# Modèles formels pour l'évaluation de la sûreté de fonctionnement des architectures logicielles d'avionique modulaire intégrée.

Charles Castel, Christel Seguin
ONERA-CERT, 2 av. E. Belin, B.P. 4025, 31055 Toulouse Cedex
{castel, seguin}@cert.fr

***Résumé***

*Le langage Altarica a été conçu pour spécifier formellement le comportement de systèmes en présence de fautes et exploiter ces spécifications à l'aide d'outils complémentaires : simulateur symbolique, model-checker, générateur d'arbre de défaillance, ... Nous proposons une méthodologie pour construire des modèles Altarica des architectures logicielles des systèmes avioniques et exploiter de tels modèles lors des analyses de sûreté de fonctionnement. Cette méthodologie aborde plus particulièrement les points suivants : comment combiner les informations sur le comportement nominal avec les analyses de mode de défaillance pour construire un modèle Altarica ? Quelles exigences peuvent être évaluées sur ce type de modèle ? En suivant quelles procédures ?Cette méthodologie a été expérimentée sur un système embarqué significatif.*

## 1 Introduction

Le projet ONERA «PRISME Avionique Nouvelle » s'intéresse à l'avionique modulaire intégrée (AMI) et étudie ce nouveau type d'architectures de systèmes embarqués sous trois points de vue: fonctionnel, performance temps réel et sûreté de fonctionnement. D'une part, il étudie comment organiser et exploiter aux mieux les informations produites lors de la réalisation d'un système AMI : cahiers des charges, spécifications, résultats d'évaluation, … pour chacun des trois points de vue. D'autre part, il se propose d'identifier les modèles et outils assistant le développement d'un système AMI, selon chacun des trois points de vue. Dans cet article, nous nous focalisons sur les travaux réalisés pour assister l'évaluation des architectures logicielles du point de vue de la sûreté de fonctionnement.

Les processus actuels d'évaluation de la sûreté de fonctionnement des systèmes avions comprennent de nombreuses analyses, conduites dès les phases les plus amont de développement des systèmes. Par exemple, l'ARP 4754 (recommandation internationale) recommande de mener trois types d'analyse, la FHA, PSSA et SSA à différents niveaux de détail (avion, systèmes, équipements). La FHA (functional hazard analysis) est réalisée pour l'ensemble des fonctions apparaissant au niveau de détail considéré. Elle identifie tout d'abord des scénarii de panne (un contexte de fonctionnement, un mode de défaillance d'une fonction et l'effet potentiel de ce mode de défaillance dans le contexte), leur gravité et des exigences que la fonction devra satisfaire pour rendre acceptable le risque d'occurrence de chaque scénario. Les exigences peuvent être quantitatives : la probabilité d'occurrence du scénario doit être inférieur à un seuil jugé acceptable, la fiabilité d'un composant doit être supérieure à un seuil… Mais l'on trouve aussi des exigences qualitatives : un contexte de fonctionnement doit être interdit, un mode de défaillance doit être masqué, … de manière à ce que le scénario ne puisse pas se produire, l'architecture du système a de bonnes propriétés (absence de point de panne unique, ségrégation, …). La tenue de telles exigences par une architecture de conception est ensuite évaluée lors de la PSSA (Preliminary System Safety Analysis). La PSSA s'appuie sur des hypothèses ou des propriétés attendues des composants qui interviennent dans l'architecture de conception ; la SSA (Safety System Analysis) a pour but d'évaluer si l'architecture effectivement réalisée satisfait les hypothèses ou propriétés supposées.

Pour mener à bien ces évaluations, les analystes réalisent différents modèles des systèmes. Certains modèles, comportementaux, sont dédiés à des simulations plus ou moins fines des systèmes. Il s'agit d'injecter des fautes et de tester si les mécanismes prévus pour tolérer ces fautes fonctionnent effectivement. Des modèles stochastiques peuvent être utilisés pour évaluer la fiabilité intrinsèque d'un système. Les arbres de défaillances sont d'autres modèles, permettant d'évaluer la probabilité d'occurrence d'un événement redouté (la racine de l'arbre), à partir des probabilités d'occurrence des événement feuilles de l'arbre, les feuilles étant combinées à l'aide d'arc « et » et « ou ».

L'ensemble de ces modèles contribue à l'évaluation de l'ensemble des exigences de sûreté de fonctionnement des systèmes. Cependant, la cohérence de ces modèles n'est pas établie ; leur multiplicité est coûteuse et laisse encore la place à beaucoup d'implicite. Le langage Altarica a été conçu au LaBri pour spécifier formellement le comportement de systèmes en présence de fautes et exploiter une unique spécification à l'aide d'outils complémentaires : simulateur symbolique, model-checker, générateur d'arbre de défaillance, simulation stochastique. Le simulateur permet d'animer des spécifications comportementales non déterministes. Des traducteurs de Altarica vers les automates, les formules booléennes et les réseaux de Petri stochastiques ont été conçus et pour certains développés afin de permettre l'utilisation respectivement du model-checker Mec, de l'outil de manipulation d'arbres Aralia et du simulateur stochastique de Fiabex. L'utilisation d'un tel langage et des outils associés semble prometteuse mais nécessite une réflexion méthodologique pour pouvoir utiliser l'approche en contexte industriel. Comment doit on construire un unique modèle Altarica pour pouvoir l'exploiter de manière probante avec l'ensemble des outils ? Généralement, les arbres de défaillances construits à la main référent des événements de haut niveau, alors que les simulations sont fines ! Comment peut-on utiliser les éléments d'analyse fonctionnelle et dysfonctionnelle disponibles pour construire les modèles ? De plus, la preuve de propriétés n'est pas une pratique acquise dans ce contexte: quelles exigences de sûreté pourront être évaluées à l'aide d'un model-checker ? Enfin, les modèles uniques supportant autant d'analyses différentes sont eux-même critiques et doivent être soigneusement validés.

Dans le cadre du projet PRISME, nous avons recherché une méthodologie d'utilisation d'Altarica permettant d'évaluer l'architecture logicielle d'un système AMI. Nous avons plus particulièrement recherché une méthodologie de construction de modèles Altarica abstraits. En effet, l'analyse de sûreté de fonctionnement débute très tôt dans le cycle de développement du système AMI pour être affinée au fur et à mesure que les choix de conceptions se précisent. Ainsi, elle peut conduire à raisonner aussi bien sur des composants logiciels inexistants au début du projet que sur des composants que l'on souhaite réutiliser. Pour mener une analyse cohérente de l'ensemble de l'architecture, nous proposons d'en réaliser un modèle formel abstrait, rassemblant de manière synthétique les caractéristiques de ses différents composants. D'autre part, on notera que les architectures logicielles des systèmes AMI peuvent être complexes. Pour maîtriser la complexité de la tâche de modélisation, le processus de modélisation doit être le plus incrémental possible. Nous proposons de construire les modèles de manière compositionelle: un modèle de comportement défaillant sera défini pour chaque fonction logicielle puis les modèles élémentaires seront assemblés pour construire le modèle complet de l'architecture. Enfin, à partir de ces modèles, nous cherchons à évaluer différentes exigences quantitatives et qualitatives. Dans cet article, nous présentons uniquement des exigences qualitatives et évaluons, par simulation symbolique et model-checking des modèles, la capacité d'une architecture à tolérer et confiner les fautes .

