



**HAL**  
open science

# Modélisation et évaluation de la sûreté de fonctionnement - De AADL vers les réseaux de Pétri stochastiques

Ana-Elena Rugina

► **To cite this version:**

Ana-Elena Rugina. Modélisation et évaluation de la sûreté de fonctionnement - De AADL vers les réseaux de Pétri stochastiques. Networking and Internet Architecture [cs.NI]. Institut National Polytechnique de Toulouse - INPT, 2007. English. NNT: . tel-00207502

**HAL Id: tel-00207502**

**<https://theses.hal.science/tel-00207502>**

Submitted on 17 Jan 2008

**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.

INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE  
LAAS CNRS

**THESE**

en vue de l'obtention du

**DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE,  
délivré par l'Institut National Polytechnique de Toulouse**

***Discipline : Systèmes Informatiques***

présentée et soutenue

par

Ana-Elena Rugina

le 19 novembre 2007

**Titre :**

Modélisation et évaluation de la sûreté de fonctionnement –  
De AADL vers les réseaux de Petri stochastiques

Dependability modeling and evaluation –  
From AADL to stochastic Petri nets

---

**Directeur de thèse: Mme. K. Kanoun**

---

**JURY**

|      |                    |              |
|------|--------------------|--------------|
| M.   | P. Feiler          | , Président  |
| Mme. | F. di Giandomenico | , Rapporteur |
| M.   | A. van Moorsel     | , Rapporteur |
| M.   | M. Kaâniche        | , Examineur  |
| M.   | C. Lemercier       | , Examineur  |



*To all those who supported me.*



*Every day you may make progress. Every step may be fruitful. Yet there will stretch out before you an ever-lengthening, ever-ascending, ever-improving path. You know you will never get to the end of the journey. But this, so far from discouraging, only adds to the joy and glory of the climb.*

*Sir Winston Churchill*



# Acknowledgements

This work has been carried out at the Laboratory of Analysis and Architecture of Systems of the French National Research Center (LAAS-CNRS). I wish to express my gratitude to the successive directors of the LAAS-CNRS, Mr. Malik Ghallab and Mr. Raja Chatila, for the facilities provided in the laboratory.

I also thank Mr. Jean Arlat, CNRS Research Director and head of the Dependable Computing and Fault Tolerance research group (TSF), for having allowed me to carry out this work in this reach research environment.

This thesis has been financed by a scholarship from the European Social Fund and has been partially conducted in the context of (1) the ASSERT European Project (Automated Proof-Based System and Software Engineering for Real-Time Applications) (2) the ReSIST Network of Excellence (Resilience for Survivability in IST).

I would like to thank all committee members, for having attentively read my dissertation:

- Peter Feiler, senior researcher at the Carnegie Mellon Software Engineering Institute (Pittsburgh, USA), who honored me by chairing this committee.
- Felicita di Giandomenico, senior researcher at “Istituto di Elaborazione dell’Informazione” – CNR (Pisa, Italy).
- Aad van Moorsel, reader at the University of Newcastle upon Tyne, (UK).
- Karama Kanoun, senior CNRS researcher.
- Mohamed Kaâniche, CNRS researcher.
- Christophe Lemercier, head of departement “Data Processing & On-board Software & Dependability” of ASTRIUM Satellites, (Toulouse).

My special thanks go to Mrs. Felicita di Giandomenico and Mr. Aad van Moorsel, who accepted the charge of being “rapporteurs”. I wish to thank them for their relevant comments.

I am most grateful to my supervisors, Mrs. Karama Kanoun and Mr. Mohamed Kaâniche, for their passion, technical and human advice, for having devoted me an important amount of their time and for the evenings spent at LAAS before my oral defense.

I warmly thank all members of the Dependable Computing and Fault Tolerance research group: permanent researchers, PhD students and interns. I am very grateful to them for their precious advice, support and friendship. I have much appreciated the family-like atmosphere and the open discussions on different topics. My experience in this group has been extremely enriching. I am very pleased to mention here my officemates (Eric, Géraldine, Ossama, Carlos and Magnos) with whom I shared unforgettable moments. I also think of previous PhD students (Ali, Taha, Cristina, Eric M., Guillaume, Nicolas), who offered me unconditional support during my moments of doubt at the beginning of my experience in LAAS-CNRS. I thank Marilena Bruffa, who contributed directly to the achievement of this work, and Eric Marsden, whose attentive reading has contributed to the improvement of my dissertation. My thanks also go to Gina, for her availability, kindness and precious help in organizing the defense.

I wish to extend my thanks to all members of the service departments of LAAS-CNRS (“Informatique et Instrumentation, Documentation, Magasin, Entretien, Direction–Gestion,



Réception–Standard, Communication”), who always allowed me to work in the best conditions.

I would also like to acknowledge here the Zonta International and the members of the Zonta club of Muret (France), for having supported my application for the Amelia Earhart fellowship. I am also very grateful to Peter Feiler (Carnegie Mellon Software Engineering Institute), Eric Conquet (European Space Agency), Bruce Lewis (US Army) and Jean Arlat (LAAS-CNRS), for their support in this context.

During my PhD, I have spent six weeks at the Carnegie Mellon Software Engineering Institute in the “Performace Critical Systems” team. I thank Peter Feiler and all the team for welcoming me. I learned many lessons, both technical and human during my stay in Pittsburgh. I am grateful to Madelaine Dusseau for the support provided in the many administrative tasks associated with this stay.

I am particularly thankful to Jean-Paul Blanquart and Dave Thomas (from ASTRIUM Satellites) for the enriching exchanges that we had along my PhD. Their thoughts helped me to mature my ideas.

In these important moments of my life, I think of my family and friends. I owe them my sincerest gratitude for their support and confidence. Finally, my warmest thanks go to Florin, who shared everything with me during the last years. I thank him for his understanding, his patience and for having supported my moods, sometimes anxious lately.

# Remerciements

Les travaux présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS. Je remercie Messieurs Malik Ghallab et Raja Chatila, qui ont assuré la direction du LAAS-CNRS depuis mon entrée, de m'avoir accueilli au sein de ce laboratoire.

Je remercie également Monsieur Jean Arlat, Directeur de Recherche au CNRS, responsable du groupe de recherche Tolérance aux fautes et Sécurité de Fonctionnement informatique (TSF), de m'avoir permis de réaliser ces travaux dans ce groupe.

Les travaux développés dans cette thèse ont été financés par une bourse du Fond Social Européen et effectués partiellement dans le cadre du projet européen ASSERT (Automated Proof-Based System and Software Engineering for Real-Time Applications) et du réseau d'excellence ReSIST (Resilience for Survivability in IST).

Je tiens à remercier tous les membres du jury, pour leur lecture attentive du mémoire :

- Peter Feiler, chercheur au « Carnegie Mellon Software Engineering Institute » (Pittsburgh, Etats-Unis), qui m'a fait l'honneur de présider ce jury.
- Felicita di Giandomenico, chercheur à « Istituto di Elaborazione dell'Informazione » – CNR (Pise, Italie).
- Aad van Moorsel, professeur à « University of Newcastle upon Tyne » (Royaume-Uni).
- Karama Kanoun, Directrice de Recherche au CNRS.
- Mohamed Kaâniche, Chargé de Recherche au CNRS.
- Christophe Lemercier, directeur du département « Data Processing & On-board Software & Dependability » de ASTRIUM Satellites, (Toulouse).

Je remercie tout particulièrement Madame Felicita di Giandomenico et Monsieur Aad van Moorsel, qui ont accepté la lourde tâche de rapporteur. Je les remercie vivement pour leurs commentaires avisés.

J'exprime ma profonde reconnaissance à Madame Karama Kanoun (Directrice de Recherche au CNRS) et Monsieur Mohamed Kaâniche (Chargé de Recherche au CNRS), pour avoir dirigé mes travaux de thèse, pour leur passion, leurs conseils tant sur le plan technique que humain, pour m'avoir consacré une partie importante de leur temps et notamment pour les soirées passées au LAAS avant ma soutenance.

Je remercie sincèrement tous les membres du groupe TSF, permanents, doctorants et stagiaires. Je leur suis très reconnaissante pour leurs conseils, support et amitié. J'ai beaucoup apprécié l'ambiance « familiale » et les discussions très ouvertes sur divers thèmes. Mon expérience au sein du groupe a été extrêmement enrichissante. Il m'est particulièrement agréable de remercier mes collègues de bureau (Eric, Géraldine, Ossama, Carlos et Magnus) avec qui j'ai partagé des moments inoubliables. Ma pensée va également vers les doctorants plus anciens (Ali, Taha, Cristina, Eric M., Guillaume, Nicolas), qui m'ont offert leur support désintéressé dans les moments de doute de mes débuts au LAAS-CNRS. Je tiens à remercier Marilena Bruffa, qui a contribué directement à l'aboutissement de ces travaux, et Eric Marsden pour sa lecture attentive, qui m'a permis d'améliorer la qualité de ce mémoire. Mes

remerciements s'adressent également à Gina pour sa disponibilité, sa gentillesse et son aide précieuse dans les derniers préparatifs pour la soutenance.

Je remercie également tous les membres des services du LAAS-CNRS (Informatique et Instrumentation, Documentation, Magasin, Entretien, Direction-Gestion, Réception-Standard, Communication) qui m'ont toujours permis de travailler dans d'excellentes conditions.

Je tiens à exprimer ma gratitude envers le Zonta International et tout particulièrement envers les membres du club Zonta de Muret, qui ont soutenu ma candidature pour la bourse Amelia Earhart. Je suis également très reconnaissante à Peter Feiler (Carnegie Mellon Software Engineering Institute), Eric Conquet (Agence Spatiale Européenne), Bruce Lewis (US Army) et Jean Arlat (LAAS-CNRS), pour leur support dans cette entreprise.

Mes remerciements vont également à Peter Feiler et à l'équipe « Performance-Critical Systems » du Carnegie Mellon Software Engineering Institute, pour m'avoir accueilli pendant six semaines. J'ai tiré de nombreux enseignements, à la fois sur le plan technique et humain, de mon séjour à Pittsburgh. Je remercie Madelaine Dusseau pour le support fourni dans les nombreuses tâches administratives liées à ce séjour.

Je remercie sincèrement Jean-Paul Blanquart et Dave Thomas (ASTRIUM Satellites) pour les échanges enrichissants que nous avons eus au long de ma thèse. Leurs réflexions m'ont aidé à mûrir mes idées et à prendre du recul.

Dans ces moments très importants de ma vie, je pense très fort à ma famille et mes amis, pour leur soutien et encouragement permanent. Enfin, je remercie Florin, qui a tout partagé avec moi ces dernières années, pour sa compréhension, sa patience et pour avoir supporté mes humeurs, parfois angoissés ces derniers temps.

# RESUME

## 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 questions fondamentales quand il s'agit d'évaluer la sûreté de fonctionnement :

- 1) Comment prendre en compte dans les modèles AADL les dépendances multiples entre les composants dans la description du comportement du système en présence de fautes ? Ces dépendances peuvent être engendrées par l'architecture du système ou les stratégies de tolérance aux fautes et de maintenance.
- 2) Comment obtenir des mesures de sûreté de fonctionnement à partir des modèles AADL ?

À l'état actuel, il n'existe pas de méthodologie pour aider les concepteurs utilisant AADL à résoudre ces deux questions. L'objectif de nos travaux est de répondre à ces deux besoins.

Nous répondons au premier besoin en guidant l'élaboration du modèle AADL de sûreté de fonctionnement par une méthode itérative, qui prend en compte progressivement les dépendances entre les composants. Pour ce faire, nous avons identifié toutes les primitives du langage AADL qui servent à la description des dépendances liées à la sûreté de fonctionnement et nous avons défini des règles de modélisation pour chaque type de dépendance. Nous définissons un *élément* de AADL comme un ensemble de primitives décrivant une dépendance. Notre méthode permet à la fois de maîtriser la complexité des modèles et de les valider progressivement. Afin de faciliter la réutilisation, nous définissons également un ensemble de sous-modèles génériques et réutilisables décrivant des architectures tolérantes aux fautes. 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>).

Nous répondons au deuxième besoin 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. La transformation 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 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 stochastic 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. Dans ce document, nous nous focalisons sur les principes généraux de la transformation et nous montrons les règles de transformation les plus représentatives.

Enfin, nous illustrons à la fois l'approche itérative de modélisation et la transformation de modèle sur un cas d'étude issu d'un système réel : le système informatique français de contrôle de trafic aérien.

Il est à noter que nos travaux autour de la modélisation de la sûreté de fonctionnement avec AADL nous ont permis de proposer des évolutions du standard AADL. Une partie de ces propositions ont déjà été intégrées dans la version actuelle du standard. Les autres seront prises en compte dans les discussions du comité de standardisation visant la prochaine version de ce standard.

Des présentations des principes de notre méthodologie de modélisation ont été effectuées dans [Rugina *et al.* 2006b], [Rugina *et al.* 2006c] et [Rugina *et al.* 2007].

En suivant le plan de la thèse, la suite de ce résumé est organisée en cinq paragraphes. Le paragraphe I discute de l'état de l'art. Le paragraphe II présente les éléments qui existent dans la version courante du standard AADL et qui constituent la base de notre approche de modélisation. Le paragraphe III est une vue d'ensemble de notre approche itérative de modélisation basée sur AADL. L'ensemble de sous-modèles réutilisables que nous avons défini pour des architectures classiques tolérantes aux fautes n'est pas détaillé ici. Cependant, il fait l'objet de la seconde partie du Chapitre 3 de la thèse. Le paragraphe IV est dédié aux règles de transformation de AADL vers RdPSG. Nous montrons ici quelques principes et règles de transformation. Le paragraphe V illustre

---

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

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

notre approche sur un sous-système du Système Français de Contrôle de Trafic Aérien. Nous finissons en donnant les conclusions et les perspectives de ces travaux.

## I Quelques 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] propose 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] proposent des algorithmes d'obtention d'arbres de fautes à partir de UML. D'autres approches 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 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 que nous proposons 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. 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.

## II Éléments 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), appels de sous-programmes et 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 composants et connexions 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

---

<sup>3</sup> Dans la suite, 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.

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.

|  |
|--|
| <b>Error Model Type [independent]</b>  |
| <pre> error model independent features   Error_Free: initial error state;   Erroneous: error state;   Failed: error state;   Temp_Fault: error event {Occurrence =&gt; poisson λ1};   Perm_Fault: error event {Occurrence =&gt; poisson λ2};   Restart: error event {Occurrence =&gt; poisson μ1};   Recover: error event {Occurrence =&gt; poisson μ2}; end independent; </pre> |
| <b>Error Model Implementation [independent.general]</b>  |
| <pre> error model implementation independent.general transitions   Error_Free-[Perm_Fault]-&gt;Failed;   Error_Free-[Temp_Fault]-&gt;Erroneous;   Failed-[Restart]-&gt;Error_Free;   Erroneous-[Recover]-&gt;Error_Free; end independent.general; </pre>   |

**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 ce document, plusieurs exemples illustreront l'usage des propagations (voir par exemple § III.2.1) et des propriétés **Guard\_Transition** (voir par exemple § IV.3.1).

### III 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 § I, 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 le modèle AADL associé.

Dans la suite, le paragraphe III.1 donne une vue d'ensemble de notre approche et le paragraphe III.2 illustre la modélisation des dépendances.

### **III.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*.

---

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

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.

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. 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 ici.

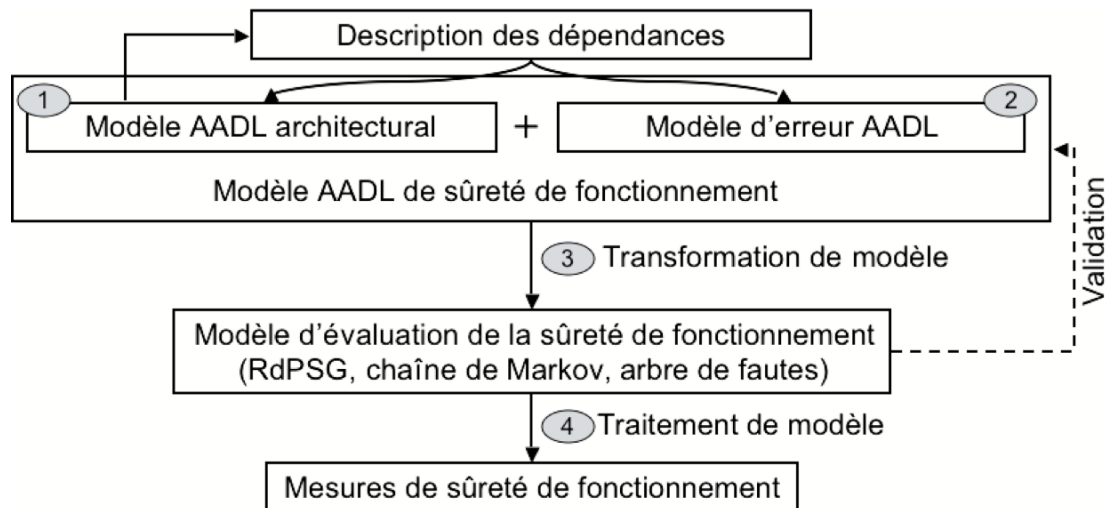


Figure 2. Approche proposée

## III.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.

### III.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.

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 *Error\_Free* à *Failed*.

