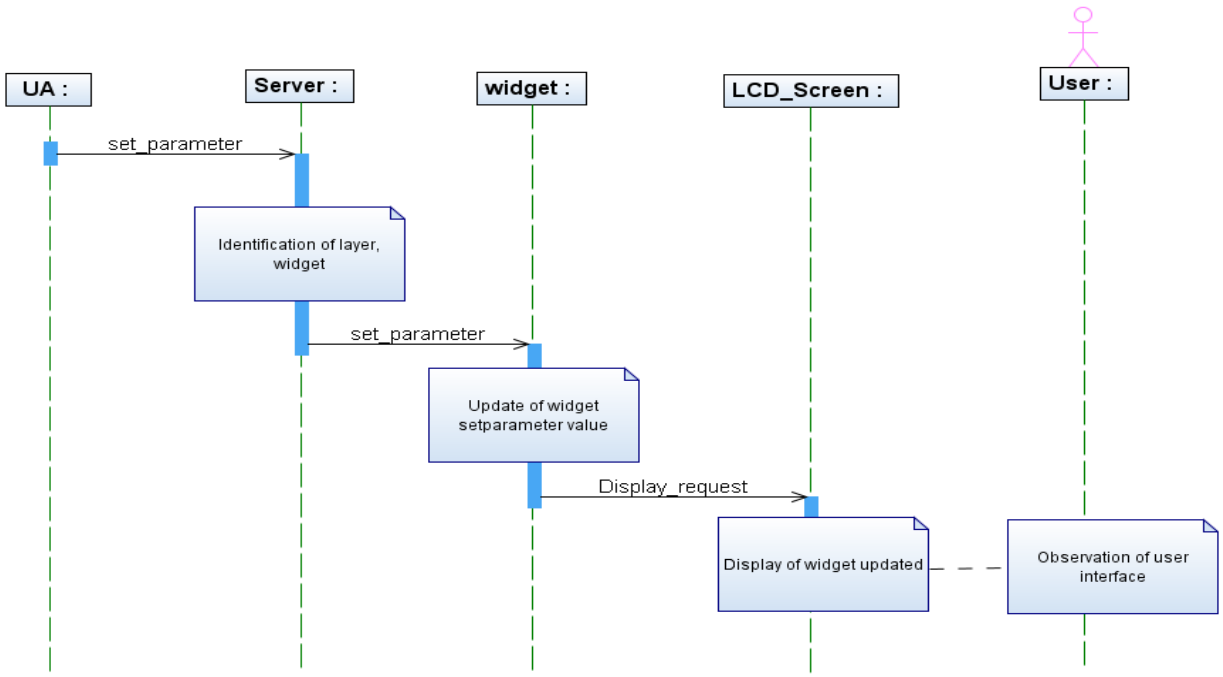


Figure III.5 Diagramme de séquence du flot de données allant de l'UA au CDS

III.1.1.2 *Exemple de l'architecture du CDS de l'Airbus A380*

La Figure III.6 présente la composition du CDS sur l'Airbus A380. Il est composé de:

- 8 DU dont 3 du côté gauche (L1, L2, L3) pour le capitaine, 3 du côté droit (R1, R2, R3) pour le co-pilote et deux centraux (C1, C2), partagés par le capitaine et le co-pilote.
- 2 KCCU, le KCCU gauche utilisé par le capitaine et le KCCU droit utilisé par le co-pilote

Les DU et les KCCU sont connectés par un réseau interne de type CAN, ce réseau CAN est redondé et ségrégué selon les côtés gauche et droite du CDS. On distingue ainsi les bus CAN 1.1 et CAN 1.2 pour la connexion des DU et du KCCU côté capitaine et les bus CAN 2.1 et CAN2.2 pour la connexion des DU et du KCCU côté co-pilote.

Le CDS communique avec les autres systèmes avion via le réseau AFDX (AFDX, 2000), pour recevoir les données à afficher sur ces systèmes et pour notifier des actions utilisateurs sur l'interface graphique de ces systèmes.

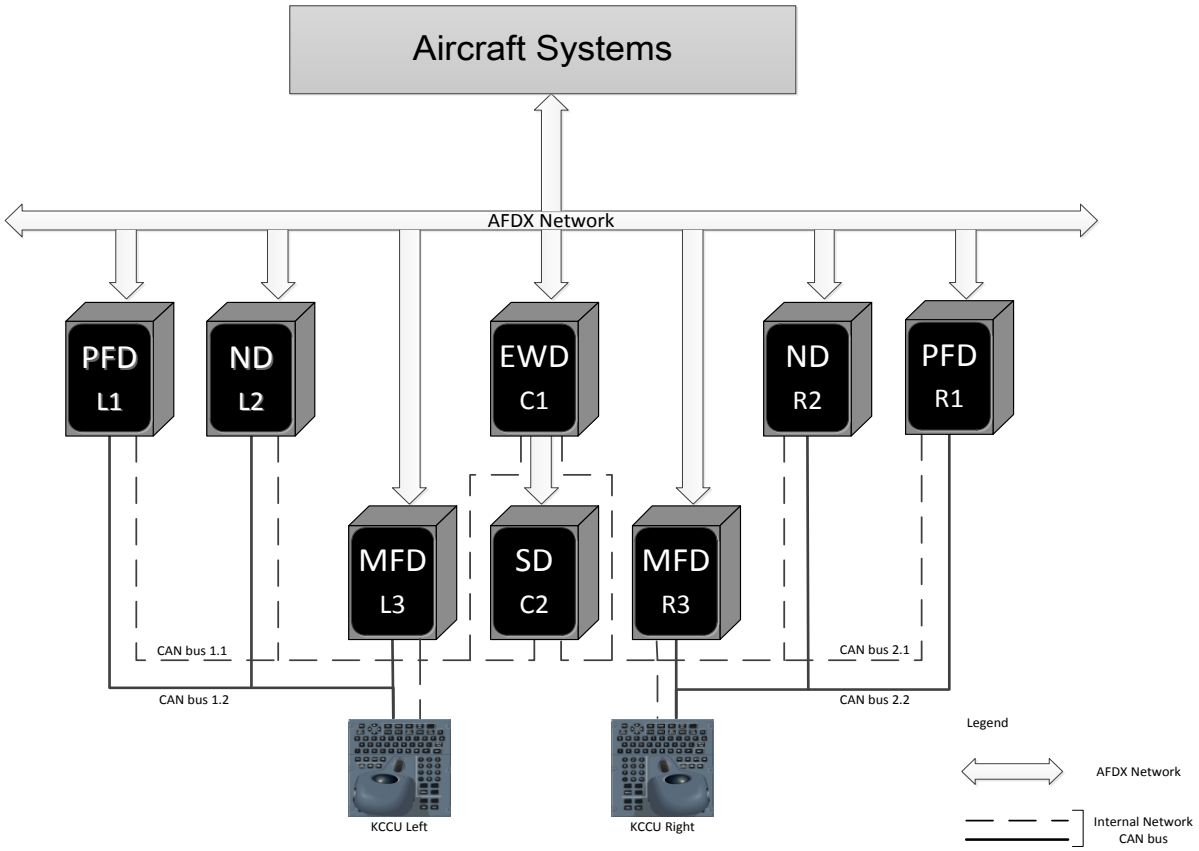


Figure III.6 Architecture matérielle du CDS A380

Les DU possèdent différentes applications des systèmes avions, et chacune des applications traitent un ensemble d'information spécifique à afficher. Les informations à afficher sont regroupées sous des formats. On peut voir l'ensemble de ces formats sur la Figure III.6 qui sont :

- Le format PFD (*Primary Flight Displays*), où sont traités les paramètres primaires de vol nécessaires pour le contrôle de l'avion à court terme tels que l'altitude et la vitesse.
- Le format ND (*Navigation Displays*), où sont traités les paramètres de navigation au cours de vol tels que la météo, le plan de vol, les autres avions dans le champ de vol.
- Le format EWD (*Engine Warning Display*) où sont traités les paramètres moteurs, les messages d'alerte.
- Le format SD (*System Display*) où sont traités les paramètres d'état des systèmes avions (fuel, air, hydraulique …)
- Le format MFD (*Multi Function Display*) où sont traités différentes fonctions de l'avion, tels que le contrôle du trafic aérien, le système de surveillance, la gestion du plan de vol.

Les formats représentent en réalité l'interface graphique d'une ou plusieurs applications des systèmes de l'avion. Ils sont constitués d'objets graphiques divers et variés pour représenter au mieux l'état du système (boutons, texte, zone d'édition, image etc.)

Le standard ARINC 661 propose notamment une bibliothèque de widgets qui sont utilisés pour la conception des applications des systèmes avions. Un widget est ici plus qu'un objet graphique, c'est un

mini composant logiciel pouvant avoir une représentation graphique, il comporte des états internes, et est capable de réagir à des stimuli externes.

III.1.2 La description des widgets dans le standard ARINC 661

La bibliothèque du standard ARINC 661 contient à ce jour une soixantaine de widgets. La description donnée par le standard pour chaque widget se résume aux parties suivantes :

- Description du widget et de ses fonctionnalités.
- Table des paramètres (*Parameter Table*) qui décrit les différents paramètres du widget. (exemple Tableau III.1).
- Table d'instanciation (*Creation Structure Table*) qui liste les différents paramètres permettant d'instancier le widget. (exemple Tableau III.2)
- Table(s) d'évènement (*Evenement Structure Table(s*)) qui décrit (vent) la structure des différents évènements que le widget peut envoyer, (exemple Tableau III.3).
- Table des paramètres modifiables à l'exécution (*Runtime Modifiable Parameters Table*). (exemple Tableau III.4).

Nous allons maintenant présenter la description d'un widget de type PushButton tel que cela est décrit dans le standard ARINC 661.

Un pushbutton est définit comme étant un bouton-poussoir permettant à l'équipage de lancer une action. Il a une représentation graphique, est interactif et peut afficher du texte.

Le Tableau III.1 présente la table des paramètres du PushButton. Cette table est divisée en trois colonnes. La première colonne correspond au nom des paramètres, elle est divisée en paramètres communs à l'ensemble des widgets et en paramètre spécifiques au widget ou à un ensemble de widget. La deuxième colonne appelée « *change* » donne des informations sur le fait qu'un paramètre est modifiable ou non. Ainsi les paramètres dont la valeur « *change* » est égale à D sont ceux qui sont modifiables uniquement lors de la phase définition, DR pour les paramètres modifiables à la phase de définition et d'exécution et R pour ceux modifiables uniquement lors de la phase d'exécution. La dernière colonne donne une description du paramètre.

Parameters	Change	Description
Commonly used parameters		
WidgetType	D	A661_PUSH_BUTTON
WidgetIdent	D	Unique identifier of the widget
ParentIdent	D	Identifier of the immediate container of the widget
Visible	DR	Visibility of the widget
Enable	DR	Ability of the widget to be activated
StyleSet	DR	Reference to predefined graphical characteristics inside CDS
PosX	D	The X position of the widget reference point
PosY	D	The Y position of the widget reference point
SizeX	D	The X dimension size (width) of the widget
SizeY	D	The Y dimension size (height) of the widget
NextFocusedWidget	D	Widget ident of next widget to be focused upon crew member validation
AutomaticFocusMotion	D	Automatic motion of the focus on widget specified in NextFocusedWidget parameter
Specific parameters		
Alignment	D	Alignment of the text within the label area of the widget LEFT RIGHT CENTER
LabelString	DR	String of the PushButton
MaxStringLength	D	Maximum length of the label text
EntryValidation	**R**	**Indicator notifying the CDS that the UA has completed processing the entry or selection event. This flag also indicates the results of that processing.** **A661_FALSE** **A661_TRUE**

Tableau III.1 Paramètres pushbutton

La table d'instanciation du PushButton est représentée au Tableau III.2. Dans cette table, des informations supplémentaires sont données sur les paramètres du widget (le type, la taille et les valeurs possibles des paramètres).

CreateParameterBuffer	Type	Size (bits)	Value/Range When Necessary
WidgetType	ushort	16	A661_PUSH_BUTTON
WidgetIdent	ushort	16	
ParentIdent	ushort	16	
Enable	uchar	8	A661_FALSE A661_TRUE **A661_TRUE_WITH_VALIDATION**
Visible	uchar	8	A661_FALSE A661_TRUE
PosX	long	32	
PosY	long	32	
SizeX	ulong	32	
SizeY	ulong	32	
StyleSet	ushort	16	
NextFocusedWidget	ushort	16	
MaxStringLength	ushort	16	
AutomaticFocusMotion	uchar	8	A661_FALSE A661_TRUE
Alignment	uchar	8	A661_LEFT A661_RIGHT A661_CENTER
LabelString	string	8 * string length + Pad	Followed by zero, one, two or three NULL character(s) to be 32 bits aligned

Tableau III.2: Table d'instanciation du PushButton

Le pushbutton envoie un seul évènement à l'UA, et cet évènement est défini selon le Tableau III.3.

EventStructure	Type	Size (bits)	Value/Description
EventIdent	ushort	16	A661_EVT_SELECTION

Tableau III.3 Structure de l'Event du pushbutton

Les paramètres que l'UA peut modifier en exécution sur le widget PushButton sont représentés au Tableau III.4.

Name of the Parameter to Set	Type	Size (bits)	ParameterIdent Used in the ParameterStructure	Type of Structure Used (Refer to Section 4.5.4.5)
Enable	uchar	8	A661_ENABLE	A661_ParameterStructure_1Byte
Visible	uchar	8	A661_VISIBLE	A661_ParameterStructure_1Byte
LabelString	string	{32}+	A661_STRING	A661_ParameterStructure_String
StyleSet	ushort	16	A661_STYLE_SET	A661_ParameterStructure_2Bytes
EntryValidation	uchar	8	A661_ENTRY_VALID	A661_ParameterStructure_1Byte

Tableau III.4 Paramètres modifiables en exécution du PushButton

L'ARINC 661 ne décrit pas le rendu graphique (look) ni le comportement des widgets (feel), cette partie est propre à chaque avionneur. A Airbus ceci est décrit au travers d'un document nommé « Look and Feel ».

Le comportement du widget est généralement décrit textuellement. D'autres widgets du standard ARINC 661 sont plus complexes, ils gèrent un nombre plus important de paramètres que le Pushbutton. Réaliser une description textuelle de leur comportement conduit facilement à des interprétations très différentes.

III.1.3 Sûreté de fonctionnement et certification sur les systèmes interactifs de cockpit

Avant de pouvoir voler, chaque avion doit obtenir son certificat de navigabilité qui garantit la conformité de l'appareil aux normes de sécurité internationales. Ainsi le développement de l'avion en général et de ces systèmes en particulier est soumis à des normes et réglementations définit par des autorités de certification (EASA, 2008). La norme AMC 25-11 fournit les moyens de conformité à la certification pour les systèmes d'affichage et de contrôle de cockpit d'avion gros porteur. Dans cette norme, une liste de défaillances au niveau du CDS est définie, le Tableau III.5 présente quelques exemples de ces défaillances.

AMC 25-11 Regulation		Severity	Failure rate	DAL
Integrity	Misleading display of any required engine indications for more than one engine	CAT + *fail safe*	]…, 10^{-9}]	A
	Misleading Display of primary flight parameter (attitude, airspeed …)	HAZ	] 10^{-9},10^{-7}]	B
	Undetected erroneous input from the display system as a control	Depends on system being controlled		

Tableau III.5 Exemple de scenario de défaillances sur le système CDS

La première ligne du tableau précise qu'un affichage erroné de toute indication moteur pour plus d'un moteur a un niveau de sévérité de type CAT (catastrophique) donc peut avoir des conséquences très dangereuses sur le pilotage de l'avion. La probabilité d'occurrence de cette défaillance doit être inférieure à 10^{-9} et la fonction réalisant l'affichage de ce paramètre doit avoir un niveau de développement de niveau A (DAL A), ce qui implique une couverture de test exhaustive. De plus pour les défaillances de niveau de sévérité catastrophique, le critère *Fail Safe* est aussi requis. Le critère *Fail Safe* précise qu'aucune panne simple ne doit conduire à une défaillance catastrophique. L'atteinte du niveau de probabilité 10^{-9} et du critère *Fail Safe* se fait généralement par l'introduction de techniques de tolérance aux fautes.

Comme nous l'avons vu dans le chapitre problématique, le cockpit a évolué ces dernières années, en particulier le CDS. Les DU du CDS étaient avant de simple afficheurs de données et sont devenus des systèmes interactifs avec l'introduction du standard l'ARINC 661. Sur ces anciennes DU, un processus de développement a été mis en place afin d'atteindre le niveau de DAL requis et un mécanisme de tolérance aux fautes de type COM/MON (vu dans le chapitre état de l'art) permettait d'atteindre le niveau de probabilité et le critère *Fail Safe* requis pour l'intégrité de l'affichage des données.

Avec l'introduction de l'ARINC 661 donc d'une capacité d'interaction logicielle avec les écrans, on se retrouve face à un comportement complètement différent des applications des DU. Ces applications sont maintenant interactives, elles ont donc la capacité de recevoir les entrées de l'utilisateur qui peuvent arriver à n'importe quel moment et doivent fournir à cet utilisateur les informations nécessaires à la réalisation de ces tâches. Ce qui n'est pas le cas des applications réalisant un simple affichage des données.

Ce nouveau fonctionnement du CDS implique de revoir le processus de développement qui permettait d'atteindre un niveau de DAL requis pour les fonctions critiques, mais aussi les mécanismes de tolérance aux fautes pour atteindre les niveaux de probabilité demandés. Ceci n'étant pas encore réalisé, la possibilité que le pilote ait d'interagir avec les écrans en utilisant le clavier et le dispositif de pointage est aujourd'hui limitée à des fonctions non critiques, donc ayant un impact inférieur ou égal à Majeur selon le tableau de classification du niveau de criticité de la DO 178B.

La Figure III.7 montre la configuration actuelle d'un CDS A380 où les zones non interactives pour les fonctions critiques sont celles hachurées, l'utilisation du curseur y est interdite. Les autres zones en motif plein sont interactives et attribuées aux fonctions dites non critiques.

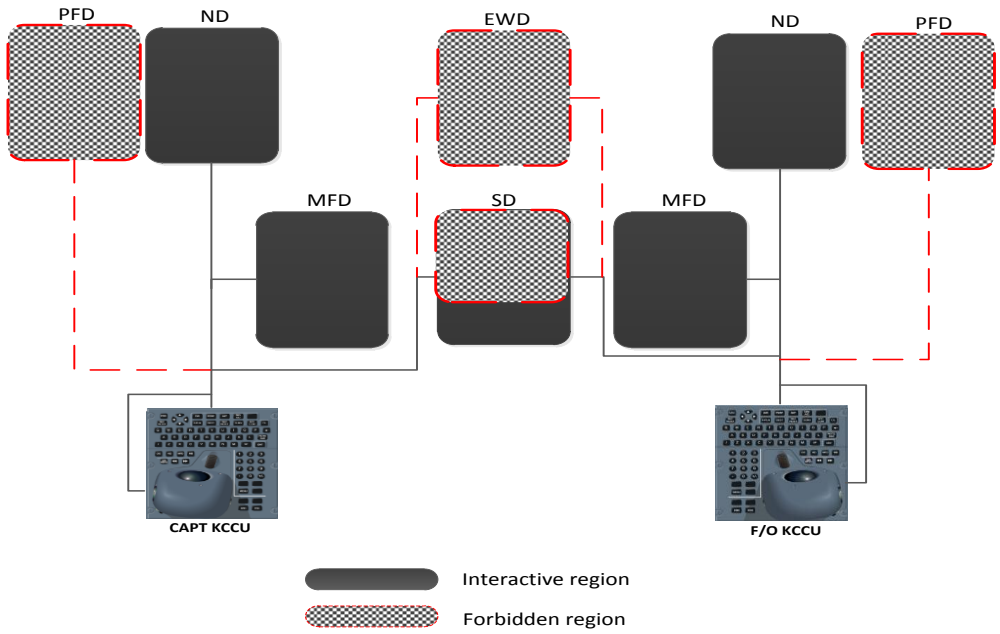


Figure III.7 Zone interactive et non interactive sur le CDS A380

Aujourd'hui les fonctions critiques du CDS sont des fonctions permettant l'affichage des paramètres primaires de vol (PFD), des paramètres moteurs (EWD) et l'affichage des systèmes (fuel, air, hydraulique etc.) (SD). Les parties du CDS gérant ces fonctions critiques ont un fonctionnement de type réactif (donc sans utilisateur dans la boucle). Ils reçoivent des informations venant soit des capteurs de l'avion, soit des calculateurs de l'avion et les affichent à l'écran. Les sorties affichées sont prévisibles en fonction des entrées reçues. Ils sont déterministes et suivent un processus de développement rigoureux en accord avec les normes de développement des systèmes avioniques critiques. L'ensemble des contrôles et des commandes sur ces fonctions est réalisé au travers des commandes physiques (ex bouton poussoir) localisées par exemple sur le panneau plafond dans le cockpit.

Pour étendre l'interactivité aux fonctions critiques, il faut répondre aux exigences de sûreté de fonctionnement des autorités de certification. Il s'agit de suivre les différents processus de développement de système critiques, permettant de s'assurer d'une couverture de test exhaustive donc d'avoir un système logiciel ou matériel sans défaut. Et aussi d'intégrer les mécanismes de tolérance aux fautes, en particulier de la redondance permettant d'atteindre les probabilités de fiabilité souhaitées.

III.1.4 L'utilisabilité du CDS

Le cockpit est souvent considéré comme un environnement matériel (hardware) permettant à l'équipage de réaliser les missions qui leur sont dévolues. L'équipage est couramment constitué de 2 opérateurs : le pilote (*captain*) et le co-pilote (*first officer*). Toutefois, les évolutions successives au niveau du cockpit ont laissé une part croissante au logiciel. Le logiciel prend ainsi une grande ampleur dans les cockpits civils avec la mise en œuvre récente du standard ARINC 661 présenté dans la section I.2.

Avec l'intégration matériel/logiciel, la conception des logiciels de pilotage et de gestion de vol se rapproche très fortement des contraintes de conception de systèmes interactifs telle que traitée dans le

domaine de l'Interaction Homme-Machine. Ainsi, si le poste de pilotage (dans son ensemble) peut être considéré comme un système de commande et contrôle complexe, sa conception relève du domaine de l'ergonomie avec bien évidemment des contraintes propres au fait que ce poste de commande et contrôle soir embarqué. La conception de cockpits a fait l'objet de nombreuses publications en particulier par la NASA et Boeing (Abbott, 2001) et (Palmer , et al., 1995).

L'apparition de la partie logicielle dans le cockpit pose de nouveaux problèmes de conception très différents de ceux mis en avant dans les travaux précédents qui se focalisaient sur l'interaction du pilote avec les composants matériel. La conception des interfaces homme machine déployées dans le cockpit devient maintenant très proches des problèmes « classiques » en interaction homme-machine où les problèmes de l'utilisabilité du logiciel sont étudiés à la fois au niveau théorique (Nielsen, 1993) mais aussi au niveau pratique (Hartmann, et al., 2008). Diverses études ont aussi montré que les méthodes visant à prendre en compte l'utilisabilité des systèmes doivent être spécialisées en fonction des différents domaines d'application (Blandford, et al., 2004).

La conception des applications interactive intégrées dans le cockpit doit correspondre à la fois aux activités à mener par l'équipage et aux contraintes liées à l'interaction dans un environnement spécifique. Selon la définition de la norme ISO 9241 (ISO9241-11, 1998) le facteur utilisabilité peut être décomposé en 3 sous-facteurs : *efficiency* (performance), *effectiveness* (efficacité) et satisfaction des utilisateurs.

La satisfaction utilisateur peut être évaluée au moyen de questionnaires comme le SUS (System Usability Scale) (Brooke, 1996) et n'est pas très important dans le contexte d'une interaction avec un système critique. En effet, la sécurité (safety) des passagers et de l'équipe est prépondérante sur les aspects confort et satisfaction. . Pour cette raison, cet aspect n'est pas abordé dans le reste de cette thèse.

La performance a un impact très lourd sur l'utilisabilité d'un cockpit dans la mesure où certaines opérations doivent être exécutées sous forte contraintes temporelles en fonction de la phase de vol (e.g. communication avec le contrôle aérien, atterrissage, gestion des alarmes). La performance des techniques d'interactions et des applications interactives dans le cockpit doit être étudiée avec un soin tout particulier sous peine de dégrader les performances de l'équipage et ce, malgré l'introduction de nouvelles technologies ayant pour objectif d'accroitre la bande passante entre les utilisateurs et le système interactif. Un facteur important de la performance de l'équipage est liée à la formation qui va leur être donnée et ce de façon relativement indépendante de la structure des applications ou des techniques d'interaction utilisées. Pour cette raison nous n'aborderons que de façon quantitative cet aspect performance dans la thèse alors que l'aspect quantitatif (temps d'exécution des tâches) reste en dehors du périmètre de la thèse. Des travaux proches des nôtres (Martinie De Almeida, et al., 2011) ont été réalisés sur ces aspects formation et sur les aspects performance (Palanque, et al., 2011) mais, malgré la proximité des travaux, une grande quantité de travail reste à faire pour les coupler aux contributions de cette thèse.

L'efficacité concerne la possibilité offerte à l'opérateur de réaliser ses tâches et d'atteindre ses buts au moyen du système interactif. Ainsi un système offrant uniquement un sous-ensemble des tâches pertinentes et un sous-ensemble des buts nécessaire pour mener à bien les missions sera jugé inefficace. L'efficacité a un impact direct sur la performance dans la mesure où un système inefficace va nécessiter de la part des opérateurs de mettre en place des méthodes détournées pour atteindre leurs buts. L'approche proposée dans cette thèse prend en compte ces aspects efficacité au moyen de modèles de tâches

décrivant les activités à réaliser par les opérateurs pour atteindre leurs buts. Toutefois, nous considérons ici uniquement les aspects élémentaires des tâches tels que la lecture d'une valeur sur un écran, la saisie d'une valeur ou le déclenchement d'une commande par activation d'un bouton de l'application interactive. La prise en compte de tâches de plus haut niveau liées aux buts des opérateurs aurait nécessité la prise en compte de données liées aux opérations des cockpits. Ceci aussi est situé en dehors du périmètre de cette thèse mais les contributions de la thèse peuvent facilement être étendues à ce genre de considérations comme nous le montrons dans le cadre de l'étude de cas du Chapitre I.

L'erreur humaine et les raisons de son occurrence ont été étudiées depuis de nombreuses années avec en particulier la contribution séminale de James Reason (Reason, 1993). La décomposition en « slips », « lapses » et « mistakes » positionne ces erreurs à différent niveaux d'abstraction (voir (Johnson, 2011) pour une présentation adaptée au domaine aéronautique). Dans la mesure où nous nous sommes uniquement intéressés aux tâches élémentaires nous avons laissé de côté la prise en compte des erreurs humaines et avons concentré nos efforts sur leur contrepartie côté machine en mettant avant tout en avant la fiabilité et la tolérance aux fautes du système interactif et en pointant les connections vers le composant humain du système interactif.

III.1.5 Hypothèses et périmètre du travail de cette thèse

Quelques précisions importantes sont à faire sur le périmètre de nos travaux.

La capacité de contrôle sur le CDS (interaction avec clavier, dispositif de pointage) étant nouvelle, le niveau de criticité n'a pas encore été fixé dans les normes avioniques (dernière ligne du Tableau III.5). Nous prenons pour hypothèse de travail que la criticité du contrôle pourra atteindre un niveau catastrophique. En effet si l'on suppose que dans le futur l'on veut réaliser le contrôle des moteurs via les écrans, leur criticité sera tout au moins égale à celle de l'affichage des paramètres moteurs.

Dans le cadre de cette étude, on se donne pour objectifs de proposer une méthode et des techniques de tolérance aux fautes permettant de tenir des exigences qualitatives et quantitatives des fonctions interactives ayant un niveau de criticité pouvant atteindre le niveau catastrophique.

Le périmètre de nos travaux se limite principalement au CDS et particulièrement aux applications développées sur la base du standard ARINC 661 et embarquées au sein du CDS. L'UA qui gère toute la partie fonctionnelle de l'interface du CDS et qui est embarqué au sein du calculateur des systèmes de l'avion ne sera étudiée que dans le but de mettre en évidence son impact sur les applications du CDS.

Le serveur est un composant logiciel embarqué dans le CDS et gère une partie des échanges de données en ARINC 611. Le périmètre des travaux de la thèse ne couvre pas dans sa globalité la sûreté de fonctionnement du CDS. Nous nous somme concentré en particulier sur les widgets qui sont des éléments fondamentaux de la réalisation du CDS.

Les systèmes avions critiques implémentent des architectures tolérantes aux fautes, c'est le cas par exemple des commandes de vol qui ont une architecture intégrant des principes de ségrégation, redondance et diversification. Nous faisons donc l'hypothèse que les données provenant de ces systèmes sont fiables.

Le fonctionnement de l'ARINC 661 est indépendant du type de bus utilisé, on ne s'intéresse donc pas à la fiabilité du bus ou du réseau de communication entre le CDS et l'UA. Nous prenons pour hypothèse qu'ils sont fiables. D'autres études étant menées dans ce domaine (Peterson, et al., 1972)

L'utilisabilité étant une notion importante dans le développement des systèmes interactifs, nous nous intéresserons à l'impact que peut avoir les différents principes proposés sur l'efficacité des tâches de l'utilisateur. Les notions d'erreur humaine ou d'ergonomie de l'interface ont un impact très important mais sont en dehors du périmètre de cette thèse.

La contribution principale que nous voulons apporter au travers de nos travaux est de proposer des approches qui aideront au développement d'un système interactif critique tel que le système d'affichage et de contrôle des cockpits d'avion civil.

III.2 PRINCIPES DE LA METHODE PROPOSEE

Nous avons vu que l'architecture d'un cockpit interactif est assez complexe. Il possède différents calculateurs, affichant des données bien particulières. Au travers des enjeux relevés dans la section précédente, on peut relever quelques points importants pour le développement d'un système interactif critique. Le premier est d'avoir un développement intégrant les principes de sûreté de fonctionnement, le second est d'avoir un système adapté à l'utilisateur. Ces deux points sont détaillés ci-après.

Au niveau de la sûreté de fonctionnement, deux objectifs sont à considérer. Le premier objectif est de concevoir un système interactif parfait, i.e. sans faute. Mais aujourd'hui, la description du comportement des systèmes interactifs utilisant le standard ARINC 661 est réalisée principalement textuellement. L'interface est ensuite codée manuellement en fonction de la spécification textuelle. L'utilisation d'une spécification textuelle est peu précise et peut conduire à des ambigüités de compréhension donc à des erreurs de conception. On peut aussi noter que les fautes de conception sont à l'origine de la majorité des défaillances rencontrée lors de l'exploitation des logiciels (Endres, 1975). Dans les environnements critiques comme l'avionique, ces fautes de conception sont éliminées via l'utilisation de méthode formelle et un processus de développement rigoureux (DO-178B, 1992). Ce premier objectif demande aussi d'avoir une couverture de test exhaustive. Il faudrait pour cela tester le fonctionnement de chaque widget, mais aussi de toute l'interface graphique du CDS, donc s'assurer que toutes les actions possibles de l'utilisateur sur l'interface ont été testées. Ceci est presqu'impossible car les interfaces sont complexes, certains formats du CDS possèdent plus d'une centaine de widgets et ont diverses configurations. C'est pourquoi il est important d'utiliser des formalismes pour garantir des propriétés et s'assurer d'une bonne couverture des tests.

Avoir un système sans défaut n'est pas toujours réaliste, tout du point de vue des fautes de conception que des fautes physiques en opération. Le second objectif de sûreté de fonctionnement à considérer est le quantitatif au travers des probabilités de fiabilité à atteindre. Ces probabilités s'obtiennent généralement par l'introduction des techniques de tolérance aux fautes. Les techniques de tolérance aux fautes permettent aussi de traiter les fautes résiduelles de conception, les fautes matérielles et les fautes venant de l'environnement du système. Elles sont conçues pour traiter des erreurs possibles et identifiées sur le système avant qu'elles ne conduisent à des défaillances. Les défaillances définies à ce jour par la réglementation cible des fonctions précises du CDS (affichage des paramètres moteurs, des paramètres primaires de vol etc.). Concevoir des mécanismes de sûreté de fonctionnement spécifiques à ces fonctions conduira vite à des limites, car de nouvelles fonctions sont constamment intégrées dans le CDS. C'est pourquoi nous voulons être le plus générique possible dans l'approche proposée, nous appuyer sur des composants de bases et non sur une fonction spécifique. L'introduction de l'ARINC 661 permet justement de voir le CDS comme une ressource de services d'affichage et de contrôle, en laissant l'aspect fonctionnel aux applications avions plus aptes à les maitriser.

Au niveau de l'utilisabilité, des études sont menées lors de la conception et des tests utilisateurs sont réalisés pour s'assurer que le système est adapté à l'utilisateur. On déplore néanmoins une fracture entre la sûreté de fonctionnement et l'utilisabilité, qui sont réalisées de façon indépendante.

L'introduction de l'interactivité dans le cockpit conduit à un dialogue plus élevé entre le système et l'utilisateur. Il devient donc important d'avoir une analyse dès la conception du système interactif de l'impact que peut avoir l'introduction de mécanisme de sûreté sur l'utilisabilité. En effet la sûreté de l'ensemble du système interactif doit intégrer à la fois la sûreté du système matériel et logiciel, mais aussi la charge de travail de l'utilisateur qui se traduira par sa facilité à prendre en main le système le cas échéant. Un système sûr peut être inutile ou dangereux si aucun utilisateur ne peut s'en servir, ou s'il est trop complexe pour l'utilisateur final. D'autre part, la sûreté du système peut conditionner son utilisabilité en rajoutant un degré de complexité au niveau de ses tâches.

Nous proposons plusieurs approches pouvant contribuer au développement de ce système interactif critique (Figure III.8) qui sont : la conception zéro défaut, la tolérance aux fautes et l'analyse des modèles de tâche afin d'expliciter l'impact de l'architecture du système sur l'utilisabilité.

Ces différentes approches ne sont pas appliquer dans ce mémoire sur des systèmes réels, nous utilisons principalement une approche à base de modèle en utilisant le formalisme ICO. Ces modèles permettent de spécifier, concevoir et valider les principes en vue d'une implémentation future.

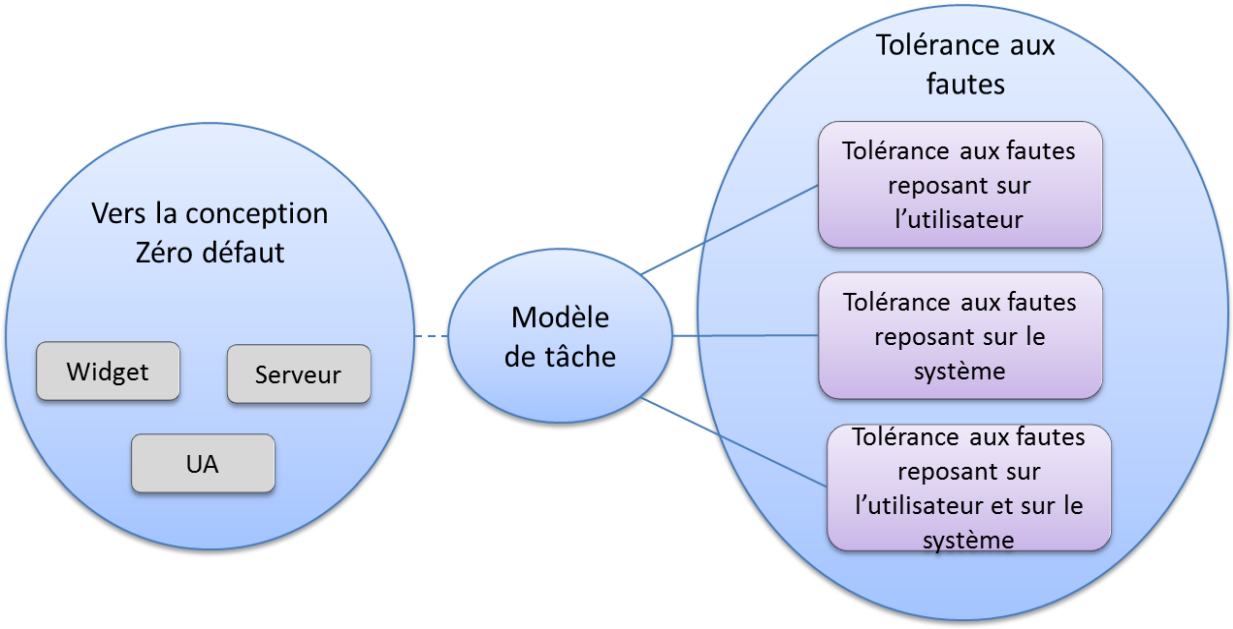


Figure III.8 les principes de la méthode proposée

La première approche est de tendre vers une conception zéro défaut. Comme expliqué précédemment, le standard ARINC 661 ne précise pas le comportement des widgets, cette partie est propre à chaque avionneur. Dans l'approche vers la conception zéro défaut, nous proposons de réaliser une description précise et non ambigüe du comportement du système interactif, c'est-à-dire de ces composants logiciels, à savoir les widgets, l'UA et le serveur. Cette approche se base sur le fait qu'une description non précise et ambigüe du système interactif est à l'origine des fautes détectées lors de l'exploitation (Leveson, 1995). L'objectif est ici de minimiser les fautes de développement résiduelles dans le composant final.

La seconde approche est l'utilisation de techniques de tolérance aux fautes. En effet, Une description précise du comportement du système interactif et un processus de développement rigoureux au travers des normes permet d'éliminer un maximum de fautes lors des phases de conception, mais ne garantit pas d'avoir un système parfait, i.e. sans faute. Il existe toujours des fautes résiduelles de conception, des fautes matérielles (bip flip), ou venant de l'environnement de l'avion qui peuvent survenir au sein du système interactif. Il est donc important de mettre en place des techniques de tolérance aux fautes afin de s'assurer que le système continue à remplir ses fonctions en dépit de l'occurrence de fautes. Nous proposons trois options tolérantes aux fautes pour les systèmes interactifs.

La première option de tolérance aux fautes est une tolérance aux fautes reposant sur l'utilisateur. L'utilisateur est l'élément important du système interactif, il est le moteur de toutes les actions de contrôle et le récepteur final des données affichées. L'idée ici est de se servir de l'utilisateur qui est un élément indépendant pour vérifier le bon fonctionnement du système.

La seconde option de la tolérance aux fautes s'appuie sur des mécanismes de redondance reposant sur le système. L'objectif de cette approche est d'adapter les techniques de tolérance aux fautes aux composants de l'application interactive et plus particulièrement à l'objet d'interaction élémentaire qu'est le widget. La fonction principale des pilotes étant de piloter l'avion et non de surveiller le système, l'idée ici est d'impacter le moins possible le pilote dans la réalisation de sa fonction.

La troisième option de la tolérance aux fautes est une approche mixte. Elle intègre à la fois la tolérance aux fautes reposant sur l'utilisateur et celle reposant sur le système. Pour des fonctions classées catastrophiques, il est demandé de n'avoir aucune panne simple pouvant conduire à des défaillances catastrophiques (critère *Fail Safe*). Cette option peut être utilisée pour respecter ce critère de *Fail Safe.* En cas de limite de l'approche de tolérance aux fautes reposant sur le système, l'utilisateur qui connaît mieux l'environnement du système et le contexte d'utilisation jouera ici le rôle de contrôleur final du système.

La dernière approche est l'explicitation de l'impact des différentes approches de tolérance aux fautes sur l'utilisabilité du système interactif. L'utilisabilité dont nous parlons ici se réduit à deux attributs de la norme ISO 9241-11 qui sont l'efficacité et l'efficience. L'efficacité désigne la capacité d'arriver à ses buts et l'efficience désigne le fait de réaliser un objectif avec le minimum de moyens engagés possibles. Pour expliciter ces attributs, nous nous reposons sur l'analyse des modèles de tâche de l'utilisateur. La réalisation des modèles de tâches sur la conception zéro défaut représente l'utilisation nominale du système interactif, elle constitue la base de comparaison des autres modèles de tâche, d'où le pointillé la Figure III.8.

Nous allons décrire en détail dans les sous-sections suivantes l'approche pour une conception zéro défaut et l'approche tolérance aux fautes. Leurs applications via l'utilisation des modèles ICO et de l'outil PetShop seront réalisées dans les Chapitre I et Chapitre I ainsi que leur impact sur l'utilisabilité. Mais avant d'expliquer ces approches, nous allons tout d'abord présenter l'ensemble des widgets du standard ARINC 661 et la classification que nous avons réalisée sur ces widgets. Cette classification a été la première phase de nos travaux, qui a ensuite servi de base pour les différentes approches proposées.

III.2.1 Les widgets au cœur de l'approche proposée

Les widgets sont les composants logiciels de base du système interactif, et ce sont eux qui composent l'interface graphique de l'UA, comme le rappelle la Figure III.9 qui montre la structure hiérarchique d'une interface graphique.

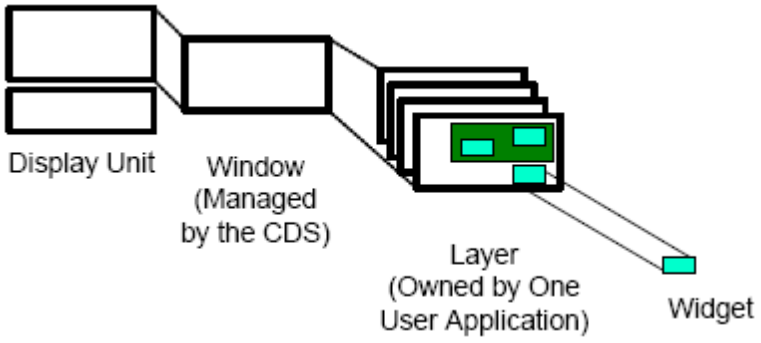


Figure III.9: Organisation hiérarchique d'une interface graphique

En parcourant la bibliothèque de widgets du document de spécification de l'ARINC661, on constate que certains widgets ont des fonctions communes. Certains servent par exemple à la saisie de données, d'autres à réaliser des commandes ou à choisir un élément dans une liste, les représentations peuvent être diverses, mais l'objectif dans l'utilisation du widget est le même.