L'article est structuré de la manière suivante. Dans une première partie, nous discutons des informations à abstraire des analyses de sûreté de fonctionnement et des spécifications fonctionnelles pour construire des modèles de propagation des pannes dans les architectures logicielles AMI. Dans une seconde partie, nous présentons le langage formel Altarica et montrons comment utiliser les informations retenues pour réaliser le modèle Altarica d'une architecture logicielle. Dans une troisième partie, nous montrons comment exploiter les modèles Altarica pour évaluer l'effectivité des mécanismes de confinement des pannes par simulation et par model-checking. La méthodologie proposée a été appliquée à un sous-système avion ; nous extrayons de ce cas d'étude les exemples présentés dans les trois parties. Pour conclure, nous comparons notre méthodologie à celles qui se dégagent de quelques travaux.

# 2 Caractérisation des architectures logicielles des systèmes AMI pour la sûreté de fonctionnement

Pour construire des modèles de propagation des pannes dans une architecture logicielle, nous examinons deux types d'informations.

- Celles extraites des spécifications fonctionnelles. D'une part elles permettent d'identifier précisément les composants qui interviennent dans l'architecture, leurs attributs de bon fonctionnement. D'autres part, elles définissent les canaux de propagation normaux dans

l'architecture i.e. les dépendances de données au sein d'un même code et les échanges de données entre les logiciels, ainsi que les moyens mis en œuvre pour détecter ou confiner les erreurs prévues et contrôler ainsi la sécurité des échanges.

- Celles extraites des analyses des modes de défaillances. Ces informations sont les duales des précédentes : elles identifient les défaillances qui peuvent affecter individuellement les composants de l'architecture ainsi que leurs effets (les modes de défaillances) sur le composant.

Les modèles de propagation de pannes vont intégrer les deux types d'informations afin d'étudier les déviations induites sur l'ensemble du système par des défaillances locales.

## *2.1 Les informations nominales*

Pour illustrer notre démarche, nous considérons un sous-ensemble de systèmes AMI, les systèmes temps réel critiques périodiques. Examinons les caractéristiques de leur architecture logicielle. Tout d'abord, ces systèmes sont composés d'un ensemble de *tâches* périodiques qui doivent produire des *données fraîches* régulièrement. D'autre part, un même logiciel peut jouer différents rôles selon les phases de vol de l'avion ou l'état courant des ressources informatiques disponibles ; ces rôles sont gérés par des *modes de fonctionnement*. De plus, certaines tâches effectuent non seulement des *calculs* mais contiennent aussi des mécanismes de confinement des erreurs ; des *tests* détectent des erreurs et du code « *contre-mesure* » fournit des données sécurisées lorsqu'une erreur est détectée.
A titre d'exemple, nous présentons la structure d'une tâche « anémomètre »

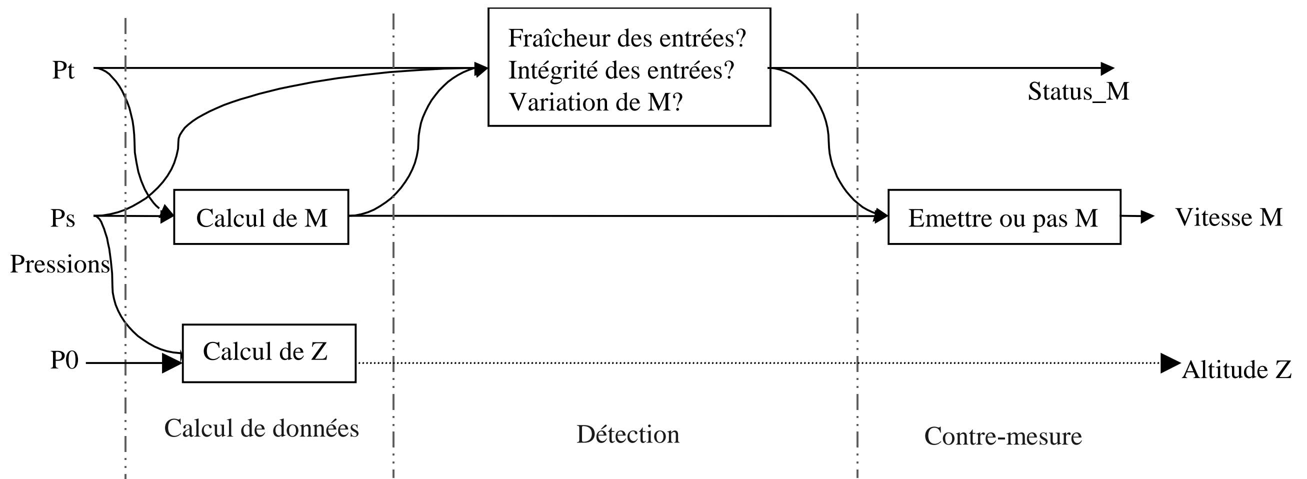


**Figure 1 : Structure d'une tâche anémomètre**

Cette tâche calcule la vitesse M et l'altitude Z de l'avion en fonction de pressions (Ps statique, Pt dynamique et P0 atmosphérique au niveau de la mer). La figure 1 indique les dépendances entre entrées et sorties. Le calcul de M dépend de Pt et Ps alors que celui de Z dépend uniquement de Ps et P0. Pour sécuriser le calcul de M, les moyens de détection mis en œuvre évaluent la fraîcheur des entrées Pt et Ps ainsi que leur plausibilité en vérifiant d'une part que les entrées sont comprises dans une certaine fourchette et d'autre part que la variation de vitesse est aussi plausible. En cas d'erreur potentielle, on choisit de ne pas émettre la vitesse calculée et l'information est communiquée au reste du système via un booléen Status_M indiquant la qualité présumée de M.
Tests et contre-mesures peuvent être purement internes à une tâche mais bien souvent, ils tirent parti de l'architecture globale. Par exemple, certains calculs peuvent être *dupliqués* afin de détecter ou masquer des défaillances, comme indiqué sur la figure 2. Dans cette architecture classique en avionique, les calculs sont dupliqués sur une voie de commande et une voie de monitoring. Les deux voies échangent et comparent leurs résultats respectifs. En cas de désaccord sur les résultats échangés, une erreur potentielle est détectée et est traitée afin de produire une sortie moins dangereuse pour le système.

