Un Point de Vue Sûreté de Fonctionnement pour la Vérification d'Architectures Abstraites

Maria Piriquito
Onera – The French Aerospace Lab
Toulouse, France
Maria-Margarida.Piriquito@onera.fr

Pierre Bieber
Onera – The French Aerospace Lab
Toulouse, France
Pierre.Bieber@onera.fr

Julien Brunel
Onera – The French Aerospace Lab
Toulouse, France
Julien.Brunel@onera.fr

David Chemouil
Onera – The French Aerospace Lab
Toulouse, France
David.Chemouil@onera.fr

ABSTRACT

La vérification de propriétés de sûreté de fonctionnement (SdF) pour les systèmes critiques est un problème primordial. Les solutions existantes imposent généralement une connaissance de l'architecture complète du système. Dans cet article, nous proposons une approche permettant de vérifier efficacement la sûreté de fonctionnement d'architectures abstraites qui sont, éventuellement, partiellement décrites. Nous présentons une méthodologie qui s'appuie sur une vue SdF et une vue architecturale du système. Des propriétés de SdF sont vérifiées au niveau de la vue idoine, et sont postérieurement validées sur l'architecture. Cette approche permet donc de décomposer des vérifications complexes de SdF sur des architectures en deux vérifications plus simples, sur deux vues différentes. Les techniques présentées sont appliquées au système hydraulique d'un avion, utilisé comme cas d'étude.

1. INTRODUCTION

La Sûreté de Fonctionnement s'est développée principalement à cause de l'évolution des systèmes critiques industriels, et se caractérise par l'analyse des défaillances et de leurs conséquences. Cela passe par une analyse exhaustive du fonctionnement du système, ainsi que des exigences que le système doit vérifier.

Les exigences de sûreté de fonctionnement déterminent ce qu'un système doit ou ne doit pas faire pour assurer son bon fonctionnement [18]. Ce domaine d'étude peut être divisé en quatre sous-domaines : *évitement des pannes*, *suppression des pannes*, *détection des pannes* et *tolérance aux pannes* [17]. Chacune de ces techniques propose des solutions pour le même problème, mais en utilisant des approches différentes.

Notre travail se place dans le domaine de la *tolérance aux pannes*, *c-a-d* les techniques qui permettent au système de continuer à fonctionner correctement même en présence de pannes. Le but principal de notre travail est de fournir une réponse rapide et efficace à la vérification de certaines exigences de tolérance aux pannes sur une architecture abstraite, et, éventuellement partiellement décrite. Dans notre approche nous décomposons les vérifications complexes de SdF en deux vérifications plus simples, une au niveau de la vue SdF, et une autre au niveau de la vue architecturale. D'autres approches au problème fournissent des résultats plus proches de la réalité, mais ce sont aussi des méthodes très complexes, par exemple Altarica [2, 13], et qui nécessitent une connaissance très approfondie du système étudié (notamment en ce qui concerne l'architecture).

Pour appliquer nos techniques de vérification nous avons besoin d'une architecture sur laquelle vérifier les propriétés. Les Langages de Description d'Architecture (ADL) sont utilisées pour décrire des architectures de système en utilisant les concepts de composants et connecteurs. Dans ce travail, nous proposons un ADL qui permet une description très simple de l'architecture d'un système à étudier. Ce langage permet la définition de composants et leur lien par l'intermédiaire de connecteurs prédéfinis. Les détails sur le langage sont donnés dans la Section 2. L'architecture décrite avec ce langage va donner naissance à la vue architecturale du système, *c-a-d* une représentation du système selon certains critères, sur laquelle des vérifications seront effectuées.

La construction du point de vue sûreté de fonctionnement est basé sur la définition de plusieurs vues sûreté de fonctionnement. Chaque vue SdF propose une solution à une exigence différente, *c-a-d* que toutes les exigences de tolérance aux pannes vont avoir une vue SdF associée. C'est l'ensemble de toutes ces vues qui compose le point de vue sûreté de fonctionnement. La définition de chaque vue est basée sur l'application successive de stratégies de tolérance aux pannes aux composants du système. Les exigences trai-

tées et la méthode de construction des vues sont décrits dans la Section 3.

Pour faire le lien entre le point de vue SdF (constitué de toutes les vues SdF) et la vue architecturale du système nous introduisons la notion de *zone*. Les *zones* mettent en relation les éléments de la vue architecturale et les stratégies de la vue SdF, et regroupent un ensemble de composants et connecteurs de l'architecture. Chaque application d'une stratégie SdF dans une vue peut donner lieu à une zone, dans le cas où tous les critères d'application de la stratégie sont vérifiés par l'architecture comprise dans la zone. Si toutes les zones sont créées et satisfont tous les critères, alors il est possible de dire que l'architecture satisfait l'exigence de tolérance aux pannes représentée par la vue SdF correspondante. La méthodologie utilisée est décrite dans la Section 3.4.

Pour mieux expliquer notre méthodologie nous proposons l'application des concepts introduits à un cas d'étude décrivant le système hydraulique d'un avion. Cet exemple est introduit dans la Section 2.2.

2. VUE ARCHITECTURALE

Dans cet article, nous introduisons un nouvel ADL qui permet la description d'une vue architecturale d'un système. Le meta-modèle qui représente les éléments de cette vue, ainsi que de la vue SdF et le lien entre les deux par les *zones* provenant des stratégies SdF utilisées est présenté dans la Figure 1.

En plus de ce qui est représenté dans le meta-modèle, d'autres contraintes liées à la vue architecturale et aux zones doivent être vérifiées :

- Les connecteurs doivent avoir au moins un InRole et un OutRole.
- Un OutPort est toujours connecté à un InRole.
- Un OutRole est toujours connecté à un InPort ou à un InRole.
- Une Stratégie est toujours lié par une zone à, au moins, un Composant.
- Un connecteur entre deux composants de la même zone appartient à la zone.
- Un connecteur entre deux composants de deux zones différentes n'appartiennent pas à la zone.

Dans cette Section nous présentons, dans un premier temps, la syntaxe du langage et, dans un deuxième temps, son utilisation en se servant du langage pour décrire le système hydraulique d'un avion.

2.1 Syntaxe et Sémantique des Connecteurs

Notre ADL permet la définition de *composants* et *connecteurs*.

Définition 1 *Un* Composant *représente une fonction du système à décrire, et contient un nom comme identifiant. Chaque composant doit avoir au moins un* port *d'entrée ou de sortie.*

Définition 2 *Un* connecteur *représente une façon de lier deux composants entre eux. Chaque connecteur doit avoir au moins un* rôle *d'entrée et un de sortie.*

Les deux éléments du langage, les composants et les connecteurs, ont un *statut*. Le statut d'un élément représente son intervention dans l'architecture à un certain moment. Les états possibles pour le statut sont *actif* ou *inactif*.

