MÉTHODOLOGIE DE MODÉLISATION ALTARICA POUR LA SÛRETÉ DE FONCTIONNEMENT D'UN SYSTÈME DE PROPULSION HÉLICOPTÈRE INCLUANT UNE PARTIE LOGICIELLE

METHODOLOGY OF ALTARICA MODELLING FOR RELIABILITY ANALYSIS OF HELICOPTER PROPULSION SYSTEM INCLUDING SOFTWARE FUNCTIONS

Sophie HUMBERT
Doctorante CIFRE - Université Bordeaux 1
Entreprise : TURBOMECA
Laboratoire de recherche : ONERA - CERT

Charles CASTEL et Christel SEGUIN
ONERA - CERT
2, avenue Édouard Belin - BP 4025
31055 TOULOUSE CEDEX

Jean-Marc BOSC et Pierre DARFEUIL
TURBOMECA
Boîte N°6
64511 BORDES CEDEX

Yves DUTUIT
Université Bordeaux 1 / LAPS
351, cours de la libération
33405 TALENCE

Résumé

L'objectif poursuivi est d'améliorer la déclinaison des exigences de sécurité du niveau système vers le niveau logiciel, en utilisant des modèles formels : AltaRica / Cecilia™ OCAS et Lustre / SCADE SUITE™ pour représenter le système à des niveaux d'abstraction différents.
Cet article présente la première étape de notre démarche. Il s'agit d'une méthodologie de modélisation fonctionnelle et dysfonctionnelle de systèmes incluant une partie logicielle, en langage formel AltaRica. Il expose également différentes façons d'exploiter le modèle, et en particulier, l'intérêt de la génération des scénarios qui conduisent aux événements redoutés, afin d'en déduire la criticité de chaque fonction logicielle.

Summary

The aim of our studies is to improve the cascading of our safety requirements from the system level down to the software level by using formal models: AltaRica / Cecilia™ OCAS and Lustre / SCADE SUITE™ to represent the system with different levels of abstraction.
This article describes the first stage of our approach. It consists of a methodology of system modelling (functional and dysfunctional) which includes software functions in AltaRica formal language. At the same time it uses different ways of operating this model, and in particular the benefits arising from the generation of scenarios which lead to undesired events, with the object of deducing the criticality of each software function.

Introduction

Les systèmes de propulsion hélicoptère comportent des moteurs dont les calculateurs de régulation et de surveillance hébergent des logiciels de contrôle de plus en plus élaborés et complexes. Pour répondre aux exigences de sécurité de ce type de système, qui sont issues des réglementations moteur (CS-EASA, FAR-FAA) et de l'hélicoptériste, il convient de bien décliner les exigences sur le système en exigences au niveau logiciel. Comme ces deux familles d'exigences portent sur des objets de granularité différente, le passage d'un niveau à l'autre peut être sujet à interprétation ou amener à formuler des exigences difficilement testables au niveau le plus concret. Pour surmonter ces difficultés, nous proposons de définir les exigences en relation avec les modèles de système et de logiciel sur lesquels nous souhaitons les évaluer. Ainsi, l'objectif poursuivi est d'améliorer le passage des exigences de sécurité du niveau système vers le niveau logiciel, en utilisant les modèles formels du système et du logiciel.

L'utilisation de modèles formels, par exemple de type SCADE SUITE™ (Esterel Technologies), pour spécifier de manière détaillée les logiciels de contrôle est une pratique établie. L'usage de modèles formels de dysfonctionnement de système (de type AltaRica par exemple) tend à se répandre. Par contre, la pratique en vigueur conduit rarement à modéliser explicitement les dispositifs logiciels inclus dans le système. Or il est intéressant de modéliser les erreurs de conception potentielles de ces dispositifs et d'identifier les plus critiques afin d'exiger leur élimination lors des phases de conception logicielle. De manière duale, les caractéristiques des mécanismes de sécurité logiciels doivent être identifiées dès les phases amont, afin de donner lieu à d'autres exigences fonctionnelles au niveau logiciel.

Dans ce contexte, l'objectif de cet article est de montrer comment mettre en œuvre les premières étapes de la démarche : modélisation de systèmes incluant des dispositifs logiciels, modélisation des exigences de sécurité allouées à ces systèmes et analyse de ces exigences. Nous montrerons également l'impact de ces analyses préliminaires sur l'évaluation des conceptions logicielles ultérieures : identification des parties logicielles critiques et des scénarios de test.

Contexte industriel

Pour développer et certifier ses moteurs, Turbomeca effectue des analyses de sécurité en modélisant ses turbomachines par arbres de défaillance. Ce formalisme est centré sur un seul événement redouté à la fois, puisqu'un arbre doit être construit pour chaque événement redouté. Par ailleurs la prise en compte des modifications du système est difficile puisqu'une modification peut impacter plusieurs événements redoutés. Ainsi, dans le cadre des activités de recherche de Turbomeca, des modèles sont élaborés en langage AltaRica. Ces modèles ont l'avantage de présenter une vue globale du système qui permet d'étudier plusieurs événements redoutés avec un seul modèle et facilite le travail de collaboration au sein de l'équipe de conception.
Jusqu'à présent, ces modèles représentaient l'architecture fonctionnelle, et les dysfonctionnements du système mais uniquement pour les composants matériels. En effet, les contributions du logiciel n'étaient pas considérées. Les modèles que nous proposons sont construits dans le but d'étudier des systèmes en phase de conception préliminaire. Nous considérons que les choix d'architectures matérielles ont été déjà faits. Le but poursuivi est d'inclure dans le modèle les fonctions logicielles envisagées afin de vérifier que la nouvelle architecture matérielle et logicielle envisagée satisfait les exigences de sécurité. Ainsi, à partir des analyses du modèle, nous chercherons à améliorer la

spécification des exigences de sécurité portant sur les fonctions logicielles.
Nous proposons ici une méthodologie de modélisation en AltaRica qui inclut dans les modèles les éventuelles erreurs de conception des fonctions logicielles.
Puis, nous montrerons comment exploiter ce modèle afin d'améliorer la déclinaison des exigences système vers le logiciel.

Présentation du cas d'étude

Pour la clarté de l'exposé, le cas d' étude est inspiré de la réalité mais n'en reprend pas toute la complexité.

Dans une turbomachine, le calculateur de régulation élabore une consigne de débit carburant puis pilote le déplacement d'un dispositif de dosage. Le but de ce cas d'étude est de modéliser les fonctions logicielles, et matérielles, en se limitant à celles qui permettent de piloter le doseur carburant. Les principaux composants de haut niveau modélisés sont le calculateur (qui a entre autres pour fonction de piloter le doseur) et le dispositif de dosage (qui a pour fonction de régler une section de passage du carburant).

