Résilience des systèmes interactifs: contribution par une architecture tolérante aux fautes

Camille Fayollas, Philippe Palanque, Jean-Charles Fabre, David Navarre, Eric Barboni, Martin Cronel, Yannick Deleris

Résilience des systèmes interactifs: contribution par une architecture tolérante aux fautes

Camille Fayollas[2,3], Philippe Palanque[3], Jean-Charles Fabre[2,4], David Navarre[3], Eric Barboni[3], Martin Cronel[3] & Yannick Deleris[1]
[1]AIRBUS Operations, 316 Route de Bayonne, 31060, Toulouse, France
Yannick.Deleris@airbus.com
[2]CNRS, LAAS, 7 Av. Du Colonel Roche, F-31400 Toulouse, France
Jean-Charles.Fabre@laas.fr
[3]ICS-IRIT, University of Toulouse, 118 Route de Narbonne, F-31062, Toulouse, France
{barboni, cronel, fayollas, navarre, palanque}@ irit.fr
[4]Univ de Toulouse, INP, LAAS, F-31400 Toulouse, France

RESUME
La recherche pour améliorer la fiabilité des systèmes interactifs s'est, pour l'instant, principalement dirigée vers la prévention d'introduction de fautes par la suppression des bugs lors du développement. Cependant, la complexité des systèmes interactifs est telle que, quels que soient ces efforts, des défaillances apparaissent lors de leur utilisation. Les causes de telles défaillances peuvent être dues à des pannes matérielles intermittentes ou (quand ces systèmes sont utilisés en haute altitude (avions, véhicules spatiaux …) être déclenchées par des fautes naturelles dues aux particules alpha dans les processeurs ou aux neutrons issus des radiations cosmiques. Cet article propose l'identification exhaustive des problématiques se posant pour améliorer la fiabilité des systèmes interactifs en prenant en compte à la fois les erreurs faites dans les phases de développement et les fautes qui peuvent se produire lors de leur utilisation. Du fait qu'aucune recherche n'a été menée à ce jour pour détecter et éliminer les fautes naturelles, l'article propose aussi une architecture logicielle pour intégrer des mécanismes de tolérance aux fautes dans les systèmes interactifs. L'article présente en particulier comment cette architecture couvre les différents composants d'un système interactif incluant les widgets, le gestionnaire de fenêtres et l'application interactive. Ces concepts sont mis en œuvre sur une application interactive de cockpits d'avions civils.

Mots Clés
Tolérance aux fautes, Architecture logicielle, Résilience, Systèmes interactifs critiques.

ACM Classification Keywords
D.2.2 [Software] Design Tools and Techniques - Computer-aided software engineering (CASE), H.5.2 [Information Interfaces and Presentation]: User Interfaces - Interaction styles

http://dx.doi.org/10.1145/2670444.2670462

INTRODUCTION

Un système critique est un système où le coût d'une potentielle défaillance serait largement supérieur au coût de développement. Il pourrait ainsi aller jusqu'à engendrer des blessures ou des pertes de vies humaines [15] (ils sont alors appelés « safety-critical » en anglais). Les systèmes interactifs critiques sont maintenant présents dans la plupart des stations de contrôle et commande comme par exemple dans les segments sol des satellites, les cockpits d'aéronefs civil ou militaire, les postes de contrôle aérien etc… La complexité et la quantité de données manipulées, le nombre de systèmes à contrôler et la grande diversité des commandes à déclencher dans des périodes de temps limité ont poussé l'introduction de techniques d'interaction évoluées dans la plupart de ces systèmes.

Construire des systèmes interactifs fiables est une tâche fastidieuse en raison de leur nature très spécifique. Le comportement de ces systèmes réactifs est piloté par événements. Comme ces événements sont déclenchés par des opérateurs humains, ces systèmes réactifs doivent réagir à des événements inattendus. En sortie, des informations (telles que l'état actuel du système) doivent être présentées à l'opérateur de manière à ce qu'elles puissent être perçues et interprétées correctement. Enfin, les systèmes interactifs exigent d'aborder les aspects matériels et logiciels ensembles (par exemple les dispositifs d'entrée et de sortie ainsi que leurs drivers).

Dans le domaine de la tolérance aux fautes, les expériences ont démontré (e.g. [35]) que les défaillances logicielles pouvaient survenir même si le développement du système avait été extrêmement rigoureux. L'une des nombreuses sources de défaillance est appelée faute naturelle [4] ; elles sont déclenchées par les particules alpha ou encore par des neutrons provenant des rayonnements cosmiques et impactent les puces. Les systèmes déployés en haute atmosphère ainsi que ceux déployés dans l'espace (e.g. vol spatiaux habités [25])

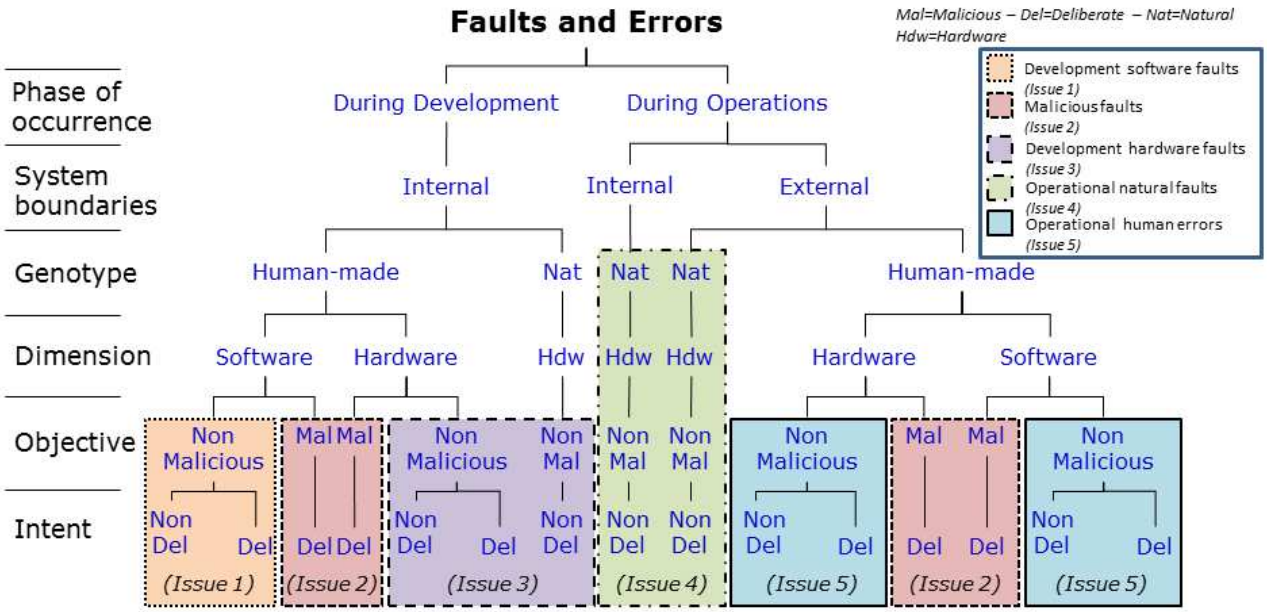

Figure 1. Typologie des fautes des systèmes informatiques (adaptée de [4]) et problématiques associées à leur fiabilité

ont les plus grandes probabilités de défaillance [46] (e.g. aéronefs). Bien que des moyens existent pour se prémunir contre la plupart de ces fautes, elles démontrent le besoin d'aller plus loin que les techniques habituelles de prévention d'occurrences de fautes durant le développement du système (généralement l'utilisation de techniques formelles et la vérification de propriétés) mais aussi celui d'identifier toutes les menaces pouvant impacter les systèmes interactifs.