Définition 3 *Un* composant inactif *est un composant qui n'introduit pas de pannes dans le système.*

Définition 4 *Un* connecteur inactif *est un connecteur qui ne propage pas les pannes dans le système.*

Pour la construction de la vue architecturale du système, tous les composants et connecteurs sont considérés comme étant actifs. L'inactivité d'un élément n'est considérée qu'à la définition d'une vue SdF.

2.1.1 Composants et Ports

Les *composants* sont un des concepts de base de notre langage et représentent les fonctions du système. Les *ports* sur un composant représentent des connexions possibles avec d'autres composants. Au niveau de la syntaxe, les composants sont des rectangles, les InPorts sont les triangles à l'intérieur de la boite, et les OutPorts sont des triangles à l'extérieur de la boite. Chaque composant a un nom écrit à son intérieur qui permet son identification dans le système :

Figure 2: Syntaxe d'un Composant nommé *Component* avec un InPort et un OutPort

Tout composant actif peut avoir un bon comportement ou alors introduire une panne dans le système. Les pannes introduites par les composants seront propagées dans le système par les connecteurs.

Dans notre ADL, un connecteur générique ne peut pas être instancié. Ainsi, nous présentons tout de suite les trois connecteurs de base inclus dans le langage, ainsi que la représentation graphique que chaque instance de ces connecteurs doit avoir. Ces connecteurs peuvent se lier entre eux pour créer des connecteurs plus complexes.

2.1.2 Le Connecteur Link

Le connecteur *Link* sert à définir une connexion standard entre deux composants. Il contient un InRole *in* et un OutRole *out*, et les données reçues dans *in* sont renvoyées vers *out* sans introduire de modifications. Une instance d'un connecteur *link* dans une architecture est représentée par :

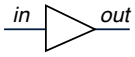

Figure 3: Syntaxe d'une instance d'un connecteur *Link*

Si le connecteur est actif, alors toute panne rentrante en *in* est transmise via *out*.

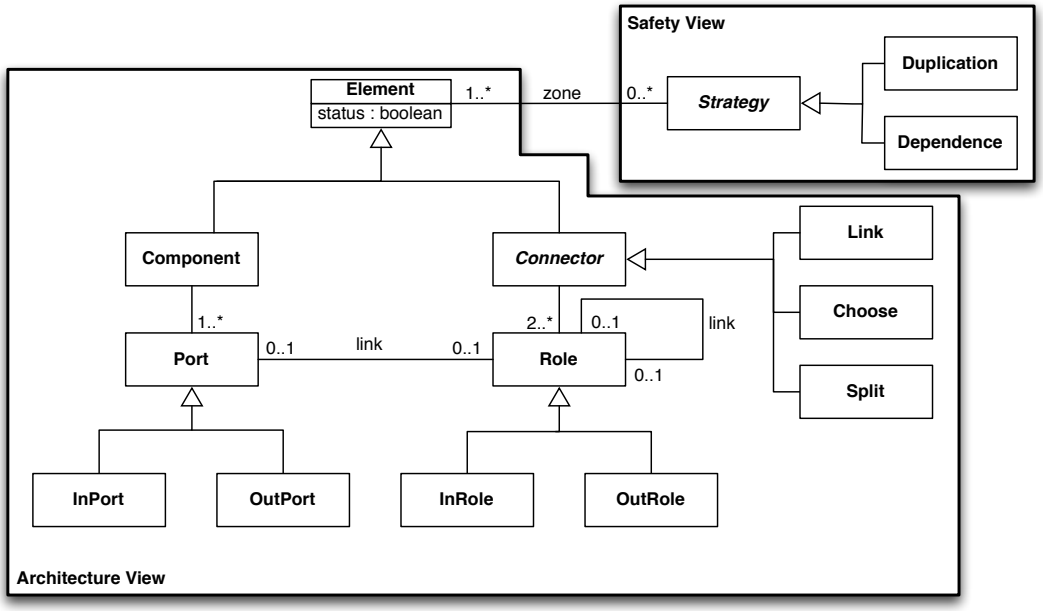

Figure 1: Meta-Modèle des Éléments des Différentes Vues de notre Approche

2.1.3 Le Connecteur Choose

Ce connecteur contient deux InRoles *in1* et *in2* et un seul OutRole *out*. Sa syntaxe est représentée visuellement par :

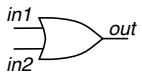

Figure 4: Syntaxe d'une instance d'un connecteur *Choose*

Pour ce qui est de son comportement, il est similaire a celui du connecteur *Link*, mais en prenant ses données soit de *in1* soit de *in2*. Il suffit que l'entrée soit disponible dans un des InPorts pour que sa valeur soit transmise vers *out*.

Dans le cas où le connecteur est actif, alors, si une panne existe dans les deux entrées *in1* et *in2* simultanément, elle sera transmise à la sortie *out*. Si une seule panne existe, elle ne sera pas propagée.

2.1.4 Le Connecteur Split

Le connecteur *Split* a le comportement inverse du connecteur *Choose*. Il a un seul InRole *in* et deux OutRoles *out1* et *out2*, et les données rentrantes, si existantes, devront être transmises vers, au moins, *out1* ou *out2*. Chaque instance de ce connecteur doit être représentée en utilisant la syntaxe :

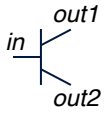

Figure 5: Syntaxe d'une instance d'un connecteur *Split*

En cas de panne dans l'entrée *in* d'un connecteur *Split* actif, alors celle-ci sera transmise aux deux sorties *out1* et *out2*.

2.1.5 Liens

Les liens entre éléments du langage (ports et rôles) sont symboliques, *c-à-d* qu'ils ne servent qu'à exprimer quel port est connecté à quel rôle. On les représente par une ligne qui lie un port à un rôle, ou un rôle à un rôle.

2.2 Le Système Hydraulique d'un Avion

Pour aider à la compréhension de notre langage, nous allons l'appliquer à un exemple basé sur le système hydraulique d'un avion. Dans cet exemple les composants utilisés sont d'une part physiques et d'autre part logiques. Chaque composant physique contient un contrôleur logique qui détermine l'état du composant : marche, ne marche pas, erroné...