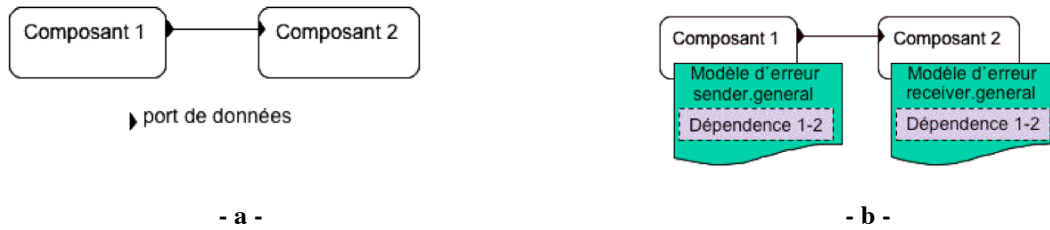


Figure 3. Dépendance architecturale.

Quand le *Composant 1* est dans l'état erroné (*Erroneous*), il envoie une propagation par la connexion unidirectionnelle et reste dans le même état. 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 *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 [sender.general]</b>  error model implementation sender.general transitions   Error_Free-[Perm_Fault]     -&gt;Failed;   Error_Free-[Temp_Fault]     -&gt;Erroneous;   Failed-[Restart]-&gt;Error_Free;   Erroneous-[Recover]     -&gt;Error_Free; (d2) Erroneous-[out Error]     -&gt;Erroneous; end sender.general; </pre>  | <pre> <b>Error Model Implementation [receiver.general]</b>  error model implementation receiver.general transitions   Error_Free-[Perm_Fault]     -&gt;Failed;   Error_Free-[Temp_Fault]     -&gt;Erroneous;   Failed-[Restart]-&gt;Error_Free;   Erroneous-[Recover]     -&gt;Error_Free; (d2) Error_Free-[in Error]     -&gt;Failed; end receiver.general; </pre>  |

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.

### III.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*.

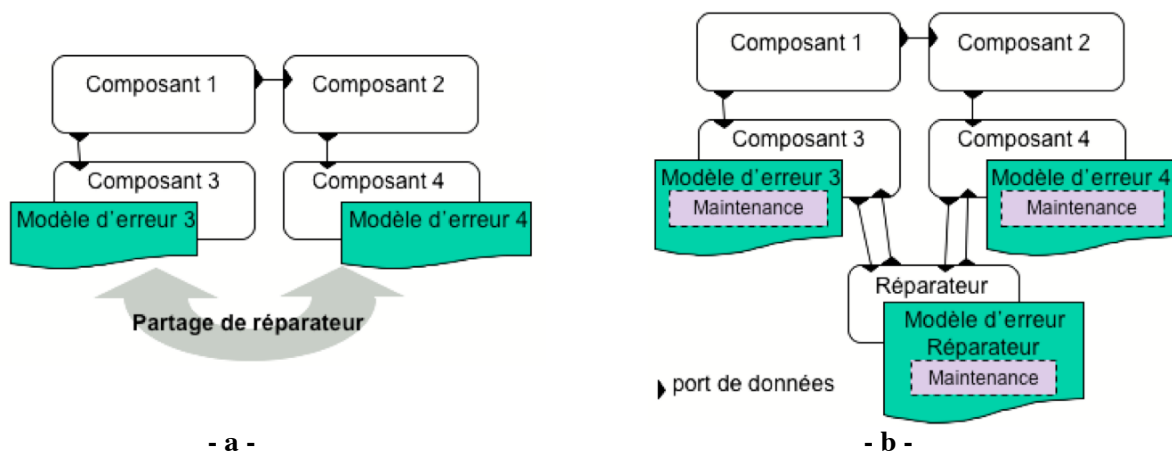


Figure 5. Dépendance de maintenance.

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.

### III.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 défectueux. 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.

## IV 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 § III. 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 la thèse (voir le Chapitre 4). Cet ensemble a été mis en oeuvre dans un outil, ADAPT (*from AADL Architectural models to stochastic Petri nets through model Transformation*).

### IV.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.

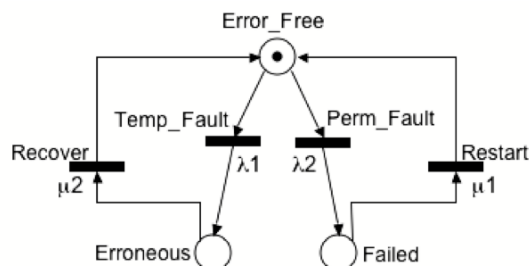


Figure 6. RdPSG correspondant au modèle d'erreur de la Figure 1.

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

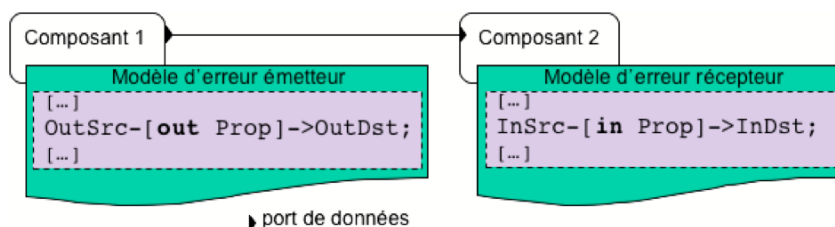
| Primitive du modèle d'erreur                           | Élément du RdPSG   |              |
|--|--|--------------|
| Etat   | Place  | ○            |
| Etat initial   | Place avec jeton   | ●            |
| Événement  | Transition RdPSG (temporisée ou immédiate)   | ⏏            |
| Propriété d'Occurrence <sup>5</sup>                    | Poids de la transition RdPSG (taux pour la distribution de Poisson ou probabilité fixe)                                    | ⏏ Temporisée |
|  |  | ⏏ Immédiate  |
| Transition AADL<br>(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) |              |

## IV.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 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*.



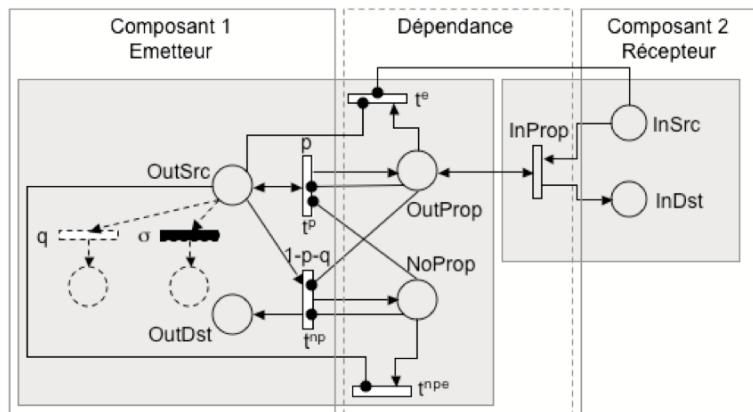
**Figure 7. Emetteur et récepteur – propagations avec noms correspondants.**

<sup>5</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.

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 existe 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 la thèse (voir le Chapitre 4).

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 ( $N_{tr}$ ) 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 :

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

- 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.

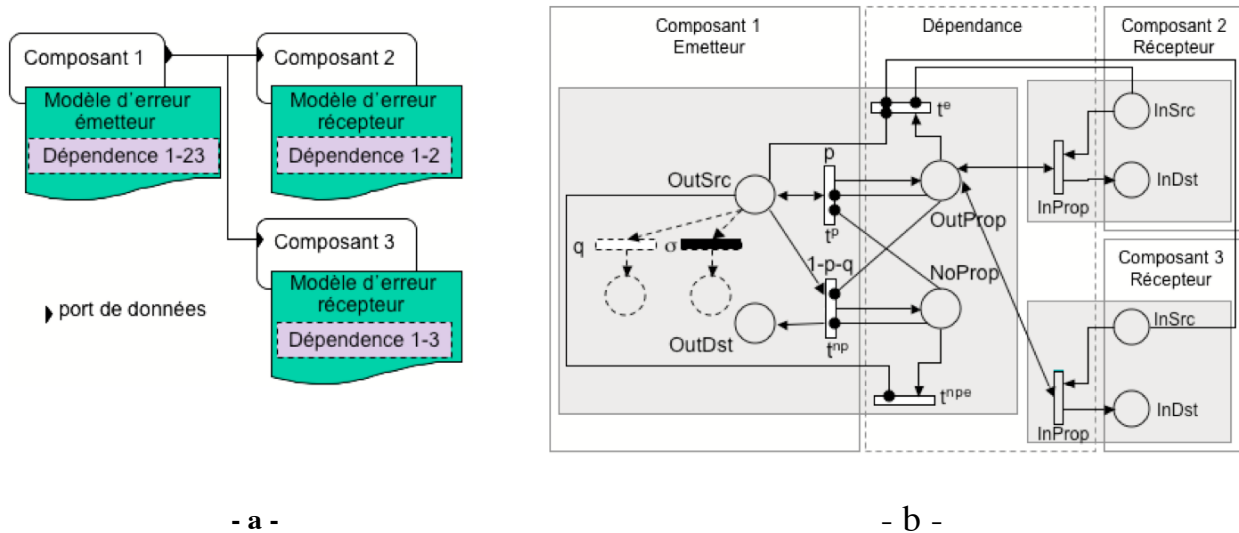


Figure 9. Propagation d'un émetteur vers deux récepteurs.

### IV.3 Systèmes avec modes opérationnels

AADL offre plusieurs mécanismes pour connecter les états logiques du modèle d'erreur aux transitions entre des modes opérationnels. Dans ce paragraphe, nous nous focalisons sur les règles de transformation pour des propriétés **Guard\_Transition**. Les autres règles sont détaillées dans la thèse (voir le Chapitre 4). Les propriétés **Guard\_Transition** 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.

#### IV.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.

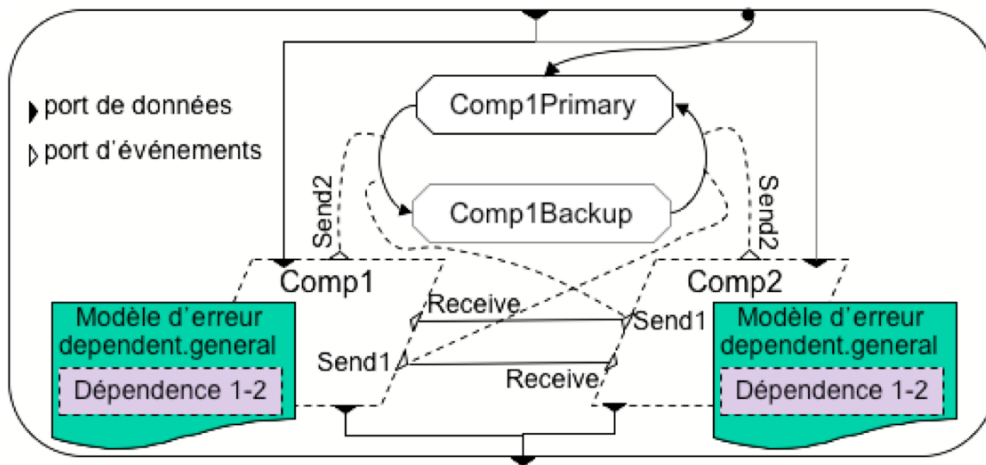


Figure 11. Modèle AADL d'un système avec modes opérationnels.

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 g4-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**.

### IV.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 §IV.3.1. La Figure 13 montre le RdPSG correspondant à la première propriété **Guard\_Transition** (lignes g1-g3) de la Figure 12.

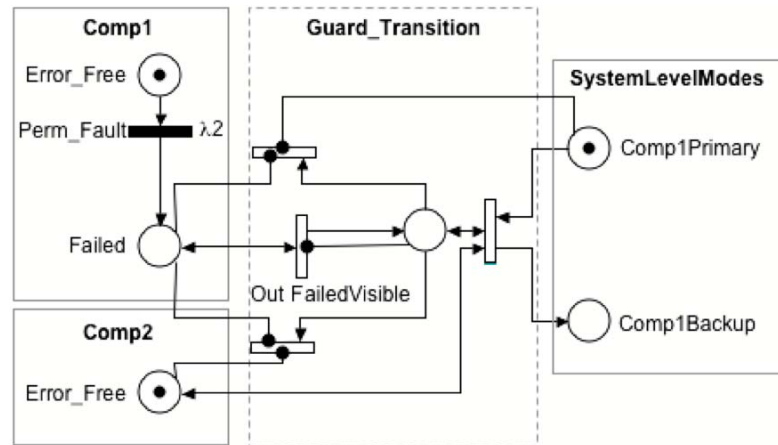


Figure 13. RdPSG modélisant la propriété **Guard\_Transition**.

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*.

## V 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. Ce système est plus détaillé dans [Kanoun *et al.* 1999]<sup>6</sup>.

Nous présentons d'abord les modèles AADL architecturaux de ces deux architectures candidates dans le paragraphe V.1. L'analyse des dépendances est présentée dans le paragraphe V.2. Le paragraphe V.3 détaille les modèles d'erreur décrivant une partie des dépendances. Le paragraphe V.4 se focalise sur la transformation de modèle de AADL vers RdPSG et le paragraphe V.5 présente un exemple de comparaison des deux architectures candidates.

### V.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 corréliser 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

<sup>6</sup> L'article cité n'aborde ni la modélisation en AADL du système considéré, ni la transformation du modèle AADL vers RdPSG.

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.

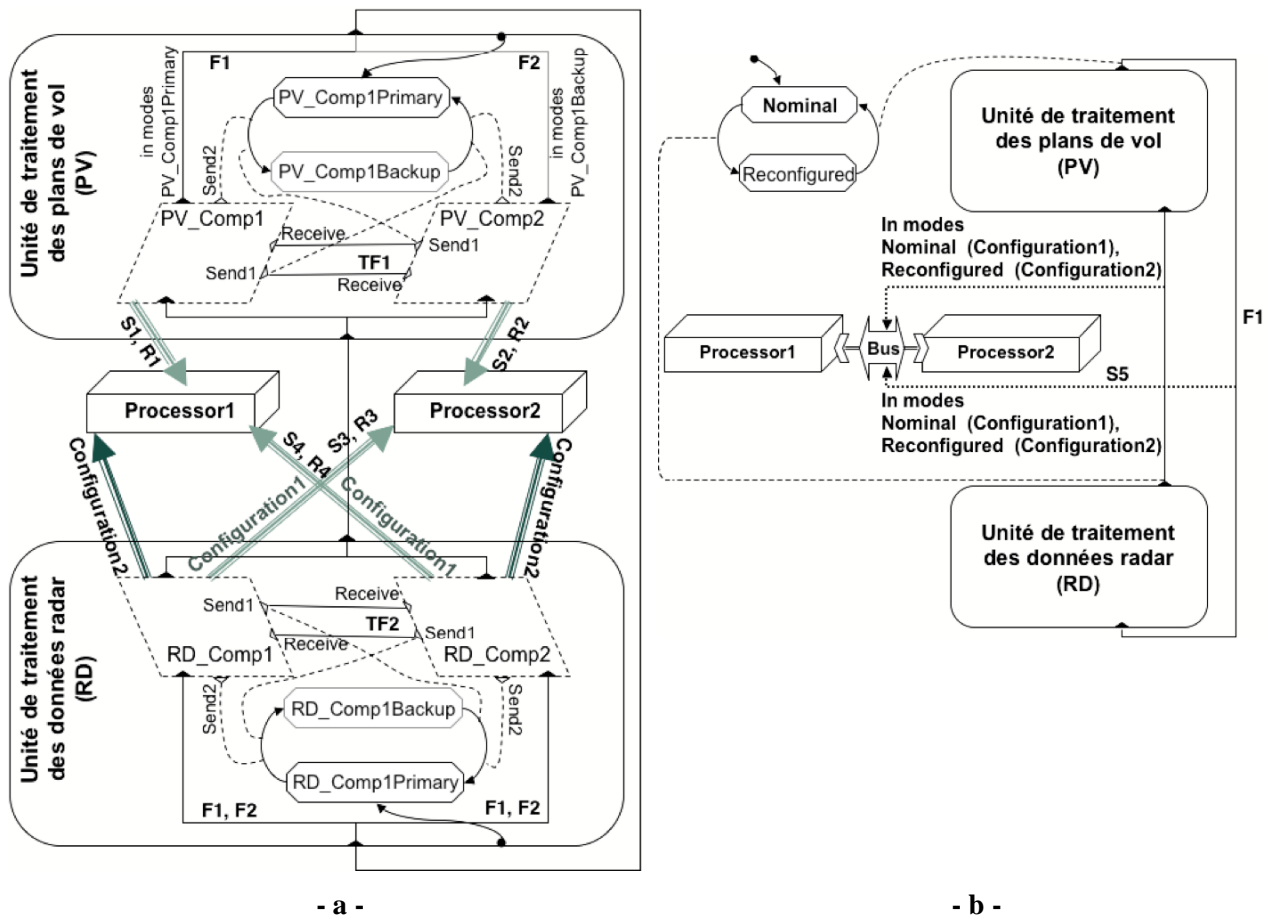


Figure 14. Modèle AADL architectural.

