



**HAL**  
open science

## Modélisation de la sûreté de fonctionnement basée sur le langage AADL et les RdPSG

Ana-Elena Rugina, Karama Kanoun, Mohamed Kaâniche

### ► To cite this version:

Ana-Elena Rugina, Karama Kanoun, Mohamed Kaâniche. Modélisation de la sûreté de fonctionnement basée sur le langage AADL et les RdPSG. *Revue des Sciences et Technologies de l'Information - Série TSI: Technique et Science Informatiques*, 2009, 28 (1), pp.7-37. hal-00755268

**HAL Id: hal-00755268**

**<https://hal.science/hal-00755268>**

Submitted on 20 Nov 2012

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Modélisation de la sûreté de fonctionnement basée sur le langage AADL et les RdPSG

Ana-Elena Rugina<sup>\*,1,2</sup> — Karama Kanoun<sup>1,2</sup>  
Mohamed Kaâniche<sup>1,2</sup>

1 CNRS ; LAAS ; 7, avenue du Colonel Roche, F-31077 Toulouse, France

2 Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS-CNRS : F-31077

Toulouse, France

\*Actuellement à EADS ASTRIUM Satellites, Toulouse

ana-elena.rugina@astrium.eads.net, {kanoun, kaaniche}@laas.fr

---

*RÉSUMÉ.* Pour des raisons d'efficacité, le souhait des concepteurs de systèmes est d'utiliser de façon intégrée un ensemble cohérent de formalismes pour décrire les spécifications et conceptions, et également pour effectuer des analyses de sûreté de fonctionnement. AADL (Architecture Analysis and Design Language) a prouvé son aptitude pour la modélisation d'architectures. Cet article présente un cadre de modélisation permettant la génération de modèles analytiques de sûreté de fonctionnement à partir de modèles AADL dans l'objectif de faciliter l'obtention de mesures de sûreté de fonctionnement comme la fiabilité et la disponibilité. Nous proposons une approche itérative de modélisation. Le modèle AADL de sûreté de fonctionnement est transformé en un RdPSG (Réseau de Petri Stochastique Généralisé) en appliquant des règles de transformation de modèle. Un outil de transformation automatique a été développé. Les mesures de sûreté de fonctionnement sont obtenues à partir du traitement du RdPSG résultant par des outils existants. L'approche est illustrée sur un sous-système du système informatique français de contrôle de trafic aérien.

*ABSTRACT.* For efficiency reasons, system designers' will is to use an integrated set of methods and tools to describe specifications, and also to perform dependability analyses. AADL (Architecture Analysis and Design Language) has proved its capacity for architectural modeling. This paper presents a modeling framework allowing the generation of dependability-oriented analytical models from AADL models, to facilitate the evaluation of dependability measures, such as reliability or availability. We propose an iterative approach for system dependability modeling using AADL. The AADL dependability model is transformed into a GSPN (Generalized Stochastic Petri Net) by applying model transformation rules. An automatic transformation tool has been developed. The dependability measures are obtained from the processing of the resulting GSPN by existing tools. The approach is illustrated on a subsystem of the French air traffic control system.

*MOTS-CLÉS :* modélisation de la sûreté de fonctionnement, évaluation, AADL, RdPSG, transformation de modèle.

*KEYWORDS:* dependability modeling, evaluation, AADL, GSPN, model transformation.

---

## 1. Introduction

La complexité croissante des systèmes informatiques entraîne des difficultés d'ingénierie système, en particulier liées à la validation et à l'analyse des performances et des exigences concernant la sûreté de fonctionnement. Des approches d'ingénierie guidée par des modèles sont de plus en plus utilisées dans l'industrie dans l'objectif de maîtriser cette complexité au niveau de la conception. Ces approches encouragent la réutilisation et l'automatisation partielle ou totale de certaines phases du cycle de développement. Elles doivent être accompagnées de langages et outils capables d'assurer la conformité du système implémenté aux spécifications. Les analyses de performance et de sûreté de fonctionnement sont essentielles dans ce contexte. La plupart des approches guidées par des modèles se basent soit sur le langage UML (OMG 2004), qui est un langage de modélisation à usage général, soit sur des ADL (langages de description d'architecture), qui sont des langages propres à des domaines particuliers.

Parmi les ADL, AADL, Architecture Analysis and Design Language (SAE-AS5506 2004), a fait l'objet d'un intérêt croissant dans l'industrie des systèmes critiques (comme Honeywell, Rockwell Collins, l'Agence Spatiale Européenne, Astrium, Airbus). AADL a été standardisé par la « International Society of Automotive Engineers » (SAE) en 2004, pour faciliter la conception et l'analyse de systèmes complexes, critiques, temps réel dans des domaines comme l'avionique, l'automobile et le spatial. AADL fournit une notation textuelle et graphique standardisée pour décrire des architectures matérielles et logicielles et pour effectuer différentes analyses du comportement et des performances du système modélisé (Feiler *et al.* 2004). Le succès de AADL dans l'industrie est justifié par sa sémantique précise, son support avancé à la fois pour la modélisation d'architectures reconfigurables et pour la conduite d'analyses. En particulier, le langage a été conçu pour être extensible afin de permettre des analyses qui ne sont pas réalisables avec le langage de base.

Dans cette optique, une annexe au standard AADL a été définie (« *AADL Error Model Annex* » (SAE-AS5506/1 2006)) pour compléter les capacités de description du langage de base. Cette annexe représente un sous-langage qui sert à décrire les caractéristiques du système modélisé en AADL liées à la sûreté de fonctionnement. La sûreté de fonctionnement d'un système est définie comme la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre (Laprie *et al.* 1996). Selon le système, l'accent peut être mis sur différents attributs de la sûreté de fonctionnement. Par exemple, la continuité de service conduit à la fiabilité et le fait d'être prêt à l'utilisation conduit à la disponibilité. Le *AADL Error Model Annex* offre des primitives permettant de décrire le comportement du système en présence de fautes (fautes, modes de défaillance, propagations d'erreurs, hypothèses de maintenance si on s'intéresse à la disponibilité). En plus de la description du comportement du système en présence de fautes, les concepteurs sont

intéressés par l'obtention de mesures quantitatives d'attributs de la sûreté de fonctionnement pertinents pour leurs systèmes, comme la fiabilité et la disponibilité.

Les développeurs de systèmes complexes critiques, utilisant un processus d'ingénierie basé sur AADL, sont confrontés à deux difficultés majeures quand il s'agit d'évaluer la sûreté de fonctionnement :

- 1) Prise en compte dans les modèles AADL les dépendances entre les composants d'un système dans la description de son comportement en présence de fautes ;. dépendances engendrées par l'architecture du système ou les stratégies de tolérance aux fautes et de maintenance.
- 2) Evaluation des mesures de sûreté de fonctionnement à partir de modèles AADL.

À l'état actuel, il n'existe pas de méthodes pour aider les concepteurs utilisant AADL à surmonter ces difficultés. C'est dans ce cadre que s'inscrivent nos travaux.

Nos travaux aident à surmonter la première difficulté en proposant une méthode itérative de construction d'un modèle AADL de sûreté de fonctionnement qui prend en compte progressivement les dépendances entre les composants. En particulier, la thèse de A. E. Rugina (Rugina 2007) a identifié toutes les primitives du langage AADL qui servent à la description des dépendances liées à la sûreté de fonctionnement, et a défini des règles de modélisation pour chaque type de dépendance. La méthode proposée dans la thèse permet à la fois de maîtriser la complexité des modèles et de les valider progressivement. Cet article a pour premier objectif de donner une vue d'ensemble de cette méthode et de son application. Les informations liées à la sûreté de fonctionnement ne sont pas enfouies dans le modèle AADL architectural. Au contraire, elles sont décrites séparément et attachées aux composants du modèle architectural, favorisant la réutilisation et la clarté du modèle AADL architectural qui peut être une base pour d'autres analyses (comme la vérification formelle (Farines *et al.* 2003), l'ordonnancement et les allocations de mémoire (Singhoff *et al.* 2005), l'allocation de ressources avec l'outil OSATE<sup>1</sup> (Open Source AADL Tool Environment), la recherche de blocages et variables non initialisées avec l'outil Ocarina<sup>2</sup>).

Nos travaux aident à surmonter la deuxième difficulté en proposant une transformation de modèle de AADL vers des réseaux de Petri stochastiques généralisés (RdPSG), qui peuvent être traités par des outils existants pour évaluer les mesures de sûreté de fonctionnement. Les RdPSG sont couramment utilisés pour faciliter l'élaboration de modèles complexes d'évaluation de la sûreté de fonctionnement de systèmes (ils permettant par exemple une construction modulaire et une vérification structurelle des modèles). La transformation de modèle de AADL vers les RdPSG est basée sur un ensemble des règles conçu pour être mis en œuvre dans un outil de transformation de modèle, de façon transparente à l'utilisateur. De

---

<sup>1</sup> <http://www.aadl.info/OpenSourceAADLToolEnvironment.html>

<sup>2</sup> <http://ocarina.enst.fr>

cette manière, la complexité de la génération du modèle RdPSG est masquée aux utilisateurs qui connaissent AADL et qui ont généralement des connaissances limitées dans le domaine des RdPSG. Afin de montrer la faisabilité de l'automatisation, nous avons implémenté un outil de transformation, ADAPT (*from AADL Architectural models to Petri nets through model Transformation*), qui s'interface avec OSATE, côté AADL, et avec Surf-2, côté RdPSG (Béounes *et al.* 1993). Nous avons défini une règle pour chacun des éléments AADL. L'ensemble de règles est donc nécessaire et suffisant pour l'obtention d'un RdPSG décrivant tous les types de dépendances que nous avons identifiés. Pour des raisons de limitation d'espace, nous nous focalisons, dans cet article, sur les principes généraux de la transformation et nous montrons les règles de transformation les plus représentatives.

Enfin, nous donnons une vue d'ensemble du cas d'étude que nous avons utilisé afin d'illustrer à la fois l'approche itérative de modélisation et la transformation de modèle. Ce cas d'étude est issu d'un système réel : le système informatique français de contrôle de trafic aérien.

Une présentation préliminaire des principes de notre méthodologie de modélisation a été effectuée dans (Rugina *et al.* 2006a) et (Rugina *et al.* 2006b). Les principales nouvelles contributions de cet article concernent : 1) la finalisation de l'ensemble des règles de transformation de AADL vers RdPSG ; un outil de transformation automatique a aussi été développé, et 2) l'illustration de l'ensemble des étapes de notre méthode sur un cas d'étude concret et représentatif.