Le système utilisé en tant qu'exemple dans cette Section est basé sur l'exemple décrit en [16]. L'exemple original présente un système hydraulique d'un avion A320, séparé en trois systèmes de distribution, deux principaux et un système de secours. Pour simplifier la présentation de notre travail nous avons simplifié l'exemple en enlevant une des lignes de distribution et en simplifiant certains fonctionnements.

Le système hydraulique que nous présentons est composé par deux lignes de distribution, composée de fournisseurs d'énergie, de pompes et de réservoirs. Dans ce cas nous considérons deux lignes de distribution, *vert*, identifiée par l'index *g*, et *jaune*, identifiée par l'index *y*. Les composants *DISTg* et *DISTy* représentent la connexion entre les lignes de distribution et le reste du système (pas représenté dans cet exemple). Pour fournir de l'énergie il y a deux moteurs de l'avion, *ENG1* et *ENG2*, ainsi qu'un générateur électrique *ELEC* pour alimenter la ligne de distribution *jaune*.

Chaque ligne de distribution est constituée d'un réservoir, *RSVg* et *RSVy*, connecté aux pompes *EDPg* et *EDPy* respectivement. En cas de défaillance au niveau de la pompe *EDPg*, il y a un autre mécanisme de secours, le *PTU* qui permet d'alimenter le système de distribution *vert* en utilisant une des pompes du système *jaune*.

En utilisant notre ADL pour définir le système, on obtient l'architecture de la Figure 6.

Cette architecture sera utilisée comme base dans les sections suivantes pour appliquer la définition des vues sûreté de fonctionnement, et pour effectuer la vérification des exigences de sûreté de fonctionnement sur cette même architecture.

3. VUE SÛRETÉ DE FONCTIONNEMENT POUR LES ARCHITECTURES

Dans cet article, nous nous focalisons sur la tolérance aux pannes. Son but est de garantir qu'un système fournisse un service même en présence de pannes de certains de ces composants. Le développement des systèmes tolérants aux pannes est essentiellement basé sur la redondance. Le système contient au moins deux composants fournissant le même service. Dans ce cas, si l'un des composants tombe en panne les autres composant sont toujours aptes à fournir le service. Le système hydraulique décrit dans la figure 6 contient plusieurs composants redondants : les lignes de distribution DISTy et DISTg, les pompes EDPg, EMPy et EDPy ou les moteurs de l'avion ENG1 et ENG2.

Le niveau de redondance présent dans l'architecture d'un système répond à des exigences de tolérances aux pannes. Pour des raisons de coûts, d'encombrement ou de masse il n'est pas souhaitable de dupliquer tous les composants d'un système. Aussi, il est possible que la panne d'un composant impacte plusieurs composants redondants. Ceci peut avoir pour effet d'invalider des exigences de SdF. Dans le système hydraulique, le réservoir RSVy n'est pas dupliqué, il est utilisé par les pompes EMPy et EDPy. Par conséquent, lorsque le réservoir ne fournit plus de fluide hydraulique, les deux pompes EMPy et EDPy ne peuvent plus fournir de fluide sous pression. Dans ce cas, c'est le système hydraulique vert qui utilise des composants indépendants de RSVy comme un réservoir RSVg, une pompe EDPg et une ligne de distribution DISTg qui fournira la puissance hydraulique.

Le but de notre travail est de proposer une façon d'analyser l'architecture d'un système afin de vérifier la tenue d'exigences de SdF.

3.1 Exigences de SdF

Les exigences de SdF sont des exigences non-fonctionnelles, elles ne contraignent pas ce que le système fait (par exemple, fournir de la puissance hydraulique) mais elles imposent des conditions sur la façon dont c'est fait (par exemple, fournir de la puissance hydraulique lorsque le moteur ENG1 est en panne).

Les exigences de tolérance aux pannes sont issues de l'analyse des risques qui évalue l'impact de la perte d'un service sur la vie des passagers, sur l'intégrité de l'avion et sur la charge de travail des pilotes. La gravité de l'impact détermine le niveau de tolérance aux pannes requis. Plusieurs types d'exigences de tolérance aux pannes existent. Nous traitons des exigences permettant de décrire les combinaisons de pannes qu'un système (ou qu'une partie du système) doit tolérer. Ces combinaisons de pannes sont habituellement caractérisées par un nombre de pannes (pannes simples, doubles ou triples). Elles peuvent également décrire un ensemble de composants susceptible de tomber en panne.

Dans notre exemple, lorsque le système hydraulique n'est plus capable de fournir de la pression hydraulique, l'avion n'est plus gouvernable ceci peut conduire à la perte de l'avion et de ses passagers. Par conséquent, l'impact de la perte totale du système hydraulique est très grave. L'architecture décrite dans la figure 6 représente une portion de système hydraulique dont la perte aurait un impact moindre. Nous considérons dans la suite deux exigences que ce système doit satisfaire :

- **Exigence 1 :** Le système hydraulique doit tolérer une panne simple,
- **Exigence 2 :** Le système hydraulique vert doit tolérer la panne d'un des deux moteurs.

3.2 Les Stratégies de Tolérance aux Pannes

Pour vérifier les exigences de SdF présentées, nous construisons une vue SdF du système. Cette vue va être basée dans la décomposition du système en utilisant des stratégies de tolérance aux fautes introduites en [3] et adaptées à notre travail. Les deux stratégies utilisées sont la *Dépendance* et la *Duplication*. Chaque stratégie comprend des hypothèses sur son applicabilité, ainsi que des propriétés garanties. La vérification des hypothèses est effectuée en analysant l'architecture fonctionnelle d'un système, et le processus est décrit dans la Section 3.4.

Définition 5 *Soit c un composant (ou ensemble de composants) du système. Alors la fonction loss(c) représente le nombre minimal de pannes nécessaires pour perdre c.*

Définition 6 *Soit c un composant (ou ensemble de composants). Si toute autre information de tolérance aux pannes est inexistante, alors :*

$$loss(c) = \begin{cases} 1 & \text{si } c \text{ est actif} \\ 0 & \text{si } c \text{ est inactif} \end{cases}$$

3.2.1 La Stratégie Dépendance

Cette stratégie permet, comme son nom l'indique, la représentation de la dépendance entre plusieurs fonctions du système. L'utilisation de cette stratégie n'apporte rien dans l'amélioration de la tolérance aux pannes. Son utilité est visible au niveau de la composition de stratégies pour créer une vue SdF. L'application de la stratégie est soumise à une hypothèse, et garantit une propriété de tolérance aux pannes.