## V.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*.

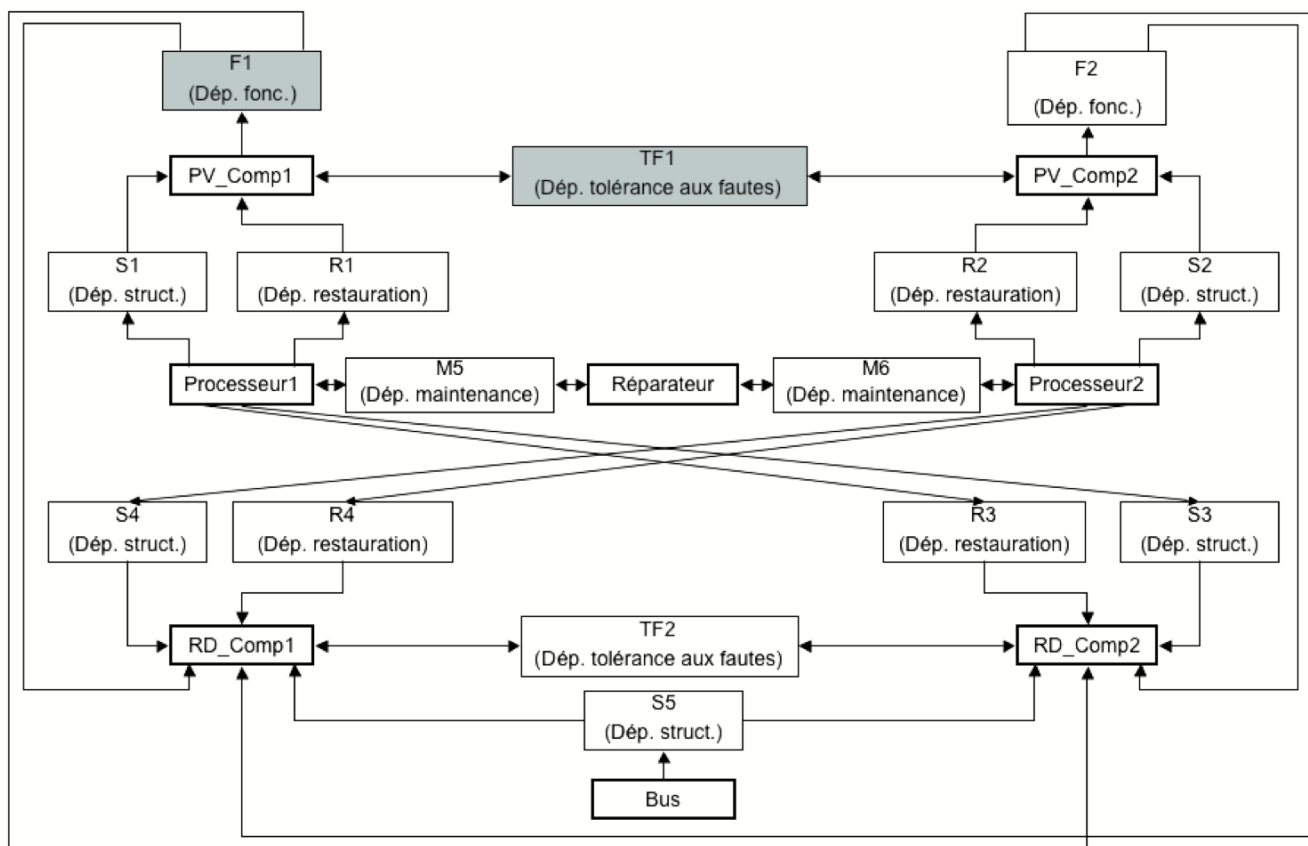


Figure 15. Diagramme bloc de dépendances pour la Configuration1.

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 des blocs de dépendances structurelle et 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.

### V.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 IV.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*.

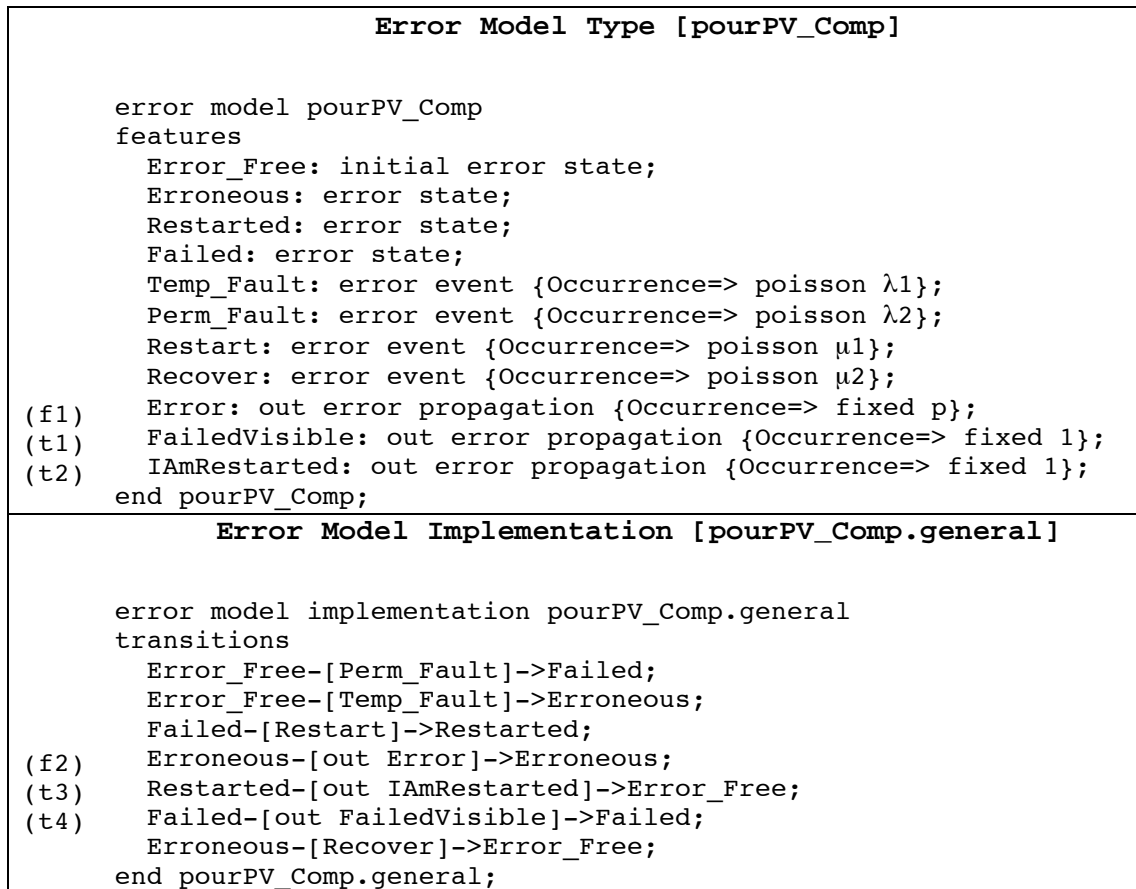
La Figure 16 présente le modèle d'erreur associé aux répliques de l'unité PV.

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*.

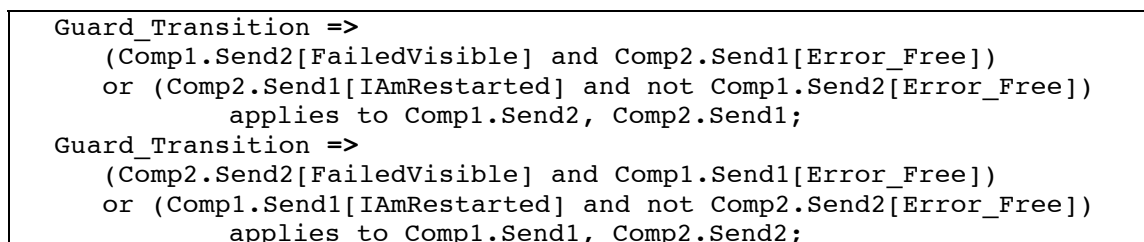
La seule différence entre le modèle d'erreur associé aux répliques de l'unité PV est la direction de la propagation *Error* et la transition AADL qu'elle déclenche. Dans le modèle d'erreur associé aux répliques de l'unité RD, *Error* est une propagation **in** qui déclenche une transition de l'état *Error\_Free*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



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



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

## V.4 Transformation du modèle AADL vers RdPSG

Pour ces deux dépendances, nous utilisons les règles de transformation présentées dans le paragraphe IV. 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 vidant les places correspondant aux propagations *out*.

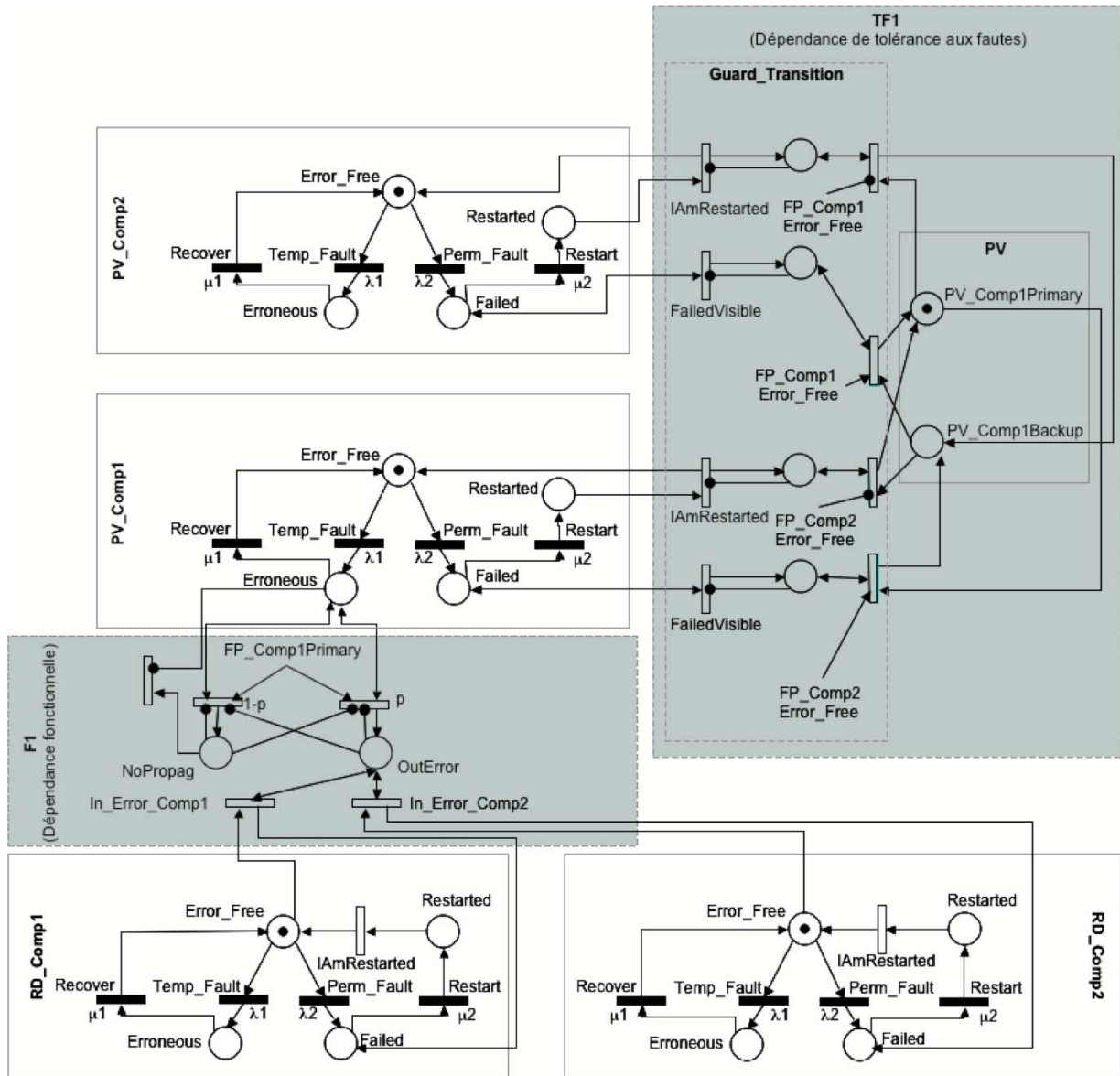
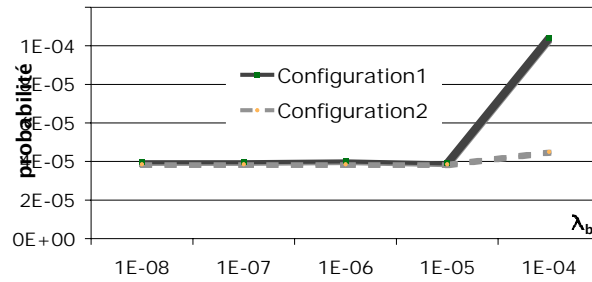


Figure 18. RdPSG du sous-système du système informatique français de trafic aérien – deux dépendances.

## V.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 19. Indisponibilité**

## Conclusion et perspectives

Nous avons 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. Nous proposons également un ensemble de sous-modèles génériques réutilisables décrivant des architectures classiques tolérantes aux fautes. Cet ensemble n'a pas été détaillé dans ce résumé. Il est décrit dans le Chapitre 3 de la thèse. Le modèle AADL de sûreté de fonctionnement 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 stochastic Petri nets through model Transformation*), mettant en œuvre nos règles de transformation.

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.

Notre expérience de modélisation de la sûreté de fonctionnement avec AADL nous a permis de proposer des évolutions du standard AADL. Les propositions les plus importantes sont décrites dans l'Annexe 1 de la thèse. Il est à mentionner qu'une partie de ces propositions a été déjà intégrée dans la version actuelle du standard.

Plusieurs directions peuvent être explorées afin d'étendre les travaux présentés ici. 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, il serait intéressant de définir des approches similaires de modélisation, permettant d'obtenir d'autres analyses (par exemple liées à la performance) à partir du même modèle, ce qui permettrait d'aboutir plus facilement à des compromis pertinents.

## Table of Contents

|  |    |
|--|----|
| Introduction.....  | 1  |
| Dissertation outline.....  | 3  |
| I    Linking Model-Driven Engineering and Dependability.....       | 5  |
| I.1    Model-driven engineering.....                               | 5  |
| I.1.1    Initiatives towards MDE.....                              | 6  |
| I.1.2    Languages for MDE.....                                    | 7  |
| I.2    Model-based dependability evaluation.....                   | 8  |
| I.2.1    Choice of measures.....                                   | 9  |
| I.2.2    Model construction.....                                   | 9  |
| I.2.3    Model processing.....                                     | 11 |
| I.3    Examples of analyses integrated into languages for MDE..... | 11 |
| I.3.1    UML-based analyses.....                                   | 12 |
| I.3.2    SysML-based analyses.....                                 | 14 |
| I.3.3    EastADL-based analyses.....                               | 14 |
| I.3.4    AADL-based analyses.....                                  | 15 |
| I.4    Proposed AADL-based dependability modeling framework.....   | 16 |
| I.4.1    Overview of our modeling framework.....                   | 17 |
| I.4.2    The AADL dependability model.....                         | 18 |
| I.4.3    AADL to GSPN model transformation.....                    | 19 |
| I.5    Conclusion.....   | 19 |
| II   Background.....   | 21 |
| II.1   AADL.....   | 21 |
| II.1.1   Core AADL language.....                                   | 22 |
| II.1.1.1   Components.....   | 22 |
| II.1.1.2   Architecture configurations.....                        | 24 |
| II.1.2   AADL Error Model Annex.....                               | 25 |
| II.1.2.1   Error model for independent components.....             | 26 |
| II.1.2.2   Error model for dependent components.....               | 27 |
| II.1.2.3   Propagation filtering and masking mechanisms.....       | 29 |
| II.1.2.3.1   Guard_In.....   | 29 |
| II.1.2.3.2   Guard_Out.....  | 31 |
| II.1.2.3.3   Comparison between Guard_In and Guard_Out.....        | 32 |
| II.1.2.3.4   Interacting Guard_In and Guard_Out properties.....    | 33 |
| II.1.2.4   Mechanisms for connecting error states to modes.....    | 34 |
| II.1.2.4.1   Guard_Event.....                                      | 34 |