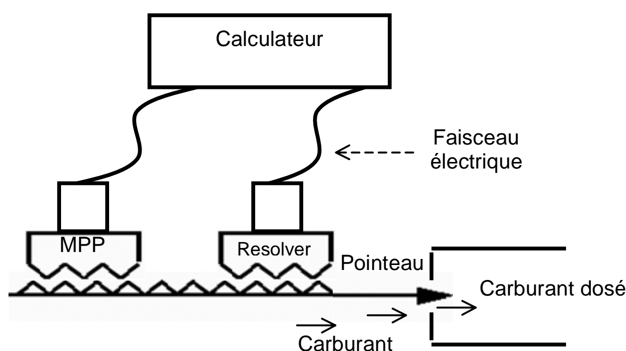

Figure 1 : le système modélisé

Le dispositif de dosage du carburant

Le calculateur commande les phases (enroulements) du moteur pas à pas (MPP) via un faisceau électrique dont le rôle est de conduire le courant d'alimentation des phases. La rotation du moteur pas à pas fait déplacer une crémaillère qui est solidaire du pointeau et qui entraîne également le resolver (voir plus loin). Le changement de position du pointeau entraîne une modification de la section de passage du carburant, ce qui permet d'augmenter ou réduire le débit carburant dirigé vers la chambre de combustion. La position du pointeau, recopiée par le resolver, est quant à elle transmise au calculateur via un faisceau électrique.

Le calculateur

Le calculateur est constitué de trois parties. Outre la partie matérielle, il y a une partie OSS (Operating System Software) et une partie CSS (Control System Software). La partie OSS correspond au logiciel de bas niveau dit « automate ». Il fournit notamment un compte-rendu de panne et une indication de non utilisation des valeurs reçues ou émises par le calculateur, lorsque celles-ci sont momentanément considérées comme douteuses (par exemple lors de la mise sous tension du calculateur). La partie CSS correspond au logiciel dit « applicatif ». Elle contient les algorithmes de calcul (calcul de débit carburant, calcul du déplacement du doseur) et des fonctions de surveillance.

L'événement redouté et le besoin de détection des pannes

Il existe plusieurs modes de défaillance conduisant à la perte du système de dosage. Le principal événement redouté relatif au système doseur étudié correspond à une valeur de débit du carburant dosé erronée et non signalée au pilote par le calculateur de régulation. Afin de se protéger contre un taux d'occurrence trop élevé de cet événement, des fonctions de détection logicielles ont été envisagées.

Les fonctions de détection et de tolérance aux pannes

Deux fonctions de détection de pannes sont envisagées pour cette commande.
La première justifie l'emploi du resolver qui recopie la position du pointeau pour détecter les pertes de toute nature (électrique ou mécanique) de la commande du moteur pas à pas. Pour ce faire, un test d'écart est envisagé entre la demande de position du moteur pas à pas et la position (notée XR) lue du resolver (c'est-à-dire l'écart entre la consigne de déplacement et le déplacement observé). Lorsque le test de cohérence détectera une panne, le calculateur considèrera la commande doseur comme perdue. Il annoncera au pilote une panne de la régulation automatique et placera le système dans un état sûr : le calculateur « gèlera » le débit carburant à la valeur qu'il avait juste avant la défaillance. Ainsi, le calculateur ne demandera plus de déplacement du doseur et l'immobilisera en alimentant en continu au moins une des phases du moteur pas à pas.

La deuxième fonction de détection envisagée concerne la détection des pannes propres au resolver. Cette détection se justifie pour être tolérant aux pannes de la surveillance, c'est-à-dire pour pouvoir continuer à piloter le débit sans surveillance de la commande du moteur pas à pas et éviter la perte de la régulation automatique. Ainsi, si une défaillance matérielle du système doseur se produit, alors elle sera détectée-localisée sur le resolver, et la surveillance sera considérée comme perdue. Dans ce cas là, le calculateur continuera à piloter le moteur pas à pas mais sans le surveiller, puisque le test de cohérence ne sera plus effectué.

Les modes de défaillance du matériel

Les défaillances matérielles prises en compte au niveau de la chaîne de commande ont pour effet la perte de la commande de position, causée par exemple par la perte du faisceau électrique du moteur pas à pas.
Au niveau de la chaîne de surveillance, les défaillances prises en compte ont pour effet la transmission d'une valeur de position erronée (défaillance du resolver ou du faisceau électrique).

Les erreurs de conception du logiciel

Par ailleurs, nous avons pris en compte les éventuelles erreurs de conception des fonctions logicielles. Le logiciel envisagé comporte trois fonctions principales : le test du resolver, le test de cohérence, et la fonction qui calcule le déplacement du doseur. Pour les fonctions de détection, nous avons modélisé deux sortes d'erreur de conception potentielle : la première est une erreur d'algorithme dont la conséquence est la détection intempestive d'erreur (une valeur correcte sera considérée comme erronée) ; la seconde est aussi une erreur d'algorithme, mais qui a pour conséquence la non-détection de valeur erronée. Pour la fonction déplacement du doseur, nous avons considéré qu'une erreur d'algorithme pouvait entraîner l'élaboration d'une consigne erronée. Nous avons également pris en compte les éventuelles erreurs de transmission des informations au niveau des acquisitions et des sorties de l'OSS.

Le langage AltaRica

AltaRica est un langage conçu au LaBRI (Laboratoire Bordelais de Recherche en Informatique), qui permet de modéliser formellement le comportement des systèmes sujets à des défaillances. C'est un langage formel événementiel (discret et non continu). Il est basé sur la notion des automates à contraintes [1].
Nous allons commencer par présenter le langage AltaRica, puis nous verrons dans la partie suivante comment nous l'utilisons.

La description d'un modèle dans ce langage met en œuvre, entre autres, les concepts suivants : description d'un nœud, description des interactions entre les nœuds et hiérarchie.

La description d'un nœud (cf. l'exemple présenté plus loin) consiste à renseigner les champs suivants :

- **flow** : déclare un ensemble de variables et leur type : booléen, énuméré, entier... ; ces variables d'entrée / sortie

représentent les informations échangées, par le nœud, avec l'extérieur ;

- **state** : déclare un ensemble de variables d'états interne du nœud et leur type : booléen, énuméré, entier… ; leur valeur représente l'état courant du nœud ;
- **event** : déclare l'ensemble des événements qui provoquent les changements d'états ; à chaque événement peut être attribué une loi de probabilité et un taux d'occurrence comme par exemple exponentielle (λ) ou Dirac(0) ;
- **trans** : décrit les changements d'états en utilisant les valeurs des variables précédentes ; une transition est un triplet de la forme (g |- ev -> aff) où « g » (nommé garde) est une condition de validation de la transition, « ev » est le nom de l'événement provoquant le changement d'états et « aff » exprime le changement d'états ;
- **assert** : décrit les valeurs prises par les variables de sortie en fonction de l'état courant et de la valeur des variables d'entrée.