Définition 7 *Soient $c_1, ..., c_n$ plusieurs composants (ou ensemble de composants) du système. Alors une dépendance entre $c_1, ..., c_n$ est définie par $dep(c_1, ..., c_n)$.*

Hypothèse 1 *Soit C l'ensemble des composants $c_1, ..., c_n$ en dépendance. Alors il existe un connecteur liant chaque composant $c \in C$ à un composant $c' \in C \setminus \{c\}$.*

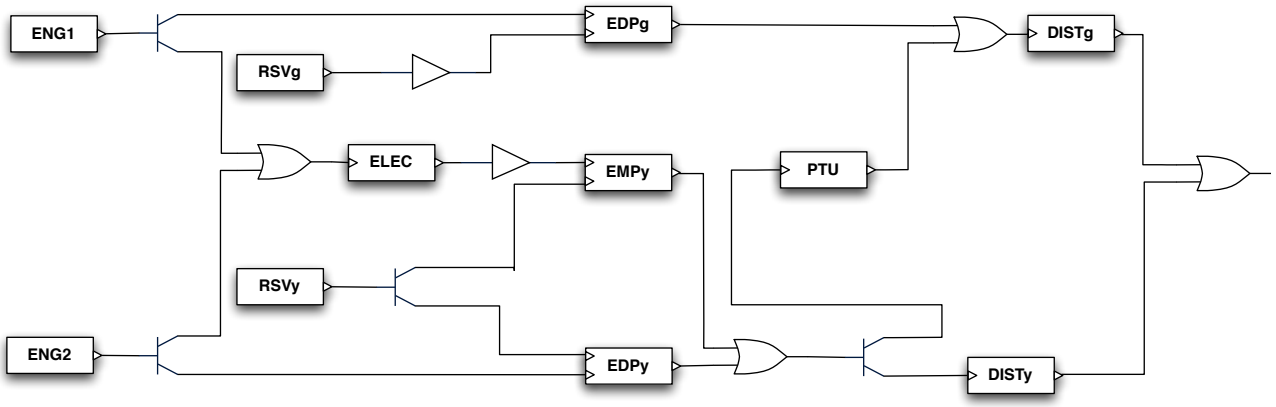

Figure 6: Architecture Fonctionnelle du Système Hydraulique d'un Avion

Cette hypothèse garantit que, effectivement, les différentes entités ne sont pas isolées, et que la perte d'une d'entre elles va déclencher la perte de l'ensemble.

Propriété Garantie 1 *La tolérance aux pannes donnée par la fonction loss appliquée à toute dépendance* $dep(c_1, ..., c_n)$ *est donnée par l'expression :*

$$loss(dep(c_1, ..., c_n)) = min(loss(c_1), ..., loss(c_n))$$

3.2.2 La Stratégie Duplication

La stratégie duplication sert à représenter deux composants (ou ensembles de composants) du système qui offrent le même service et qui peuvent être utilisés de façon indépendante. L'application de cette stratégie est soumise à trois hypothèses et garantit une propriété de tolérance aux pannes.

Définition 8 *Soit* c_1 *et* c_2 *deux composants (ou ensemble de composants) du système. Alors une duplication entre* c_1 *et* c_2 *est définie par* $dup(c_1, c_2)$.

Hypothèse 2 *Il existe un connecteur actif du type* Choose *auquel* c_1 *et* c_2 *sont liés.*

Cette hypothèse sert à garantir que c_1 et c_2 fournissent leur service au même composant du système, introduisant ainsi une redondance. Deux composants qui fournissent le même service, mais utilisés à des endroits différents du système n'introduisent pas une redondance.

Hypothèse 3 *Il n'y a aucun connecteur ou composant actif (autre qu'un connecteur du type* Choose*) entre les composants qui constituent* c_1 *et ceux qui constituent* c_2.

Hypothèse 4 *Aucun composant qui constitue* c_1 *est présent dans la constitution de* c_2. *Aucun composant qui constitue* c_2 *est présent dans la constitution de* c_1.

Ces deux hypothèses permettent d'établir l'indépendance entre c_1 et c_2. Deux composants sont dits indépendants si une panne dans un des composants n'affecte pas l'autre composant. Ainsi, s'il n'y a pas de connecteurs actifs entre les composants de c_1 et ceux de c_2, les pannes de c_1 n'auront pas d'influence sur les pannes de c_2 et vice-versa.

L'hypothèse 4 est liée à l'hypothèse 2, dans la mesure où les pannes de c_1 et de c_2 doivent être indépendantes. Si c_1 et c_2 partagent un composant, une panne sur ce composant risque d'affecter les deux entités et l'indépendance ne sera donc pas vérifiée.

Propriété Garantie 2 *La tolérance aux pannes donnée par la fonction loss appliquée à toute duplication* $dup(c_1, c_2)$ *est donnée par l'expression :*

$$loss(dup(c_1, c_2)) = loss(c_1) + loss(c_2)$$

Dans la Section qui suit nous appliquerons ces deux stratégies pour construire deux vues SdF du système qui répondent chacune à une des exigence de tolérance aux pannes présentées dans la Section 3.1. L'architecture sur laquelle nous nous basons est celle de la Figure 6.

3.3 Vue Sûreté de Fonctionnement

Pour vérifier chaque exigence de tolérance aux pannes nous construisons une vue SdF qui lui correspond. Chaque exigence aura ainsi sa vue SdF. Une vue SdF est créée par l'application successive de stratégies de tolérance aux pannes, et représentée sous la forme d'un arbre d'application de stratégies dans lequel les nœuds sont donnés par les stratégies appliquées, et les feuilles par les composants du système. À ce niveau l'architecture du système doit déjà exister. Par contre, pour construire les vues SdF, nous n'avons besoin que de l'information sur les composants disponibles, et pas sur les liens entre eux.

Pour aider à la compréhension de la création des vues SdF, nous allons maintenant créer les deux vues qui correspondent aux deux exigences SdF introduites dans la Section 3.1 sur les composants de l'architecture de la Figure 6.

3.3.1 Vue SdF pour l'Exigence 1

L'exigence 1 affirme que le système hydraulique doit tolérer une panne simple, ce qui veut dire que, même en présence d'une panne dans un des composants, il y a toujours de la puissance hydraulique disponible au niveau du composant *DISTg* ou du composant *DISTy*.