Cet article traite, dans la première section, de l'identification d'une liste de problématiques qui se posent pour améliorer la fiabilité des systèmes interactifs critiques. La deuxième section traite de l'identification de plusieurs solutions existantes pour répondre à ces problématiques. Une partie de ces problématiques a été en effet étudiée précédemment par la communauté de recherche sur les systèmes interactifs comme par exemple les erreurs provenant des opérateurs. La troisième section porte sur une des questions qui a rarement été étudiée dans ce domaine de recherche : les fautes naturelles qui se produisent indépendamment de l'effort déployé au cours des phases de développement et propose une architecture logicielle tolérante aux fautes pour remédier à ces défauts. Afin d'illustrer les concepts versity Press, 1990. applicabilité, la quatrième section du du document présente leur utilisation dans le cadre de l'application de Flight Control Unit présente dans les cockpits modernes interactifs d'avions civils. La cinquième section permet de positionner l'approche proposée dans cet article par rapport aux contributions précédentes des auteurs. Enfin, la dernière section conclut le papier et présente les perspectives de ce travail par rapport aux problématiques présentées dans les premières sections.

PROBLEMATIQUES SOULEVEES PAR LA FIABILISATION DES SYSTEMES INTERACTIFS

Typologie des fautes

Pour s'assurer qu'un système informatique soit fiable [4], c'est-à-dire qu'il soit protégé de tout type de défaillances afin que celles-ci ne puissent pas apparaître ou que leur apparition n'ait pas de conséquence catastrophique sur le système et son environnement, le concepteur de système doit prendre en compte tout type de fautes pouvant impacter le système et déclencher ces défaillances. Dans le but de classifier intégralement les fautes pouvant affecter un système informatique, Avizienis et al. ont défini dans [4] une typologie. Cette typologie a permis d'identifier 31 classes de fautes élémentaires. La figure 1 présente une vue simplifiée de cette typologie et rend explicite deux grandes catégories, visibles en haut de la figure : i) les fautes apparaissant pendant le développement (ce qui inclut les mauvaises conceptions et les erreurs de développement) ou ii) les fautes apparaissant pendant le fonctionnement du système (à droite de la figure), ce qui inclut les erreurs comme les ratés, les lapsus et les fautes définis dans [42].

Nous proposons l'organisation des feuilles de la taxonomie en 5 différents groupes amenant chacun une problématique différente :

- *Fautes de développement logiciel (problématique 1)* : introduites involontairement durant le développement du système.
- *Fautes malveillantes (problématique 2)* : introduites délibérément pour provoquer le dysfonctionnement du système (prise de contrôle extérieur, déni de service inopiné, crash du système).

- *Fautes de développement matériel (problématique 3)* : ayant une cause naturelle ou humaine et impactant le matériel durant sa conception.
- *Fautes naturelles en opération (problématique 4)* : causées par un phénomène naturel. Elles affectent le matériel et surviennent pendant le fonctionnement du système, elles sont donc susceptibles d'affecter aussi le logiciel.
- *Erreurs humaines en opération (problématique 5)* : résultantes d'une action humaine pendant le fonctionnement du système. Elles peuvent être matérielles et logicielles et peuvent être délibérées ou non. Le lien entre la typologie et les erreurs humaines dites « classiques » définies dans [42] peut être fait en associant les fautes délibérées aux erreurs ou violations [40] et les non-délibérées avec les ratés et les lapsus.

Approches existantes de prise en compte des fautes

Les études précédentes en sûreté de fonctionnement ont mené à l'identification de 4 moyens d'améliorer la fiabilité d'un système [4] et [15]:

- *La prévention des fautes*: permet d'éviter autant que possible l'introduction de fautes pendant le développement du système. On emploie généralement des techniques de développement rigoureuses, de la formalisation, de la modélisation pour prouver certaines propriétés, …
- *La suppression de fautes* : permet de réduire le nombre de fautes pouvant survenir: i) pendant le développement du système, généralement en utilisant des techniques de test et d'injection de fautes ou ii) pendant la phase d'exploitation du système en faisant par exemple de la maintenance corrective.
- *La tolérance aux fautes*: permet d'éviter la défaillance de service en présence de fautes via leur détection et leur recouvrement. La détection permet d'identifier la présence de fautes, leur type et leur source. Le recouvrement est généralement réalisé en ajoutant de la redondance ou de la diversification d'un logiciel et permet d'éviter la défaillance.
- *Prévision des fautes* : consiste à estimer le nombre, l'incidence et les conséquences probables de fautes, généralement en dressant une évaluation statistique de la fréquence et de l'impact des fautes.

APPROCHES DE FIABLISATION DES SYSTEMES INTERACTIFS

Comme vu dans la section précédente, la fiabilisation d'un système informatique peut être réalisée si et seulement si toutes les problématiques présentées ci-dessus sont couvertes. La suite de cette section présente les différentes approches existantes pour les traiter tout en prenant en compte les spécificités des systèmes interactifs.

Problématique 1 : Prise en compte des fautes de développement logiciel

Du fait des spécificités des systèmes interactifs [51], les approches standard d'ingénierie des logiciels ne peuvent pas être réutilisées telles quelles pour rendre les systèmes interactifs fiables. De ce fait, dans la communauté de l'ingénierie des systèmes interactifs, de nombreux travaux de recherche ont été réalisés pour raffiner et étendre ces approches. Ces travaux incluent les architectures logicielles [8], les techniques de description formelle et de vérification [33, 24, 9, 16, 31, 14], ou le test [12, 10, 37]. La plupart de ces travaux s'est concentrée sur la suppression des fautes durant le développement du logiciel.

Dans le domaine des cockpits interactifs (que nous présentons dans l'étude de cas), Barboni et al. ont proposé dans [5] de telles méthodes pour décrire de manière complète et non ambiguë les widgets, les gestionnaires de fenêtres et les contrôleurs de dialogue ; couvrant ainsi la partie logicielle des systèmes interactifs. Ceci est également en accord avec le standard de développement DO 178B/C qui recommande l'utilisation de techniques de description formelle [44].

Problématique 2 : Prise en compte des fautes malveillantes

Cet aspect a été très étudié dans le domaine des interactions homme-machine, et a conduit à la création d'un symposium sur la protection de la vie privée appelé SOUPS (Symposium on Usable Privacy and Security) ; où sont discutées les questions d'utilisabilité des systèmes d'authentification [52]. Les questions d'intégrité et de protection des données sont laissées aux domaines liés à la sécurité informatique [43].

Cependant, les problèmes de sécurité informatique spécifiquement liés aux systèmes interactifs critiques n'ont pas encore été traités. De plus, ils commencent tout juste à être abordés dans le domaine des systèmes embarqués, comme présenté dans [18] pour les systèmes embarqués avioniques.

Problématique 3 : Prise en compte des fautes de développement matériel

Les composants matériels utilisés pour les systèmes interactifs sont assez similaires à ceux utilisés dans les systèmes informatiques conventionnels. La prise en compte des fautes de développement matériel peut donc être traitée avec des processus de développement usuels, comme ceux décris dans le standard DO-254 [19], qui fournit un guide de développement des composants électroniques embarqués.

Cependant, dans le cas d'un système interactif, on doit prendre en compte la fiabilité de l'interaction en plus de la fiabilité du système informatique de base. Par exemple, l'introduction d'une interaction multi-touch dans un cockpit ne pourra être réalisée que s'il a été prouvé que l'écran possède une faible probabilité de défaillance (e.g. moins de 10^{-9} fautes par heure de vol).