Les interactions entre les nœuds sont décrites par leurs connexions et / ou par des mécanismes particuliers tels que la synchronisation des événements.

La hiérarchie permet une représentation à des niveaux d'abstraction différents (par exemple un sous-système peut être raffiné en plusieurs fonctions). Elle permet aussi de regrouper un ensemble de nœuds dans un autre.

Utilisation du langage

La modélisation en AltaRica consiste à identifier les nœuds, les décrire et les interconnecter.
Chaque **nœud** modélise un composant (ou fonction), ou bien un ensemble de composants (hiérarchie).
Les **variables de flux** représentent les données échangées entre les composants (ou ensemble de composants).
Le **type des variables** représente le niveau d'abstraction des valeurs véhiculées. On entend par niveau d'abstraction le degré de détail modélisé. Par exemple, une variable numérique entière peut être abstraite en plusieurs ensembles de valeurs (ou classe) : au lieu d'avoir le domaine de valeur {1,2,3,4,5}, nous pouvons l'abstraire en deux classes « inférieur à 3 » et « strictement supérieur à 3 » ou encore, si l'on s'intéresse à la qualité de la valeur, « correcte » ou « erronée ».
Les **états** représentent les modes de fonctionnement et les modes de dysfonctionnement des composants (ou fonctions).
Les **événements** modélisent les défaillances ou les reconfigurations qui provoquent les changements d'états décrits dans les transitions.
Les **transitions** représentent les aspects temporels des modes de fonctionnement et de dysfonctionnement ainsi que la cause des événements (intrinsèques ou induits i.e. extrinsèques).
Enfin les **assertions** décrivent la fonction du nœud (expression des relations entre les entrées, les états et les sorties).

Illustration

Nous prenons pour exemple une fonction logicielle appelée « test limites » afin d'illustrer la modélisation d'un nœud.
Cette fonction a pour but de vérifier que la **valeur à tester** qu'elle reçoit en **entrée** est comprise entre deux bornes minimum et maximum. En **sortie**, elle fournit un « **statut de détection** » qui informe sur la validité de la valeur testée.
On considère que cette fonction a **trois états**. Un état de fonctionnement « **nominal** » dans lequel elle détecte correctement et deux états de dysfonctionnement. Le premier correspond à la perte du test (« **test_perdu** ») : une valeur erronée n'est pas détectée alors qu'en mode de fonctionnement nominal elle le serait. Nous considérons que cet état est atteint en cas d'erreur de conception de la fonction logicielle (dont l'origine peut être une erreur de programmation ou bien, une spécification de la fonction logicielle incomplète ou inadéquate). Comme nous considérons qu'il n'y a pas de correction de l'algorithme en vol, c'est un **état puit**. Le second état de dysfonctionnement est une « **fausse alarme** », c'est-à-dire que la fonction détecte une valeur comme étant erronée alors qu'elle est réellement correcte. C'est aussi un **état puit** car une détection de panne est irréversible jusqu'à la fin du vol (même si la valeur à tester redevient correcte par la suite).
Ces deux états résultent de l'apparition d'**événements**, notés respectivement «**perte_détection**» et «**détection_intempestive**» représentatifs d'une erreur de conception de la fonction. (Ils pourraient aussi être représentatifs de défaillances du matériel implémentant cette fonction, mais elles n'ont pas été modélisées dans cet exemple).
Le domaine de valeur de la variable numérique « **entrée à tester** » est abstrait en un **type énuméré** « **correcte** » ou « **erronée** » (on entend par erronée une valeur supérieure à la borne maximum ou inférieure à la borne minimum). Le domaine de valeur de la variable de sortie « **statut_détection** » est **booléen**.

Voici le code AltaRica de la fonction logicielle « test_limites ».
Les commentaires sont précédés des caractères // .

```
node  test_limites

flow      // déclaration des variables de flux
 entrée_à_tester : {correcte, erronée} : in ;   // entrée
 statut_détection : bool : out ;        //sortie
// true correspond à détection et false à non détection

state     // déclaration des états
 état : {nominal, fausse_alarme, test_perdu} ;

event     // déclaration des événements
 détection_intempestive, perte_détection;

trans     // changements d'états
 état = nominal |- détection_intempestive -> état := fausse_alarme;
 état = nominal |- perte_détection -> état:= test_perdu;

// les états « fausse alarme » et « test_perdu » sont permanents : ils ne sont l'état de départ d'aucune transition.
// leur événement associé est intrinsèque : la garde ne dépend que de l'état interne du composant.

assert    // détermination de la valeur de la variable de sortie
 statut_détection = case
        {
         état = nominal and entrée_à_tester = erronée : true,
         état = fausse_alarme : true,  // détection
         état = test_perdu : false,      // non_détection
         else false    // état = nominal and entrée_à_tester = correcte
        }

// Remarque concernant l'usage du « case » utilisé dans les assertions :
sortie=case {condition1:valeur1, condition2:valeur2, else valeur3}
signifie si condition1 est vrai alors sortie=valeur1 sinon si condition2 est vrai alors sortie=valeur2 sinon sortie=valeur3.
```

La démarche méthodologique de modélisation

L'objectif de cette démarche est de disposer d'un modèle formel du système à analyser qui représente les propriétés intéressantes du point de vue de la sécurité. Dans ce but et de manière à sélectionner uniquement les éléments importants pour cette analyse nous suivons les étapes suivantes : identification des éléments d'analyse de sûreté de fonctionnement, construction du modèle en AltaRica, et enfin, validation et vérification du modèle.
Notre démarche s'inspire des travaux référencés [2], [3] et affine les principes décrits ces les références .

Identification des éléments à modéliser

La première étape consiste à recueillir les informations nécessaires à la construction d'un modèle orienté sûreté de fonctionnement. Pour cela on va s'intéresser aux aspects fonctionnels et dysfonctionnels du système.

Tout d'abord on **recense toutes les fonctions,** y compris les fonctions de sûreté : détection, isolation, reconfiguration. Étant intéressés par la propagation des défaillances, il est nécessaire d'identifier les influences entre les différentes fonctions. Pour cela

une aide est apportée par l'**étude de l'architecture** du système qui permet d'identifier une partie importante de ces influences (données échangées entre les fonctions).
Pour chaque fonction, nous explicitons **leurs différents modes de fonctionnement**. Plus précisément, un mode de fonctionnement peut être nominal ou dégradé, mais il peut aussi représenter différentes configurations de fonctionnement. Par exemple, une fonction peut avoir un comportement différent selon les phases de vol : décollage, croisière, atterrissage.