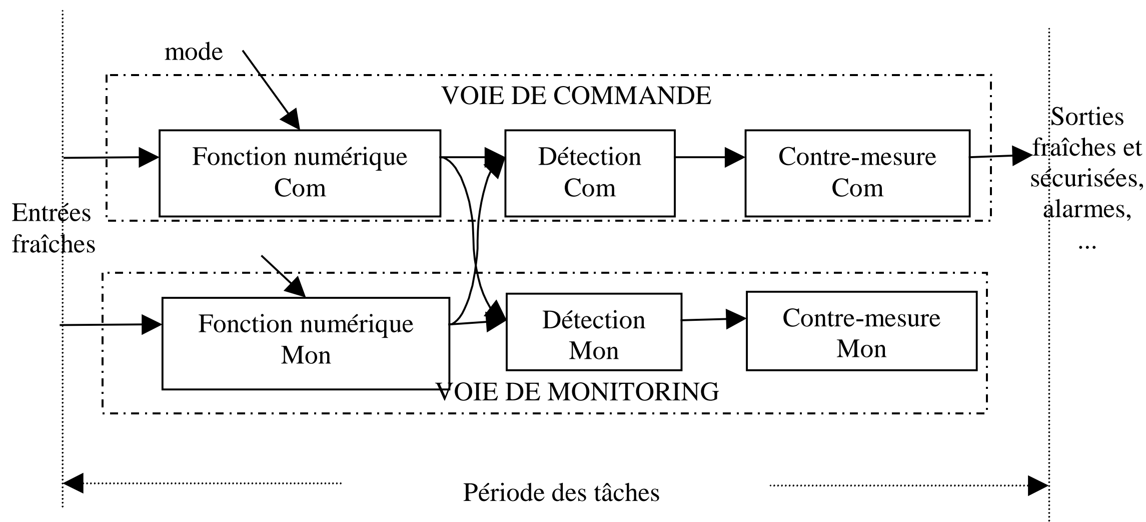


**Figure 2 : Exemple d'architecture logicielle pour un système critique temps réel**

## *2.2 Les informations extraites des analyses de sûreté de fonctionnement*

Les tâches composant l'architecture logicielle réalisent des fonctions dont les défaillances sont caractérisées dans les analyses des modes de défaillance. Pour analyser l'architecture logicielle avec un certain niveau d'abstraction, les scénarios de défaillances suivants sont considérés.

| Défaillances | Effets (modes) des défaillances |
|---|---|
| Défaillances externes traduites par la qualité des entrées<br>• Une entrée est absente, le logiciel n'est pas protégé.<br>• Une entrée est incorrecte, le logiciel n'est pas protégé. | • Les résultats des calculs dépendant de cette entrée peuvent être incorrects.<br>• Les résultats des calculs dépendant de cette entrée peuvent être incorrects. |
| Défaillances induites par une erreur de mode.<br>• Le mode d'utilisation de la fonction n'est pas cohérent avec la phase opérationnelle effective. | • Les résultats des calculs dépendant du mode peuvent être incorrects. |
| Défaillance de la fonction<br>• La fonction est perdue, aucun calcul n'est effectué.<br>• La partie « calcul de données » est erronée. | • Aucune sortie n'est produite.<br>• Les sorties sont produites mais sont incorrectes quelle que soit la qualité des valeurs en entrées. |
| Défaillance de la détection<br>• La détection est intempestive.<br>• La détection est inefficace ou perdue. | • Quelle que soit la situation, une erreur est signalée.<br>• Quelle que soit la situation, aucune erreur n'est signalée. |

**Figure 3 : Les défaillances d'une architecture logicielle et leurs effets**

Cette liste de scénarios ne vise pas l'exhaustivité mais constitue plutôt une base minimale dont on peut s'inspirer pour analyser les tâches périodiques d'une architecture logicielle de type AMI.

On notera que l'analyse des défaillances d'un unique composant peut être relativement simple à mener. Par contre, il est plus difficile de prévoir comment ces défaillances primaires se propagent au sein du système. Pour assister l'analyse de la propagation, nous proposons de réaliser des modèles Altarica de l'architecture complète et d'exploiter ces modèles à l'aide des outils formels disponibles.

# 3 Modèles Altarica d'une architecture logicielle AMI

Altarica est un langage conçu au Labri pour modéliser formellement les systèmes et leur défaillance. Son expressivité permet non seulement de décrire un large spectre de système mais aussi d'exploiter ces modèles par des outils complémentaires : simulateur symbolique, générateur d'arbre de défaillances et model checker.

Le langage Altarica est hiérarchisé et compositionnel. Un modèle de système est un nœud principal qui est composé de sous-nœuds. Chaque sous-nœud peut à son tour contenir des sous-nœuds. Un nœud (**`node`**) possède des entrées/sorties (**`flow`**) et des variables d'état internes (**`state`**). Les variables internes mémorisent des états atteints après l'occurrence d'événements externes (**`event`**). Les évolutions possibles des états internes sont décrits par un ensemble de transitions (**`trans`**). Les liens entre variables observables et internes sont décrits à l'aide d'assertions booléennes (**`assert`**).

La structure syntaxique d'un nœud simple se résume de la manière suivante :

```
node anemo

flow
      pt, ps, m, … : data;
      all_m_in_correct, … : boolean ;

state
      anemo_state: {correct_state, wrong_computation …} ;

event
      computation_error, …;

/* Relations entre flots et/ou etats internes */
assert
      all_m_in_correct= (pt=correct & ps=correct);
      anemo_nominal_comp= ~(anemo_state=wrong_computation |
                              anemo_state=no_refresh);

/* Evolution des états internes */
trans
      anemo_state=correct_state |-computation_error->
                              anemo_state:=wrong_computation;

edon
```

**Figure 4 : Structure d'un noeud Altarica[1]**

Les sous-nœuds échangent des flux d'entrées/sorties ; leur interconnexion est effectuée dans le nœud principal par des assertions particulières, indiquant que les sorties d'un nœud sont les entrées d'autres nœuds. D'autre part, des événements externes peuvent affecter simultanément plusieurs sous-nœuds. On modélise ceci à l'aide de vecteurs de synchronisation, identifiant les événements concernés dans les sous-nœuds.

Le lecteur intéressé pourra trouver plus de détails dans [Arnold-al00]. Dans la suite, nous mettons l'accent sur la démarche suivie pour construire un modèle Altarica d'une architecture logicielle AMI et donnons un bref aperçu du langage à travers la façon dont nous l'utilisons dans le projet.

## *3.1 Domaines abstraits*

Une étape importante dans la construction des modèles est l'identification des domaines de valeurs des variables observables. Nous proposons de raisonner sur la qualité des données ou sur de larges classes de valeurs plutôt que sur les valeurs effectives. Ainsi, nous utiliserons des domaines de calculs abstraits.

Les valeurs abstraites pertinentes sont définies en examinant :

[1] &, |, ~ dénotent respectivement les connecteurs booléens « et », « ou », « non »

1. les qualités des données perçues par les détections dans la spécification fonctionnelle. Par exemple, dans le cas de la tâche anémomètre citée plus haut, des tests sont prévus pour repérer les données non plausibles ou manquantes.
2. Les qualités des données après défaillances.

Par exemple, dans le cas de la tâche anémomètre, la tâche a trois entrées, les pressions `p0`, `ps`, `pt` de type réel, deux sorties réelles, la vitesse `m` et l'altitude `z` et deux sorties booléennes `status_m` et `status_z`, indiquant si les sorties numériques sont valides ou pas. Si nous nous référons au tableau de défaillances et aux tests prévus dans la tâche présentée plus haut, les valeurs abstraites d'une donnée numérique seront `correct` ou `incorrect` ou `missing`. Les types énumérés de ce type peuvent être déclarés en Altarica.

```
domain
      data= {correct, incorrect, missing};
```

**Figure 5 : Déclaration de domaine en Altarica**

Pour les statuts, nous conservons le domaine booléen initial, qui est suffisant pour l'analyse, et prédéfini en Altarica.

### *3.2 Etats internes et événements*

Puis nous recherchons les éléments qui interviennent dans le contrôle des tâches.