Plusieurs études ergonomiques ont été menées dans ce domaine, nous nous sommes basés sur les travaux de Christelle Farenc (Farenc, 1997) qui est parti de la constatation qu'un certain nombre de groupes d'objets graphiques étaient concernés par les mêmes règles ergonomiques pour réaliser une typologie. La typologie ici étant constituée de classes qui regroupent les objets graphiques concernés par les mêmes règles ergonomiques. Les classes formant une arborescence à plusieurs niveaux d'abstraction, les sous-classes héritant des règles ergonomiques des classes précédentes dans la hiérarchie.

Un des avantages de cette typologie est qu'elle est indépendante d'un environnement et peut donc être appliquée pour une interface aéronautique. Notre objectif n'est pas de revoir toutes les règles ergonomiques et leur application sur les widgets, une large littérature existent déjà dessus (Mayhew , 1992) (Vanderdonckt, 1994) et la plupart des widgets étant identiques à ceux retrouvés dans les boîtes à outils des interfaces graphiques d'autres applications. Partant donc de la bibliothèque des widgets on les allouera à la classe correspondante dans la typologie. La typologie a été enrichie pour s'adapter au standard, nous avons donc créé de nouvelles classes pour positionner de nouveaux types de widgets.

Les avantages qu'on tire de faire une telle classification sont :

1. Mieux suivre l'évolution du standard ARINC 661. La typologie permet en effet d'avoir une vue organisée de l'ensemble des widgets du standard, et permet d'observer assez rapidement si un widget a été ajouté ou supprimé dans une classe.
2. Sélectionner de façon précise le widget correspondant au besoin de l'utilisateur . La typologie classe les widgets selon les règles ergonomiques, donc en fonction d'un objectif d'utilisation. Si l'on veut par exemple réaliser une interface où l'utilisateur peut saisir une donnée, on ira dans la classe

de la typologie regroupant l'ensemble des widgets de saisie de données, et on sélectionnera plus finement le widget correspondant aux attentes d'utilisation.

3. Identifier les règles s'appliquant à des groupes de widgets, via les règles ergonomiques qui sont appliquées au départ pour classer les widgets.
4. Diminuer le temps de recherche des règles et recommandations. Ceci en sachant que les widgets d'une même classe sont liés par les mêmes règles et recommandations.

La Figure III.10 représente la typologie des widgets, les classes avec un point noir sont celles issues de la typologie de Christelle Farenc, le reste a été ajouté dans le cadre de notre étude. On y retrouve 4 niveaux d'abstraction, le widget étant le premier niveau d'abstraction.

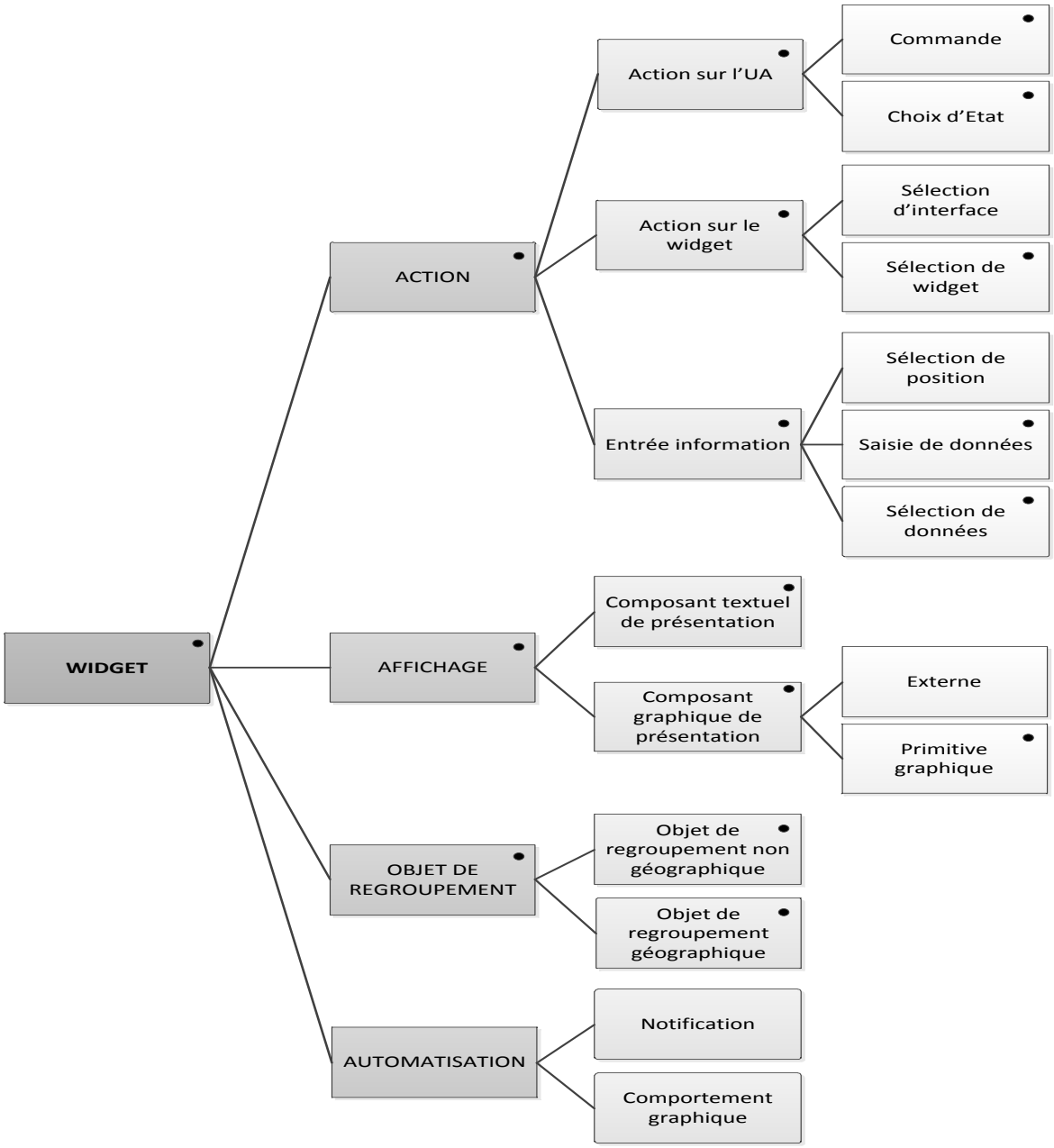


Figure III.10 Typologie de widgets

III.2.1.1.1.1 Les widgets d'action

Les widgets d'Action sont des widgets à travers lesquels l'utilisateur peut agir sur le système, pour entrer une donnée ou lancer un traitement via une commande. Cette classe comporte trois sous-classes.

1. **Action sur l'UA** : tous les widgets sur lesquels l'utilisateur peut agir soit pour lancer un traitement, soit pour changer l'état du système. Elle se subdivise en 2 sous-classes :
    - **Commande** : tous les objets sur lesquels l'utilisateur peut agir, et qui ont pour propriété de déclencher un traitement de l'UA (exemple le pushbutton).
    - **Choix d'état** : tous les widgets sur lesquels l'utilisateur peut agir, et qui ont pour propriété de faire changer l'état de l'UA (exemple passage d'un état ON à OFF).
2. **Action sur le widget** : tous les widgets sur lesquels l'utilisateur peut agir soit pour changer "la visualisation" de l'interface, soit pour sélectionner un objet de l'interface. Ces objets sont des objets de l'interface qui ne servent ni à changer l'état du système, ni à lancer un traitement. Elle se divise en deux groupes :
    - **Sélection d'interface** : tous les widgets de l'interface qui permettent à l'utilisateur d'agir sur l'interface pour changer l'état de l'interface dans le sens : changer la vue de l'interface. (exemple le TabbedPanelGroup qui permet le choix d'un onglet d'affichage).
    - **Sélection de widget** : tous les objets de l'interface qui sont activables, mais qui ne déclenchent aucun traitement ni changement de l'état du système.
3. **Entrée information** : tous les objets qui servent à l'entrée de données ou de document (texte, graphique, ...) par la saisie ou par la sélection. Elle comporte les sous-classes suivantes :
    - **Saisie de données** : tous les widgets qui permettent la saisie de données par l'utilisateur
    - **Sélection de données** : tous les widgets qui contiennent des données que l'utilisateur va pouvoir sélectionner.
    - **Sélection de position** : Widgets qui permettent d'envoyer des informations sur la position de l'utilisateur sur l'interface (position du curseur).

Ce sont les plus importants du point de vue de la sûreté de fonctionnement car une sortie erronée peut déclencher une fonction au niveau de l'UA, éventuellement critique.

III.2.1.1.1.2 Les widgets d'affichage

Les widgets d'affichage sont des widgets affichés à l'écran et sur lequel l'utilisateur ne peut interagir. On distingue deux catégories

1. **Composant textuel de présentation** : tous les widgets textuels (titre, libellé, texte message) qui ont en commun d'avoir une fonction informative et/ou nominative (i.e. qui permettent de "présenter" d'autres objets)
2. **Composant graphique de présentation** : tous les widgets de présentation purement graphiques. Ces widgets permettent d'identifier la nature des widgets qu'ils composent par un attribut graphique : matérialisation, pictogramme, pictogramme ascenseur, pictogramme menu système, pictogramme mise en icône, pictogramme objet, pictogramme valeur, point insertion. Elle comporte 2 sous-groupes :
    - **Externe** : tous les éléments graphiques provenant d'une source de données externe à l'UA. (vidéo, bases de données CDS).

- **Primitive graphique** : Tous les widgets avec une forme graphique de base (triangle, rectangle, cercle).

III.2.1.1.1.3 Les widgets de regroupement

Ces widgets regroupent d'autres widgets. Les widgets contenus (regroupés) ont leur "vie propre" en dehors du widget de regroupement. Cette classe regroupe des objets comme une "boîte ou fenêtre". Elle comporte deux catégories :

1. **Widget de regroupement géographique** : tous les widgets qui regroupent géographiquement d'autres widgets. Ces widgets ont un lien de localisation avec l'objet de regroupement.
2. **Widget de regroupement non géographique** : Tous les widgets qui sont composés d'autres widgets et qui vérifie certaines propriétés (visibilité, disponibilité, exclusivité dans la sélection …) sur l'ensemble de ses widgets.

Ces widgets sont aussi importants vis-à-vis de la sûreté de fonctionnement, en effet une erreur sur ces widgets peut se propager à un ensemble de widgets, pouvant créer un ensemble d'affichage erroné mais cohérent. Cet affichage erroné peut entrainer une erreur au niveau du pilote qui déclenchera des actions non souhaitées. Il est important de noter ici que les défaillances catastrophiques sont souvent la conséquence d'une cascade d'erreurs, qui impliquent ou non les choix du pilote.

III.2.1.1.1.4 Les widgets d'automatisation

Les widgets d'automatisation gèrent le comportement autonome de l'interface, et comporte les groupes suivants :

1. **Notification** : ses widgets envoient des notifications à l'UA sans entrée utilisateur sur l'interface
2. **Comportement Graphique** : widgets gérant des comportements graphiques (exemple saut curseur, focus) suite à une entrée utilisateur

La Figure III.11 montre la typologie avec l'ensemble des widgets du standard ARINC 661 du supplément 3. Les widgets de la même classe ont des comportements similaires.

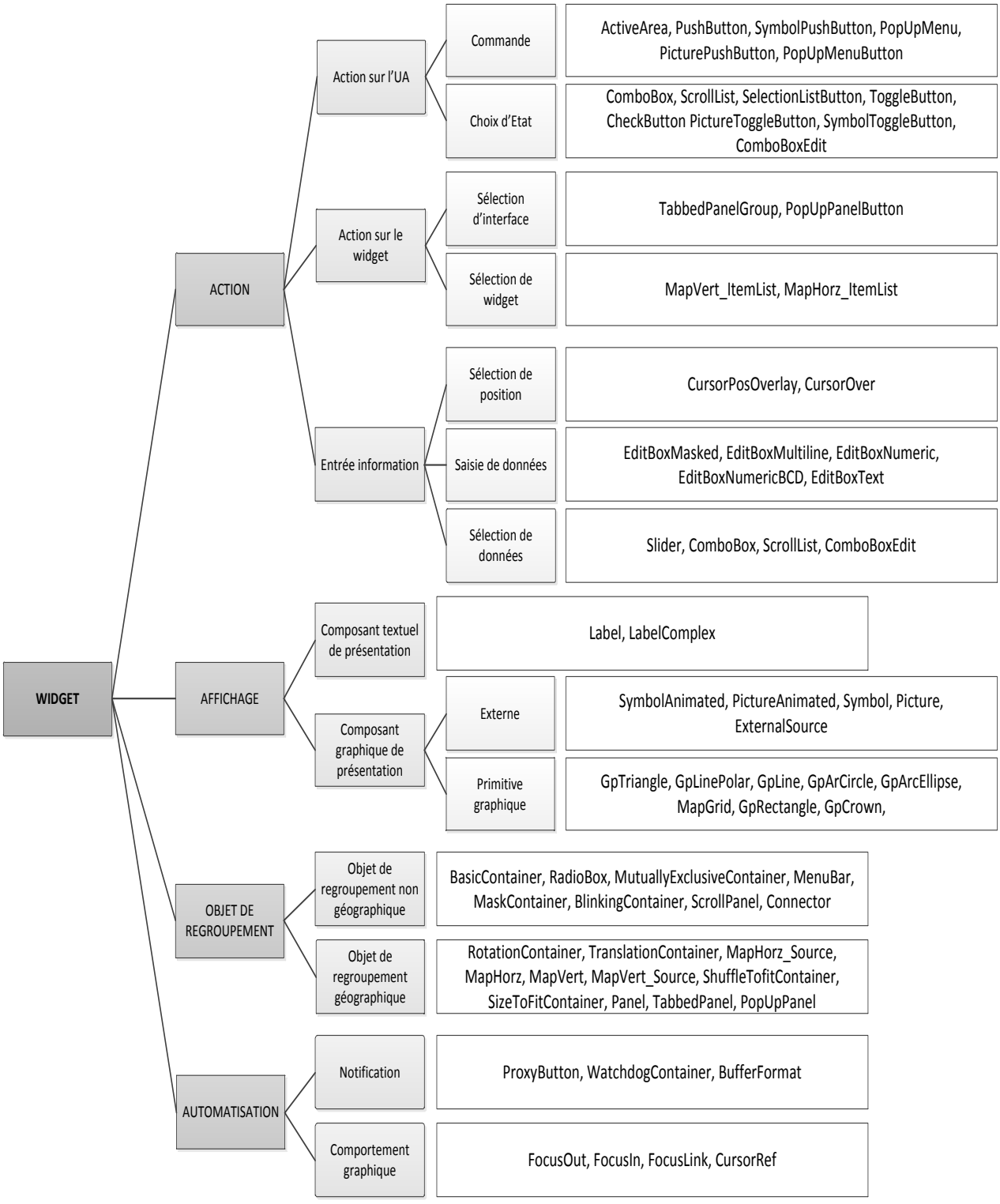


Figure III.11 Typologie des widgets du standard ARINC 661 supplément 3

III.2.2 **Approche vers une conception zéro défaut**

L'approche zéro défaut consiste à avoir une description complète, précise et non ambigüe des composants du système interactif et ceci au travers des techniques de description formelle. Les techniques de description formelle permettent la vérification et la validation des propriétés du système et facilitent ainsi son implémentation. L'utilisation d'une description précise aide les concepteurs à une bonne compréhension du fonctionnement du système, à identifier les problèmes de fonctionnement possibles et à mettre en place les barrières de sécurité adéquates.

Nous utilisons le formalisme d'Objets Coopératif Interactifs (ICO) pour réaliser la description complète et précise du comportement des composants du système interactif. Comme nous l'avons vu au Chapitre I, ICO est basé sur les réseaux de Petri haut niveau. Le formalisme ICO permet de décrire les états du système et ses changements d'états, grâce à la fonction d'activation de maintenir une consistance entre l'état d'un composant et son apparence graphique, et grâce à la fonction de rendu d'indiquer comment les évènements sur les systèmes d'entrée seront reçus par des transitions du composant, ainsi que le lien entre la disponibilité des évènements d'entrées de l'utilisateur et des transitions du composant.

Comme nous l'avons précisé précédemment, l'utilisation des techniques de description formelle permettent la vérification et la validation des propriétés du système. Le formalisme ICO est basé sur les réseaux de Petri haut niveau et bénéficie donc des techniques d'analyse des réseaux de Petri. On distingue parmi ces techniques d'analyse, les graphes de marquage (Buchholz, P., et al., 2002), les trappes et les siphons pour vérifier l'absence de blocage des réseaux de Petri (Jiao, L, et al., 2002), les techniques de réductions de réseaux (Berthelot, G., et al., 1979), ou encore l'analyse des invariants. L'outil Petshop qui est l'environnement d'édition des ICO, permet l'analyse des invariants de places et de transitions sur la matrice d'incidence des modèles ICO. L'analyse des invariants permet d'avoir des propriétés structurelles du réseau. Il serait aussi intéressant d'ajouter l'analyse des propriétés dynamiques. La vérification de ces propriétés est importante, mais nous ne pourrons pas les réaliser dans nos travaux. Nous allons plutôt nous concentrer sur la description du comportement du système interactif.

Nous nous intéressons dans cette partie uniquement aux composants logiciels du système interactif, sur la Figure III.3, les principaux composants logiciels du système interactif sont, le serveur (*Event manager*, *Display manager*), les *widgets*, et les *UA*. La description du comportement se faisant au travers des modèles de réseaux de Petri haut niveau et d'un outil d'édition, nous parlerons dans la suite plus de modélisation que de description.

III.2.2.1 *Modélisation du comportement des widgets*

Nous avons vu au niveau des modèles d'architecture des systèmes interactifs au Chapitre I, que ceux-ci était décomposé en composant pour faciliter la modifiabilité du système. Ainsi la partie présentation du système interactif dans le modèle Arch comporte deux composants :

- Le composant d'interaction logique qui inclut les données à présenter à l'utilisateur et les évènements à générer à l'utilisateur. Il maintient une représentation logique des widgets qui est indépendante de la plate-forme.

– Le composant boîte à outils qui implémente la partie physique de l'interaction avec l'utilisateur, il représente les périphériques d'entrée-sortie ainsi que l'ensemble des widgets. Ce composant dépend de la plate-forme. C'est ici qu'une description graphique du comportement du widget est réalisée, sa forme, les couleurs etc.

La modélisation du comportement des widgets que nous réalisons correspond uniquement à la partie composant d'interaction logique du modèle Arch. Nous voulons avoir une approche générique indépendante de la plate-forme. Ainsi le rendu graphique des widgets qui diffère beaucoup selon les avionneurs et selon les systèmes n'est pas traité dans ce mémoire.

La modélisation du comportement du widget consiste à décrire de façon précise au travers des modèles, les états du widget en fonction des évènements de l'utilisateur sur les périphériques d'entrées et des commandes venant de l'application avion (UA), et tous les autres traitements possibles qui peuvent être réalisés par le widget.

La modélisation du comportement des widgets, nécessite (Figure III.12) :

- La bibliothèque des widgets du standard ARINC 661 : pour avoir la liste des widgets et des paramètres les caractérisant. Nous partirons de la typologie de widget réalisée précédemment pour sélectionner quelques widgets dont le comportement représente au mieux l'ensemble des widgets. Nous réalisons ensuite la modélisation des widgets sélectionnés.
- De connaître l'environnement d'interaction, c'est-à-dire les périphériques d'entrées/Sorties, et les types d'interactions réalisées. Dans un cockpit d'avion, les périphériques d'entrée ce sont des claviers et dispositif de pointage et les périphériques de sorties des écrans LCD. Les interactions sont de type WIMP (Windows, Icons, Menus, Pointing device).
- La technique de description formelle que nous allons utiliser est ICO, basé sur les réseaux de Petri à objet.
- De préciser des comportements interdits sur les widgets, ceci afin d'avoir une programmation défensive en intégrant directement dans le modèle de comportement des barrières de sécurité. Ces comportements interdits seront fait au travers d'assertions.

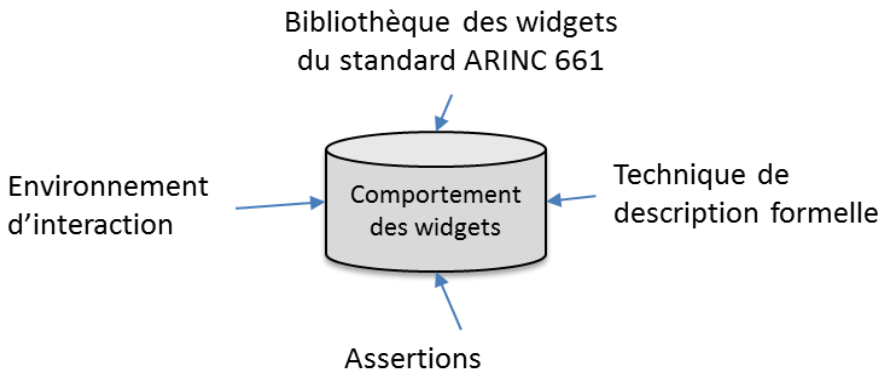


Figure III.12 Spécification des widgets

Nous allons expliquer ci-dessous les assertions sur les widgets

III.2.2.1.1 Les assertions sur les widgets

Une assertion est une expression logique permettant d'effectuer en ligne un contrôle de vraisemblance sur les objets d'un programme. La valeur de l'expression logique est à VRAI si l'état est correct et à Faux dans le cas contraire ; dans ce dernier cas, une exception est levée. (Arlat, et al., 2006).

Pour définir ces assertions, regardons tout d'abord la structure générale d'un widget. Un widget interactif peut être vu comme un composant à deux entrées et deux sorties (Figure III.13).

Figure III.13 Interface d'un pushbutton

Au niveau des entrées on distingue :

- Une entrée venant des périphériques d'entrée (clavier et dispositif de pointage), pour notifier des actions de l'utilisateur sur ces périphériques (*User Inputs*).
- Une entrée venant des applications clientes pour mettre à jour les paramètres internes de widget (*Setparameters*).

Au niveau des sorties on distingue :

- Une sortie vers les applications clientes pour notifier d'un évènement de l'utilisateur sur les périphériques d'entrée (*Events*).
- Une sortie vers les écrans, pour notifier à l'utilisateur de la mise à jour d'un paramètre du widget (*Rendering*).

Selon le sens de traitement de données, on distingue deux types de flot de données sur le widget.

- Le **flot de contrôle** décrit l'ensemble des évènements transmis du système d'affichage et de contrôle (CDS) vers les applications clientes (UA).
- Le **flot d'affichage** décrit l'ensemble des commandes transmises des applications clientes (UA) vers le système d'affichage et de contrôle (CDS).

Le Tableau III.6 résume le comportement normal d'un widget par rapport aux flots de données et les modes de défaillance qu'on peut y rencontrer. Les comportements normaux correspondent ici à une formulation informelle des assertions du widget.

Flot de contrôle	**Comportement normal**	A la réception de l'évènement du périphérique d'entrée approprié, si le widget est dans l'état défini pour envoyer l'évènement, l'évènement est envoyé
	Modes de défaillance	A la réception de l'évènement du périphérique d'entrée (clic, saisie clavier), le widget est dans un état défini pour envoyer l'évènement mais l'évènement n'est pas envoyé
		Sans réception de l'évènement du périphérique d'entrée, le widget envoi l'évènement
		Le widget envoi l'évènement alors qu'il n'est pas dans un état approprié
		L'évènement est envoyé avec une valeur corrompu (ne s'applique qu'aux widgets ayant ce paramètre en plus)
		Les évènements sont envoyés dans le mauvais ordre (ne s'applique qu'aux widgets ayant ce paramètre en plus)
Flot d'affichage	**Comportement normal**	A la réception d'une commande de l'application cliente, le widget doit mettre à jour son paramètre conformément à la valeur envoyée par l'application cliente
	Modes de défaillance	A la réception de la commande de l'application cliente, le widget ne met pas à jour ses paramètres internes
		Sans réception de la commande de l'application cliente, le widget modifie ses paramètres internes (label, valeur numérique, état visible …)
		Les données mis à jour sont corrompues (modification du label, de la valeur saisie, mauvais état de visibilité)

Tableau III.6 Assertions niveau widget

Ces assertions se déclinent de façon différente selon les classes de widget de la typologie. Les actions de l'utilisateur sont assez génériques sur les périphériques d'entrées (Clic, saisie de donnée, scroll), par contre les évènements et les paramètres diffèrent d'un widget à l'autre.

Ci-dessous un exemple plus précis de la formulation d'une assertion sur un widget de type PushButton.

Comme nous l'avons vu dans la section précédente, le pushbutton est un widget dont les paramètres modifiables en exécution (*setparameter*) sont : *Visible*, *Enable*, *LabelString* et *Styleset*. (Figure III.14)

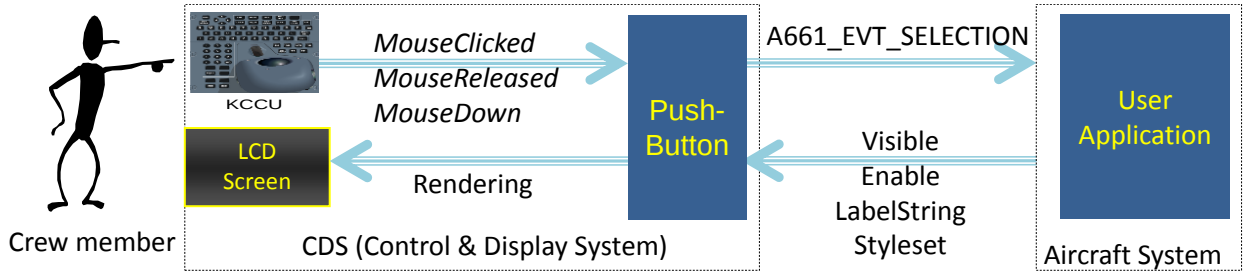


Figure III.14 Communication sur le Pushbutton

Les différents évènements des périphériques d'entrées qu'il peut recevoir sont : *MouseClicked*, *MouseReleased* et *MouseDown*.

Le pushbutton n'envoi l'évènement (Event) A661_EVT_SELECTION à l'UA que lorsqu'il est visible, Enable et qu'il a reçu un clic.

L'assertion exécutable du widget pushbutton pour le flot de contrôle se traduirait donc par la Figure III.15.

Assertion pushbutton (comportement normal)

L'Event A661_EVT_SELECTION est envoyé à l'UA si :

Réception Event Mouseclick et enable= true et visible= true

Figure III.15 Assertion flot de contrôle pushbutton

Pour le flot d'affichage, l'assertion consisterait à vérifier qu'après réception d'un *setparameter*, la valeur courante a bien changé. Des vérifications sur la taille des chaînes de caractère reçues, sur la validité de la plage de valeurs numériques sont des contrôles qui peuvent être ajoutés, et une exception sera levée en cas d'erreur.

Le code des assertions diffère selon le widget. Pour tous les widgets de la catégorie action, la précondition pour interagir dessus est que le widget soit visible et enable, c'est maintenant le type et l'ordre de l'action utilisateur qui diffèrent.

Les widgets des *classes action sur l'UA* et *action sur le widget* ne reçoivent que les évènements du dispositif de pointage (clic) par contre le nombre de clic varie selon les widgets.

Les widgets de la *classe saisie de données* ont un comportement plus complexe, puisqu'ils reçoivent à la fois les évènements du clavier et du dispositif de pointage, et en plus un ordre d'exécution est requis.

Les widgets de la *classe sélection de données* et *sélection de position,* doivent à la fois détecter les clics et certains mouvements du dispositif de pointage comme le scroll et le glissement du curseur.

Les widgets de la *classe affichage* et *regroupement* ne sont pas interactifs et doivent juste s'assurer de d'intégrité du flot d'affichage. Par contre certains widgets de regroupement ont des propriétés spécifiques à vérifier. Par exemple un widget *radiobox*, doit s'assurer qu'un seul widget est sélectionné à la fois parmi les widgets dont il est contenant.

Les widgets de la *classe automatisation* ont chacun un comportement assez particulier, ils n'ont pas de comportement généralisable, il faut étudier chacun de ses widgets indépendamment.

III.2.2.2 *Modélisation du comportement de l'UA*

La sûreté de fonctionnement d'un système interactif inclus la sûreté de fonctionnement du CDS et aussi de l'UA. L'approche conception zéro défaut implique alors la modélisation de l'application cliente (UA). Cette modélisation est dépendante de la fonctionnalité applicative, elle n'est pas générique mais utile pour établir des scénarios de test du CDS. C'est l'UA qui demande les mises à jour des paramètres des widgets et qui reçoit des évènements levés par le widget suite à une action de l'utilisateur. Il est important de modéliser le comportement de l'UA afin de s'assurer que le comportement du widget est conforme à celui de l'UA.

Cette modélisation consiste à décrire sous forme de modèle, l'ensemble des états dans lequel l'application peut se trouver, et pour chacun de ces états l'ensemble des évènements auxquels elle peut réagir.

Pour réaliser la modélisation de l'UA on a besoin des éléments suivants (Figure III.16):

- La bibliothèque des widgets, car l'interface graphique de l'UA est composée de widgets.
- La fonction du système avion, afin de définir la fonctionnalité de son interface graphique et par conséquent des principaux widgets qu'elle va utiliser.
- Le protocole d'échange qui est ici de type client-serveur, le client étant l'UA et le serveur le CDS.
- La technique de description formelle, qui est ICO.

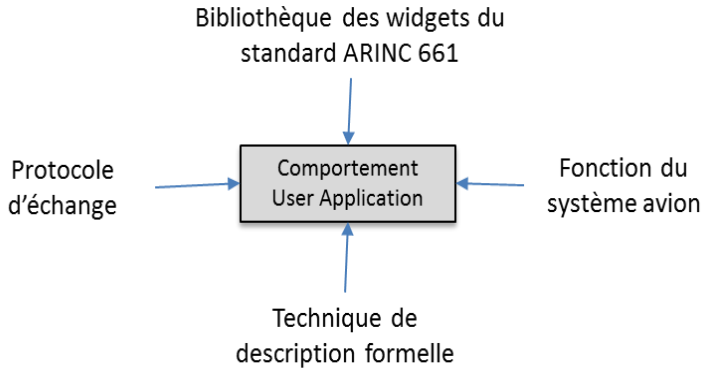


Figure III.16 Spécification UA

III.2.2.3 *Modélisation du comportement du serveur*

Comme nous l'avons vu dans la description de l'architecture matérielle et logicielle du cockpit interactif, c'est le serveur qui réceptionne les évènements venant des périphériques d'entrées (clavier/dispositif de pointage), identifie l'instance du widget sur lequel il y a eu l'action de l'utilisateur pour le transmettre l'évènement. C'est aussi le serveur qui reçoit les demandes de mise à jour venant de l'UA et les transmet à l'instance du widget concerné.

La modélisation de serveur est assez complexe, et elle a été réalisée en grande partie durant la thèse de Barboni (Barboni, 2006). La modélisation du comportement du serveur consiste à décrire sous forme de modèle les éléments suivants :

- La gestion de l'instanciation des widgets
- La gestion des évènements utilisateur (clavier/dispositif de pointage) et leur impact sur les widgets
- La gestion d'une partie des techniques d'interaction (la gestion du *focus*, de la visibilité des *layers*, du *highlight*).
- Etc.

Comme nous l'avons expliqué dans le périmètre de non travaux, nous ne le traitons pas dans ce mémoire. Bien que nous ayons ajouté certains comportements dans le serveur par rapport au modèle initial réalisé dans (Barboni, 2006) pour permettre la réalisation de nos travaux. Ces ajouts ne sont pas significatifs vis-à-vis des enjeux de sûreté de fonctionnement et d'utilisabilité pour être présenté ici.

III.2.3 **Approche Tolérance aux fautes**

Afin de traiter les fautes résiduelles de conception, les fautes matérielles ou de l'environnement du système, il est important d'intégrer des mécanismes de tolérance aux fautes. Nous proposons ici trois options de techniques de tolérance aux fautes pour les systèmes interactifs.

La première option de tolérance aux fautes est une tolérance aux fautes reposant sur l'utilisateur. La seconde option est une tolérance aux fautes reposant sur le système, et la troisième option est une approche mixte de la tolérance aux fautes, reposant à la fois sur l'utilisateur et sur le système. Nous allons détailler chacune de ses options ci-dessous.

III.2.3.1 *Tolérance aux fautes reposant sur l'utilisateur*

L'objectif ici est de se servir de l'utilisateur qui est un composant indépendant pour vérifier le bon fonctionnement du système.

Nous présentons cette approche de tolérance aux fautes par rapport au flot de contrôle et au flot d'affichage des widgets présentés lors de la description des assertions des widgets dans la sous-section précédente.

Le flot de contrôle

Pour s'assurer que chaque évènement de l'utilisateur a été traité correctement par le système, l'utilisateur devra confirmer son action. Une demande de confirmation est utilisée sur la plupart des systèmes interactifs, pour lancer un traitement ou pour quitter une application. Dans le spatial par exemple, certaines commandes à distance pouvant avoir des conséquences dangereuses pour l'opération requiert trois clics (Poupart, et al., 2008).

Les demandes de confirmation permettent à l'utilisateur de vérifier que le système à bien pris en compte le contrôle qu'il a réalisé, ceci avant de le confirmer. Si la demande de confirmation affiche un message erroné, l'utilisateur détectera qu'il y a eu une défaillance dans le système.

La Figure III.17, montre un exemple de principe de fonctionnement.

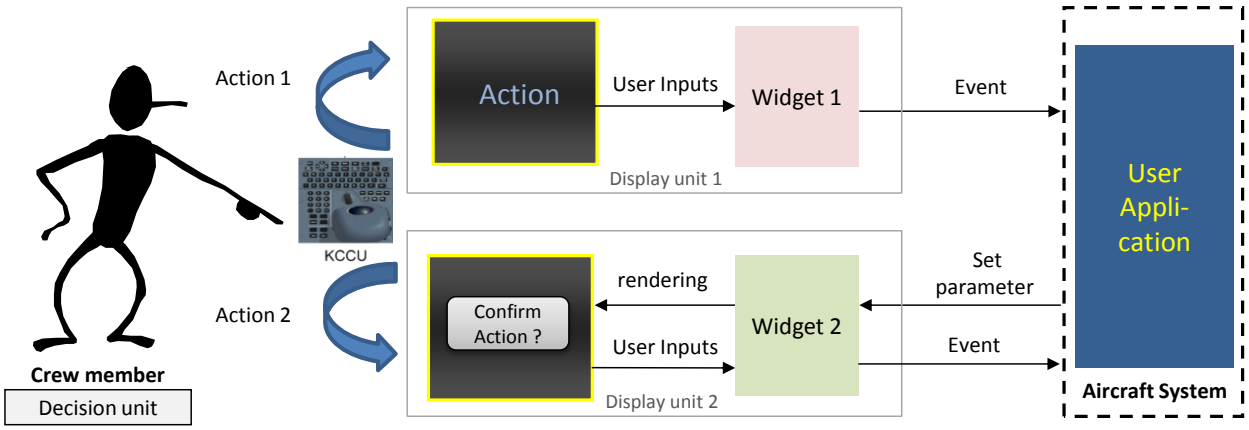


Figure III.17 Intégrité du flot de contrôle

La confirmation est faite via le même périphérique d'entrée qui a servi pour réaliser de contrôle. (KCCU)

La demande de confirmation peut être faite à différent endroit de l'interface du CDS. Soit sur la même interface qui a servi pour réaliser le contrôle, dans ce cas pour éviter les fautes de conception, il faudra que le module logiciel affichant la boîte de confirmation, soit différent de celui traitant le contrôle. On peut utiliser ici, le partitionnement spatial et temporel ((ARINC653, 1997). Soit en affichant la boîte de confirmation sur un écran (une interface) différent de celui ayant servi pour réaliser le contrôle, cette configuration permet le traitement de fautes communs du matériel (c'est celle qui est présenté à la Figure III.17).

Le Flot d'affichage

Pour s'assurer que la donnée affichée est correcte. Le principe ici est d'afficher de façon redondante l'information sur deux unités d'affichage (Figure III.18), l'utilisateur devra ensuite faire une comparaison de ces informations afin de détecter s'il y a incohérence ou pas.

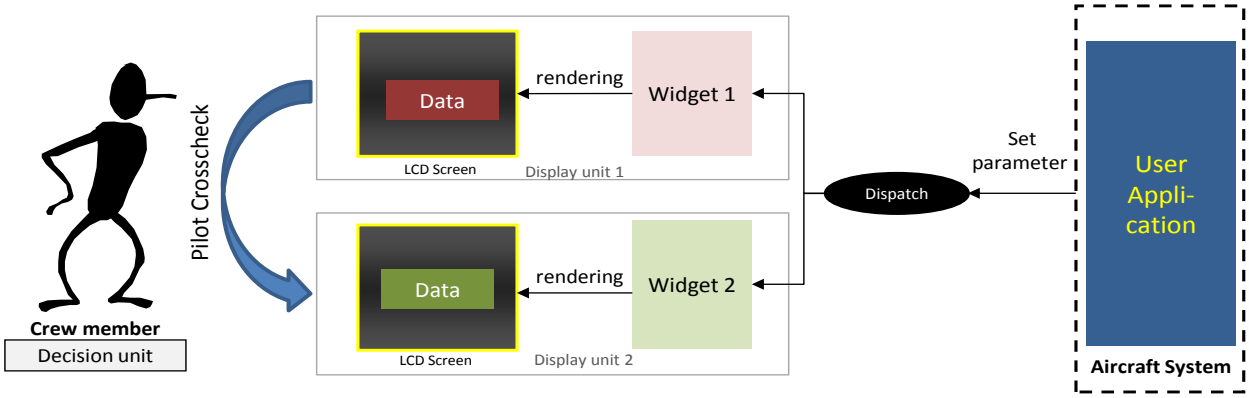


Figure III.18 Intégrité flot d'affichage avec utilisateur

L'élément de décision qui est ici l'utilisateur est de nature différente aux unités de calculs, ce qui assure une réelle ségrégation et diversification. L'hypothèse que celui-ci est sans défaut est assez forte, car on attribue 80% des accidents d'avion aux erreurs humaines (Wiegmann, et al., 2003), néanmoins la capacité d'analyse d'un équipage ne peut être remplacée.

Une telle approche n'est pas nouvelle, puisqu'elle est utilisée aujourd'hui dans des cas dégradés dans le cockpit. En effet en cas d'alerte sur une donnée affichée, l'équipage doit effectuer des contre-vérifications sur des calculateurs affichant de façon redondante la donnée pour identifier le calculateur défaillant.

Dans notre étude on considère que cette approche est utilisée dans les cas de fonctionnements normaux. L'analyse des modèles de tâches permettra de déterminer son impact sur les utilisateurs.

III.2.3.2 *Tolérance aux fautes reposant sur le système*

Comme nous l'avons vu au niveau de l'état de l'art sur la tolérance aux fautes au Chapitre I, les techniques de tolérance aux fautes sont basées sur 3 principes, la redondance, la diversification et la ségrégation. La notion de composant autotestable permet de renforcer la détection des erreurs, dues à des fautes matérielles et de développement, et ainsi améliorer la couverture d'hypothèses dans certains mécanismes de tolérance aux fautes.

Dans cette approche de tolérance aux fautes reposant sur le système, nous proposons d'appliquer les mécanismes d'autotestabilité aux composants de base de l'interaction que sont les widgets.

Selon la définition d'un composant autotestable que nous avons vu au Chapitre I, un composant autotestable est obtenu en ajoutant à un composant purement fonctionnel, un module de contrôle observant par exemple si certaines propriétés entre les entrées et les sorties du composant fonctionnel sont vérifiés. Le module de contrôle permet aussi de vérifier le comportement du bloc fonctionnel, soit à partir d'assertions exécutables élémentaires soit à partir d'un modèle comportemental du bloc fonctionnel, ce qui est le cas dans notre approche.

Nous avons vu à la Figure III.13, qu'un widget possède deux entrées et deux sorties et l'on divise ses flots de communication en deux :

- Flot d'affichage : en entrée ce sont les *setparameters* reçus des applications clientes pour mettre à jour les paramètres du widget, et la sortie du flot est l'affichage (*Rendering*) à l'écran pour notifier à l'utilisateur de la mise à jour du paramètre du widget.

- Flot de contrôle : en entrée on a les évènements utilisateurs venant des périphériques d'entrées (*user inputs*), et en sortie les *Events* levés par le widget pour notifier à l'application cliente les actions de l'utilisateur.

Le Tableau III.6 montre les assertions à vérifier sur le widget, ce tableau présente aussi les modes de défaillance redoutés suite au non-respect des propriétés suivantes sur le widget :

- Au niveau du flot de contrôle, on doit s'assurer qu'à la réception de l'évènement du périphérique d'entrée approprié, si le widget est dans un état défini pour lever un évènement, l'évènement est bien levé pour une notification à l'application cliente

- Au niveau du flot d'affichage, on doit s'assurer qu'à la réception d'une commande de l'application cliente, le widget doit mettre à jour le paramètre correspondant conformément à la valeur reçue de l'application cliente, ceci pour un affichage correct à l'utilisateur.

Pour détecter les modes de défaillance, nous proposons le modèle du widget autotestable représenté à la Figure III.19. Il est composé d'un bloc fonctionnel, et d'un bloc de contrôle qui est constitué d'une version simplifiée de la fonction et d'un module de comparaison.