Une fois les fonctions du système recensées, il faut s'intéresser aux aspects dysfonctionnels, i.e. à la manière dont elles peuvent remplir leur rôle de manière insatisfaisante. Pour cela nous allons nous appuyer sur le **recueil des exigences de sécurité système**. Ces exigences se trouvent généralement dans les spécifications du système, héritées des phases amont d'allocation des exigences.
A partir de ces exigences, il faut étudier les modes de dysfonctionnement qui sont ceux des composants (matériels ou logiciels), nécessaire à la réalisation des fonctions du système.
Pour un composant physique (matériel), ce sont les modes de défaillance (par exemple une dérive ou une perte franche pour un capteur). Pour les fonctions logicielles, ils représentent à la fois d'éventuelles erreurs de conception et des défaillances du matériel implémentant cette fonction. Une fois ces modes de dysfonctionnement établis, il s'agit de préciser leurs **caractéristiques** : **la cause du dysfonctionnement** (intrinsèque, induit par propagation, cause commune), **leur caractère temporel** (permanents, intermittents, dormants…) ...

Construction du modèle

La deuxième étape a pour objectif de fournir un modèle AltaRica du système en exploitant toutes les informations recueillies lors de l'étape précédente.
A partir du recensement de toutes les fonctions et de l'étude de l'architecture, nous déduisons les **nœuds** AltaRica (node) et **leurs variables d'entrée et de sortie** (flow). A partir des modes de fonctionnement et de dysfonctionnement, nous déduisons **les états** (state).
Les changements d'états interviennent lorsqu'il se produit un événement. **Les événements** (event) dans le cas des composants matériels, modélisent les défaillances. Dans le cas de fonction logicielle, il n'y a pas d'événement à proprement parler : si une fonction contient des erreurs de conception, elle les contiendra tout au long du vol. Il s'agit plutôt d'un état initial. Cependant, selon des événements extérieurs mal cernés à ce niveau de l'analyse, ce défaut est visible ou pas. Par conséquent, nous utiliserons des états internes indiquant qu'un type de défaut est activé et nous observerons les changements d'états via des événements spécifiques.
Le type de chaque variable d'entrée, de sortie et d'état représente le niveau d'abstraction souhaité. Il s'agit de faire un compromis entre précision et complexité. Cette activité est contrainte par une notion de cohérence et de compatibilité avec le niveau d'analyse souhaité. D'une part il est nécessaire d'avoir une perception de niveau similaire pour les différents objets du modèle. En particulier il faut être cohérent dans la définition des différents domaines de valeur des données échangées et des états. Lorsque l'analyste choisit les mots clés définissant les états de chaque composant ou fonction, il définit implicitement un domaine d'abstraction. Celui-ci dépend entre autres de la technologie à modéliser, de la connaissance du système, des exigences à vérifier et des analyses précédentes. D'autre part, il est nécessaire que les valeurs manipulées soient en rapport avec l'expression des résultats attendus des analyses qui seront menées sur le modèle.

Cette partie déclaration est suivie de la description de la dynamique du nœud : les transitions (trans) et les assertions (assert).
Les transitions (ainsi que les mécanismes de synchronisation entre les événements) représentent les aspects temporels du système à modéliser. Par exemple une panne permanente sera représentée par un état puits (aucun événement ne peut en faire sortir). Par ailleurs, la cause des événements est aussi traduite dans les transitions : une défaillance induite par un autre nœud peut être codée soit par une synchronisation d'événement soit par une propagation de valeur (flow) associée à la garde (condition d'activation) d'une transition.
Par exemple, si un événement est interne au composant, alors la garde ne sera fonction que de l'état interne du composant. Par contre, si la cause de l'événement est externe, alors la garde sera fonction d'une variable d'entrée (qui propage la panne) et / ou de l'état interne du composant. Voici l'exemple d'une défaillance externe dont les effets sont temporaires :
état = actif and entrée_alim = false |- update -> état = inactif;
état = inactif and entrée_alim = true |- update -> état =actif;
L'événement noté « update » est un événement fictif qui permet de modéliser la propagation de panne (le composant n'est pas lui-même défaillant). Nous avons vu dans la présentation du langage AltaRica, qu'à chaque événement peut être attribué une loi de probabilité et un taux d'occurrence comme par exemple exponentielle (10^{-6}) ou Dirac(0). Dans l'exemple cité ci-dessus, à l'événement update est affecté la loi Dirac(0) qui signifie que le changement d'états est effectué dès que la garde est vérifiée.
Finalement, **les assertions** décrivent les valeurs des variables de sortie en fonction de l'état courant du composant, et de ses variables d'entrée.

Enfin, lorsque l'état du système dépend d'événements extérieurs comme par exemple les phases de vol, il faut les modéliser. Pour cela, il est possible de créer un nœud spécifique qui décrit l'environnement extérieur du système. Une solution est de créer un nœud qui représenterait les différentes configurations de vol. Sa variable d'état prendrait par exemple les valeurs représentant les différentes configurations {décollage, croisière, atterrissage}. Les événements traduisant les changements de configuration pourraient être « début croisière » pour traduire le changement d'états de décollage vers croisière et « fin croisière » pour traduire le passage de croisière vers atterrissage.
Une variable de sortie prendrait pour valeur la valeur courante de la variable d'état.

Lorsque tous les composants sont modélisés (par les nœuds), reste à construire l'architecture du système en connectant leurs entrées et leurs sorties.

Validation - Vérification du modèle

Une fois le modèle construit, il faut valider qu'il représente bien la réalité et vérifier qu'il est correctement construit.

La validation consiste à s'assurer que les modèles individuels (nœuds) et que le modèle global représentent bien les propriétés réelles du dispositif étudié intéressant les aspects sécurité. Il faut aussi s'assurer que cette représentation se fait à un niveau d'abstraction adéquat. Cette validation est réalisée d'une part en confrontant le modèle avec des spécialistes du domaine (par exemple par simulation de propagation de pannes) et, d'autre part en vérifiant que certains aspects (données échangées, comportement, composition) de la réalité sont bien représentés.

La vérification a pour objectif de s'assurer que la syntaxe du modèle est correcte et qu'il présente certaines propriétés : déterminisme, cohérence... Dans le cas d'un modèle formel tel qu'AltaRica cette vérification peut être assistée par des outils [4], [5], [6].
Le module OCAS (Outil de Conception et d'Analyse Système) de l'atelier Cecilia™ de Dassault Aviation propose un contrôle de cohérence automatique [7]. Il permet d'une part de vérifier la complétude et la cohérence du modèle, et d'autre part de détecter si le modèle est statique ou dynamique et s'il est bouclé ou non. En outre, il est possible de faire des simulations interactives, et des traductions vers des model-checkers sont disponibles.