1. Dans la spécification fonctionnelle, il s'agit des modes d'utilisation de la fonction et des événements qui déclenchent le passage à un mode (ordre du pilote de passer en mode automatique par exemple). Ainsi, nous distinguons deux concepts : le mode (ou état interne de la fonction) et l'événement externe, perceptible qui induit l'état. Ceci correspond exactement aux concepts d'états et d'événements prédéfinis dans Altarica.
2. Les deux concepts apparaissent aussi dans les analyses de sûreté de fonctionnement : les défaillances sont des événements externes qui induisent des modes de fonctionnement défaillants.

L'anémomètre admet un seul mode de fonctionnement nominal `correct_state`. Par contre, cette tâche réalise des calculs numériques et des détections et les défaillances peuvent affecter les deux types de calcul. Nous retiendrons donc comme états défaillants, les états `correct_state`, `wrong_computation`, `no_refresh`, `no_detection`, `wrong_alarm`, résultant respectivement des événements `computation_error`, `refresh_loss`, `detection_loss`, et `wrong_detection`.

### *3.3 Dynamique des états internes*

La dynamique des états internes dépend des contraintes applicatives pour ce qui concerne les logiques de changement de mode opérationnel et des hypothèses sur les pannes pour ce qui touche aux modes de défaillance. Par exemple, pour modéliser qu'un composant ne peut être affecté que par une panne unique à un instant et que les pannes sont permanentes, nous utiliserons l'automate suivant.

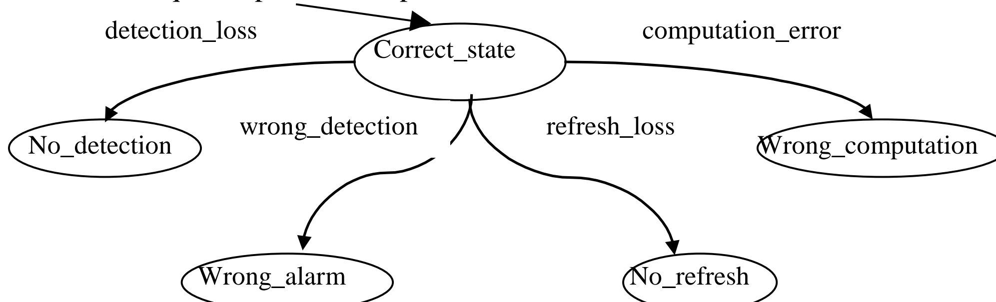


**Figure 6 : Dynamique des modes de défaillance de la fonction « anémo »**

Les défaillances ne peuvent avoir un effet (changement de mode de défaillance) que si elles se produisent dans l'état correct. De plus, le nouvel état défaillant est permanent car aucun arc ne part des états. Tout se passe donc comme si une seule panne permanente pouvait affecter le composant. Cet automate est construit en Altarica en décrivant chacune des transitions (voir par exemple la déclaration

de la transition faisant passer de l'état `correct_state` à l'état `wrong_computation` à la rubrique `trans` dans la figure 4).

### 3.4 Calculs abstraits

Pour finir, il nous reste à décrire les calculs, détections et contre-mesures abstraits en tenant compte des modes opérationnels et de défaillance courants. Nous avons choisi de spécifier ces calculs abstraits à l'aide de tables de décision, en s'inspirant de [Parnas 92]. Une table est construite pour chaque flux de sortie du nœud et une ligne de la table se présente de la manière suivante :
« si le mode=x et les valeurs des flux en entrées satisfont la contrainte c, alors la valeur du flux de sortie est y ».
Cela se code naturellement en Altarica à l'aide d'assertions booléennes du type `a=>b` (si `a` alors `b`, équivalent à `~a|b`). Ainsi, l'assertion de la figure 7 se lit : si le mode de défaillance courant est nominal pour ce qui relève des calculs et toutes les entrées de m sont correctes alors m est correct. (Les expressions booléennes `anemo_nominal_comp` et `all_m_in_correct` sont définies dans la figure 4).

```
(anemo_nominal_comp & status_m & all_m_in_correct) => m=correct;
```

**Figure 7 : Assertion Altarica décrivant le calcul de m.**

En utilisant l'ensemble de ces principes, nous obtenons pour l'anémomètre la spécification Altarica présentée en annexe.

### 3.5 Assemblage des noeuds

Ces principes guident la réalisation des nœuds Altarica décrivant chacun des composants atomiques du système (une tâche). Puis l'ensemble des nœuds sont interconnectés pour construire le modèle Altarica de l'architecture complète. La connexion est établie simplement en mettant en correspondance les sorties d'un nœud avec les entrées d'un autre. La seule contrainte à respecter est de s'assurer que les flux connectés ont le même type. Ceci est possible en suivant notre méthodologie car nous avons pris le soin d'identifier les valeurs abstraites pertinentes lors des analyses préliminaires de sûreté de fonctionnement. Ainsi, les domaines de valeurs pertinents pour l'ensemble du système sont prévus avant de connecter les nœuds.
Les flux sont les principaux vecteurs de propagation de pannes. Nous utilisons aussi la synchronisation d'événement pré-définie dans Altarica pour indiquer les pannes communes qui peuvent affecter simultanément deux composants.

## 4 Exploitation d'un modèle Altarica d'architecture logicielle AMI

Nous allons à présent montrer comment ces modèles formels sont exploités. Nous expliquons tout d'abord les éléments de sémantique utilisés pour simuler les modèles Altarica. Puis nous présentons les vérifications utilisées pour valider nos modèles. Enfin nous montrons comment les résultats d'une simulation ou d'une preuve contribuent à l'évaluation d'exigences de sûreté.

### 4.1 Principes de simulation de modèles Altarica

La simulation d'un modèle Altarica permet d'observer comment évolue l'ensemble des flux ou états internes lorsqu'un événement externe se produit.
Nous appelons *état global* du système le vecteur des valeurs affectées aux états internes des composants du système et aux flux échangés par les composants. Les états globaux *admissibles* doivent satisfaire l'ensemble des assertions figurant dans le modèle Altarica.
Une transition de la forme `garde(f1, …, fl, s1, …, sk)|-evt-> sj := vj;` est *franchissable* dans un état global quand l'événement `evt` est présent et l'état global satisfait la condition `garde` portant sur les flux `f1, …, fl` et les états internes `s1, …, sk`. Les événements sont supposés être asynchrones (un seul événement à la fois se produit) à moins d'être explicitement synchronisés par des vecteurs de synchronisation.

Les modèles Altarica peuvent être non-déterministes : dans un état global, le franchissement d'une transition peut conduire à plusieurs autres états globaux successeurs ou à … aucun. En effet, les successeurs potentiels sont construits de la manière suivante. Une nouvelle valeur `vj` est affectée à l'état interne `sj` alors que les valeurs des autres états internes restent inchangées. Par contre les valeurs des flux peuvent changer librement, dans la mesure où leurs nouvelles valeurs et celles des états internes demeurent compatibles avec l'ensemble des assertions.
Les états globaux atteints après franchissement d'une séquence de transitions à partir d'un *état initial* sont appelés *états atteignables*.

Ces principes de simulation fixent la sémantique opérationnelle des modèles Altarica et l'évaluation de modèles Altarica se traduit par une exploration des états atteignables. L'exploration peut se faire selon différentes stratégies. Nous appellerons « preuve » les explorations exhaustives (réalisées par model-checking ou démonstration) par opposition aux « tests » qui ne garantissent pas l'exhaustivité.
Les modèles Altarica construits en suivant la démarche proposée sont évalués en trois temps :

1. Validation syntaxique (contrôle de la syntaxe et du typage)
2. Evaluation de la correction intrinsèque des modèles. Il s'agit de valider les modèles en prouvant qu'ils satisfont bien des propriétés indépendantes de l'application modélisée.
3. Evaluation d'exigences de sécurité.