|             |  |    |
|-------------|--|----|
| II.1.2.4.2  | Guard_Transition .....                                     | 36 |
| II.1.2.4.3  | Activate / Deactivate transitions .....                    | 38 |
| II.1.2.5    | Error model abstractions .....                             | 38 |
| II.2        | Petri nets .....   | 40 |
| II.2.1      | Place/Transitions Petri nets .....                         | 40 |
| II.2.2      | Generalized Stochastic Petri nets .....                    | 42 |
| II.3        | Conclusion .....   | 43 |
| III         | AADL Dependability Modeling: Guidelines and Patterns ..... | 45 |
| III.1       | Modeling independent components .....                      | 46 |
| III.2       | Modeling dependencies .....                                | 46 |
| III.2.1     | On the use of out and in propagations .....                | 47 |
| III.2.2     | Modeling structural and functional dependencies .....      | 48 |
| III.2.3     | Modeling maintenance dependencies .....                    | 48 |
| III.2.3.1   | Shared maintenance facility .....                          | 49 |
| III.2.3.2   | Priority to a component's maintenance .....                | 51 |
| III.2.4     | Modeling fault-tolerance dependencies .....                | 52 |
| III.3       | Fault-tolerance patterns .....                             | 54 |
| III.3.1.1   | Hardware fault-tolerance .....                             | 54 |
| III.3.1.1.1 | N-modular redundancy .....                                 | 55 |
| III.3.1.1.2 | Cold standby sparing .....                                 | 56 |
| III.3.1.1.3 | Warm standby sparing .....                                 | 57 |
| III.3.1.1.4 | Hot standby sparing .....                                  | 58 |
| III.3.1.1.5 | Active dynamic redundancy .....                            | 61 |
| III.3.1.2   | Software fault-tolerance .....                             | 64 |
| III.3.1.2.1 | N-version programming .....                                | 65 |
| III.3.1.2.2 | Recovery block .....                                       | 65 |
| III.3.1.2.3 | N self-checking programming .....                          | 65 |
| III.3.1.3   | Summary and observations .....                             | 66 |
| III.4       | Conclusion .....   | 68 |
| IV          | AADL to GSPN Model Transformation .....                    | 69 |
| IV.1        | Overview of the transformation .....                       | 69 |
| IV.2        | Transforming error models of independent components .....  | 70 |
| IV.3        | Transformation of basic dependency elements .....          | 71 |
| IV.3.1      | Out propagations .....                                     | 72 |
| IV.3.1.1    | Rule presentation .....                                    | 72 |
| IV.3.1.2    | Rule formalization .....                                   | 73 |
| IV.3.2      | In propagations .....                                      | 74 |
| IV.3.2.1    | Rule presentation .....                                    | 74 |
| IV.3.2.2    | Rule formalization .....                                   | 75 |

|          |  |     |
|----------|--|-----|
| IV.3.3   | Name-matching in – out propagations .....                            | 75  |
| IV.3.3.1 | Rule presentation .....  | 76  |
| IV.3.3.2 | Rule formalization .....   | 76  |
| IV.3.4   | Generalization to multiple receivers .....                           | 76  |
| IV.3.5   | On the choice of transformation rules of in – out propagations ..... | 78  |
| IV.4     | Transforming propagation filtering and masking mechanisms .....      | 79  |
| IV.4.1   | Guard_In .....   | 79  |
| IV.4.1.1 | Rule presentation .....  | 79  |
| IV.4.1.2 | Rule formalization .....   | 81  |
| IV.4.2   | Guard_Out .....  | 82  |
| IV.4.2.1 | Rule presentation .....  | 83  |
| IV.4.2.2 | Rule formalization .....   | 84  |
| IV.4.3   | Interacting Guard_In and Guard_Out properties .....                  | 85  |
| IV.4.3.1 | Cascading Guard_Out - Guard_In .....                                 | 85  |
| IV.4.3.2 | Cascading Guard_Out - Guard_Out .....                                | 86  |
| IV.5     | Mechanisms for connecting error states to modes .....                | 87  |
| IV.5.1   | Guard_Event .....  | 88  |
| IV.5.1.1 | Rule presentation .....  | 88  |
| IV.5.1.2 | Rule formalization .....   | 89  |
| IV.5.2   | Guard_Transition .....   | 90  |
| IV.5.2.1 | Rule presentation .....  | 91  |
| IV.5.2.2 | Rule formalization .....   | 92  |
| IV.5.3   | Activate / deactivate transitions .....                              | 93  |
| IV.5.3.1 | Rule presentation .....  | 93  |
| IV.5.3.2 | Rule formalization .....   | 95  |
| IV.6     | Transforming error model abstractions .....                          | 96  |
| IV.6.1   | Transforming abstract error models .....                             | 96  |
| IV.6.2   | Transforming derived error models .....                              | 96  |
| IV.6.2.1 | Rule presentation .....  | 97  |
| IV.6.2.2 | Rule formalization .....   | 98  |
| IV.7     | Taking into account architecture configurations .....                | 98  |
| IV.7.1   | Rule presentation .....  | 99  |
| IV.7.2   | Rule formalization .....   | 100 |
| IV.8     | Scalability analysis .....   | 101 |
| IV.9     | Conclusion .....   | 102 |
| V        | Case Study: Subsystem of the Air Traffic Control System .....        | 103 |
| V.1      | System description .....   | 103 |
| V.1.1    | AADL architectural models .....                                      | 104 |
| V.1.2    | Dependency analysis .....  | 106 |

|          |  |     |
|----------|--|-----|
| V.2      | AADL dependability model and transformation to GSPN .....          | 108 |
| V.2.1    | Iteration 1: independent components .....                          | 108 |
| V.2.1.1  | Assumptions.....   | 108 |
| V.2.1.2  | AADL dependability model.....                                      | 109 |
| V.2.1.3  | AADL to GSPN model transformation .....                            | 111 |
| V.2.2    | Iteration 2: structural dependency from processor to process ..... | 112 |
| V.2.2.1  | Assumptions.....   | 112 |
| V.2.2.2  | AADL dependability model.....                                      | 112 |
| V.2.2.3  | AADL to GSPN model transformation .....                            | 113 |
| V.2.3    | Iteration 3: recovery dependency from processor to process.....    | 114 |
| V.2.3.1  | Assumptions.....   | 114 |
| V.2.3.2  | AADL dependability model.....                                      | 114 |
| V.2.3.3  | AADL to GSPN model transformation .....                            | 115 |
| V.2.4    | Iteration 4: structural dependency from bus to RD processes .....  | 116 |
| V.2.4.1  | Assumptions.....   | 116 |
| V.2.4.2  | AADL dependability model.....                                      | 117 |
| V.2.4.3  | AADL to GSPN model transformation .....                            | 117 |
| V.2.5    | Iteration 5: functional dependency from FP to RD processes .....   | 119 |
| V.2.5.1  | Assumptions.....   | 119 |
| V.2.5.2  | AADL dependability model.....                                      | 119 |
| V.2.5.3  | AADL to GSPN model transformation .....                            | 120 |
| V.2.6    | Iteration 6: functional dependency between FP processes .....      | 121 |
| V.2.6.1  | Assumptions.....   | 121 |
| V.2.6.2  | AADL dependability model.....                                      | 121 |
| V.2.6.3  | AADL to GSPN model transformation .....                            | 122 |
| V.2.7    | Iteration 7: fault-tolerance dependency between FP processes ..... | 122 |
| V.2.7.1  | Assumptions.....   | 122 |
| V.2.7.2  | AADL dependability model.....                                      | 122 |
| V.2.7.3  | AADL to GSPN model transformation .....                            | 123 |
| V.2.8    | Iteration 8: fault-tolerance dependency between RD processes ..... | 124 |
| V.2.8.1  | Assumptions.....   | 124 |
| V.2.8.2  | AADL dependability model.....                                      | 124 |
| V.2.8.3  | AADL to GSPN model transformation .....                            | 125 |
| V.2.9    | Iteration 8: global reconfiguration strategy .....                 | 125 |
| V.2.9.1  | Assumptions.....   | 125 |
| V.2.9.2  | AADL dependability model.....                                      | 126 |
| V.2.9.3  | AADL to GSPN model transformation .....                            | 126 |
| V.2.10   | Iteration 10: maintenance dependency between processors .....      | 127 |
| V.2.10.1 | Assumptions .....  | 127 |

|   |   |     |
|---|---|-----|
| V.2.10.2  | AADL dependability model .....                                | 127 |
| V.2.10.3  | AADL to GSPN model transformation.....                        | 127 |
| V.3   | Quantitative dependability evaluation.....                    | 128 |
| V.3.1   | Comparison with respect to the failure rate of the bus.....   | 128 |
| V.3.2   | Comparison with respect to the FT policy.....                 | 129 |
| V.4   | Conclusion .....  | 129 |
| Conclusion.....   |   | 131 |
| Future research directions .....                                    |   | 132 |
| Appendix A: Error Model Annex Evolution Proposals.....              |   | 135 |
| A.1   | Occurrence properties .....                                   | 135 |
| A.2   | Link between modes and the Error Model Annex constructs ..... | 135 |
| A.3   | Guard_In property without applies to clause.....              | 136 |
| A.4   | Inheritance and refinements.....                              | 137 |
| Appendix B: General Rule for Emptying GSPN Propagation Places ..... |   | 139 |
| Appendix C: Model Transformation Tool .....                         |   | 141 |
| C.1   | A developer's perspective .....                               | 141 |
| C.1.1   | gspnModel: Ecore metamodel .....                              | 142 |
| C.1.2   | dependency .....  | 143 |
| C.1.3   | aadl2gspn .....   | 143 |
| C.2   | A user's perspective .....                                    | 144 |
| References .....  |   | 145 |



## List of Figures

|   |    |
|---|----|
| Figure I-1. <i>Modeling framework</i> .....   | 17 |
| Figure II-1. <i>Component categories - AADL graphical notation</i> .....                      | 22 |
| Figure II-2. <i>Port categories - AADL graphical notation</i> .....                           | 23 |
| Figure II-3. <i>Different component implementations (b and c) for the same type (a)</i> ..... | 24 |
| Figure II-4. <i>System with modal architecture configurations</i> .....                       | 25 |
| Figure II-5. <i>Error model example for independent component</i> .....                       | 27 |
| Figure II-6. <i>Error model instance association</i> .....                                    | 27 |
| Figure II-7. <i>Graphical notation for error model instance</i> .....                         | 27 |
| Figure II-8. <i>Error model example for component with interactions (sender-side)</i> .....   | 28 |
| Figure II-9. <i>Error propagation (recipient-side)</i> .....                                  | 28 |
| Figure II-10. <i>Guard_In property syntax</i> .....   | 30 |
| Figure II-11. <i>Guard_In property example</i> .....  | 30 |
| Figure II-12. <i>Architectural view of a Guard_In property</i> .....                          | 31 |
| Figure II-13. <i>Guard_Out property syntax</i> .....  | 31 |
| Figure II-14. <i>Guard_Out property example</i> .....   | 32 |
| Figure II-15. <i>Architectural view of a Guard_Out property</i> .....                         | 32 |
| Figure II-16. <i>Cascading Guard_Out - Guard_In properties</i> .....                          | 33 |
| Figure II-17. <i>Cascading Guard_Out - Guard_Out properties</i> .....                         | 34 |
| Figure II-18. <i>Guard_Event property syntax</i> .....  | 35 |
| Figure II-19. <i>Guard_Event property example</i> .....                                       | 35 |
| Figure II-20. <i>Architectural view of a Guard_Event property</i> .....                       | 36 |
| Figure II-21. <i>Guard_Transition property syntax</i> .....                                   | 37 |
| Figure II-22. <i>Guard_Transition property example</i> .....                                  | 37 |
| Figure II-23. <i>Architectural view of Guard_Transition property</i> .....                    | 37 |
| Figure II-24. <i>Activate/deactivate transitions</i> .....                                    | 38 |
| Figure II-25. <i>Derived_State_Mapping expression definition</i> .....                        | 39 |
| Figure II-26. <i>Architectural view of error model abstractions</i> .....                     | 40 |
| Figure II-27. <i>Examples of PN</i> .....   | 41 |
| Figure II-28. <i>Examples of GSPN</i> .....   | 43 |
| Figure III-1: <i>Transition and state visible from outside</i> .....                          | 47 |
| Figure III-2. <i>Structural dependency example</i> .....                                      | 48 |
| Figure III-3. <i>Maintenance dependency example</i> .....                                     | 49 |
| Figure III-4. <i>Error model for component with maintenance dependency</i> .....              | 50 |
| Figure III-5. <i>Error model for shared maintenance facility</i> .....                        | 51 |



|  |    |
|--|----|
| Figure III-6. <i>Textual AADL dependability model – shared maintenance facility</i> .....            | 51 |
| Figure III-7. <i>Refinement of dependent.general</i> .....   | 52 |
| Figure III-8. <i>Textual AADL dependability model – maintenance with priority</i> .....              | 52 |
| Figure III-9. <i>Fault-tolerance dependency example</i> .....  | 53 |
| Figure III-10. <i>Failure propagation in error model sender.general</i> .....                        | 54 |
| Figure III-12. <i>N-modular redundancy pattern</i> .....   | 55 |
| Figure III-13. <i>Textual AADL dependability model – N-modular redundancy</i> .....                  | 55 |
| Figure III-14. <i>Cold standby sparing pattern</i> .....   | 56 |
| Figure III-15. <i>Textual AADL dependability model – cold standby sparing</i> .....                  | 57 |
| Figure III-16. <i>Warm standby sparing pattern</i> .....   | 58 |
| Figure III-17. <i>Textual AADL dependability model – warm standby sparing</i> .....                  | 58 |
| Figure III-18. <i>AADL architectural model of the hot standby sparing pattern (variant 1)</i> .....  | 59 |
| Figure III-19. <i>Textual AADL dependability model – hot standby sparing (variant 1)</i> .....       | 59 |
| Figure III-20. <i>AADL architectural model of the hot standby sparing pattern (variant 2)</i> .....  | 60 |
| Figure III-21. <i>Textual AADL dependability model – hot standby sparing (variant 2)</i> .....       | 61 |
| Figure III-22. <i>Active dynamic redundancy pattern with self-checking replicas (variant 1)</i> .... | 62 |
| Figure III-23. <i>Textual AADL dependability model – active dynamic redundancy (variant 1)</i> .     | 63 |
| Figure III-24. <i>Active dynamic redundancy pattern with self-checking replicas (variant 2)</i> .... | 63 |
| Figure III-25. <i>Textual AADL dependability model - active dynamic redundancy (variant 2)</i> .     | 64 |
| Figure IV-1. <i>Illustration of the transformation rule for independent components</i> .....         | 71 |
| Figure IV-2. <i>AADL transition triggered by an out propagation</i> .....                            | 72 |
| Figure IV-3. <i>Transformation rule for AADL transition triggered by an out propagation</i> .....    | 72 |
| Figure IV-4. <i>Transformation rule for in propagation</i> .....                                     | 74 |
| Figure IV-5. <i>Sender and Receiver – in-out name matching propagations</i> .....                    | 75 |
| Figure IV-6. <i>GSPN modeling a propagation from a sender to a receiver</i> .....                    | 76 |
| Figure IV-7. <i>Propagations from one sender to two receivers</i> .....                              | 77 |
| Figure IV-8. <i>Alternative transformation rule for out propagation</i> .....                        | 79 |
| Figure IV-9. <i>Guard_In property syntax</i> .....   | 79 |
| Figure IV-10. <i>Guard_In property</i> .....   | 80 |
| Figure IV-11. <i>Transformation rule for Guard_In property</i> .....                                 | 81 |
| Figure IV-12. <i>Guard_Out property syntax</i> .....   | 82 |
| Figure IV-13. <i>Guard_Out property</i> .....  | 83 |
| Figure IV-14. <i>Transformation rule for Guard_Out property</i> .....                                | 84 |
| Figure IV-15. <i>Cascading Guard_Out - Guard_In properties</i> .....                                 | 85 |
| Figure IV-16. <i>GSPN modeling of cascading Guard_Out - Guard_In properties</i> .....                | 86 |
| Figure IV-17. <i>Cascading Guard_Out - Guard_Out properties</i> .....                                | 87 |
| Figure IV-18. <i>GSPN modeling of cascading Guard_Out - Guard_Out properties</i> .....               | 87 |
| Figure IV-19. <i>Guard_Event property syntax</i> .....   | 88 |

|   |     |
|---|-----|
| Figure IV-20. <i>Guard_Event</i> property.....  | 89  |
| Figure IV-21. <i>Transformation rule for Guard_Event</i> property .....                                 | 89  |
| Figure IV-22. <i>Guard_Transition</i> property syntax.....  | 91  |
| Figure IV-23. <i>Guard_Transition</i> property.....   | 91  |
| Figure IV-24. <i>Transformation rule for Guard_Transition</i> property.....                             | 92  |
| Figure IV-25. <i>Architectural view of activate/deactivate transitions</i> .....                        | 94  |
| Figure IV-26. <i>Activate/deactivate transitions - Transformation rule</i> .....                        | 94  |
| Figure IV-27. <i>Derived_State_Mapping</i> definition .....   | 96  |
| Figure IV-28. <i>Example of derived error model</i> .....   | 97  |
| Figure IV-29. <i>GSPN modeling of the derived error model</i> .....                                     | 97  |
| Figure IV-30. <i>System with modal architecture configurations</i> .....                                | 99  |
| Figure IV-31. <i>Modal configuration - Transformation rule</i> .....                                    | 100 |
| Figure IV-32. <i>State space analysis</i> .....   | 102 |
| Figure V-1. <i>Candidate architectures</i> .....  | 104 |
| Figure V-2. <i>AADL architectural model of Configuration1</i> .....                                     | 105 |
| Figure V-3. <i>AADL architectural model of Configuration2</i> .....                                     | 105 |
| Figure V-4. <i>Dependency Block Diagram of Configuration1</i> .....                                     | 107 |
| Figure V-5. <i>Error model for independent processor</i> .....  | 109 |
| Figure V-6. <i>Error model for independent bus</i> .....  | 110 |
| Figure V-7. <i>Error model for independent process</i> .....  | 110 |
| Figure V-8. <i>Textual AADL dependability model - processor</i> .....                                   | 111 |
| Figure V-9. <i>GSPN modeling an independent processor</i> .....   | 111 |
| Figure V-10. <i>GSPN modeling an independent bus</i> .....  | 111 |
| Figure V-11. <i>GSPN modeling an independent process</i> .....  | 111 |
| Figure V-12. <i>Error model for structural dependency (sender side)</i> .....                           | 112 |
| Figure V-13. <i>Error model for structural dependency (recipient side)</i> .....                        | 113 |
| Figure V-14. <i>GSPN modeling a processor and two processes with structural dependencies</i> .....      | 113 |
| Figure V-15. <i>Error model for recovery dependency (sender side)</i> .....                             | 115 |
| Figure V-16. <i>Error model for recovery dependency (recipient side)</i> .....                          | 115 |
| Figure V-17. <i>GSPN modeling a processor and two processes with recovery dependencies</i> .....        | 116 |
| Figure V-18. <i>Error model for structural dependency bus – RD process (recipient side)</i> .....       | 117 |
| Figure V-19. <i>Error model for structural dependency bus – RD process (sender side)</i> .....          | 117 |
| Figure V-20. <i>GSPN of the bus and two processes with structural dependency (Configuration1)</i> ..... | 118 |
| Figure V-21. <i>GSPN of the bus and two processes with structural dependency (Configuration2)</i> ..... | 118 |
| Figure V-22. <i>AADL architectural model of Configuration1 – refined with modal connections</i> .....   | 119 |
| Figure V-23. <i>Error model for functional dependency FP process – RD process (sender side)</i> .....   | 120 |