La suite de cet article est organisée de la manière suivante. Le paragraphe 2 discute de l'état de l'art. Le paragraphe 3 présente les éléments qui existent dans la version courante du standard AADL et qui constituent la base de notre approche de modélisation. Le paragraphe 4 est une vue d'ensemble de notre approche itérative de modélisation basée sur AADL. Le paragraphe 5 est dédié aux règles de transformation de AADL vers RdPSG. Le paragraphe 6 illustre notre approche sur un sous-système du Système Français de Contrôle de Trafic Aérien. Pour finir, le paragraphe 7 donne les conclusions et les perspectives de ces travaux.

## **2. Travaux connexes**

La plupart des publications visant l'intégration d'analyses de sûreté de fonctionnement et de performance dans des langages utilisés pour les approches d'ingénierie guidée par des modèles se sont focalisés sur UML, car UML est un langage de modélisation à usage général. Toutefois, des efforts significatifs ont ciblé deux ADLs : EastADL (Debruyne *et al.* 2004) et AADL. Les approches utilisées dans ce contexte se basent sur l'enrichissement du langage ciblé et des transformations de modèle vers des modèles d'analyse.

En considérant les travaux portant sur UML, le projet européen HIDE (Bondavalli *et al.* 2001) a proposé une méthode pour analyser et évaluer automatiquement la sûreté de fonctionnement à partir de modèles UML. Une transformation de modèle a été définie à partir de diagrammes UML structurels et comportementaux vers des RdPSG, pour l'évaluation de la sûreté de fonctionnement. D'autre part, (Pai & Bechta Dugan 2002) et (Fernandez Briones *et al.* 2006) ont élaboré des algorithmes d'obtention d'arbres de fautes à partir de UML. D'autres approches intéressantes ont été développées afin d'obtenir des mesures de performance à partir de UML. Par exemple, (López-Grao *et al.* 2004) se sont focalisés sur la transformation de diagrammes d'activité en RdPSG. (Bernardi *et al.* 2002) proposent de transformer des diagrammes de séquence et « statecharts » en RdPSG. En revanche, (Kloul & Kuster-Filipe 2006) prennent en compte le diagramme global d'interaction de UML2.0 et le diagramme de séquence, qui sont transformés dans le formalisme PEPA.

En considérant les travaux portant sur EastADL, le consortium du projet européen ATESSST (Chen *et al.* 2007) vise à intégrer des analyses de sûreté de fonctionnement et de performance dans ce langage conçu pour répondre aux besoins de l'industrie automobile.

Notre contribution est parallèle aux travaux reportés ci-dessus, car notre objectif est de répondre aux besoins des utilisateurs du langage AADL, qui souhaitent obtenir des mesures de sûreté de fonctionnement. La plupart des articles publiés autour des analyses basées sur AADL se sont concentrés sur l'extension du langage pour faciliter la vérification formelle, comme dans les cas de (Hugues *et al.* 2007) et du projet COTRE (Farines *et al.* 2003). D'autre part, (Singhoff *et al.* 2005) et (Sokolsky *et al.* 2006) proposent des méthodes pour effectuer des analyses d'ordonnancement. La contribution la plus proche de la nôtre est celle de (Joshi *et al.* 2007). Elle présente un outil interne de Honeywell qui permet la génération d'arbres de fautes à partir de modèles AADL enrichis avec des éléments de l'annexe des modèles d'erreur (*AADL Error Model Annex*). L'outil est interfacé avec OSATE, similairement à notre outil. D'un point de vue critique, cette contribution ne fournit pas de méthodologie de modélisation et ne guide pas l'utilisateur dans la construction du modèle AADL de sûreté de fonctionnement. En même temps, la transformation de modèle de AADL vers les arbres de faute n'est pas détaillée. De plus, cette approche cible uniquement la fiabilité et ne peut pas être utilisée pour évaluer d'autres mesures de sûreté de fonctionnement, telles que la disponibilité, si les composants ne sont pas stochastiquement indépendants.

Les chaînes de Markov constituent un cadre plus adéquat pour la modélisation de la sûreté de fonctionnement des systèmes en prenant en compte les dépendances entre les composants. Habituellement, elles sont générées à partir de formalismes de plus haut niveau comme les réseaux de Petri stochastiques généralisés (RdPSG). Ces derniers permettent de vérifier structurellement le modèle avant la génération de la chaîne de Markov. Ces facilités pour la vérification sont très utiles quand on traite de grands modèles. La méthode de modélisation proposée dans cet article s'inspire

de l'approche (Kanoun & Borrel 2000). Cette dernière a pour objectif la maîtrise de la construction et la validation progressive de modèles de sûreté de fonctionnement sous forme de RdPSG. Dans cet article nous nous intéressons spécifiquement à la modélisation au niveau AADL et à la génération à partir de ces modèles AADL de RdPSG permettant d'obtenir des mesures quantitatives de sûreté de fonctionnement.

### 3. Eléments de AADL

Ce paragraphe fournit au lecteur les concepts de base et éléments d'utilisation du langage AADL, conformément à la version actuelle du standard. La méthodologie que nous présentons par la suite se base sur ces éléments.

En AADL, les systèmes sont modélisés comme des ensembles de *composants logiciels* (processus, fils d'exécution appelés « threads », sous-programmes, données) qui interagissent via des *ports* (de données ou d'événements) et qui s'exécutent sur des *composants matériels* (processeurs, mémoire, bus).

Afin de dissocier l'interface et l'implémentation, chaque composant AADL a deux niveaux de description : le *type* et l'*implémentation*. Le type décrit comment l'environnement « voit » ce composant (en termes de propriétés et caractéristiques). Une ou plusieurs implémentations peuvent être associées au même type, correspondant à différentes structures du composant en termes de sous composants, connexions (entre les ports des sous composants), modes opérationnels.

Un système peut avoir plusieurs modes opérationnels qui correspondent soit à des valeurs différentes pour les propriétés, soit à des configurations différentes. Le basculement de l'une à l'autre se fait soit suite à un changement de phase opérationnelle, soit suite à une reconfiguration du système après la défaillance d'un de ses éléments. AADL fournit un support pour définir des configurations architecturales et le passage entre les modes. La dynamique des modes opérationnels influence les mesures de sûreté de fonctionnement comme la disponibilité. Par conséquent, ils doivent être pris en compte dans le modèle de sûreté de fonctionnement. En pratique, les transitions entre modes opérationnels sont déclenchées par des événements qui arrivent à travers des ports d'événements. Le modèle défini par l'ensemble des composants du système et des modes opérationnels est appelé par la suite « *modèle AADL architectural* ».

Un modèle de sûreté de fonctionnement est un modèle contenant des informations telles que modes de défaillance, politiques de réparation et propagations d'erreur. Un *modèle AADL de sûreté de fonctionnement* est un modèle architectural annoté avec des modèles d'erreur qui contenant ces informations. La syntaxe et la sémantique des modèles d'erreur est spécifiée par l'annexe standardisée des modèles d'erreur (« Error Model Annex »). Les modèles d'erreurs représentent des automates stochastiques décrivant des comportements en présence de fautes. Ils sont associés à des éléments du modèle AADL architectural. Au

moment de l'association, il est possible d'adapter un modèle d'erreur générique provenant d'une librairie. La suite du paragraphe détaille l'utilisation des modèles d'erreur.

Comme pour un composant du langage AADL de base, un modèle d'erreur est spécifié sous forme d'un type et d'une ou plusieurs implémentations appropriées pour la réalisation de différentes analyses de sûreté de fonctionnement. Le type déclare des états (*error states*), des événements internes au composant (*error events*) et des propagations (*error propagations*<sup>3</sup>) qui circulent à travers de connexions et liaisons du modèle AADL architectural. Les implémentations déclarent des *transitions* entre les états et des propriétés stochastiques d'Occurrence pour les événements et les propagations sortantes. Les transitions sont déclenchées par des événements et propagations déclarés dans le type. Les propriétés d'Occurrence spécifient le taux d'arrivée ou la probabilité d'occurrence pour les événements et propagations. La Figure 1 montre un modèle d'erreur d'un composant logiciel considéré comme indépendant (sans propagations). Par conséquent, nous avons choisi le nom *independent* pour son type et *independent.general* pour son implémentation. Nous considérons deux types de fautes : temporaires et permanentes. Une faute temporaire mène le composant dans un état erroné (*Erroneous*) tandis qu'une faute permanente le mène dans un état défaillant (*Failed*). Une faute temporaire peut être traitée par des mécanismes de recouvrement internes permettant au composant de retrouver son état initial. Une faute permanente requiert le redémarrage du composant.

```

Error Model Type [independent]

error model independent
features
  Error_Free: initial error state;
  Erroneous: error state;
  Failed: error state;
  Temp_Fault: error event {Occurrence => poisson λ1};
  Perm_Fault: error event {Occurrence => poisson λ2};
  Restart: error event {Occurrence => poisson μ1};
  Recover: error event {Occurrence => poisson μ2};
end independent;
```

---

<sup>3</sup> Dans la suite de l'article, nous omettrons dans tout contexte non ambigu le terme « erreur » quand nous ferons référence aux états, événements, propagations et transitions. Notons que les états peuvent représenter des états de bon fonctionnement, les événements peuvent représenter des réparations et les propagations peuvent représenter toute notification.



```

Error Model Implementation [independent.general]
error model implementation independent.general
transitions
  Error_Free-[Perm_Fault]->Failed;
  Error_Free-[Temp_Fault]->Erroneous;
  Failed-[Restart]->Error_Free;
  Erroneous-[Recover]->Error_Free;
end independent.general;

```

**Figure 1.** Exemple de modèle d'erreur sans propagations.

Les modèles d'erreur de différents composants ne peuvent communiquer qu'à travers les connexions et liaisons du modèle architectural.

La mécanique d'une propagation est la suivante. Une propagation *out* est modélisée dans un modèle d'erreur source. Elle arrive selon une propriété d'Occurrence spécifiée par l'utilisateur. Le modèle d'erreur source envoie la propagation à travers tous les ports et liaisons du composant auquel le modèle d'erreur est associé. Par conséquent, une propagation *out* arrive à un ou plusieurs modèles d'erreurs associés à des composants récepteurs. Si un modèle d'erreur récepteur déclare une propagation *in* avec le même nom que la propagation *out* reçue, la propagation *in* peut influencer son comportement, en déclenchant des transitions entre des états et/ou des modes opérationnels. Dans certains cas, il est souhaitable de modéliser comment sont gérées des propagations provenant de plusieurs sources. Ceci est modélisé par des propriétés de type « Guard » associées aux ports. Ces propriétés permettent de spécifier des filtres et des conditions de masquage pour les propagations.

Les états logiques (tels que défaillant et en bon fonctionnement) d'un composant sont décrits indépendamment des modes opérationnels. Le langage permet d'établir une connexion entre les états logiques et les modes opérationnels. Par exemple, l'occurrence d'une transition entre modes opérationnels est éventuellement contrôlée par la spécification de propriétés *Guard\_Transition* associées à des ports, portant sur la configuration d'états de plusieurs composants.