Application au cas d'étude

Nous allons appliquer la méthodologie de modélisation au cas d'étude présenté précédemment.

1 : Recenser toutes les fonctions

Nous avons regroupé les fonctions en quatre catégories : les fonctions de la chaîne matérielle de la commande du doseur, les

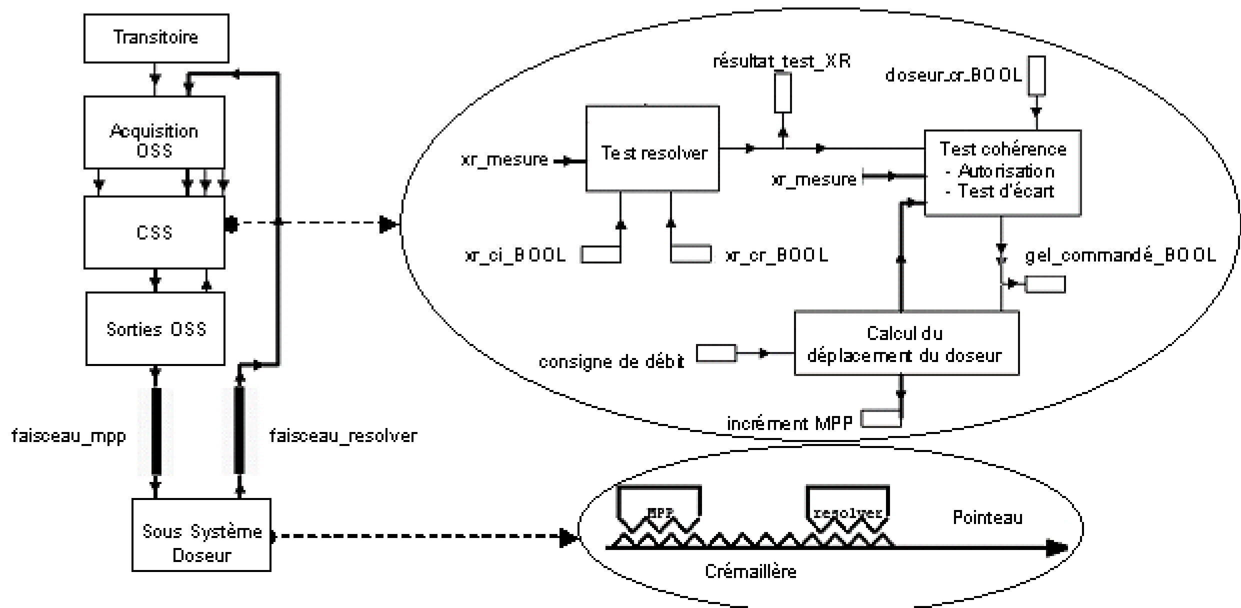

Figure 2 : modèle OCAS du cas d'étude

fonctions de la chaîne matérielle de surveillance du doseur, les fonctions logicielles et les fonctions OSS.
Les principales fonctions liées à la commande du doseur sont **le faisceau électrique calculateur - moteur pas à pas** (noté faisceau_mpp), **le moteur pas à pas**, **la crémaillère** et **le pointeau**. Celles de la chaîne de surveillance sont **le resolver** et **le faisceau électrique resolver - calculateur**. Les fonctions logicielles envisagées sont **le calcul de la consigne de déplacement du doseur**, **le test du resolver** et **le test de cohérence** (constitué d'une partie autorisation d'effectuer le test, et d'une partie test d'écart en la consigne de déplacement du doseur et le déplacement réel observé par le resolver). **Les fonctions OSS** sont les **acquisitions** et **sorties** du calculateur.
Par ailleurs, nous avons vu que les événements extérieurs pouvaient aussi être modélisés. Lors de phénomènes transitoires (par exemple lors de la mise sous tension du calculateur), les valeurs numériques (par exemple la mesure du resolver) peuvent être momentanément inutilisables. Pour ne pas détecter à tort une erreur temporaire, on inhibe le test du resolver le temps de la période transitoire (cette information transite via le compte rendu de non utilisation fourni par l'OSS). Nous avons donc modélisé un nœud qui traduit des **phénomènes transitoires** afin d'observer leurs effets sur le système.

2 : Identifier les données échangées entre les composants

Les données échangées sont représentées par les liens entre les différentes fonctions, figure 2.

3 : Étudier les états de fonctionnement des composants

Le dispositif de dosage peut être gelé à la demande du calculateur (détection d'erreur par le test de cohérence) . Il a donc deux modes de fonctionnement : un mode de régulation nominal, et un mode de fonctionnement gel_commandé (en situation de recueil). Ainsi, les fonctions liées à la commande doseur, hormis le faisceau électrique, (c'est à dire **le moteur pas à pas**, **la crémaillère** et **le pointeau**) héritent de ces deux modes de fonctionnement. Ils sont concrétisés au sein de chaque composants précédemment cités par les états **« régulation nominal »** et **« gel commandé»**. Dans ce dernier état, ils ne reçoivent plus de commande de déplacement. Il sont donc immobilisés.
Toutes les autres fonctions ont un seul état de fonctionnement **nominal**.

4 : Recueillir les exigences de sécurité

Une exigence de haut niveau est que la régulation du carburant ne soit pas erronée non détectée par le calculateur. En déclinant cette exigence système de haut niveau sur le système du cas d'étude, le principal événement redouté est que la quantité de carburant dosé soit erronée et non signalée au pilote (i.e. non détecté par le calculateur). Nous rappelons que notre objectif est de modéliser les dysfonctionnements du système et du logiciel, afin de déterminer les impacts des erreurs de conception du logiciel sur les événements redoutés système. Comme nous n'avons pas envisagé d'étudier la quantification des erreurs de conception logicielles, nous allons analyser l'occurrence de cet événement redouté qualitativement.

5 : Identifier les états de dysfonctionnement des composants

Le moteur pas à pas a un état de dysfonctionnement : **« perte de l'activation »**. Le moteur pas à pas ne se déplace plus malgré la consigne qu'il reçoit. L'effet local est une absence de commande de déplacement de la crémaillère.
Le pointeau peut être résistant au déplacement. Selon le degré de résistance, cela peut entraîner une perte de pas du moteur pas à pas ou non. Nous avons donc deux états de dysfonctionnement intrinsèques : **« résistance faible »** et **« résistance élevée »**.
Ces deux états de dysfonctionnement induisent deux états de dysfonctionnement de **la crémaillère « résistance faible »** et **« résistance élevée »**. En effet, si le pointeau est résistant, la crémaillère devient aussi résistante au déplacement.
Pour résumer, les états de dysfonctionnement intrinsèques ou *induits* sont : pour le moteur pas à pas « perte de l'activation », pour la crémaillère : « *résistance faible* » et « *résistance élevée* », et pour le pointeau : « résistance faible », « résistance élevée ».