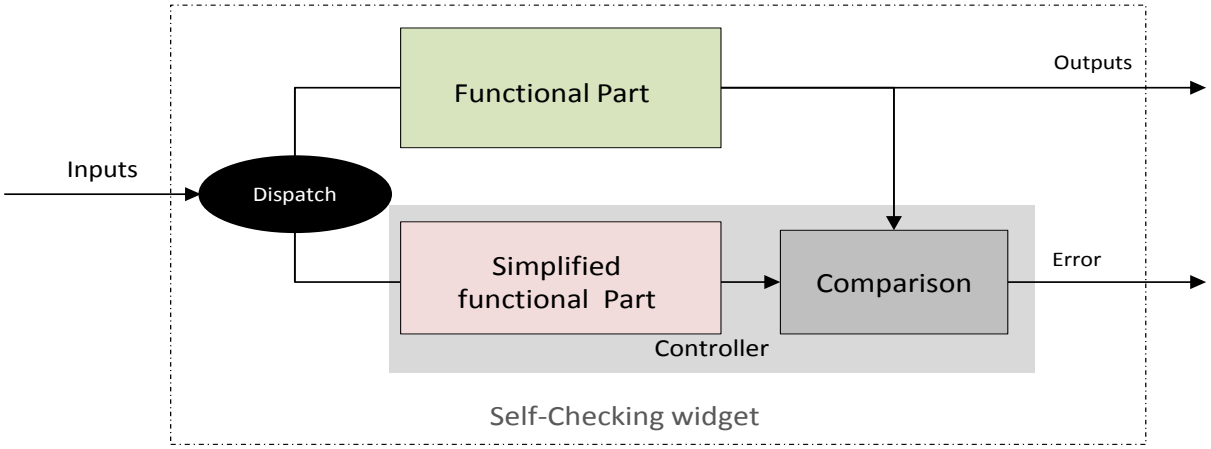


Figure III.19 : modèle de widget autotestable

Le bloc fonctionnel du widget est celui de l'approche « vers la conception zéro défaut », où le comportement précis du widget est décrit : les états du widgets, la gestion des *setparameter,* les évènements levés par le widget, les évènements reçus des périphériques d'entrées et les évènements permettant le rendu graphique du widget.

Le module de contrôle a pour but la vérification des propriétés de sûreté que l'on veut garantir. Il est constitué pour cela d'une fonction simplifiée qui est un modèle réduit du comportement du widget, seuls les paramètres et caractéristiques pertinents pour vérifier les propriétés de sûreté sont décrits dans la fonction simplifiée. Le module de contrôle est aussi constitué d'un bloc de comparaison qui a pour rôle de vérifier la cohérence entre les résultats du bloc fonctionnel et de la fonction simplifiée. Les comparaisons réalisées permettent de vérifier la cohérence des sorties en fonction des entrées.

La tolérance aux fautes est fournie par l'exécution en parallèle d'au moins deux composants autotestables (N>=2), et chaque composant est responsable de décider de l'acceptabilité de son résultat, on parle dans ce cas de programmation N autotestable. Les autres techniques de tolérance aux fautes basées sur la diversification telles que le bloc de recouvrement et la programmation N version, sont très similaires à la programmation N autotestable.

Nous nous concentrons dans nos travaux tout d'abord sur la notion de composant autotestable, et plus particulièrement sur le comportement du contrôleur. Sachant que l'architecture N-autotestable des widgets est obtenue sur simple réplication d'un widget autotestable, cette notion est donc fondamentale pour mettre en œuvre des stratégies de tolérance aux fautes.

Les Figure III.20 et Figure III.21, présentent le fonctionnement du widget autotestable selon le flot de contrôle et le flot d'affichage.

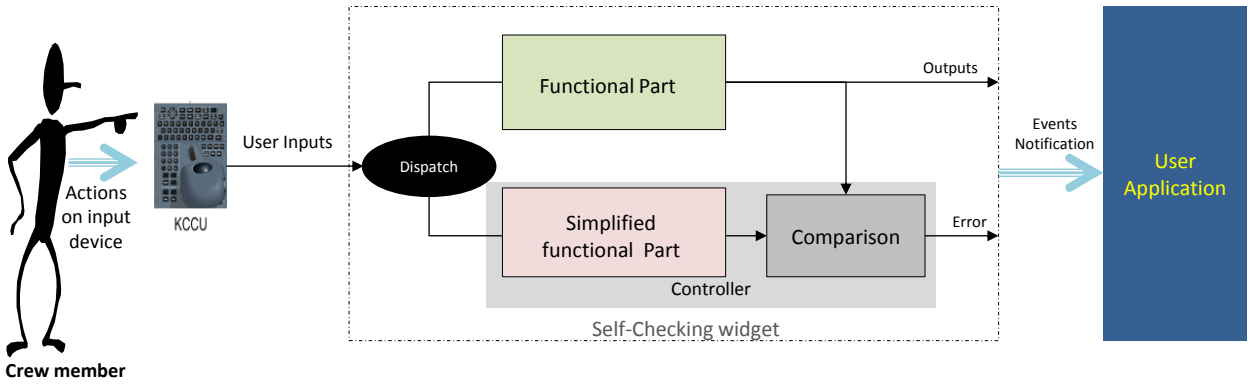


Figure III.20 : Architecture de traitement du flot de contrôle par un widget autotestable

Le fonctionnement est le suivant au niveau du flot de contrôle :

- Les évènements clavier et dispositif de pointage initiés par l'utilisateur sont dupliqués vers le composant fonctionnel et la fonction simplifiée: (*Dispatch*)
- Le composant fonctionnel et la fonction simplifiée effectue en parallèle le traitement de l'évènement en fonction de l'état interne du widget. Les résultats des deux blocs sont ensuite envoyés au module de comparaison.
- Le module de comparaison effectue une comparaison des résultats, s'il y a incohérence une erreur est levée, sinon seule la sortie du composant fonctionnel sera reçue par l'UA.

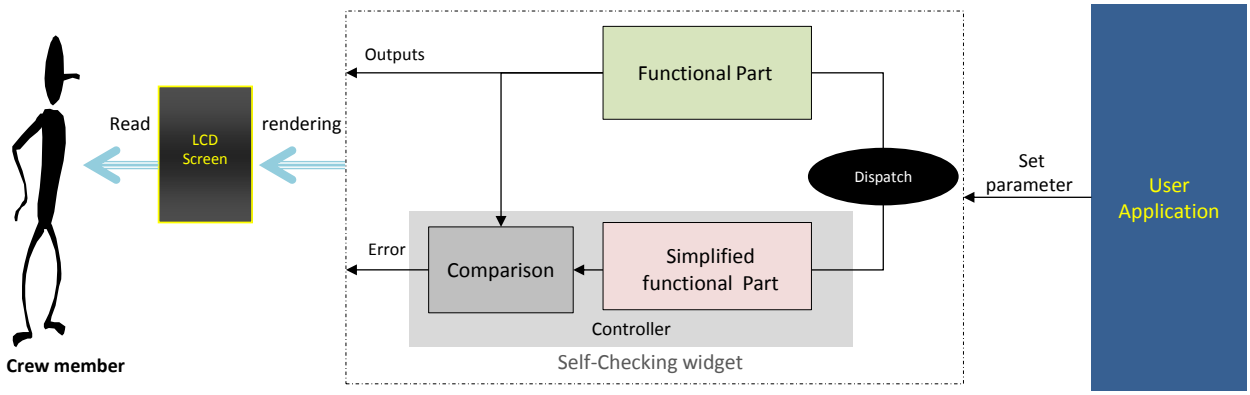


Figure III.21 : Architecture de traitement du flot d'affichage par un widget autotestable

Le flot d'affichage fonctionne sur le même principe que le flot de contrôle.

- Duplication des *setparameter* venant de l'UA vers le composant fonctionnel et sa fonction simplifiée.
- Traitement en parallèle du *setparameter* par le composant fonctionnel et la fonction simplifiée et envoie des résultats au bloc de comparaison.
- Comparaison des résultats par le bloc de comparaison, s'il y a incohérence une erreur est levée, sinon seule la sortie du composant fonctionnel est envoyée.

On fait l'hypothèse ici que le bloc de comparaison est sans défaut. Les calculs qu'il effectue sont pour la plupart assez simples, ce sont des comparaisons sur des valeurs booléennes, entières, flottantes ou des chaînes de caractères.

La modélisation du widget autotestable en ICO est réalisée au Chapitre I .

III.2.3.3 *Approche mixte de la tolérance aux fautes*

L'approche mixte de la tolérance aux fautes est celle reposant sur l'utilisateur et sur le système. Pour des fonctions classées catastrophiques, il est demandé de n'avoir aucune panne simple pouvant conduire à des défaillances catastrophiques (critère *Fail Safe*). L'approche mixte peut être utilisée pour respecter ce critère de *Fail Safe.* En cas de limite de l'approche de tolérance aux fautes reposant sur le système, l'utilisateur qui connaît mieux l'environnement du système et le contexte d'utilisation jouera ici le rôle de contrôleur final du système. Le principe est le suivant :

Au niveau du flot de contrôle : le système intègre toujours la notion de widget autotestable, et il est demandé à l'utilisateur de confirmer son action. Par contre la confirmation doit se faire en utilisant un périphérique d'entrée différent de celui ayant servi à réaliser le contrôle (le KCCU). Dans les autres options de tolérance aux fautes, le KCCU est utilisé pour réaliser le contrôle et pour confirmer l'action de contrôle. Il constitue donc un potentiel point unique de défaillance. Utiliser un moyen différent de contrôle offre à l'utilisateur une chaîne indépendante de confirmation.

Au niveau du flot d'affichage : on a une redondance dans l'affichage comme sur la tolérance aux fautes reposant sur l'utilisateur, chaque chaîne d'affichage intégrant une architecture autotestable. L'utilisateur devra ensuite réaliser une contre-vérification pour s'assurer du bon fonctionnement du système.

III.2.4 **Les modèles de tâches**

L'explicitation de l'impact des différentes approches de tolérance aux fautes sur l'utilisabilité et en particulier sur l'efficacité et l'efficience sera faite via l'analyse des modèles de tâche.

Nous utilisons les modèles HAMSTERS (Human-centered Assessment and Modeling to Support Task Engineering for Resilient Systems) (Navarre, et al., 2010) dont l'explication des principes de bases ont été données au niveau du chapitre état de l'art. Le Tableau III.7 rappelle les types de tâche des modèles HAMSTERS.

Pictogramme	Explication
	Les tâches abstraites sont des tâches qui généralement n'ont pas de type défini, car il s'agit de tâches composées d'autres sous-tâches de types différents. Ce type de tâche est représenté par une tache de peinture.
	Les tâches *sub-routine* représentent un groupe de tâches qui vont être défini dans un autre modèle de tâche.
	Les tâches systèmes sont des tâches effectuées par le système en interne. Ce type de tâche est représenté par un circuit imprimé.

	Les tâches interactives illustrent l'action de l'utilisateur sur le système (input), l'envoi d'information du système vers l'utilisateur (output), voir les deux (in/out) en parallèle. Ce type de tâches est représenté par un personnage face à un écran. Le sens de la flèche entre le personnage et l'écran représente le sens du flux d'information
	Les tâches utilisateurs correspondent à des actions qu'effectue l'utilisateur indépendamment du système. Il peut s'agir d'une tâche motrice, d'une tâche cognitive ou d'une tâche perceptive. Ce type de tâche est représenté par un personnage dont la main (tâche motrice), les oreilles (tâche perceptrice) ou la pensée (tâche cognitive) est mise en évidence selon le type de tâche.

Tableau III.7 Les types de tâches HAMSTERS

Les modèles de tâche seront réalisés au niveau de chaque application des approches conception zéro défaut et tolérance aux fautes.

Nous réalisons les modèles de tâche sur deux types d'interaction génériques que peut avoir l'utilisateur sur le système interactif:

- L'interaction en sortie : qui consiste à la lecture de données, donc à la gestion de flot d'affichage côté système.
- L'interaction en entrée : qui consiste à une action de l'utilisateur sur les périphériques d'entrée (clavier/dispositif de pointage). Nous prendrons plus précisément les cas de saisie de données et de réalisation de commande.

Nous analysons ainsi l'activité de l'utilisateur sur le système selon la lecture de donnée, la saisie de donnée et la réalisation de commande. On représentera les différentes tâches du système, les tâches interactives et les tâches de l'utilisateur (motrice, cognitive et perceptive) nécessaires à la réalisation de son activité.

La charge de travail de l'utilisateur sera évaluée en faisant la somme de l'ensemble de ses tâches. Toutefois, le même effort n'est pas demandé dans la réalisation d'une tâche, les tâches cognitives font intervenir de la mémoire et du raisonnement et nécessite dans ce cas plus d'effort que les tâches perceptives ou motrices. Dans la somme des tâches, nous donnons un poids plus élevé aux tâches cognitives. Une analyse plus fine sur les tâches peut être réalisée selon les connaissances dans le domaine de l'interaction homme machine (tels que la loi de Fitts (Fitts, 1954) ou le modèle de processeur humain (Card, et al., 1986), mais ce n'est pas l'objet de nos travaux.

III.3 CONCLUSION

Dans ce chapitre, nous avons décrit le système interactif qui fait l'objet de notre étude et les approches proposées. Nous nous intéressons ici au système d'affichage et de contrôle d'un cockpit d'avion civil dont l'interaction logicielle est basée sur le standard ARINC 661 et le paradigme d'interaction est principalement de type WIMP.

Face aux enjeux en termes de sûreté de fonctionnement et d'utilisabilité pour le développement de ce système interactif critique, nous avons proposé dans ce chapitre des approches intégrant ces différents aspects.

Trois approches sont proposées : La première approche consiste à avoir une conception zéro défaut, cette conception zéro défaut implique d'avoir une description précise et non ambigüe des composants du système interactif et ceci au travers de l'utilisation de technique formelle. La deuxième approche est l'intégration de techniques de tolérance aux fautes pour traiter les fautes résiduelles de conception, les fautes matérielles et de l'environnement du système. Trois options de techniques de tolérance aux fautes ont été proposées : la tolérance aux fautes reposant sur l'utilisateur, la tolérance aux fautes reposant sur le système et la tolérance aux fautes mixte qui repose à la fois sur l'utilisateur et sur le système. La dernière approche est l'utilisation de modèle de tâche afin d'expliciter l'impact des différentes architectures tolérantes aux fautes sur l'utilisabilité.

Nous allons maintenant présenter dans les chapitres suivants l'application des différentes approches au travers d'une étude de cas et du formalisme ICO.

Chapitre IV. Application de l'approche vers la conception zéro défaut et impact sur l'utilisabilité

Résumé du chapitre

L'objectif de ce chapitre est de montrer l'application de l'approche vers la conception zéro défaut via l'utilisation de la technique de description formelle ICO. Cette approche consiste à avoir une description complète, précise et non ambigüe des composants du système interactif. Les composants du système interactif que nous allons modéliser ici sont les widgets du standard ARINC 661 et l'application cliente (UA).

L'application de cette approche est faite au travers de l'étude de cas d'une application avionique de type FCU (Flight Control Unit). Après une description du cadre général de modélisation de l'étude de cas, nous présentons la modélisation détaillée des widgets composant l'application choisie avant de passer à la modélisation de l'application. Nous réalisons enfin les modèles de tâche de l'utilisateur sur cette approche.

IV.1 PRESENTATION DE L'ETUDE DE CAS ET DE SON ENVIRONNEMENT DE MODELISATION

L'étude de cas que nous allons présenter ci-dessous, est celle que nous allons utiliser sur l'ensemble de ce mémoire, non seulement pour l'application zéro défaut de ce chapitre, mais aussi pour l'application de la tolérance aux fautes. Elle est basée sur un système réel utilisé dans les cockpits d'avion Airbus. Le système sur lequel est basée l'étude de cas est le panneau de commande FCU (Flight Control Unit).

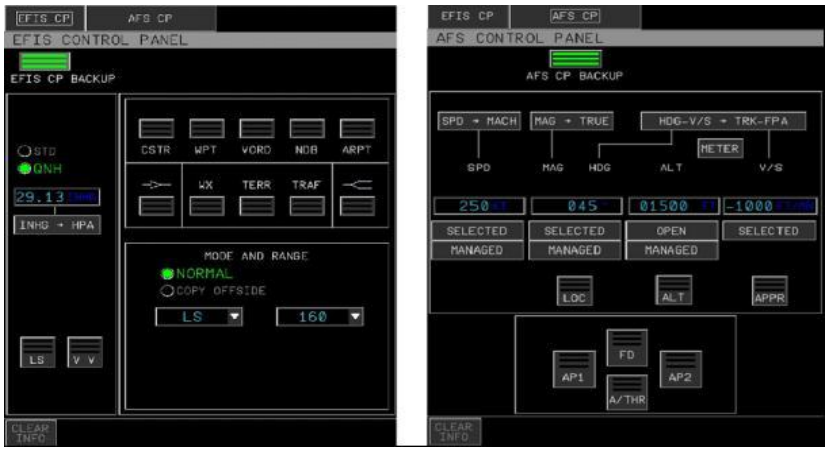
IV.1.1 Définition informelle de l'étude de cas

Le FCU est un panneau de commande physique composé de plusieurs boutons poussoirs et de commutateurs rotatifs etc. Il est situé à l'avant du cockpit et est divisé en deux types de panneaux (Figure IV.1) :

- Deux panneaux de commande de type EFIS (Electronic Flight Information system) qui permettent de configurer les écrans de pilotage et de navigation (PFD et ND) et de régler l'altitude barométrique. On distingue, un panneau pour le capitaine (*CAPT EFIS control Panel*) et un pour le co-pilote (*F/O EFIS Control Panel*).
- D'un panneau de commande de type AFS (Auto Flight System) qui permet le paramétrage du pilote automatique.

Figure IV.1 FCU (Flight Control Unit)

En cas de défaillance du panneau physique du FCU, une interface logicielle appelée FCU backup est utilisée (Figure IV.2). Le FCU backup est basée sur le standard ARINC 661, la partie fonctionnelle de son interface est donc pilotée par une UA. Elle permet de faire les mêmes configurations que le FCU réel et comporte deux pages (EFIS CP, et AFS CP).



Page EFIS CP *Page AFS CP*

Figure IV.2 Interface FCU backup A380

La position du FCU physique et du FCU backup (logiciel) dans l'environnement du cockpit de l'A380 est présenté à la Figure IV.3.

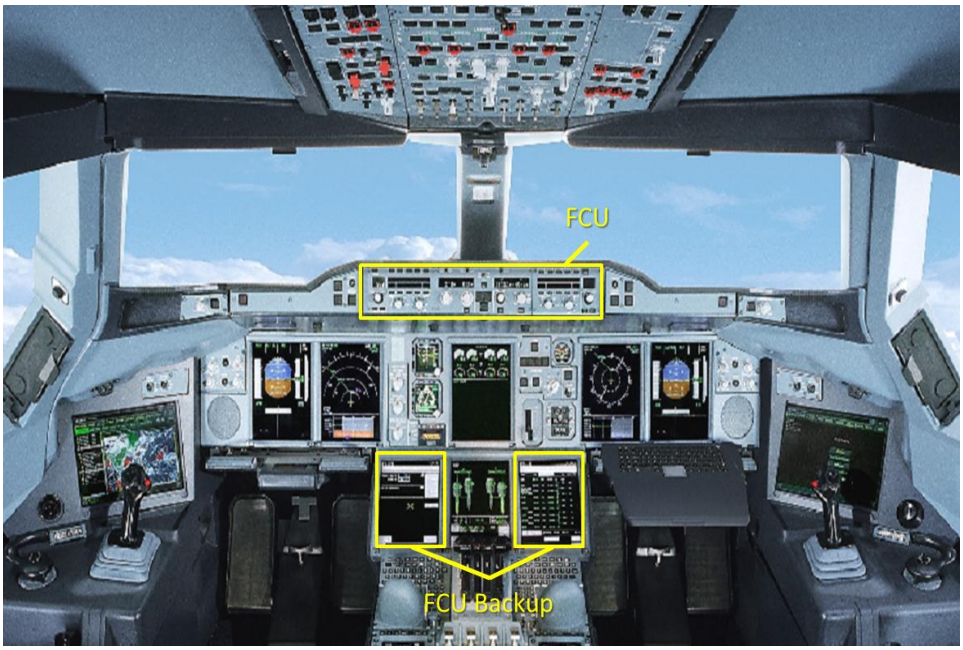


Figure IV.3 Position du FCU et du FCU Backup dans l'environnement du cockpit A380

Les fonctions paramétrées sur le FCU physique sont pour la plupart critiques, et les calculateurs de commande de vol qui y sont connectés intègrent depuis toujours des mécanismes de tolérance aux fautes. Le câblage des différents boutons est ségrégué, redondé et diversifié afin de détecter des défaillances.

Le FCU backup n'étant utilisé qu'en cas de défaillance du FCU physique, et donc dans des conditions déjà dégradées, il est quant à lui classé comme étant non critique; mais à des fins d'étude de cas nous nous sommes placés dans un contexte opérationnel où l'application graphique ne serait plus utilisée comme backup, mais comme application de base et donc comme un système critique.

Pour couvrir l'ensemble des widgets du standard ARINC 661, nous avons modifié la structure du FCU backup. Nous sommes partis de la typologie de widgets que nous avons présenté au Chapitre I, et nous nous sommes assurés que chaque classe de widget de cette typologie devait être représenté dans l'étude de cas.

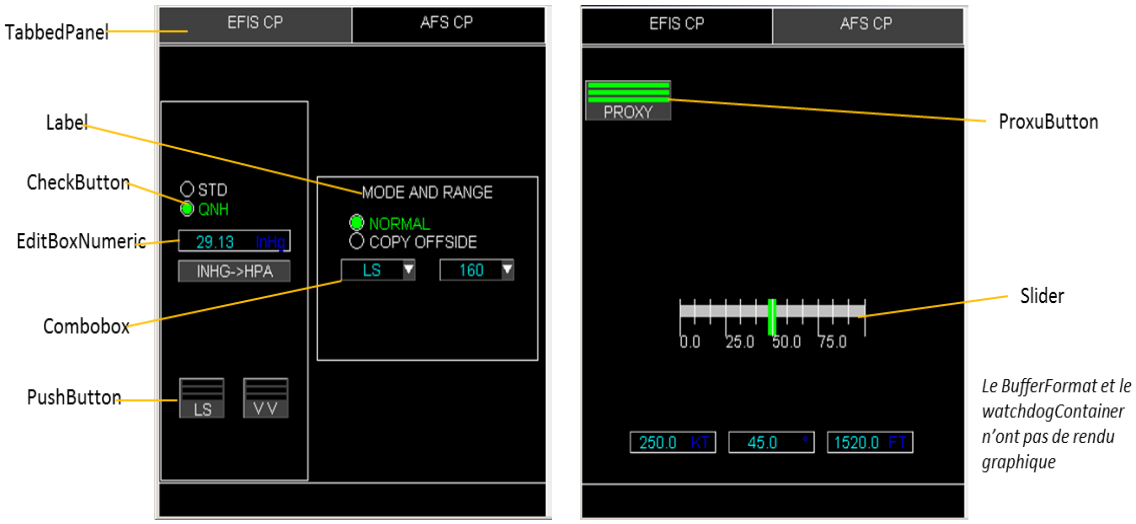
Ainsi, nous avons diminué le nombre de widgets qui se retrouvaient plusieurs fois et rajouté de nouveaux widgets pour représenter les autres classes de la typologie. L'ensemble des widgets de l'étude de cas est présenté au Tableau IV.1.

Nous appellerons dans la suite notre étude de cas FCU ou lieu de FCU backup.

Widgets présents sur le FCU backup	Widgets rajoutés
PicturePushButton	ProxyButton
ComboBox	Watchdog Container
CheckButton	BufferFormat
TabbedPanelGroup	Slider
EditBoxNumeric	
Label	
Panel	
TabbedPanel	

Tableau IV.1 Liste des widgets de l'étude de cas FCU

L'interface finale du FCU est toujours constituée de deux pages de type EFIS et AFS avec la page EFIS qui garde presque la même structure que le FCU Backup de base. Par contre la page AFS a été complètement refaite, car sur le FCU backup elle était constituée des widgets déjà présents sur la page EFIS (Pushbutton, EditBoxNumeric). Sur le FCU elle est essentiellement constituée de widgets rajoutés (ProxyButton, WatchdogContainer, BufferFormat, Slider). (Figure IV.4)



Page EFIS_CP *Page AFS_CP*

Figure IV.4 Interface de l'étude de cas FCU

En définissant cette étude de cas, nous avions pour objectif qu'elle remplisse les caractéristiques suivantes :

- Représenter l'ensemble des classes de widgets du standard ARINC 661 (Action, affichage, regroupement, automatisation)
- Représenter l'ensemble des flots de contrôle et d'affichage
- Avoir un contexte opérationnel valide, ici le FCU
- Représenter les perspectives d'évolution du cockpit, qui est ici d'utiliser une interface logicielle pour les actions critiques.

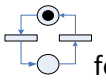
IV.1.2 Architecture logicielle de modélisation de l'étude de cas

Comme présenté dans le chapitre précédent à la Figure III.3, l'architecture générale de l'interactivité via le standard ARINC 661 fait intervenir des périphériques d'entrées/sorties, un serveur de gestion des évènements d'entrées/sorties, une bibliothèque de widgets et l'UA. Si l'on fait une correspondance entre les composants du modèle d'architecture ARCH et celui de l'interactivité avec l'ARINC 661, on obtient le schéma de la Figure IV.5. On a ici un modèle imbriqué de l'architecture ARCH.

D'un côté, le serveur qui est situé au niveau du CDS a en charge l'instanciation des widgets, la gestion des évènements utilisateurs et leur impact sur les widgets, la gestion d'une partie des techniques d'interaction, la gestion du curseur. En plus de cela il a aussi en charge l'organisation de l'affichage des pages des différentes UAs et des reconfigurations de page (Navarre, et al., 2008). C'est un composant assez complexe. Le serveur constitue en lui-même un mini modèle ARCH, avec une gestion de la logique d'interaction et du dialogue.

De l'autre côté, l'UA et les composants du système avions représentent les parties dialogue et noyau fonctionnel de l'autre modèle ARCH.

La Figure IV.5 présente aussi l'architecture de modélisation en ICO de l'étude de cas. On y retrouve les éléments du formalisme ICO, tels que la fonction d'activation et la fonction de rendu qui font le lien entre l'aspect comportement du système (UA) et l'aspect graphique. La fonction d'activation a pour rôle de récupérer les évènements (au sens ARINC 661) envoyé par le CDS, il modifiera en conséquence le comportement de l'UA, et il modifiera l'activation ou la désactivation de certains widgets en fonction du comportement de l'UA. La fonction de rendu a pour rôle de modifier l'état des widgets en fonction du changement de l'état de l'UA. Les communications entre la fonction d'activation et la fonction de rendu se font principalement par des évènements tels que vu à la section II.3 du Chapitre I.

Tous les blocs de la figure ayant le symbole  font partis du périmètre de notre étude et sont modélisés en réseau de Petri haut niveau.

Comme nous l'avons précisé dans le Chapitre I, nous ne montrerons pas la modélisation du serveur dans ce mémoire. Nous précisons néanmoins que l'ensemble du comportement du serveur n'a pas été modélisé en réseau de Petri haut niveau. C'est le cas du rendu graphique, dont nous avons voulu rendre indépendant de l'architecture de modélisation. Le rendu graphique peut être réalisée au travers d'un langage ou d'un outil dédié tel que, SVG (Scalable Vector Graphics), ou en code Java (Barboni, et al., 2007) (Navarre, et al., 2009). Cette indépendance permet de s'adapter aux différents environnements graphiques tels que le cockpit d'un avion. C'est pourquoi nous montrons sur la Figure IV.5 qu'une seule partie du serveur est modélisé en réseau de Petri haut niveau, et cette partie correspond à l'instanciation des widgets, la gestion d'une partie des techniques d'interaction et la gestion des évènements utilisateurs.

Le bloc de widgets de la figure, représente le modèle de comportement de chaque widget de l'étude de cas représenté au Tableau IV.1. Ces widgets communiquent avec l'UA en conformité avec le standard ARINC 661, par l'envoi d'*Events* et en recevant des *setparameters*.

Nous expliquons ci-dessous la modélisation du comportement des widgets et de l'UA.

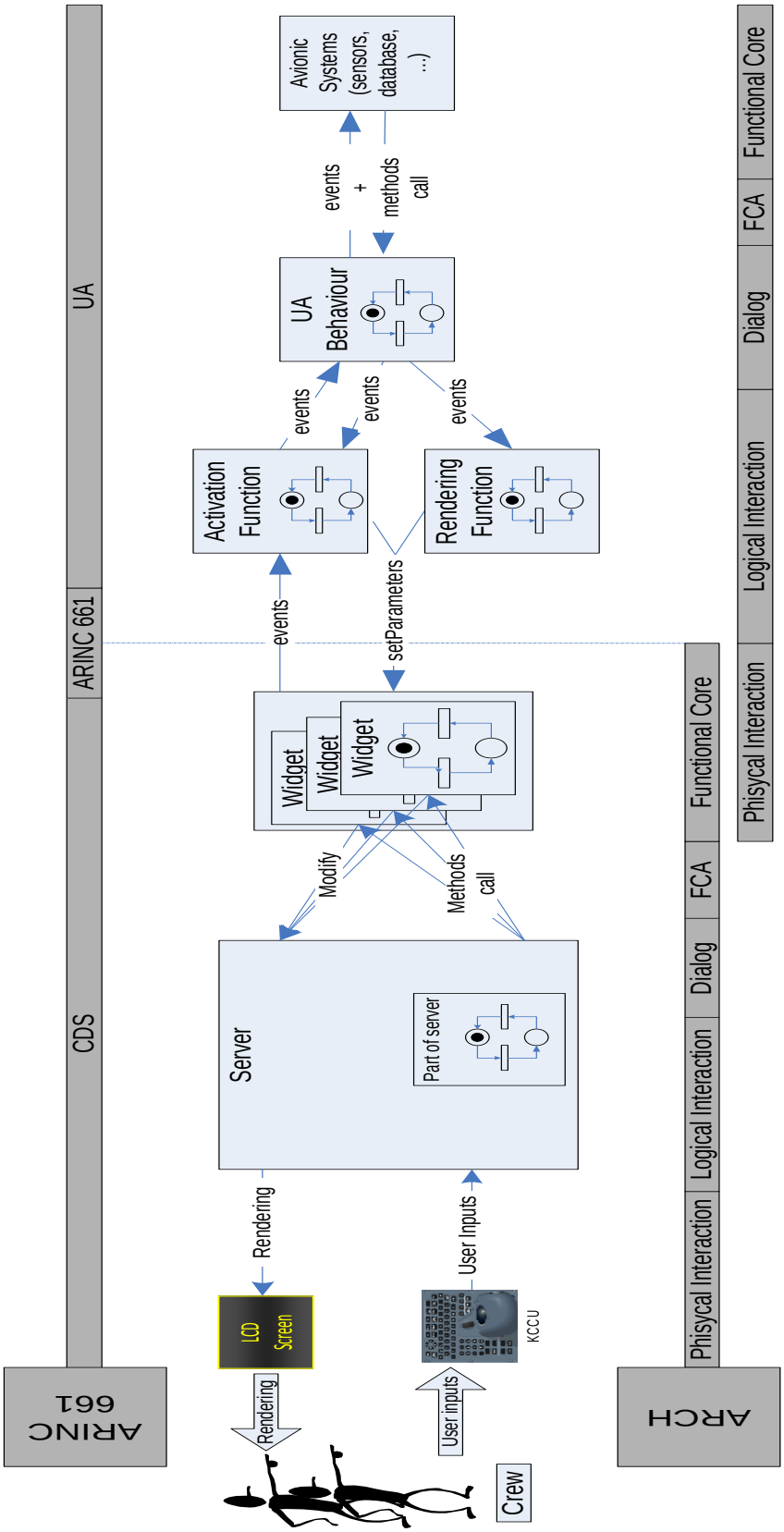


Figure IV.5 Architecture de modélisation de l'étude de cas

IV.2 MODELISATION DU COMPORTEMENT DES WIDGETS

Après avoir présenté le cadre général de modélisation, nous présentons dans cette partie les modèles de widget.

Le Tableau IV.2 présente l'ensemble des widgets de l'étude de cas et leurs différents *Events* et *setparamaters* qui sont modifiables en cours d'exécution. Certains widgets ont des paramètres en communs, comme par exemple les widgets d'action qui ont tous les paramètres visible, enable et styleset. De façon générale chaque widget a un comportement particulier résultant de la combinaison de ses paramètres et évènements spécifiques.

Classe de la typologie	**Widgets**	**Events**	**Setparameters**	
			Communs	**Spécifiques**
Action sur l'UA	PicturePushButton	A661_EVT_SELECTION	A661_ENABLE A661_VISIBLE A661_STYLE_SET	A661_STRING A661_PICTURE_REFERENCE
	ComboBox	A661_EVT_SEL_ENTRY_CHANGE		A661_ STRING_ARRAY A661_ NUMBER_OF_ENTRIES A661_SELECTED_ENTRY A661_OPENING_ENTRY
	CheckButton	A661_EVT_STATE_CHANGE		A661_STRING A661_INNER_STATE_CHECK
Action sur le widget	TabbedPanelGroup	A661_EVT_TABBED_PANEL_CHANGE		A661_ACTIVE_TABBED_PANEL
Entrée informations	EditBoxNumeric	A661_EVT_EDITBOX_OPENED A661_EVT_STRING_CONFIRMED A661_EVT_STRING_CHANGE_ABORTED		A661_VALUE A661_STRING A661_ENTRY_VALID A661_MINMAX_VALUES A661_TICS_COARSE A661_TICS_FINE A661_CURSOR_POS_BYTE A661_FORMAT_STRING A661_LEGEND_POSITION
	Slider	A661_EVT_VALUE_CHANGE		A661_VALUE
Objet de regroupement	Panel			A661_POS_XY A661_POS_X A661_POS_Y A661_SIZE_X A661_SIZE_Y A661_SIZE_XY
	TabbedPanel			A661_STRING A661_PICTURE_REFERENCE
Affichage	Label			A661_VISIBLE A661_STRING A661_POS_XY A661_POS_X A661_POS_Y A661_STYLE_SET A661_ORIENTATION A661_COLOR_INDEX A661_FONT
Automatisation	ProxyButton	A661_EVT_SELECTION		A661_TARGET_WIDGET_ID A661_ENABLE

	BufferFormat			A661_BUFFER_OF_PARAM
	Watchdog Container	A661_EVT_WATCHDOG_EXPIRED A661_EVT_WATCHDOG_NORMAL		A661_ REFRESH A661_ SHOW_FAIL

Tableau IV.2 Caractéristiques des widgets de l'Etude de cas FCU

La description en ICO de plusieurs widgets du FCU a été réalisée durant la thèse de (Barboni, 2006). Nous avons étendu ses travaux en adaptant le comportement de certains widgets pour correspondre à notre étude de cas, et aussi en décrivant de nouveaux widgets tels que : La ComboBox, Le ProxyButton, Le BufferFormat, le WatchdogContainer, le Slider.

IV.2.1 Modélisation du comportement d'un widget

Le schéma de la Figure IV.6 présente un cadre général dans lequel un widget évolue en cours d'exécution.

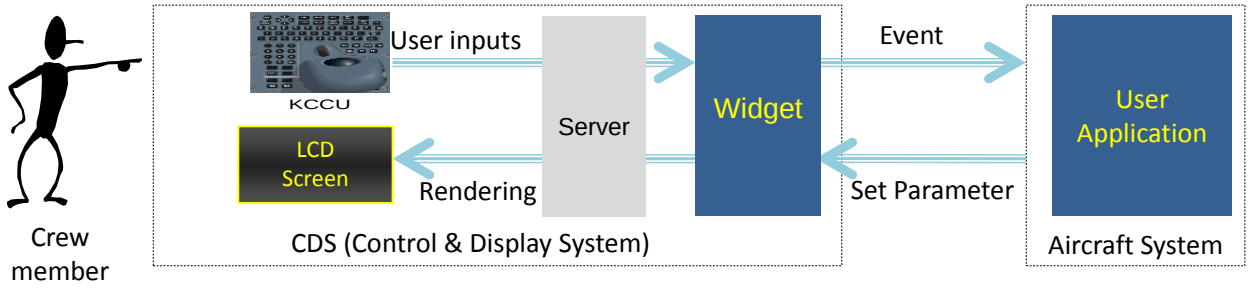


Figure IV.6 Architecture de communication widget et UA

Cette figure illustre la relation qui existe entre le widget et l'UA d'un côté et avec les périphériques d'entrées/sorties de l'autre.

Comme nous l'avons vu au Chapitre I, le standard ARINC 661 ne décrit pas le rendu graphique, ni le comportement des widgets. Ceci est propre à chaque avionneur. De même les avionneurs peuvent apporter des extensions au standard pour ajouter des fonctionnalités particulières à leurs widgets.

Au sein d'Airbus, on retrouve notamment une section précisant les traitements d'exceptions en cas de détection d'erreur sur les *setparameter* des widgets. Ces exceptions sont levées par exemple en cas de détection d'une chaîne de caractères incorrecte sur des plages de valeurs non respectées. Le Tableau IV.3 montre comment est défini ces exceptions sur quelques widgets de l'étude de cas.

Widgets	Critère d'erreur	Paramètre de widget concerné	Exceptions levées
PicturePushButton	LabelString length > MaxStringLength	A661_STRING	A661_ERROR_STRING_LENGTH
	If Picture reference is out of range of possible values associated to one application	A661_PICTURE_REFERENCE	A661_ERROR_OUTOFRANGE
EditBoxNumeric	Legend String length > MaxStringLength	A661_STRING	A661_ERROR_STRING_LENGTH
	Value is not compliant with the Format String when Format String contains one of '+', '_', '#', '.	A661_VALUE	A661_ERROR_CONFIGURATION
	Format String length> MaxFormatStringLength	A661_FORMAT_STRING	A661_ERROR_STRING_LENGTH
	Value < MinValue	A661_VALUE	A661_ERROR_MIN
	Value > MaxValue	A661_VALUE	A661_ERROR_MAX
	MinValue is not compliant with the Format String when Format String contains one of '+', '_', '#', '.	A661_MINMAX_VALUES	A661_ERROR_CONFIGURATION
	MaxValue is not compliant with the Format String when Format String contains one of '+', '_', '#', '.	A661_MINMAX_VALUES	A661_ERROR_CONFIGURATION

Tableau IV.3 Evènement d'erreur sur quelques Widgets

Suite à ces explications, la représentation ICompoNets d'un widget est représentée à la Figure IV.7. La notion d'ICompoNet a été expliqué au chapitre Etat de l'art, nous utilisons cette représentation afin d'avoir une vision abstraite des composants, mieux voir leur interface (évènement en entrée et en sortie) avant de montrer le comportement interne réalisé en ICO.

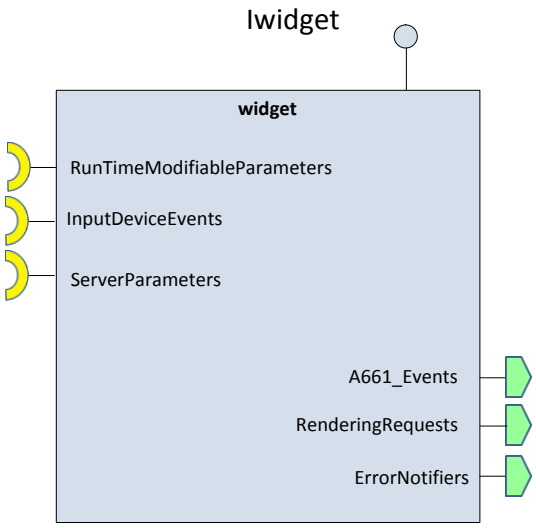


Figure IV.7 Modèle Composant d'un Widget

Trois types d'entrées sont identifiés :

- Les *RuntimeModifiableParameters* qui correspondent à l'ensemble des *setparamater* modifiable en exécution et envoyés par l'UA pour mettre à jour les paramètres du widget.
- Les *InputDeviceEvents* qui correspondent aux évènements venant des périphériques d'entrées (KCCU) et retransmis par le serveur.
- Les *serverparameters* qui correspondent aux paramètres nécessaires au serveur pour gérer le rendu graphique du widget.

Trois types de sorties :

- Les *A661_EVENTS* qui correspondent aux évènements levés par le widget tel que défini dans le standard ARINC 661.
- Les *RenderingRequests* qui correspondent aux paramètres envoyés par le widget au Display manager du serveur pour le rendu graphique.
- Les *ErrorNotifiers* qui correspondent aux exceptions sur le widget.

L'enveloppe ICompoNet du widget ainsi défini, il ne nous reste plus qu'à décrire son comportement interne avec le formalisme ICO.

IV.2.2 Exemple de modélisation du comportement de certains widgets

Nous présentons dans cette section la modélisation du comportement des widgets sur deux exemples de widgets : le PicturePushButton et l'EditBoxNumeric.

IV.2.2.1 *Modélisation du comportement du PicturePushButton*

Un PicturePushButton est un Pushbutton donc un bouton-poussoir qui peut afficher une image. C'est un widget interactif qui produira un évènement sous l'action d'un utilisateur sur les périphériques d'entrées (KCCU) comme illustré à la Figure IV.8. Le bloc serveur n'est pas présenté sur la figure pour des soucis de présentation et aussi afin de se focaliser sur le widget.

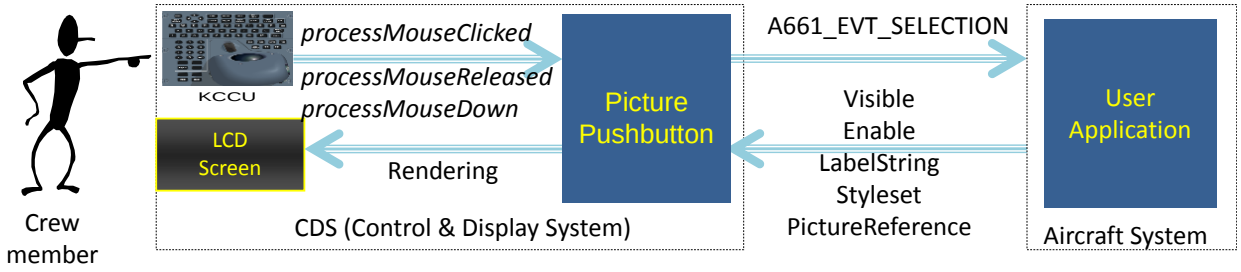


Figure IV.8 Interface PicturePushbutton

La représentation ICompoNet du PicturePushButton est représentée à la Figure IV.9.