Une solution possible est de dire que la ligne de distribution

vert, alimentée par le moteur *ENG1* établit une duplication avec la ligne de distribution *jaune* alimentée par *ENG2*. Ainsi, la première vue de sûreté de fonctionnement obtenue est la suivante :

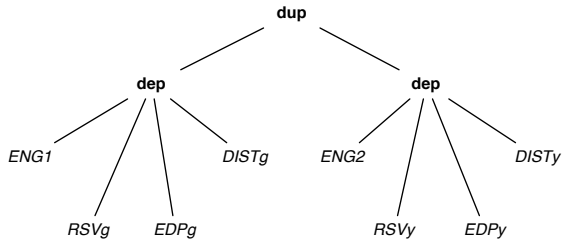

Figure 7: Une vue SdF pour l'exigence 1

Chaque vue SdF est représentée par un arbre d'application de stratégies. Les différentes stratégies appliquées sont représentées en **gras**, et les fonctions du système utilisées dans la construction de la vue SdF sont représentées en *italique*.

Une autre représentation de la vue est donnée par :

$$\begin{aligned} &Vue\_SdF\_Exg1\_SysHyd = dup(\\ &\quad dep(ENG1, RSVg, EDPg, DISTg), \\ &\quad dep(ENG2, RSVy, EDPy, DISTy)) \end{aligned}$$

Une fois la vue SdF construite, il faut s'assurer qu'elle satisfait bien l'exigence donnée. Dans le cas de l'exigence 1, on veut que $loss(Vue\_SdF\_Exg1\_SysHyd) = 2$.

Le calcul est effectué en utilisant les propriétés garanties de chaque stratégie, introduites dans la Section 3.2 est :

$$\begin{aligned} &loss(Vue\_SdF\_Exg1\_SysHyd) = \\ &\quad loss(dep(ENG1, RSVg, EDPg, DISTg)) + \\ &\quad loss(dep(ENG2, RSVy, EDPy, DISTy)) \end{aligned}$$

dont le résultat est, effectivement, 2. Cela veut dire que deux pannes sont nécessaires pour perdre les deux lignes de distribution considérées, c-à-d que le système tolère une panne simple comme le demande l'exigence.

Pour cette analyse tous les composants ne sont pas utilisés. Les composants inutilisés sont considérés comme inactifs, c-à-d qu'ils n'introduisent pas de pannes dans le système.

3.3.2 Vue SdF pour l'Exigence 2

L'exigence 2 dit que le système hydraulique vert doit tolérer la panne d'un des deux moteurs. Cela veut dire que, même si un des deux moteurs tombe en panne, le système arrive quand même à fournir de la puissance hydraulique.

Pour la construction de cette vue SdF nous commençons par créer une duplication entre les fonctions *ENG1* et *ENG2*, vu que c'est au niveau de ces fonctions que l'exigence est basée. Plusieurs décompositions des fonctions du système peuvent être effectuées. L'important est que la décomposition proposée satisfasse l'exigence. Pour cette exigence, nous proposons la décomposition suivante :

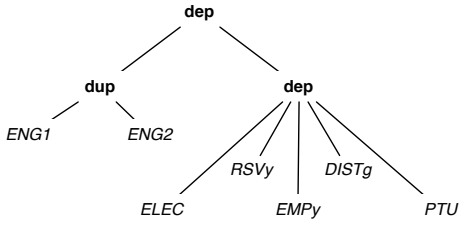

Figure 8: Une vue SdF pour l'exigence 2

Encore une fois, les différentes stratégies appliquées sont représentées en **gras**, et les fonctions du système utilisées dans la construction de la vue SdF sont représentées en *italique*.

Une autre représentation de la vue est donnée par :

$$\begin{aligned} &Vue\_SdF\_Exg2\_SysHyd = dep(\\ &\quad dup(ENG1, ENG2), \\ &\quad dep(ELEC, RSVy, EMPy, DISTg, PTU)) \end{aligned}$$

Pour la vérification de l'exigence 2 sur cette vue SdF if faut tenir compte de l'impact de la perte d'un des deux composants spécifiques. Dans ce cas nous considérons la duplication comme étant un seul élément du système, c-à-d, $loss(dup(ENG1, ENG2) = 1$.

L'exigence est vérifiée si $loss(Vue\_SdF\_Exg2\_SysHyd) = 1)$, c-à-d si, dans le cas où nous ne disposons que d'un seul moteur, il faut quand même une panne en plus pour faire tomber la distribution. En utilisant les propriétés garanties par les différentes stratégies utilisées nous obtenons :

$$\begin{aligned} &loss(Vue\_SdF\_Exg2\_SysHyd) = min(\\ &\quad loss(dup(ENG1, ENG2)), \\ &\quad loss(dep(ELEC, RSVy, EMPy, DISTg, PTU))) \end{aligned}$$

En effectuant le calcul nous pouvons vérifier que le résultat est celui souhaité, et donc que la vue SdF proposée satisfait bien l'exigence 2.

3.4 Lien entre les Vues SdF et la Vue Architecturale par Intermédiaire des Zones

Dans cette section, nous présentons le lien entre les vues SdF et la vue architecturale du système en introduisant la notion de *zone*. L'introduction de ce concept donne naissance à plusieurs vues architecturales, basées sur la vue architecturale de base, mais avec l'information sur les *zones*.

Définition 9 *Une* zone *est une entité abstraite composée par des composants et des connecteurs de l'architecture, et qui représente une partie du système. Les connecteurs entre deux composants d'une même zone appartiennent à la zone aussi. Les connecteurs entre des composants de deux zones différentes n'appartiennent pas a la zone.*

C'est à travers des *zones* que les exigences de tolérance aux pannes utilisées pour créer les vues SdF vont être vérifiées sur l'architecture du système.