Dans la suite de cet article, plusieurs exemples illustreront l'usage des propagations (voir par exemple § 4.2.1) et des propriétés *Guard\_Transition* (voir par exemple § 5.3.1).

#### 4. L'approche de modélisation

L'évaluation de la sûreté de fonctionnement se décompose en trois étapes : 1) la définition des mesures à évaluer, 2) la construction du modèle d'évaluation de la sûreté de fonctionnement qui décrit le comportement du système en présence de fautes et 3) le traitement du modèle afin d'obtenir les valeurs des mesures

recherchées. Plusieurs mesures quantitatives peuvent être considérées pour caractériser la sûreté de fonctionnement du système étudié, en fonction du domaine d'application et de la criticité des modes de défaillance. Pour des systèmes complexes, la principale difficulté dans la construction du modèle de sûreté de fonctionnement est due aux dépendances entre les composants du système. Les dépendances sont de plusieurs types, identifiés dans (Kanoun & Borrel 2000) : structurelles, fonctionnelles ou liées à la tolérance aux fautes et à la stratégie de maintenance et de restauration. Les échanges de données entre composants entraînent des dépendances fonctionnelles. L'exécution d'un processus sur un processeur entraîne une dépendance structurelle entre le fil d'exécution et le processeur. Le changement du mode opérationnel selon une politique de tolérance aux fautes représente une dépendance de tolérance aux fautes. Le partage d'un dispositif de réparation entre plusieurs composants entraîne une dépendance de maintenance et de restauration.

Les dépendances fonctionnelles, structurelles et de tolérance aux fautes constituent des *dépendances architecturales* qui apparaissent généralement sur le modèle AADL architectural. Il faut tenir compte également des dépendances dues à la maintenance, qui n'apparaissent pas dans le modèle architectural.

Les utilisateurs ont besoin d'une approche structurée pour modéliser systématiquement les dépendances afin d'éviter des erreurs dans le modèle du système. Dans notre approche, le modèle AADL de sûreté de fonctionnement est construit de manière itérative, ce qui permet à la fois de maîtriser sa complexité et de le valider progressivement. Plus concrètement, dans une première itération, nous modélisons les comportements des composants du système en présence de leurs propres fautes et événements de réparation uniquement. Par conséquent, les composants sont modélisés comme s'ils étaient *isolés* du reste du système. Dans les itérations suivantes, nous complétons le modèle en introduisant progressivement les dépendances entre les composants. Le modèle AADL de sûreté de fonctionnement est mis à jour à chaque itération. La prise en compte d'une dépendance conduit à ajouter uniquement de nouvelles informations dans le modèle existant (en termes de propagations) ou à le modifier avant d'ajouter de nouvelles informations.

Le modèle d'évaluation de la sûreté de fonctionnement peut être généré en une seule fois à la fin des itérations, à partir du modèle AADL global. Il peut être également généré lors de chaque itération. Pour les raisons évoquées dans § 2, dans cet article nous nous intéressons à la génération d'un RdPSG à partir du modèle AADL. La vérification à chaque étape des propriétés du RdPSG permet de valider progressivement le RdPSG et par conséquent le modèle AADL associé.

Dans la suite, le paragraphe 4.1 donne une vue d'ensemble de notre approche et le paragraphe 4.2 illustre la modélisation des dépendances.

#### 4.1. Vue d'ensemble

Une vue d'ensemble de notre approche de modélisation, composée de quatre étapes, est illustrée par la Figure 2.

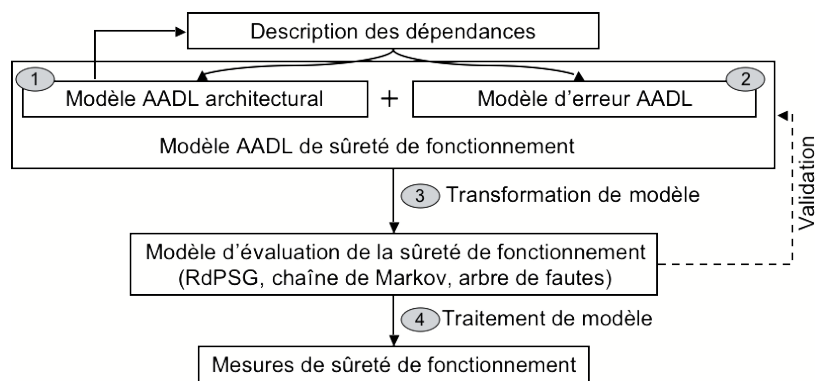
**La première étape** est consacrée à la modélisation de l'architecture du système en AADL<sup>4</sup> (c'est-à-dire, sa structure en termes de composants et les modes opérationnels de ces composants).

**La seconde étape** est dédiée à la construction des modèles d'erreur associés aux composants du modèle architectural. Le modèle d'erreur du système est une composition de l'ensemble de modèles d'erreur associés aux composants, en prenant en compte les dépendances entre ces derniers. La construction du modèle d'erreur tient compte des dépendances architecturales et des hypothèses liées à la maintenance.

Le modèle AADL architectural et le modèle d'erreur du système forment le *modèle AADL de sûreté de fonctionnement*.

**La troisième étape** vise à construire un modèle d'évaluation de la sûreté de fonctionnement à partir du modèle AADL de sûreté de fonctionnement à l'aide de règles de transformation de modèle.

**La quatrième étape** est dédiée au traitement du modèle d'évaluation de la sûreté de fonctionnement afin d'obtenir des mesures.



**Figure 2.** Approche proposée

<sup>4</sup> Le modèle AADL architectural peut être disponible à ce stade s'il a déjà été construit pour d'autres analyses.

Pour obtenir le modèle AADL de sûreté de fonctionnement, l'utilisateur doit effectuer la première et la deuxième étapes décrites ci-dessus. La troisième étape est conçue pour être automatisée. Dans cet article nous nous focalisons sur la génération d'un RdPSG à partir du modèle AADL. La quatrième étape est entièrement basée sur des algorithmes classiques de traitement des modèles RdPSG. Ne faisant pas l'objet de notre travail, cette étape n'est pas détaillée dans cet article.

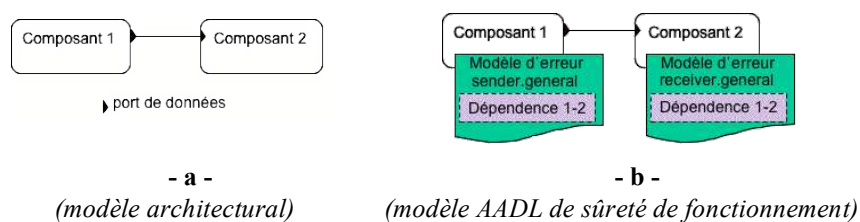
#### 4.2. Modélisation avec dépendances en AADL

Les dépendances structurelles et fonctionnelles sont engendrées par le modèle AADL architectural. L'utilisateur doit enrichir cet ensemble en ajoutant les dépendances liées à la maintenance et à la tolérance aux fautes. Dans le cas d'un système complexe, l'ensemble complet des dépendances peut être résumé dans un *diagramme bloc de dépendances* pour donner une vue globale des composants du système et de leurs interactions.

La suite de ce paragraphe illustre comment modéliser en AADL i) une dépendance architecturale et ii) une dépendance de maintenance. La première est déjà prise en compte dans le modèle AADL architectural initial, alors que la seconde n'intervient qu'au niveau de l'analyse de sûreté de fonctionnement et elle n'apparaît pas dans le modèle AADL architectural.

##### 4.2.1. Dépendance architecturale

La dépendance est supportée par le modèle architectural et doit être modélisée dans les modèles d'erreur associés aux composants dépendants, en spécifiant respectivement des propagations sortantes et entrantes et leurs impacts sur les modèles d'erreur. Un exemple est donné dans la Figure 3. La Figure 3-a présente le modèle AADL architectural (le *Composant 1* envoie des données au *Composant 2*). La Figure 3-b montre le modèle AADL de sûreté de fonctionnement où un modèle d'erreur est associé à chacun des deux composants pour décrire la dépendance.



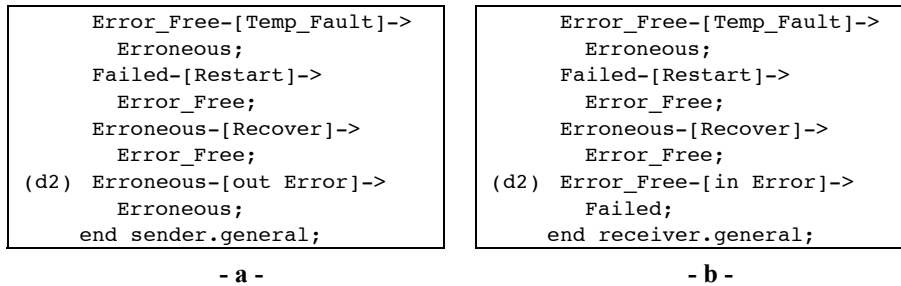
**Figure 3.** Dépendance architecturale.

Le modèle d'erreur de la Figure 4-a (*sender.general*) est associé au *Composant 1*. Il prend en compte la partie côté émetteur de la dépendance du *Composant 1* vers le *Composant 2*. Ce modèle d'erreur est une extension de celui de la Figure 1 qui représente le comportement d'un composant comme s'il était isolé (il ne déclare pas des propagations). Le modèle d'erreur *sender.general* déclare une propagation *out Error* (voir la ligne d1 de la Figure 4-a) dans le type et une transition AADL déclenchée par la propagation *out* dans l'implémentation (voir la ligne d2 de la Figure 4-a). L'occurrence de la propagation *out Error* est caractérisée par une probabilité fixe  $p$ . Le modèle d'erreur associé au *Composant 2* (*receiver.general*) est similaire. La seule différence est la direction de la propagation *Error*. Cette propagation *in* déclenche une transition de l'état *Error\_Free* à *Failed*.

Quand le *Composant 1* est dans l'état erroné (*Erroneous*), il envoie une propagation par la connexion unidirectionnelle et reste dans le même état<sup>5</sup>. Par conséquent, la propagation entrante *Error* cause la défaillance du composant récepteur *Component 2*. Les propagations *in - out Error* définies respectivement dans le modèle d'erreur associé au *Composant 2* et au *Composant 1* ont des noms identiques. De telles propagations sont appelées dans la suite *propagations avec noms identiques*.