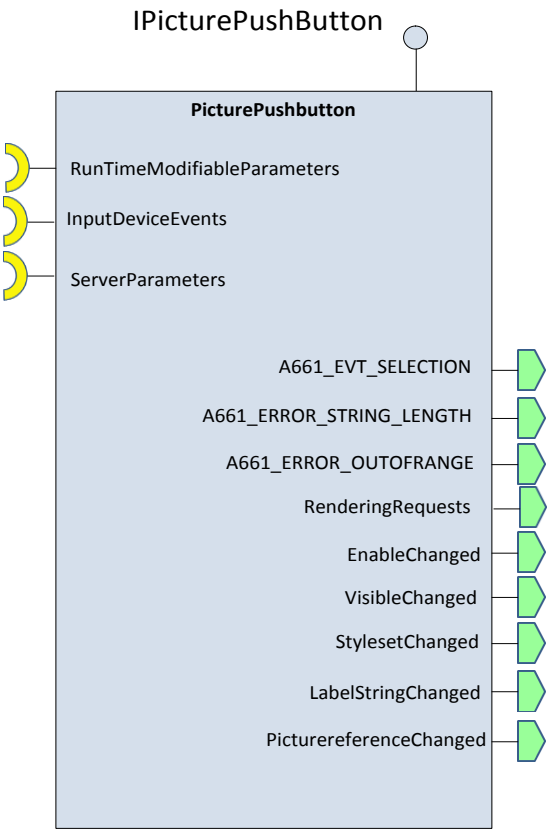


Figure IV.9 Modèle ICompoNet du PicturePushButton

L'implémentation par un réseau de Petri des différentes interfaces est expliquée dans les paragraphes suivants.

Les InputDeviceEvents qui correspondent aux évènements venant des périphériques d'entrées et retransmis par le serveur est commune à tous les widgets interactifs. Ce réceptacle correspond donc aux évènements souris (MousePressed, MouseClicked, MouseReleased, MouseMove) et aux évènements claviers (Validationkey, Abortkey, Normalkey) dont le modèle ICO est présenté à la Figure IV.10.

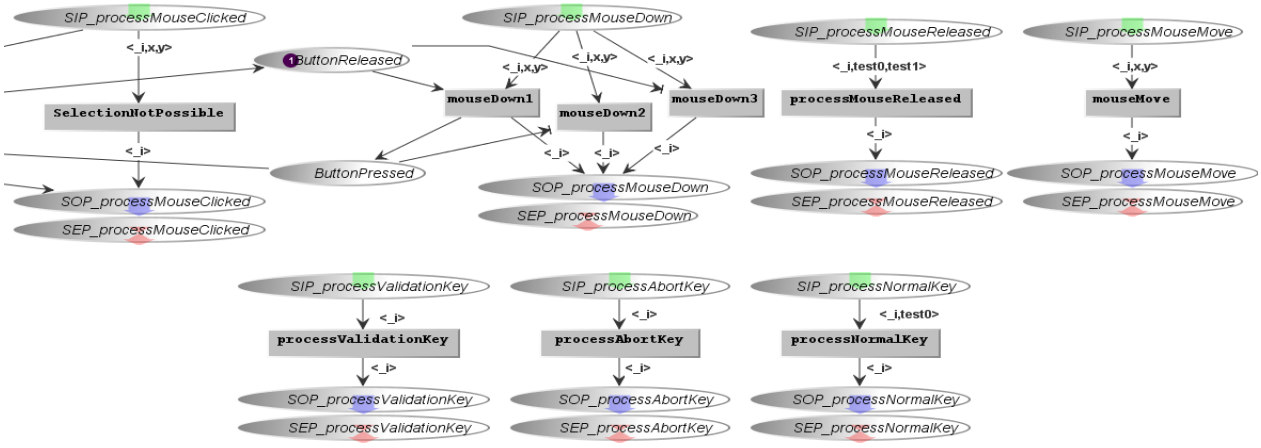


Figure IV.10 : Gestion des InputDeviceEvents

Le PicturePushButton utilise ici seuls les évènements mousePressed, MouseClicked et MouseReleased. L'évènement mouseMove n'a aucun impact sur son comportement de même que les évènements du clavier (validationKey, Abortkey, normalkey).

Les *RuntimeModifiableParameters* du PicturePushButton correspondent aux *setparameters* (A661_Visible, A661_Enable, A661_Styleset, A661_String, A661_PictureReference).

Les Figure IV.11 et Figure IV.12 présentent le fonctionnement de la mise à jour des paramètres *Visible* et *Enable*.

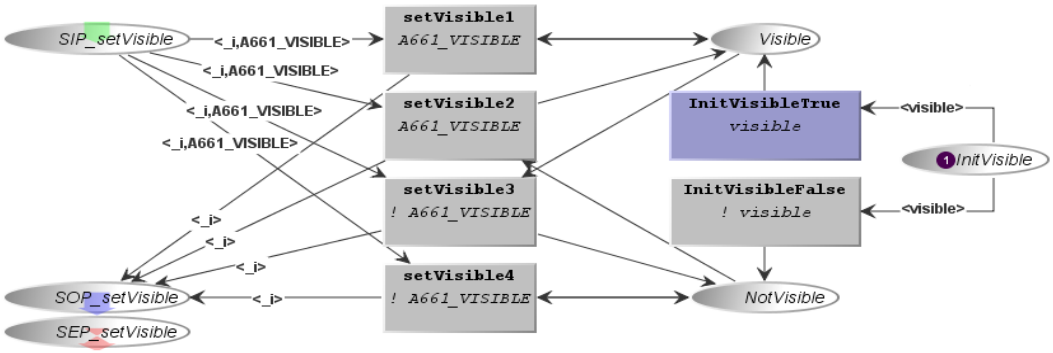


Figure IV.11 Gestion du paramètre Visible

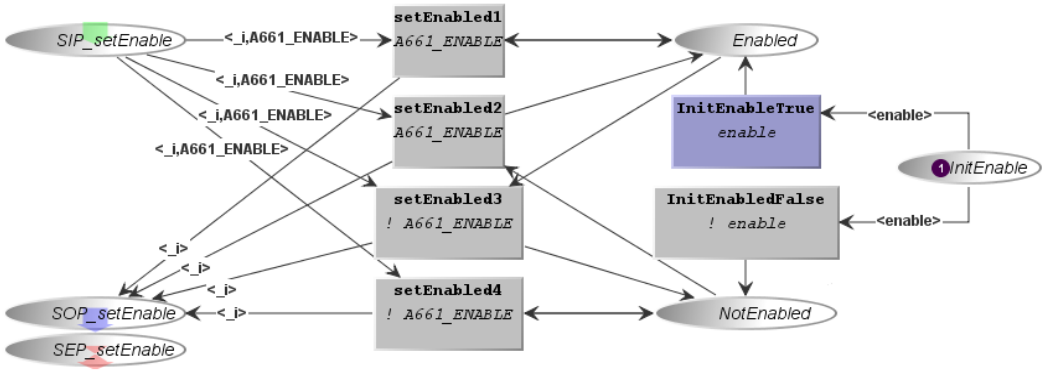


Figure IV.12 Gestion du paramètre Enable

La valeur booléenne à l'initialisation des paramètres *A661_visible* (respectivement *A661_enable*) est stockée par un jeton à la place *InitVisible* (respectivement *InitEnable*). Ce jeton est consommé selon sa valeur et mis dans la place *Visible* ou *Notvisible* (respectivement *Enabled* ou *NotEnabled*) à l'initialisation.

Ensuite lors de l'exécution, lorsqu'un objet appelle la méthode *setVisible* (respectivement .*setEnable*), un jeton contenant une valeur booléenne est déposé dans la place SIP correspondante. Ce jeton est ensuite traité par l'une des quatre transitions setVisible (respectivement. *setEnable*), afin de placer un jeton dans la place *Visible* (respectivement. *Enable*) ou *NotVisible* (respectivement. *NotEnable*) en fonction du cas.

- La 1ère transition *setvisible1* exprime le fait que si le PicturePushButton est déjà visible et qu'on reçoit un appel de méthode demandant de le mettre à *visible*. Alors on reste à *visible*.

- La 2ième transition *setvisible2* exprime le fait que si le PicturePushButton est dans *notvisible* et qu'on reçoit une demande de visibilité, alors on prend le jeton qui est dans *notvisible* pour le mettre dans *Visible*

C'est le même principe pour le cas *Notvisible*.

La Figure IV.13 présente la gestion du paramètre *LabelString* du PicturePushButton. A la réception de l'appel de méthode demandant la modification du paramètre, une vérification de la taille du label est effectuée. Si la taille n'est pas conforme, l'évènement d'erreur A661_ERROR_STRING_LENGTH est levé, si elle est conforme une mise à jour du label est effectuée.

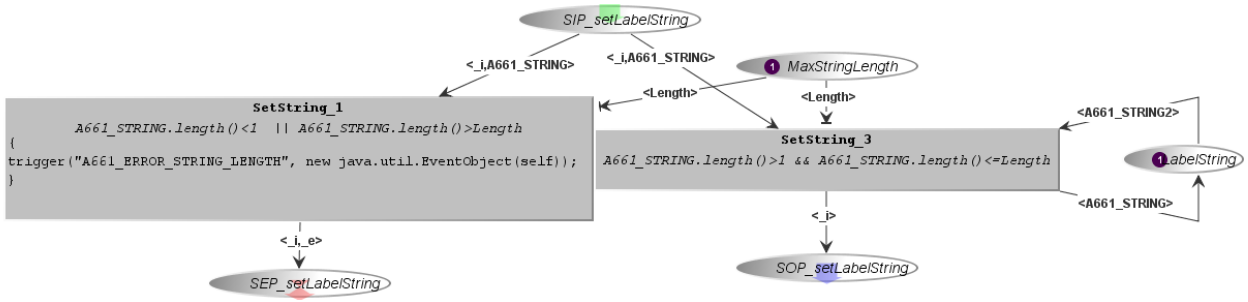


Figure IV.13 Gestion du paramètre LabelString

La Figure IV.14 présente la gestion du paramètre *styleset*. A la réception d'un nouveau style d'affichage du PicturePushButton, l'ancien est remplacé par le nouveau.

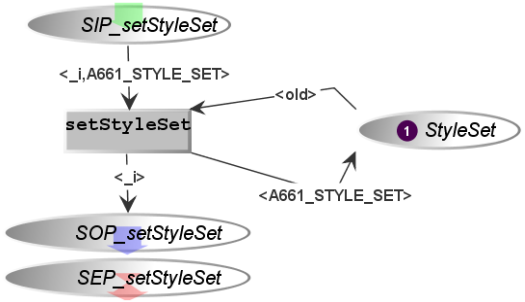


Figure IV.14 Gestion du paramètre Styleset

La Figure IV.15 présente la gestion du paramètre PictureReference. A la réception de l'appel de méthode demandant la mise à jour de l'image affichée sur le PicturePushButton, une vérification de l'existence de la référence associée à l'image est effectuée. Si la référence n'existe pas, l'évènement d'erreur A661_OUT_OF_RANGE_ERROR est levé, si elle existe la mise à jour de l'image est effectuée.

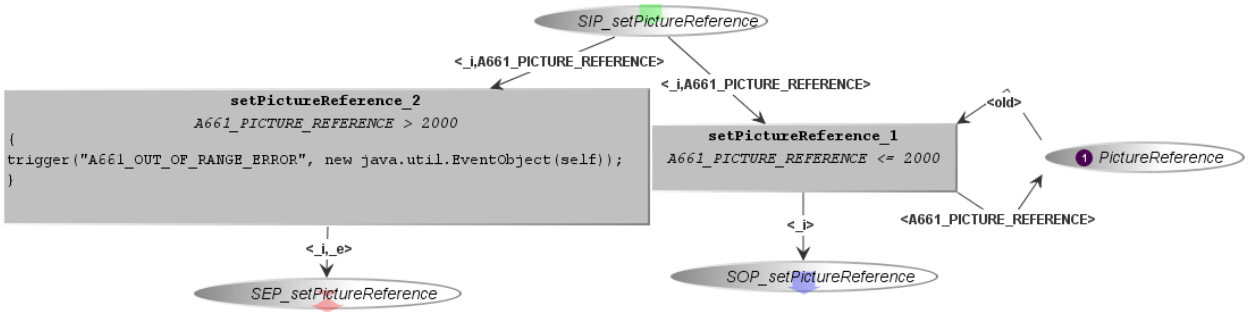


Figure IV.15 Gestion du paramètre PictureReference

La Figure IV.16 présente la gestion du paramètre *highlight* qui correspond au paramètre nécessaire au serveur pour gérer le rendu graphique du PicturePushButton. La figure montre aussi comment le changement du paramètre *highlight* est influencé par la valeur du paramètre *Enable*. En effet, les arcs de tests partant des places *Enabled* et *NotEnabled* contraignent l'aiguillage de l'appel de la méthode *setHighlighted*, en faisant franchir la transition *setHighlighted1* s'il y a un jeton dans la place *Enabled* (à ce moment-là, la valeur contenue dans la place *highlighted* change), ou en faisant franchir la transition *setHighlighted2* s'il y a un jeton dans la place *NotEnabled* (à ce moment-là, la valeur contenue dans la place *highlighted* est fixé à faux).

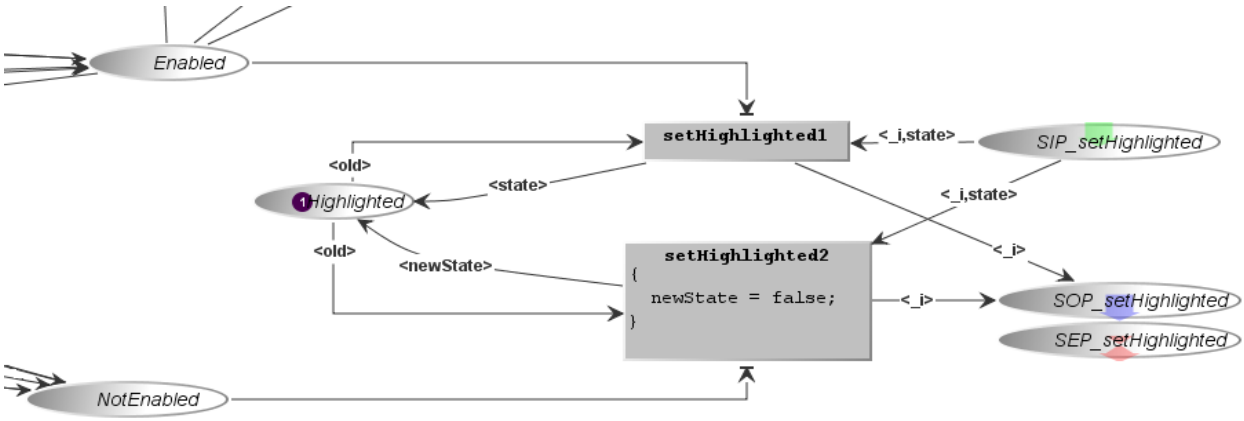


Figure IV.16 Gestion de l'état highlighted en fonction de l'Enable

La Figure IV.17 présente la gestion de l'évènement A661_EVT_SELECTION. L'assertion définie pour le pushbutton au Chapitre I, spécifiait que l'évènement ne devait être envoyé que si le widget était visible et enable et qu'il ait reçu un évènement de clic.

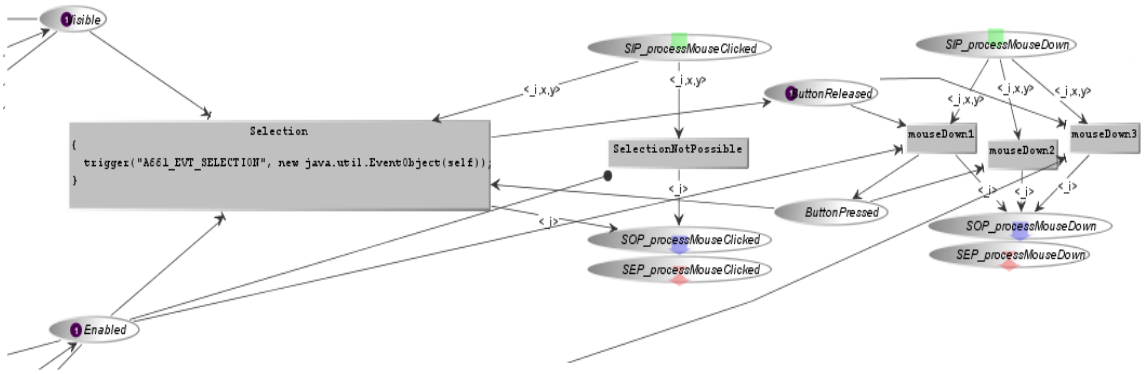


Figure IV.17 Gestion de l'envoi de l'Event du PicturePushButton

La Figure IV.19 présente le modèle de comportement global du PicturePushButton, la gestion des différents *setparameters* y est décrite ainsi que l'assertion d'envoi de l'évènement A661_EVT_SELECTION.

Le rendu graphique du widget est effectué en connectant les places des *setparameter* du widget à un outil dédié de gestion du rendu graphique. La Figure IV**.**18 montre quelques exemples d'images de rendu graphique d'un PicturePushButton.

PicturePushButton à l'état Enable=true

PicturePushButton lors du passage de la souris dessus (focus)

PicturePushButton à l'état Enable=false

Figure IV.18 Rendu graphique PicturePushButton

La gestion des paramètres de *Visible*, *Enable*, *LabelString*, et *styleset* est identique pour les widgets possédant ces paramètres.

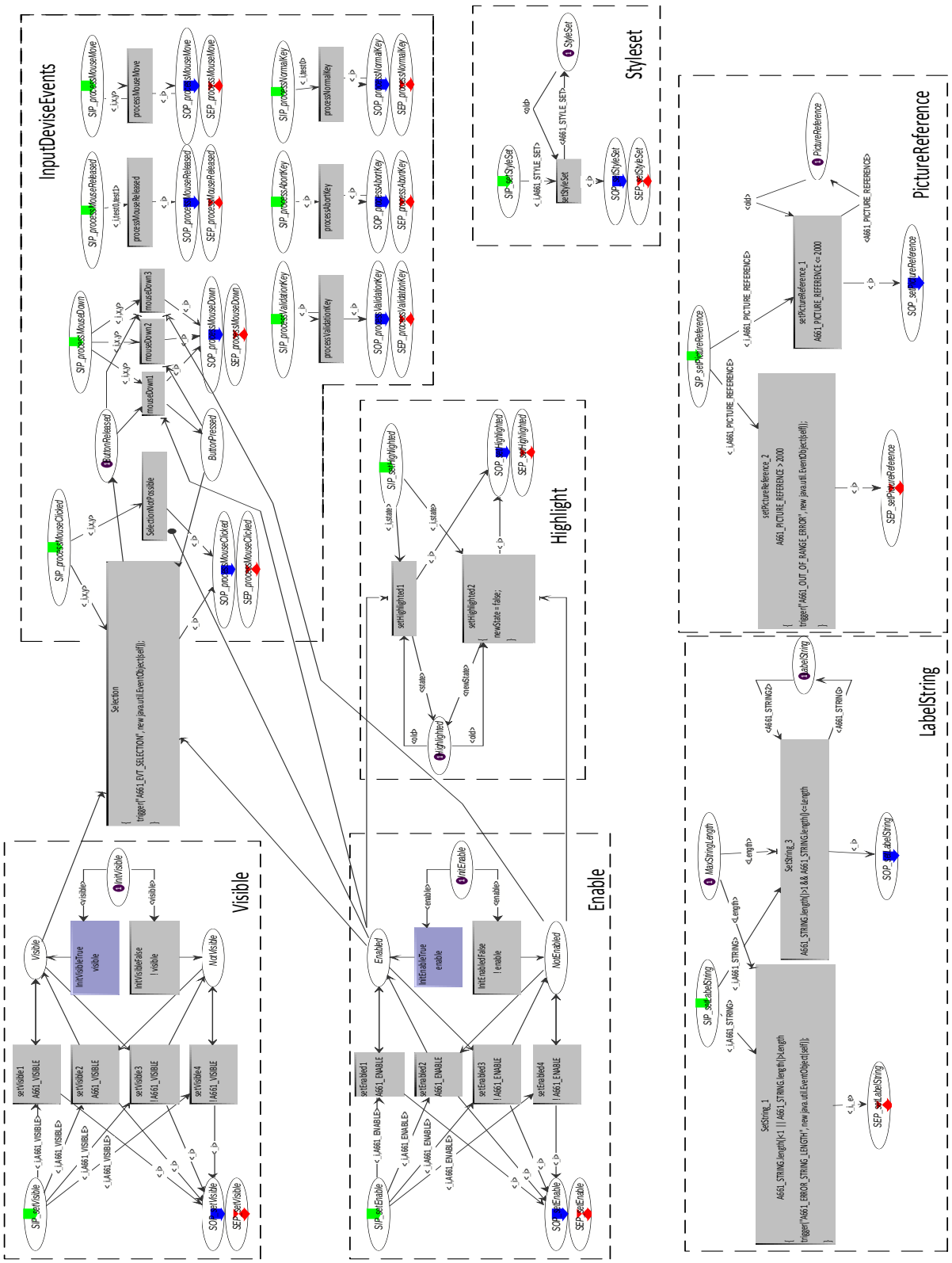


Figure IV.19 Comportement global du PicturePushButton

IV.2.2.2 *Modélisation du comportement de l'EditBoxNumeric*

L'EditBoxNumeric est un widget de saisie de donnée, son comportement est assez complexe, car il gère tous les évènements des périphériques d'entrées (clavier/dispositif de pointage), notifie plusieurs types d'évènement à l'UA, et peut recevoir jusqu'à treize *setparameters* (Figure IV.20).

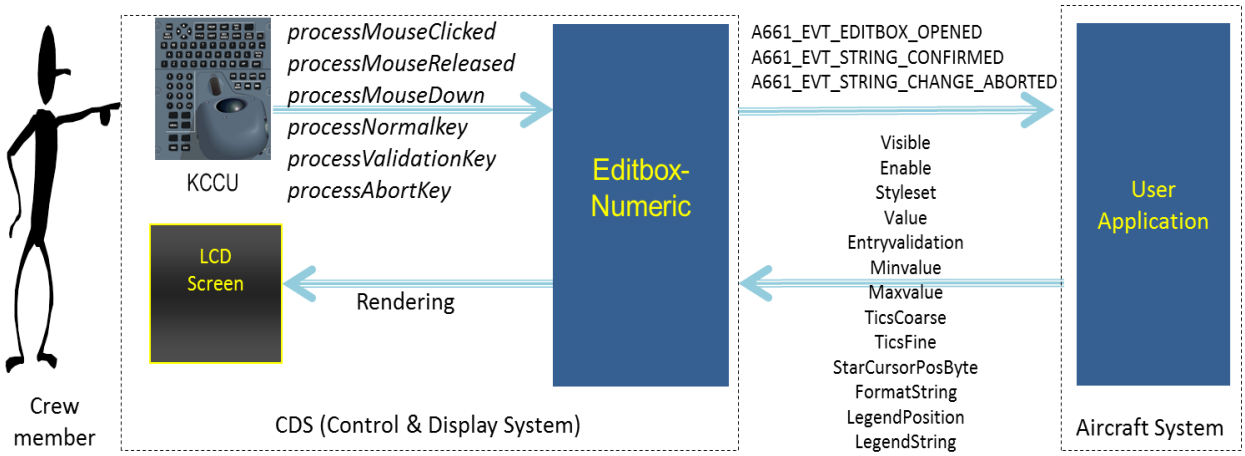


Figure IV.20 Interface EditBoxNumeric

Sa représentation ICompoNet est représentée à la Figure IV.21.

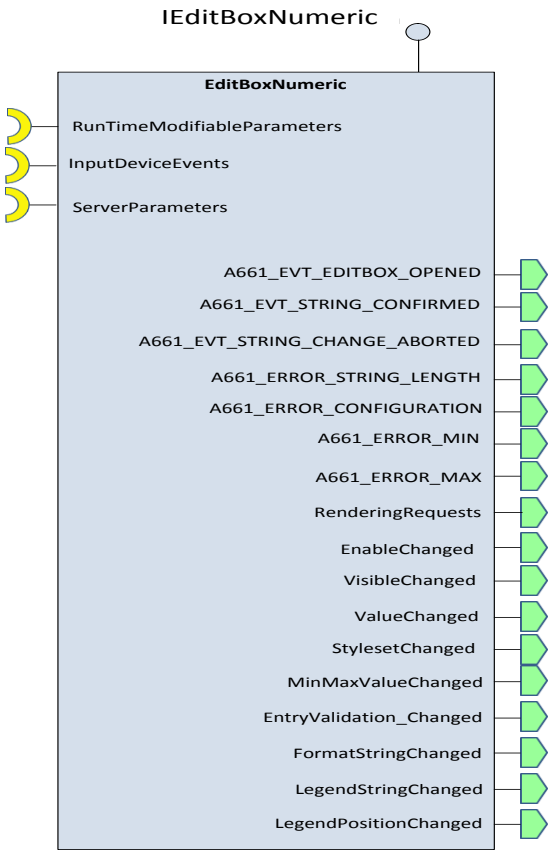


Figure IV.21 Modèle ICompoNet de l'EditBoxNumeric

Comme pour le PicturePushButton, les *InputDeviceEvents* correspondent aux évènements venant des périphériques d'entrées, à la différence que l'EditBoxNumeric utilise l'ensemble des évènements du dispositif de pointage et du clavier, dont la représentation ICO est donnée à la Figure IV.10.

L'entrée *serverparameters* qui est l'ensemble des paramètres nécessaires au serveur pour gérer le rendu graphique du widget intègre ici plus d'éléments que le paramètre *highlight*. Etant un élément de saisie de donnée, le rendu graphique du widget varie selon les états dans lequel le widget peut se trouver avant, pendant et après la saisie de la donnée. Ce sont ces états que nous décrivons ci-dessous.

L'EditBoxNumeric comporte trois états :

- *Idle* : c'est l'état durant lequel l'EditBoxNumeric est visible et enable mais ne reçoit aucune action de l'utilisateur.
- *Editing* : C'est l'état durant lequel l'utilisateur est entrain de saisir sa donnée dans l'EditBoxNumeric
- *WaitingForUA* : c'est l'état durant lequel l'EditBoxNumeric attend la validation de l'UA sur la valeur saisie.

De ces états trois scénarios de comportement sont possibles :

1. L'EditBoxNumeric est en mode *Idle*, l'utilisateur clique sur l'EditBoxNumeric qui envoie une notification à l'UA en envoyant l'évènement A661_EVT_EDITBOX_OPENED. L'utilisateur saisie sa donnée (état *editing*), durant cette période l'EditBoxNumeric reçoit les évènements clavier (*processNormalKey*). A la fin de sa saisie, l'utilisateur valide sa saisie en appuyant sur entrée (*processvalidationkey*), une notification est alors envoyée à l'UA avec la valeur saisie au travers de l'évènement A661_EVT_STRING_CONFIRMED. Si l'UA valide la saisie, la donnée saisie devient celle active sur l'EditBoxNumeric.
2. L'autre scénario possible est qu'au lieu de valider sa saisie, l'utilisateur l'annule en appuyant sur le bouton Echap du clavier. Dans ce cas l'évènement A661_EVT_STRING_CHANGE_ABORTED est notifié à l'UA et la valeur qui était affichée avant la saisie est remise active sur l'EditBoxNumeric.
3. Le dernier scénario possible est que l'utilisateur ne valide jamais la donnée saisie. Dans ce cas au bout d'un temps défini l'EditBoxNumeric revient automatiquement dans l'état *Idle* et considère qu'il y a abandon de saisie en notifiant à l'UA par l'envoi de l'évènement A661_STRING_CHANGE_ABORTED.

Pour éviter que l'utilisateur interagisse avec d'autres widgets lors de la saisie dans l'EditBoxNumeric, on fait disparaitre le curseur. Durant cette période, le serveur est configuré pour que tous les évènements sur les périphériques soient envoyés qu'au widget EditBoxNumeric actif, on parle ici de mode « *caging* ».

Le Réseau de Petri de la Figure IV.22, présente la gestion de l'état *Idle*. Si le widget n'est pas *Enabled* et qu'on reçoit l'évènement de clic alors l'évènement n'est pas traité (transition *processMouseClicked3*). S'il n'y a pas de jeton dans la place *Idle*, c'est-à-dire que le widget est dans un autre état (*Editing* ou *waitingforUA*) l'évènement de clic n'est pas non plus traité (transition ProcessMousclilcked2). Si par contre le widget est *visible*, *Enabled* et est dans l'état *Idle* et qu'il reçoit un clic, on passe en mode *caging* et l'évènement A661_EDITBOX_OPENED est notifié à l'UA.

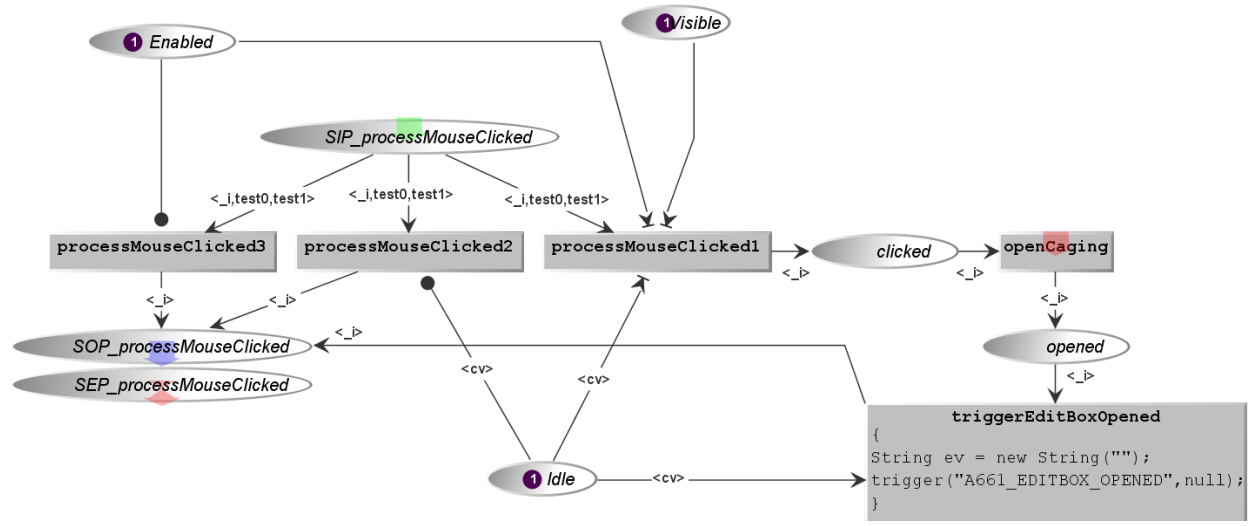


Figure IV.22 Gestion état Idle

La Figure IV.23 est un zoom sur les liens entre les états de l'EditBoxNumeric (*Idle*, *Editing*, *waitingforUA*) tels que décrit dans les scénarios ci-dessus.

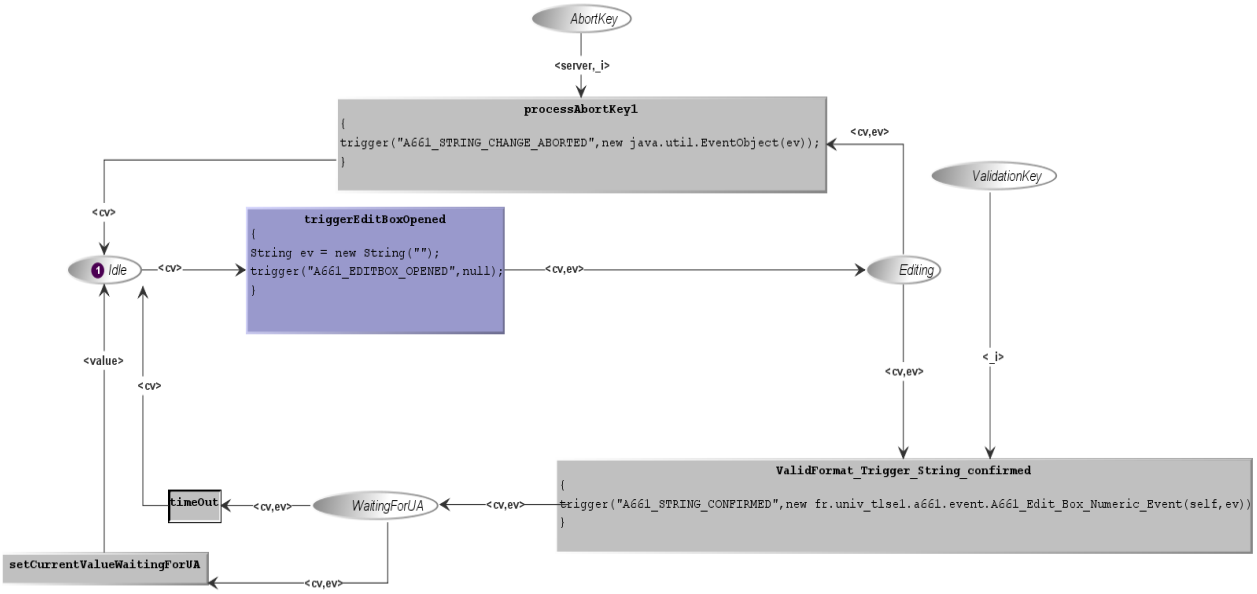
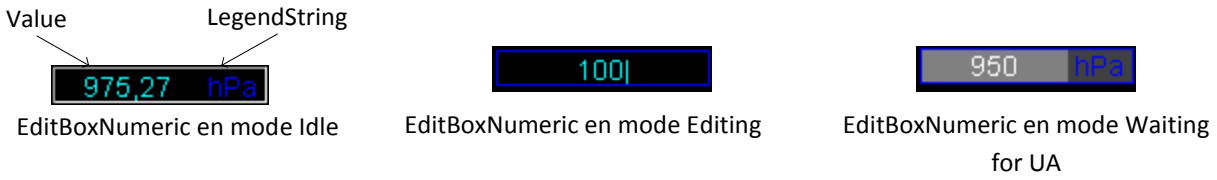


Figure IV.23 Etat editing de l'EditBoxNumeric

La Figure IV.25 présente le modèle de comportement de l'EditBoxNumeric. Pour des raisons de présentation le détail des gestions de paramètre *Enable*, *visible*, *styleset* n'ont pas été mis, sachant qu'ils sont identiques à ceux du PicturePushButton. La figure est découpée selon les états du widget. Le rectangle *Etat*_Idle a été expliqué à la Figure IV.22. Le rectangle *Etat_editing* est le comportement du widget en mode édition, les évènements claviers (*ProcessAbortKey*, *ProcessvalidationKey*, *processNormalKey*) y sont traités, ceux-ci impactant la fermeture du mode caging. Une vérification du format de saisie y est aussi effectué (place formatString). Le rectangle *Etat_waitingForUA* traite l'envoi de l'évènement

A661_EVT_STRING_CONFIRMED, attend la validation de l'UA via les paramètres *entryvalidation*, ensuite une vérification de la plage de valeur est réalisée avant l'affichage effectif de la valeur saisie.

Le rendu graphique correspondant de chaque mode de l'EditBoxNumeric est représenté à la Figure IV.24.



EditBoxNumeric en mode Idle | EditBoxNumeric en mode Editing | EditBoxNumeric en mode Waiting for UA

Figure IV.24 Rendu Graphique EditBoxNumeric

On peut voir au travers de ces différents modèles de widget (pushbutton et EditBoxNumeric) que le comportement d'un widget peut être assez complexe. Leur utilisation dans un environnement critique, nécessite donc d'avoir une description formelle et précise. De plus les différentes capacités d'analyse et de simulation de PetShop nous ont permis de tester et de s'assurer que les modèles de comportement des widgets correspondaient bien aux besoins.

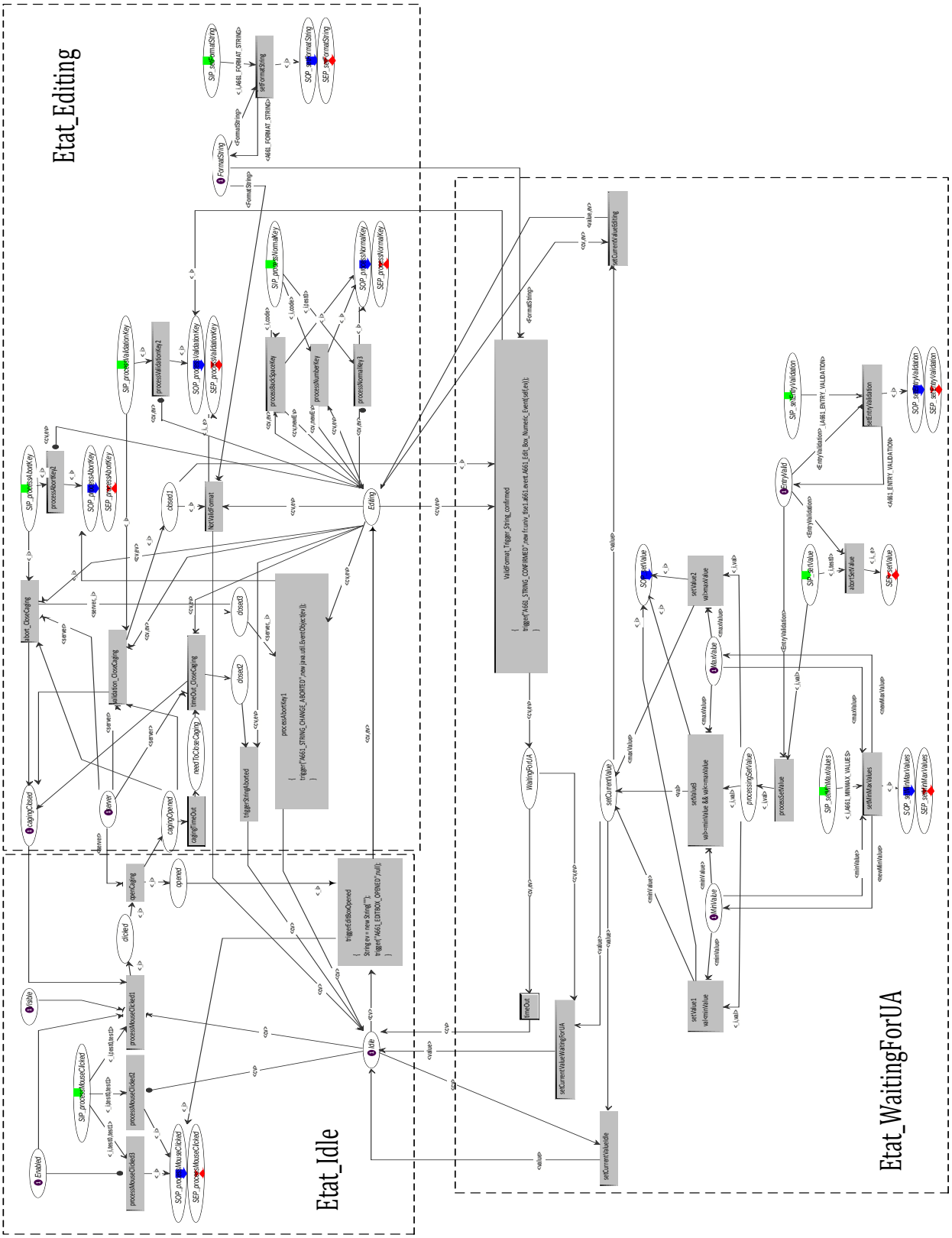


Figure IV.25 Modèle global de l'EditBoxNumeric

IV.3 MODELISATION DU COMPORTEMENT DE L'UA FCU

La seconde partie de notre approche zéro défaut est la description précise du comportement de l'application cliente (UA) du système avion.

Tel que nous l'avons vu, l'ARINC 661 utilise un concept de fenêtrage, avec une organisation hiérarchique en Window → Layer → Widget. La layer étant la plus haute entité du CDS connecté à l'UA offrant l'espace d'affichage de ses widgets.

Une layer peut être organisée de diverses manières. De façon générale elle est découpée en zone d'interaction. Ces zones sont délimitées par des widgets de regroupements, et les objets de regroupements contiennent divers widgets (action, affichage).

Figure IV.26 montre la structure de la layer de l'UA FCU. C'est l'UA qui gère la partie fonctionnelle de l'interface graphique et coordonne ainsi les liens fonctionnels entre les widgets de sa layer pour avoir un affichage cohérent.

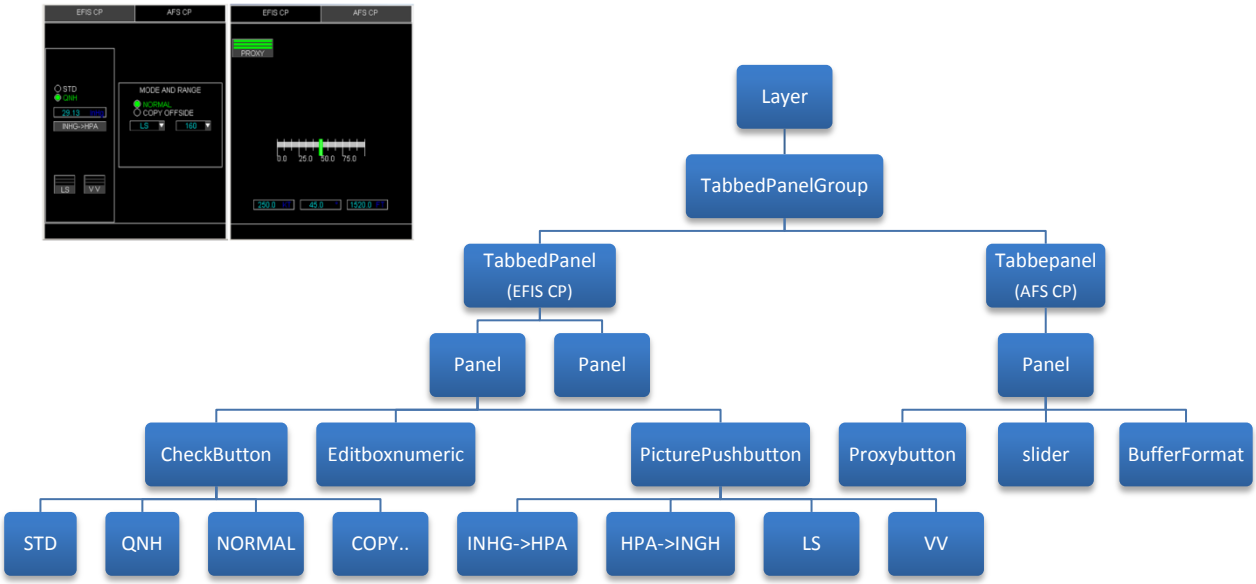


Figure IV.26 Organisation de la layer de l'UA FCU

Nous allons présenter ci-dessous le comportement de chaque bloc de l'UA FCU. Pour des raisons de mise en page, nous découperons ici le modèle de l'UA en sous-parties : La page EFIS CP et la page AFS CP.