Le faisceau électrique, identique pour la chaîne de commande et de surveillance, a un seul état de dysfonctionnement intrinsèque **« erreur_transmission »**. En effet, il peut être défaillant en transmettant une information erronée. Il en est de même pour **le resolver**, et **les fonctions OSS**.

Enfin, nous avons pris en compte les éventuelles erreurs de conception des fonctions logicielles décrites dans la présentation du cas d'étude. Les états correspondant pour **les fonctions de détection** sont **« perte du test »** et **« fausse alarme »** (cf. l'exemple de la fonction « test_limite »). Pour la fonction **calcul du déplacement du doseur**, nous avons pris en compte un seul état de dysfonctionnement : **« consigne_erronée »**.

Le nœud qui représente les **phénomènes transitoires** a aussi un état intrinsèque « **transitoire** » qui peut être considéré, par abus de langage, comme un état de dysfonctionnement.

6 : Définir la cause des changements d'états

Nous venons de voir comment les états dysfonctionnels sont atteints. Nous avons vu que les fonctions liées à la commande doseur, hormis le faisceau électrique, ont deux états de fonctionnement : un état régulation nominal, et un état gel commandé nominal. Le passage de l'état régulation à l'état gel commandé se produit lorsque le test de cohérence détecte une erreur (à raison, ou à tort en cas de fausse alarme). La propagation de la commande de gel est modélisée par une variable de flux. Par exemple, pour le pointeau, nous avons la transition :
état=nominal and commande=gel |-update-> état=gel_commandé
L'événement update a pour loi : Dirac(0) (le changement d'états aura lieu dès que la commande prendra la valeur gel).

7 : Étudier le caractère temporel des états

Les défaillances matérielles ne sont pas réparables en vol et sont considérée comme permanentes. **Les erreurs de conception des fonctions logicielles ou OSS** ne sont pas corrigibles en vol. Tous les **états** de dysfonctionnement correspondants sont donc **permanents** (état puits). Il en est de même pour les états de fonctionnement gel commandé.
L'**état « transitoire »** du nœud représentant l'événement extérieur au système n'est pas permanent. En effet, les phénomènes transitoires sont **temporaires**. Voici à titre d'exemple ses transitions :
état = nominal |- phénomènes_transitoires -> état = transitoire ;
état = transitoire |- fin_transitoires -> état = nominal.

8 : Déterminer le degré d'abstraction des données échangées

Le degré d'abstraction a été implicitement défini lors du choix des états. Nous allons commencer par décrire les flux qui circulent dans la **chaîne de commande du doseur**. Ils doivent propager la consigne de dosage. Cette **consigne** peut être **correcte ou erronée**. L'origine d'une consigne erronée peut être soit une défaillance de la chaîne de commande du doseur (faisceau électrique, moteur pas à pas, pointeau), soit une erreur lors de l'élaboration de la consigne (erreur de conception de la fonction de calcul du déplacement du doseur). Nous devons aussi véhiculer **l'origine de la panne** jusqu'au test de cohérence car, une consigne erronée due à une défaillance de la chaîne de commande sera détectée par le test de cohérence (sous réserve que la chaîne de surveillance fonctionne correctement). Par contre, si la chaîne de commande fonctionne correctement, une erreur d'élaboration de la consigne ne sera pas détectée par le test de cohérence. En effet, le pointeau sera déplacé conformément à la consigne d'incrément (qui est certes erronée), mais le test de cohérence ne détectera pas d'écart entre la consigne de déplacement et le déplacement observé par le resolver.
Ces flux doivent aussi véhiculer **l'ordre de gel commandé** car dans ce cas, le sous système de dosage devra être immobilisé (donc chacun des composants du sous système de dosage devra passer dans l'état gel_commandé).
Pour résumé, voici le **domaine d'abstraction des flux qui circulent depuis l'élaboration de la consigne (CSS) jusqu'au pointeau** :
- validité de la commande : « correcte » ou « erronée » ;
- erreur à l'élaboration de la consigne : « vrai » ou « faux » ;
- nature de la commande : « régulation » ou « gel commandé » .

Concernant les **variables de la chaîne de surveillance du doseur**, elles doivent propager l'information de la **position du resolver**. Soit elle est **correcte**, soit elle est **erronée**. Mais, l'origine d'une position erronée est soit une défaillance de la chaîne de surveillance, soit une défaillance de la chaîne de commande. Dans ce cas là, le resolver recopie une valeur erronée dont il n'est pas à l'origine. Ainsi ce type d'erreur ne sera pas détecté par le test du resolver. Par contre les défaillances dues à la chaîne de surveillance seront détectées par le test du resolver. Ainsi, nous faisons circuler dans les flux **l'origine de la panne**.

Pour résumé, voici le domaine d'abstraction des flux qui circulent depuis le resolver jusqu'au test du resolver (xr_mesure) :
- validité de la position_resolver : « correcte » ou « erronée » ;
- origine de la panne : « commande » ou « surveillance ».
Dans les assertions, le couple de valeurs « erronée ; surveillance » conduira à une détection de panne du test du resolver (contrairement au couple « erronée ; commande »).

Les flux du CSS sont représentés sur la figure 2. Les variables avec le suffixe BOOL et le résultat du test du resolver sont booléens ; la consigne de débit (issue du calcul amont du CSS non modélisé ici) est supposée correcte et l'incrément du moteur pas à pas est du type structuré :
- validité de la commande : « correcte » ou « erronée » ;
- erreur à l'élaboration de la consigne : « vrai » ou « faux » ;
- nature de la commande : « régulation » ou « gel commandé » .

Par ailleurs, il est nécessaire de propager les défaillances. Cela est fait par l'intermédiaire d'une variable de flux qui nous permettra d'effectuer les changements d'états induits. Ici, nous avons donc créé une **variable** depuis le pointeau vers la crémaillère **qui reflète la valeur courante de l'état du pointeau**. Elle peut prendre les valeurs : « ok » ou « résistance faible » ou « résistance élevée ». Cette variable sera utilisée dans les gardes des transitions de la crémaillère afin d'effectuer les changements d'états vers les états de dysfonctionnement « résistance élevée » ou « résistance faible » selon les cas.

A présent, chaque nœud peut être construit et connecté aux autres. L'architecture du système modélisé est présentée dans la figure 2. Les parties OSS et CSS, le sous système doseur, et le nœud qui représente les éventuels phénomènes transitoires y apparaissent.