<pre> <b>Error Model Type [sender]</b>  error model sender features   Error_Free:     initial error state;   Erroneous: error state;   Failed: error state;   Temp_Fault: error event   {Occurrence=&gt; poisson <math>\lambda</math>1};   Perm_Fault: error event   {Occurrence=&gt; poisson <math>\lambda</math>2};   Restart: error event   {Occurrence=&gt; poisson <math>\mu</math>1};   Recover: error event   {Occurrence=&gt; poisson <math>\mu</math>2}; (d1) Error:out error propagation   {Occurrence =&gt; fixed p}; end sender; </pre>	<pre> <b>Error Model Type [receiver]</b>  error model receiver features   Error_Free:     initial error state;   Erroneous: error state;   Failed: error state;   Temp_Fault: error event   {Occurrence =&gt; poisson <math>\lambda</math>1};   Perm_Fault: error event   {Occurrence =&gt; poisson <math>\lambda</math>2};   Restart: error event   {Occurrence =&gt; poisson <math>\mu</math>1};   Recover: error event   {Occurrence =&gt; poisson <math>\mu</math>2}; (d1) Error: in error propagation end receiver; </pre>
<pre> <b>Error Model Implementation</b> <b>[sender.general]</b>  error model implementation sender.general transitions   Error_Free-[Perm_Fault]-&gt;   Failed; </pre>	<pre> <b>Error Model Implementation</b> <b>[receiver.general]</b>  error model implementation receiver.general transitions   Error_Free-[Perm_Fault]-&gt;   Failed; </pre>

<sup>5</sup> Il est à noter qu'il est possible de déclarer des transitions déclenchées par des propagations *out* dont l'état source est différent de l'état destination.

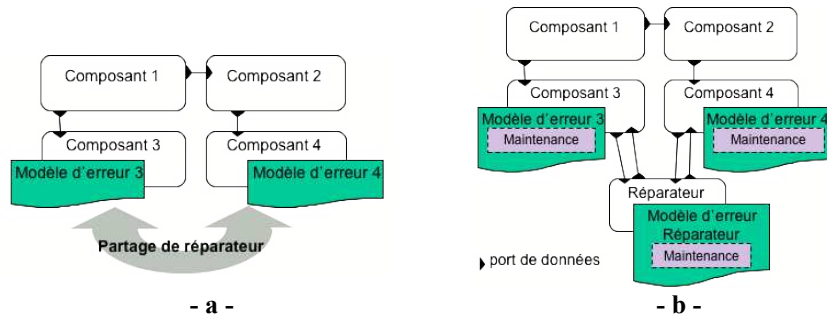


**Figure 4.** Exemple de modèles d'erreur avec dépendance.

Généralement, un composant du modèle architectural peut recevoir des propagations provenant de plusieurs composants émetteurs. Il est parfois nécessaire de ne prendre en compte certaines de ces propagations que dans certains contextes. Ces conditions sont spécifiées en utilisant des propriétés de `Guard` dans lesquelles les conséquences d'un ensemble de propagations provenant de plusieurs émetteurs sur un récepteur sont spécifiées au travers d'expressions booléennes.

#### 4.2.2. Dépendance de maintenance

Des composants qui ne sont pas dépendants au niveau architectural peuvent le devenir à cause de la stratégie de maintenance. Dans ce cas, le modèle architectural nécessite des ajustements pour supporter la description des dépendances liées à la stratégie de maintenance. Comme les modèles d'erreur interagissent seulement par des propagations qui passent par des éléments architecturaux (par exemple connexions et liaisons), la dépendance de maintenance doit avoir un support architectural. En d'autres termes, à part les composants de l'architecture, nous sommes conduits à ajouter un composant dans le modèle architectural pour décrire la stratégie de maintenance. La Figure 5-a montre un exemple de modèle AADL de sûreté de fonctionnement. Dans cette architecture, le *Composant 3* et le *Composant 4* n'interagissent pas au niveau de l'architecture AADL car il n'y a pas de dépendance architecturale. Cependant, si nous supposons que les deux composants partagent un réparateur, la stratégie de maintenance doit être prise en compte dans les modèles d'erreur correspondants. Par conséquent, il est nécessaire de représenter le réparateur au niveau du modèle architectural, comme montré dans la Figure 5-b, pour modéliser explicitement la dépendance de maintenance entre le *Composant 3* et le *Composant 4*.



**Figure 5.** *Dépendance de maintenance.*

Les modèles d'erreur des composants dépendants ont également besoin d'être ajustés. Par exemple, pour représenter le fait que le *Composant 3* ne peut redémarrer que si le *Composant 4* est en état de bon fonctionnement, il faut décomposer l'état défaillant du *Composant 3* pour faire la distinction entre un état en attente d'autorisation de redémarrage et un état à partir duquel le *Composant 3* est autorisé à redémarrer.

#### 4.2.3. Aspects pratiques

L'ordre de prise en compte des dépendances n'a pas d'impact sur le modèle AADL de sûreté de fonctionnement final. Toutefois, il peut avoir un impact sur la réutilisation des modèles intermédiaires. Il est conseillé de guider le choix de l'ordre de prise en compte des dépendances en fonction de l'analyse ciblée. En général, les dépendances liées à la tolérance aux fautes et à la maintenance sont modélisées à la fin puisque leur description est très liée aux autres dépendances. Un objectif majeur de l'évaluation de la sûreté de fonctionnement est de sélectionner les politiques de tolérance aux fautes et de maintenance les mieux adaptées pour l'application.

Afin de permettre l'évaluation des mesures de sûreté de fonctionnement, l'utilisateur doit définir des classes d'états pour le système. Par exemple, si l'utilisateur souhaite évaluer la fiabilité ou la disponibilité, il est nécessaire de définir les états considérés défaillants. Si, de plus, l'utilisateur souhaite évaluer des mesures liées à la sécurité-innocuité, il est nécessaire de définir également les états considérés catastrophiques. En AADL, les classes d'états sont définies au travers d'un modèle d'erreur dérivé associé au système et décrivant les états de ce dernier comme une expression booléenne faisant référence aux états de ses composants.

## 5. Règles de transformation

Le modèle RdPSG du système est construit par la transformation du modèle AADL de sûreté de fonctionnement en suivant une approche modulaire. Le RdPSG

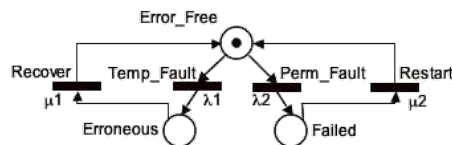
du système est composé de sous-réseaux interconnectés. Un sous-réseau est associé à un composant ou à une dépendance.

Afin d'aboutir à un RdPSG contenant toutes les informations nécessaires à l'évaluation des mesures de sûreté de fonctionnement, nous avons défini des règles de transformation pour tous les éléments AADL décrivant les types de dépendances identifiés au § 4. Toutes les règles sont définies afin d'assurer par construction les propriétés syntaxiques du RdPSG (borné et sans boucle infinie formée d'une suite de transitions instantanées). Les règles sont systématiques et automatisables. Le RdPSG résultant est indépendant des outils (nous n'utilisons pas des prédicats ou des caractéristiques dépendantes des outils). Néanmoins, nos règles sont simplifiables pour cibler certains outils évolués de traitement de RdPSG.

Dans les trois paragraphes suivants, nous présentons quelques dépendances en utilisant des éléments AADL (formés de primitives du langage) et nous donnons les règles de transformation correspondantes. Les règles de transformation que nous présentons ici sont les plus représentatives. Elles s'appliquent aux i) composants isolés, ii) propagations *in - out* avec noms identiques (type de dépendance très fréquent) et iii) des systèmes avec modes opérationnels (indispensables pour décrire des stratégies de tolérance aux fautes ou des systèmes multi-phasés). L'ensemble complet des règles est présenté dans (Rugina 2007). Cet ensemble a été mis en oeuvre dans un outil, ADAPT (*from AADL Architectural models to Petri nets through model Transformation*) (Rugina et al. 2008).

### 5.1. Composants isolés







Dans le cas d'un composant isolé ou dans le cas d'un ensemble de composants indépendants, la transformation est plutôt directe, car un modèle d'erreur représente un automate stochastique, comme montré dans l'exemple de la Figure 1. La transformation du modèle d'erreur de la Figure 1 nous conduit au RdPSG de la Figure 6. Le nombre de jetons dans un réseau composant est toujours 1 (un composant ne peut pas être dans plusieurs états à la fois). Le Tableau 1 montre les règles de transformation appliquées.



**Figure 6.** RdPSG correspondant au modèle d'erreur de la Figure 1.



**Tableau 1.** Règles de transformation pour composants isolés

Primitive du modèle d'erreur	Élément du RdPSG	
Etat	Place	
Etat initial	Place avec jeton	
Événement	Transition RdPSG (temporisée ou immédiate)	
Propriété d'Occurrence <sup>6</sup>	Poids de la transition RdPSG (taux pour la distribution de Poisson ou probabilité fixe)	 Temporisée
		 Immédiate
Transition AADL (Etat_Src-[Événement] -> Etat_Dest)	Arcs connectant des places (corresp. aux Etat_Src et Etat_Dest en AADL) via transition RdPSG (corresp. à l'Événement AADL)	

### 5.2. Propagations in – out

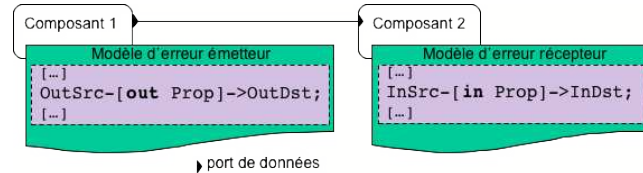
Dans le cas le plus général, une propagation *out* déclarée dans un modèle d'erreur émetteur pourrait être déclenchée à partir de  $n$  transitions AADL dans ce même modèle d'erreur (par exemple une propagation *Failed* pourrait être propagée à partir d'un état *FailStopped* et à partir d'un état *FailRandom*). Des propagations *in* à noms identiques pourraient être déclarées dans  $r \geq 2$  modèles d'erreur récepteurs et pourraient déclencher  $m_j$  transitions AADL dans chaque récepteur  $j$  ( $j = 1 \dots r$ ). Nous avons identifié et analysé plusieurs règles de transformation pour la même spécification AADL pour des propagations *in* – *out* avec noms identiques. Nous avons choisi la règle la plus adaptée à l'automatisation, car l'objectif est de cacher la génération du RdPSG à l'utilisateur en automatisant complètement la transformation.

Nous présentons d'abord le cas général d'une paire de propagations *in* – *out* avec noms identiques déclarées dans deux composants connectés. Ensuite nous présentons la règle de transformation.

Dans la Figure 7, le *Composant 1* joue le rôle de l'émetteur de propagation et il envoie des propagations nommées *Prop* par la connexion qui arrive au *Composant 2*. L'occurrence d'une propagation *out Prop* dans le *Composant 1* déclenche également un changement d'état dans ce même composant qui passe de

<sup>6</sup> En AADL, l'occurrence d'un événement est une propriété caractérisée par un couple de formé d'un mot clé désignant une probabilité fixe (*fixed*) ou une distribution (*Poisson* ou *nonstandard*) et d'une valeur numérique ou symbolique représentant soit la probabilité d'occurrence, soit le paramètre de la distribution. Comme nous utilisons les RdPSG, nous considérons uniquement les probabilités fixes et les distributions de Poisson. Il est à noter que, dans le RdPSG, ces poids seront ensuite normalisés selon le contexte de franchissement.

l'état *OutSrc* à l'état *OutDst*. Le *Composant 2* joue le rôle du récepteur. S'il reçoit une propagation nommée *Prop*, il passe de l'état *InSrc* à l'état *InDst*.