IV.3.1 **Modélisation du comportement de la Page EFIS CP**

La page EFIS CP comporte les widgets suivants, CheckButton, EditBoxNumeric, PicturePushButton. Certains de ces widgets sont gérés de façon indépendante par l'UA, d'autres ont des fonctionnalités liées au niveau de l'UA. Nous présentons succinctement la gestion fonctionnelle de ces widgets par l'UA.

IV.3.1.1 *Gestion exclusive du groupe de CheckButton*

Dans le couple de CheckButton (STD /QNH) de la page EFIS CP, un seul est sélectionnable à la fois. Cette dépendance fonctionnelle dans la gestion des CheckButton au niveau UA est représentée par la Figure IV.27.

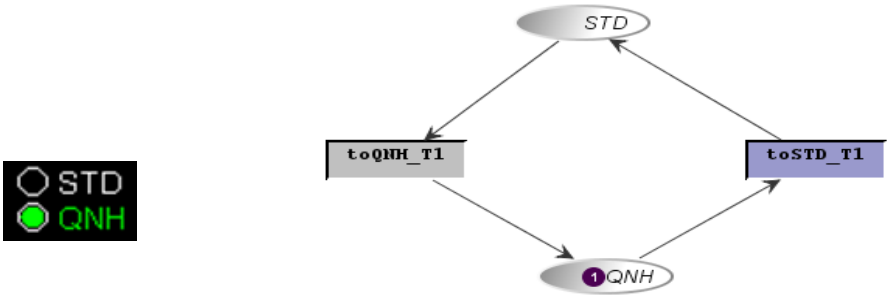


Figure IV.27 Gestion du couple de CheckButton (SDT/QNH) par l'UA FCU

Lorsqu'il y a un jeton dans la place QNH, cela veut dire que c'est le CheckButton QNH qui est sélectionné sur l'interface FCU. Les transitions représentent le changement de sélection.

La fonction d'activation indiquant comment les évènements sur le système d'entrée seront reçus par les transitions du réseau de Petri ci-dessus est représentée par le Tableau IV.4.

Action	**Evènement envoyé**	**Méthode d'activation**
Clic checkbutton STD	toSTD	setEnabled ()
Click checkbutton QNH	toQNH	setEnabled ()

Tableau IV.4 Fonction d'activation des CheckButton STD/QNH sous forme textuelle

La fonction de rendu qui maintient la consistance entre l'état du Réseau de Petri et son apparence est représentée par le Tableau III.6

Place	**Evènement**	**Méthode de rendu**
QNH	Réinitialisation	setInnerStateSelect ()
QNH	Jeton_entré	setInnerStateSelect ()
STD	Jeton_entré	setInnerStateSelect ()

Tableau IV.5 Fonction de rendu des CheckButton STD/QNH sous forme textuelle

Ces fonctions d'activation et rendu sont aussi exprimés en réseau de Petri.

Ces couples de CheckButton peuvent aussi être contenus dans un widget de type radioBox dont la fonctionnalité est de gérer l'exclusivité de sélection.

IV.3.1.2 *Gestion des PicturePushButton*

Le modèle de fonctionnement des PicturePushButton au niveau UA est représenté par le Réseau de Petri de la Figure IV.28 .

L'exemple est donné ici pour le pushbutton LS. Les places LS_OFF et LS_ON représentent respectivement les cas où le pushbutton est désactivé ou activé.

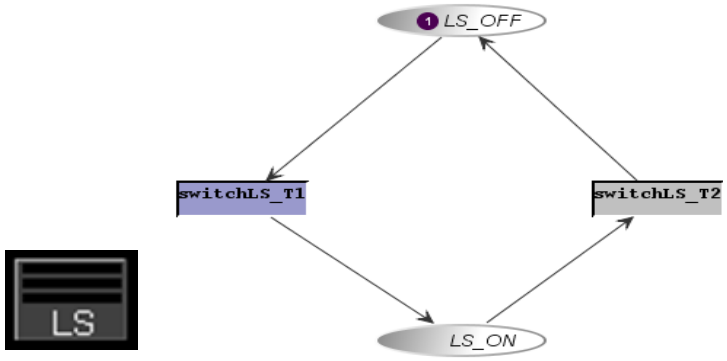


Figure IV.28 Gestion d'un PicturePushButton par l'UA

Les fonctions d'activation et de rendu correspondant du pushbutton sont représentées par les Tableau IV.6 et Tableau IV.7 .

Action	**Évènement envoyé**	**Method d'activation**
Clic sur le bouton	switchLS	setEnable ()

Tableau IV.6 Fonction d'activation PicturePushButton LS

Place	**Evènement**	**Méthode de rendu**
LS_OFF	Réinitialisation	setLabelString (« LS »)
LS_OFF	Jeton_entré	setPicturereference (« a »)
LS_ON	Jeton_entré	setPicturereference (« b »)

Tableau IV.7 Fonction de rendu PicturePushButton LS

IV.3.1.3 *Gestion des ComboBox*

L'étude de cas FCU comporte deux ComboBox, un pour la sélection d'un mode d'affichage du plan de vol et un second pour la sélection du range. Le comportement d'une ComboBox au niveau UA est assez simple, quand il y a sélection d'une donnée sur la liste, cette donnée est reçu par l'UA (transition *switchRange*) qui stocke la donnée (place *Range*). (Figure IV.29)

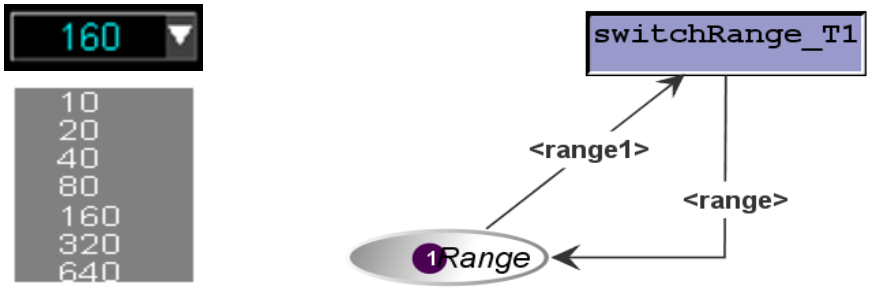


Figure IV.29 Modèle ComboBox

IV.3.1.4 *Gestion des couples EditBoxNumeric/PicturePushButton*

L'UA FCU permet l'entrée des valeurs de pression en Hecto pascal et en pouce mercure et la conversion de ces valeurs d'une unité à l'autre. Cette conversion est réalisée avec deux pushbutton et deux EditBoxNumeric superposés (Figure IV.30).

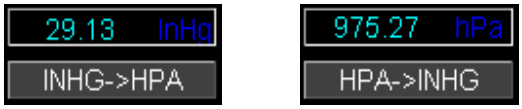


Figure IV.30 Images de l'EditBoxNumeric et du PushButton

Le comportement des couples EditBoxNumeric et pushbutton sont liés, contrairement aux autres widgets de l'interface qui ont des comportements indépendants. Le modèle ICO de leur comportement est représenté à la Figure IV.31. Ce comportement peut être amélioré en utilisant un ToggleButton à la place des PicturePushButton.

Les deux EditBoxNumeric affichent la valeur courante et permettent de saisir une nouvelle valeur (transition *ChangeInHg*, *ChangeHpa*) Lors de la saisie d'une nouvelle valeur, cette valeur sera comparée aux bornes limites et si elle n'est pas comprise entre les bornes, elle sera fixée au niveau de la borne max ou min correspondante. (Transition CorrectInHg 1, 2 3).

Les deux PicturePushButton permettent quant à eux de changer d'unité (par l'intermédiaire des transitions toINHG et toHPa) et donc de choisir quelle EditBoxNumeric et quel PicturePushButton afficher (InhgSeleced ou hPaSelelcted).

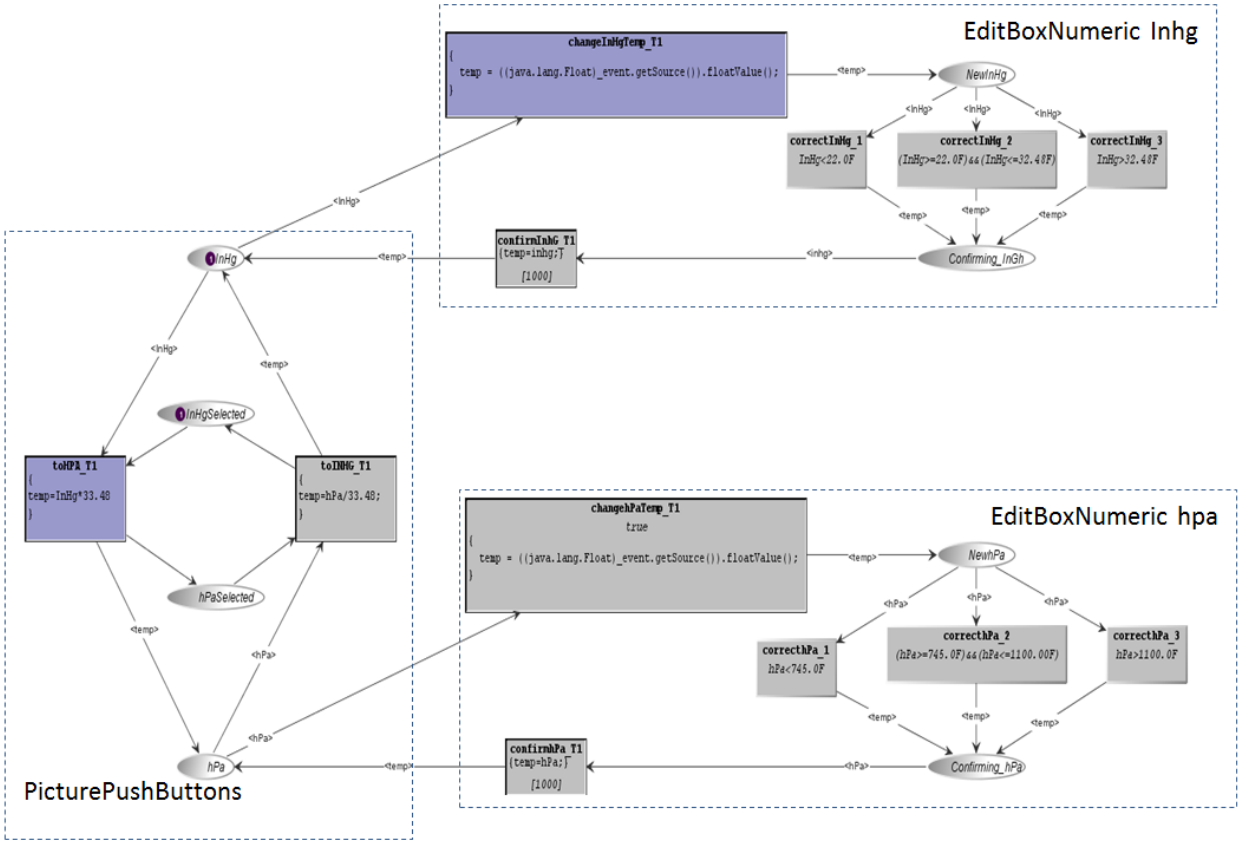


Figure IV.31 Modèle de conversion EditBoxNumeric/Pushbutton

IV.3.2 **Gestion de la page AFS CP**

La page AFS CP comporte les widgets suivants, ProxyButton, Slider, WatchdogContainer, BufferFormat, EditBoxNumeric, PushButton. Nous présentons ci-dessous la gestion de quelques-uns de ces widgets au niveau de l'UA.

IV.3.2.1 *Gestion du ProxyButton*

Le bouton physique auquel est lié le ProxyButton dans notre étude de cas est le bouton « P » du clavier et le bouton virtuel est le pushbutton « Proxy » sur la page AFS. Le rôle que nous lui avons donné est de mettre la page AFS à ON ou OFF selon qu'on appui sur le bouton physique ou sur le pushbutton Proxy.

Le ProxyButton n'a aucun comportement au niveau UA, c'est plutôt le comportement du pushbutton auquel il est associé qui est représenté, et ce pushbutton est géré de la même manière que le PicturePushButton.

IV.3.2.2 *Gestion du BufferFormat*

Le rôle du BufferFormat dans notre étude de cas est de recevoir de l'UA une mise à jour périodique de trois paramètres (Speed, mag et hdg). C'est la transition « ChangesValues » du réseau de Petri de la Figure IV.32 qui fournit les nouvelles valeurs, avec une périodicité de 800 ms.

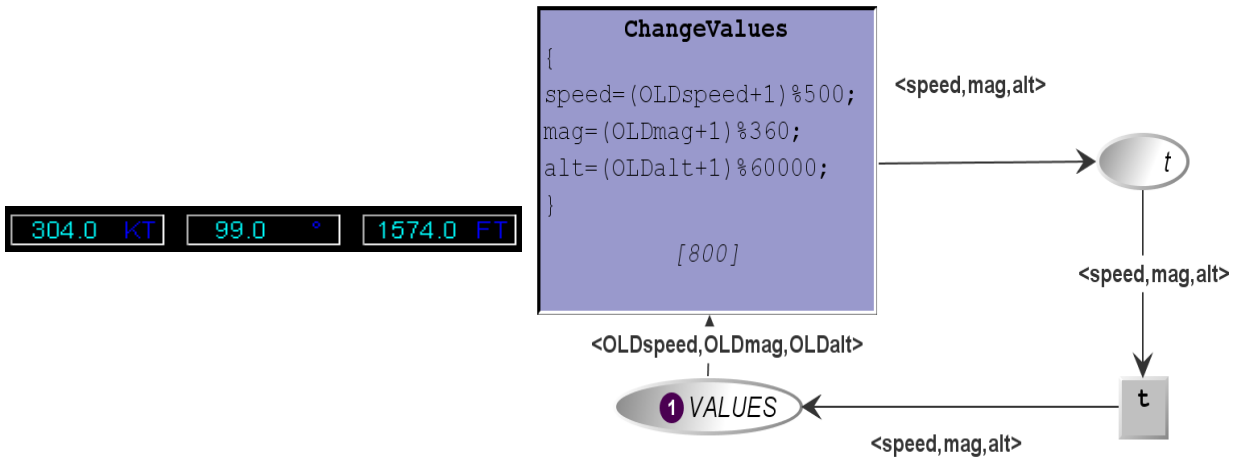


Figure IV.32 Modèle BufferFormat

IV.3.2.3 *Gestion du Slider*

A chaque changement de valeur du Slider, l'UA va recevoir la nouvelle valeur, franchir la transition *ChangeSliderValue* et changer la valeur stockée dans l'UA (place *Slider_value*).

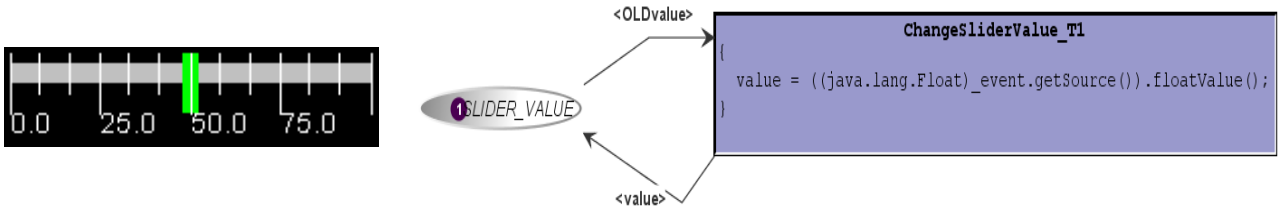


Figure IV.33 Modèle du Slider

L'UA FCU est ainsi le résultat de la fusion de tous ces réseaux de Petri.

IV.4 ANALYSE DES MODELES DE TACHE

Cette section est consacrée à l'analyse de l'activité de l'utilisateur sur un système interactif résultant d'une conception zéro défaut. Cette activité correspond à celle présente actuellement sur le CDS sur les applications développées en ARINC 661 et qui sont non critiques. En effet la conception zéro défaut apporte une amélioration dans le développement des applications, dans l'élimination des fautes, mais ne change pas la manière dont les données sont perçues par l'utilisateur, ni comment l'utilisateur va interagir avec le système. L'activité de l'utilisateur correspond ici à son activité nominale.

Comme précisé au Chapitre I, nous analysons l'activité de l'utilisateur en réalisant les tâches selon deux aspects qui sont : les interactions en sortie (lecture d'une donnée), et les interactions en entrée (saisie d'une donnée).

Pour mieux expliquer ces modèles de tâche, nous présentons d'abord l'architecture du système, en nous focalisant sur les éléments sur lesquels l'utilisateur peut interagir (périphériques d'entrées et sorties). La communication entre le CDS et les systèmes avions est simplifiée ainsi que les mécanismes internes de communication dans le CDS, ceux-ci ayant déjà été expliqués.

IV.4.1.1 *Interaction en sortie*

Pour réaliser la lecture d'une donnée, l'architecture du CDS et le modèle de tâche sont présentés aux Figure IV.34 et Figure IV.35.

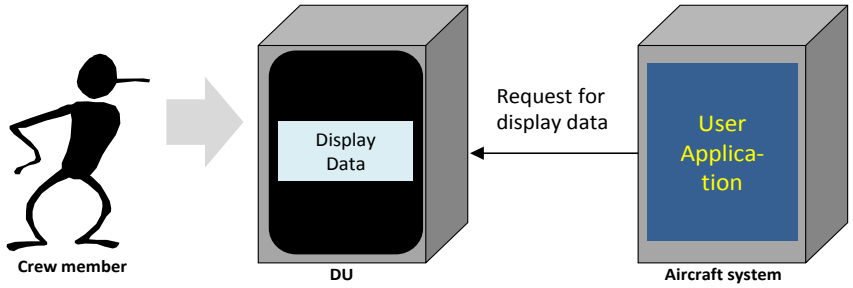


Figure IV.34 Architecture du système pour l'affichage de donnée cas zéro défaut

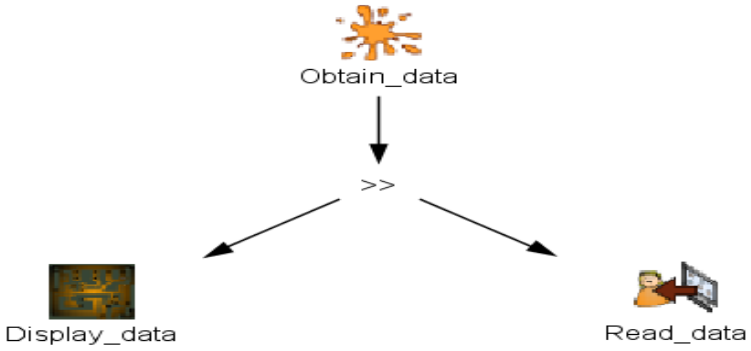


Figure IV.35 Modèle de tâche de l'interaction en sortie cas zéro défaut

Au niveau de l'architecture, l'application avion envoi une requête d'affichage (exemple un label) et celle-ci est traité par le calculateur du système d'affichage et de contrôle (DU) qui affiche la valeur.

Au niveau du modèle de tâche, après l'affichage par la DU (*Display_Data*), l'utilisateur peut lire la donnée (*Read_data*). Sa tâche est assez simple, il n'a qu'une action à réaliser qui est la lecture de l'information.

IV.4.1.2 *Interaction en entrée*

Pour réaliser la saisie d'une donnée. L'architecture système est tout aussi simple (Figure IV.36). L'utilisateur se sert du clavier et du dispositif de pointage (KCCU) pour saisir sa donnée, une notification de saisie est ensuite envoyée à l'application avion.

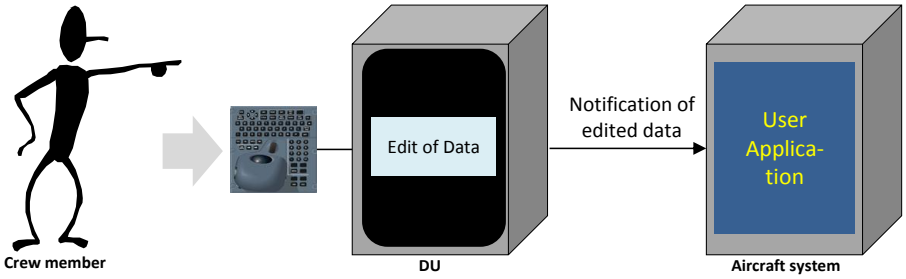


Figure IV.36 Architecture du système pour la saisie de données cas zéro défaut

La Figure IV.37 présente le modèle de tâche de la saisie de donnée.

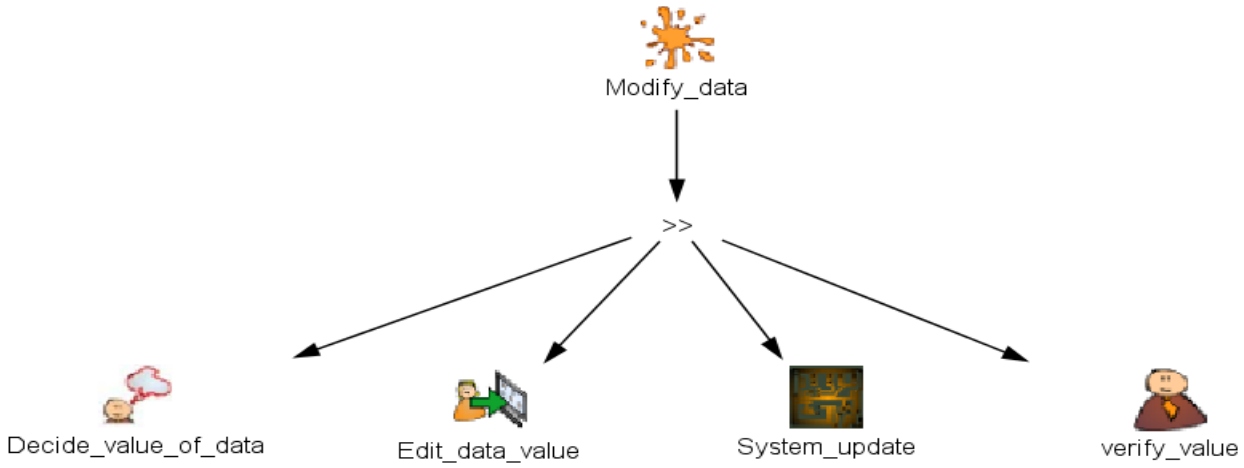


Figure IV.37 Modèle de tâche d'entrée d'une donnée cas zéro défaut

Après avoir décidé de la donnée à saisir (*Decide_value_of_data*) qui est une tâche cognitive. L'utilisateur réalise l'édition effective de cette donnée sur le KCCU (*Edit_data_value*), le système valide la donnée et la met effectivement à jour sur l'interface (*System_update*), c'est l'état *waitingforUA* que nous avons vu dans la description du comportement de l'EditBoxNumeric au Chapitre I. Suite à cette mise à jour par le système, l'utilisateur vérifie que la donnée effective affichée est bien celle qu'il a saisie. On distingue au total une tâche interactive et deux tâches de l'utilisateur indépendantes du système (cognitive et perceptible).

IV.5 CONCLUSION

Nous avons réalisé dans ce chapitre l'application de la conception zéro défaut, ceci en réalisant la modélisation des widgets et de l'UA d'un système interactif. Nous avons constitué une étude de cas qui représentait au mieux l'ensemble des widgets du standard ARINC 661 et qui est basée sur une application avionique réelle.

L'utilisation du formalisme ICO a permis de spécifier et modéliser de façon précise l'ensemble du comportement des widgets et l'application FCU. On y a décrit :

- Les états des widgets en fonction des évènements de l'utilisateur sur les périphériques d'entrées et des commandes venant de l'application avion (UA).
- L'ensemble des états dans lequel l'application peut se trouver, pour chacun de ces états l'ensemble des évènements auxquels elle peut réagir.
- En fonction de l'état et de l'évènement reçu les opérations à effectuer ainsi que la représentation graphique de l'application présentée à l'utilisateur.

La modélisation de l'étude de cas du FCU en langage formel ICO au travers de l'outil PetShop permet d'avoir une description formelle au travers des propriétés des réseaux de Petri et précise du comportement des widgets et des applications clientes (UA).

A titre indicatif, le Tableau IV.8 présente la taille des modèles en termes de nombre de place et de transitions.

Classe typologie	Widget	Places	Transitions
Action sur l'UA	ComboBox	78	45
	PicturePushButton	56	30
	Checkbutton	60	37
Action sur widget	TabbedPanelGroup	53	20
Entrée informations	EditBoxNumeric	92	55
	Slider	55	32
Affichage	Label	36	15
Objet de regroupement	TabbedPanel	25	12
	Panel	22	15
Automatisation	ProxyButton	23	16
	BufferFormat	33	34
	WatchdogContainer	25	15

	Page AFS	7	7
	Page EFIS	18	22

Tableau IV.8 Taille Etude de cas

On constate sur le tableau que le widget EditBoxNumeric est celui dont la taille des modèles est la plus grande. C'est le widget le plus complexe que nous avons modélisé, il traite des évènements clavier/dispositif de pointage et possède plusieurs états. Un widget qui lui est semblable est la ComboBox à la différence qu'au lieu de saisir des données, on en sélectionne dans une liste. De façon globale les widgets de la grande classe action ont des tailles assez importantes, de par leur caractère interactif, et les autres ont des tailles plutôt moyennes, ceux-ci étant très peu en interaction avec l'utilisateur.

Au niveau de l'application cliente (UA), les widgets n'ont pas beaucoup de lien fonctionnel entre eux, c'est pourquoi on a des modèles assez simples.

Pour des besoins de prototypage et de tests, la partie rendue graphique de l'étude de cas a été réalisé en java via l'outil Netbeans. Ce prototypage a permis de réaliser les tests d'interaction et de validé les modèles de comportement des widgets et de l'UA.

Cette étude de cas illustre la faisabilité de la description formelle et précise du système interactif grâce à la technique de description formelle ICO.

Nous avons enfin réalisé les modèles de tâche de l'utilisateur dans les cas d'une interaction en sortie et d'une interaction en entrée. Les tâches de l'utilisateur sont ici assez simples et correspondent à son activité nominale sur le système. Il sera plus intéressant de comparer ces modèles à ceux de l'approche tolérance aux fautes.

Chapitre V. Application des diverses approches de tolérance aux fautes et étude de leur impact respectif sur l'utilisabilité

Résumé du chapitre

L'objectif de ce chapitre est de présenter successivement l'application des trois options de la tolérance aux fautes que sont la tolérance aux fautes reposant sur l'utilisateur, la tolérance aux fautes reposant sur le système et l'approche mixte de la tolérance aux fautes, reposant à la fois sur l'utilisateur et sur le système.

Nous détaillons plus particulièrement l'option tolérance aux fautes reposant sur le système via l'utilisation du formalisme ICO. Cette option consiste à concevoir des widgets autotestables, ce qui est une solution nouvelle dans le domaine de l'interaction homme machine.

Les options de tolérance aux fautes reposant sur l'utilisateur ou mixte, sont basées soit sur la description précise des widgets réalisée suivant l'approche conception zéro défaut, soit en exploitant les widgets autotestables. C'est pourquoi nous détaillons dans ce chapitre que l'architecture générale (matérielle et logicielle) du système pour ces deux options sans entrer dans les détails de modélisation au travers du formalisme ICO. Par ailleurs nos travaux concernent plus particulièrement la détection des erreurs et non le recouvrement de l'erreur.

Les architectures résultant de ces options de tolérance aux fautes ont un impact sur l'utilisabilité. Pour évaluer cet impact, nous modélisons les tâches des opérateurs avec l'architecture tolérante aux fautes et sans architecture tolérante aux fautes, ensuite comparons les modèles de tâche obtenus.

V.1 APPLICATION DE L'APPROCHE TOLERANCE AUX FAUTES REPOSANT SUR L'UTILISATEUR ET ANALSE DE L'IMPACT SUR L'UTILISABILITE

Le CDS (control and Display System) est composé de plusieurs ressources d'affichage et de contrôle (8 DU et 2 KCCU sur la configuration A380, et 6 DU sur une configuration A320). Les DU sont des calculateurs possédants des ressources logicielles pour le traitement des données venant de différentes UA.

L'architecture du CDS offre de par sa composition matérielle et logicielle des capacités de redondance, ségrégation et diversification de données. Comme nous l'avons vu sur la Figure III.6 dans une configuration A380, certaines informations sont affichées de manière redondante dans le cockpit (côté gauche et côté droit). Deux pilotes sont également présent dans le cockpit ce qui contribue aussi en une surveillance mutuelle dans les tâches à réaliser. Les pilotes sont ainsi formés pour gérer une interface homme machine riche.

L'idée principale de l'approche proposée ici est la suivante : une défaillance pouvant toujours survenir dans le système malgré une conception zéro défaut (exemple un bit flip conduisant à une corruption des informations à afficher), on se sert de l'utilisateur qui est un élément indépendant du système pour surveiller son bon fonctionnement.

Nous présentons les architectures selon les flots d'affichage et de contrôle en conformité aux flots de données que l'on rencontre sur le widget, c'est-à-dire les données allant du système avion vers le CDS (flot d'affichage) et les données allant du CDS au système avion (flot de contrôle). Les architectures sont présentées selon trois tâches que le pilote effectue régulièrement : lire une donnée, effectuer une commande et saisir une donnée.

V.1.1 Architecture de tolérance aux fautes sur le flot d'affichage

Le principe de la tolérance aux fautes sur le flot d'affichage est d'afficher de façon redondante l'information à l'utilisateur. L'utilisateur devra ensuite réaliser une comparaison des informations afin de détecter s'il y a incohérence ou pas. Cette redondance peut être effectuée sur deux écrans différents comme le montre la Figure V.1.

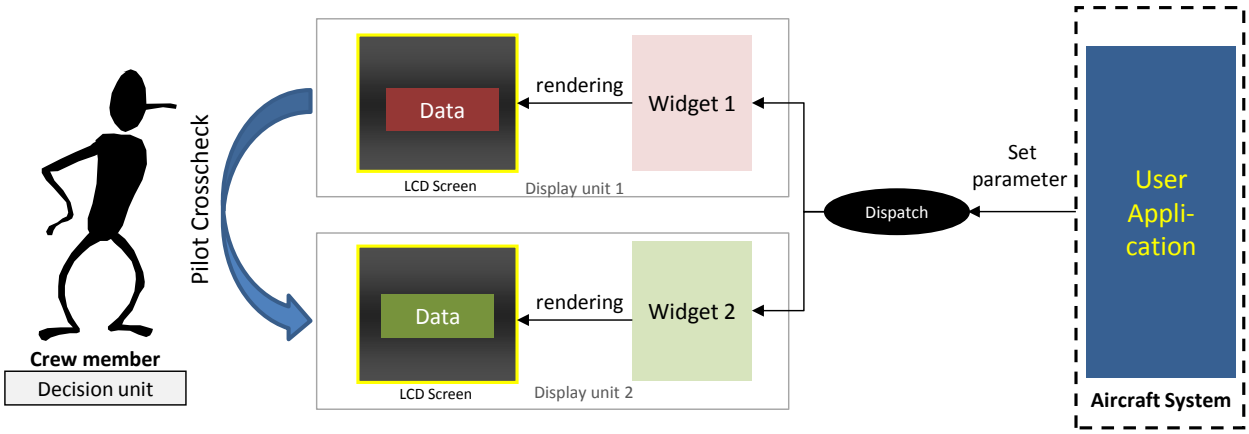


Figure V.1 Architecture de tolérance aux fautes du flot d'affichage reposant sur utilisateur

Ce type de redondance dans l'affichage n'est pas nouveau au sein du cockpit, comme nous allons le présenter dans l'exemple ci-dessous.

V.1.1.1 *Exemple d'un contrôle d'intégrité d'affichage actuel*

La Figure V.2 présente les formats PFD affichés sur le CDS et son instrument de secours l'ISIS (Integrated Standby Instrument System). L'ISIS est un calculateur qui affiche de façon redondante les informations critiques du PFD (altitude, attitude, vitesse etc...) Les sources de données de l'ISIS et du PFD sont différentes. L'ISIS est un calculateur ségrégué et diversifié du calculateur effectuant l'affichage du format PFD, ainsi le matériel et le logiciel sont différents et les équipes de conception des deux systèmes sont également différents ceci pour éviter tout point commun de défaillance.

L'ISIS est certes utilisé en cas de perte du format PFD donc pour des raisons de disponibilité, mais il est aussi affiché de manière permanente, en même temps que le PFD et sert ainsi de contrôle d'intégrité des informations affichées. On peut noter les recommandations ci-dessous dans les documents safety d'Airbus pour la détection d'une donnée erronée.

"*Crew detection: The crew detects the erroneous display by general monitoring of flight parameters on both sides PFDs and ISIS*"

Figure V.2 Redondance d'affichage dans le cockpit (A380)

Au total dans le CDS on a trois formats de type PFD affichés, le PFD du côté capitaine, le PFD du côté co-pilote et l'ISIS. Le contrôle de ces trois éléments permet à l'équipage de vol de détecter l'affichage erroné.

V.1.2 Architecture de tolérance aux fautes sur le flot de contrôle

Le principe de la tolérance aux fautes du flot de contrôle est de demander à l'utilisateur de confirmer les actions qu'il effectue. Par l'acte de confirmation, l'utilisateur vérifie ainsi que le système a bien pris en compte sa requête, donc que le système n'est pas défaillant.

La demande de confirmation peut être faite à différents endroits de l'interface du CDS. Soit sur la même DU qui a servi à réaliser le contrôle, soit sur une DU différente.

La Figure V.3 présente l'architecture pour la tolérance aux fautes du flot de contrôle, dans le cas où l'utilisateur doit confirmer l'action réalisée sur une DU différente, celle-ci permet notamment d'éviter certaines fautes transitoires du matériel. On distingue dans cette architecture :

- Deux unités d'affichage, une qui reçoit la première action de l'utilisateur et la transmet à l'UA, et une seconde unité d'affichage qui affiche un message de demande de confirmation et sur lequel l'utilisateur doit confirmer son action.

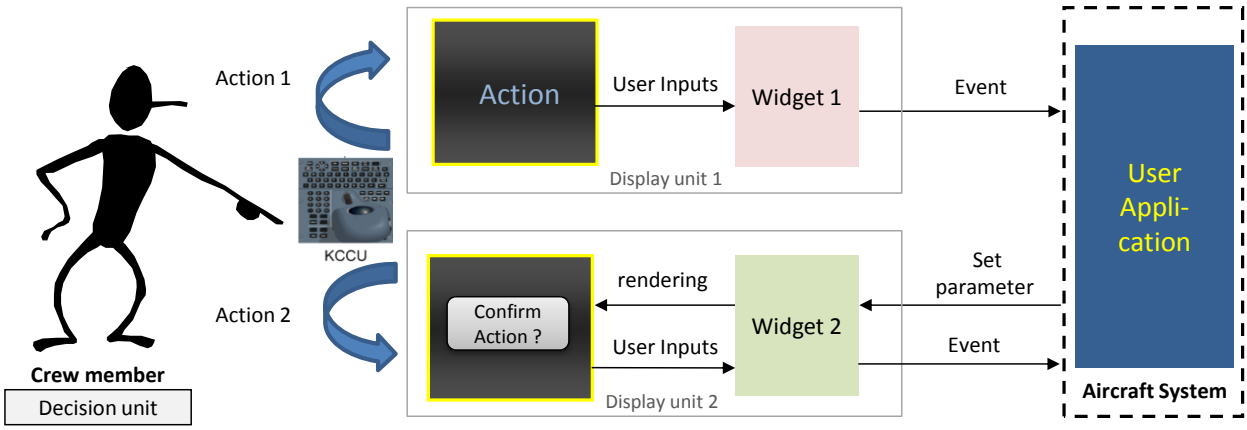


Figure V.3 Architecture de tolérance aux fautes du flot de contrôle reposant sur utilisateur

Si l'on prend l'exemple d'un widget de type EditBoxNumeric, lorsque l'utilisateur saisie une donnée et la valide. La donnée saisie est réaffichée sur une DU différente de celle de la saisie pour une demande de confirmation par l'utilisateur. Ce n'est qu'à la réception de cette confirmation que l'UA prendra en compte la donnée saisie. Ainsi le widget qui a servi a réalisé l'action de saisie (*widget 1*) est différent de celui réalisant la confirmation (*widget 2*), ce qui assure une ségrégation dans le traitement du flot de contrôle.

Si une erreur dans la DU1 conduit à une corruption de la donnée saisie, lors de la demande de confirmation sur la DU2, l'utilisateur verra l'erreur et ne confirmera pas l'action.

V.1.3 Description des tâches de l'utilisateur

De même que pour la conception zéro défaut, nous modélisons les tâches pour analyser l'impact sur l'utilisabilité des diverses configurations (interaction en entrée et interaction en sortie).

V.1.3.1 *Interaction en sortie*

Les modèles de tâche pour la lecture d'une donnée sont représentés aux Figure V.4 et Figure V.5. Ces modèles ont été réalisés par rapport à l'architecture de tolérance aux fautes sur le flot d'affichage de la Figure V.1.

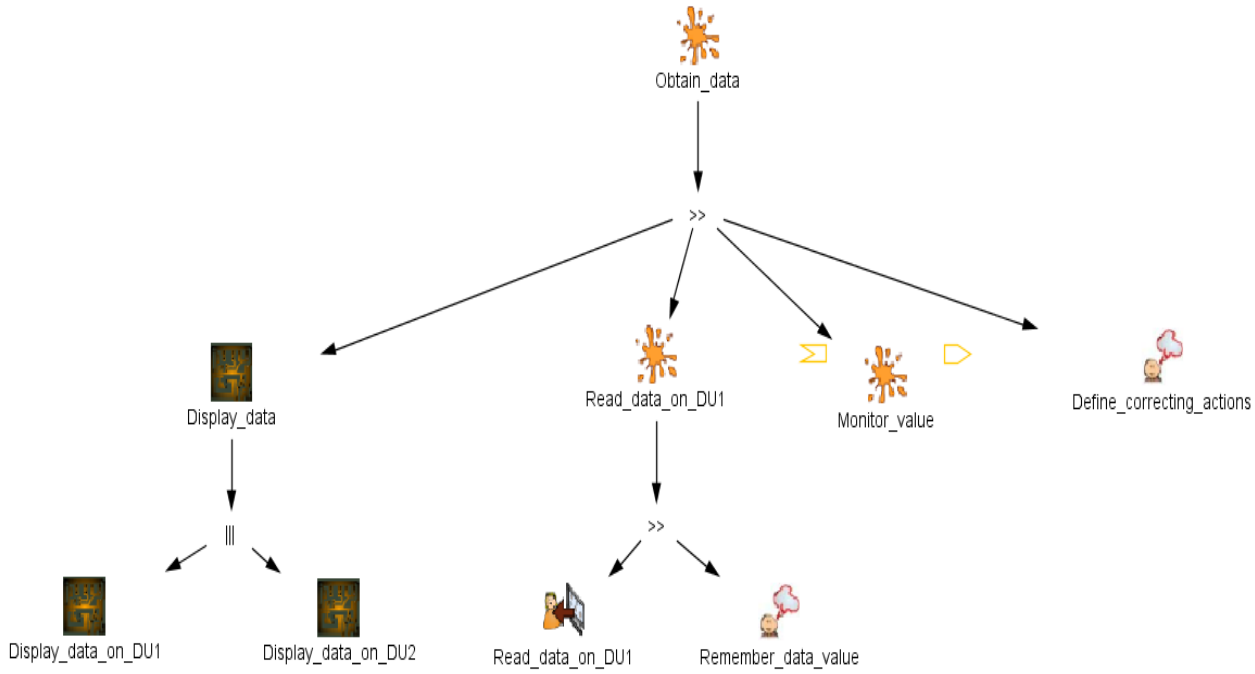


Figure V.4 Modèle de tâche de l'interaction en sortie pour la tolérance aux fautes reposant sur l'utilisateur (1)

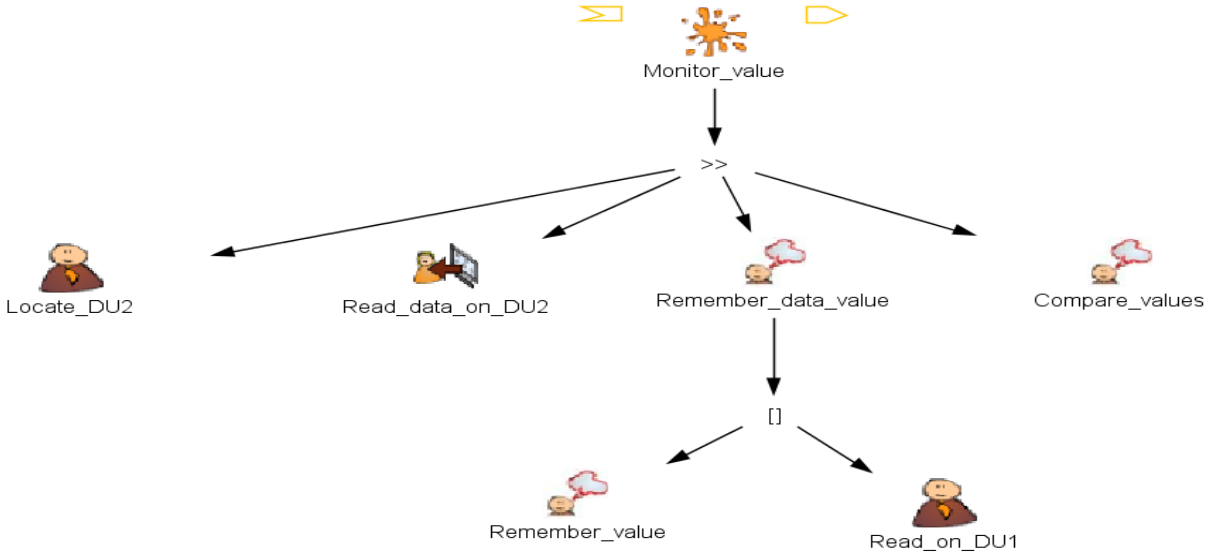


Figure V.5 Modèle de tâche de l'interaction en sortie pour la tolérance aux fautes reposant sur l'utilisateur (2)

Le modèle de la Figure V.4 doit se lire de la façon suivante, après affichage des données sur les deux DU du CDS (*Display_data_on_DU1* et *Display_data_on_DU2*), l'utilisateur effectue la lecture de la première donnée sur la première chaîne d'affichage (*Read_data_on_DU1*) et doit se rappeler la valeur de la donnée (*remember_data_value*). Il doit ensuite vérifier l'intégrité de ce premier affichage sur la deuxième chaîne d'affichage (M*onitor_value*) dont le détail est donné sur la Figure V.5 et qui consiste tout d'abord à localiser

où est affichée la seconde donnée (*Locate_DU2*), ensuite à la lire (*Read_data_on_DU2*) et réaliser une comparaison (*Compare_values*). Dans le cas où il aurait oublié la première donnée lue, il faudra qu'il relise la première chaîne d'affichage. En cas d'incohérence dans la comparaison il devra faire des actions correctives (*Define_correcting_actions*) qui peut par exemple consister à redémarrer les calculateurs.