Analyse du modèle

Il existe plusieurs méthodes et outils d'analyse de modèle AltaRica. Nous citerons ici Cecilia™ OCAS (Dassault Aviation) [7] qui propose une vue graphique du modèle, et permet entre autres d'effectuer des simulations interactives de propagation de pannes et des analyses de sécurité comme par exemple : calcul de coupes minimales, calcul de séquences et génération automatique d'arbres de défaillance.

Les événements redoutés

Pour étudier les événements redoutés, il faut les rendre observables en les incluant dans le modèle. En langage AltaRica, les événements redoutés sont représentés par des observateurs qui sont modélisés comme les autres composants. Les combinaisons des valeurs des variables d'entrée indiquent que le système se trouve ou non dans un état correspondant à un événement redouté modélisé. Dans notre cas d'étude, si le déplacement du pointeau est erroné et qu'il n'est pas en état de gel commandé, alors l'événement redouté : quantité de carburant dosé erronée non signalée au pilote par le calculateur aura lieu.

Les moyens d'analyse

Il existe plusieurs outils et types d'analyse orientés sûreté de fonctionnement : la recherche des effets d'une ou plusieurs défaillances, la recherche des causes d'un événement, et l'évaluation de propriétés (qualitatives ou quantitatives).
La simulation (injection de pannes) permet d'observer la propagation de défaillances. L'arbre de défaillance expose tous les scénarios qui conduisent à un événement redouté et permet d'évaluer la satisfaction de propriétés quantitatives et qualitatives par extraction de coupes minimales. Les séquences proposent les scénarios de N événements conduisant à un événement redouté en tenant compte, contrairement aux coupes minimales, de l'ordre d'apparition des événements (N est le nombre d'événements que l'on souhaite voir apparaître dans les scénarios). Enfin, le model-checking permet l'évaluation de propriétés qualitatives avec la prise en compte du futur (contrairement aux arbres de défaillances et aux séquences qui ne prennent en compte que les événements qui se sont produits jusqu'à l'événement étudié [8], [9].

Toutes ces méthodes sont applicables sur des systèmes statiques et non bouclés. Par contre, lorsqu'un modèle est dynamique, on ne peut plus générer automatiquement d'arbres de défaillance. Dans ce cas, il est conseillé de générer les séquences qui prennent en compte l'ordre d'apparition des événements. Par ailleurs, lorsque le modèle est bouclé instantanément (c'est-à-dire lorsque la valeur d'une entrée d'un composant dépend de la valeur de l'une de ses sorties), il est conseillé de reprendre le modèle pour le déboucler. En effet, dans ce cas, il ne peut pas être généré d'arbre de défaillance, les séquences peuvent être incomplètes, et il ne peut pas être effectué de model-checking (avec l'outil SMV [10] par exemple).

Les résultats du cas d'étude

Le modèle de notre cas d'étude est dynamique (présence de changements d'états induits). Il est donc intéressant de générer (automatiquement avec Cecilia™ OCAS) les séquences d'événements qui mènent aux événements redoutés afin d'identifier les scénarios critiques.
Nous avons étudié les scénarios qui conduisent à l'événement redouté : quantité de carburant dosé erronée non signalée par le calculateur au pilote. Voici les séquences obtenues :

déplacement_doseur . erreur_conception
test_resolver . fausse_détection & MPP. perte_activation
resolver . defaillance & MPP. perte_activation
faisceau_xr . transmission_erronée & MPP. perte_activation
acquisition_xr_OSS . erreur & MPP. perte_activation
test_cohérence . perte_détection & MPP. perte_activation

Nous avons un événement simple qui conduit à cet événement redouté : erreur de conception de la fonction logicielle calcul du déplacement du doseur.
Les séquences de deux événements qui conduisent à l'événement redouté correspondent à une perte d'activation du moteur pas à pas, qui est théoriquement détectable par le test de cohérence, lorsque ce test n'est pas disponible. Les causes d'indisponibilité du test sont :

- une fausse détection du test du resolver (ce qui implique que le test de cohérence n'est plus effectué) ;
- une défaillance sur la chaîne matérielle de surveillance détectée par le test du resolver ;
- une erreur OSS qui a pour effet la transmission erronée de l'information XR ;
- une erreur de conception du test de cohérence (qui implique qu'il ne détecte pas une erreur).

Exploitation des résultats du cas d'étude

Dans ce cas d'étude, nous n'avons étudié qu'un seul événement redouté. Cependant, dans la réalité, il y a d'autres paramètres à prendre en compte. L'exploitation des résultats et les conclusions tirées quant aux criticités des fonctions, ne sont valables que sous les hypothèses énoncées dans ce cas d'étude.

Les séquences présentées ci-dessus nous permettent d'identifier la criticité des fonctions logicielles, et donc le niveau de précaution qu'on doit y apporter lors de la conception du logiciel et le niveau de test lors de la phase de vérification.
Par exemple, nous voyons qu'une erreur de conception au niveau du calcul du déplacement du doseur dont l'effet serait d'élaborer une consigne de déplacement erronée, conduit directement à l'événement redouté étudié ici. Elle est donc critique. Nous proposons alors d'écrire l'exigence de sécurité sur la fonction logicielle : la fonction de calcul du déplacement du doseur ne doit pas contenir d'erreur qui conduit à une commande erronée. Par ailleurs, une fausse détection d'erreur par le test du resolver implique que le test de cohérence n'est plus effectué. Si nous exprimons l'exigence : aucune fausse détection sur le test du resolver, alors nous réduisons le nombre de scénarios qui conduisent à l'événement redouté. Par contre, une fausse détection du test de cohérence n'apparaît pas dans les séquences. Nous en déduisons que pour cet événement redouté, en ce qui concerne les fausses détections, la fonction test du resolver est plus critique que la fonction logicielle test de cohérence. Par contre, en ce qui concerne la détection de toutes les erreurs, la fonction du test de cohérence est plus critique que la fonction du test du resolver. Nous pourrions donc écrire une exigence sur la fonction logicielle test de cohérence : toutes les erreurs doivent être détectées.

Certains outils de modélisation logicielle (comme SCADE SUITE™ par exemple) possèdent un outil qui permet de vérifier automatiquement et formellement des propriétés sur le modèle. Une fois les exigences exprimées sur les fonctions logicielles, nous pourrons écrire des propriétés que doit satisfaire le modèle du logiciel afin de satisfaire ces exigences de sécurité. Par exemple, pour la fonction test du resolver, le modèle devra vérifier la propriété : toute détection du test du resolver implique qu'il se soit produit une panne sur la chaîne de surveillance. Pour la fonction test de cohérence : toutes différences entre la consigne et la mesure du resolver doivent être détectées.
La suite de nos études consistera à écrire ces propriétés pour qu'elles soient vérifiables sur le modèle logiciel. Par ailleurs, nous pourrons nous assurer de l'efficacité du logiciel en se focalisant sur les cas de pannes combinées matérielles et logicielles mis en avant grâce aux séquences. Il s'agira par exemple de cibler les tests sur certains jeux de valeurs des variables d'entrée. Ainsi, nous chercherons à vérifier par ces tests sur le logiciel que les scénarios critiques, identifiés par les séquences, ne peuvent pas se produire.