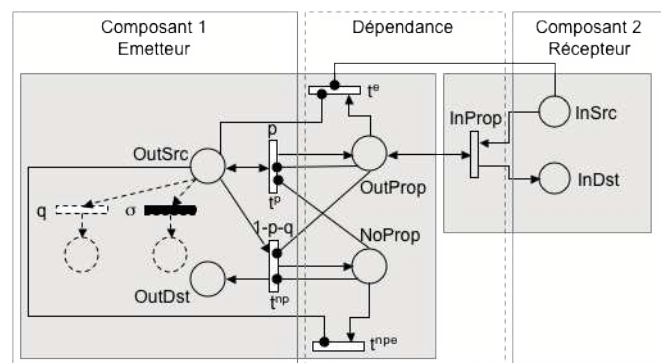


**Figure 7.** Emetteur et récepteur – propagations avec noms correspondants.

La règle de transformation consiste à découpler les propagations *in* et *out* dans le RdPSG à travers une place intermédiaire qui représente le fait qu'une propagation *out Prop* a eu lieu, comme montré dans la Figure 8 (place *OutProp*).

Un jeton arrive dans la place *OutProp* quand une transition RdPSG ( $t^p$ ) correspondant à la propagation *out* (et caractérisée par sa probabilité d'Occurrence  $p$ ) arrive. L'existence d'un jeton dans la place *OutProp* permet de tirer une transition RdPSG immédiate *InProp* (si la place *InSrc* du *Composant 2* est marquée) qui correspond à la propagation *in*. Cette place intermédiaire est vidée par la transition  $t^e$  quand la place correspondant à l'état source de la propagation *out* dans le composant émetteur est vide et quand *InProp* n'est pas sensibilisée. Nous ne vidons pas cette place directement lors du tir de la transition RdPSG correspondant à la propagation *in*, car nous devons mémoriser l'occurrence de la propagation *out*. Cette mémoire est utilisée par les autres règles de transformation.

La place *NoPropag* modélise la situation de la non-occurrence de la propagation *out Prop* quand le *Composant 1* est dans l'état *OutSrc*. Si *out Prop* n'arrive pas, la transition  $t^{np}$  est tirée. Sa probabilité tient compte de la somme des probabilités de tous les événements et les propagations déclenchant des transitions à partir de l'état *OutSrc*. La transition  $t^{npe}$  vide la place *NoProp* quand le composant a quitté *OutSrc*.



**Figure 8.** Propagation de l'émetteur au récepteur. Règle de transformation

Il est à noter que, dans le modèle AADL de sûreté de fonctionnement, chaque dépendance est modélisée dans les modèles d'erreur impliqués dans la dépendance. Dans le RdPSG, la dépendance est modélisée par un sous-réseau, obtenu à partir d'informations qui existent dans (au moins) deux modèles d'erreur dépendants. Une propagation *in* n'a aucun sens si elle ne correspond pas à une propagation *out* déclarée dans le modèle d'erreur d'un autre composant. Par conséquent, l'ensemble de propagations *in-out* ayant des noms identiques forme un élément AADL.

La formalisation de cette règle de transformation apparaît dans (Rugina 2007). Par manque d'espace, nous ne la présentons pas dans cet article.

Dans le cas général de  $n$  transitions AADL déclenchées par une propagation *out*, avec des propagations *in* correspondantes dans plusieurs modèles d'erreur récepteurs, une transition RdPSG est créée pour chaque transition AADL déclenchée par la propagation *out* dans le modèle d'erreur émetteur et une transition RdPSG est créée pour chaque transition AADL déclenchée par la propagation *in* dans les récepteurs. Le nombre de transitions RdPSG ( $N_r$ ) nécessaires pour décrire les propagations *in Prop* dans  $r$  composants récepteurs comme effets de  $n$  propagations émises par un composant émetteur est donné par l'expression [1] ci-après :

$$N_r = n * \sum_{j=1}^r m_j, \forall r \geq 1 \quad [1]$$

où  $n$  = le nombre de transitions AADL déclenchées par la propagation *out* dans le modèle d'erreur émetteur;  
 $r$  = le nombre de modèles d'erreur récepteurs;  
 $m_j$  = le nombre de transitions AADL déclenchées par la propagation *in* dans le modèle d'erreur récepteur  $j$ .

La Figure 9-a montre un modèle AADL avec un émetteur et deux récepteurs. A partir de ce dernier, nous obtenons le RdPSG de la Figure 9-b.

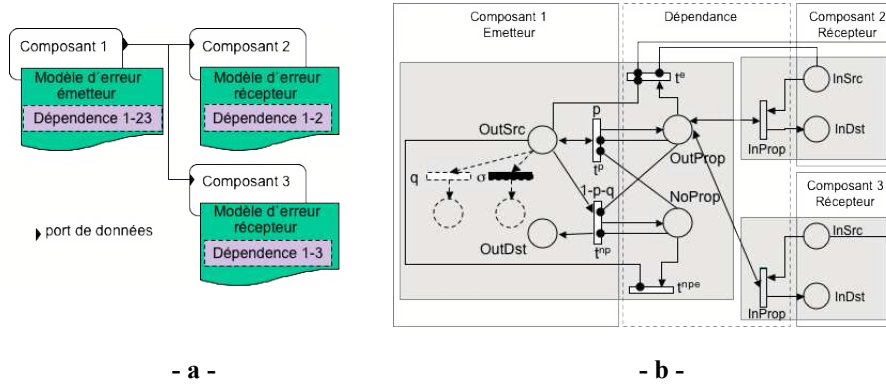


Figure 9. Propagation d'un émetteur vers deux récepteurs.

### 5.3. Systèmes avec modes opérationnels

AADL offre plusieurs mécanismes pour connecter les états logiques du modèle d'erreur aux transitions entre des modes opérationnels. Pour des raisons de limitation d'espace, dans ce paragraphe, nous nous focalisons sur les règles de transformation pour des propriétés `Guard_Transition`. Ces dernières conditionnent l'occurrence d'une transition entre modes opérationnels en fonction d'états de plusieurs composants du système (connectés ou liés aux composants qui déclarent ces propriétés) afin de modéliser de façon globale l'évolution du système. Leur syntaxe est donnée en forme Backus-Naur dans la Figure 10.

```

Guard_Transition ::= boolean_expr applies to EventPort {, EventPort}*;
EventPort ::= outEventPortOfSubcomp | inEventPortOfComp
boolean_expr ::= conjunction | boolean_expr OR boolean_expr
conjunction ::= variable | conjunction AND conjunction
variable ::= EventPort[ [StateOrPropagation | NOT StateOrPropagation] ]

```

Figure 10. Syntaxe des propriétés `Guard_Transition`.

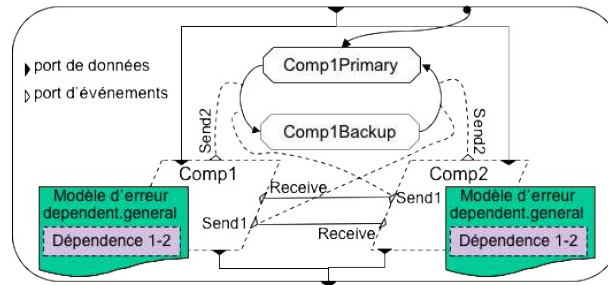
Dans la suite, nous présentons successivement la modélisation en AADL d'un exemple de système avec modes opérationnels, la règle de transformation et son illustration sur l'exemple.

#### 5.3.1. Modélisation AADL de propriétés `Guard_Transition`

Nous présentons d'abord un exemple de système avec deux modes opérationnels dans la Figure 11 et nous montrons dans la Figure 12 l'association d'une propriété `Guard_Transition` aux ports impliqués dans les transitions entre modes

opérationnels. La configuration d'états nécessaire pour déclencher une transition entre modes opérationnels est exprimée comme une expression booléenne construite à partir de variables symbolisant des états et des propagations.

Dans la Figure 11, le système est représenté en utilisant la notation graphique AADL. Il contient deux composants actifs identiques et deux modes opérationnels (*Comp1Primary* et *Comp1Backup*). Le système est initialement dans le mode *Comp1Primary*. La transition du mode *Comp1Primary* vers le mode *Comp1Backup* est régie par des propagations arrivant par les ports *Send2* de *Comp1* et *Send1* de *Comp2*. Elle peut être déclenchée par exemple si *Comp1* défaille quand *Comp2* est en état de bon fonctionnement. Dans ce cas, *Comp2* doit prendre la main.



**Figure 11.** Modèle AADL d'un système avec modes opérationnels.

Le même modèle d'erreur est associé à *Comp1* et à *Comp2*. Il est basé sur le modèle d'erreur pour des composants isolés (voir la Figure 1). Il déclare, en plus de ce dernier, une propagation out *FailedVisible* qui notifie la défaillance du composant et qui est utilisée dans les propriétés *Guard\_Transition*.

Les propriétés *Guard\_Transition* sont associées aux ports impliqués dans les transitions entre modes opérationnels. Une transition entre deux modes opérationnels a lieu si la propriété *Guard\_Transition* associée au port nommé dans la transition est vraie. Dans notre exemple, la transition du mode *Comp1Primary* au mode *Comp1Backup* a lieu si *Comp1* envoie une propagation out *FailedVisible* et si, en même temps, *Comp2* est dans l'état *Error\_Free*. (voir lignes g1-g3 de la Figure 12). La condition complémentaire doit être vraie pour que la transition du mode *Comp1Backup* au mode *Comp1Primary* ait lieu (voir lignes g5-g6 de la Figure 12).

```

annex Error_Model {**
(g1)   Guard_Transition =>
(g2)   (Comp1.Send[FailedVisible] and Comp2.Send[Error_Free])
(g3)   applies to Comp1.Send;
(g4)   Guard_Transition =>
(g5)   (Comp2.Send[FailedVisible] and Comp1.Send[Error_Free])
(g6)   applies to Comp2.Send;
**};

```

**Figure 12.** Associations de propriétés *Guard\_Transition*.

### 5.3.2. Transformation des propriétés *Guard\_Transition*

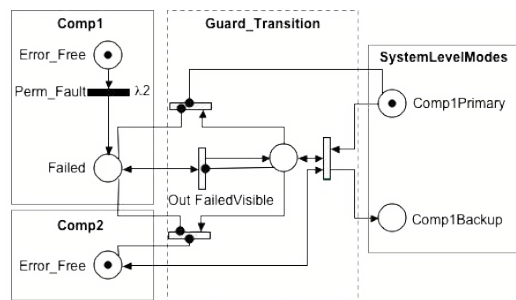
Les modes opérationnels sont directement transformés en places du RdPSG.