On distingue donc au total 8 tâches réalisées par l'utilisateur.

V.1.3.2 *Interaction en sortie*

Les modèles de tâche pour la saisie de données sont représentés aux Figure V.6 et Figure V.7, et ont été réalisés par rapport à l'architecture de tolérance aux fautes sur le flot de contrôle de la Figure V.3.

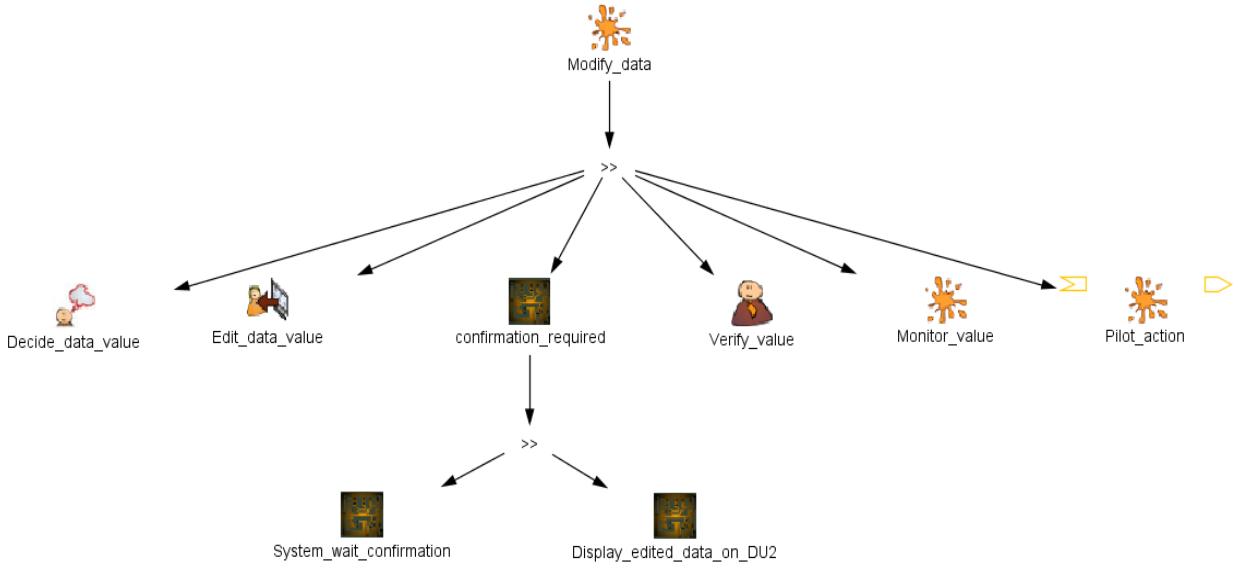


Figure V.6 Modèle de tâche de l'interaction en entrée pour la tolérance aux fautes reposant sur l'utilisateur (1)

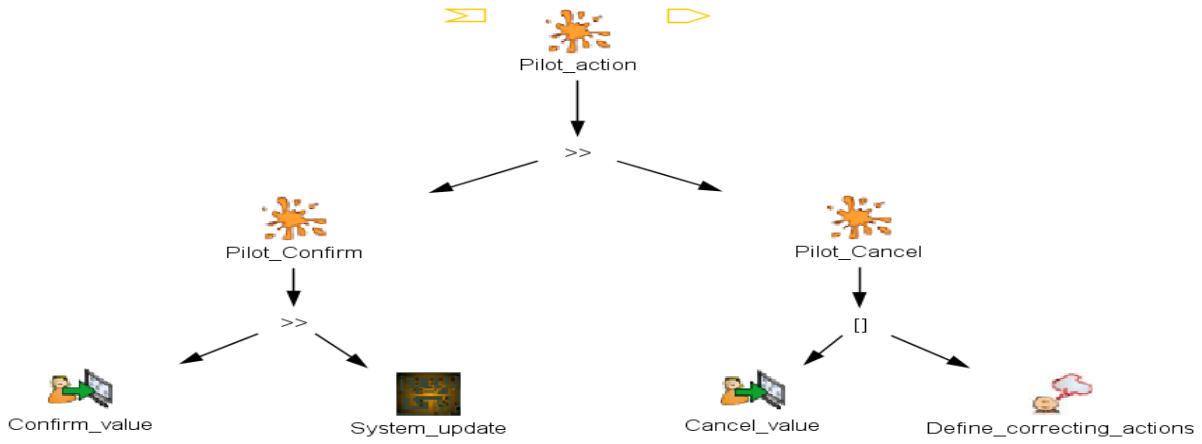


Figure V.7 Modèle de tâche de l'interaction en entrée pour la tolérance aux fautes reposant sur l'utilisateur (2)

Dans la Figure V.6, l'utilisateur décide de la valeur à saisir et saisie la donnée, le système reste en attente de la confirmation de la donnée et affiche une fenêtre de confirmation sur une chaîne indépendante (*Display_edited_data_on_DU2*). Avant de confirmer la donnée sur cette seconde chaîne, l'utilisateur devra vérifier si l'affichage est conforme à ce qu'il a saisi sur la première chaîne (*monitor_value*) et enfin valider ou annuler son action (*Pilot_action*) sur la Figure V.7. On distingue au total 11 tâches réalisées par l'utilisateur.

Par rapport à la conception zéro défaut on remarque déjà une augmentation importante sur l'activité de l'utilisateur.

Au niveau de l'interaction en sortie, on est passé d'une action simple sur la conception zéro défaut, à près huit actions sur la tolérance aux fautes reposant sur l'utilisateur.

Au niveau de l'interaction en sortie, on est passé de trois actions sur la conception zéro défaut à près de onze actions sur la tolérance aux fautes reposant sur l'utilisateur.

En fonction des phases de vol qui sont plus consommatrices de temps pour les pilotes, un tel accroissement de l'activité peut être jugé rédhibitoire.

La section suivante montre l'application de notre deuxième approche de tolérance aux fautes reposant sur le système et propose une étude de son impact sur l'activité de l'utilisateur.

V.2 APPLICATION DE L'APPROCHE TOLERANCE AUX FAUTES REPOSANT SUR LE SYSTEME ET ANALYSE DE L'IMPACT SUR L'UTILISABILITE

Dans cette option de tolérance aux fautes reposant sur le système, nous proposons d'appliquer les mécanismes d'autotestabilité aux composants de base de l'interaction que sont les widgets. Bien que la tolérance aux fautes soit fournie par l'exécution en parallèle d'au moins deux composants autotestables, nous nous concentrons sur la spécification du comportement du widget autotestable et particulièrement de la partie contrôleur. Globalement, la tolérance aux fautes sera obtenue par réplication des widgets autotestables.

Nous rappelons tout d'abord les principes de widgets autotestables que nous avons vu au Chapitre I, avant de montrer son application grâce au formalisme ICO sur quelques exemples de widgets que sont le PicturePushButton et l'EditBoxNumeric. Ces deux widgets ont été choisis parce qu'ils sont les plus utilisés et représentent différents niveau de complexité dans le comportement d'un widget. Le PicturePushButton a comportement simple que l'EditBoxNumeric.

V.2.1 Modélisation du comportement des widgets autotestables

V.2.1.1 *Modélisation du comportement d'un widget autotestable*

Nous avons vu qu'un widget possède deux flots de communication que sont le flot d'affichage et le flot de contrôle. Les modes de défaillances sur ces différents flots sont décrits au Tableau V.1 ci-dessous.

Flot de Contrôle	Sans réception de l'évènement (clic, saisie clavier) du périphérique d'entrée, le widget envoi l'évènement
	A la réception de l'évènement du périphérique d'entrée, le widget est dans un état défini pour envoyer l'évènement mais l'évènement n'est pas envoyé
	Le widget envoi l'évènement alors qu'il n'est pas dans un état approprié
	L'évènement est envoyé avec une valeur corrompu (ne s'applique qu'aux widgets ayant ce paramètre en plus)
	Les évènements sont envoyés dans le mauvais ordre (ne s'applique qu'aux widgets ayant plusieurs évènements)
Flot d'affichage	Sans réception de la commande de l'application cliente, le widget modifie ses paramètres internes (label, valeur numérique, état visible …)
	A la réception de la commande de l'application cliente, le widget ne met pas à jour ses paramètres internes
	Les données mises à jour sont corrompues (modification du label, de la valeur saisie, mauvais état de visibilité)

Tableau V.1 : les modes de défaillance du widget

Pour détecter les modes de défaillance d'un widget, nous proposons une architecture de widget autotestable représentée à la Figure V.8 qui est basée sur le modèle décrit dans (Laprie, et al., 1996). Elle est composée d'un bloc fonctionnel (*Functional part*), et d'un bloc de contrôle (*controller*) qui est constitué de la fonction simplifiée (*simplified functional part*) et d'un module de comparaison (*comparison*).

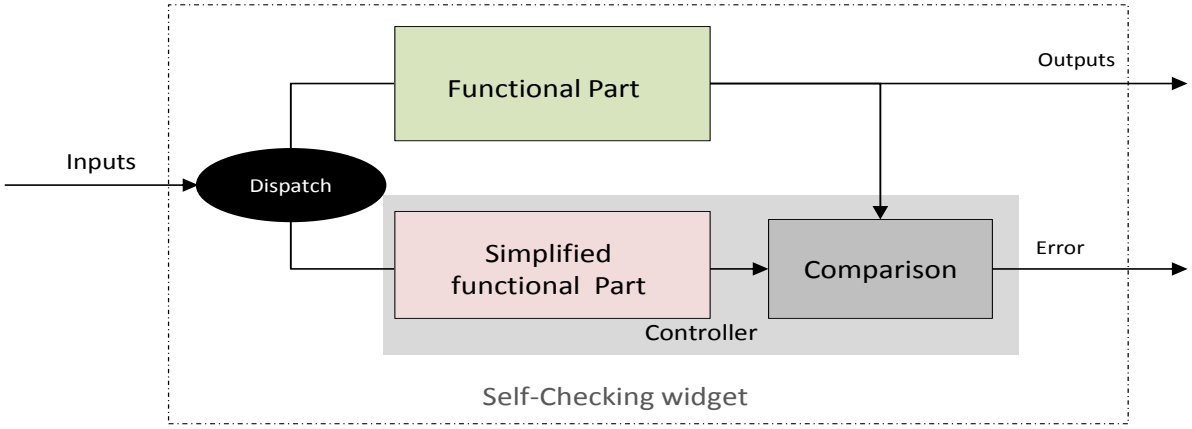


Figure V.8 : Architecture d'un widget autotestable

Le bloc fonctionnel du widget est celui décrit dans l'approche « vers la conception zéro défaut » vu au chapitre IV, dans lequel nous avons décrit le comportement précis du widget est décrit : chacun des états du widget, la gestion des *setparameters,* les évènements levés par le widget, les évènements reçus des périphériques d'entrées (représentés sous forme d'appel de méthodes dans nos modèles) et les paramètres servant au rendu graphique du widget.

Le module de contrôle a pour but la vérification des propriétés de sûreté que l'on veut garantir, ces propriétés de sûreté sont basées sur les modes de défaillances définit au Tableau V.1. Il est constitué pour cela d'une fonction simplifiée et d'un bloc de comparaison.

La fonction simplifiée ne comporte que les paramètres et caractéristiques pertinents pour vérifier les propriétés de sûreté. La fonction simplifiée doit connaître les états du bloc fonctionnel, les paramètres modifiés par l'UA, les évènements reçus des périphériques d'entrées, les exceptions levées. La simplification réalisée n'est pas toujours visible pour les widgets simples, car le comportement de la fonction simplifiée peut être identique à celui de la fonction dans ce cas.

Le module de comparaison a pour rôle de vérifier la cohérence entre les résultats du bloc fonctionnel et la fonction simplifiée. Les comparaisons réalisées permettent de vérifier la cohérence des sorties dans la mesure où le bloc fonctionnel et la fonction simplifiée reçoivent les mêmes entrées.

Les Figure V.9 et Figure V.10, présentent le fonctionnement du widget autotestable selon le flot de contrôle et d'affichage.

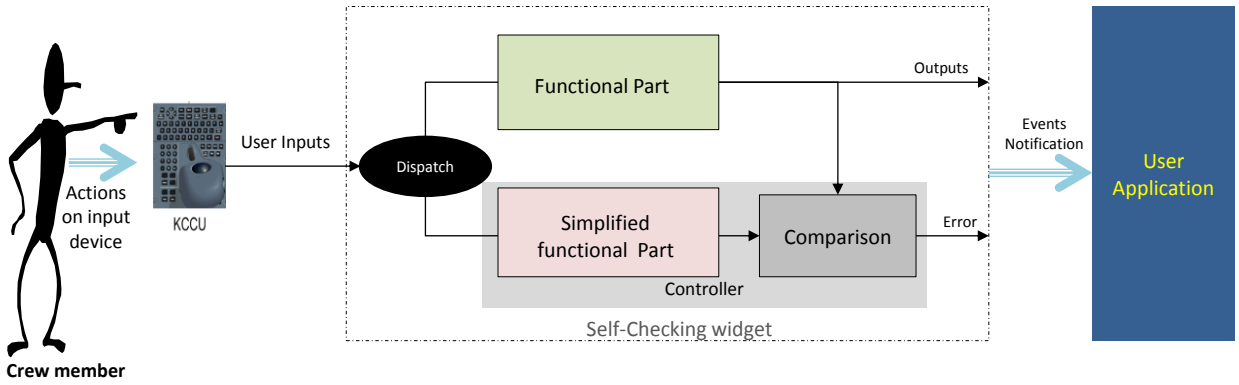


Figure V.9 : Traitement du flot de contrôle par un widget autotestable

Le fonctionnement est le suivant au niveau du flot de contrôle :

- Les évènements du clavier et du dispositif de pointage initiés par l'utilisateur sont dupliqués vers le composant fonctionnel et la fonction simplifiée: (*Dispatch*)
- Le composant fonctionnel et la fonction simplifiée effectue en parallèle le traitement de l'évènement en fonction de l'état interne du widget. Les résultats des deux blocs sont ensuite envoyés au module de comparaison.
- Le module de comparaison effectue une comparaison des résultats, s'il y a incohérence une erreur est levée en plus de la sortie du composant fonctionnel qui est toujours reçue par l'UA.

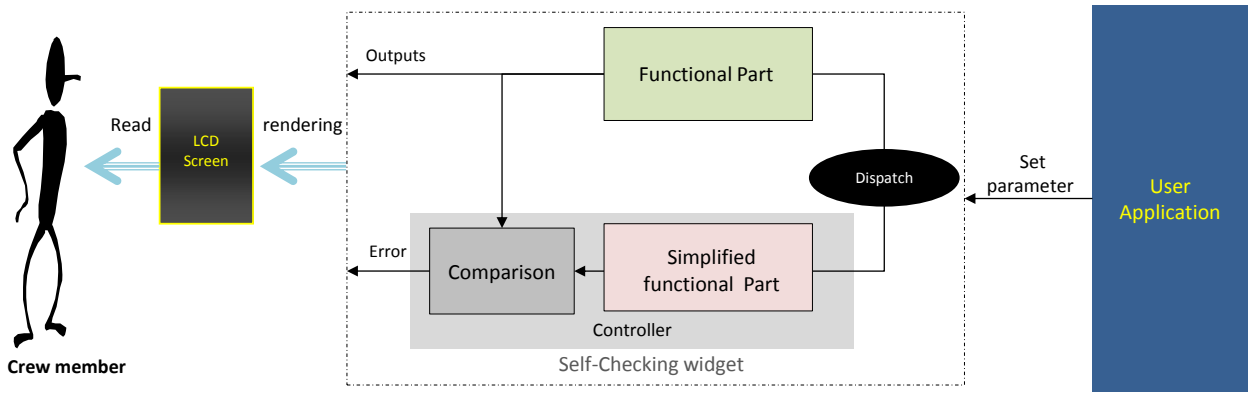


Figure V.10 : Traitement du flot d'affichage par un widget autotestable

Le flot d'affichage fonctionne sur le même principe que le flot de contrôle.

- Duplication des *setparameter* venant de l'UA vers le composant fonctionnel et la fonction simplifiée.
- Traitement en parallèle du *setparameter* par le composant fonctionnel et la fonction simplifiée et envoie des résultats au bloc de comparaison.
- Comparaison des résultats par le bloc de comparaison, s'il y a incohérence une erreur est levée, sinon seule la sortie du composant fonctionnel est envoyée.

La section suivante présente l'application du widget autotestable en utilisant le formalisme ICO et ceci au travers des exemples du PicturePushButton et de l'EditBoxNumeric.

V.2.1.2 *Exemple de modélisation du comportement de certains widgets autotestables*

V.2.1.2.1 Modélisation du comportement d'un PicturePushButton Autotestable

La modélisation du comportement d'un PicturePushButton a été présentée dans le chapitre précédent. Cette description constitue la partie fonctionnelle du PicturePushButton autotestable. Nous avons vu que les évènements des périphériques d'entrées gérés par le PicturePushButton sont les évènements produits par le dispositif de pointage (*MousePressed*, *MouseReleased* et *MouseClicked*). L'assertion exécutable vérifiée au niveau du flot de contrôle est la levée de l'évènement A661_EVT_SELECTION à la réception de l'évènement *MouseClicked* et ceci lorsque le widget est dans l'état *visible* et *enable*. Les paramètres que l'application cliente (UA) peut mettre à jour sur le widget PicturePushButton sont : la visibilité (A661_VISIBLE), la disponibilité (A661_ENABLE), le label (A661_STRING), le style d'affichage (A661_StyleSet) et l'image (A661_PictureReference).

Les modes de défaillance d'un PicturePushButton peuvent se résumer au Tableau V.2.

Flot de Contrôle	Sans réception d'un clic (*MouseCliked*) du dispositif de pointage, le widget envoi l'évènement A661_EVT_SELECTION
	A la réception d'un clic (*MouseCliked*), le widget est visible et enable mais l'évènement A661_EVT_SELECTION n'est pas envoyé.
	L'évènement A661_EVT_SELECTION est envoyé alors que le widget n'est pas dans un état approprié.
	Un évènement autre qu'A661_EVT_SELECTION est envoyé
Flot d'affichage	Le widget met à jour un ou plusieurs de ses paramètres (A661_VISIBLE, A661_ENABLE, A661_STRING, A661_StyleSet, A661_PictureReference) sans réception d'une demande de mise à jour par l'UA.
	A la réception d'une demande de mise à jour d'un de ses paramètres, le PicturePushButton ne met pas à jour le ou les paramètres concernés.
	Les paramètres mis à jour sont corrompus.

Tableau V.2 Modes de défaillance d'un PicturePushButton

L'assemblage ICompoNet d'un PicturePushButton autotestable est représenté sur la Figure V.11. Cet assemblage montre les composants ICompoNet du widget autotestable que sont le bloc fonctionnel du PicturePushButton, la fonction simplifiée et le bloc de comparaison. L'ensemble des services fournis (*facets*), des services reçus (*receptacle*), des évènements gérés par chaque composant et les liens entre les composants y sont aussi représentés.

Nous décrivons ci-dessous le comportement de chaque bloc ICompoNet du PicturePushButton autotestable spécifié à l'aide du formalisme ICO. Bien que les comportements puissent sembler évidents, nous les détaillerons ici pour montrer la capacité du formalisme ICO et de l'outil Petshop à spécifier et simuler des systèmes interactifs.

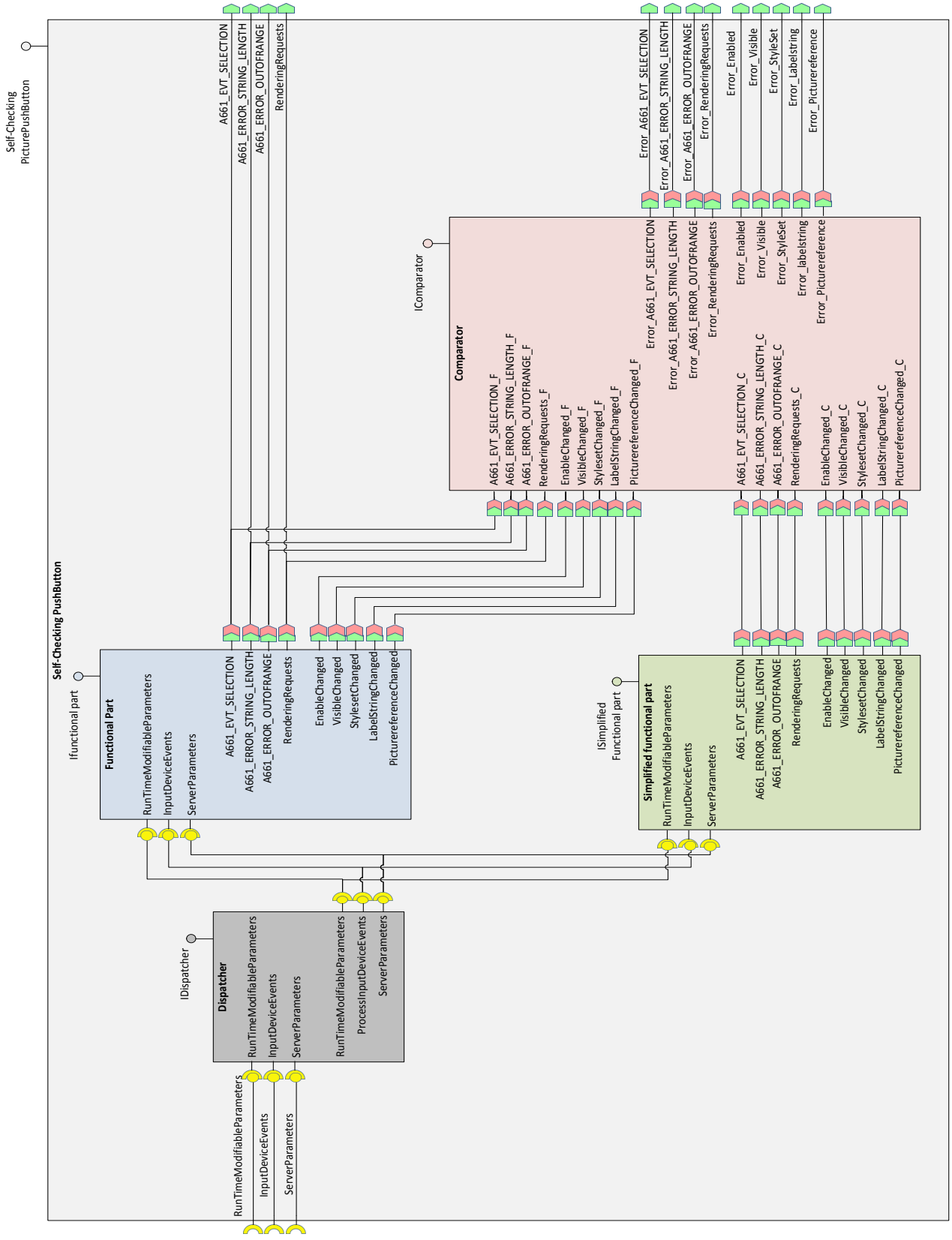


Figure V.11 Assemblage ICompoNet d'un PicturePushButton autotestable

V.2.1.2.1.1 Comportement de la fonction simplifiée

Le modèle de la fonction simplifiée est un modèle réduit du comportement du PicturePushButton. Conformément aux modes de défaillance décrite sur le widget, les différents paramètres et caractéristiques que la fonction simplifiée doit intégrer pour la vérification des propriétés de sûreté sont : l'ensemble des paramètres modifiables en exécution par l'UA, les évènements levés par le widget qui dépendent des évènements reçus des périphériques d'entrées, les paramètres servant au rendu graphique et les états du widget.

D'autres simplifications ont été apportées dans la gestion de certains paramètres, en effet certains détails de leur comportement n'étant pas utile pour le contrôle d'intégrité. C'est le cas par exemple de la gestion des paramètres *visible* et *enable*. Il n'y a plus de distinction visuelle entre l'état *visible* ou *enabled* (valeur du jeton à True) et l'état *notvisible* et *NotEnabled* (valeur du jeton à *False*). Pour connaitre l'état du paramètre, il faut connaitre la valeur du jeton dans la place *visible* (respectivement *Enabled*) (voir Figure V.12).

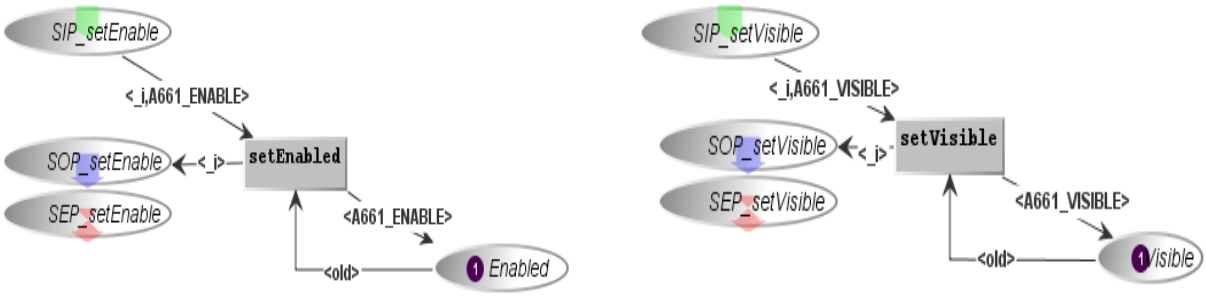


Figure V.12 Gestion des paramètres visible et Enable dans le modèle de la fonction simplifiée

V.2.1.2.1.2 Comportement du bloc Dispatcher

Le dispatcher a pour rôle de recevoir à la fois les *setparameters* venant des UA et les évènements des périphériques d'entrées (Clavier/dispositif de pointage) puis de les transférer ensuite au bloc fonctionnel et à la fonction simplifiée.

Le dispatcher fonctionne sur un principe de file d'attente, les appels de service sont stockés et traités dans l'ordre. Il assure aussi la synchronisation dans le traitement du bloc fonctionnel et de la fonction simplifiée, c'est-à-dire que les deux blocs traitent les mêmes appels de services à un moment donné. Cette synchronisation est faite pour assurer que la comparaison des données effectuée par le comparateur porte bien sur le traitement des mêmes entrées.

La Figure V.13 présente la gestion du paramètre *set_Visible* par le dispatcher avec en dessous le comportement équivalent du paramètre *visible* dans le bloc fonctionnel et dans le modèle de la fonction simplifiée.

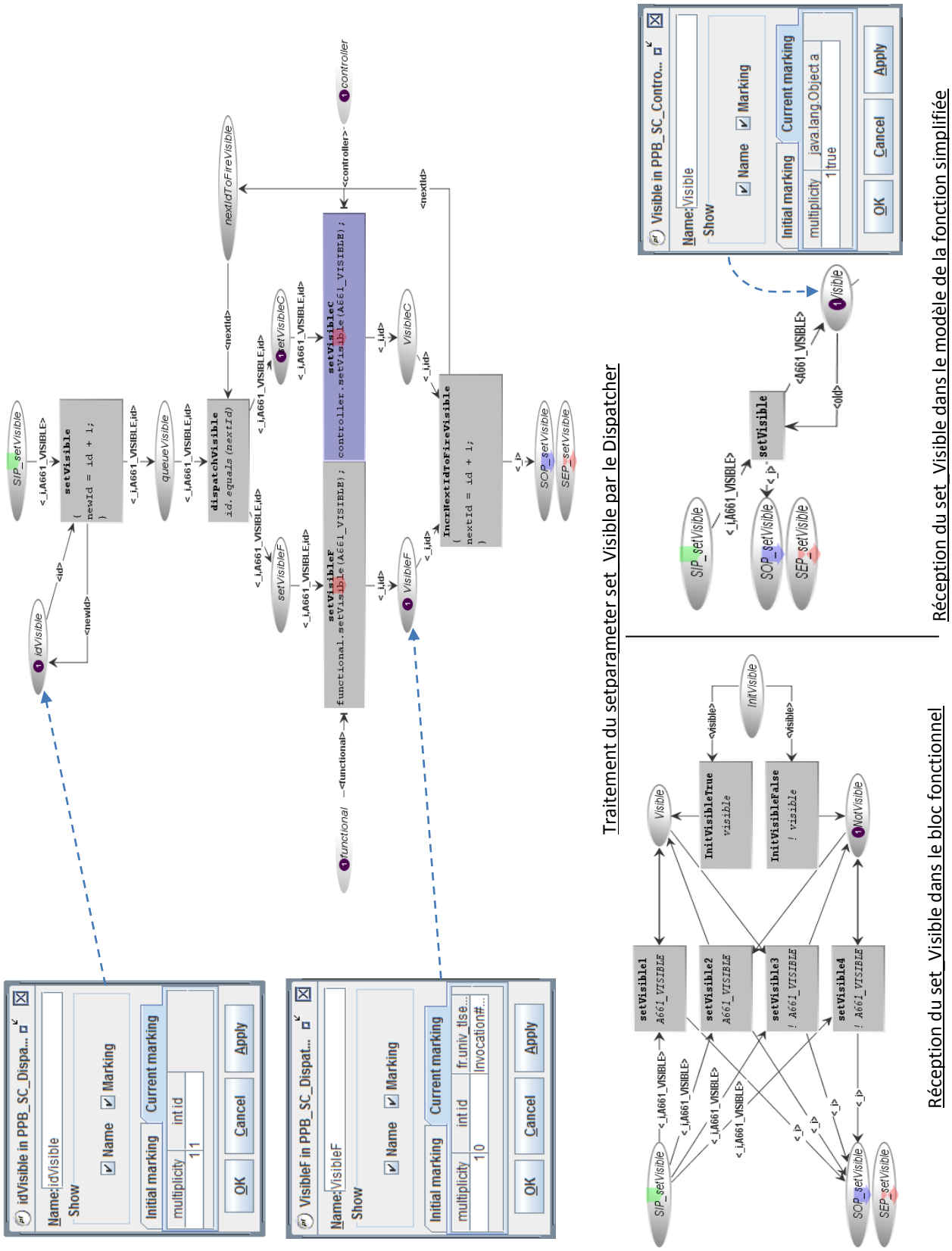


Figure V.13 Gestion du transfert du paramètre set_Visible

Lorsque l'UA demande une mise à jour du paramètre *visible*, elle effectue un appel de méthode *setvisible* avec la valeur du paramètre (*true* ou *false*). Dans le modèle, ceci est représenté par un appel de service en déposant un jeton dans la place d'entrée *SIP_setVisible*. Le premier appel de service sera affecté d'un identifiant égal à 0 (place *idVisible*). L'appel de service est ensuite transféré au bloc fonctionnel et à la fonction simplifiée (transitions *setVisibleF*, *setVisibleC*).

Les modèles ICO de gestion du paramètre *Visible* dans le bloc fonctionnel et dans la fonction simplifiée (partie bas de la Figure V.13) reçoivent (suite à ce transfert) un jeton dans leur place *SIP_setVisible*. La Figure V.13 présente le cas où l'UA demande de mettre le paramètre *visible* à *false*. Le transfert de la commande vers le bloc fonctionnel a été effectué (franchissement transition *setVisibleF*), on peut le voir sur le réseau de Petri du bloc fonctionnel que la valeur a bien été mise à jour car on a un jeton dans la place *Notvisible*. Par contre le transfert vers la fonction simplifiée n'est pas encore effectué, et on peut voir sur le réseau de Petri correspondant de la Figure V.13 que la valeur du jeton dans la place *visible* est encore à true.

Pour assurer le non entrelacement dans le traitement des appels de services, le dispatcher ne traitera la commande suivante de visibilité que s'il a reçu les retours du bloc fonctionnel et de la fonction simplifiée (franchissement de la transition *IncrNextIdToFireVisible*).

Pour assurer un traitement des appels de service dans l'ordre, chaque appel de service est associé à un identifiant dont la valeur est incrémentée à chaque appel. Les appels en attente sont stockés dans la place *queueVisible*. Ensuite la transition *dispatchVisible* ne prendra que le jeton qui porte l'identifiant correspondant au numéro suivant par rapport à celui que le dispatcher a traité. On peut voir sur la figure que la place *VisibleF* a pour id 0, c'est l'identifiant de l'appel de service en cours de traitement, et la place *idVisible* a pour id 1, c'est l'identifiant que portera le prochain appel de service. Lorsque le dispatch aura reçu le retour de la fonction simplifiée, on trouvera dans la place *NextIdToFireVisible* un id égal à 1. Ainsi, la transition *dispatchVisible* ne prendra dans la file d'attente que le jeton ayant pour id 1.

La structure du dispatcher est générique, permettant de gérer de la même manière tous les autres *setparameter* et évènements KCCU.

V.2.1.2.1.3 Comportement du module de comparaison

Le module de comparaison a pour rôle de vérifier la cohérence entre les résultats du bloc fonctionnel et les résultats du modèle de la fonction simplifiée. Les comparaisons réalisées permettent de vérifier la cohérence des sorties en fonction des entrées.

Nous allons présenter un exemple de comparaison sur un *setparameter* et sur un *Event* pour montrer la surveillance qui est réalisée sur le flot d'affichage et le flot de contrôle. Ceux-ci étant les seules communications en ARINC 661 sur le widget.

Comme *setparameter* nous prenons l'exemple du paramètre LabelString et comme *Event*, l'évènement A661_EVT_SELECTION.

Nous précisons tout d'abord que le module de comparaison fonctionne essentiellement avec la notion d'évènement des ICO tel que nous l'avons vu au Chapitre I. Cette notion d'évènement fait intervenir un émetteur permettant l'abonnement à d'autres modèles ICO et un récepteur permettant de recevoir les évènements auxquels le modèle est abonné.

V.2.1.2.1.3.1 Gestion du LabelString dans le comparateur

Quatre comparaisons sont réalisées :

1. Il y a eu modification du paramètre du côté du bloc fonctionnel sans qu'il y ait modification du même paramètre dans la fonction simplifiée.
2. Il y a eu modification du paramètre du côté de la fonction simplifiée sans qu'il y ait modification du paramètre dans le bloc fonctionnel.

On considère dans ces deux premiers cas que soit il y a eu une mise à jour du paramètre sans réception d'une demande de mise à jour de l'UA, soit il y a bien eu une demande de mise à jour du paramètre par l'UA, mais cette demande n'a pas été prise en compte. La défaillance peut ici venir soit du bloc fonctionnel soit de la fonction simplifiée. Il est difficile de savoir lequel est défaillant.

3. Il y a eu modification du paramètre des deux côtés mais la valeur enregistrée n'est pas la même. on considère dans ce cas qu'il y a eu une corruption de la valeur par l'un des blocs. (bloc fonctionnel ou fonction simplifiée)
4. Il y a eu modification du paramètre des deux côtés et les valeurs sont identiques : on considère qu'il n'y a pas eu d'erreur.

La surveillance du paramètre LabelString par le module de comparaison en ICO est présentée à la Figure V.14.

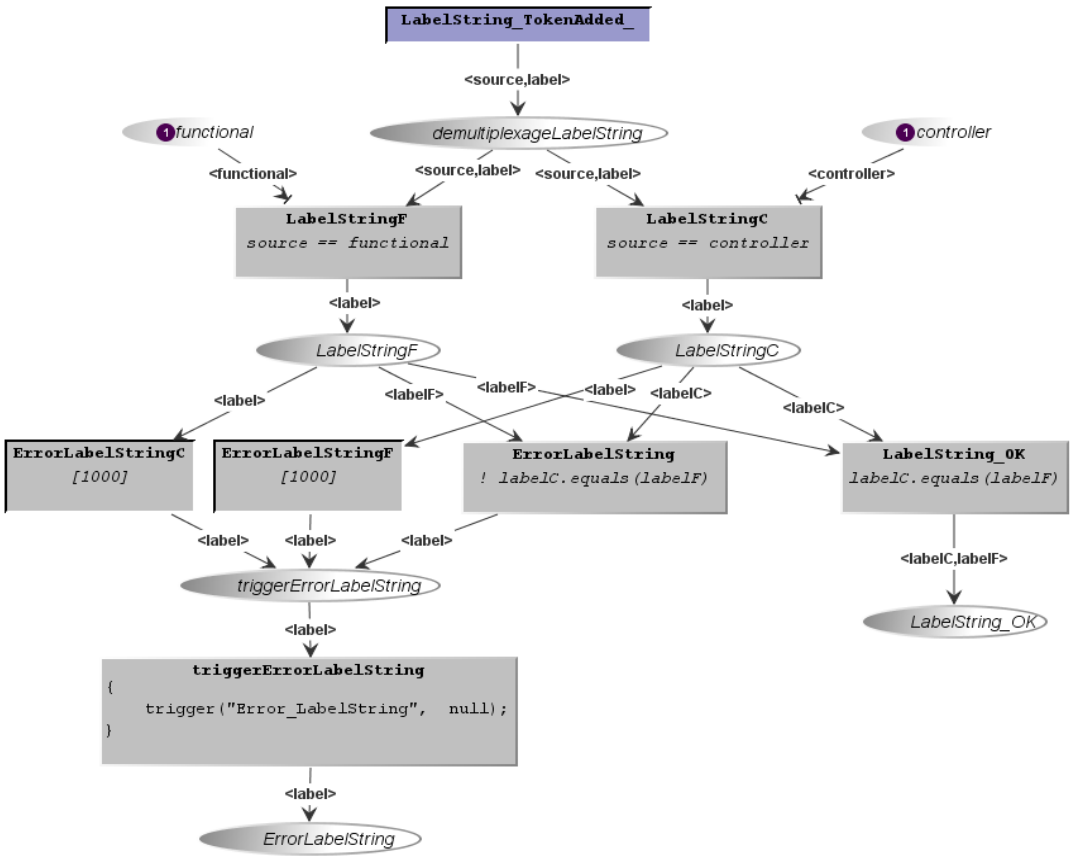


Figure V.14 Comparaison des valeurs envoyées par le bloc fonctionnel et la fonction simplifiée dans le cas du paramètre LabelString

Nous avons décidé d'employer les notions d'émetteur et récepteur d'évènements pour décrire la communication entre le bloc fonctionnel, la fonction simplifiée et le comparateur. Les modèles du bloc fonctionnel et de la fonction simplifiée se comportent comme des émetteurs d'évènements et celui du comparateur comme des récepteurs d'évènements.

La transition *LabelString_TokenAdded* sur la Figure V.14 est une transition synchronisée, elle écoute les évènements *TokenAdded* du bloc fonctionnel et de la fonction simplifiée. Ainsi à chaque fois qu'un jeton est déposé dans la place *LabelString* du bloc fonctionnel ou de la fonction simplifiée (i.e. qu'il y a une mise à jour du paramètre *LabelString)*, la transition *LabelString_TokenAdded* de la Figure V.14 est franchie.

Le point de réception des paramètres *LabelString* étant unique au sein du bloc de comparaison, une distinction des sources de réception est ensuite réalisée afin d'identifier si la source est le bloc fonctionnel ou la fonction simplifiée (transition *LabelStringF* pour le bloc fonctionnel, *LabelStringC* pour la fonction simplifiée), ceci est réalisé en comparant la source reçue et la valeur d'instance du bloc fonctionnel et de la fonction simplifiée stockée dans les places *functional* et *controller* de la figure.

La détection des erreurs est modélisée à l'aide des transitions suivantes dans le modèle du comparateur.

- La transition *ErrorLabelStringC* est franchie lorsqu'on reçoit une mise à jour du bloc fonctionnel sans recevoir celle de la fonction simplifiée.
- La transition *ErrorLabelStringF* est franchie lorsqu'on reçoit une mise à jour de la fonction simplifiée sans recevoir celle du bloc fonctionnel.
- La transition *ErrorLabelString* est franchie lorsqu'on reçoit les mises à jour du bloc fonctionnel et de la fonction simplifiée, mais avec des valeurs différentes.
- La transition *LabelString_OK* est franchie lorsqu'on a reçu des mises à jour dans le bloc fonctionnel et la fonction simplifiée et avec des valeurs identiques. C'est le cas où l'on considère qu'il n'y a pas eu d'erreur.

On note aussi qu'un autre cas d'erreur est possible lorsque les valeurs du LabelString sont erronées mais identiques, et ceci n'est pas détecté comme une erreur dans notre modélisation.

Nous choisissons arbitrairement de ne pas faire de distinction entre les types d'erreur et de ne faire qu'une seule notification d'erreur par paramètre. Ici c'est l'évènement d'erreur *Error_LabelString* qui est levé. Il aurait été facile de représenter le type d'erreur par un paramètre dans le modèle.

V.2.1.2.1.3.2 Gestion de l'A661_Evt_Selection dans le comparateur

Le principe de surveillance d'un *Event* est assez similaire à celui d'un *setparameter*. La particularité de l'évènement A661_EVT_SELECTION est qu'aucune valeur n'est transmise. Par contre, il existe des évènements du standard ARINC 661 qui transmettent une valeur, pour ces évènements la surveillance est identique à celle d'un *setparameter* qui a été présenté ci-dessus.

Les comparaisons consistent donc à vérifier que l'évènement a bien été levé dans le bloc fonctionnel ainsi que dans la fonction simplifiée.

Les tests réalisés au niveau du bloc de comparaison consistent donc à vérifier que l'évènement a bien été levé dans le bloc fonctionnel et dans la fonction simplifiée.