|  |     |
|--|-----|
| Figure V-24. Error model for functional dependency FP process – RD process (recipient side)..... | 120 |
| Figure V-25. GSPN modeling a FP process and the two RD processes with functional dependency..... | 121 |
| Figure V-26. Error model for functional dependency between FP processes.....                     | 122 |
| Figure V-27. Error model for FT dependency between FP processes.....                             | 123 |
| Figure V-28. Textual AADL dependability model for FP_Comp1, with dependency FT1' ...             | 123 |
| Figure V-29. GSPN modeling the two FP processes with FT dependency .....                         | 124 |
| Figure V-30. GSPN modeling the two RD processes with FT dependency.....                          | 125 |
| Figure V-31. Modeling the global reconfiguration strategy .....                                  | 126 |
| Figure V-32. GSPN modeling the two RD processes with FT dependency.....                          | 126 |
| Figure V-33. Refined AADL architectural model of Configuration1.....                             | 127 |
| Figure V-34. GSPN modeling the two processors with maintenance dependency .....                  | 128 |
| Figure V-35. Unavailability of RD with respect to $\lambda_b$ .....                              | 129 |
| Figure A-1. Conflicting Guard_In properties.....   | 137 |
| Figure B-1. General GSPN for out propagation.....  | 139 |
| Figure B-2. Emptying an out propagation place - Example.....                                     | 140 |
| Figure C-1. Overview of the model transformation tool .....                                      | 141 |
| Figure C-2. Ecore metamodel for GSPN.....  | 143 |

---

# Introduction

---

The increasing complexity of new-generation systems raises major concerns in various critical application domains, in particular with respect to the validation and analysis of performance, timing and dependability-related requirements. During the last decade, engineering approaches aimed at mastering this complexity during the development process have emerged and are being increasingly used in industry. Component-based engineering and, more recently, model-driven engineering address the problem of complexity by promoting reuse and partial or total automation of certain phases of the development process. These engineering approaches must be supported by languages and tools that provide means to ensure that the implemented system complies with its specifications. In particular, it is necessary to integrate analyses of quality attributes<sup>7</sup> (such as dependability and performance) in the development process.

Currently, the definition of model-driven engineering approaches is the subject of many efforts both from industry and academia. One important concern in these approaches is the choice of the most appropriate languages and tools to be used. Generally, this choice depends on the application domain and on the variety and maturity of the tools that support a particular language. Most of the model-driven engineering approaches under development rely either on UML (Unified Modeling Language), which is a general-purpose modeling language, or on ADLs (architecture description languages) that are usually domain-specific, or on a combination of both.

In order to ensure that the system complies with its functional and quality specifications, the languages used in model-driven approaches must support analyses related to system behavior, performance and dependability. In traditional development processes, each type of analysis is based on a dedicated model, which requires substantial amount of training to be used effectively. Performing several analyses of quality attributes will benefit a lot from using a single model. This also contributes to the reduction of the development cost by facilitating model reuse.

Considering the derivation of analyses of quality attributes from such modeling languages, a significant amount of research has been carried out based on UML. On the other hand, AADL (Architecture Analysis and Design Language) has received a growing interest from the embedded safety-critical industry (e.g., Honeywell, Rockwell Collins, Lockheed Martin, the European Space Agency, Astrium, Airbus) during the last years. AADL has been standardized in 2004 under the auspices of the International Society of Automotive Engineers (SAE), to support the design and analysis of complex real-time safety-critical systems in avionics, automotive, space and other application domains. AADL provides a standardized textual and graphical notation for describing software and hardware system architectures and their functional interfaces. The serious consideration of AADL by the embedded safety-critical industry is justified by AADL's advanced support for modeling reconfigurable architectures and for analyzing quality attributes.

Similarly to UML users, AADL users are interested in analyzing quality attributes based on AADL models. To this end, the core AADL language has been designed to be extensible to

---

<sup>7</sup> Quality attributes are also referred to as non-functional properties in the literature.

accommodate analyses that the core language does not completely support. In particular, the AADL Error Model Annex has been standardized in 2006 to complement the description capabilities of the core language by providing features with precise semantics to be used for describing dependability-related characteristics in AADL models (e.g., faults, failure modes, repair policies, error propagations). Besides describing the systems' behavior in the presence of faults, developers are interested in obtaining quantitative measures of relevant dependability properties such as reliability and availability and they are confronted with two fundamental questions:

- 1) How to take into account the various dependencies between the components of the system in the presence of faults. These dependencies are inherent to the architecture or they are due to fault-tolerance and maintenance policies.
- 2) How to obtain dependability measures from an AADL model.

Currently, there is no methodology for helping developers to solve these problems. In this dissertation, we propose an AADL-based modeling framework aiming at filling these gaps.

Our framework relies on the core AADL language and on the AADL Error Model Annex. The system architecture is described using the core AADL language, while the dependability-related information is described separately, using Error Model Annex constructs. Then, the architectural model is annotated with Error Model Annex constructs. The annotations can be easily abstracted away. This feature enhances the reusability and the readability of the AADL architectural model that can be also used for other analyses.

Additionally, we take advantage of the existence of mature dependability-oriented analytical modeling techniques that are generally based on the use of fault trees, Markov chains and GSPNs (Generalized Stochastic Petri Nets). In particular, Markov chains are able to capture various functional and stochastic dependencies among components and they allow the evaluation of various measures related to dependability (such as reliability, availability and maintainability), and dependability and performance (i.e., performability measures). To facilitate the generation of large state-space models, high-level specification languages such as GSPNs are generally used as they can automatically be converted into Markov chains. Also, GSPNs provide efficient means for structural model verification and analysis, before the Markov chain generation. Such verification support facilities are very useful when dealing with large models. During the last decade, various approaches have been defined to support the systematic construction and validation of dependability models based on GSPNs and their extensions. They are a source of inspiration for our framework that aims at favoring model reuse and evolvability in the context of an AADL-based engineering process.

Our contributions are the following.

- 1) Elaboration of a structured dependability modeling approach for building AADL dependability models. Indeed, relying on a methodology is a necessity when dealing with complex systems. To support reusability and to master complexity, in our approach, the AADL model is built iteratively, progressively taking into account dependencies between components. This allows incremental validation of the model.
- 2) Guidance on using the AADL language for modeling different types of behaviors of system components in the presence of faults. In particular, we show that the development of patterns is very useful to facilitate the modeling of fault-tolerance behavior and to enhance the reusability of the models.
- 3) Definition of model transformation rules allowing the generation of GSPNs from AADL models. The set of model transformation rules has been designed to be automated. In this way, the model transformation is completely transparent to the user

and the complexity of analytical models is hidden to end-users who generally have a limited knowledge of GSPNs. A tool implementing the model transformation can be interfaced with one of the existing GSPN processing tools to evaluate dependability measures. We have shown the feasibility of the automation by implementing the tool ADAPT (*from AADL Architectural models to stochastic Petri nets through model Transformation*). It generates GSPNs both in a generic XML/XMI format and in the input format specific to the dependability evaluation tool Surf-2 [Béounes *et al.* 1993].

- 4) Proposals for the evolution of the AADL Error Model Annex. Our dependability modeling experience with AADL allowed us to identify necessary evolutions of the AADL Error Mode Annex. We have submitted these proposals to the AADL standardization committee, which decided to integrate some of them in the first version of the standard while the others are considered for its second version.

## **Dissertation outline**

This dissertation is structured into five chapters.

Chapter I presents the context and the motivation of our work. It briefly discusses the main ideas of model-driven engineering and the classical techniques related to model-based dependability evaluation. Concerning model-driven engineering, we report significant initiatives towards the definition of such approaches and we briefly present the modeling languages that are often considered in their contexts. We give an overview of the steps to be followed in model-based dependability evaluation, i.e., the choice of measures, model construction and model processing. Related work concerning contributions towards the integration of different types of analyses into languages for model-driven engineering are reviewed. After having identified the area to be explored in this context, we give an overview of our iterative modeling framework based on AADL.

Chapter II provides the necessary background related to the two modeling languages on which we base our work. The first part of this Chapter is dedicated to AADL. The main concepts of the AADL core language (i.e., components and architecture configurations) are briefly presented before detailing the constructs of the AADL Error Model Annex. They allow users to add dependability-related information to an AADL architectural model. The second part of the Chapter is dedicated to Petri nets and to the particularities of GSPNs.

Chapter III gives guidelines for efficiently building AADL dependability models. It focuses on modeling independent components and then different types of dependability-related dependencies. We have identified all AADL constructs necessary for the description of dependencies and we defined modeling rules for each of the dependencies. The Chapter also provides a set of patterns, modeling classical fault-tolerance policies for software and hardware systems. These patterns aim at enhancing model reusability. We show how they can either be instantiated or adapted to comply with more specific assumptions before being instantiated. The guidelines and patterns are to be used in the context of our iterative modeling framework presented in Chapter I and illustrated in Chapter V.

Chapter IV is dedicated to model transformation from AADL to GSPNs. The definition of the model transformation is based on the dependency modeling rules of Chapter III and on systematic transformation rules. We present and formalize the exhaustive set of transformation rules for all the AADL Error Model Annex constructs presented in Chapter II. This set is necessary and sufficient to obtain a GSPN taking into account all the dependability-related behaviors that have been identified as necessary and sufficient for

dependability evaluation. The definition of the rules favors the modularity of the GSPN and they can be easily implemented. We also discuss issues related to the scalability of the transformation and practical implementation.

Chapter V aims at illustrating the use of our entire framework on a subsystem of the French Air Traffic Control System. We show how the AADL dependability model is built iteratively, by taking into account the guidelines of Chapter III. In particular, we make use of fault-tolerance patterns to model the two fault-tolerance mechanisms of the system. For one of the mechanisms, we instantiate the most appropriate pattern after having customized it, while for the other one, it is sufficient to instantiate the pattern without modification. Even though the AADL to GSPN model transformation is meant to be hidden from the user, we show the GSPN obtained for each iteration. We finish by showing two examples of quantitative dependability analyses obtained by processing the GSPN. These results allow us to compare candidate architectures of the system with respect to their availability. Many other results may be obtained from similar analyses. In this dissertation we do not focus on the analysis, but on obtaining a dependability evaluation model from an AADL model.

Three appendices are also included.

Appendix A enumerates our most important evolution proposals for the AADL Error Model Annex. They have been identified during the application of our modeling framework to the case study presented in Chapter V. Some of them have already been integrated in the current version of the standard.

Appendix B is dedicated to the generalization of a transformation rule of Chapter IV. It is presented in the appendix for didactical reasons.

Appendix C provides an overview of ADAPT, the model transformation tool that we have developed. It is based on the Eclipse Modeling Framework and interfaces the Open Source AADL Tool Environment (OSATE)<sup>8</sup> and Surf-2 on the GSPN side.

---

<sup>8</sup> <http://www.aadl.info/OpenSourceAADLToolEnvironment.html>

---

# I Linking Model-Driven Engineering and Dependability

---

Our goal is to integrate dependability modeling and evaluation in a model-driven engineering (MDE) development process based on AADL (Architecture Analysis and Design Language) [Feiler *et al.* 2006]. Classically, dependability evaluation is based on processing dependability-oriented analytical models such as Markov chains or fault trees. The objective of this Chapter is to give an overview of our AADL-based dependability modeling framework in its context. To this end, we first present briefly the concepts related to MDE and to model-based dependability evaluation. Then, we survey contributions aiming at integrating analyses into languages used in MDE and we give an overview of our proposal.

This Chapter is structured as follows. Section I.1 presents the main initiatives of model-driven engineering and their associated languages. Section I.2 focuses on the classical model-based dependability evaluation process. Section I.3 surveys work related to ours. Section I.4 gives an overview of our proposal regarding the integration of dependability modeling and evaluation in a model-driven engineering approach based on AADL. Section I.4 concludes this chapter.

## ***I.1 Model-driven engineering***

Model-driven Engineering (MDE) refers to the systematic use of models as primary engineering artifacts throughout the development process [Bézivin 2006]. Models represent abstractions of physical systems. They are mainly used for two purposes:

- 1) To allow engineers to reason about the system's properties by focusing only on aspects that are relevant at the current stage.
- 2) To allow the different stakeholders to communicate on a common basis.

In industry, models and designs assumed to reflect aspects of the real system are usually built at early stages in the development process and it is difficult to update them according to the system's evolution, especially in complex large-scale systems. MDE focuses on methods and tools for mapping models to implementations and for keeping trace of the evolution of models and implementations. In MDE frameworks, the development process is based on model transformations. They aim at capturing component-based architectures in a platform-independent way and then transform them to platform-specific models that are finally transformed into actual implementations. MDE takes advantage of the engineering method of component-based development [Brown & Wallnau 1996] in which large-scale systems are built by assembling their architectures from previously-existing COTS components that may come from different sources and may run on different platforms. The principles of MDE are as follows:

- Models are expressed in a well-defined notation or language and are the means for understanding systems;



- A system is built based on set of models by imposing *transformations* between models;
- *Metamodels*, representing the formal semantics of models, facilitate the automation of model transformations by tools;
- The whole process is based on industry *standards*.

The following two subsections present successively initiatives for defining and implementing MDE approaches, and the languages adopted to support them. All initiatives use either UML (Unified Modeling Language), or an ADL (Architecture Description Language), or a combination of both. UML is a general-purpose object-oriented language, while ADLs are generally domain-specific. In particular, ADLs are well adapted for modeling system configurations. Also, the focus on the conceptual architecture and the explicit consideration of connectors differentiate ADLs from object-oriented notations.

### I.1.1 Initiatives towards MDE

The main MDE initiative was taken by the OMG under the name of Model-Driven Architecture (MDA) [Brown 2004]. MDA is based on several types of models: computation independent model (CIM), platform independent model (PIM), platform specific model (PSM) that is further refined in platform model (PM) and implementation specific model (ISM). MDA tools should support model transformations between these models going from a model that supports analysis to executable code.

Besides the OMG initiative, the industry expressed its interest for MDE. Many companies (such as EADS Astrium, Raytheon, Lockheed Martin) are currently transitioning from a paper-based development process to a MDE process [Blanquart *et al.* 2006, Slaby & Baker 2006, Waddington & Lardieri 2006]. Several consortiums formed of industrial and academic partners focused on putting into practice the MDE concepts and on building tools to support them. A non-exhaustive list of such projects is given hereafter.

- *ASSERT*<sup>9</sup>: European project led by the European Space Agency, aiming at improving the development process for critical embedded real-time systems by employing a proven by design approach. The languages considered in this project are UML (Unified Modeling Language) and AADL (Architecture Analysis and Design Language).
- *ATESST*<sup>10</sup>: European project led by Volvo Technology, aiming at delivering an ADL able to provide means to handle the complexity and improve safety, reliability, cost and development efficiency of automotive electronic systems. The project is based on the ADL EastADL resulting from a former project, *EAST-EEA*<sup>11</sup>.
- *COTRE*<sup>12</sup>: aiming at defining a modeling method combining formal and semi-formal approaches in an industrial process ensuring continuity and traceability from the architectural stage to the implementation stage on the hardware target. COTRE selected AADL to support this modeling method.
- *ModelWare*<sup>13</sup>: European project led by Thales Research and Technology, aiming at building a complete infrastructure for large-scale deployment of model-driven

---

<sup>9</sup> <http://www.assert-online.net/>

<sup>10</sup> <http://www.atesst.org/>

<sup>11</sup> <http://www.east-eea.net/>

<sup>12</sup> <http://www.laas.fr/COTRE/brindex.html>

<sup>13</sup> <http://www.modelware-ist.org/>

development strategies and at validating the infrastructure in several business domains. This project is based on UML.

- *Neptune*<sup>14</sup>: European project led by CS, aiming at developing a method and the necessary tools supporting the UML [Canals *et al.* 2002].
- *Spices*<sup>15</sup>: European project bringing together industry and academia from France, Spain and Belgium, aiming at proposing a MDE approach based on AADL going from PIM models to actual implementations [Delanote *et al.* 2007].
- *Topcased*<sup>16</sup>: led by Airbus, aiming at optimizing development costs for critical embedded systems by building an open source CASE (Computer Aided Software Engineering) environment with goals such as minimizing ownership costs, ensuring the independence of development platforms and integrating advances made in the academic world. The languages supported by the open source CASE environment are AADL, UML and SysML.