L'expression booléenne de la propriété *Guard\_Transition* doit être tout d'abord mise sous une forme normale disjonctive (FND). Chaque conjonction est transformée en une transition RdPSG immédiate connectée avec :

- les places correspondant aux états et propagations *out* qui apparaissent dans la conjonction par des arcs bidirectionnels ou arcs inhibiteurs (dépendant de l'existence ou non de négations dans l'expression).
- les places correspondant aux modes opérationnels qui apparaissent dans la transition déclenchée par le port auquel est associée la propriété *Guard\_Transition*.

Une place intermédiaire correspondant à une propagation *out* est vidée quand aucune transition reliée à cette place n'est sensibilisée.

Nous illustrons la règle de transformation sur l'exemple de système avec modes opérationnels décrit au §5.3.1. La Figure 13 montre le RdPSG correspondant à la première propriété *Guard\_Transition* (lignes g1-g3) de la Figure 12.



**Figure 13.** RdPSG modélisant la propriété *Guard\_Transition*.

Si, dans l'exemple au-dessus, l'expression booléenne FND était formée de plusieurs conjonctions, alors plusieurs transitions RdPSG seraient connectées aux places *Comp1Primary* et *Comp1Backup*.

## 6. Cas d'étude

Dans ce paragraphe, nous utilisons notre cadre de modélisation pour comparer deux architectures candidates pour un sous-système du système français de Contrôle de Trafic Aérien. Les caractéristiques de ce système sont détaillées dans (Kanoun *et al.* 1999).

Nous présentons d'abord les modèles AADL architecturaux de ces deux architectures candidates dans le paragraphe 6.1. L'analyse des dépendances est présentée dans le paragraphe 6.2. Le paragraphe 6.3 détaille les modèles d'erreur décrivant une partie des dépendances. Le paragraphe 6.4 se focalise sur la transformation de modèle de AADL vers RdPSG et le paragraphe 6.5 présente un exemple de comparaison des deux architectures candidates.

### 6.1. Architectures candidates et leurs modèles en AADL

Le sous-système que nous considérons ici est formé de deux unités logicielles distribuées et tolérantes aux fautes qui s'exécutent sur une architecture bi-processeur. Les deux unités logicielles sont chargées du traitement respectif des plans de vol (PV) et des données provenant des radars (RD). L'unité PV fournit aux contrôleurs les informations relatives aux avions présents dans leur secteur de contrôle. L'unité RD élabore, à partir des données des issues des radars, une image de la situation aérienne. Les unités PV et RD échangent des données afin de corrélérer les plans de vol. Le sous-système doit avoir une disponibilité élevée.

Nous considérons deux architectures candidates, que nous nommons *Configuration1* et *Configuration2*, pour ce sous-système. Les unités PV et RD ont la même structure (présentée déjà dans la Figure 11), c'est-à-dire que chacune de ces deux unités est formée de deux répliques (*PV\_Comp1*, *PV\_Comp2* et *RD\_Comp1*, *RD\_Comp2*) : l'une ayant le rôle primaire (fournisseur de service) et l'autre ayant le rôle secondaire (secours pour le primaire). Les deux architectures candidates utilisent deux processeurs. Chaque réplique d'une unité logicielle s'exécute sur un processeur. Dans la *Configuration1*, les répliques initialement primaires des unités PV et RD (*PV\_Comp1* et *RD\_Comp1*) s'exécutent sur des processeurs différents. (*PV\_Comp1* s'exécute sur *Processor1* et *RD\_Comp1* s'exécute sur *Processor2*). Dans la *Configuration2*, les répliques initialement primaires des unités PV et RD s'exécutent sur le même processeur : *Processor1*. L'ensemble du sous-système a deux modes opérationnels : *Nominal* et *Reconfigured*. Les connexions entre des répliques s'exécutant sur des processeurs différents sont liées à un bus. Par

conséquent, ces liaisons dépendent du mode opérationnel du sous-système. Une défaillance du bus entraîne la défaillance d'une réplique de l'unité RD. La réplique primaire de l'unité PV échange des données avec les deux répliques de l'unité RD.

La Figure 14 présente les modèles des deux architectures candidates en utilisant la notation graphique AADL. Pour des raisons de clarté, nous montrons les liaisons entre les répliques des unités logicielles (fils d'exécution) et les processeurs dans la Figure 14-a et les liaisons entre les connexions et le bus dans la Figure 14-b.

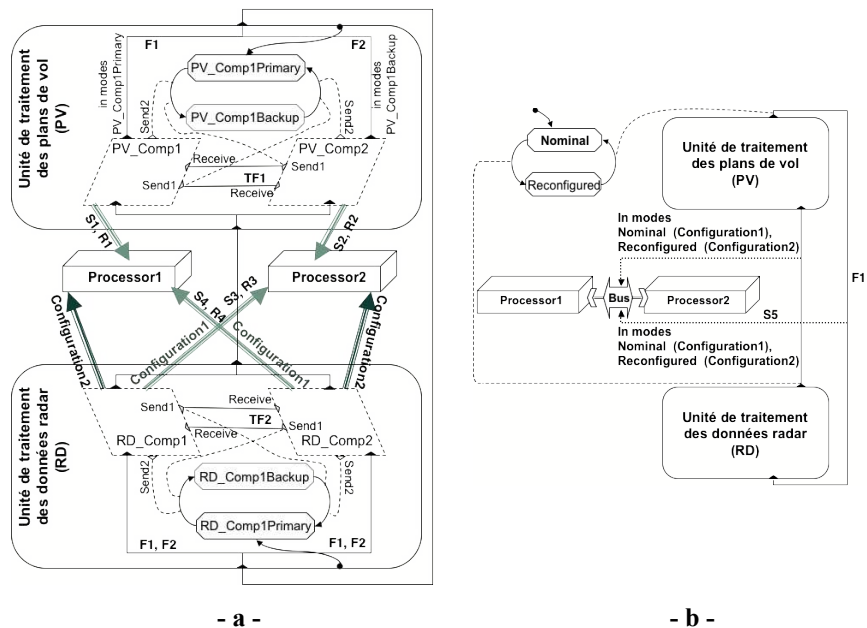


Figure 14. Modèle AADL architectural.

## 6.2. Analyse des dépendances

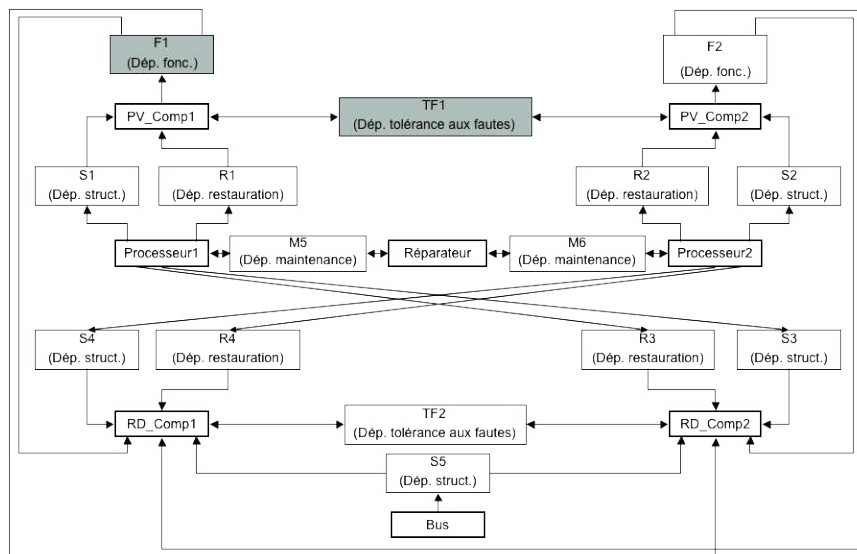
Nous avons pris en compte les dépendances suivantes :

- Dépendance structurelle entre chaque processeur et les fils d'exécution qui s'exécutent au-dessus. Les fautes du matériel peuvent se propager au logiciel qui s'exécute dessus. Ces dépendances (S1-4 dans la Figure 14) résultent des liaisons des fils d'exécutions aux processeurs.
- Dépendance structurelle entre le bus et les répliques de l'unité RD. Si le bus défaille, la connexion rompue liée à ce bus provoque la défaillance de l'unité RD dans le mode opérationnel *Nominal* de la *Configuration1* et dans le mode *Reconfigured* de la *Configuration2*. Cette dépendance (S5 dans la Figure 14) résulte des liaisons des connexions au bus.



- Dépendances fonctionnelles entre l'unité PV et l'unité RD. Le fil d'exécution actif de l'unité PV peut propager des erreurs vers les deux répliques de l'unité RD. Ces dépendances (F1-2 dans la Figure 14) ont comme support les connexions des répliques de l'unité PV vers les répliques de l'unité RD. Nous considérons que les erreurs de l'unité RD ne se propagent pas vers l'unité PV même s'il y a une connexion de RD vers PV.
- Dépendance de maintenance entre deux processeurs qui partagent un réparateur qui n'est pas simultanément disponible pour les deux composants. Cette dépendance n'est pas visible dans la Figure 14.
- Dépendance de restauration entre chaque processeur et les fils d'exécution qui s'exécutent au-dessus. Si un fil d'exécution défaille, il ne peut pas être redémarré si le processeur sur lequel il s'exécute est défaillant. Ces dépendances (R1-4 dans la Figure 14) résultent des liaisons des fils d'exécution aux processeurs.
- Dépendance de tolérance aux fautes entre les répliques des unités PV et RD. Si la réplique primaire défaille mais l'autre réplique est en état de bon fonctionnement, les deux répliques changent de rôle. Ensuite la réplique défaillante est redémarrée. Ces dépendances (TF1-2 dans la Figure 14) ont comme support les connexions entre les répliques de PV et de RD.

La Figure 15 résume les dépendances entre les composants de la *Configuration1*.



**Figure 15.** Diagramme bloc de dépendances pour la *Configuration1*.

Nous avons construit le modèle AADL de sûreté de fonctionnement de manière itérative, en intégrant d'abord les dépendances structurelles et fonctionnelles et

ensuite les dépendances de maintenance, de restauration et de tolérance aux fautes. Le diagramme bloc des dépendances pour la *Configuration2* est similaire. Dans la *Configuration2*, *Processor1* est lié au *RD\_Comp1* par un bloc de dépendance structurelle et un bloc de dépendance liée à la maintenance. *Processor2* est lié au *RD\_Comp2* par les mêmes types de blocs.

Nous illustrons l'approche en détaillant les deux blocs grisés de la Figure 14, F1 et TF1, représentant respectivement la dépendance fonctionnelle entre une réplique de l'unité PV et les deux répliques de l'unité RD et la dépendance de tolérance aux fautes entre les répliques de l'unité PV.