Le concepteur du système interactif pourra néanmoins augmenter la fiabilité d'un système interactif en adaptant l'interface ou le logiciel sous-jacent. Par exemple, une interaction tactile peut être fiabilisée en remplaçant un geste linéaire horizontal par un geste oblique afin de ne pas dépendre uniquement d'une ligne de la grille tactile.

Problématique 4 : Prise en compte des fautes naturelles

Comme expliqué dans l'introduction, la prise en compte des fautes naturelles n'a pratiquement pas été étudiée dans le domaine de l'interaction homme-machine et peu de contributions sont disponibles. La proposition de [32] présente la reconfiguration dynamique de la technique d'interaction ou la possibilité de la réorganisation des écrans quand les défaillances matérielles le nécessitent.

Cependant, cette solution n'est pas suffisante et ces aspects ne peuvent être ignorés lorsque l'on traite de la fiabilité de systèmes interactifs critiques. Ceci est particulièrement vérifié pour les systèmes interactifs critiques tels que ceux présents dans les systèmes déployés dans la haute atmosphère (avions) ou dans l'espace (vol spatiaux habités) car une plus forte probabilité d'occurrence de fautes concerne ces systèmes [46,25]. Ces fautes naturelles démontrent la nécessité d'aller plus loin que la prévention classique des fautes durant le développement en prenant en compte les fautes pouvant apparaitre en opération.

Problématique 5 : Prise en compte des erreurs humaines

Les erreurs humaines sont étudiées depuis de nombreuses années et peuvent être prévenues ou tolérées [17] et de nombreux moyens ont d'ores et déjà étés développés pour la prise en compte de cette problématique. Un premier moyen est de réaliser des barrières [26], qu'elles soient logicielles (e.g. un bouton de confirmation) ou matérielles (e.g. un détrompeur). Bien entendu, les connaissances acquises de l'étude des accidents et des tentatives de violations de barrières doivent être réutilisées pour prévenir la répétition des accidents [7]. Ceci peut-être effectué, par exemple, en incluant des barrières sociotechniques qui pourront être modélisées par des techniques de description formelle afin de prévenir les fautes de développement [6]. Les erreurs humaines peuvent aussi être évitées par la formation spécialisée comme décrit dans [30] ainsi que par un meilleur design de l'aide contextuelle [38] et des manuels d'utilisation [13]. Enfin, un autre moyen de prévenir ces erreurs est d'étudier le comportement humain pour modifier le design des systèmes [41], [36] ou en proposer de meilleurs [49].

Remarques conclusives sur les approches de fiabilisation des systèmes interactifs

La typologie présentée nous permet d'identifier toutes les fautes pouvant affecter un système interactif critique. Elle permet le groupement de certains types de fautes et met donc en exergue les problématiques spécifiques à la fiabilisation des systèmes interactifs critiques. Dans cette section, nous avons présenté les différentes solutions existantes relatives aux problématiques identifiées. Bien que certaines d'entre-elles aient été largement étudiées, d'autres, en revanche, l'ont été beaucoup moins et nécessitent d'être traitées afin de permettre le développement de systèmes interactifs fiables :

- *Problématique 2 :* Les fautes provenant d'attaques malveillantes,
- *Problématique 3 :* Les fautes matérielles de développement,
- *Problématique 4 :* Les fautes naturelles en opération.

Dans le domaine de la sûreté de fonctionnement, la problématique 2 est principalement résolue par confinement, i.e. complète isolation des systèmes entre eux et plus particulièrement, isolation du reste du monde. Cette solution peut s'avérer problématique compte tenu des maintenances à effectuer qui requerraient alors un accès physique au système. De plus, les communications sont de plus en plus sans fil et ouvrent naturellement les systèmes sur l'extérieur, éliminant ainsi le confinement. Pour ces raisons, la problématique 2 reste à étudier prioritairement. La problématique 3 dépend principalement des constructeurs et du domaine des composants. La problématique 4 a, quant à elle, un fort potentiel d'impacter les systèmes interactifs en opération et nous y consacrerons donc le reste de cet article.

UNE ARCHITECTURE LOGICIELLE POUR DES APPLICATIONS INTERACTIVES TOLERANTES AUX FAUTES

Cette section présente une solution permettant de construire des applications interactives tolérantes aux fautes. Elle propose ainsi une solution à la problématique des fautes naturelles en opération (problématique 4). Dans ce but, nous proposons d'appliquer, en les adaptant, des architectures de tolérance aux fautes aux différents éléments d'une application interactive en se basant sur le modèle ARCH introduit ci-dessous (cf. Figure 2).

Une architecture générique pour les systèmes interactifs

Dans le cadre de cet article, nous nous intéressons particulièrement aux techniques d'interaction classiques (appelées WIMP) pour lesquelles l'affichage et la manipulation se font au travers d'un ensemble prédéfini de widgets. Même si de nombreuses techniques d'interaction bien plus sophistiquées ont déjà été proposées par les chercheurs et industriels du domaine des interfaces homme-machine, la manipulation indirecte reste la plus adaptée aux systèmes critiques car elle correspond aux standards des milieux critiques, comme par exemple le standard ARINC 661 qui définit les interactions dans les cockpits d'avions civils [3]. De plus, les techniques d'interaction indirectes ont déjà été étudiées et standardisées par IBM en 1989 [27].

De telles standardisations présentent de nombreux avantages améliorant significativement la fiabilité (les composants sont réutilisés et donc très largement testés) ; les coûts de développement (les composants sont développés par des entités tierces et sont donc utilisés tel quel par de nombreux client) ainsi que les temps de développement (les développeurs se concentrant alors sur l'assemblage des composants et non sur leur design).

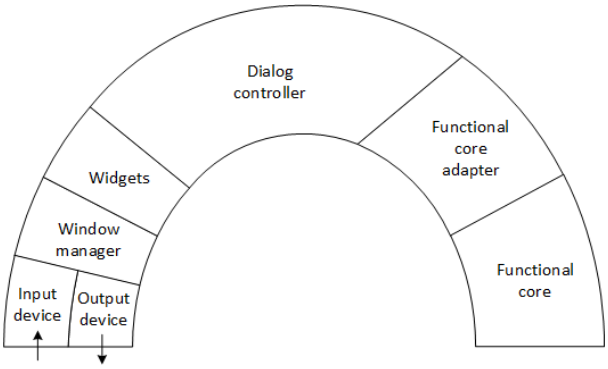

Figure 2. Architecture générique pour les systèmes interactifs adaptée du modèle ARCH [8][1]

La Figure 2 présente une version modifiée de l'architecture logicielle ARCH présentée dans [8]. Cette architecture décrit l'organisation fonctionnelle de la majorité des systèmes interactifs. Elle est utilisée pour représenter de manière exhaustive la variété d'éléments pour lesquels des mécanismes de sûreté de fonctionnement ont été ajoutés:

- *Les périphériques d'entrée et de sortie*: ce sont généralement des claviers, souris et écran… mais ils peuvent être plus complexes (e.g. périphérique combiné comme le KCCU (Keyboard Cursor Control Unit) dans les cockpits). Ils permettent l'interaction physique entre l'humain et la machine.
- *Le gestionnaire de fenêtres* : il encapsule les drivers des périphériques et gère le lien entre les périphériques et le reste de l'application. Par exemple, il est responsable de l'affichage du curseur, de l'identification du widget ciblé par l'action de l'utilisateur (picking), la transmission des événements utilisateurs vers celui-ci ainsi que du rendu graphique de l'information sur le périphérique de sortie.
- *Les widgets*: ce sont les composants de base de l'interaction. Ils sont représentés dans une boîte séparée sur la Figure 2 car nous les considérons comme des composants logiciels indépendants requérant ainsi un mécanisme de tolérance aux fautes dédié comme celui introduit dans [48].
- *Le contrôleur de dialogue*: il décrit les états de l'application ainsi que son comportement.
- *L'adaptateur de code fonctionnel* et *le noyau fonctionnel* ils encapsulent les fonctions non-interactives du système. Pour cela, nous ne nous concentrerons pas sur ces composants bien qu'ils apparaitront dans l'étude de cas.