Conclusion

L'objectif de ces travaux est d'assister la dérivation d'exigences de sécurité des systèmes critiques incluant des composants logiciels. Pour cela nous proposons dans un premier temps d'analyser à quelles conditions l'architecture matérielle et logicielle envisagée satisfaisait bien les exigences de sécurité système. Dans cet article, nous avons défini et illustré une démarche permettant de mener à bien cette analyse. Nous proposons d'expliciter les grandes fonctions et les mécanismes de sécurité réalisés par des composants logiciels ou matériels du système. De plus nous considérons les modes de défaillance des éléments matériels au même titre que les erreurs de conception logicielles potentielles. Cette approche originale nous permet d'étudier l'effet combiné des défaillances matérielles et logicielles au niveau du système dès les phases amont de conception.
La combinatoire des cas à analyser étant traditionnellement grande, la mise en œuvre de l'approche nécessite de disposer de modèles de propagation de panne compositionnels et d'outils d'analyse associés. Nous avons proposé d'utiliser le langage de modélisation AltaRica et l'environnement d'édition et d'analyse Cecilia™ OCAS de Dassault Aviation. Nous avons montré comment les modèles AltaRica pouvaient être construits pour mener à bien les analyses souhaitées.
Les résultats d'analyse considérés sont les séquences d'événements élémentaires qui mettent en avant les erreurs de conception logicielles et les défaillances matérielles qui participent à l'occurrence d'événements redoutés. Ainsi, en ce qui concerne les fonctions logicielles, nous avons montré que nous pouvions déduire (à partir des séquences) des exigences sur les fonctions logicielles.
Les travaux les plus proches des nôtres s'inscrivent dans le courant d'évaluation de sûreté fondé sur l'utilisation de modèles formels dynamiques. Des travaux sur le sujet sont notamment conduits dans le cadre du projet ISAAC [11] et divers environnement de modélisation et d'évaluation ont été développés et testés. Néanmoins, le problème de la méthodologie de modélisation amont d'architecture logicielle et matérielle n'est pas abordé. Ciblé sur le logiciel seul, parmi les travaux les plus anciens, on citera [12] qui proposait d'injecter des déviations dans des modèles fonctionnels de logiciels et d'étudier l'effet de ces déviations par simulation. Plus récemment [13] a repris cette démarche et a tenté d'évaluer ce type de modèles par preuve de propriétés. Dans les deux cas, les auteurs se limitent à des analyses de composant purement logicielles dans des phases relativement avancées de conception. D'autre part, dans le cadre des extensions du langage UML, des travaux proposent des profils pour faciliter les analyses de sécurité des architectures logicielles dans les phases plus amont [14]. Ils proposent des moyens de modélisation et d'analyse alternatifs mais ne définissent pas de méthodologie de modélisation et n'abordent pas le problème de la dérivation d'exigences.
Nos travaux futurs vont porter sur la concrétisation de ces exigences et des séquences menant à l'événement redouté. Nous

allons considérer des spécifications détaillées des fonctions logicielles écrites en SCADE SUITE™ (Esterel Technologies). Il s'agira alors de préciser la définition des erreurs types à éliminer en fonction des détails des algorithmes. En précisant les scénarios précédemment obtenus, on s'efforcera de valider les hypothèses posées en amont. En concrétisant les exigences jusqu'à les rendre testables, nous proposons de démontrer l'absence d'erreur critique. Pour démontrer la satisfaction de ces exigences, nous les transformerons en énoncés formels sur le modèle logiciel et nous prouverons formellement qu'elles ne peuvent pas se produire.
Ainsi, nous espérons améliorer la spécification des fonctions logicielles en y ajoutant les exigences logicielles de sécurité obtenues grâce au modèle AltaRica.

Références

[1] A. Arnold, A. Griffault, G. Point et A. Rauzy. The AltaRica formalism for describing concurrent systems, Fundamenta Informaticae, 40:109-124, 2000.
[2] : C. Castel et C. Seguin, Modèles formels pour l'évaluation de la sûreté de fonctionnement des architectures logicielles d'avionique modulaire intégrée, AFADL, 2001.
[3] : P. Bieber, C. Castel, C. Kehren et C.seguin, Analyse des exigences de sûreté d'un système électrique par model-cheking, Actes du congrès Lambda-Mu 14, 2004.
[4] Gérald Point, AltaRica : Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement. Thèse, LaBRI, Université Bordeaux I, Janvier 2000.
[5] Aymeric Vincent, Conception et réalisation d'un vérificateur de modèles AltaRica, Thèse, LaBRI, Université Bordeaux I, Décembre 2003.
[6] A. Rauzy, Mode automata and their compilation into fault trees. Reliability Engineering and System Safety, 78:1-12, 2002.
[7] : Manuel utilisateur OCAS V3.2, Dassault Aviation, 2005.
[8] C. Kehren, C. Seguin, P. Bieber; C. Castel, C. Bougnol, J.-P. Heckmann et S. Metge. Advanced Multi-System Simulation Capabilities with AltaRica, proceedings of Safety Critial System Conference, Rhodes Island, USA, august 2004.
[9] P. Bieber, C. Bougnol, C. Castel, J.-P. Heckmann, C. Kehren, S. Metge et C. Seguin, Safety assessment with AltaRica - lessons learnt based on two aircraft system studies, 18th IFIP World Computer Congress, Topical Day on New Methods for Avionics Certification, Toulouse (France), 2004.
[10] : K. L. MacMillian, The SMV language, rapport interne Cadence Berkeley Labs, 1999.
[11] O. Akerlund, P. Bieber, E. Boede, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, O. Lisagor, A. Lüdtke, S. Metge, C. Papadopoulos, T. Peikenkamp, L. Sagaspe, C. Seguin, H. Trivedi et L. Valacca, ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects, proceedings Embedded Real Time Software 2006, Toulouse (France), Janvier 2006.
[12] J.D. Reese et N. G. Leveson, Software deviation analysis, proceeding ICSE97, Boston USA, 1997.
[13] A. Joshi et M. Heimdahl, Model-based safety analysis of Simulink models using SCADE Design Verifier, proceeding Safecomp 2005, Fredrikstad (Norway), Septembre 2005, vol 3688 LNCS, Springer.
[14] N. Addouche, C. Antoine et J. Montmain ; Combining extended UML models and formals methods to analyze real-time systems, Poceedings Safecom 2005, Fredrikstad (Norway), Septembre 2005, vol 3688 LNCS, Springer.