## I.1.2 Languages for MDE

To be used effectively, MDE must be supported by languages and standards. The projects presented in the preceding subsection adopted one or several languages among the following: UML (Unified Modeling Language), SysML (System Modeling Language), EastADL and AADL (Architecture Analysis and Design Language). These languages are briefly described hereafter.

**UML** [OMG 2007b] is a general purpose modeling language (tending to be software-centric). It has been standardized by the OMG (Object Management Group). It allows users to describe systems through different types of diagrams, which express the system's structure (e.g., class, component, composite structure diagrams) or behavior (e.g., activity, use case, state machine diagrams). Every diagram gives a specific view of the system, which is useful during certain phases of the development cycle or for certain types of applications. UML can be tailored to a specific domain through *profiles*. Profiles are collections of *stereotypes* and *tagged values* applied to elements, attributes, methods and links. Currently, there are two profiles supporting the assessment of quality attributes: the Schedulability, Performance and Time (SPT) profile, and the QoS and fault-tolerance characteristics and mechanisms (QoS&FT) profile. Recently, [Bernardi & Merseguer 2007] proposed a skeleton for a Dependability Analysis (DA) profile. Many tools, most of which are commercial, have been developed to support UML.

**SysML** [OMG 2007a] is a standard UML profile that defines a domain-specific language for system engineering. It supports specification, analysis, design, verification and validation of systems formed of hardware, software, personnel and facilities. SysML uses seven of the UML's thirteen diagrams and adds two new diagram types: *requirement* diagrams (used for requirements management) and *parametric* diagrams (used for quantitative analyses – such as performance-related – on the system).

**EastADL** [Debruyne *et al.* 2004] is an architecture description language dedicated to automotive embedded systems, developed in the context of the EAST-EEA project and currently refined in the ATESSST project. This language is aligned on the SysML standard and

---

<sup>14</sup> <http://neptune.irit.fr/>

<sup>15</sup> <http://www.spices-itea.org/>

<sup>16</sup> <http://www.topcased.org/>

also on the automotive standard AUTOSAR<sup>17</sup>. EastADL supports vehicle feature variability of product families, vehicle environment modeling to perform validation, structural and behavioral modeling of software and hardware and requirements modeling. The language is structured in five abstraction layers from the vehicle level to the operational level. The ATESSST project aims at integrating safety analysis in EastADL.

**AADL** [SAE-AS5506 2004] is a textual and graphical ADL that provides precise execution semantics for modeling the architecture of software systems and their target platform. It has been approved and published as an international standard by the International Society of Automotive Engineers (SAE). A prototype of AADL was previously developed by Honeywell under US Government sponsorship (DARPA and others) to prove the concept. This prototype, called MetaH, has been used extensively to validate the concepts now in AADL. AADL is characterized by all the properties that an ADL should provide (composition, abstraction, reusability, configuration, heterogeneity, analysis) [Shaw & Garlan 1994]. It has substantial support for modeling reconfigurable architectures. From the analysis point of view, [Medvidovic & Taylor 2000] showed that, compared to other ADLs (e.g., ACME, C2, Darwin, Rapide, Wright), AADL/MetaH provides more advanced support for analyzing quality attributes. AADL allows analyzing the impact of different architecture choices (such as scheduling policy or redundancy scheme) on a system's properties [Feiler *et al.* 2004]. These characteristics led to its serious consideration in the embedded safety-critical industry (e.g., Honeywell, Rockwell Collins, Lockheed Martin, the European Space Agency, Astrium, Airbus) during the last years. Our work related to the integration of dependability modeling and evaluation into an MDE approach focuses on AADL. AADL is further detailed in Section II.1.

## ***1.2 Model-based dependability evaluation***

Model-based dependability evaluation is one of the two means for fault forecasting [Laprie *et al.* 1996]. It is useful from the design to the operational phases of a system, as it helps to make decisions about the architecture of the system and about the fault-tolerance policies to be implemented. Dependability evaluation can be conducted in two ways, depending on the target of the analysis:

- 1) *Ordinal evaluation*: aims at identifying and ranking failures or the alternative mechanisms considered for avoiding failures;
- 2) *Probabilistic evaluation*: aims at evaluating in terms of probabilities certain dependability attributes.

Dependability is classically evaluated based on dedicated analytical models. Some types of models are only suitable for one of the two types of analysis (i.e., ordinal or probabilistic). For example, FMEA (failure modes and effects analysis) is specific for ordinal evaluation while Markov chains are suitable for probabilistic evaluation. On the other hand, reliability block diagrams and fault trees are suitable for both types of dependability analysis. Dependability models describe the behavior of the system in the presence of faults. For probabilistic evaluation, stochastic probabilities or processes must characterize model parameters. In this dissertation, we focus on integrating probabilistic dependability evaluation into an MDE approach (based on AADL). The probabilistic dependability evaluation process requires:

---

<sup>17</sup> <http://www.autosar.org>

## CHAPTER I

- 1) The *definition of quantitative measures* to be evaluated. Dependability measures of interest are generally extracted from the system requirements.
- 2) *Model construction*. This phase consists in selecting the most adapted modeling technique allowing the evaluation of the selected measures.
- 3) *Model processing*. This phase corresponds to the computation of measures.

The following sections detail successively the three phases enumerated above.

### I.2.1 Choice of measures

The users can perceive the life of a system as an alternation between two service states:

- 1) *Correct service*: where the system delivers correctly its service;
- 2) *Incorrect service*: where the system does not deliver correctly its service.

A *failure* is the transition from correct to incorrect service and a *restoration* is the transition from incorrect to correct service. The alternance between correct and incorrect service is quantified by the following dependability measures [Avizienis *et al.* 2004]:

- 1) *Reliability*: measures the continuous delivery of correct service, or the time to failure;
- 2) *Availability*: measures the delivery of a correct service with respect to the alternation between correct and incorrect service;
- 3) *Maintainability*: measures the continuous delivery of incorrect service, or the time to restoration after the last failure;
- 4) *Safety*: measures the time to catastrophic failure. This measure is an extension of reliability.

The measures to be evaluated depend on the system requirements. For example, web services need high availability, satellites need high reliability and transportation systems need to be safe. Our dependability modeling framework does not target a particular measure. We give general guidance for building models allowing the evaluation of the measures enumerated above.

### I.2.2 Model construction

Depending on the target analysis and on the complexity of the system, one can consider using a particular type of analytical model. Analytical models supporting dependability evaluation are grouped in two classes: state space models (e.g., Markov chains) and non-state space models (e.g., fault trees).

Non-state space models are concise, rather easy to build and have efficient processing methods. For example, fault trees [Barlow *et al.* 1975] may be used even at a high level of abstraction, as soon as scenarios of occurrence of a feared event are identified. A fault tree consists of successive levels of events connected by logical gates (**AND**, **OR**). It is noteworthy that dynamic fault trees [Bechta Dugan *et al.* 1992] extend standard ones by defining additional dynamic gates, able to model more complex behavior. The undesirable event analyzed is the root of the tree. The principle for building a fault tree is that each event is broken down up to events considered elementary that are independent of others and whose probabilities can be estimated. The principle for analyzing a fault tree is to identify the

minimal cuts, which are sets of events that can lead to the undesirable event at the root of the tree. One may compute the probability of occurrence of the root event based on the probabilities of occurrence of the elementary events. This kind of computation allows obtaining reliability and safety measures. Non-state space models are not suitable for capturing strong dependencies between system components. In addition, they are not well adapted for measuring system availability.

Dependability evaluation of complex systems requires not only modeling the failure but also the repair behaviour of hardware and software system components and the numerous interactions between them, resulting in complex models. Depending on the dependability measures to be evaluated, the modeling level of detail can furthermore increase this complexity. State-space models, in particular homogeneous Markov chains, are commonly used to model the dependability of systems. The latter are able to capture various functional and stochastic dependencies among components and allow evaluation of various measures related to dependability and performance (i.e. performability measures) based on the same model, when a reward structure is associated to them. The resulting model is referred to as a Reward Markov model.

To facilitate the generation of large state-space models, higher-level specification languages such as GSPNs (Generalized Stochastic Petri Nets with timed and immediate transitions) are generally used. GSPNs are recognized as a powerful modeling tool for performance and dependability evaluation of concurrent and distributed systems. Thus, they are supported by numerous tools for dependability evaluation, (e.g., Surf-2 [Béounes *et al.* 1993], Möbius [Deavours *et al.* 2002], Sharpe [Hirel *et al.* 2000], GreatSPN [Bernardi *et al.* 2001], SPNP [Ciardo & Trivedi 1993b]). They can be automatically converted to Markov chains. In addition, their advantages are that they allow modular and hierarchical modeling for component-based systems and also provide means for structural verification of the model. Such verification support facilities are very useful when dealing with large models. However, GSPNs have a major drawback related to the difficulty of the model construction for large and complex systems. Several research contributions, which may be grouped in two classes, have been published to address this problem [ReSIST 2006]:

- *Model construction* techniques focusing on mastering the model complexity by defining rigorous construction rules. The general idea of these techniques is to build the model of a system by composing sub-models corresponding to components or functions of the system. Examples are [Bondavalli *et al.* 1999b, Fota *et al.* 1999, Kanoun & Borrel 2000]. [Bondavalli *et al.* 1999b] addresses phased-mission systems: the upper-level models the mission phases and the lower level details the behaviour of the system inside each phase. [Fota *et al.* 1999] and [Kanoun & Borrel 2000] present approaches for constructing a GSPN of a complex system from the GSPNs of its components taking into account the interactions between the components. These approaches are referred to as *block modeling approach* and *incremental approach* respectively.
- *Model generation* techniques focusing on automating model transformation from a higher level description language. Examples of transformations from UML models to GSPNs are [Majzik & Bondavalli 1998b, Bondavalli *et al.* 2001, Bernardi & Donatelli 2003]. [Majzik & Bondavalli 1998b] and [Bondavalli *et al.* 2001] (the HIDE project) define a three-step transformation: the first step deals with extracting relevant dependability information from the UML model. The second step allows definition of a general Timed Petri net. The latter is translated to the specific input formats of PN tools during the third step. [Bernardi & Donatelli 2003] help the PN construction by providing reusable PN models for some UML classes and by suggesting a structure of interaction of the model components.

Our work fits into the category of model generation techniques, as our AADL-based dependability modeling framework includes an automatic AADL to GSPN transformation. We take advantage of the ability of GSPNs to express modularity and complex interactions and, at the same time, we allow them to be automatically generated to hide the complexity of their generation from the end user. GSPNs are presented from a technical viewpoint in Section II.2.2.

### 1.2.3 Model processing

There are two main problems related to model processing:

- 1) *model largeness*: due to the inherent complexity of the systems to be modeled and to the modeling details taken into account. In addition, Markov chains obtained by automatic generation from GSPNs are usually not optimal.
- 2) *stiffness*: due to the different orders of magnitude between the rates of failure-related events and the rates of fault-tolerance-related parameters.

Several techniques have been published to address model largeness. They can be grouped into two categories as suggested in [Trivedi et al. 1994]: *largeness avoidance* and *largeness tolerance*.

Largeness avoidance addresses model processing in an exact way or using approximate solutions [Balbo *et al.* 1988, Ciardo & Trivedi 1993a, Ciardo & Miner 1999]. These techniques are based on processing small sub-models in isolation and then integrating the results in a single small overall model. They lead to a gain in memory (by avoiding complete storage of the model) and in computation time. From a practical point of view and to the best of our knowledge, most of these techniques are efficient when the sub-models are loosely coupled and become hard to implement when interactions are too complex.

Largeness tolerance aims at improving the numerical algorithms for processing large GSPNs by defining reduction rules allowing the elimination of immediate transitions and vanishing markings [Blakemore 1989, Chiola & Donatelli 1991, Ajmone Marsan *et al.* 1995]. These reduction techniques can be applied to models obtained from model construction and generation techniques.

Similarly to the state-space explosion problem, stiffness has been addressed by *stiffness avoidance* techniques [Bobbio & Trivedi 1986] (that target processing methods that remain stable for stiff models) and *stiffness tolerance* techniques [Bolch *et al.* 1998] (that remove stiffness by processing non-stiff submodels using aggregation and disaggregation techniques).

Model processing is very important, as it has a great impact on the results of the dependability analysis. It is thus necessary to use performant and accurate processing techniques. In this dissertation, we rely entirely on existing methods and tools for processing the dependability model. Thus, we do not discuss further the related issues. In particular, our AADL-based dependability modeling framework focuses on building the AADL dependability model in a systematic way, and then on transforming this model into a GSPN that can be processed by existing tools.

## 1.3 Examples of analyses integrated into languages for MDE

Software architecture modeling for dependability analysis has received growing interest during the last two decades. Early approaches focused on the development of analytical

models to analyze the sensitivity of application reliability to the software structure and the reliabilities of its components (see e.g., [Bechta Dugan & Lyu 1995, Laprie *et al.* 1995] and the survey presented in [Goseva Popstojanova & Trivedi 2001]). More recently, the emergence of component-based and model-driven engineering led to the proliferation of research activities on methodologies allowing the analysis of quality attributes (e.g., performance- and dependability-related characteristics) based on general-purpose architectural models, that allow several analyses to be performed on the same model. Early approaches to analyse quality attributes at architectural level include Attribute-Based Architectural Styles [Klein *et al.* 1999]. Besides the classical features of an architectural style (i.e., component types and their topology and interactions, and benefits and drawbacks of using the style), an Attribute-Based Architectural Style also includes specification of a quality attribute described by measures, stimuli, properties and a known analytical model for the attribute. The Architecture Tradeoff Analysis Method [Kazman *et al.* 1999] is based on the use of Attribute-Based Architectural Styles. The goal of this method is to discover risks related to architectural decisions rather than to provide quantitative analyses. More recently, contributions started to focus on integrating verification and quality attribute analyses in languages that support model-driven engineering (MDE) approaches. MDE approaches may use several models, each of them representing a different view of the system. These models may even be specified in different modeling languages. Thus, some contributions aim at defining generic methods able to support several available modeling languages and technologies. [Radjenovic & Paige 2006] proposes the Architecture Information Modeling Language that includes a generic metamodel supporting several modeling languages from the same platform. The latter also supports change control mechanisms to keep models representing views of a system synchronized. In addition, it provides support for safety cases by using the Fault Propagation and Transformation Calculus [Wallace 2005].

Most of the published contributions towards integrating verification and quality attributes analyses in languages for MDE have been focused on the UML, since UML is a general-purpose language. However, significant efforts also targeted EastADL (mainly through the ATESSST – Advancing Traffic Efficiency and Safety through software technology – project) and AADL. Usually, the contributions propose to enhance a general-purpose model with analysis-specific information and then to derive from the enhanced model a specific analysis model. The next subsections briefly report work related to ours, aiming at integrating several types of analyses into MDE approaches based on UML, SysML, EastADL and AADL.

### **1.3.1 UML-based analyses**

A significant amount of research has been carried out in order to integrate analyses related to dependability, performance or property verification into UML.

Obtaining dependability analyses from UML models preoccupied many researchers. For didactical reasons, we group the contributions towards this goal in two categories:

- 1) Those proposing model transformations from UML diagrams to GSPNs.
- 2) Those focusing on obtaining other types of dependability evaluation models, such as fault trees or reliability block diagrams.

Before mentioning the approaches in the second category, we detail the first category, which is closer to our approach. Some related work addressing performance evaluation and property verification is discussed at the end of this section.

The European project HIDE [Majzik & Bondavalli 1998a], [Bondavalli et al. 1999a], [Bondavalli *et al.* 2001 ] proposed a method to automatically analyse dependability based on UML models. This approach is based on transforming structural UML diagrams (use case, class, object, deployment diagrams) into GSPNs. In addition, behavioral diagrams such as statecharts are used to derive how failures of objects and nodes lead to failure of (sub)-systems. The transformation is defined in three steps: the first step deals with extracting relevant dependability information from the UML model. The second step allows definition of a general Timed Petri net that is translated in specific Petri Net tools for analysis during the third step. Technical details about the transformation technique can be found in [Majzik & Bondavalli 1998c], [Majzik et al. 2003]. An intermediate model is constructed before generating the dependability model. In order to obtain the intermediate model, UML elements are mapped into hardware, software or composite elements while relations between elements are mapped into one of the following types of dependencies: *uses service of*, *interacts with*, and *is composed of*. The target Petri net consists of three sub-nets: one for the fault activation, another for propagation of basic events and a third that models service restoration.

[Huszerl et al. 2002] focuses on quantitative dependability analyses of UML behavioral models for embedded systems. In order to perform dependability analyses, UML models must be extended with explicit categorization of failure states and events, and with probabilistic information. The standard extension mechanisms of UML allow expressing the additional necessary information through tagged values and stereotyped states and events. The proposed analysis method is based on an UML to Stochastic Reward Nets transformation algorithm. The latter has been chosen as it generalizes classical Petri Nets by rewards (measures) and by assigning guards and distributions to transitions. The transformation approach is modular and guided by composition rules. It is based on the one used in the HIDE project. Faults such as loss, duplication or corruption of events are explicitly modelled.