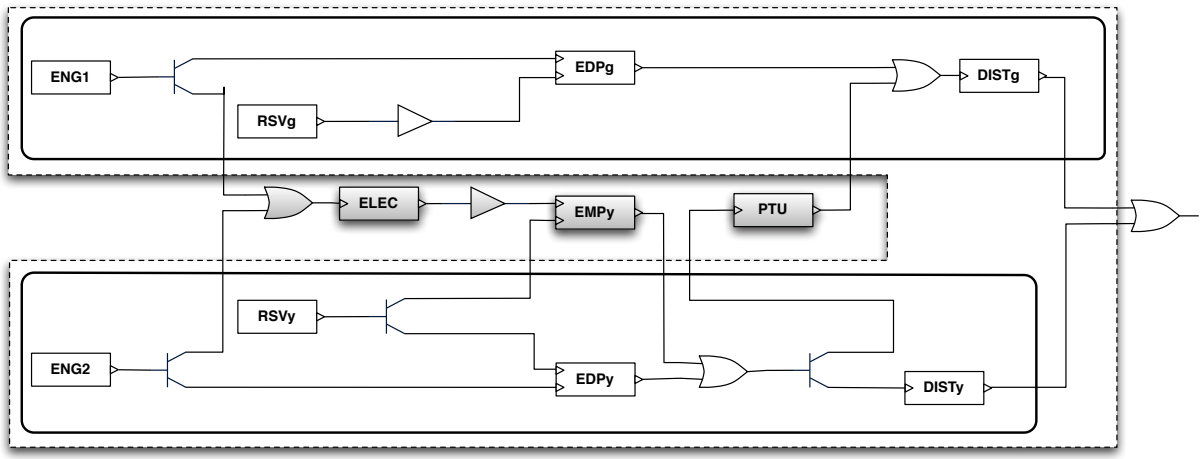

Figure 9: Vue Architecturale Correspondant à l'Exigence 1

Pour créer la vue architecturale correspondant à une exigence nous utilisons la vue SdF de cette même exigence. Chaque nœud de l'arbre qui représente la vue, et qui correspond à une stratégie de tolérance aux pannes, va donner lieu à une zone dans l'architecture. Comme défini dans la Section 3.2, chaque stratégie est basée sur certaines hypothèses. C'est au niveau de chaque zone que ces hypothèses doivent être vérifiées.

3.4.1 Vue Architecturale pour l'Exigence 1

La génération d'une vue architecturale pour une exigence consiste à parcourir l'arbre qui représente la vue SdF pour l'exigence voulue, et à introduire dans l'architecture une zone pour chaque stratégie utilisée. La vue architecturale pour l'exigence 1 de la Section 3.3 est présentée dans la Figure 9.

Les formes en **gras** et coins arrondis représentent les zones obtenues par la stratégie de tolérance aux pannes dépendance. Les formes en pointillé représentent les zones obtenues via la stratégie duplication. Tous les composants et connecteurs inactifs sont représentés en gris.

Pour garantir que l'architecture satisfait l'exigence souhaitée, il faut que, pour toutes les zones définies, les hypothèses de la stratégie qui était à son origine soient vérifiées.

Pour les zones originaires d'une dépendance il faut vérifier l'existence de, au moins, pour chaque composant de la zone, un connecteur qui le lie à un autre composant de la zone. Dans notre exemple ceci est vérifié pour les deux zones de ce type.

Pour la vérification de la seule zone originaire d'une stratégie duplication, il faut vérifier les trois hypothèses décrites dans la Section 3.2.2.

Soient c_1 et c_2 les résultats de l'application de la stratégie dépendance utilisés dans la stratégie duplication. Alors, il faut prouver qu'il existe un connecteur actif du type *Choose* auquel c_1 et c_2 sont liés.

Proof Hypothèse 2. Les composants *DISTg* et *DISTy* sont tous les deux connectés au même connecteur *Choose* (visible dans l'architecture). Vu que chaque composant appartient à un élément composé différent, alors c_1 et c_2 ont tous les deux une sortie vers le même composant *Choose*, et la propriété est vérifiée. □

La deuxième hypothèse à prouver dit qu'aucun composant ou connecteur actif (autre que *Choose*) n'existe entre les composants constituant c_1 et ceux constituant c_2.

Proof Hypothèse 3. Dans l'architecture il n'existe qu'un seul connecteur actif entre c_1 et c_2, et ce connecteur est du type *Choose*. L'hypothèse est ainsi vérifiée. □

La dernière hypothèse considère qu'aucun composant est commun à c_1 et a c_2.

Proof Hypothèse 4. Les composants qui constituent c_1 sont : *ENG1*, *RSVg*, *EDPg* et *DISTg*, et ceux qui constituent c_2 sont : *ENG2*, *RSVy*, *EDPy* et *DISTy*. Aucun composant n'existe dans c_1 et c_2 simultanément. L'hypothèse est donc vérifiée. □

Comme toutes les les zones satisfont bien les hypothèses prévues par la stratégie utilisée pour les construire, il est alors possible de dire que la vue architecturale obtenue est correcte, et correspond a la vue sûreté de fonctionnement qui satisfait l'exigence 1. Ceci veut dire que l'architecture satisfait, elle aussi, l'exigence 1.

3.4.2 Vue Architecturale pour l'Exigence 2

En effectuant le même processus pour la vue SdF correspondant à l'exigence 2 de la Section 3.3 nous obtenons la vue architecturale présentée dans la Figure 10.

Tout comme dans le premier exemple, les hypothèse de la stratégie dépendance sont facilement vérifiées. Pour la zone originaire d'une duplication entre les composants *ENG1* et

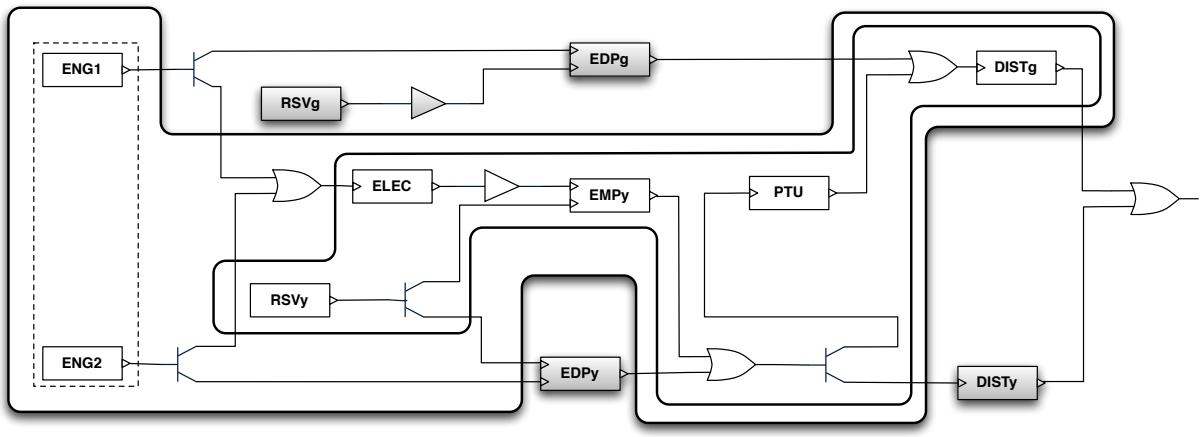

Figure 10: Vue Architecturale Correspondant à l'Exigence 2

ENG2 il faut vérifier les hypothèses d'application de la stratégie :

Proof Hypothèse 2. Les composants *ENG1* et *ENG2* sont tous les deux connectés au même connecteur *Choose* (ce qui est visible dans l'architecture). L'hypothèse est donc vérifiée. □

Proof Hypothèse 3. Dans l'architecture il n'existe aucun autre connecteur que *Choose* entre *ENG1* et *ENG2*. L'hypothèse est ainsi vérifiée. □

Proof Hypothèse 4. Les composants *ENG1* et *ENG2* sont deux composants différents. L'hypothèse est donc vérifiée. □

Vu que toutes les zones satisfont bien les propriétés de la stratégie utilisée pour les construire, il est possible de dire que la vue architecturale proposée est correcte et correspond a la vue sûreté de fonctionnement qui satisfait l'exigence 2. Ceci veut dire que la vue architecturale de base satisfait, elle aussi, l'exigence 2.