La section suivante est une courte introduction aux architectures tolérantes aux fautes utilisées dans le domaine de la sûreté de fonctionnement. Nous présenterons ensuite les hypothèses sous-jacentes au travail présenté ainsi qu'une description des fautes que nous cherchons à éviter.

Travaux connexes sur les architectures tolérantes aux fautes

Dans le domaine de la sûreté de fonctionnement, de nombreuses approches ont été étudiées pour introduire des mécanismes de tolérance aux fautes au sein des systèmes informatiques. Les techniques les plus utilisées sont : les composants auto-testables [29, 53, 50], la programmation en N-Version [54] et la programmation N-auto-testable [29] (qui est un hybride entre la programmation N-Version et les composants auto-testables). L'architecture présentée dans ce papier est basée sur les composants auto-testables comme de nombreuses stratégies de sûreté de fonctionnement en dépendent. En effet, la couverture de détection d'erreurs d'un composant auto-testable est très bonne et rend les stratégies de réplication possibles afin de pouvoir assurer une continuité de service. C'est d'ailleurs dans ce but que nous proposons également une architecture N-auto-testable qui permet le recouvrement de fautes.

Selon [29], une approche par composants auto-testables consiste à ajouter de la redondance interne à un programme pour qu'il puisse vérifier sa propre dynamique en exécution. Elle est aussi appelé COM-MON [50] ; COM correspondant à COMmande et MON à MONiteur. Cette redondance peut être implantée de 2 manières différentes :

1. Un programme (le COM) auquel on ajoute un programme vérifiant son exécution (le MON, habituellement appelé checker dans ce cas).
2. Un programme (le COM – aussi appelé fonctionnel dans ce cas) auquel on adjoint une variante de ce programme (habituellement appelée contrôleur) et un mécanisme de comparaison ; le MON est alors composé de l'ensemble contrôleur/comparateur.

Dans les deux cas, les fautes naturelles sont détectées par les mécanismes de redondance présents dans le MON si les deux composants logiciels sont exécutés de manière ségréguée, pour éviter les fautes engendrées par les dégâts collatéraux (une faute dans le COM déclenchant une faute dans le MON) et les fautes de mode commun (une faute naturelle impactant à la fois le COM et le MON). L'approche par composants auto-testables a donc la capacité d'assurer la détection des fautes et la

[1] De même qu'en ce qui concerne les aspects tolérance aux fautes, nous avons décidé de garder le vocabulaire anglais pour les figures de l'article dans le but d'éviter les confusions liées à la traduction.

notification d'erreurs lorsque les fautes sont détectées (e.g. quand il y a une différence de comportement entre le COM et le MON).

Le recouvrement de fautes n'est pas possible avec cette approche et il faut donc l'étendre. Ceci peut être fait en utilisant les concepts des composants N-auto-testables qui peuvent être considérés comme de la redondance active, comme expliqué dans la section 5.2 de [11].

Hypothèses d'étude et principales défaillances prises en compte

Cette étude porte sur la fiabilisation du logiciel par rapport aux fautes naturelles, nous considèrerons donc que les erreurs humaines sont hors du champ d'application. Cette hypothèse est très forte mais viable puisque l'on peut considérer que les erreurs humaines n'influencent pas les fautes naturelles du système et vice-versa. La contribution se concentre donc sur les défaillances fonctionnelles des trois principales briques de l'architecture logicielle (ARCH) présentée à la Figure 2 i.e. l'application (le contrôleur de dialogue), les widgets et le gestionnaire de fenêtres. Les périphériques d'entrée et sortie ainsi que le noyau fonctionnel sont considérés comme étant hors du champ d'application de cet article, ils sont traités dans d'autres travaux connexes comme [23].

L'architecture que nous proposons permet de s'assurer que le système interactif traite correctement les événements d'entrée venant de l'opérateur, et effectue correctement le rendu des paramètres reçus du noyau fonctionnel. Pour rentrer dans les détails, nous cherchons à gérer les trois défaillances fonctionnelles possibles qui pourraient mener à une catastrophe (e.g. un crash d'avion) :

- **Affichage erroné**: affichage erroné des données reçues du noyau fonctionnel (e.g. un widget reçoit 10,11 et affiche 23,58);
- **Contrôle erroné**: Transmission d'une action différente de celle accomplie par l'utilisateur (e.g. l'utilisateur clique sur le bouton 1 et l'événement MouseMove est transmis à l'application);
- **Contrôle autonome**: un événement est transmis alors que l'utilisateur n'a pas accompli d'action sur le système (e.g. un événement MouseClick est transmis sans aucune intervention de l'utilisateur).

Description AADL des architectures

Les architectures présentées dans ce papier seront décrites dans cette section en utilisant la représentation graphique du langage AADL (Architecture Analysis and Design Language) [45]. AADL est un langage de description (textuel et graphique) standardisé en 2004 par la société SAE (Society of Automotive Engineers) pour aider à l'analyse et à la conception d'architectures système. Le langage AADL a principalement été utilisé pour décrire les systèmes temps réels complexes dans les systèmes avioniques et spatiaux mais il a aussi été utilisé dans d'autres domaines. Nous l'utilisons dans cet article pour décrire graphiquement les architectures tolérantes aux fautes que nous proposons pour en évaluer la fiabilité. La description est adaptée pour le domaine d'application du cockpit par l'ajout d'une représentation du pilote, de ses interactions avec le système ainsi qu'un symbole «engrenage » pour représenter le traitement des données. La Figure 3 introduit les composants et les connexions en AADL utilisés dans ce papier.

AADL component representation	Meaning of the component
System	**System**: Hierarchical organization of components.
Processor	**Processor**: Hardware component including a CPU, memory, bus ... and allowing software execution.
Virtual Processor	**Virtual processor**: Logical resource allowing software execution.
Bus	**Bus**: Provides physical connectivity between execution platform components.
Device	**Device**: Interface to external environment (typically physical components interfacing with the environment.
Event Data Ports — Output / Input	**Event Data Ports**: Ports allowing directional transfer of queued messages.
Bus access feature	**Bus access feature**: To model required access to shared bus.

Figure 3. Composants AADL utilisés dans l'article

Une architecture tolérante aux fautes pour les composants interactifs

Dans cette section, nous présentons tout d'abord en détail une architecture auto-testable (mécanisme présenté rapidement au-dessus) qui permet d'assurer la **détection** des fautes, puis nous présentons deux architectures permettant leur **recouvrement** : i) une architecture N-auto-testable, où le recouvrement est assuré de manière autonome par le système et ii) une architecture avec recouvrement impliquant l'opérateur.

Une architecture auto-testable pour la détection de fautes appliquée aux composants interactifs

La Figure 4 présente l'architecture d'un composant usuel (Figure 4-a) comparé à un composant auto-testable (Figure 4-b). Cette figure rend explicite les différences entre les deux architectures. Elle est utilisée ici pour présenter la manière d'implanter les composants auto-testables pour des applications interactives. Pour cela, il faut suivre deux étapes : implantation du MON et son confinement par rapport au COM.