### 6.3. Modèles d'erreur de F1 et TF1

Nous décrivons d'abord les deux dépendances. Ensuite, nous présentons les modèles d'erreur correspondants.

- **F1 : Dépendance fonctionnelle entre un fil d'exécution de l'unité PV et les fils d'exécution de l'unité RD.** Une erreur propagée de la réplique primaire de l'unité PV (*PV\_Comp1* ou *PV\_Comp2*) vers les deux répliques de l'unité RD (*RD\_Comp1* et *RD\_Comp2*) entraîne leur défaillance. Notons que les répliques de l'unité RD ne propagent pas d'erreurs vers les répliques de l'unité PV. De plus, une erreur propagée d'une réplique de l'unité PV n'a pas d'impact sur l'autre réplique de l'unité PV. En d'autres termes, nous ne pouvons pas utiliser le même modèle d'erreur pour les répliques de l'unité PV et de l'unité RD. Le modèle d'erreur associé aux répliques de l'unité RD doit déclarer une propagation *in Error* qui correspond à la propagation *out* déclarée dans le modèle d'erreur associé aux répliques de l'unité PV.
- **TF1 : Dépendance de tolérance aux fautes entre les fils d'exécution de l'unité PV.** Le comportement que nous modélisons est basé sur celui spécifié dans le paragraphe 5.3 (pour des systèmes avec modes opérationnels). En plus de la prise de relais par la réplique secondaire quand la réplique primaire défaille, nous considérons que, si les deux répliques défont, la première redémarrée fournit le service et l'unité PV sera configurée dans le mode opérationnel correspondant. Pour modéliser ce comportement, nous associons des modèles d'erreur aux composants *PV\_Comp1* et *PV\_Comp2* et nous utilisons des propriétés *Guard\_Transition* sur les ports *out Send* des deux répliques. Ces propriétés *Guard\_Transition* sont des extensions de celles présentées dans la Figure 12. La description du comportement en cas de double défaillance se fait en utilisant une notification de la fin de la procédure de redémarrage avant de passer à l'état *Error\_Free*.

La Figure 16 présente le modèle d'erreur associé aux répliques de l'unité PV.

Les lignes f1-f2 correspondent à F1, tandis que les lignes t1-t4 correspondent à TF1. Le composant peut propager des erreurs (propagation *out Error*) mais il ne peut pas être influencé par des propagations d'erreurs car il ne déclare pas une propagation *in Error*. La fin de la procédure de redémarrage est notifiée (propagation *out IAmRestarted*) avant de passer à l'état *Error\_Free*.

```

Error Model Type [pourPV_Comp]

error model pourPV_Comp
features
  Error_Free: initial error state;
  Erroneous: error state;
  Restarted: error state;
  Failed: error state;
  Temp_Fault: error event {Occurrence=> poisson λ1};
  Perm_Fault: error event {Occurrence=> poisson λ2};
  Restart: error event {Occurrence=> poisson μ1};
  Recover: error event {Occurrence=> poisson μ2};
(f1) Error: out error propagation {Occurrence=> fixed p};
(t1) FailedVisible: out error propagation {Occurrence=> fixed 1};
(t2) IAmRestarted: out error propagation {Occurrence=> fixed 1};
end pourPV_Comp;

Error Model Implementation [pourPV_Comp.general]

error model implementation pourPV_Comp.general
transitions
  Error_Free-[Perm_Fault]->Failed;
  Error_Free-[Temp_Fault]->Erroneous;
  Failed-[Restart]->Restarted;
(f2) Erroneous-[out Error]->Erroneous;
(t3) Restarted-[out IAmRestarted]->Error_Free;
(t4) Failed-[out FailedVisible]->Failed;
  Erroneous-[Recover]->Error_Free;
end pourPV_Comp.general;

```

**Figure 16.** *Modèle d'erreur pour PV\_Comp.*

La seule différence entre le modèle d'erreur associé aux répliques de l'unité PV est la direction de la propagation *Error* et la transition AADL qu'elle déclenche. Dans le modèle d'erreur associé aux répliques de l'unité RD, *Error* est une propagation *in* qui déclenche une transition de l'état *Error\_Free* vers l'état *Failed*.

La Figure 17 présente les propriétés *Guard\_Transition* qui spécifient les conditions qui permettent aux transitions entre modes opérationnels d'avoir lieu en tenant compte du comportement de tolérance aux fautes décrit précédemment.

Les transitions entre modes opérationnels ont lieu : 1) si un composant envoie la propagation *FailedVisible* et si en même temps l'autre composant est en bon fonctionnement (état *Error\_Free*) ou 2) si un des composants envoie la propagation

*IAmRestarted* et en même temps l'autre composant n'est pas en bon fonctionnement (signifiant qu'une double défaillance est arrivée et que le premier composant a été redémarré avant le second).

```

Guard_Transition =>
  (Comp1.Send2[FailedVisible] and Comp2.Send1[Error_Free])
  or (Comp2.Send1[IAmRestarted] and not Comp1.Send2[Error_Free])
  applies to Comp1.Send2, Comp2.Send1;
Guard_Transition =>
  (Comp2.Send2[FailedVisible] and Comp1.Send1[Error_Free])
  or (Comp1.Send1[IAmRestarted] and not Comp2.Send2[Error_Free])
  applies to Comp1.Send1, Comp2.Send2;

```

**Figure 17.** Propriétés *Guard\_Transition* associées aux ports *Send* des fils d'exécution de l'unité PV.

#### 6.4. Transformation du modèle AADL vers RdPSG

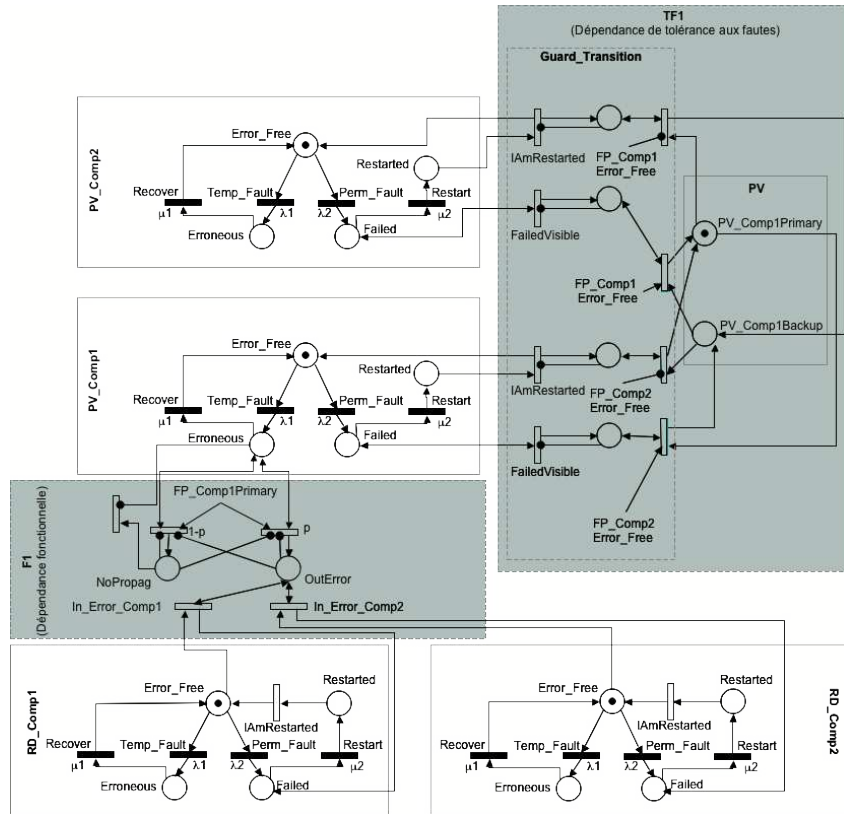
Pour ces deux dépendances, nous utilisons les règles de transformation présentées dans le paragraphe 5. Nous avons d'abord pris en compte la dépendance fonctionnelle F1. Il est à noter que le tir de la transition *Out\_Error* est conditionné par l'existence d'un jeton dans la place *PV\_Comp1Primary* (car uniquement la réplique primaire de l'unité PV peut propager des erreurs). Ensuite nous avons pris en compte la dépendance de tolérance aux fautes TF1. La partie grisée de la Figure 18 présente la partie du RdPSG qui correspond aux deux dépendances mentionnées ci-dessus. Le reste de la figure représente les sous-réseaux correspondant aux composants *RD\_Comp1*, *RD\_Comp2*, *PV\_Comp1* et *PV\_Comp2*. Pour des raisons de clarté, nous n'avons pas représenté les transitions qui vident les places correspondant aux propagations *out*.

#### 6.5. Evaluation de mesures quantitatives

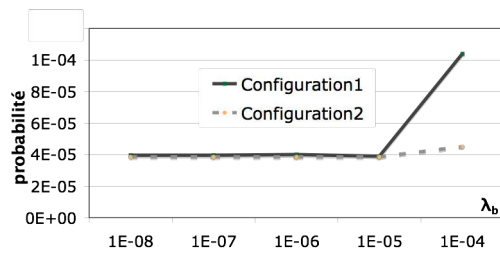
A partir de la chaîne de Markov sous-jacente au RdPSG intégrant l'ensemble des dépendances, on peut évaluer différentes mesures pour comparer la sûreté de fonctionnement des architectures candidates considérées. A titre d'exemple, la Figure 19 donne les indisponibilités des deux architectures candidates, évaluées par l'outil Surf-2.

Dans la Figure 19 le paramètre qui varie est le taux d'occurrence de la défaillance du bus,  $\lambda_b$ .  $\lambda_b \leq 10^{-6}/h$  correspond à un bus redondant. Pour la *Configuration1*, l'impact de ce paramètre est important quand  $\lambda_b \geq 10^{-5}/h$ . La *Configuration2* est beaucoup moins influencée par  $\lambda_b$ , car en mode opérationnel *Nominal*, la communication entre les deux unités logicielles ne passe pas à travers le

bus. La Figure 19 montre que, d'un point de vue pratique, si  $\lambda_b \geq 10^{-5}/h$ , la *Configuration2* est recommandée. Dans le cas contraire ( $\lambda_b < 10^{-5}/h$ ), les deux architectures candidates sont équivalentes du point de vue de leurs indisponibilités.



**Figure 18.** RdPSG du sous-système du système informatique français de trafic aérien – deux dépendances.



**Figure 19.** Indisponibilité

## 7. Conclusion et perspectives

Cet article a présenté une approche itérative pour la modélisation de la sûreté de fonctionnement de systèmes informatiques en utilisant le langage AADL comme point de départ et les RdPSG comme formalisme intermédiaire. L'objectif de cette approche est de masquer la complexité des modèles analytiques traditionnels aux utilisateurs qui sont familiarisés avec AADL et qui n'ont pas de connaissances approfondies concernant ces modèles analytiques. Ainsi, nous leur facilitons l'obtention des mesures de sûreté de fonctionnement.