Besides the contributions detailed above that propose transformations from UML to Petri nets for dependability analysis, there are other initiatives that consider model transformations from UML to other analytical models. The purpose of [Fernandez Briones *et al.* 2006] is to help software engineers to find the software architecture that best meets safety and cost requirements by integrating safety modeling into UML designs used by the software engineering teams. They propose to automatically generate fault trees and FMECA (failure modes, effects and criticality analysis) models from UML models using a new safety profile. The automation is based on Eclipse and the Eclipse Modeling Framework. [Lu *et al.* 2005] proposes to model fault tree elements (events, gates and edges) in an UML model by using a UML profile for safety. This makes possible to build the fault tree and the system's model together. The work presented in [Giese et al. 2004] aims at performing hazard analysis of UML models described by restricted component and deployment diagrams. Fault trees are generated after a hierarchical failure classification. [Zarras et al. 2004] describes a way to analyse dependability of web services specified in UML. First, the UML model is enriched with properties characterising the failure behaviour of elements of the web services. Then, it is mapped to block diagrams, fault trees and Markov chains. [Pai & Bechta Dugan 2002] presents a framework facilitating automatic dependability analysis based on UML class, object and deployment diagrams. This framework is based on deriving dynamic fault trees from the UML diagrams. [Zarras & Issarny 2000] focuses on assessing software reliability at the architectural level. Reliability Block Diagrams are derived from UML collaboration and statechart diagrams.

Besides dependability, modeling approaches aimed at supporting performance-related analyses based on UML have also been investigated. They are generally based on transforming a subset of UML diagrams into Petri nets. [López-Grao *et al.* 2002, Merseguer



& Campos 2004] propose a transformation from each UML behavioral diagram into GSPN. The use case diagram is a basis for computing the system usage by actors. Statechart and activity diagrams are transformed into GSPN that represent a performance model of the whole system. The sequence diagram is transformed into a GSPN representing a performance model of a particular execution scenario of the system. The performance evaluation process comprises three steps: the extension of UML diagrams with performance annotations, the transformation to GSPN and the composition of submodels. A tool implements the transformation and saves the GSPN in the format of the GreatSPN tool [Bernardi *et al.* 2001]. [Bernardi *et al.* 2002] focuses on transforming statechart and sequence diagrams into GSPN. The transformation is performed separately for each diagram and then composition is used to obtain the complete model. [King & Pooley 1999] presents a technique for transforming UML statechart and collaboration diagrams into SPN (stochastic Petri nets). Statecharts together with collaboration diagrams provide a full description of how the system works. On the one hand the statechart diagram is a state diagram for each object and on the other hand the collaboration diagram shows how objects interact. The generation of a stochastic Petri Net model from the UML model is performed mainly by the association of the states of the statechart diagram to places in a Petri Net, and of the state changes to transitions in the Petri Net. The Collaboration diagrams guide the combination of individual Petri Nets. [Mitton & Holton 2000] proposes a method of mapping UML statecharts onto the Stochastic Process Algebra PEPA in order to assess performability.

Approaches for integrating property verification into UML are based on generating non-stochastic models aimed for qualitative analysis. For instance, the works presented in [Saldhana & Shatz 2000] and [Baresi 2002] are meant to help the validation of UML specifications whereas [Elkoutbi & Keller 1998] and [Elkoutbi *et al.* 2002] propose approaches allowing scenario integration for systems including objects described in UML.

### **I.3.2 SysML-based analyses**

To the best of our knowledge, there are only few contributions aiming at integrating quality attribute analyses to SysML. They mainly focus on performance analysis. In [Jarraya *et al.* 2007], the goal is to analyse performance of SysML activity diagrams. The SysML model is transformed into discrete-time Markov chains (DTMC), which are analysed using the PRISM model checker [Kwiatkowska *et al.* 2005]. [Viehl *et al.* 2006] presents an approach for formal and simulation-based performance analysis of systems specified with UML/SysML. In this approach, sequence diagrams are used for the definition of control flow and timing. An architectural model of the system, using SysML assemblies, describes the mapping of activities and communication to components. For performance analysis, the SysML model is transformed into a communication dependency graph.

### **I.3.3 EastADL-based analyses**

EastADL is the result of efforts of the automotive industry towards creating an ADL that provides a systematic way of integrating functional design and implementations of automotive systems, while taking into account specific analyses, information management support and lifecycle organization. Examples of analyses of interest for automotive computer control systems are model-checking of behavioral properties, performance (end-to-end response time, worst-case execution time, precedence, resource sharing), reliability and safety analyses. The ATESSST project aims at integrating in EastADL explicit support for these analyses. [Chen *et al.* 2007] provides an initial concept of extending EastADL for safety

analysis through HiP-HOPS [Papadopoulos & McDermid 1999]. HiP-HOPS is a method for safety analysis through a combination of classical techniques such as hazard analysis, failure modes and effects analysis, and fault tree analysis. According to the initial roadmap of the project, EastADL should integrate a hierarchical model of errors, described by a metamodel.

### I.3.4 AADL-based analyses

AADL allows describing separately the analysis-related information that may be plugged into the architectural model. Since this information is not embedded in the architectural model, the user can easily unplug or replace it. This feature enhances the reusability and the readability of the AADL architectural model that can be used unmodified for several analyses (formal verification of functional properties [Farines *et al.* 2003], schedulability [Sokolsky *et al.* 2006] and memory requirements [Singhoff *et al.* 2005], fault tree analysis [Joshi *et al.* 2007], resource allocation with the Open Source AADL Tool Environment (OSATE)<sup>18</sup>, search for deadlocks and un-initialized variables with the Ocarina toolset<sup>19</sup>).

Honeywell has been at the origin of significant efforts for integrating dependability analyses first to MetaH [Vestal 1998] and then to AADL, with the definition of the AADL Error Model Annex [SAE-AS5506/1 2006b], that allows associating *error models*, representing behaviours in the presence of faults, with AADL architectural models. Furthermore, [Binns & Vestal 2004] considers the generation of a safety model of a system modeled with AADL by composing the models of its subcomponents. An abstract model may also be directly associated to the component. The abstract model is intended to be an acceptable approximation of the concrete model, generated by composition. The authors explore the relationship between abstract and concrete models. Very recently, [Joshi *et al.* 2007] presented a proprietary prototype tool for generating fault trees from AADL models. The generation is achieved in three steps: extraction of a system instance error model, generation of an internal fault tree and formatting the fault tree for a specific analysis tool. This tool is built as a set of Eclipse plug-ins and uses the OSATE support for traversing AADL models. From a critical viewpoint, these contributions do not tackle issues related to model reusability and they do not detail the derivation of the dependability evaluation model from the AADL model enriched with error models.

Other published work on analyses using AADL has focused on the extension of the language capabilities to support formal verifications. For example, the COTRE project [Farines *et al.* 2003] provides a design approach bridging the gap between formal verification techniques and requirements expressed in ADLs. AADL system specifications can be imported in the newly defined COTRE language. A system specification in COTRE language can be transformed into timed automata, Time Petri nets or other analytical models. Also, a transformation from AADL models to Colored Petri Nets, aiming at formally verifying certain properties through model checking, is presented in [Hugues *et al.* 2007]. [Sokolsky *et al.* 2006] and [Singhoff *et al.* 2005] aim at analysing schedulability based on AADL models, respectively by formal analysis based on state-space exploration and by simulation.

---

<sup>18</sup> <http://www.aadl.info/OpenSourceAADLToolEnvironment.html>

<sup>19</sup> <http://ocarina.enst.fr>

## ***1.4 Proposed AADL-based dependability modeling framework***

The AADL Error Model Annex provides features with precise semantics to be used for describing dependability-related characteristics in AADL models (faults, failure modes, repair policies, error propagations, etc.). However, at the current stage, no methodology and guidelines are available to help the developers in the use of the proposed notations to describe complex dependability models reflecting real-life systems with multiple interactions and dependencies between components.

Our goal is to provide means that facilitate the evaluation of various quantitative dependability measures (e.g., reliability, availability) based on AADL models. To this end, we propose:

- 1) a dependability modeling framework based on a **structured method for the construction of the AADL model** and on model transformation from AADL to dependability evaluation models.
- 2) a set of **AADL fault-tolerance patterns** that enable model reusability.

For complex systems, the main difficulty for dependability model construction arises from dependencies between the system components.

A structured approach is necessary to model dependencies in a systematic way, to avoid errors in the resulting model of the system and to facilitate its validation. In our approach, presented in [Rugina *et al.* 2006c, Rugina *et al.* 2006b, Rugina *et al.* 2007], the AADL dependability-oriented model is built in an iterative way. More concretely, in the first iteration, we build the model of the system's components, representing their behavior in the presence of their own faults and repair events only. They are thus modeled as if they were independent. In the following iterations, we introduce dependencies between the component models in an incremental manner. This approach is further detailed in subsections I.4.1, I.4.2 and I.4.3.

Dependencies are of several types, identified in [Kanoun & Borrel 2000]: structural, functional, those related to the fault-tolerance and those related to the recovery and maintenance policies. Exchange of data or transfer of intermediate results from one component to another is an example of functional dependency. The fact that a thread runs on a processor induces a structural dependency between the thread and the processor. Changing the operational mode of a component according to a fault-tolerance policy (e.g., leader/follower) represents a fault-tolerance dependency. Sharing a maintenance facility between several execution platform components leads to a maintenance dependency. Having to follow a strict recovery order for application components is an example of recovery dependency.

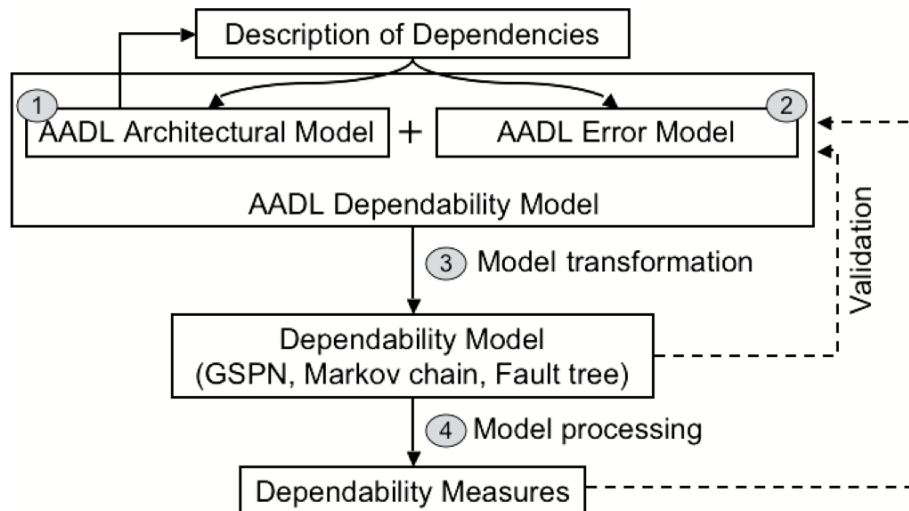
Functional, structural and fault-tolerance dependencies can be grouped into an *architectural dependency class*, as they are triggered by physical or logical connections between the dependent components at architectural level. On the other hand, maintenance and recovery dependencies are not always visible at the level of the functional architectural model. This is for example the case when maintenance facilities are shared between several components and the maintenance facilities are not modeled in the functional architectural model.

Fault-tolerance dependencies and mechanisms can be modeled in AADL and stored in libraries of reusable AADL dependability models. Then they may be used as *patterns*. To be used in a particular system, a pattern must be instantiated and customized if necessary. A master-slave redundancy pattern in AADL has been presented in [Feiler *et al.* 2004]. It aimed at easing the understanding of the functional architecture by clearly showing what is

replicated in the architectural model and what the active system components are. Our patterns additionally include a customizable layer of dependability-related information (error/failure and recovery behavior) and of dynamics necessary for evaluating dependability measures. Our work complements other existing initiatives that investigated the development of fault-tolerance patterns based on object-oriented approaches and UML ([Islam & Devarakonda 1996, Beder *et al.* 2000, Tichy *et al.* 2004]) or other languages ([Kehren *et al.* 2004]). The use of patterns and, more generally, dependability modeling at the architectural level favors the reduction of recurrent dependability modeling work and enhances the understandability of the dependability model (thus reflecting the modularity of the architecture) [Laprie & Kanoun 1996] and allows the designer to reason about fault-tolerance and to assign exceptional behavior responsibilities among components [de Lemos 2006]. At the same time, we can evaluate dependability measures (i.e., availability, reliability, safety) based on the AADL model. This allows predicting the effects of particular architectural decisions on the dependability of the software system [Klein *et al.* 1999]. Other analyses (e.g., related to performance) may be performed on the same AADL model, to highlight for example the tradeoff between the benefits of a certain fault-tolerance pattern and its impact on the application's performance [Feiler *et al.* 2004].

### I.4.1 Overview of our modeling framework

An overview of our iterative modeling framework, which is decomposed in four main steps, is presented in Figure I-1.



**Figure I-1.** Modeling framework

The *first step* is devoted to the modeling of the system architecture in AADL (in terms of components and operational modes of these components). This AADL architectural model may be available if it has been already built for other purposes in the MDE process.

The *second step* concerns the building of the AADL error models describing the dependability-related information associated with components of the architectural model. The error model of the system is a composition of the set of components' error models, taking into account the dependencies between these components.

The architectural model and the error model of the system form a dependability-oriented AADL model, referred to as the *AADL dependability model* in the rest of the dissertation.

Modeling a dependency may either require to add new information into the model or to modify the existing model and to add new information (i.e., states and propagations).

The *third step* aims at building a dependability evaluation model, such as those presented in Section I.2.2 (e.g., a fault tree, a Markov chain or a GSPN), from the AADL dependability model, based on model transformation rules. Given the advantages of GSPN over fault trees and Markov chains (i.e., the possibilities of performing structural model verification, and of building a modular model, reflecting the structure of the architectural model), we focus on deriving GSPNs from AADL models. The transformation rules are systematic in order to allow the automation of the transformation. Also, the resulting GSPN is tool-independent (i.e., we do not use tool-specific features or predicates). We have developed a tool prototype to automate our transformation. Its implementation follows a MDE approach based on Eclipse and the Eclipse Modeling Framework, similarly to the approaches of [Fernandez Briones *et al.* 2006] and [Joshi *et al.* 2007] that aim at obtaining fault trees respectively from UML and AADL models. Our tool saves the GSPN both under a generic XML/XMI format and under the format of the dependability modeling and evaluation tool Surf-2 [Béounes *et al.* 1993]. Our prototype is presented in Appendix C.

The *fourth step* is devoted to the dependability evaluation model processing to evaluate quantitative measures characterizing dependability attributes. This step is entirely based on existing processing algorithms and tools. Therefore, it is not presented in this dissertation.

To obtain the AADL dependability model, the user must perform the first and second steps described above. The third step is automatic in order to hide the complexity of the GSPN from the user. Chapter IV provides all the transformation rules that are necessary and sufficient to build such an automated transformation tool.

The iterative approach can be applied to the first two steps only or to the first three steps together. In both cases, the AADL dependability model is updated at each iteration. In the latter case, the AADL dependability model is validated against its specification, based on the analysis and validation of the GSPN model, after each iteration.

The two following subsections present briefly (1) the AADL dependability model construction (first and second steps of our framework) and (2) the model transformation process (third step of our framework). These topics are further detailed respectively in Chapters III and IV.

## I.4.2 The AADL dependability model

Architectural dependencies are deduced from the AADL architectural model, based on the analysis of the connections and bindings present in this model. To these dependencies one has to add recovery and maintenance dependencies that are not represented in the AADL architectural model. We recommend to summarize the full set of dependencies in a *dependency block diagram* to provide a global view of the system components and interactions. In the dependency block diagram, each component and each dependency are represented as distinct blocks. Blocks are connected through arcs. Their directions identify the directions of dependencies. This diagram and the AADL architectural model are used to build the AADL error model incrementally. Once the AADL error models of components considered independently are built, the dependencies are added gradually, based on the description of dependencies.

Considering maintenance and recovery policies may lead to the addition of components in the architectural model. Also, modeling fault-tolerance policies (from scratch or by instantiating a fault-tolerance pattern) may lead to adjusting the architectural model, since the

architectural model depends on the fault-tolerance policy (which determines for example the number of replicas and the decision-making components).

### **I.4.3 AADL to GSPN model transformation**

The GSPN model of the system is built from the transformation of the AADL dependability model following a modular approach. The transformation rules defined in Chapter IV are systematic in order to facilitate their automation. In this way, the complexity of the GSPN generation is hidden from the user. The resulting GSPN has the same structure as the dependency block diagram: it is formed of a set of interacting subnets, where a subnet is associated with a component or a dependency block identified in the dependency block diagram. Two types of GSPN subnets are distinguished:

- 1) A *component net* is associated with each component and describes the component's behavior in the presence of its own faults and repair events.
- 2) A *dependency net* models the behavior associated with the corresponding dependency.

The modular structure of the GSPN allows the user to validate the model progressively, as the GSPN is enriched with a subnet each time a new dependency is added in AADL model.

## **I.5 Conclusion**

Besides allowing communication between different stakeholders, MDE aims at better integrating quality attributes analyses into the development cycle. In particular, this allows performing tradeoff analyses with respect to architectural decisions.