- Pour chaque composant interactif, on doit définir un MON et de nombreuses solutions sont disponibles. [47] a proposé d'appliquer un mécanisme auto-testable aux widgets en utilisant la seconde implantation présentée dans les travaux connexes. Il propose aussi l'utilisation d'une spécification formelle [33] pour décrire l'ensemble des composants du widget auto-testable (Dispatcher, COM, MON and Comparateur). Cette solution pose des problèmes d'implantation tels que la

spécification d'un composant qui autorise le dispatching des événements d'entrée. [20] a proposé d'appliquer la philosophie auto-testable à un niveau plus élevé, en utilisant la première implantation présenté dans les travaux connexes. Le challenge est ici la définition du programme (MON) vérifiant l'exécution du COM. Cette solution permet de grouper, par exemple, l'ensemble des moniteurs en un seul moniteur pour l'intégralité de l'application.

- Un mécanisme auto-testable ne suffit pas pour assurer la détection des fautes, il est donc nécessaire de confiner le COM du MON. En effet, une faute impactant un composant (e.g. le COM), peut également affecter le second composant (e.g. le MON). Cela serait le cas si tous les composants d'une architecture étaient exécutés dans la même partition. Le standard ARINC 653 [2] définit un partitionnement qui assure la ségrégation spatiale et temporelle pour le domaine de l'aviation civile. L'architecture proposée dans le papier rend explicite la ségrégation entre le COM et le MON, représentés ici par deux processeurs virtuels différents que l'on peut considérer comme deux partitions ARINC 653 séparées.

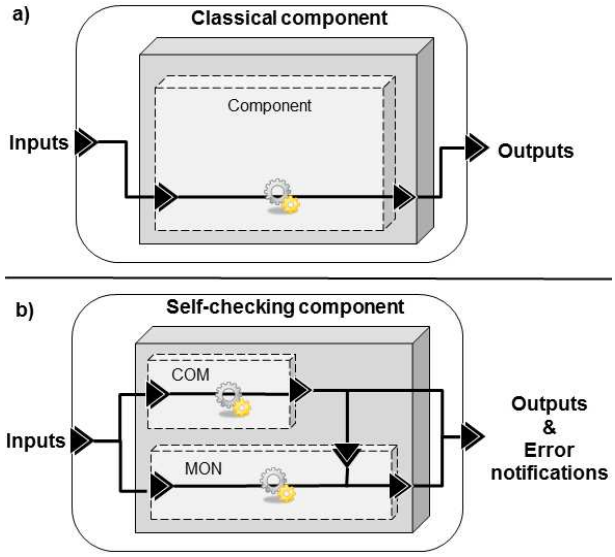

Figure 4. Architecture d'un composant usuel (a) comparée à celle d'un composant tolérant aux fautes (détection seule) (b)

Architectures pour le recouvrement de fautes appliquée aux composants interactifs

Les composants auto-testables sont capables de détecter la présence d'une faute et de déclencher la notification associée. Pour pouvoir ajouter du recouvrement et tolérer ainsi les fautes, plusieurs solutions de conception sont envisageables. Dans cet article nous en proposons deux, qui seront illustrées dans l'étude de cas :

- *L'architecture N-auto-testable (automatique)*
  L'architecture N-auto-testable présentée dans la Figure 5 est composée de deux ou plus composants auto-testables numérotés séquentiellement. Les événements d'entrée sont transmis à tous les composants auto-testables et sont traités en parallèle. Toutes les sorties et notifications d'erreur sont transmises à un mécanisme de décision. Si le premier composant fournit un résultat sans erreur, ce dernier est transmis tel quel en tant que résultat du composant N-auto-testable. Sinon, le mécanisme de décision ne prendra plus en considération le résultat du premier composant auto-testable et prendra à la place les résultats du suivant. Ceci se répète jusqu'à ce que le dernier composant défaille, ce n'est qu'à ce moment qu'une notification d'erreur est envoyée. Si tous les widgets auto-testables sont défaillants, le widget N-auto-testable ne fait plus que de la détection de faute, sans faire de recouvrement. Si un des widgets auto-testables traite une donnée sans erreur, la faute précédente est alors détectée, recouverte, et ne sera pas notifiée.
  Cette architecture est particulièrement efficace car elle permet la diversification des composants auto-testables, ce qui pourrait se faire en les faisant développer par différentes équipes avec différents langages de programmation. Cette possibilité de diversification rend cette architecture très précieuse même si elle en devient naturellement plus chère à réaliser. Un autre aspect important repose sur le fait que le recouvrement est automatique et ne nécessite pas de support extérieur.
- *L'architecture impliquant l'utilisateur*
  Il est possible de faire du recouvrement d'une autre manière en introduisant un mécanisme spécifique de gestion des notifications d'erreur par l'utilisateur. Dans ce cas, le principe est de notifier l'opérateur lorsqu'une faute est détectée par le système, laissant le recouvrement à la charge de l'opérateur. Cette approche présente l'avantage de garder l'opérateur dans la boucle de contrôle et lui permet donc de garder une marge de décision pour la résolution des fautes. Cependant, la charge de travail supplémentaire rend problématique cette approche lors de l'apparition de multiples fautes dans un temps limité. On voit ainsi apparaitre un problème de conflit entre utilisabilité et fiabilité. Il est possible d'évaluer l'impact en termes d'utilisabilité (et plus précisément en termes d'efficience) en décrivant et en analysant la complexité des tâches liées au recouvrement. Une approche systématique pour évaluer ce type d'impact a été testée sur 4 architectures de tolérance aux fautes différentes et peut être trouvée dans [22].

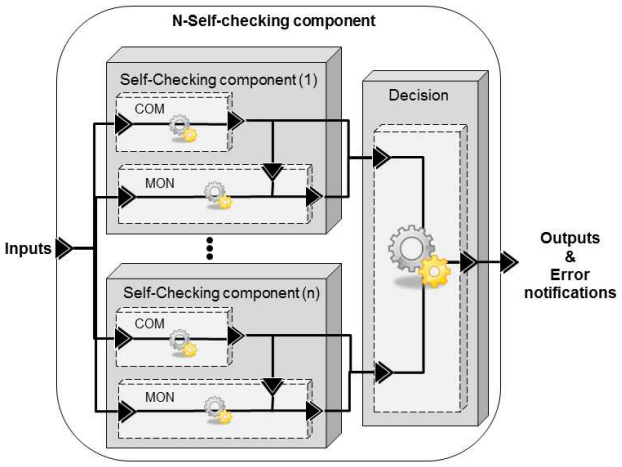

Figure 5. Une architecture tolérante aux fautes (incluant la détection et le recouvrement) d'un composant interactif

Une architecture pour des systèmes interactifs tolérants aux fautes

Lorsque l'on considère la fiabilité des systèmes informatiques, la fiabilité de l'ensemble du système est celle de l'élément le moins fiable. Nous avons proposé précédemment des architectures pour les composants interactifs capables d'effectuer la détection et le recouvrement des fautes. Si l'on suit l'architecture ARCH de la Figure 2, de tels mécanismes doivent être définis pour tous les composants i.e. les périphériques d'entrée et de sortie, les drivers, le gestionnaire de fenêtres, les widgets et l'application, c.à.d. le dialogue et le noyau fonctionnel.

L'ajout de mécanismes de tolérance aux fautes à ces composants soulève des problèmes du fait des contraintes temporelles imposées par la (courte) boucle de feedback immédiat nécessaire au gestionnaire d'événement des périphériques d'entré. Nous travaillons sur ces aspects, à la fois dans le contexte des interactions WIMP et des interactions directes et muli-touch [23].