Avec ce travail nous avons proposé une méthodologie qui, en faisant des vérifications simples au niveau d'une vue sûreté de fonctionnement d'un système, et en réalisant d'autres vérifications simples au niveau d'une vue architecturale, est capable de déterminer si une architecture satisfait ou pas une exigence de tolérances aux pannes.

4. COMPARAISON ENTRE APPROCHES

La sûreté de fonctionnement est classée en différents domaines. L'étude de la *fiabilité* d'un système, offre la possibilité de quantifier la qualité d'un système par rapport à sa probabilité de défaillance. La *tolérance aux pannes* introduit les méthodologies qui permettent l'augmentation de la fiabilité d'un système.

L'objectif de ce travail ne se centrait pas sur le calcul de la fiabilité d'un système, ni sur les techniques de tolérance aux pannes pour augmenter cette fiabilité, mais sur la vérification de certaines propriétés de sûreté de fonctionnement sur une architecture d'un système. Les travaux sur les techniques de tolérance aux pannes, notamment sur les modèles d'architecture pour la tolérance aux pannes [3] sont essentiels pour notre travail car il nous permettent la définition de nouvelles stratégies de tolérance aux pannes.

Les travaux les plus importants dans la vérification d'exigences de tolérance aux pannes sur des architectures proposent des langages ou modèles de description d'architectures qui permettent la vérification de propriétés liées à la sûreté de fonctionnement.

Nous ne nous comparons pas aux travaux qui proposent des ADL et des méthodes pour introduire des exigences fonctionnelles dans le système, comme les ADL Darwin [11, 12], Rapide [10], ACME [8] ou SysML [1], vu que, même si nous proposons un ADL dans notre travail, cet ADL est très simple et ne sert que de moyen de décrire des architectures abstraites.

Le noyau de notre travail est dans la vérification d'exigences de sûreté de fonctionnement (non-fonctionnelles) sur les architectures. Les approches principales, qui nous ont inspiré pour ce travail, sont les travaux sur la sûreté de fonctionnement en utilisant les langages AltaRica [2, 13], EastADL [5], AADL [7], et EASIS (Electronic Architecture and System engineering for Integrated Safety systems)[6].

AltaRica permet la formalisation du comportement des systèmes en présence de pannes, ainsi que l'analyse de modèles en utilisant les nombreux outils disponibles. La vue dysfonctionnelle d'un modèle AltaRica est possible grâce à l'introduction d'événements qui modélisent la défaillance des composants. Sa capacité à réaliser des modèles compositionnels et hiérarchiques lui permet de modéliser des systèmes complexes. Ce langage très riche était à la base de notre travail. Notre but, par contre, c'était de proposer des vérifications de SdF, éventuellement avec des résultats mois précis que ceux obtenus avec AltaRica, sur des architectures simples. Les modèles AltaRica décrivent tout le système en détail, y compris son fonctionnement, tandis que nous imposons juste les différents composants du système et les liens entre eux.

Les travaux présentés par Kehren dans [9] se basent sur

la définition du comportement des systèmes en présence de pannes, modélisés en utilisant AltaRica. Il propose des méthodes qui assistent la modélisation et évaluation qualitative de l'architecture de sûreté de fonctionnement de systèmes embarqués complexes. La partie de son travail la plus proche du nôtre se base dans la validation d'exigences à base de motifs. Pour faire cette vérification il procède à la détection de motifs dans l'architecture et, après, à la vérification de la satisfaction des propriétés internes et d'environnement. Notre approche est très similaire. La grande différence se trouve dans l'identification des motifs dans l'architecture, que, dans notre cas, nous obtenons à partir des stratégies de tolérance aux pannes utilisées dans l'analyse de SdF. La vérification des exigences dans son approche est beaucoup plus formelle et se base sur le raffinement d'architectures, tandis que notre approche se base sur des propriétés très simples à vérifier sur les composants et connecteurs utilisés dans chaque stratégie.

En ce qui concerne EastADL, son but principal est d'intégrer des analyses de sûreté de fonctionnement et de performance dans le langage, de façon à répondre aux besoins de l'industrie automobile. Le langage en soit ne couvre que des aspects fonctionnels, pourtant, le consortium du projet européen ATESST (Advancing Traffic Efficiency and Safety through Software Technology) [4] vise à intégrer d'autres analyses, y compris des analyses de sûreté de fonctionnement et de performance dans ce langage.

EASIS (Electronic Architecture and System engineering for Integrated Safety systems) [6] représente un partenariat entre plusieurs manufacturiers et distributeurs de véhicules, créateurs d'outils et instituts de recherche européens qui vise le développement de technologies pour la définitions de systèmes embarqués critiques. La sûreté de fonctionnement est étudiée pour chaque module individuel. Les modules sont, postérieurement, assemblés pour créer le système complet. L'analyse complète de SdF sur toute l'architecture n'est donc pas possible dans cette approche.

Le langage AADL [7] permet la conception et l'analyse de systèmes complexes, critiques et temps réel dans plusieurs milieux industriels. Le langage comporte une extension qui sert à décrire les caractéristiques du système modélisé liées à la sûreté de fonctionnement [15]. Pour ça ils définissent un sous-langage qui peut être utilisé pour déclarer des modèles d'erreur et les associer aux composants de la spécification de l'architecture.

Dans ses travaux [14], Rugina propose le guidage de l'élaboration des modèles AADL de sûreté de fonctionnement par une méthode itérative, qui prend en compte progressivement les dépendances entre les composants. Pour ce faire, toutes les primitives du langage AADL qui servent à la description des dépendances liées à la sûreté de fonctionnement ont été identifiées, et des règles de modélisation pour chaque type de dépendance ont été définies. Dans son approche elle vise la construction d'un modèle d'architecture en utilisant AADL, ainsi que la construction de modèles d'erreur associés à chaque composant du modèle architectural. La compositions des modèles d'erreur de chaque composant forment le modèle d'erreur du système.