Notre approche vise à assister l'utilisateur dans la construction structurée du modèle AADL de sûreté de fonctionnement. Ce dernier est transformé en un RdPSG qui peut être traité par des outils existants. Pour faciliter l'évolution du modèle, nous proposons que le modèle AADL de sûreté de fonctionnement soit construit de manière itérative, en modélisant progressivement les dépendances entre composants.

La transformation du modèle AADL en RdPSG est conçue pour être transparente pour l'utilisateur. Par conséquent, elle est basée sur des règles de description des dépendances dans le modèle AADL et sur des règles systématiques de transformation de modèle, destinées à une mise en œuvre automatique. Nous avons implémenté un outil, ADAPT (*from AADL Architectural models to Petri nets through model Transformation*) (Rugina *et al.* 2008), mettant en œuvre nos règles de transformation.

Dans cet article, nous avons montré les principes de la transformation et une partie des règles. La transformation peut être effectuée de manière itérative, à chaque fois que le modèle AADL de sûreté de fonctionnement est enrichi. Ainsi, le RdPSG peut être validé progressivement. Par conséquent, le modèle AADL correspondant peut être aussi validé progressivement.

Nous avons illustré l'approche proposée sur un sous-système du système informatique français de contrôle de trafic aérien.

Plusieurs directions peuvent être explorées afin d'étendre les travaux présentés dans cet article. Dans un premier temps, il serait intéressant d'appliquer notre cadre de modélisation à des cas d'étude complexes issus de différents domaines d'application, ce qui permettrait d'étudier son adéquation dans divers contextes. Une deuxième direction devrait être dédiée à l'étude de la mise à l'échelle de notre transformation de modèle. En effet, nous avons identifié une limitation liée à la croissance exponentielle de la taille du RdPSG avec la variation du nombre de composants recevant des propagations. Actuellement, il est possible d'appliquer des méthodes de réduction afin de rendre le RdPSG compact. Toutefois, la génération d'un RdPSG qui n'est pas compact peut s'avérer difficile pour des grands systèmes. Une piste à explorer est la recherche d'un algorithme de transformation basé sur un parcours initial du modèle et sur l'identification de composants identiques et de comportements équivalents. Enfin, dans nos travaux, nous avons choisi de générer des RdPSG à partir de modèles AADL afin de permettre d'obtenir des mesures

quantitatives de sûreté de fonctionnement. En effet, les RdPSG sont couramment utilisés pour les analyses de sûreté de fonctionnement et nous disposons d'un outil basé sur les RdPSG pour l'évaluation des mesures de sûreté de fonctionnement. Il est tout à fait aussi intéressant de considérer d'autres langages cibles, comme par exemple les algèbres de processus qui reçoivent une attention croissante dans la communauté de la sûreté de fonctionnement et qui semblent particulièrement adaptés aux approches à base de composants, ceci grâce à leur modularité intrinsèque.

Ce travail a été partiellement financé par 1) la Commission Européenne (projet européen intégré ASSERT No. IST 004033 – [www.assert-online.net](http://www.assert-online.net) et réseau d'excellence ReSIST No. IST 026764 – [www.laas.fr/RESIST/](http://www.laas.fr/RESIST/)), 2) le Fond Social Européen et 3) le Zonta International.

## 8. Bibliographie

- C. Béounes, M. Aguéra, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. M. d. Souza, D. Powell and P. Speisser, "Surf-2: a program for dependability evaluation of complex hardware and software systems", 23rd IEEE Int. Symposium on Fault Tolerant Computing, (Toulouse, France), pp.668-673, 1993.
- S. Bernardi, S. Donatelli and J. Merseguer, "From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models", 3rd Int. Workshop on Software and Performance, (Rome, Italie), pp.35-45, ACM Press, 2002.
- A. Bondavalli, M. D. Cin, D. Latella, I. Majzik, A. Pataricza and G. Savoia, "Dependability Analysis in the Early Phases of UML Based System Design", International Journal of Computer Systems - Science & Engineering, 16 (5), pp.265-275, 2001
- D. Chen, L.-O. Bertsson, H. Lönn and M. Törngren, Advancing Traffic Efficiency and Safety through Software Technology (ATESST) Deliverable D.2.2.1 - Elicitation of Representative and Relevant Analysis and V&V Techniques Contract Nb. 2004-026976, 2007, [http://www.atesst.org/home/liblocal/docs/ATESST Deliverable D2.2.1 v1.0.pdf](http://www.atesst.org/home/liblocal/docs/ATESST%20Deliverable%20D2.2.1%20v1.0.pdf).
- V. Debruyne, F. Simonot-Lion and Y. Trinquet, "EAST-ADL - An Architecture Description Language", in 18th IFIP World Computer Congress, ADL Workshop, (Toulouse, France), pp.53-62, 2004.
- J.-M. Farines, B. Berthomieu, J.-B. Bodeveix, P. Dissaux, P. Farail, M. Filali, P. Gauffillet, H. Hafidi, J.-L. Lambert, P. Michel and F. Vernadat, "The Cotre project: rigorous software development for real time systems in avionics", 27th IFAC/IFIP/IEEE Workshop on Real Time Programming, (Zielona Gora, Pologne), 2003.
- P. H. Feiler, D. P. Gluch, J. J. Hudak and B. A. Lewis, "Pattern-Based Analysis of an Embedded Real-time System Architecture", 18th IFIP World Computer Congress, ADL Workshop, (Toulouse, France), pp.83-91, 2004.

- J. Fernandez Briones, M. de Miguel, J. P. Silva and A. Alonso, "Integration of Safety Analysis and Software Development Methods", 1st Int. Conf. on System Safety Engineering, (Londres, Royaume Uni), pp.275-284, 2006.
- J. Hugues, F. Kordon, L. Pautet and T. Vergnaud, "A Factory To Design and Build Tailorable and Verifiable Middleware", in Workshop on Networked Systems: Realization of Reliable Systems on Top of Unreliable Networked Platforms (Monterey Workshop Series, 12th edition, 2005) 4322, LNCS, pp.123-144, Springer-Verlag, 2007.
- A. Joshi, S. Vestal and P. Binns, "Automatic Generation of Static Fault Trees", Workshop on Architecting Dependable Systems of The 37th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks, (Edinburgh, Royaume Uni), pp. 172-177, 2007.
- K. Kanoun and M. Borrel, "Fault-tolerant systems dependability. Explicit modeling of hardware and software component-interactions", IEEE Transactions on Reliability, 49 (4), pp.363-376, 2000.
- K. Kanoun, M. Borrel, T. Morteveille and A. Peytavin, "Availability of CAUTRA, a Subset of the French Air Traffic Control System", IEEE Transactions on Computers, 48 (5), pp.528-535, 1999.
- L. Kloul and J. Kuster-Filipe, "Modelling Mobility with UML2.0 and PEPA Nets", Int. Conf. on Application of Concurrency to System Design, (Turku, Finlande), pp.153-164, 2006.
- J.-C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillemin, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac and P. Thévenod, Guide de la Sûreté de Fonctionnement, Cepaduès Editions, 1996.
- J. P. López-Grao, J. Merseguer and J. Campos, "From UML Activity Diagrams To Stochastic Petri Nets: Application to Software Performance Engineering", 4th Int. Workshop on Software and Performance, (Redwood City, CA, Etats-Unis), pp.25-36, 2004.
- OMG, Unified Modelling Language Specification: v 2.0, <http://www.omg.org>, Octobre 2004.
- G. J. Pai and J. Bechta Dugan, "Automatic Synthesis of Dynamic Fault Trees from UML System Models", 13th International Symposium on Software Reliability Engineering (ISSRE'02), (Annapolis, Etats-Unis), pp.243-254, 2002.
- A. E. Rugina, Modélisation et Evaluation de la Sûreté de Fonctionnement - De AADL vers les Réseaux de Petri Stochastiques, Doctorat, Institut National Polytechnique de Toulouse, Novembre 2007, <http://tel.archives-ouvertes.fr/tel-00207502/fr/>.
- A. E. Rugina, K. Kanoun and M. Kaâniche, "An Architecture-based Dependability Modeling Framework using AADL", 10th IASTED Int. Conf. on Software Engineering and Applications, (Dallas, Etats-Unis), pp.222-227, 2006a.
- A. E. Rugina, K. Kanoun and M. Kaâniche, "Modélisation de la sûreté de fonctionnement à partir du langage AADL", 15ème Congrès de Maîtrise des Risques et de Sûreté de Fonctionnement, (Lille, France), vol. 2, pp.294-300, 2006b.
- A. E. Rugina, K. Kanoun and M. Kaâniche, "The ADAPT Tool: From AADL Architectural Models to Stochastic Petri Nets through Model Transformation", à paraître dans 7<sup>th</sup> European Dependable Computing Conference (EDCC-7), Kaunas, Lituanie, Mai 2008.



- SAE-AS5506, SAE Architecture Analysis and Design Language (AADL), International Society of Automotive Engineers, Novembre 2004.
- SAE-AS5506/1, SAE Architecture Analysis and Design Language (AADL) Annex Volume 1, Annex E: Error Model Annex, International Society of Automotive Engineers, Juin 2006.
- F. Singhoff, J. Legrand, L. Nana and L. Marcé, "Scheduling and Memory Requirements Analysis with AADL", SIGAda Int. Conf. on Ada, (Atlanta, Etats-Unis), pp.1-10, 2005.
- O. Sokolsky, I. Lee and D. Clarke, "Schedulability Analysis of AADL Models", 20th Parallel and Distributed Processing Symposium, (Rhodes, Grèce), 2006.

Article reçu le  
Article accepté le

*Ana-Elena Rugina a terminé son doctorat en systèmes informatiques sûrs de fonctionnement au LAAS-CNRS. Ses travaux de thèse ont porté sur la modélisation et l'évaluation de la sûreté de fonctionnement dans un contexte d'ingénierie basée sur des modèles AADL. Ana-Elena Rugina travaille actuellement à EADS ASTRIUM Satellites.*

*Karama Kanoun est Directeur de recherche au CNRS, responsable du groupe "Tolérance aux fautes et sûreté de fonctionnement informatique", au LAAS. Ses travaux de recherche ont trait à l'évaluation de la sûreté de fonctionnement, tant du matériel que du logiciel, domaines dans lesquels elle est l'auteur d'environ 150 publications dans des revues et conférences nationales et internationales.*

*Mohamed Kaâniche est Chargé de recherche au CNRS, membre du groupe "Tolérance aux fautes et sûreté de fonctionnement informatique", au LAAS. Ses recherches portent sur la modélisation analytique et l'évaluation expérimentale de la sûreté de fonctionnement et de la sécurité de systèmes informatiques, vis-à-vis de fautes accidentelles et de malveillances.*