Comportement détaillé d'éléments interactifs tolérants aux fautes

Les travaux connexes proposent habituellement des solutions génériques ayant un haut niveau d'abstraction comme celles que nous avons proposées dans la section précédente. Il faut bien sûr raffiner ces solutions lors des phases d'implantation pour permettre de décrire en détail les connexions entre les périphériques d'entrée-sortie et chacun des composants, ainsi que les connexions entre ces composants eux-mêmes. Il faut ensuite produire une description bas niveau de la variété des composants de l'architecture. De tels comportements peuvent être relativement complexes et leur intégration bien plus encore. [47] a démontré que l'utilisation de descriptions formelles dédiées aux systèmes interactifs, comme ICO [33] permet de décrire le comportement des widgets auto-testables. Une des difficultés est alors la production d'une seconde version du comportement pour respecter la diversification dans le but d'éviter les fautes de mode commun entre le COM et le MON comme détaillé dans [48].

APPLICATION DE L'ARCHITECTURE DE TOLERANCE AUX FAUTES A UNE ETUDE DE CAS

Pour illustrer l'approche proposée, nous présentons une mise en œuvre de l'architecture auto-testable sur une application interactive de cockpit appelée FCU Backup (Flight Control Unit Backup).

Description informelle

L'application FCU Backup est conçue pour permettre à l'équipage technique (pilote, copilote) d'interagir avec l'autopilote (AP) et de configurer les affichages de vol et de navigation. Cette application est affichée sur deux des afficheurs LCD du cockpit (un pour chaque pilote). L'équipage peut ainsi interagir avec l'application en utilisant un dispositif appelé Keyboard and Cursor Control Unit (KCCU en bas et à gauche de la Figure 6) qui combine un trackball et un clavier dans un unique composant matériel.

Architecture tolérante aux fautes pour le FCU Backup

La Figure 6 présente une vue détaillée de l'architecture (suivant l'architecture ARCH – cf. Figure 2) du FCU Backup, où le périphérique d'entrée est le KCCU et le périphérique de sortie est l'écran LCD du cockpit. Le gestionnaire de fenêtres permet de les connecter aux

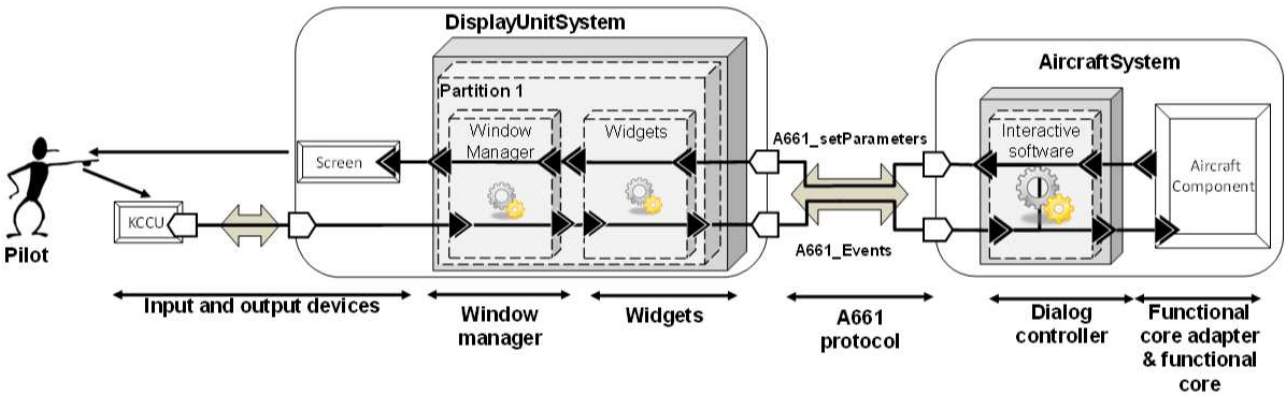

Figure 6. Architecture non tolérante aux fautes de l'étude de cas

widgets qui interagissent avec le contrôleur de dialogue via le protocole défini par le standard ARINC 661 [3]. Le contrôleur de dialogue est, quant à lui, directement relié aux systèmes avioniques.

Il est important de souligner que l'architecture doit respecter le standard ARINC 661 [3] puisqu'il est le seul standard utilisé dans le domaine des avions commerciaux civils. Il définit les widgets et leurs paramètres mais aussi les protocoles de communication entre les widgets et l'application à laquelle ils appartiennent. Les noms utilisés pour décrire les événements et leurs paramètres sont A661_Events et A661_setParameters comme décrit dans le standard. Dans de précédents travaux [5], chaque composant était modélisé en utilisant le formalisme ICO (excepté pour le gestionnaire de fenêtres pour lequel certaines parties ont été décrites en code JAVA) dans le but de traiter la fiabilité d'une telle application.

La Figure 7 présente l'extension de cette architecture pour la rendre tolérante aux fautes. Cette extension suit précisément l'architecture auto-testable présentée précédemment. Les widgets et le gestionnaire de fenêtres sont ici implantés comme des composants auto-testables auquel on a adjoint un composant MON. Ils sont donc capables d'envoyer des notifications d'erreur au contrôleur de dialogue qui contient des mécanismes de recouvrement. Le contrôleur de dialogue est lui aussi un composant auto-testable composé d'un COM et d'un MON. Du fait des contraintes d'espace, nous ne traiterons pas ici le contenu du composant moniteur. Il peut être implanté comme un moniteur unique ou encore comme une multitude de moniteurs assemblés, par exemple un pour chaque widget et un pour le gestionnaire de fenêtres. Sur notre prototype, nous avons testé les deux implantations présentées dans la section précédente (soit le mécanisme auto-testable automatique et celui mettant en jeu l'utilisateur). Plus d'information sur cette approche peut être trouvée dans [21].

Avec l'architecture mettant en jeu l'utilisateur, lorsqu'une faute est détectée, le système affiche un message pour que l'équipage fasse un redémarrage de l'application. Dans ce cas, si la faute détectée est transitoire (et donc non permanente) elle sera supprimée avec le redémarrage du système.

Avec l'architecture N-auto-testable, les composants logiciels de l'avion intègrent la gestion des erreurs déclenchées par les composants auto-testables tels que les widgets ou le gestionnaire de fenêtres. La Figure 7 représente ce comportement par les liens étiquettés *Error notifications* entre les boîtes des widgets et du contrôleur de dialogue ainsi qu'entre le composant de traitement d'erreur du système avionique situé dans la partition 3 et le MON du contrôleur de dialogue situé dans la partition 2. Les autres composants de l'application interactive (comme les périphériques d'entrée-sortie, le noyau fonctionnel etc.) doivent également être adjoints de mécanismes de tolérance aux fautes pour être résilients face aux fautes naturelles. Ce n'est pas présenté dans ce cas d'utilisation, mais on peut le réaliser en leur appliquant les architectures présentées dans ce papier (architecture auto-testable et N-auto-testable).