Cette approche est itérative, *c-a-d* a chaque itération des nouvelles information sont ajoutées. Dans un premier temps chaque composant du système est décrit, en introduisant leur comportement face aux pannes. Ensuite, des dépendances de plusieurs types sont définies entre les composants. Les vérifications de SdF sont faites au niveau de ces dépendances. Dans notre travail, nous utilisons une approche dans laquelle l'architecture est séparée de l'analyse SdF (nous avons une vue architecturale du système, ainsi qu'un point de vue SdF). Les vérifications sont effectuées d'une part au moment de l'analyse SdF, et d'une autre au moment de la vérification de l'architecture. Ceci implique une connaissance minimale de l'architecture au moment de l'analyse sûreté de fonctionnement, sans, pour autant, avoir besoin que l'architecture soit complètement définie.

Dans toutes ces solutions les propriétés de sûreté de fonctionnement sont vérifiées sur des modèles et/ou éléments d'architecture complètement définis. La définition du comportement des différents éléments du langage est souvent nécessaire. Ainsi, des analyses très précises sont effectuées au détriment de la complexité des calculs. La principale différence entre les approches présentées et la notre réside dans l'aspect complexité. Dans notre approche le but n'est pas de fournir une solution précise, mais de vérifier si une solution est possible ou pas, de façon à aider le processus d'analyse de sûreté de fonctionnement. Notre analyse peut être effectuée très tôt dans le développement, vu que seulement une connaissance de base de l'architecture est nécessaire.

5. CONCLUSIONS ET PERSPECTIVES

Avec ce travail, nous proposons une méthodologie pour vérifier des exigences de sûreté de fonctionnement, notamment de tolérance aux pannes, dans des architectures de systèmes. Pour ce faire, nous définissons deux vues pour chaque exigence : une vue sûreté de fonctionnement et une vue architecturale. En faisant certaines vérifications au niveau de chaque vue, il est possible de déterminer si une architecture satisfait ou pas une certaine exigence.

Ce travail est encore en cours de développement, et les améliorations à apporter prochainement sont nombreuses.

Un des premiers points à faire évoluer est l'inclusion de pannes au niveau des connecteurs. Pour le moment nous considérons que seuls les composants peuvent introduire des pannes dans le système. Cela nous a permis de définir une sémantique préliminaire de propagation des pannes pour chaque connecteur défini, et de tester ainsi notre méthode. En réalité, les connecteurs ont eux aussi un comportement, et ce comportement peut compromettre le système aussi bien qu'un composant. Pour inclure l'introduction de pannes par les connecteurs il faudra changer leur sémantique pour que la propagation des pannes considère aussi des pannes internes.

Pour le moment nous ne proposons que deux stratégies de tolérance aux pannes pour construire la vue sûreté de fonctionnement. Cette bibliothèque de stratégies devrait être augmentée pour pouvoir répondre à plus d'exigences. Nous envisageons d'étudier les stratégies pertinentes pour tolérer la corruption des données telles que l'architecture COMMAND-MONITORING ou la triplication avec vote. Une vérification plus formelle des hypothèses associées à chaque stratégie et

vérifiées par chaque zone est prévue.

Nous envisageons aussi la possibilité d'effectuer des simulations basées sur une sémantique de propagation de fautes attribuée à chaque connecteur. Il serait, ainsi, possible de déterminer les composants affectés par une panne à un endroit précis du système. Une autre sémantique envisagée est une sémantique AltaRica pour les éléments de notre ADL, ce qui permettrait l'utilisation des outils existants pour des modèles AltaRica.

Un autre ajout à ce travail sera le développement d'un outil qui permet la définition d'architectures en utilisant l'ADL proposé dans cet article, et qui permet de créer les vues SdF et architecturales, ainsi que la vérification des différentes propriétés sur les deux vues. Nous envisageons aussi la possibilité d'effectuer des simulations de pannes en utilisant notre outil. Ces simulations seraient basées sur une sémantique de propagation de fautes attribuée à chaque connecteur. Il serait, ainsi, possible de déterminer les composants affectés par une panne à un endroit précis du système.

6. REFERENCES

[1] SysML Specification. http ://www.sysml.org/, Janvier 2012.

[2] André Arnold, Gérald Point, Alain Griffault, and Antoine Rauzy. The AltaRica Formalism for Describing Concurrent Systems. *Fundam. Inform.*, 40(2–3) :109–124, 2000.

[3] Julien Brunel. New Platform Concept - Modélisation AltaRica d'un modèle PAM et Utilisation de Motifs d'Architecture. Technical Report RT-DTIM 2/14050, Onera - The French Aerospace Lab, Juillet 2009.

[4] D. Chen, M. Törngren, and L. H. Advancing Traffic Efficiency and Safety through Software Technology (ATTESST). Deliverable D.2.2.1 - Elicitation of Representative and Relevant Analysis and V&V Techniques ATESST, 2007.

[5] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. EAST-ADL - An Architecture Description Language - Validation and Verification Aspects. In P. Dissaux, M. Filali, P. Michel, and F. Vernadat, editors, *Architecture Description Language*, page 15 p. Kluwer Academic Publishers, 2004. Contribution à un ouvrage.

[6] EASIS Consortium. Electronic Architecture and System Engineering for Integrated Safety Systems. Technical report, Information Society Technologies, 2004.

[7] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL) : An Introduction. Technical report, Software Engineering Institute, 2006.

[8] Garlan, David and Monroe, Robert T. and Wile, David. Acme : Architectural Description of Component-Based Systems. In Gary T. Leavens and Murali Sitaraman, editor, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, New York, NY, USA, 2000.

[9] Christophe Kehren. *Motifs formels d'architecture de systèmes pour la sûreté de fonctionnement*. PhD thesis, École Nationale Supérieure de lAéronautique et de l'Espace, Décembre 2005.

[10] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Trans. Software Eng.*, 21(9) :717–734, 1995.

[11] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[12] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. In *In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.

[13] Gérald Point. *AltaRica : Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement*. PhD thesis, LaBRI, Université Bordeaux I, Janvier 2000.

[14] Ana-Elena Rugina. *Modélisation et évaluation de la sûreté de Fonctionnement - De AADL vers les réseaux de Petri Stochastiques*. PhD thesis, Institut National Polytechnique de Toulouse, Novembre 2007.

[15] SAE-AS5506/1. *Annex E : Error Model Annex*. SAE Architecture Analysis and Design Language (AADL) Annex Volume 1, 2006.

[16] Laurent Sagaspe. *Allocation sûre dans les systèmes aéronautiques : Modélisation, Vérification et Génération*. PhD thesis, Université Bordeaux I, Décembre 2008.

[17] Neil Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1996.

[18] Neil Storey. Design for Safety. In *Towards System Safety : Proc. 7th Safety-Critical Systems Symposium*, pages 1–25, 1999.