Nous détaillons seulement les deux derniers points.

### *4.2 Critères de correction des modèles*

Nous testons tout d'abord, conformément à l'approche de Parnas, la cohérence et la complétude des tables de décision contenues dans chacun des nœuds. Une table de décision est complète lorsque l'ensemble des conditions envisagées dans chaque ligne couvre tous les états globaux possibles. Rappelons que la table de décision relative à l'évolution d'un flux est une assertion booléenne du type:

```
((cond1(f1, …, fn, s1, …, sm)=>val1(fj)) &
((cond2(f1, …, fn, s1, …, sm)=>val2(fj)) &
…
((condk(f1, …, fn, s1, …, sm)=>valk(fj)) ;
```

où `f1, …, fn, s1, …, sm` sont des flux considérés comme des entrées, `fj` est un flux de sortie, `s1, …, sm` sont des états internes. La formule `condi(f1, …, fn, s1, …, sm)` exprime une condition sur les valeurs des flux d'entrée et des états internes. La formule `vali(fj)` caractérisant les valeurs admises pour `fj` sous la condition `condi(f1, …, fn, s1, …, sm)`.
La table de décision est complète si pour toutes les valeurs que peuvent prendre les flux d'entrée et les états internes, au moins l'une des conditions est satisfaite. Cela revient à vérifier que la formule

```
cond1(f1, …, fn, s1, …, sm)|… |condk(f1, …, fn, s1, …, sm);
```

est toujours vraie. Etant donné que les flux utilisés sont booléens ou prennent leur valeurs dans des domaines de valeurs finis, la validité de cette formule peut être évaluée automatiquement par un model-checker.

Une seconde vérification porte sur la cohérence de chaque table. Lorsque les conditions `condi` ne sont pas disjointes, deux lignes d'une table peuvent proposer d'affecter deux valeurs distinctes à un même flux. Pour éviter ce problème, il suffit de prouver que les paires de conditions sont deux à deux disjointes. Comme dans le cas précédent, cette preuve peut être automatisée, en évaluant par exemple la validité de formules telles que :

```
~condi(f1, …, fn, s1, …, sn) | ~condj(f1, …, fn, s1, …, sn)
```

Enfin, de manière générique nous vérifions que, en l'absence de pannes et d'entrées incorrectes, toutes les sorties sont correctes. La vérification porte d'abord sur les composants atomiques, i.e. les nœuds Altarica de base, puis sur l'architecture complète, i.e. sur le nœud Altarica connectant l'ensemble des nœuds de base. Ceci permet principalement de détecter des erreurs typographiques dans les nœuds et des problèmes de connexions inter nœuds.
Etant donnée la structure des modèles Altarica utilisés, nous avons choisi d'effectuer cette vérification automatiquement à l'aide d'un model-checker (et pas par simulation). Nous procédons de la manière suivante.

Nous construisons un nœud « environnement sûr » qui produit uniquement des données correctes et dont l'état interne peut prendre deux valeurs : `not_faulty` et `faulty`. Initialement, l'état interne est `not_faulty`. Une transition est prévue pour passer de l'état sain `not_faulty` à l'état défaillant `faulty` lorsqu'un événement `fault` se produit. Pour bloquer le tir de cette transition, il suffit d'indiquer que la garde de la transition est la condition `false`, qui n'est jamais vérifiée.
Puis nous connectons les flux de sortie de cet environnement aux flux d'entrée des composants les plus externes. Pour bloquer le tir des pannes des composants de l'architecture, il suffit de synchroniser l'événement `fault` de l'environnement avec les événements modélisant les défaillances des autres composants.
Enfin, le model-checker vérifie que les flux de sortie ont tous la valeur correcte pour tous les états atteignables dans cet environnement.
Le concept de nœud «environnement » est utilisé ici pour contraindre les états à explorer lors d'une preuve. Le principe est aussi applicable pour restreindre une simulation. De plus, ce concept peut être décliné de plusieurs façons pour déterminer si le modèle satisfait des exigences dépendant davantage du contexte applicatif, comme nous allons le voir dans le paragraphe suivant.

### *4.3 Evaluation de propriétés de sûreté de fonctionnement*

Des objectifs de preuve ou de tests variés contribuent à l'évaluation de la capacité d'une architecture logicielle à tolérer les erreurs. En effet, les exigences de sûreté de fonctionnement sont bien souvent explicitement formulées en terme de probabilité d'occurrence de pannes mais se déclinent plus concrètement en propriété qualitative de l'architecture. Nous nous intéressons ici à ces propriétés qualitatives. A titre d'exemples, nous proposons :

| | |
|---|---|
| Qualité des détections | Efficacité des détections : en présence d'au moins une entrée non correcte et de x pannes, l'alarme est immédiatement donnée.<br>Absence de fausse alerte : en présence de x pannes, lorsque l'alarme est donnée, soit une entrée n'est pas correcte, soit un composant est dans un état défaillant. |
| Qualité du confinement | En présence de x pannes, aucune sortie erronée n'est jamais produite. |
| Occurrence d'événement redouté | En présence de x pannes, le système n'atteint jamais un état redouté. |

**Figure 8 : Exigences de sûreté de fonctionnement qualitative**

Lorsque x=0, cela revient à évaluer la correction du système en l'absence de panne.
Lorsque x=1, nous traitons les configurations de panne unique. Nous avons vu dans la section précédente comment modéliser l'absence de panne à l'aide d'un modèle spécifique d'environnement. Nous utilisons un procédé similaire pour modéliser qu'une panne unique peut se produire. Le modèle d'environnement est celui décrit dans la figure 9. Une défaillance ne peut se produire que dans l'état `not_faulty`. Or, dès qu'une défaillance f1, … ou fn se produit, l'état puits `faulty` est atteint et aucune autre défaillance ne pourra se produire.