POSITIONNEMENT DE L'APPROCHE PAR RAPPORT AUX CONTRIBUTIONS ANTERIEURES

La Figure 7 rend explicite la séparation des composants dans les différentes partitions. Cette séparation est fondamentale pour assurer la tolérance aux fautes si l'on veut pouvoir prendre en compte les fautes communes entre le COM et le MON. La manière d'allouer les composants dans les partitions suivant le standard ARINC 653 [2] a été présentée dans [20]. Cette architecture est générique et peut être exploitée pour la conception de tout système interactif si la tolérance aux fautes fait partie des exigences. Toutefois, le comportement des composants peut-être aussi une source d'erreurs. Dans des travaux antérieurs, nous avons

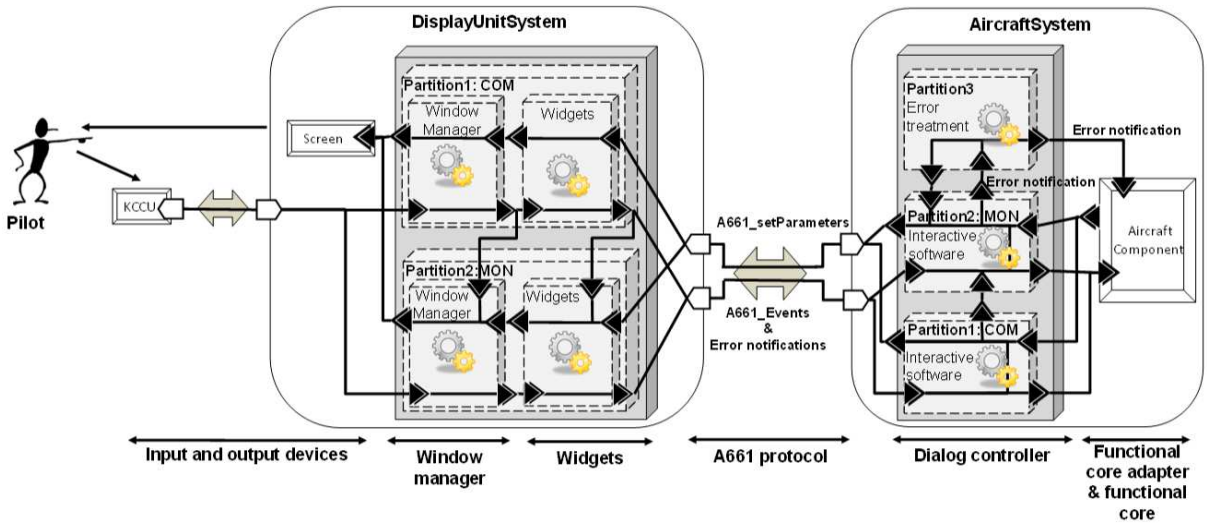

Figure 7. Architecture tolérante aux fautes appliquée à l'étude de cas FCU Backup

proposé une technique de description pour le comportement des widgets [5], des widgets auto-testables [47] ainsi qu'une description dissociée du COM et du MON (cette dernière étant à base de propriétés) [21]. Il est intéressant de noter que l'architecture proposée est l'aboutissement de travaux de longue haleine assurant la détection, le recouvrement et le confinement des fautes.

CONCLUSION ET PERSPECTIVES

Dans cet article nous avons proposé une taxonomie qui rassemble plusieurs travaux de recherche dans les domaines de la sécurité (comportements malveillants), de la tolérance aux fautes, des méthodes formelles ainsi que des facteurs humains qui sont 4 domaines traités généralement de manière séparée. Dans ce cadre, nous avons identifié les besoins pour résoudre les problèmes liés à la fiabilité des systèmes interactifs, et plus particulièrement, ceux se trouvant dans les systèmes critiques. Nous avons ainsi défini les contours d'une liste de problématiques à résoudre pour augmenter la fiabilité des systèmes interactifs.

La seconde partie de ce papier se concentrait sur une architecture auto-testable pour les systèmes interactifs, capable de détecter les fautes naturelles. Nous avons également proposé deux architectures candidates pour effectuer le recouvrement de telles fautes: la première est automatique tandis que la seconde est pilotée par un utilisateur (chacune ayant ses avantages et limitations détaillés dans l'article).

L'article a ensuite présenté comment de telles architectures peuvent être appliquées à un cas réel d'utilisation : l'application FCU Backup. L'application de telles architectures à ce cas d'utilisation permet alors la détection et le recouvrement des fautes matérielles pouvant affecter l'application. Cependant, l'adjonction de telles architectures soulève encore d'autres problèmes à identifier et à résoudre.

Premièrement, cet article s'est concentré sur les techniques d'interaction WIMP, l'application de telles architectures risque d'être plus problématique lors du passage aux interactions post-WIMP comme la manipulation directe ou les interactions multi-touch pour lesquelles les entrées et les sorties sont bien plus entrelacées.

Deuxièmement, l'ajout des composants nécessaires à la tolérance aux fautes (moniteurs, dispacheur, comparateurs…) aux applications interactives augmente significativement leur taille (approximativement par 5 pour les widgets auto-testables) même si l'ensemble des fonctionnalités reste constant. Il y a donc un besoin grandissant de développer des techniques de description formelle capable de passer à l'échelle et dédiées à la description des systèmes interactifs.

REMERCIEMENTS

Ces travaux ont été financés en partie par Airbus via le contrat R&T Display System X31WD1107313 et les projets FENICS, financés par CORAC.

REFERENCES

1. Accot J., Chatty S., Maury S. and Palanque P. Formal Transducers: Models of Devices and Building Bricks for Highly Interactive Systems 4th Eurographics workshop on "design, specification and verification of Interactive systems", 1997, Springer Verlag.
2. ARINC 653 Avionics Application Software Standard Interface. ARINC Specification 653. Airlines Electronic Eng. Committee July 15, 2003
3. ARINC 661 Cockpit Display System Interfaces to User Systems. ARINC Specification 661. Airlines Electronic Eng. Committee 2002.
4. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. In IEEE Trans. on Dependable and Secure Computing, vol.1, no.1, pp. 11- 33, 2004
5. Barboni, E., Conversy, S., Navarre, D., Palanque, P. Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification. DSVIS 2006. LNCS 4323, pp. 25–38.
6. Basnyat, S, Palanque, P, Schupp, B, Wright, P (2007) Formal socio-technical barrier modelling for safety-critical interactive systems design (2007) Safety Science, Vol 45, n°5, June, ISSN: 0925-7535
7. Basnyat S., Chozos N. and Palanque P. Multidisciplinary perspective on accident investigation. Reliability Engineering & System Safety Volume 91, n° 12, 2006, Pages 1502-1520
8. Bass, L., Little, R., Pellegrino, R., Reed, S., Seacord, R., Sheppard, S., and Szezur, M. R. "The Arch Model: Seeheim Revisited." UI Developpers' Workshop. 1.0 (1991).
9. Bastide, R., Palanque, P., Navarre, D., A Tool-Supported Design Framework for Safety Critical Interactive Systems, Interacting with computers, 2003, vol. 15/3, pp. 309–328.
10. Bastide, R., Sy, O., Palanque, P., Navarre, D. Formal specification of CORBA services: experience and lessons learned. ACM Conf. on Object-Oriented Prog. Sys. Lang. and Applications (OOPSLA'2000). ACM Press, p105-117.
11. Bass L., Clements P., Kazman R., Software Architecture in Practice, Addison Wesley, Reading, Mass., 1998.
12. Bowen J. and Reeves S.. 2011. UI-driven test-first development of interactive systems. In Proceedings of the 3rd ACM symposium on Engineering interactive computing systems (EICS '11). ACM, New York, NY, USA, 165-174.
13. Bowen J. and Reeves S. 2012. Modelling user manuals of modal medical devices and learning from the experience. ACM symp. on Engineering interactive computing systems (EICS '12). ACM, 121-130.
14. Bowen J. and Reeves S. Modelling safety properties of interactive medical systems. ACM symp. on Engineering interactive computing systems (EICS '13). ACM DL, 91-100.
15. Bowen J. and Stavridou V. Formal Methods, Safety-Critical Systems and Standards. Software Engineering Journal, 8(4):189–209, July 1993.
16. Campos, J and Harrison, M. D. Formally verifying interactive systems: A review. 4th EG workshop on Design, Specification and Verification of Interactive Systems DSVIS '97. 109-124. 1997. Springer Verlag.
17. Dearden, A. M and Harrison, M. D. Formalising human error resistance and human error tolerance. Proc. of 5th Int. Conf. on Human-Machine Interaction and Artificial Intelligence in Aerospace. 1995. EURISCO