Si l'évènement a été levé par le bloc fonctionnel et pas par la fonction simplifiée (vice versa), Plusieurs hypothèses de défaillance sont possibles :

- Il n'y a pas eu de clic de l'utilisateur, mais un des blocs (entre le bloc fonctionnel et la fonction simplifiée) a levé l'évènement
- Il y a eu un clic de l'utilisateur, mais un des blocs n'a pas levé l'évènement
- Il y a eu un clic de l'utilisateur, mais un bloc n'était pas dans l'état approprié pour lever l'évènement (par exemple était indisponible avec le paramètre enable à false)

La gestion de l'évènement A661_EVT_SELECTION dans le module de comparaison est présentée à la Figure V.15.

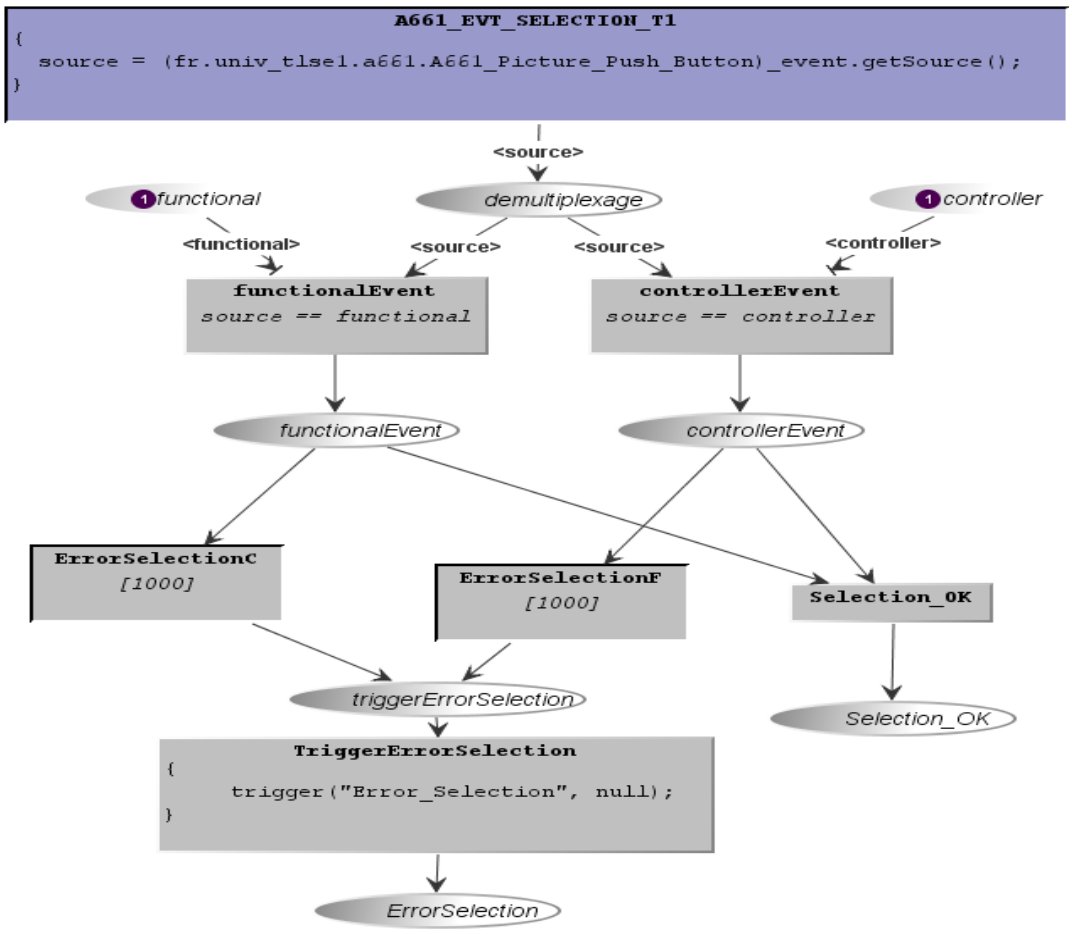


Figure V.15 Comparaison de l'A661_EVT_SELECTION

Lorsque l'évènement A661_EVT_SELETION est levé à la fois par le bloc fonctionnel et la fonction simplifiée, dans ce cas la transition *A661_EVT_SELECTION_T1* de la Figure V.15 est franchie deux fois. Le point de réception étant unique, une identification des sources est ensuite effectuée. (De même que pour le modèle du *setparameter* présenté ci-dessus)

La modélisation de la détection des erreurs est réalisée de la façon suivante :

- La transition *ErrorSelectionC* est franchie lorsque l'évènement A661_EVT_SELECTION est levé par le bloc fonctionnel et pas par la fonction simplifiée

- La transition *ErrorSelectionF* est franchie lorsque l'évènement A661_EVT_SELECTION est levé par la fonction simplifiée, mais pas par le bloc fonctionnel.

Une particularité dans la notion d'évènement ICO est que le bloc de comparaison n'écoute que l'évènement A661_EVT_SELECTION, si à la suite d'une défaillance, un autre évènement qu'A661_EVT_SELECTION est levé, cet évènement ne sera pas reçu par le comparateur. Le comparateur ne sera donc pas notifié si d'autres Events sont produits suite à une erreur.

V.2.1.2.2 Modélisation du comportement d'un EditBoxNumeric autotestable

La modélisation du comportement d'un EditBoxNumeric a été présentée dans le Chapitre I. Ce modèle correspond la partie fonctionnelle de l'EditBoxNumeric autotestable. L'EditBoxNumeric étant un élément de saisie de donnée, il gère l'ensemble des évènements clavier et du dispositif de pointage, notifie plusieurs évènements à l'UA et possède plusieurs paramètres qui peuvent être mis à jour par l'UA. En plus de ces paramètres, nous avons dû décrire plus finement les états du widget pour mieux définir son rendu graphique avant, pendant et après la saisie.

Selon le tableau générique des modes de défaillance sur un widget, le raffinement au niveau de l'EditBoxNumeric donne le Tableau V.3.

Flot de Contrôle	Sans saisie de donnée, une notification de saisie est envoyée à l'UA
	Suite à une saisie de donnée, aucune notification de saisie n'est envoyée à l'UA
	Suite à une saisie de donnée, le widget n'est pas dans l'état approprié pour envoyer une notification à l'UA, mais la notification est envoyée
	Une notification de saisie est envoyée à l'UA avec une valeur différente de la valeur saisie
	Les évènements levés par le widget sont envoyés dans un ordre différent de celui réalisé par l'utilisateur.
Flot d'affichage	Le widget met à jour un ou plusieurs de ses paramètres sans réception d'une demande de mise à jour par l'UA.
	A la réception d'une demande de mise à jour d'un de ses paramètres, l'EditBoxNumeric ne met pas à jour le ou les paramètres concernés.
	Les paramètres mis à jour sont corrompus.

Tableau V.3: Les modes de défaillance d'un EditBoxNumeric

Pour détecter les modes de défaillance, nous proposons l'assemblage ICompoNet de l'EditBoxNumeric autotestable qui est représenté sur la Figure V.16. On y retrouve le bloc fonctionnel, la fonction simplifiée, le bloc de comparaison, ainsi que les types de données échangés entre les différents composants.

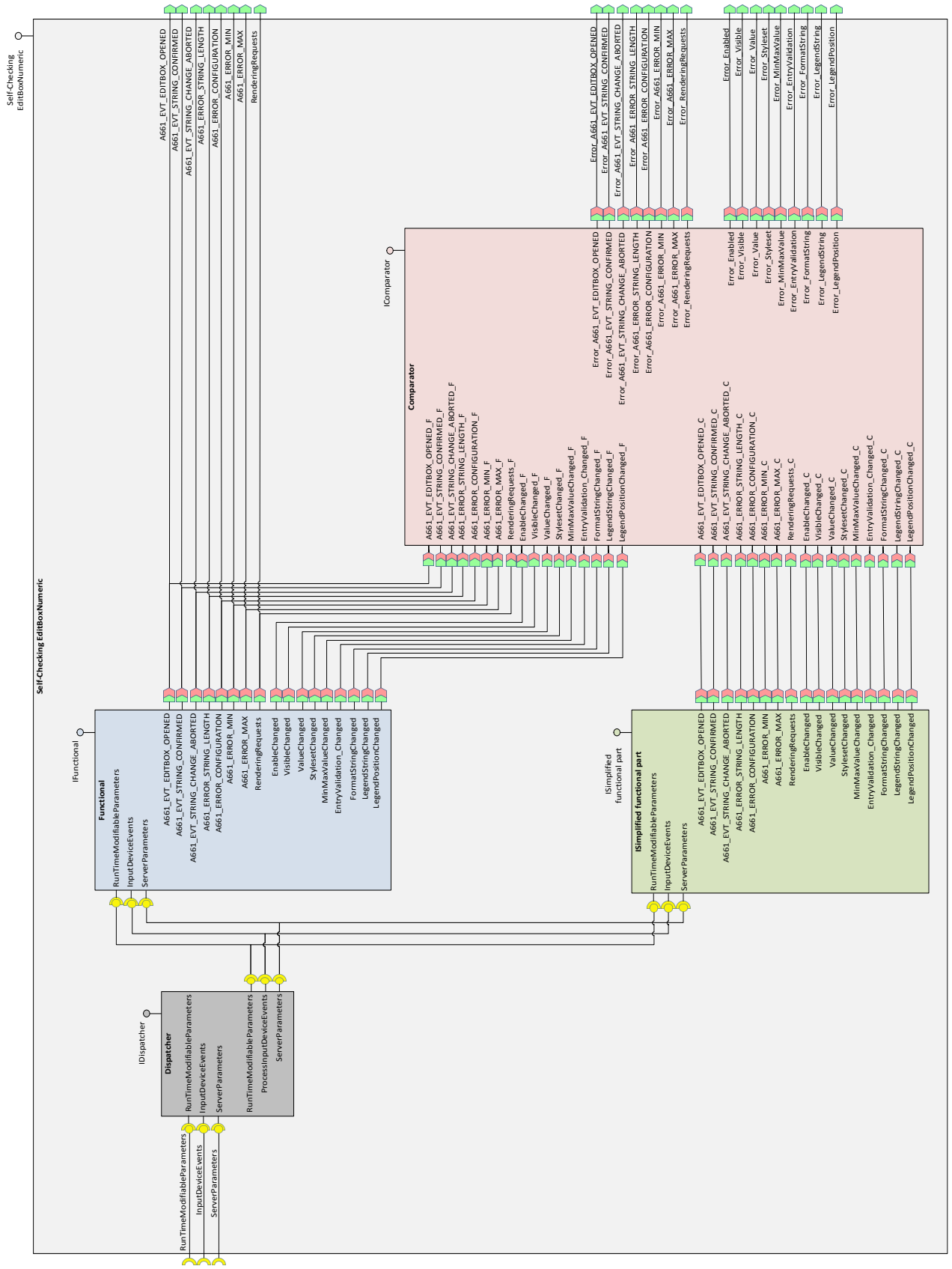


Figure V.16 Assemblage ICompoNet d'un EditBoxNumeric autotestable

Le comportement du bloc Dispatcher est identique à celui du PicturePushButton, chaque paramètre venant de l'UA et les évènements des périphériques d'entrée sont transférés au bloc fonctionnel et à la fonction simplifiée.

La plupart des comparaisons réalisées dans le bloc de comparaison sur les paramètres et les évènements, suivent le même principe que ceux présentés sur le PicturePushButton.

Nous ne présentons ci-dessous que les particularités liées à l'EditBoxNumeric. Ces particularités concernent la fonction simplifiée et quelques caractéristiques du module de comparaison.

V.2.1.2.2.1 Comportement de la fonction simplifiée

Pour réaliser une surveillance des évènements levés par le bloc fonctionnel et des paramètres mis à jour, la fonction simplifiée doit également avoir les mêmes évènements et paramètres pour que la comparaison soit cohérente.

Au niveau du flot de contrôle, ce qui nous intéresse particulièrement ce sont les évènements levés et la valeur saisie. L'état où l'utilisateur est entrain de saisir la donnée (*Editing*) n'est pas intéressant pour la sûreté, en effet il n'est pas intéressant de comparer chaque touche de clavier réalisée, surtout que celle-ci peut être effacée par l'utilisateur. On attendra donc la validation de la saisie pour fixer un point de comparaison. Il en est de même des paramètres *TicsCoarse* et *TicsFine* (qui permettent un ajustage grossier ou fin de la donnée) qui ne sont de fait pas pris en compte pour la comparaison et ne sont donc pas représentés dans la fonction simplifiée.

Les états *Idle* et *WaitingForUA* sont nécessaires pour la surveillance, car ils permettent de s'assurer que le widget est dans l'état approprié pour lever un évènement.

Ce sont ici les principales simplifications apportées au modèle de l'EditBoxNumeric. Le modèle ICO de la fonction simplifiée de l'EditBoxNumeric correspond à celui de la Figure IV.25 que nous avons vu Chapitre I avec les simplifications expliqué ci-dessus. De même la gestion des paramètres *visible* et *enable* a été simplifiée et correspond à celle vue sur la fonction simplifiée de PicturePushButton.

V.2.1.2.2.2 Gestion du module de comparaison

Le module de comparaison a pour rôle de vérifier la cohérence entre les résultats du bloc fonctionnel et la fonction simplifiée. Les mêmes principes de comparaison que sur le PicturePushButton sont réalisés sur les *setparameter* et sur les *Events* de l'EditBoxNumeric et ces comparaisons permettent de détecter la plupart des défaillances listées au Tableau V.3. Une défaillance qui n'est pas couverte dans ce tableau est celle concernant l'ordre d'envoi des évènements. Comme nous l'avons vu dans la description du bloc fonctionnel de l'EditBoxNumeric au Chapitre I, ce widget gère plusieurs évènements et les envois dans un ordre spécifique en fonction des actions de l'utilisateur.

C'est pourquoi une surveillance supplémentaire est ajoutée dans le comparateur. Cette surveillance consiste à vérifier l'ordre des évènements levés par l'EditBoxNumeric à savoir A661_EVT_EDITBOX_OPENED, A661_EVT_STRING_CONFIRMED et A661_EVT_STRING_ABORTED.

Ainsi si l'évènement A661_EVT_STRING_CONFIRMED est levé sans qu'on ne reçoive avant A661_EVT_EDITBOX_OPENED cela veut dire qu'il n'y a peut-être pas eu de saisie de donnée. Il en est de même si l'on reçoit A661_EVT_STRING_ABORTED sans recevoir A661_EVT_EDITBOX_OPENED.

La Figure V.17 présente la vérification des séquences d'envoie des évènements de l'EditBoxNumeric au niveau du comparateur.

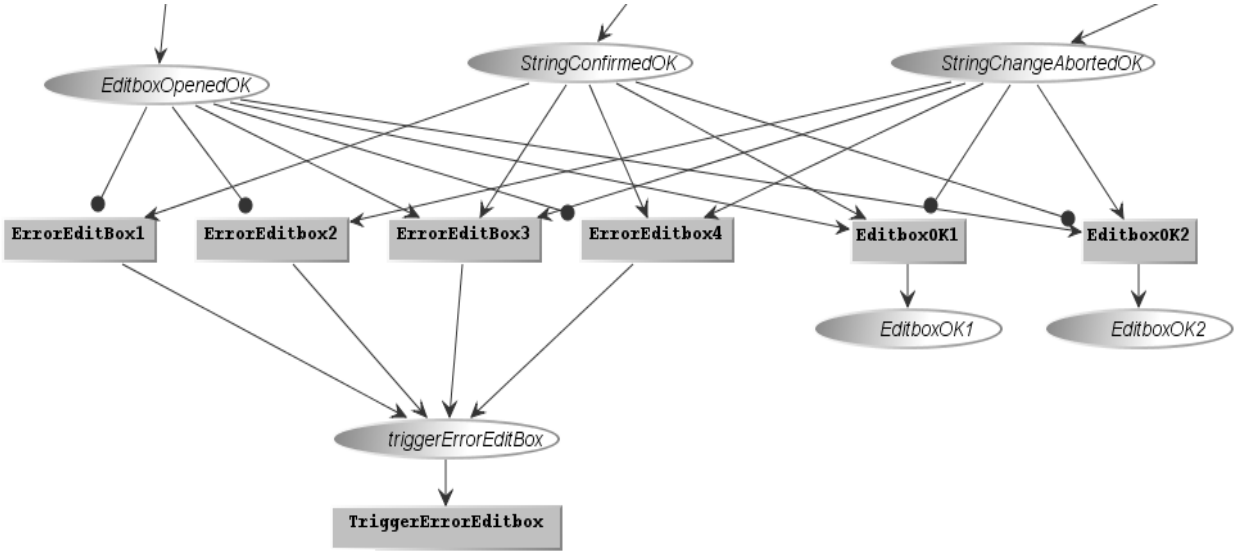


Figure V.17 Comparaison de la séquence des évènements de l'EditBoxNumeric

Les places *EditboxOpenedOK*, *StringConfirmedOK* et *StringChangeAbortedOK* correspondent au cas où la comparaison des évènements A661_EVT_STRING_CONFIRMED, A661_EVT_EDITBOX_OPENED et A661_EVT_STRING_ABORTED venant du bloc fonctionnel et de la fonction simplifiée est correcte.

Il existe deux options où l'on considère que l'ordre de réception des évènements est correct, donc qu'il n'y a pas eu d'erreur. La première option est quand on reçoit les évènements A661_EVT_EDITBOX_OPENED et A661_EVT_STRING_CONFIRMED, elle correspond au cas où l'utilisateur saisie la donnée et la valide en appuyant sur la touche entrée du clavier, dans ce cas c'est la transition *EditboxOK1* qui est franchie. La deuxième option est quand on reçoit les évènements A661_EVT_EDITBOX_OPENED et A661_EVT_STRING_ABORTED, elle correspond au cas où l'utilisateur entreprend l'action de saisir une donnée mais abandonne finalement cette action en appuyant par exemple sur la touche Echap du clavier, dans ce cas c'est la transition *EditboxOK2* qui est franchie.

Les différents cas où il y a détection d'erreur sont :

- On reçoit A661_EVT_STRING_CONFIRMED sans recevoir A661_EVT_EDITBOX_OPENED (transition ErrorEditBox1)
- On reçoit A661_EVT_STRING_ABORTED sans recevoir A661_EVT_EDITBOX_OPENED (transition ErrorEditbox2)
- On reçoit les trois évènements (transition ErrorEditBox3)
- On reçoit les évènements A661_EVT_STRING_ABORTED et A661_EVT_STRING_CONFIRMED sans recevoir A661_EVT_EDITBOX_OPENED (transition ErrorEditbox4)

Les aspects temporels n'ont pas été pris en compte ici, on peut encore affiner la surveillance en vérifiant que le comparateur reçoit bien A661_EVT_EDITBOX_OPENED avant A661_EVT_STRING_CONFIRMED, ou A661_EVT_EDITBOX_OPENED avant A661_EVT_STRING_ABORTED.

V.2.2 Validation et synthèse

Nous avons montré au travers des exemples de PicturePushButton autotestable et d'EditBoxNumeric autotestable comment le contrôleur était conçu pour détecter les modes de défaillance que nous avons identifiés.

Grâce à la capacité de simulation de l'outil PetShop nous avons pu jouer différents scénarios de comportement du widget autotestable, comportement normal et modes de défaillance afin de valider la capacité de détection du module de comparaison.

Pour valider le fonctionnement au niveau du flot d'affichage, nous avons procédé de la façon suivante lors de l'exécution de l'étude de cas du FCU.

- Demander des mises à jour des paramètres de widgets par l'UA.
- Insérer au niveau du bloc fonctionnel ou de la fonction simplifiée du widget des valeurs corrompues
- Modifier les paramètres de mise à jour de l'UA par la valeur corrompue
- Observer que le bloc de comparaison détecte bien qu'il y a incohérence entre la valeur du bloc fonctionnel et celle de la fonction simplifiée.

Pour valider le fonctionnement au niveau du flot de contrôle, nous avons procédé de la façon suivante lors de l'exécution de l'étude de cas du FCU.

- Réaliser des actions utilisateur sur l'interface du FCU (clic, saisie de donnée)
- Insérer au niveau du bloc fonctionnel ou de la fonction simplifiée du widget des évènements simulant des faux clics.
- Observer que le bloc de comparaison détecte bien qu'il y a incohérence entre le fait que la fonction simplifiée lève un évènement et pas le bloc fonctionnel.

L'application de ces scénarios nous ont permis de valider le comportement du modèle du widget autotestable, mais aussi de voir que la capacité de détection n'était pas parfaite. On relève ainsi:

- Qu'une valeur erronée identique du côté du bloc fonctionnel et de la fonction simplifiée n'est pas détectée comme une erreur.
- Un évènement erroné à la fois du côté du bloc fonctionnel et de la simplifiée n'est pas détecté par le bloc de comparaison.

La procédure de validation que nous avons réalisée ici est faite à base de modèle et n'est pas complète. Nous nous sommes concentrés dans notre étude à décrire le comportement du widget autotestable et particulièrement de la partie contrôleur. Nous ne sommes pas allé jusqu'à la réalisation de tout le mécanisme de tolérance aux fautes. En employant des mécanismes de tolérance aux fautes, il serait nécessaire d'étudier la robustesse du système par exemple par des techniques d'injection de fautes (Arlat, et al., 1990).

Le Tableau V.4 donne un aperçu de la taille des modèles des widgets autotestables dont le PicturePushButton et l'EditBoxNumeric. L'effort dans la modélisation est assez important, on y retrouve un rapport d'environ six par rapport à la partie fonctionnelle du widget. Cette taille de modèle dépend de la

modélisation que nous avons choisie, des optimisations sont toujours possibles. Le bloc Dispatcher peut par exemple être un bloc à l'extérieur du composant autotestable et être modéliser à l'intérieur du protocole de communication entre les différents composants logiciels du système interactif CDS.

		Places	**Transitions**
PicturePushButton autotestable	Dispatcher	142	70
	Bloc fonctionnel	56	30
	Fonction simplifiée	52	18
	Comparateur	58	73
	Total	**308**	**191**
EditBoxNumeric Autotestable	Dispatcher	201	100
	Bloc Fonctionnel	92	55
	fonction simplifiée	85	46
	Comparateur	119	141
	Total	**497**	**342**

Tableau V.4 Taille des modèles du PicturePushButton et de l'EditBoxNumeric autotestables

La modélisation des widgets autotestables a été présenté dans (Tankeu, et al., 2011) (Fabre, et al., 2011).

V.2.3 **Description des tâches opérateur**

Dans cette section, nous modélisons les tâches utilisateurs pour analyser l'impact sur l'utilisabilité, des mécanismes de widgets autotestables. Dans la mesure où les actions à réaliser par le pilote en cas de détection d'erreur ne sont pas prises en compte, l'utilisation de widgets autotestables ne modifie pas les tâches à réaliser et n'a donc pas d'impact négatif

V.2.3.1 *Interaction en sortie*

Pour réaliser la lecture d'une donnée avec la tolérance aux fautes centrée sur le système, le schéma d'architecture équivalent est celui de la Figure V.10 vu précédemment. La détection de l'erreur est transparente à l'utilisateur, puisqu'elle est effectuée par le widget autotestable. Le modèle de tâche est ici identique à celui de l'approche vers la conception zéro défaut que nous avons présenté à la Figure IV.35. Ce n'est qu'en cas de détection d'erreur qu'une action corrective peut être envisagée par l'utilisateur.

V.2.3.2 *Interaction en entrée*

Pour réaliser la saisie d'une donnée avec la tolérance aux fautes centrée sur le système, le schéma d'architecture équivalent est celui de la Figure V.9 vu précédemment. De même que pour l'interaction en sortie, la détection d'erreur est transparente à l'utilisateur. Le modèle de tâche est identique à celui de l'approche vers la conception zéro défaut que nous avons présenté à la Figure IV.37

V.3 APPLICATION DE L'APPROCHE MIXTE DE LA TOLERANCE AUX FAUTES ET ANALYSE DE L'IMPACT SUR L'UTILISABLITE

L'approche mixte de la tolérance aux fautes est celle reposant sur l'utilisateur et sur le système. Pour des fonctions classées catastrophiques, il est demandé de n'avoir aucune panne simple pouvant conduire à des défaillances catastrophiques (critère *Fail Safe*). L'approche mixte peut être utilisée pour respecter ce critère de *Fail Safe.* En cas de limite de l'approche de tolérance aux fautes reposant sur le système, l'utilisateur qui connaît mieux l'environnement du système et le contexte d'utilisation jouera ici le rôle de contrôleur final du système

De même que pour les sections précédentes de ce chapitre, nous présentons l'approche selon le flot d'affichage et le flot de contrôle.

V.3.1 Architecture de tolérance aux fautes sur le flot d'affichage

Comme nous l'avons expliqué dans les hypothèses et le périmètre, la partie système avion donc l'UA n'est pas dans le périmètre de notre étude. On considère que chaque système avion critique implémentera une architecture tolérante aux fautes, donc les données venant de ces systèmes sont considérées fiables.

Pour assurer la tolérance aux fautes du flot d'affichage, nous proposons une redondance dans l'affichage similaire à celle proposée dans l'approche de tolérance aux fautes reposant sur l'utilisateur. Cependant ici, chaque chaîne d'affichage intègre des widgets autotestables.

La Figure V.18 présente l'architecture correspondante pour l'approche mixte de tolérance aux fautes sur le flot d'affichage.

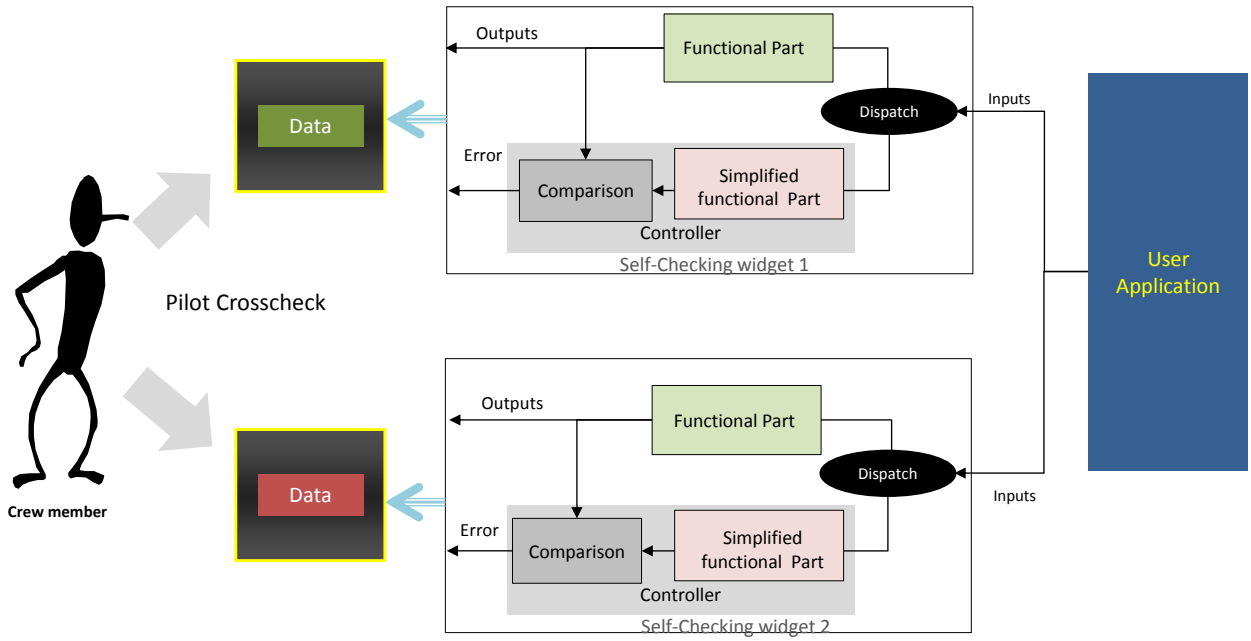


Figure V.18 Architecture du flot d'affichage de l'approche mixte de tolérance aux fautes

Pour éviter tout point unique de défaillance au niveau du CDS, les chaînes réalisant les affichages doivent être ségrégués et diversifiés, donc être exécutées sur des calculateurs différents.

V.3.2 Architecture de tolérance aux fautes sur le flot de contrôle

La Figure V.19 présente l'architecture proposée de tolérance aux fautes sur le flot de contrôle, l'activité de l'utilisateur consiste ici à saisir une donnée et à la valider.

Le fonctionnement est plus complexe que celle des autres approches, car en plus de demander à l'utilisateur de réaliser la confirmation de la donnée saisie sur un calculateur différent, l'idée principale ici est d'utiliser un moyen dédié de confirmation, ségrégué et diversifié de celle utilisé pour la saisie. En effet dans les autres approches de tolérance aux fautes, le KCCU était utilisé à la fois pour saisir la donnée et pour confirmer ou non la donnée saisie. Une erreur au sein du KCCU pouvait constituer un point unique de défaillance et donc au non-respect du critère *Fail Safe*. Ce moyen dédié de confirmation correspond sur la figure au « *Confirmation PushButton* », qui peut être un bouton physique ou logiciel, mais communiquant avec l'UA par un chemin indépendant de celui utilisé pour saisir la donnée.

On intègre également des widgets autotestables dans l'architecture du système pour assurer la détection des erreurs lors de la saisie et de l'affichage du message de demande de confirmation.

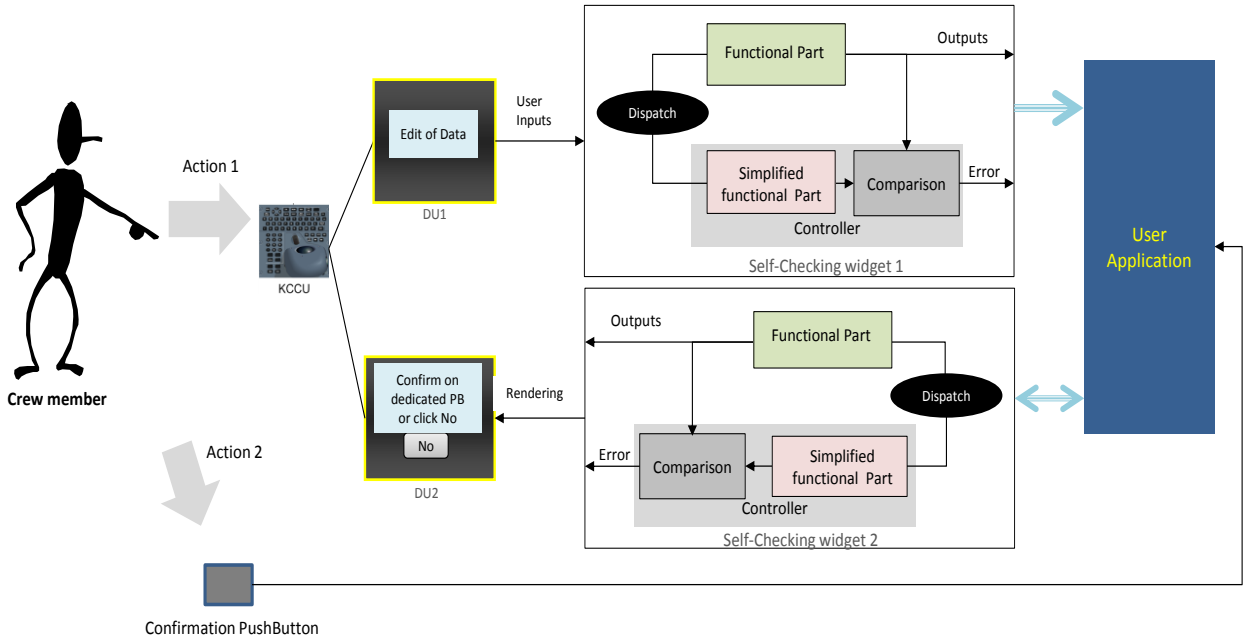


Figure V.19 Architecture du flot de contrôle de l'approche mixte de tolérance aux fautes

V.3.3 Description des tâches opérateur

De même que pour les autres approches, nous modélisons les tâches pour analyser l'impact sur l'utilisabilité des diverses configurations d'architecture des Figure V.18 et Figure V.19.

V.3.3.1 *Interaction en sortie*

Pour la lecture de la donnée, le modèle de tâche ici est quelque peu similaire à celui de la tolérance aux fautes reposant sur l'utilisateur que nous avons présenté aux Figure V.4 et Figure V.5. Une difficulté supplémentaire sera rencontrée au niveau de l'utilisateur pour la localisation de la seconde chaîne d'affichage qui sera ségrégué et diversifié de la première.

V.3.3.2 *Interaction en entrée*

Pour la saisie de la donnée, les tâches de l'opérateur restent quelque peu similaires à celui de la tolérance aux fautes reposant sur l'utilisateur. Une difficulté supplémentaire sera rencontrée pour la localisation du bouton dédié de confirmation. Les modèles de tâche correspondant sont représentés aux Figure V.20 et Figure V.21.

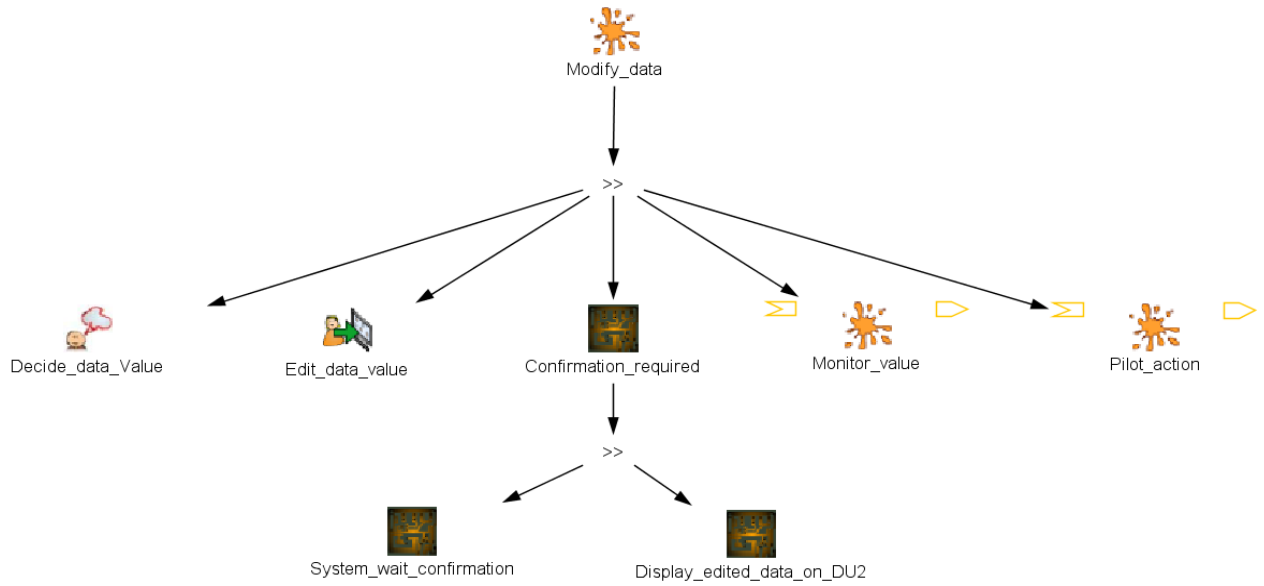


Figure V.20 Modèle de tâche de l'interaction en entrée de l'approche mixte (1)

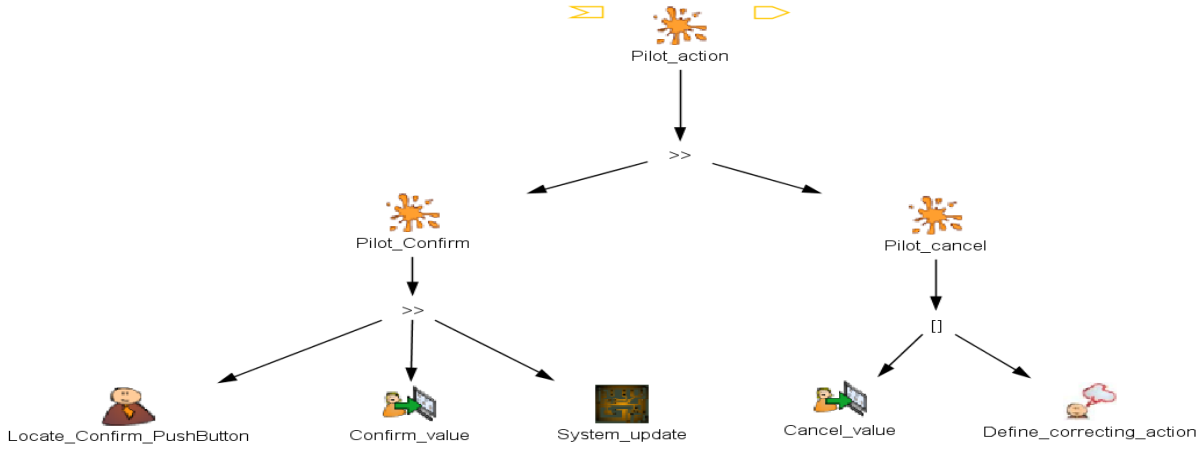


Figure V.21 Modèle de tâche de l'interaction en entrée de l'approche mixte (2)

La Figure V.20 présente le début de l'activité de l'utilisateur qui consiste à décider de la valeur à saisir et à la saisir sur l'interface via l'utilisation du KCCU. Le système reste ensuite en attente de la confirmation de la donnée saisie et affiche une fenêtre de demande de confirmation sur une chaîne indépendante. L'utilisateur doit ensuite vérifier que la valeur affichée sur la fenêtre de confirmation est conforme à celle qu'il a saisie (*Monitor_Value*), et ensuite confirmer ou annuler son action (*Pilot_action*).

La Figure V.21 détaille l'action du pilote, ainsi pour confirmer la valeur saisie, il doit d'abord localiser ou se trouve le bouton de confirmation. Si par contre, il voit une incohérence entre la valeur saisie et la valeur affichée, il peut décider de faire des actions correctives (*Define_correcting_action*).

V.4 ANALYSE DE L'IMPACT DES APPROCHES SUR L'UTILISABILITE

L'objectif de cette section est de faire une analyse générale des impacts des différentes approches de conception zéro défaut et de tolérance aux fautes sur l'utilisabilité.

Le Tableau V.5 résume la taille des modèles de tâche en fonction de la criticité de l'interaction telle que nous l'avons vu dans les sections précédentes.

Architectures		**Nombre de tâches**	**Complexité**
Conception zero défaut	Interaction en sortie	1	1 (simple)
	Interaction en entrée	3	3 (simples)
Tolérance aux fautes reposant sur le système	Interaction en sortie	1	1 (simple)
	Interaction en entrée	3	3 (simples)
Tolérance aux fautes reposant sur l'utilisateur	Interaction en sortie	8	5 simples + 3 complexes
	Interaction en entrée	11	8 simples + 3 complexes
Approche mixte de la tolérance aux fautes	Interaction en sortie	11	8 simples + 3 complexes
	Interaction en entrée	15	12 simples + 3 complexes

Tableau V.5 Taille des modèles de tâche en fonction de la criticité de l'interaction

Le nombre de tâches représenté dans le tableau ci-dessus ne prend pas en compte les actions correctives de l'utilisateur en cas de détection d'erreur, on remarque ainsi que le nombre de tâche au niveau de la tolérance aux fautes reposant sur le système est identique à celle de l'approche zéro défaut. Si l'on doit prendre en considération les actions correctives de l'utilisateur, le nombre de tâche au niveau de la tolérance aux fautes reposant sur le système sera plus élevé.

Les tâches complexes correspondent ici à certaines tâches cognitives qui nécessitent un certain effort cérébral (exemple faire une comparaison, réaliser des actions correctives). Les tâches simples correspondent aux tâches interactives ou perceptible qui nécessite un effort moteur de la part de l'utilisateur (lire une donnée, localiser une donnée). Les tâches complexes nécessitant plus d'effort par rapport aux tâches simples, nous leur avons donné une pondération de 5 et les simples ont une pondération de 1. On obtient ainsi :

Difficulté de la tâche = Nombre de tâches simples + 5 x Nombre de tâches complexes

Les poids sont ici attribués pour une analyse de la performance. On peut les ajuster selon les connaissances théoriques disponibles dans le domaine de l'interaction homme machine (telles que la loi de Fitts (Fitts, 1954) ou le modèle de processeur humain (Card, et al., 1986), dans ce cas il faudrait une description plus précise de l'interface utilisateur et donc de l'UA correspondante (nombre de boutons, de pages, de texte affichés), mais l'objectif des travaux de cette thèse ne couvre pas cet aspect.

La complexité de l'architecture du système est présentée au Tableau V.6.

Architecture	Ordre de probabilité	Description de l'architecture
Conception Zéro défaut	$< 10^{-5}$	Pas de mécanisme de tolérance aux fautes implémenté. Description précise et formelle contribuant à l'élimination des fautes. Architecture identique à celle de l'interaction non-critique d'aujourd'hui.
Tolérance aux fautes reposant sur le système	$< 10^{-7}$	Mécanisme de tolérance aux fautes niveau widget. L'interface du système vis-à-vis de l'utilisateur ne change pas par rapport à la conception zéro défaut, mais l'architecture des calculateurs doit intégrer des notions de redondance, ségrégation et diversification.
Tolérance aux fautes reposant sur l'utilisateur	$< 10^{-7}$	Mécanisme de tolérance aux fautes reposant sur l'utilisateur. Moins de complexité de calcul au niveau du calculateur par rapport à la tolérance aux fautes reposant sur le système, car le bloc de comparaison est supprimé pour être délégué à l'utilisateur. Double affichage et demande de confirmation.
Approche mixte de la tolérance aux fautes.	$< 10^{-9}$	Pour respecter le critère Fail-Safe, on effectue une combinaison de mécanisme de tolérance aux fautes reposant sur le système et reposant sur l'utilisateur. On retrouve à la fois une complexité au niveau de l'interface et dans les calculateurs, cette complexité qui est due à l'intégration des notions de redondance, ségrégation et diversification.

Tableau V.6 Description de l'architecture du système des différentes approches

Les valeurs de la colonne «niveau de criticité» est donné ici à titre indicatif et ne constitue pas une valeur exacte suite à des tests qu'on aurait menés. Nous partons du principe que la conception zéro défaut correspondrait à une utilisation actuelle de l'interactivité qui est classé non-critique, donc à un niveau de type Majeur (10^{-5}) selon la classification de la DO178B. Ensuite l'approche mixte de tolérance aux fautes est faite pour respecter les interactions catastrophiques devant respecter le critère *Fail Safe* et selon le tableau de classification de la DO178B, le niveau catastrophique requiert un objectif probabiliste $< 10^{-9}$. Ainsi la tolérance aux fautes reposant sur le système ou sur l'utilisateur se situe donc au niveau dangereux de la classification (10^{-7}).