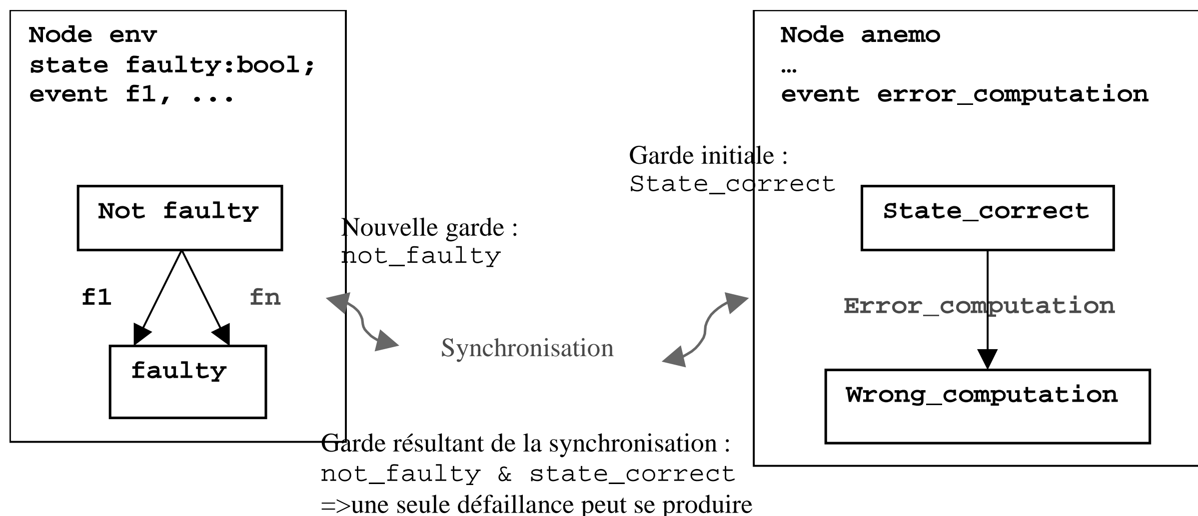


**Figure 9 : Utilisation d'un environnement pour moduler les modèles de panne**

Etant donné que les flux prennent des valeurs abstraites indiquant leur qualité, la caractérisation des sorties erronées est souvent triviale. Si l'on se réfère à l'exemple de l'anémomètre, il s'agit de sorties dont la valeur est `incorrect`. Les tests de nos modèles sont très simples à dépouiller et il est même possible de prouver automatiquement qu'un modèle d'architecture satisfait ou non les exigences du tableau 8.
On notera cependant que les modèles abstraits proposés ne permettent pas de juger de la pertinence d'une fourchette de valeurs concrètes ni de la pertinence des mécanismes de détection pris un à un. Par contre, ces modèles montrent la place qu'occupe chacun des mécanismes élémentaires et permettent d'évaluer l'architecture du système.

## *4.4 Application*

Cette démarche d'évaluation a été expérimentée sur le cas d'un sous-système embarqué qui contrôle le déplacement de surfaces avion en fonction des ordres du pilote et des paramètres de vol de l'avion. Quatre composants de base sont identifiés :

- l'anémomètre présenté plus haut, qui calcule la vitesse m et altitude z de l'avion,
- le gestionnaire de vol (Fm), qui élabore un ordre de déflexion de gouverne à partir de la position xy et l'altitude z de l'avion, du plan de vol courant, de l'ordre du pilote et du mode de pilotage (automatique ou manuel),
- le pilote automatique (Pa), qui calcule l'angle de déflexion correspondant à l'ordre reçu
- le gestionnaire d'alarme (Fw) qui émet une alarme si les paramètres du calculs semblent erronés.

Les dépendances fonctionnelles entre les différents composants sont représentées plus précisément sur la figure 10. L'anémomètre, le pilote automatique et le gestionnaire d'alarme incluent chacun des tests et contre-mesure spécifiques. L'événement redouté pour ce système est de produire un angle de déflexion erroné (`incorrect` ou `missing`) et de ne pas le détecter (`alarme` vaut `false`).

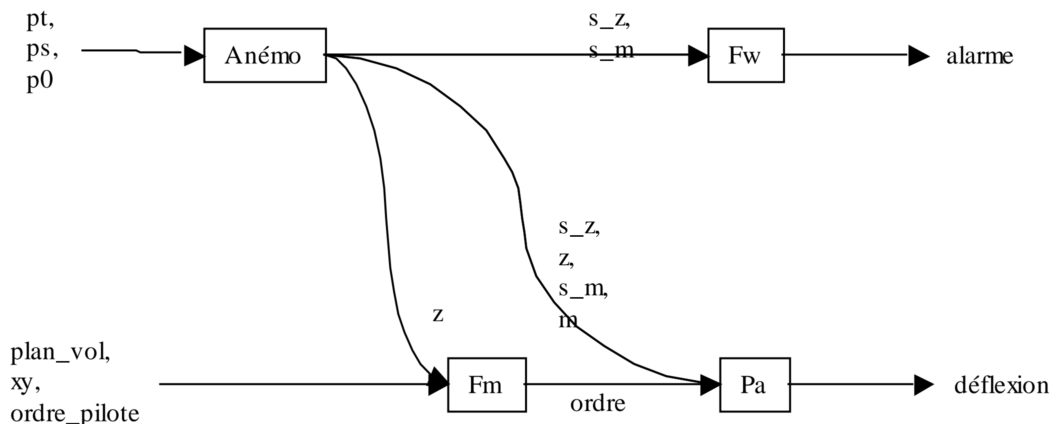


**Figure 10 : Architecture fonctionnelle simple**

Le gestionnaire de vol Fm ne présente pas de moyen de détection et la détection prévue dans le Pa analyse les statuts fournis par l'anémomètre ainsi que la fraîcheur des entrées. Sur l'architecture de la figure 10, une simple simulation montre alors que l'événement redouté peut se produire en l'absence de défaillance interne. Il suffit que certaines entrées du Fm soient fraîches mais incorrectes. Ce résultat a été aussi retrouvé en tentant de prouver que l'événement redouté ne se produisait pas en l'absence de défaillance. La preuve a échoué et le contre-exemple fourni correspond au scénario de simulation invalidant. On montre aussi qu'une unique défaillance (sur la détection de l'anémomètre ou celle du Fw ou encore sur le Pa) conduit à l'événement redouté.

Une architecture avec redondance est proposée pour remédier à ces problèmes. Les voies de calcul sont dupliquées, on dispose d'une voie de commande et d'une voie de monitoring ; les anémomètres sont tripliqués. De nouveaux mécanismes de détection sont introduits avec de nouvelles défaillances potentielles: il s'agit des voteurs sur les sorties des trois anémomètres et des comparateurs entre les résultats échangés par les voies de commande et de monitoring (voir figure 11).

Cet exemple montre bien comment la complexité de l'architecture croît dès que l'on intègre des mécanismes de sûreté de fonctionnement : ici, on passe de 4 à 16 composants ! Dans ce dernier cas, le test devient plus lourd à mettre en œuvre et nous avons évalué le modèle de cette architecture uniquement à l'aide d'un model-checker. Au moment où l'expérience a été réalisée, le traducteur de Altarica vers Mec n'était pas disponible. Aussi avons nous utilisé l'outil SMV [MacMillan99] dont le langage d'entrée permet de modéliser aisément des modèles Altarica du type de ceux qui ont été écrits. Les modèles Altarica ont été systématiquement traduits dans le langage d'entrée de SMV. Cette traduction a été immédiate car nous utilisons les assertions Altarica soit comme des tables de décisions de flux de sortie, soit comme des définitions de variables booléennes auxiliaires et que le langage de SMV permet d'écrire des définitions analogues.

Les spécifications SMV et Altarica sont très compactes : nous utilisons des modèles génériques de voteur, d'anémomètre, … et nous déclarons des instances de ces nœuds de base pour construire le modèle de l'architecture complète. Une douzaine de pages suffit pour décrire l'architecture de la figure 11 et le nœud anémo présenté en annexe donne une idée de la complexité de chacun des composants. Les performances de SMV sur ce type de modèle sont très satisfaisantes : réponse instantanée avec production de contre-modèles lisibles.