18. Dessiatnikoff A., Nicomette V., Alata E., Deswarte Y., Leconte B., Combes A. and Simache C. Securing Integrated Modular Avionics computers, 32nd Digital Avionics System Conference (DASC), October 6-10 2013:
19. DO-254 - Design Assurance Guidance for Airborne Electronic Hardware, RTCA Inc, EUROCAE, April 2000.
20. Fayollas C., Fabre J-C., Navarre D., Palanque P. and Deleris Y. Fault-Tolerant Interactive Cockpits for Critical Applications: Overall Approach. 4th Int. Workshop on Software Engineering for Resilient Systems (SERENE 2012), LNCS, Springer Verlag. pp. 134-155.
21. Fayollas C., Fabre J.-C., Palanque P., Cronel M., Navarre D., Deleris Y. A software-implemented fault-tolerance approach for control and display systems in avionics. 20th IEEE Pacific Rim Int. Symp. on Dependable Computing. LNCS.
22. Fayollas C., Martine C., Palanque P., Deleris Y., Fabre J.-C., Navarre D. An approach for assessing the impact of dependability on usability: application to interactive cockpits. European Dependable Computing Conf.. pp 198-209. 2014
23. Hamon A., Palanque P., Silva J-L., Deleris Y. and Barboni E. Formal description of multi-touch interactions. Symposium on Engineering interactive computing systems (EICS '13). ACM, 207-216
24. Harrison M. & Dix A. A state model of direct manipulation. In M. Harrison and H. Thimbleby (eds.) Formal Methods in Human Computer Interaction, p. 129-151, Cambridge University Press, 1990.
25. Hecht H. and Fiorentino E. Reliability assessment of spacecraft electronics. In Annual Reliability and Maintainability Symp., pages 341–346. IEEE, 1987.
26. Hollnagel, E. Barriers and Accident Prevention. 2004. Ashgage.
27. IBM (1989) Common User Access: Advanced Interface Design Guide. IBM, SC26-4582-0
28. Ladry J-F, Navarre D., Palanque P. Formal Description Techniques to Support the Design, Construction and Evaluation of Fusion Engines for SURE (Safe Usable, Reliable and Evolvable) Multimodal Interfaces. Int. Conf. on Multimodal Interfaces (ICMI 2009), ACM, p. 135-142.
29. Laprie, J-C., Arlat, J., Béounes, C., Kanoun, K. Definition and Analysis of hardware and software Fault-Tolerant Architectures, IEEE computer, vol.23, no.7, pp.39-51, 1990.
30. Martinie C., Palanque P., Navarre D., Winckler M., and Poupart E. 2011. Model-based training: an approach supporting operability of critical interactive systems. Proc. of ACM symp. on Engineering interactive computing systems (EICS '11). ACM DL, 53-62
31. Masci, P., Ayoub, A., Curzon, P., Harrison M., Lee, I. and Thimbleby, H. Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example. ACM symp. on Engineering interactive computing systems (EICS '13). ACM DL, 81-90.
32. Navarre, D., Palanque, P., Basnyat, S., (2008) Usability Service Continuation through Reconfiguration of Input and Output Devices in Safety Critical Interactive Systems. 27th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP 2008) LNCS 5219, pp. 373–386.
33. Navarre, D., Palanque, P., Ladry, J., and Barboni, E. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability, ACM TOCHI, 2009, V. 16, 4, pp. 1-56
34. Neema S., Bapty T., Shetty S. & Nordstrom S. 2004. Autonomic fault mitigation in embedded systems. Eng. Appl. Artif. Intell. 17, 7 (October 2004), 711-725.
35. Nicolescu B., Peronnard P., Velazco R., and Savaria Y. Efficiency of Transient Bit-Flips Detection by Software Means: A Complete Study. Proc. of the 18th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT '03). IEEE Computer Society, 377-384.
36. Palanque P. & Basnyat S. Task Patterns for Taking into account in an efficient and systematic way both standard and erroneous user behaviours. 6th International Conference on Human Error, Safety and System Development, Springer Verlag pp. 123-139.
37. Palanque P., Barboni E., Martinie C., Navarre D., and Winckler M. A model-based approach for supporting engineering usability evaluation of interaction techniques. Proc. of ACM symp. on Engineering Interactive Computing systems (EICS '11). ACM DL, 21-30
38. Palanque P., Bastide R., Dourte L. Contextual Help for Free with Formal Dialogue Design. In proceedings of the Fifth International Conference on Human-Computer Interaction, (HCI International '93), 1993, Volume 2
39. Pnueli A Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. LNCS n° 224 p.510-584. Springer Verlag 1986.
40. Polet, P, Vanderhaegen, F, and Wieringa, P. Theory of safety related violation of system barriers. Cognition Technology & Work, 4, 3, 171-179. 2002.
41. Rajkomar A. and Blandford A. 2012. A distributed cognition model for analysing interruption resumption during infusion administration. Proc. of 30th European Conf. on Cognitive Ergonomics (ECCE '12). ACM DL, 108-111.
42. Reason, J. (1990). Human Error, Cambridge University Press (et sa trad : L'erreur humaine, PUF Le travail humain, 1993)
43. Reiter M.K. and. Stubblebine S. G. Authentication metric analysis and design. ACM Trans. Inf. Syst. Secur. 2, 2 (May 1999), 138-158.
44. RTCA-EUROCAE. Software Considerations in Airborne Systems and Equipment Certification. DO-178B / ED-12B, December (1992)
45. SAE-AS5506B: SAE Architecture Analysis and Design Language (AADL), Interanctional Society of Automotive Engineers, Warrendale, PA, USA (September 2012).
46. Schroeder B., E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In ACM SIGMETRICS, pages 193–204, Seattle, WA, June 2009.
47. Tankeu-Choitat, A., Navarre, D., Palanque, P., Deleris, Y., Fabre, J.-C., Fayollas, C. Self-checking components for dependable interactive cockpits using formal description techniques. In Proc of 17th IEEE Pacific Rim Int. Symp. on Dependable Computing (PRDC 2011), 10p.
48. Tankeu-Choitat A., FabreJ-C., Palanque P., Navarre D., Deleris Y. Self-Checking Components for Dependable Interactive Cockpits. Work. on Dependable Computing (EWDC 2011), ACM DL 10p.
49. Thimbleby H., Gimblett A. Dependable keyed data entry for interactive systems. Proceedings of the 4th Int. Workshop on Formal Methods for Interactive Systems (FMIS 2011)
50. Traverse, P., Lacaze, I. and Souyris, J. Airbus fly-by-wire: a total approach to dependability, Proc. IFIP World Computer Congress, pp. 191-212. 2004.
51. Wegner P. 1997. Why interaction is more powerful than algorithms. Commun. ACM 40, 5 (May 1997), 80-91.
52. Wright N., Patrick A. S., and Biddle R.. Do you see your password?: applying recognition to textual passwords. Proc. of Symp. on Usable Privacy and Security (SOUPS '12). ACM DL, 14 pages
53. Yau S.S, R.C Cheung, "Design of self-Checking Software", proc. Int. Conf. on Reliable Software, Los Angeles, CA, USA, IEEE Computer Society Press, 1975, pp. 450-457.
54. Yeh Y. C. (Bob), "Triple-Triple Redundant 777 Primary Flight Computer", IEEE Aerospace Applications Conf., pp 293-307, 1996