La Figure V.22 présente l'évolution de la difficulté de la tâche dont nous avons vu la formule ci-dessus en fonction des approches proposées. Elle est tracée en fonction des résultats sur l'interaction en entrée du Tableau V.5 et de la criticité du système du Tableau V.6.

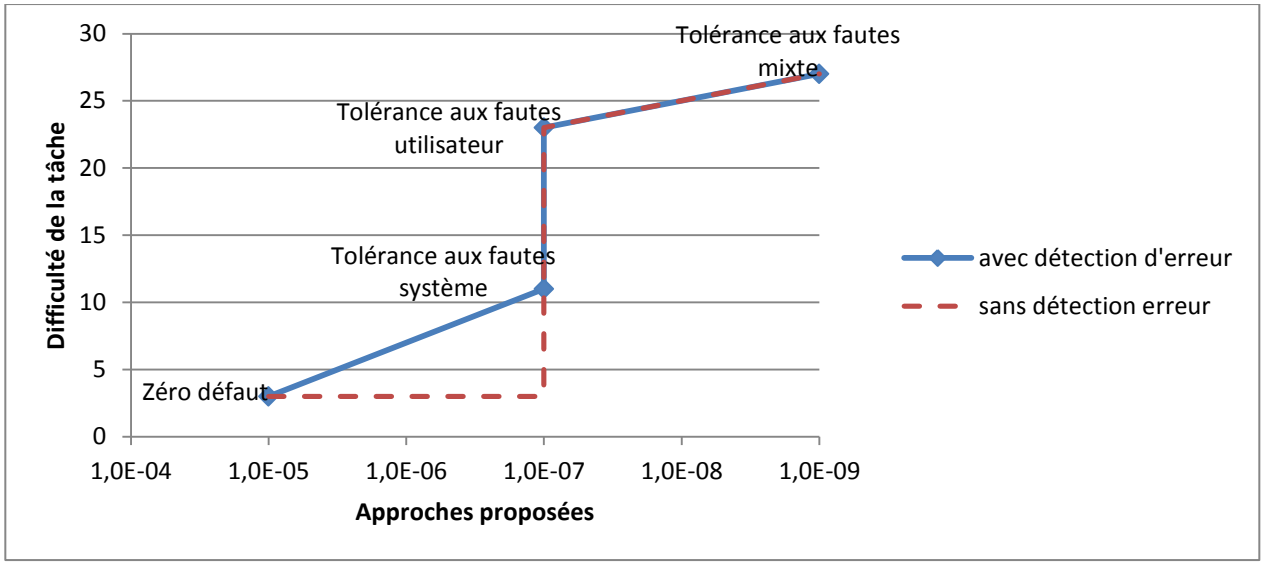


Figure V.22 Evolution de la difficulté de la tâche en fonction de criticité de système

On remarque ainsi que généralement plus la criticité augmente plus la charge de travail de l'utilisateur augmente aussi. Lorsque la tolérance aux fautes repose sur le système, l'utilisateur n'est pas impacté par la détection d'erreur. Néanmoins pour atteindre un critère *Fail Safe* il est nécessaire d'utiliser toutes les ressources du système interactif donc d'intégrer aussi l'utilisateur dans la détection.

Ces courbes ont été réalisées par rapport aux activités courante de l'utilisateur sur un système interactif de cockpit (saisir une donnée, réaliser une commande, lire une donnée), elles permettent donc d'illustrer l'impact que peut avoir l'implémentation des techniques de tolérance aux fautes sur l'ensemble du système interactif, donc à la fois sur le cœur de calcul et sur l'utilisateur.

Il est évident que l'augmentation de la charge de travail de l'utilisateur peut contribuer à l'augmentation de la probabilité d'occurrence des erreurs humaines. Un exemple classique est celui des fenêtres de confirmation, s'il y a trop de demandes de confirmation, l'utilisateur a tendance à ne plus lire le texte afficher et confirmer l'action.

Ces résultats mettent en avant la nécessité d'analyser conjointement la sûreté de fonctionnement et l'utilisabilité du système (Navarre , et al., 2010). La description des modèles de tâche sur les différentes architectures tolérante aux fautes du système interactif est une première approche pour mener cette étude.

Conclusion générale et perspectives

La capacité d'interagir avec les écrans via un clavier et un dispositif de pointage a été introduite récemment dans les systèmes d'affichage et de contrôle des cockpits d'avion civil. Cette capacité d'interaction est à ce jour limitée à des fonctions non critiques, et pour l'étendre à des fonctions critiques, il faut répondre à des exigences de sûreté de fonctionnement liées au développement des systèmes avioniques critiques, mais aussi s'assurer que l'intégration des exigences de sûreté de fonctionnement n'impacte pas sur l'utilisabilité du système.

Pour répondre à ces besoins, nous avons proposé différentes approches pour contribuer au développement d'un tel système interactif critique. La première approche est de tendre vers une conception zéro défaut et ceci au travers de l'utilisation d'une technique de description formelle. La deuxième approche est l'intégration de mécanisme de tolérance aux fautes et la troisième approche est l'explicitation des deux premières approches sur la charge de travail de l'utilisateur. Nous avons ensuite montré la faisabilité de ces approches en réalisant leurs applications sur un cas d'étude et en se reposant sur l'utilisation du formalisme d'Objet Coopératif Interactif (ICO).

Nous nous sommes intéressés aux composants logiciels du système interactif et plus particulièrement aux widgets. Les widgets étant les composants logiciels de base du système et de l'interaction avec l'utilisateur. L'approche zéro défaut a consisté à utiliser le formalisme ICO afin de décrire de façon précise le comportement des widgets et aussi de l'application avion gérant l'animation des widgets (UA). On a ainsi décrit les états des widgets en fonction des évènements de l'utilisateur sur les périphériques d'entrées et des commandes venant de l'application avion (UA), l'ensemble des états dans lequel peut se trouver l'UA et les évènements auxquels elle peut réagir. L'utilisation de technique formelle est importante pour les systèmes critiques, elles présentent l'avantage de fournir une description complète, précise et non ambiguë du système et permet aussi la vérification et la validation des propriétés du système lors des phases de conception, facilitant ainsi son implémentation. L'utilisation d'une description précise aide les concepteurs à une bonne compréhension du fonctionnement du système, à identifier les fautes possibles et à mettre en place les barrières de sécurité adéquates. Cette première approche est une contribution importante, puisqu'aujourd'hui la description du comportement des widgets est généralement faite de façon textuelle et peut conduire à des ambigüités donc à l'introduction de fautes lors des phases de conception. L'application de cette approche a été présentée au Chapitre I.

Une spécification précise du comportement et un processus de développement rigoureux permet d'éliminer un maximum de fautes lors des phases de conception, mais ne garantit pas d'avoir un système parfait. Il est donc important d'intégrer des techniques de tolérance aux fautes afin de traiter les fautes résiduelles de conception, les fautes matérielles ou de l'environnement du système. Nous avons donc proposé trois options de tolérance aux fautes pour les systèmes interactifs :

- La première option est la tolérance aux fautes reposant sur l'utilisateur, elle consiste principalement à réaliser de la redondance dans l'affichage et à effectuer des confirmations pour valider une action. On retrouve cette technique sur la plupart des interfaces des systèmes interactifs actuels.

- La deuxième option est la tolérance aux fautes reposant sur le système, elle consiste principalement en la conception de widget autotestable. Nous y avons décrit les différents modes de défaillances des widgets et modéliser l'architecture autotestable constituée d'un bloc fonctionnel et d'un contrôleur à l'aide du formalisme ICO. Cette notion de widget autotestable est une proposition nouvelle dans le domaine de l'interaction homme machine.
- La troisième option est une approche mixte de la tolérance aux fautes reposant à la fois sur le système et sur l'utilisateur. Elle est utilisée pour les fonctions très critiques classées catastrophiques.

Nous avons ensuite réalisé les modèles de tâches de l'utilisateur en fonction des différentes options de tolérance aux fautes et avons ainsi constaté que plus le niveau de criticité de l'interaction augmente, plus la charge de travail de l'utilisateur s'en trouve impactée. Cette analyse conjointe de la sûreté de fonctionnement et de l'utilisabilité au travers des modèles de tâches est une autre contribution importante de nos travaux. C'est une proposition concrète permettant d'analyser plus tôt lors de la conception du système les aspects de sûreté de fonctionnement et d'utilisabilité, ces deux aspects étant généralement traités de façon indépendante. L'application de cette approche de tolérance aux fautes a été décrite au Chapitre I.

Nous pouvons ainsi résumer notre contribution en **une approche à base de modèles pour le développement d'un système interactif sûr de fonctionnement et avec analyse de l'impact sur l'utilisabilité**.

Perspectives

Ce travail s'inscrit en préambule d'un vaste projet de recherche sur les cockpits du futur (CORAC). Il pose les axes de réflexions pour le développement de systèmes interactifs critiques.

Comme nous l'avons expliqué, les approches proposées sont principalement à base de modèles et n'ont pas été appliquées sur des systèmes réels. Ces modèles permettent de spécifier, concevoir et valider les principes en vue d'une implémentation future. Une première perspective serait justement d'analyser l'intégration des widgets autotestables dans un environnement réel du système d'affichage et de contrôle de cockpit d'avion. L'architecture que nous avons proposée du widget autotestable intégrait déjà la possibilité de ségréguer le bloc fonctionnel et le bloc contrôleur, cette ségrégation dans un système réel peut être faite soit au vue de l'ARINC 653 avec une ségrégation spatiale et temporelle au sein d'un même composant matériel, soit sur des composants matériels différents. Il faudrait ensuite analyser l'impact de l'intégration de widgets autotestables sur l'architecture et sur les performances du système. Une réelle campagne d'injection de fautes devra ensuite être réalisée afin de mieux voir la couverture de détection de fautes du contrôleur et de l'ajuster si nécessaire.

L'autotestabilité a été appliquée ici au niveau du widget, il faudrait également étudier leur impact dans le cas d'agrégation de widgets. Par exemple sur les widgets de type regroupement, certains possèdent des propriétés qui impactent leurs widgets enfants, il serait donc intéressant de voir les répercussions d'une détection d'erreur sur ces widgets au niveau de l'ensemble de l'interface, mais aussi analyser les effets de cascade d'erreurs.

Une évolution peut également être apportée sur l'outil Petshop pour intégrer des vues de l'architecture au sens ICompoNet afin de mieux voir les liens de communication entre les différents modèles. Il n'est pas toujours facile de naviguer entre les réseaux de Petri. Cette évolution permettrait d'avoir une vue hiérarchique du système interactif, allant d'une vision de haut niveau représentant juste l'architecture à une vision de bas niveau représentant en détail le comportent d'un composant du système.

L'utilisation d'une technique de description formelle telle qu'ICO présente l'avantage de permettre la vérification et la validation des propriétés de comportement du système. Ces analyses sont d'autant plus importantes pour l'avionneur qui doit démontrer la complétude et la pertinence de la validation et de la vérification des exigences écrites en langage formel aux autorités de certification. L'outil Petshop qui est l'environnement d'édition des ICO, permet à ce jour seulement l'analyse des invariants de places et de transitions. Il serait également intéressant d'ajouter plus d'algorithme de preuves de propriétés, telles que les graphes de marquage, vérifier l'absence de blocage des réseaux de Petri et les aspects de contrôle de propriétés temps réel. Ajouter également une capacité de génération de code pourrait être une piste à envisager comme support à l'implémentation.

Côté utilisabilité une analyse plus fine des tâches et des erreurs humaines doit être faite. Des travaux récents ont déjà été mené afin d'intégrer l'outil Petshop et l'outil Hamster (Navarre, et al., 2010) permettant d'avoir en parallèle l'exécution du comportement du système et les tâches réalisées par l'utilisateur. Ajouter plus d'analyse sur les erreurs humaines, et les indices de charge de travail de l'utilisateur à ne pas dépasser serait intéressant. L'on sait déjà que selon les phases de vol, la charge de travail de l'équipage n'est pas la même. Les phases de décollage et d'atterrissage requièrent beaucoup de manœuvres et d'attention par rapport à la phase de croisière, les techniques de tolérance aux fautes pourraient être adaptées selon ces différentes phases du vol. On proposerait par exemple d'avoir une tolérance aux fautes reposant sur le système lors des phases où la charge de travail de l'équipage est élevée pour ne pas l'impacter dans la réalisation de ses tâches, et dans les phases ou sa charge de travail est basse, l'intégrer dans la surveillance des systèmes.

Liste des publications

Tankeu-Choitat A., D. Navarre, P. Palanque, Y. Deleris, J-C Fabre **An approach for assessing both usability and dependability of interactive systems: application to interactive cockpits**. In proceedings of HCI Aero 2010 conference. Cape Canaveral, USA, Nov. 1-3 2010. pp. 123-135.

Adrienne Tankeu-Choitat, Jean-Charles Fabre, Philippe Palanque, David Navarre, Yannick Deleris. **Self-Checking Components for Dependable Interactive Cockpits** (regular paper). 13th European Workshop on Dependable Computing (EWDC 2011), Pisa, ACM DL ISBN 978-1-4503-0284-5.

Adrienne Tankeu-Choitat, Jean-Charles Fabre, Philippe Palanque, David Navarre, Yannick Deleris, Camille Fayollas. **Self-Checking Components for Dependable Interactive Cockpits using Formal Description Techniques. The 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2011), Pasadena, USA, Dec. 12-14, 2011.**

Arnaud Hamon, Philippe Palanque, Célia Martinie, David Navarre, Adrienne Tankeu-Choitat and Eric Barboni. **Interactive Cockpits Applications: Modelling and Validation using a Petri-net based Formalism. Embedded Real Time Software and Systems (ERTS2), Toulouse, France, Feb 1-2 2012**

Bibliographie

Abbott Kathy Human Factors Engineering and Flight Deck Design [Report]. - [s.l.] : CRC Press LLC, 2001.

Abrams M [et al.] UIML: An Appliance-Independent XML User Interface Language. In A. Mendelzon,editor [Conference] // 8th International World-Wide Web Conference WWW‟8. - Amsterdam : Elsevier Science Publishers, 1999.

ACM SIGHI Curriculum for Human-Computer Interaction [Book]. - New York : ACM special interest Group on Computer-Human Curriculum Development Group, 1992.

AFDX ARINC 664 P2-2 Aircraft data network Part 2-Ethernet physical and data link layer specification [Book]. - [s.l.] : Aeronautical Radio Inc, 2000.

Annett K and Duncan J Task Analysis and Training Design [Journal] // Journal of Occupational Psychology. - 1967. - Vol. 41. - pp. 211-221.

Appert C and Beaudouin-Lafon M SwingStates: adding state machines to the swing toolkit [Conference] // UIST'06. - Montreux : ACM, 2006.

Appert caroline [et al.] FlowStates: Prototypage d'applocations interactives avec des flots de données et des machines à états [Conference] // IHM09, French speaking sonference on Human-Computer interaction. - Grenoble : ACM Press, 2009.

ARINC653 Avionics Application Standard Software Interface [Book]. - [s.l.] : Aeronautical Radio, 1997.

ARINC661 Cockpit Display System Interfaces to User Systems [Book]. - [s.l.] : Prepared by Airlines Electronic Engineering Committee, 2010. - Vol. ARINC 661 specification 4.

Arlat J [et al.] Fault Injection for Dependability Validation, A methodology and some Applications [Article] // IEEE Transactions on Software Engineering. - 1990.

Arlat Jean [et al.] “Fault Tolerance”, in Encyclopedia of Computer Science and Information Systems (J. Akoka, I. Comyn-Wattiau, Coordinators) — Part 1: The Technological Dimension of Information Systems, Section 2: Architecture and Systems (M. Banâtre, Editor) [Book]. - [s.l.] : Vuibert, Paris (France), 2006. - Vols. ISBN: 2-7117-4846-4. (in French)..

ARP4754 Certification Considerations for Highly-Integrated Or Complex Aircraft [Report]. - 1996.

ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil [Report].

Avizienis A The N-Version Approach to Fault-Tolerant Software [Journal]. - [s.l.] : IEEE Trans. Softw., 1985. - Vol. 12.

Azevedo P, Merrick R and Roberts D OVID to AUIML - user-oriented interface modelling [Conference] // 1st International Workshop ”Towards a UML Profile for Interactive Systems Development” TUPIS‟00. - York : [s.n.], 2000.

Ball T [et al.] Several interfaces, single logic [Report]. - Chicago : Technical report, Loyola University, 2000.

Barboni E. [et al.] Addressing Issues Raised by the Exploitation of Formal Specification Techniques for Interactive Cockpit Applications [Journal]. - 2006.

Barboni E. [et al.] Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification [Conference] // DSVIS 2006. - 2006. - pp. 25-38.

Barboni Eric [et al.] A Formal Description Technique for Interactive Cockpit Applications Compliant with ARINC Specification 661 [Conference] // Industrial Embedded Systems, SIES '07 . - Lisbon : IEEE, 2007.

Barboni eric Méthodes formelles pour les composants logiciels appliqués aux systèmes interactifs critiques [Book]. - Toulouse : Université Toulouse III, 2006.

Bastide R. and Palanque P. Petri nets with objects for specification, design and validation of user-driven interfaces [Conference] // Interact 90. - 1990. - pp. 27-31.

Bastide rémi Thèse: Objets Cooperatifs: un formalisme pour la mmodélisation des système concurrents [Report]. - Toulouse : Université Paul Sabatier, 1992.

Baumeister L.K, John B.E and Byrne M.D A Comparison of Tools for Building GOMS Models Tools [Conference] // ACM Conference on human Factors in Computing Systems CHI 2000. - [s.l.] : ACM, 2000. - pp. 502-509.

Berry Gérard [et al.] ESTEREL: A formal method applied to avionic software development [Journal] // Science of Computer Programming. - 2000. - pp. 5-25.

Berry Gérard and Gonthier Georges The Esterel Synchronous Programming Language: Design, Semantics, Implementation [Journal] // Science of Computer Programming vol. 19. - 1992. - pp. 87-152.

Berry Gérard The foundations of esterel [Conference] // In Proof, language, and interaction. - Cambridge, MA, USA : MIT Press, 2000. - pp. 425-454.

Berry Richard E Common User Access - A consistent and Usable human-computer interface for the SAA environments [Book]. - [s.l.] : IBM Systems Journal, 1988. - Vols. 27, N°3.

Berthelot, G., Roucairol, G. and Valk, R. Reductions of Nets and Parallel Programs, [Conference] // Proceedings of the Advanced Course on General Net Theory of Processes and Systems: Net Theory and Applications. - 1979. - pp. 277-290.

Bias Randolph G and Mayhew Deborah J Cost-Justifying Usability, an Update for the Internet Age [Book]. - [s.l.] : Elsevier, Inc., 2005.

Blanch R and Beaudouin-Lafon M Programming rich interactions using the hierarchical state machine toolkit [Conference] // the Working Conference on Advanced Visual interfaces AVI '06. - Venezia : ACM, 2006.

Blandford Ann [et al.] Analytical usability evaluation for digital librairies: a case study [Conference] // In proceedings of the 4th ECM/IEEE-CS joint conference on Digital librairies (JCDL 04) . - [s.l.] : ACM, 2004.

Bourget M.L A Toolkit for Creating and Testing Multimodal Interface Designs [Conference] // UIST'02. - Paris : [s.n.], 2002.

Brooke J SUS: a quick and dirty usability scale [Article] // Usability Evaluation in industry. - London : Taylor and Francis, 1996.

Buchholz, P. and Kemper, P. Hierarchical Reachability Graph Generation for Petri Nets. Form. Methods Syst [Report]. - 2002.

Büchi Martin and Weck Wolfgang A plea for grey-box components [Conference] // Workshop on Foundations of Component-Based Systems. - Zürich : [s.n.], September 1997.

Buxton W A three-state model of graphical input [Conference] // IFIP Tc13 Third interational Conference on Human-Computer interaction. - Amsterdam : North-Holland Publishing Co, 1990.

Card Stuart K, Moran Thomas P and Newell Allen The Model Human Processor: An Engineering Model of Human Performance [Article] // Handbook of Perception and Human Performance. - 1986. - Vol. Vol 2: Cognitive Processes and performance. - pp. 1-35.

Collins D Designing Object-oriented user interfaces [Conference]. - Readwoods City : CA: Benjamin/Cummings Publishing Inc.., 1995.

CORAC [Online] // http://www.aerorecherchecorac.com.

Coutaz Joëlle and Nigay Laurence A Generic Platform for Addressing the Multimodal Challenge [Conference] // CHI'95. - Denver : ACM, 1995.

COUTAZ Joelle Interface Homme-Ordinateur: conception et Réalisation. [Book]. - [s.l.] : Dunod Informatique, 1990.

Coutaz Joëlle PAC, on Object Oriented Model for Dialog Design [Conference] // Interact'87. - 1987.

CRIF15 EASA Certifictaion review item F-15 : Software formalised requirements validation and verifictaion [Report]. - [s.l.] : EASA, 2010.

Diaz M Les réseaux de Petri : Modèles fondamentaux ouvrage collectif sous la direction de Michel Diaz [Book]. - [s.l.] : Hermes Science Publications, 2001.

DO-178B Software Considerations in Airborne Systems and Equipment Certification [Report]. - [s.l.] : Radio Technical Commission for Aeronautics (RTCA) European Organization for, 1992.

DO-254 Design Assurance Guidance for Airborne Electronic Hardware [Report]. - [s.l.] : Radio Technical Commission for Aeronautics (RTCA), 2000.

Dragicevic P and Fekete J Support for input adaptability in the ICON toolkit [Conference] // ICMI'04. - [s.l.] : ACM, 2004.

DRAGICEVIC Pierre Un modèle d'interaction en entrée pour des systèmes interactifs multi-dispositifs hautement configurables [Book]. - Nantes : Université de Nantes, 2004.

Duke D and Harrison M. D Abstract interaction objects [Conference] // Eurographics '93. - Oxford : Blackwell Publishers, 1993. - pp. 25-36.

Dumas B [et al.] Strengths and weaknesses of software architectures for the rapid creation of tangible and multimodal interfaces [Conference] // Tangible and Embedded interaction TEI '08. - Bonn, germany : ACM, 2008.

EASA certification specifications, including airworthiness codes and acceptable means of [Book]. - [s.l.] : European Aviation Safety Agency, 2008.

EJB ExpertGroup JSR 220: Entreprise JavaBeans, version 3.0 [Book]. - Santa Clara, California : Sun Microsystems INC, 2006.

Endres Albert An analysis of errors and their causes in system programs [Conference] // In Proceedings of the international conference on Reliable software. - [s.l.] : ACM, 1975. - pp. 327-336.

Esteban O and Chatty S Whizz'Ed: a visual environment for building highly interactive interfaces [Conference] // Interact'95. - 1995.

ESTEREL Language [Online]. - http://www-sop.inria.fr/meije/esterel/esterel-eng.html.

Fabre Jean-Charles [et al.] Self-Checking Components for Dependable Interactive Cockpits using Formal Description Techniques [Conference] // The 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2011). - Pasadena, USA : IEEE, 2011.

Farenc Christelle Thèse de doctorat: ERGOVAL : une méthode de structuration des règles ergonomiques permettant l'évaluation automatique d'interfaces graphiques [Report]. - Toulouse : Université de Toulouse, 1997.

Fichefet J Introduction aux réseaux de Petri [Report]. - université de Namur : Support de cours, 1995.

Fitts P.M The information capacity of the human motor system in controlling the amplitude of the movement. [Journal]. - [s.l.] : Journal of Experimental Psychology, 1954. - Vol. 47. - pp. 381-391.

Goldberg Adele and Robson David Smalltalk-80: The Language and its Implementation [Journal]. - Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1983.

Green Mark A survey of three dialogue models [Journal] // ACM Trans. Graph. - 1986. - pp. 244-275.

Gulliksen J and Goransson B Usability Design: Integrating User Centered System Design in the Software Development Process [Conference] // IFIP INTERACT03. - Zurich, Switzerland : [s.n.], 2003. - p. 1025.

Halbwachs Nicolas [et al.] Programmation et Vérification des systèmes réactifs: Le langage Lustre [Journal] // Technique et Science Informatique. - 1991. - pp. 139-158.

Halbwachs Nicolas [et al.] The synchronous dataflow programming language Lustre [Journal] // Proceedings of the IEEE. - 1991. - pp. 1305-1320.

Hartmann Jan, Sutcliffe Alistair and De Angeli Antonella Towards a theory of user judgment of aesthetics and user interface quality [Conference] // ACM Trans. Compu-Hum. Interact. - 2008.

Hartson R and Hix D Human-computer interface developement: Concepts and systems for its management [Conference] // in 1989. - 1989. - pp. 5-92.

ISO/IS8807 LOTOS a Formal Description Technique Base on temporal Ordering of Observational Behaviour [Report]. - 1988.

ISO9241-11 Exigences ergonomiques pour travail de bureau avec terminaux à écrans de visualisation (TEV) – Partie 11: lignes directrices relatives à l'utilisabilité [Report]. - [s.l.] : ISO Central Secretariat, 1998.

Jaffe Matthew Saul Completeness, Robustness, and Safety in Real-Time Software Requirements Specifications: a Logical Positivist Looks at Requirements Engineering [Report]. - Irvine, CA, USA : University of California at Irvine, 1988.

Jiao, L, Cheung, T. and Lu, W. Characterizing Liveness of Petri Nets in Terms of Siphons. In Proceedings of the 23rd international Conference on Applications and theory of Petri Nets [Conference] // Lecture Notes In Computer Science. - London : Springer-Verlag, 2002. - Vol. 2360..

Johnson C.W An introduction to human error, Interaction and the development of Safety-Critical Systems [Article] // A Handbook of Human-Machine Interaction: A Human Centred Design Approach. - Kent, UK : [s.n.], 2011. - G. Boy.

Johnson P and Johnson H Knowledge Analysis of Task: Task Analysis and Specification for Human- [Journal] // Engineering the Human Computer Interface. - Maidenhead, : [s.n.], 1989.

Kieras D and Polson P.G A generalized transition network representation for interactive systems [Conference] // CHI'83. - Boston : ACM, 1983.

König W.A, Rädle R and Reiterer H Interactive Design of Multimodal User Interfaces - Reducing technical and visual complexity [Journal]. - [s.l.] : Springer, Journal on Multimodal User Interfaces, 2010.

Krasner Glenn E and Pope Stephen T A cookbook for using the model-view controller user interface paradigm in Smalltalk-80 [Conference]. - 1988. - pp. 26-49.

Ladry Jean-François Une notion et un processus outillé pour le développement de systèmes interactifs multimodaux critiques [Report]. - Toulouse : Université de Toulouse, 2010.

Laprie Jean-Claude [et al.] Basic Concepts and Taxonomy of Dependable and Secure Computing [Journal]. - [s.l.] : IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, 2004. - Vol. 1.

Laprie Jean-Claude [et al.] Definition and Analysis of hardware and software fault Tolerant Architectures [Journal]. - [s.l.] : Computer 23, 1990. - Vol. 7. - pp. 39-51.

Laprie Jean-Claude [et al.] Guide de La Sûreté De Fonctionnement [Book]. - [s.l.] : Cepadues, 1996.

Lee P.A and Anderson T Fault Tolerance: Principles and Practice [Report]. - [s.l.] : Springer-Verlag New York, Inc, 1990.

Leveson nancy G Safeware: System Safety and Computers [Book]. - [s.l.] : Addison-Wesley, 1995. - Vols. ISBN: 0-201-11972-2.

Limbourg Q [et al.] USIXML: A Language Supporting Multi-path Development of User Interfaces [Conference] // In proceedings of EHCI-DSVIS 2004. - [s.l.] : LNCS, 2004.

Littlewood Bev and Strigini Lorenzo Software Reliability and Dependability: a Roadmap [Conference] // Proceedings of the Conference on The Future of Software Engineering (ICSE '00). - [s.l.] : ACM, 2000. - pp. 175-188.

Luyten K and Coninx K An XML-based runtime user interface description language for mobile computing devices [Conference] // Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS‟2001. - Glasgow : Springer Verlag, 2001.

Martinie De Almeida Célia [et al.] Model-Based Training: An Approach Supporting Operability of Critical Interactive Systems: Application to Satellite Ground Segments [Conference] // EICS 2011, Engineering Interactive Computing Systems. - Pisa, Italy : ACM Press, 2011.

Mayhew Deborah J Principles and guidelines in software User Interface Design [Book]. - [s.l.] : Prentice-Hall, Englewood Cliffs (New Jersey), 1992.

McDermid J Life cycle support in the Ada environment [Book]. - [s.l.] : Ada Lett., 1983. - Vol. III.

Müller A, Forbrig P and Cap C H Model-based user interface design using markup concepts [Conference] // In Proc. Of 8th International Workshop on Design, Specification, Verification of Interactive Systems DSV-IS‟2001. - Glasgow : Springer-Verlag, 2001.

Myers Brad and Rosson mary beth survey on user interface programming [Conference] // In Human factors in computing systems, proceedings SIGGHI. - Monterey : ACM, 1992. - pp. 195-202.

Navarre David [et al.] An Approach for assessing both usability and dependability of interactive systems: application to interactive cockpits [Conference] // HCI Aero 2010. - Cape Canaveral, USA : ACM, 2010.

Navarre D. [et al.] A Formal Description of Multimodal Interaction Techniques [Conference] // Interact 2005. - 2005. - pp. 170-183.

Navarre D. [et al.] A Tool Suite for Integrating Task and System Models through Scenarios [Conference] // DSV-IS 2001. - Glasgow : [s.n.], 2001. - pp. 88-113.

Navarre D. [et al.] On the Benefit of Synergistic Model-based Approach for Safety Critical Interactive System Testing [Conference] // TAMODIA2007. - 2007.

Navarre D. Contribution à l'ingénierie en Interaction Homme-Machine: une technique de description formelle et un environement pour une modélisation et une exploitation synergique des tâches et du système // PhD Thesis. - 2001.

Navarre D., Bastide R. and Palanque P. A tool-supported design framework for safety critical interactive systems [Journal] // Interacting with Computers. - 2003. - 3 : Vol. 15. - pp. 309-328.

Navarre D., Palanque P. and Bastide R. A Formal Description Technique for the Behavioural Description of Interactive Applications Compliant with ARINC 661 Specification [Conference] // HCI'Aero 2004. - 2004.

Navarre David [et al.] A model-based tool for interactive prototyping of highly interactive applications [Conference] // the 12th IEEE International Workshop on Rapid System Prototyping. - [s.l.] : IEEE Press, 2001.

Navarre David [et al.] An Architecture and a Formal Description Technique for the Design and Implementation of Reconfigurable User Interfaces [Conference] // Design Specification and Verification of Interactive Systems, DSVIS'08. - Kingston, Ontario, Canada : Springer-Verlag Berlin, 2008.

Navarre David [et al.] Formal Description Techniques for Human-Machine Interfaces - ModelS-Based Approaches for the Design and Evaluation of Dependable Usable Interactive Systems. [Book]. - [s.l.] : Handbook of Human-Machine Interaction. Ashgate., 2010.

Navarre David [et al.] ICOs: A Model-Based User Interface Description Technique dedicated to Interactive Systems Addressing Usability, Reliability and Scalability [Conference] // ACM Transactions on Computer-Human Interaction (TOCHI). - [s.l.] : ACM, 2009.

Navarre David Contribution à l'ingénierie en Interaction Homme-Machine: une technique de description formelle et un environement pour une modélisation et une exploitation synergique des tâches et du système // PhD Thesis. - 2001.

Nielsen Jakob Usability Engineering [Book]. - San Francisco : Morgan Kaufman, 1993.

Nigay Laurence and Coutaz Joëlle A design space for multimodal systems [Conference] // INTERCHI '93. - 1993. - pp. 172-178.

OMG Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 [Book]. - [s.l.] : Object Management Group Inc, 2008.

OMG CORBA [Online] // http://www.omg.org/spec/CORBA.

Palanque P. [et al.] Supporting Usability Evaluation of Multimodal Man-Machine Interfaces for Space Ground Segment Applications Using Petri net Based Formal Specification. [Conference] // SpaceOps 2006. - Rome : [s.n.], 2006. - p. CDRom.

Palanque P. and Bastide R. Synergistic Modelling of Tasks, Users and Systems using Formal Specification Techniques [Journal] // Interacting with Computers. - 1997. - 2 : Vol. 9. - pp. 129-153.

Palanque P. and Bastide R. Verification of an Interactive Software by analysis of its formal specification [Conference] // Interact'95. - Lillehammer : [s.n.], 1995. - pp. 191-197.

Palanque P., Bastide R. and Paternò F. Formal Specification as a Tool for Objective Assessment of Safety-Critical Interactive Systems [Conference] // INTERACT 97. - 1997. - pp. 323-330.

Palanque P., Navarre D. and Gaspard-Boulin H. MEFISTO MEthod version 1 [Report]. - 2000.

Palanque Philippe [et al.] A tool supported Model-Based Approach for engineering usability Evaluation of interaction techniques [Conference] // EICS 2011. - Pisa, Italy : ACM Press, 2011.

Palanque Philippe [et al.] Supporting Usability Evaluation of Multimodal Man-Machine Interfaces for Space Ground Segment Applications Using Petri net Based Formal Specification. [Conference] // SpaceOps 2006. - Rome : [s.n.], 2006. - p. CDRom.

Palanque Philippe Modélisation par Objets Cooperatifs Interactifs d'interfaces homme -machine [Report]. - Toulouse : Université Toulouse, 1992.

Palmer Michael [et al.] A Crew-Centered Flight Deck Design Philosophy for High-Speed Civil Transport (HSCT) Aircraft [Report]. - Hampton, Virginia : NASA, 1995.

Paternò F and Faconti G On the use of LOTOS to describe graphical interaction [Conference] // In Proceedings of the HCI'92 Conference on People and Computers VII. - 1992.

Paternò F. and Santoro C. Preventing user errors by systematic analysis of deviations from the system task model [Journal] // Int. J. Hum.-Comput. Stud.. - [s.l.] : Academic Press, Inc., 2002. - 2 : Vol. 56. - pp. 225--245.

Paternò F., Breedvelt-Schouten I. M. and de Koning N. Deriving Presentations from Task Models [Conference] // Proceedings of the IFIP TC2/TC13 WG2.7/WG13.4 Seventh Working Conference on Engineering for Human-Computer Interaction. - 1999. - pp. 319-337.

Paternò F., Santoro C. and Tahmassebi S. Formal Models for Cooperative Tasks: Concepts and an Application for EnRoute Air Traffic Control [Conference] // DSVIS 1998. - 1998. - pp. 71-86.

Peterson W.W and Weldon E. J Error-Correcting Codes [Journal]. - Cambridge, MA, USA. : MIT Press , 1972.

Petri Karl Adam Kommunikation mit Automaten [Report]. - [s.l.] : Technical University Darmstadt ed, 1962.

Pfaff Günther E. User interface management [Conference] // In Workshop on User Interface Management Systems. - Secaucus, NJ, USA : Springer-Verlag, 1985.

Platt David S Introducing Microsoft .Net, Third Edition [Book]. - [s.l.] : Microsoft Press, 2003.

Poupart P, Basnyat S and Palanque P A Model-Based Approach Centred on Operational Procedures for the Development of Reliable and Usable Ground Segment Systems [Conference] // SpaceOps. - Heidelberg, germany : [s.n.], 2008.

Preece J. [et al.] Human-Computer Interaction [Book]. - 1994. - 0201627698.

Puerta A and Eisenstein J A common representation for interaction data [Conference] // In Proc. Of the 7th International Conference on Intelligent User Interfaces. - Santa Fe : ACM press, 2002.

Rabéjac Christophe Auto-surveillance logicielle pour applications [Book]. - Toulouse : Thèse en informatique, Institut National, 1995.

Randel B System structure for software fault tolerance [Conference] // International conference on Reliable software. - Los Angeles, California : ACM, 1975. - pp. 437-449.

Rauterberg M An Iterative-Cyclic Software Process Model [Conference] // SEKE'1992. - 1992. - pp. 600-607.

Reason James L'erreur Humaine [Book]. - [s.l.] : PUF, Presses Universitaires de France, 1993.

Rieder R, Raposo A.B and Pinho M.S A methodology to specify three-dimensional interaction using Petri Nets [Journal]. - [s.l.] : J. Vis. Lang. Comput, 2010. - 21.

Romero Miguel and de Lara Juan VALIDACIÓN Y VERIFICACIÓN DE INTERFACES DE USUARIO EN EL ÁMBITO DEL DESARROLLO BASADO EN MODELOS [Conference] // XV Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2006. - Barcelona : © CIMNE, 2006.

Scade Scade Display [Online]. - http://www.esterel-technologies.com/products/scade-display/.

Scapin D and Pierret-Golbreich C Towards a Method for Task Description: MAD. [Conference] // the Work with display units Conference WWU'89. - Amsterdam : Elsevier Science Publishers, 1989.

Sibert J.L, Hurley W.D and Bleser T.W An object-oriented user interface management system [Conference] // SIGGRAPH '86. - New York : ACM, 1986.

Souchon N and Vanderdonckt J A review of XML compliant user interface description languages [Article] // Lecture notes in computer science In Design, specification and verification - interactive systems. - 2003. - Vol. Vol. 2844. - pp. 377-391.

Sun Microsystems JavaBeans API specification [Book]. - [s.l.] : Graham Hamilton, 1997. - Vol. version 1.01.

Szyperski Clemens Component Software : Beyond Object-Oriented Programming [Book]. - Boston,MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002.

Tankeu Choitat Adrienne [et al.] Self-Checking components for dependable Interactive Cockpits [Conference] // 13th European Workshop on Dependable Computing (EWDC 2011). - Pisa, Italy : ACM, 2011.

Tatsukawa K Graphical toolkit approach to user interaction description [Conference] // CHI'1991. - New York : ACM, 1991.

Traverse P, Lacaze I and Souyris J Airbus Fly-By-Wire: A Total [Conference] // Proceedings 18th IFIP World Computer Congress,. - 2004. - pp. 191-212.

UIMS A metamodel for the runtime architecture of an interactive system [Conference] // The UIMS Tool Developpers Workshop, SIGCHI. - 1992. - pp. 32-37.

Van der Veer G, Van der Lenting B. F and Bergevoet B.A Groupware Task Analysis - Modeling Complexity [Journal] // Acta Psychologica . 91. - 1996. - pp. 297-322.

Vanderdonckt Jean Guide ergonomique des interfaces homme-machine [Book]. - Namur : Presses Universitaires de Namur, 1994.

Vanderdonckt Jean Règles ergonomiques de sélection d'objet interactifs pour une information simple [Conference] // 6ième Colloque Ergonomie et Informatique Avancée. . - Biarritz : [s.n.], 1998.

Wasserman anthony I User software engineering and the design of interactive systems [Conference] // In proceeding of the 5th international conference on software engineering (ICSE'81). - Piscataway, NY, USA : IEEE Press, 1981. - pp. 387-393.

Wiegmann D.A and Shappell S.A A human Error Approach to Aviation Accident Analysis: The Human Factors analysis and Classification System. [Book]. - [s.l.] : Farnham, UK: Ashgate Publishing, 2003.

Willans J.S and Harrison M.D Prototyping Pre-implementation Designs of Virtual Environment Behaviour [Conference] // the 8th IFIP international Conference on Engineering For Human-Computer interaction. - London : Springer-Verlag, 2001.

Yeh Y.C Triple-Triple Redundant 777 Primary Flight Computers [Conference] // Proceedings IEEE Aerospace Applications Conference,. - Aspen, CO : [s.n.], 1996. - pp. pp. 293-307..

Résumé

Depuis l'A380 et avec l'introduction du standard ARINC 661, les systèmes d'affichage et de contrôle des cockpits sont passés d'un rôle de simple afficheur, à celui d'un système interactif permettant à l'équipage d'interagir sur les écrans grâce à l'utilisation d'un ensemble clavier/dispositif de pointage appelé KCCU. L'utilisation de cette nouvelle capacité d'interaction est à ce jour limitée à des interactions avec des systèmes avions non critiques. Pour envisager son extension à des systèmes critiques il faut se poser la question du respect d'exigences de sureté de fonctionnement imposées à de tels systèmes sans pour autant diminuer son niveau d'utilisabilité. Dans cette optique, nous proposons dans le cadre de nos travaux de recherche, différentes approches pour contribuer au développement d'un tel système interactif critique. La première approche est de tendre vers une conception zéro défaut, en réalisant une description précise et non ambigüe des composants logiciels du système interactif en utilisant une technique de description formelle. La seconde approche est l'utilisation de techniques de tolérance aux fautes car il existe toujours des fautes résiduelles de conception, des fautes matérielles ou venant de l'environnement. Dans ce cas, l'utilisation de technique de tolérance aux fautes permet au système de continuer à remplir ses fonctions en dépit de l'occurrence de fautes. La troisième approche est l'explicitation de l'impact des différentes approches de tolérance aux fautes sur l'utilisabilité du système interactif. Cette explicitation est faite au travers de la réalisation et de l'analyse des modèles de tâche, décrivant l'activité de l'utilisateur du système.

Mots-clés: Système interactif, Tolérance aux fautes, widgets, description formelle

Abstract

Since the Airbus A380 and with the introduction of ARINC 661 standard, the glass cockpits are being replaced by interactive cockpits, by allowing the crew to control aircraft systems through display unit by using keyboard and cursor control unit (KCCU). Currently only secondary aircraft systems which are non-critical are managed using such interactive cockpits. To be able to generalize such features to critical aircraft system, the main question remains to understand how to match dependability requirements for such systems while preserving usability properties. To reach the goal of using such interactive techniques within safety critical aircraft systems, our research work has followed three main directions. The first approach is to tend to zero default design, by realizing the precise and unambiguous description of software components of interactive system, using formal description technique. The second approach consists in the use of fault tolerant mechanisms, to treat design residual fault, physical fault or environmental fault. These fault tolerant mechanisms enable the continuity of service despite the occurrence of fault. The third approach is the clarification of the impact of different fault tolerant mechanisms on the usability of the interactive system. This clarification is done by using and analyzing task models, describing the user activity of the system.

Keywords: Interactive system, fault-tolerance, widgets, formal description