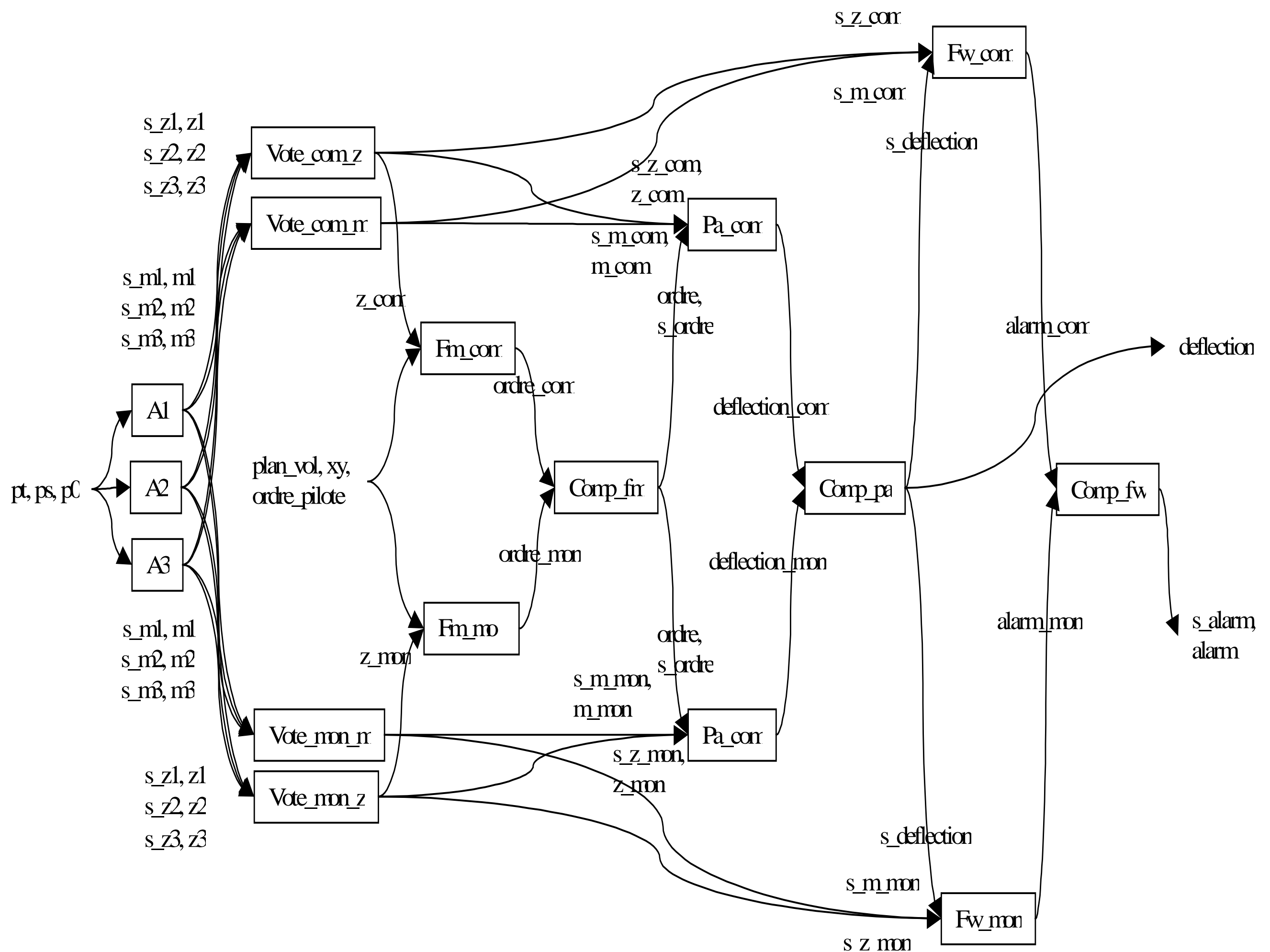


**Figure 11 : Architecture logicielle avec redondance**

# 5 Conclusion

Nous recherchions une méthodologie de modélisation des architectures logicielles des systèmes AMI en vue de l'évaluation de leurs capacités à tolérer et confiner les fautes, qui satisfasse plusieurs contraintes : possibilité de réaliser des modèles formels abstraits, compositionnels, construits en liaison avec les autres analyses de sûreté de fonctionnement et les analyses fonctionnelles.
L'expérience réalisée a permis de dégager les premiers éléments d'une telle méthodologie. Nous montrons que la modélisation formelle ne se substitue pas aux analyses de mode défaillances classiques mais tire partie des informations obtenues au cours de ces analyses amont pour construire des modèles de propagation de panne facilitant l'évaluation de la tenue d'exigence de sûreté de fonctionnement. Ainsi, la démarche proposée permet de faire un pont entre les analyses de sûreté les plus amont et les activités d'évaluation. On notera aussi que le principe de construction de modèles exploite les informations disponibles dans les spécifications fonctionnelles et que notre proposition permet de relier analyses de sûreté et fonctionnelle.

Le choix du langage Altarica nous semble très satisfaisant. Son expressivité a permis de réaliser naturellement des modèles abstraits et compositionnels. La séparation de la logique d'évolution des états internes (sous forme de transitions) et des aspects plus fonctionnels (les assertions) est très appréciable : les spécifications sont plus lisibles, des patterns de spécification apparaissent plus facilement. La notion d'assertion est particulièrement intéressante. Elle permet de construire des modèles de niveaux d'abstraction variés. Ici, les assertions décrivent des calculs abstraits déterministes (une seule valeur de sortie est prévue dans une configuration d'entrée) ou pas. Le non déterminisme a été en particulier utilisé pour modéliser grossièrement une défaillance qui conduit un voteur à produire

indifféremment deux valeurs de sortie possibles. De plus, les assertions ont permis de coder des tables de décision dont la robustesse et la cohérence ont été évaluées aisément à l'aide des outils de model-checking. Enfin, synchronisation d'événements et flux de données permettent de propager les pannes de manières variées.

L'expressivité d'Altarica est aussi suffisamment limitée pour que les modèles se prêtent en théorie à de multiples analyses automatiques (model-checking, génération d'arbres de défaillance, simulation stochastique). L'un de nos objectifs est de montrer comment ces différents outils peuvent être utilisés de manière complémentaire lors de l'évaluation de sûreté de fonctionnement des architectures logicielles. La démarche d'évaluation présentée dans cet article porte uniquement sur l'exploitation du model-checking. Son application au cas d'étude montre que ces analyses peuvent être effectivement réalisées en pratique sur des modèles abstraits du type de ceux que nous réalisons. En effet, les performances obtenues avec SMV sont très satisfaisantes et laissent penser que la génération d'arbres de défaillances pourrait être réalisable sur nos modèles abstraits. Pour l'instant, la traduction vers SMV a été effectuée manuellement mais des traducteurs vers des model-checker et des outils d'analyses d'arbre de défaillance sont en cours d'étude. Ces nouveaux outils devraient permettre d'appliquer notre démarche de model-checking à des cas d'études plus gros et d'étayer nos réflexions sur la génération d'arbres de défaillances.

A notre connaissance, les éléments de méthodologie proposés sont relativement originaux. En effet, on trouve d'une part des travaux portant sur l'utilisation de démonstrateurs de théorèmes ou de model-checker pour évaluer une architecture logicielle particulière (voir par exemple [Rushby92], [Pagani-al95]). Mais ces applications d'outils de preuve ne dégagent pas une méthodologie de modélisation et d'évaluation pouvant s'appliquer à plusieurs cas. D'autre part, on trouve des travaux portant sur la modélisation de systèmes logiciels en vue de la génération d'arbre de défaillance. Par exemple, les auteurs de [Fenelon-McDermid92][McDermid-Pumfrey94] ont le même souci de produire des modèles abstraits et compositionnels, permettant de conduire des analyses de sûreté de fonctionnement (ils se limitent à la génération d'arbre de défaillances). Ils présentent une notation hiérarchique permettant de décrire comment chaque composant d'un système transforme des modes de défaillance en entrées en modes de défaillance de sortie, à l'aide d'assertions proches des assertions Altarica. Les auteurs proposent une démarche de modélisation « top down », partant d'une modélisation abstraite que l'on raffine progressivement. La méthodologie proposée facilite la recherche des modes de défaillance et des événements redoutés mais ne définit pas d'approche de validation des modèles. De plus, ces modèles nous semblent difficiles à relier aux spécifications fonctionnelles. Ce lien avec les spécifications fonctionnelles est par contre établi dans [Reese-Leveson97]. Les auteurs proposent un guide méthodologique pour définir des déviations sur les entrées d'un ensemble de spécifications fonctionnelles formelles et définissent un outil permettant de calculer les scénarios pouvant conduire à un événement redouté. Cette méthodologie a pour avantage de ne pas maintenir deux types de modèles (les spécifications fonctionnelles et le modèle de comportement du système en présence de fautes) mais ne permet pas de traiter des défaillances internes des fonctions. De plus, l'absence de modèles explicites du comportement en présence de défaillance ne permet pas un couplage aisé avec d'autres outils que ceux développés par les auteurs. En fait, l'approche de modélisation la plus proche de la notre semble être celle proposée par [Garret-al95]. Les auteurs proposent une notation graphique pour modéliser les dépendances de fonctionnement entre composants d'un architecture logicielle temps réel en présence de fautes et un outil pour générer des arbres de défaillances à partir des modèles. Le niveau de perception du système est différent du notre : les auteurs s'intéressent à la structure temps réel de l'architecture logicielle (logique d'enchaînement de tâche, allocation de tâche sur des processeurs, logique de reconfiguration, …). La proposition présentée dans cet article se situe à un niveau plus abstrait, en faisant abstraction de la structure d'implantation. Dans le cadre du projet PRISME, nous avons aussi été amené à étudié des modèles de déploiement d'une architecture logicielle sur une architecture matérielle. Nos travaux se poursuivent pour en particulier tracer les dépendances entre des modèles de deux niveaux d'abstraction.

# 6 Bibliographie

[Arnold-al00] « Altarica – Manuel méthodologique – Version 2.0. » A. Arnold, G. Point, A. Griffault, A. Rauzy. rapport interne Labri Université de Bordeaux, mai 2000.
[Fenelon-MacDermid92] « Integrated techniques for software safety analysis», P. Fenelon, J.A. Mac Dermid, IEE colloquium on Hazard Analysis, publisher « Institution of Electrical engineers », 1992
[Fenelon-al94] « Towards integrated safety analysis and design », ACM Applied Computing Review, Aout 1994.
[MacMillan99] « The SMV language » K.L. MacMillan, rapport interne Cadence Berkeley Labs, mars 1999
[Garrett-al95] « The dynamic flowgraph methodology for assessing the dependability of embedded software systems », C.J. Garrett, S.B. Guarro, G.E. Apostolakis, IEEE systems, man and cybernetics, vol 25 no. 5, pages 824-840, mai 1995
[Pagani-al95] « Verification experiments on a large fault-tolerant distributed system », F. Pagani, C. Seguin, P. Siron, V. Wiels, workshop AMAST « Model and Proof », Bordeaux, France, juin 1995
[Parnas 92] « Tabular representations of relations », D. L. Parnas, technical report, CLR report no 260, Mc Master University, Hamilton, Ontario, octobre 1992.
[Reese-Leveson97] « Software Deviation Analysis », J.D. Reese, N. G. Leveson, proceeding ICSE97, Boston USA, 1997.
[Rushby92] « Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems » dans *Formal Techniques in Real-Time and Fault-Tolerant Systems,* LNCS 571, Springer Verlag, janvier 1992.

# 7 Annexe : spécification altarica de l'anémomètre

```
domain

       data= {correct, incorrect, missing};

       /******************/
       /* Anemo function */
       /******************/

node anemo

flow
       /* Inputs */
       pt, ps, p0: data;

       /* Outputs */
       m, z: data;
       status_m, status_z: bool;

       /* Auxiliary variables */
       all_z_in_correct,
       all_m_in_correct,
       anemo_nominal_comp,
       anemo_nominal_detect: bool;

state
       anemo_state: {correct_state, wrong_computation, no_refresh, no_detection,
wrong_alarm} ;

event
       computation_error, refresh_loss, detection_loss, wrong_detection;

/************************/
/* Auxiliary definitions */
/************************/
assert

all_z_in_correct= (p0=correct & ps=correct);
all_m_in_correct= (pt=correct & ps=correct);
```

```
anemo_nominal_comp= ~(anemo_state=wrong_computation | anemo_state=no_refresh);
anemo_nominal_detect= ~(anemo_state=wrong_alarm | anemo_state=no_detection);

/***********************************************/
/* Output computation laws including:          */
/*     - fault detection                        */
/*     - nominal behaviour with counter measure */
/*     - faulty behaviour                       */
/***********************************************/

/* law_sz */
/* Test: correctness, freshness and variation of z inputs + input pt */
/* Propagation of the test result by the status status_z */
       /* Nominal tests */
(anemo_nominal_detect & (all_z_in_correct & pt=correct)=> status_z) &
(anemo_nominal_detect & ~(all_z_in_correct & pt=correct)=> ~status_z) &
/* Faulty tests */
(anemo_state=wrong_alarm => ~status_z) &
(anemo_state=no_detection => status_z);

/* law_sm */
/* Test: correctness, freshness and variation of m inputs */
/* Propagation of the test result by the status status_m */
       /* Nominal tests */
(anemo_nominal_detect & all_m_in_correct => status_m) &
(anemo_nominal_detect & ~all_m_in_correct => ~status_m) &
/* Faulty tests */
(anemo_state=wrong_alarm => ~status_m) &
(anemo_state=no_detection => status_m);

/* law_z */
(anemo_nominal_comp & status_z & all_z_in_correct => z=correct) &
(anemo_nominal_comp & status_z & ~all_z_in_correct => z=incorrect) &
       /* Counter measure : the computed data is not provided */
(anemo_nominal_comp & ~status_z => z=missing) &
       /* Faulty computations */
(anemo_state=wrong_computation => z=incorrect) &
(anemo_state=no_refresh => z=missing) ;

/* law_m */
(anemo_nominal_comp & status_m & all_m_in_correct => m=correct) &
(anemo_nominal_comp & status_m & ~ all_m_in_correct => m=incorrect) &
       /* Counter measure : the computed data is not provided */
(anemo_nominal_comp & ~status_m => m=missing) &
       /* Faulty computations */
(anemo_state=wrong_computation => m=incorrect) &
(anemo_state=no_refresh => m=missing) ;

/*********************************************************************/
/* Evolution of the function state according to its current state and event */
/*********************************************************************/
trans

anemo_state=correct_state |-refresh_loss-> anemo_state:=no_refresh;

anemo_state=correct_state |-computation_error-> anemo_state:=wrong_computation;

anemo_state=correct_state |-detection_loss-> anemo_state:=no_detection;

anemo_state=correct_state |-wrong_detection-> anemo_state:=wrong_alarm;

edon
```