Before giving an overview of our modeling framework, we have reported the major initiatives towards MDE engineering processes. Each initiative is based on the use of a modeling language or of a set of modeling languages and intends to integrate quality attributes analyses in the development cycle. We have then focused on the elements to take into account for dependability analysis purposes. The dependability modeling and analysis process is based on the definition of meaningful dependability measures, and on the construction and processing of the dependability model. Our framework has two sources of inspiration: our iterative dependency-driven modeling approach is inspired by largeness tolerance techniques published in the dependability modeling and evaluation literature, while the model transformation from AADL to GSPN is inspired by MDE, since one of the principles of MDE is to hide complexity and to achieve a correct-by-construction system by using as much as possible automatic model transformations. Finally, we have surveyed related work aiming at integrating different analyses to the modeling languages adopted by the MDE initiatives.

In Chapter II, we give further details on AADL and GSPNs, the two languages used in our framework. Chapter III details the AADL dependability model construction (first and second steps of our modeling framework presented in Figure I-1) while Chapter IV presents the AADL to GSPN model transformation rules (third step of our modeling framework presented in Figure I-1). These rules have been implemented in an Eclipse-based tool that is the subject of Appendix C. In Chapter V, we show an example of application of our framework to a case study issued from a real-life system.



---

## II Background

---

This section is devoted to the introduction of the two modeling languages used in our framework. Section II.1 gives an overview of AADL (Architecture Analysis and Design Language) by describing relevant characteristics for the comprehension of our modeling approach and of the AADL to GSPN transformation rules. Section II.2 gives an overview of Petri nets with emphasis on Generalized Stochastic Petri Nets, the variant of Petri nets of interest in the context of dependability evaluation.

### II.1 AADL

AADL is an Architecture Description Language (ADL) designed for the specification, analysis, and automated integration of performance-critical, embedded, real-time systems. The Society of Automotive Engineers standardized it in 2004 [SAE-AS5506 2004]. The development and the standardization of AADL is based on MetaH [Vestal 1998], an ADL accompanied by a non-commercialized toolset, developed at Honeywell Technology Laboratories under the sponsorship of the US Defense Advanced Research Projects Agency (DARPA) and US Army Aviation and Missile Command (AMCOM).

The AADL standard allows describing both software architectures and execution platform architectures using one of its three complementary syntaxes:

- *Textual*: this is the reference syntax. It is more expressive than the graphical syntax and clearer than the XML syntax. It allows a complete description of an AADL model.
- *Graphical*: this syntax is complementary to the textual syntax, as it provides a global view of a system's architecture. However, the graphical syntax is not sufficient to describe complex and large systems.
- *XML*: this syntax is as expressive as the textual one and is designed to be a tool interchange format.

For clarity reasons, we mainly use the graphical syntax throughout this dissertation. We only introduce the textual syntax when it is necessary for the reader to have a more detailed view of the AADL architectural model.

An AADL description consists of a set of component declarations. These declarations can be instantiated and connected to form a particular system architecture description. System descriptions in AADL allow a system designer to analyze system schedulability, sizing, dependability, and other quality attributes<sup>20</sup> and to evaluate architectural tradeoffs. Most of these analyses of quality attributes require that the architectural model be enriched with analysis specific information. To this end, the AADL language has been designed to be extensible through annexes. The Error Model Annex is a standardized annex [SAE-AS5506/1 2006b] that complements the description capabilities of the core AADL language by providing a textual syntax with precise semantics to be used for describing dependability-

---

<sup>20</sup> Quality attributes are also referred to as *non-functional properties* in the literature, in opposition to *functional properties* that directly address the system's functions.



related characteristics in AADL architectural models (faults, failure modes, repair policies, error propagations, etc.). An AADL architectural model can be annotated with dependability-related information and the resulting annotated model can be used as an input to dependability analysis during different phases of the development cycle. A detailed guide for dependability modeling using AADL is available in [Feiler & Rugina 2007].

The remainder of this section is structured as follows. Section II.1.1 briefly presents the core language while section II.1.2 gives an overview of the Error Model Annex.

## II.1.1 Core AADL language

In AADL, systems are modeled as hierarchical collections of interacting application components and a set of execution platform components. The application components are bound to the execution platform.

AADL allows describing “bounded” architectures, i.e., it does not allow modeling a system that may have an unknown or a variable number of components. It allows however the definition of reconfigurable architectures through the use of the concept of operational mode. The components and connections of an architectural model may be declared as active in some of the operational modes of the system and inactive in some other operational modes.

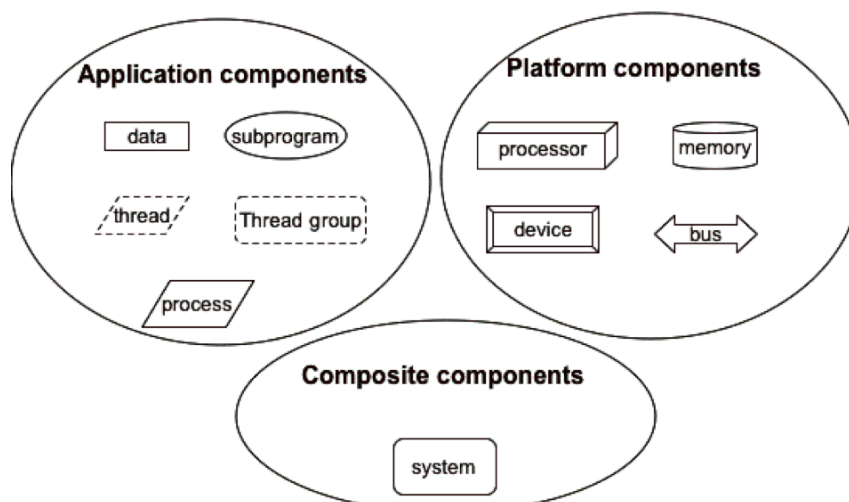
The remainder of this section first describes components - the AADL building blocks - and then introduces architecture configurations.

### II.1.1.1 Components

AADL defines three component categories:

- application components: processes, threads, subprograms, data
- execution platform components: processors, memory, buses and devices
- composite components: system

Figure II-1 shows the graphical notation for the different kinds of components.



**Figure II-1.** Component categories - AADL graphical notation

Each AADL system component has two levels of description: the *component type* and the *component implementation*. The type describes how the environment perceives that

component, i.e., its properties and features. Examples of features are **in** and **out** ports that represent access points to the component. Three categories of ports are distinguished in AADL: *event ports*, *data ports* and *event data ports*, modeling respectively control, data, and control and data flows. These port categories are represented in Figure II-2 using the AADL graphical notation.



**Figure II-2.** *Port categories - AADL graphical notation*

Interactions between AADL components are described by:

- *Connections*: A connection links an **out** port of a component to an **in** port of another component or to an **out** port of the enclosing component. The origin and destination of a connection must be ports of the same category. Depending on the category of the ports, the connection represents a control flow (in the case of event ports), a data flow (in the case of data ports) or a control and data flow (in the case of event data ports) between components.
- *Bindings* of application components to platform components: For example, a data component may be bound to a memory component or a thread to a processor.

One or more component implementations may be associated with the same component type, corresponding to different implementation structures of the component in terms of subcomponents, connections (between subcomponents' ports) and operational modes. Figure II-3-a shows a component type, identified by the name *ComputingUnit*, having an **in** data port named *input* and an **out** data port named *output*. Figure II-3-b and Figure II-3-c show examples of different implementations (*ComputingUnit.join* and *ComputingUnit.Split*) in terms of subcomponents for the component type *ComputingUnit*. In Figure II-3, we assume that *ComputingUnit* must compute a result value based on the data received through the port *input*. The result value is made visible outside *ComputingUnit* through the port *output*. The AADL graphical notation is used to represent *ComputingUnit* as a process and its subcomponents as threads. **In** ports are represented as arrow-heads pointing towards the component and **out** ports are represented as arrow-heads pointing towards outside. Connections are represented as lines connecting ports.

In the implementation *ComputingUnit.join* of Figure II-3-b, three threads are assigned the three different steps of the computation. The last step is assigned to the thread *Compute3* that computes the global result based on a partial result from *Compute2*. It sends the result value out through its out data port *o*, which is connected to the port *output* of *ComputingUnit*. In the implementation *ComputingUnit.split* of Figure II-3-c, an additional thread is used to achieve in parallel the second step of the computation. We assume that the thread *Compute3* first computes a mean value of the two results sent respectively by *Compute21* and *Compute22* and then computes the global result, based on the mean value. *Compute3* sends out the result through its out data port *o*, which is connected to the port *output* of *ComputingUnit*.

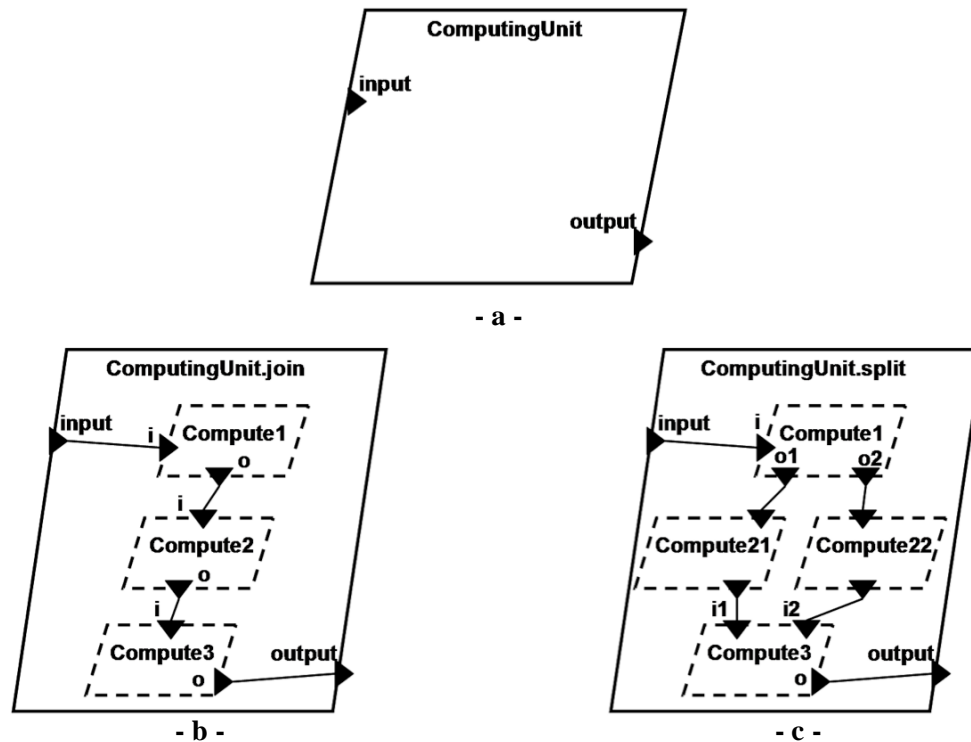


Figure II-3. Different component implementations (b and c) for the same type (a)

### II.1.1.2 Architecture configurations

Dynamic aspects of system architectures are captured with the AADL operational mode<sup>21</sup> concept. Different modes of a system or a system component represent different architecture configurations and connection topologies, as well as different sets of property values to represent changes in non-functional characteristics such as performance and reliability.

*Modes* may represent fault-tolerance modes or different phases, potentially with different dependability characteristics, in a phased-mission system. *Mode transitions* model dynamic operational behavior and are triggered by *architectural events* arriving through named event ports. Such an architectural event models either a signal generated as a result of a change related to the dependability of the system, or a signal generated at the end of a phase.

Architectural events that go through a named event port cannot be distinguished one from the other, as they do not have names. As a consequence, any architectural event going through an event port named in a mode transition triggers that mode transition. A user can specify that a mode transition is triggered by several different architectural events by naming multiple ports in a mode transition. In that case, any architectural event arriving through any of the named ports triggers that mode transition. An event port named in a mode transition can be:

- an **out** event port of a subcomponent;
- an **in** event port of the component declaring the mode transition;
- a local event port<sup>22</sup> visible only in the component that owns it and representing either a call to a pre-declared **Raise\_Event** subprogram in a thread, or an

<sup>21</sup> *Operational modes* will be referred to as *modes* in the rest of the dissertation.

<sup>22</sup> The errata document to the AADL standard provides the ability to declare such local ports.

abstraction for a port of a subcomponent that is not declared at the current level of detail of the architectural model. A local event port is denoted by *self.localPortName* in mode transitions.

Figure II-4 presents a system with modes and modal architecture configurations. This system has three components: *Component1*, *Component2* and *Component3* (represented as system components), and two modes: *m1* and *m2* (represented as hexagons). The initial mode is *m1* (it is represented in black and is pointed by an arrow coming from a black circle drawn on the edge of the system). The mode transitions from *m1* to *m2* and from *m2* to *m1* are triggered by architectural events arriving respectively through the **out** event ports *outp1* and *outp2* of *Component3*. *Component3* does not have an **in modes** statement, i.e., it is active in both modes. It receives data from outside the system. We assume that, based on the data received, *Component3* decides whether to delegate the data processing to *Component1* or to *Component2*. To put in practice its decision, *Component3* initiates the appropriate mode transitions by generating and sending events through its **out** event ports. *Component1* and the connections to and from it are active in mode *m1*. *Component2* and the connections to and from it are active in mode *m2*. Thus, the system must be in mode *m1* if *Component1* should process the data and provide the result through the **out** data port of the system, and it must be in mode *m2* otherwise.

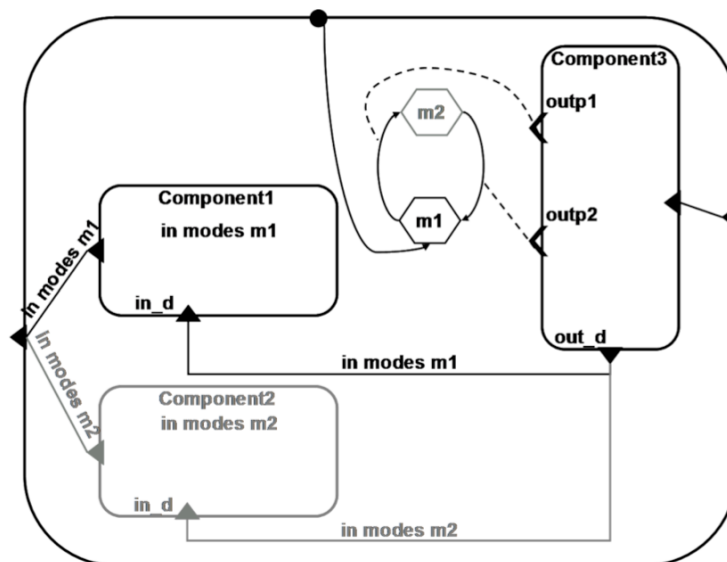


Figure II-4. System with modal architecture configurations

Semantically, a component inactive in a given mode does not communicate with the rest of the system in that mode. Similarly, a connection or a binding that is inactive in a given mode does not exist in that mode.

## II.1.2 AADL Error Model Annex

The AADL Error Model Annex supports the definition of reusable *error models* within libraries. Error models represent (stochastic) state machines that describe behavior in terms of logic error states in the presence of faults, repair events and error propagations. The user associates error models with application components, execution platform components, as well as the connections between them. When an error model is associated with a component, it is possible to customize it by setting component-specific values for the arrival rate or the

probability of occurrence for error events and error propagations declared in the error model. A user may choose to further customize the error model to a component by declaring component-specific logic guards that determine the effect of incoming error propagations on the local state machine or the effect of local events (faults, repairs) on other components.

The remainder of this section is structured as follows. Section II.1.2.1 presents the concept of error model and illustrates it for independent components. Section II.1.2.2 extends the error model of Section II.1.2.1 to take into account dependencies between components. Section II.1.2.3 gives an overview of propagation filtering and masking mechanisms that allow customizing the error models to architecture. Section II.1.2.4 focuses on the connection between error states and operational modes. Section II.1.2.5 is dedicated to error model abstractions for hierarchical systems.

### II.1.2.1 Error model for independent components

In the same way as for AADL components, error models have two levels of description: the *error model type* and the *error model implementation*. Unlike AADL components that may be declared using a textual or a graphical notation, error models may be only described textually according to the current specification of the AADL Error Model Annex (version 1.0).

The error model type is identified by a name (here *independent*) and declares a set of **error states**, **error events**<sup>23</sup> (internal to the component) and **error propagations**. **Occurrence** properties specify the arrival rate or the occurrence probability of events and propagations<sup>24</sup>. The error model implementation is identified by the type name it corresponds to (here *independent*) and by its name (here *general*). It declares **error transitions**<sup>25</sup> between states, triggered by events and propagations declared in the error model type.

Figure II-5 shows an example of error model for an independent component (without propagations). We distinguish two types of faults: temporary and permanent. A temporary fault leads the component in an erroneous state while a permanent fault leads it in a failed state. When a temporary fault is processed, we assume that the component recovers regaining its error free state. A permanent fault requires restarting the component. It models a component that may fail and that can be restarted to regain its error free state. It is noteworthy that several error models may model an independent component. For example, one can consider permanent faults only, or the fact that the restart procedure may be unsuccessful.

The user can choose an error model from a library and associate an instance of it with a component's implementation or with a connection. This association is specified through the **Model** property declared in an *Error Model annex subclause*, as shown in Figure II-6. Note that the association is only possible using the standard textual AADL Error Model Annex syntax.

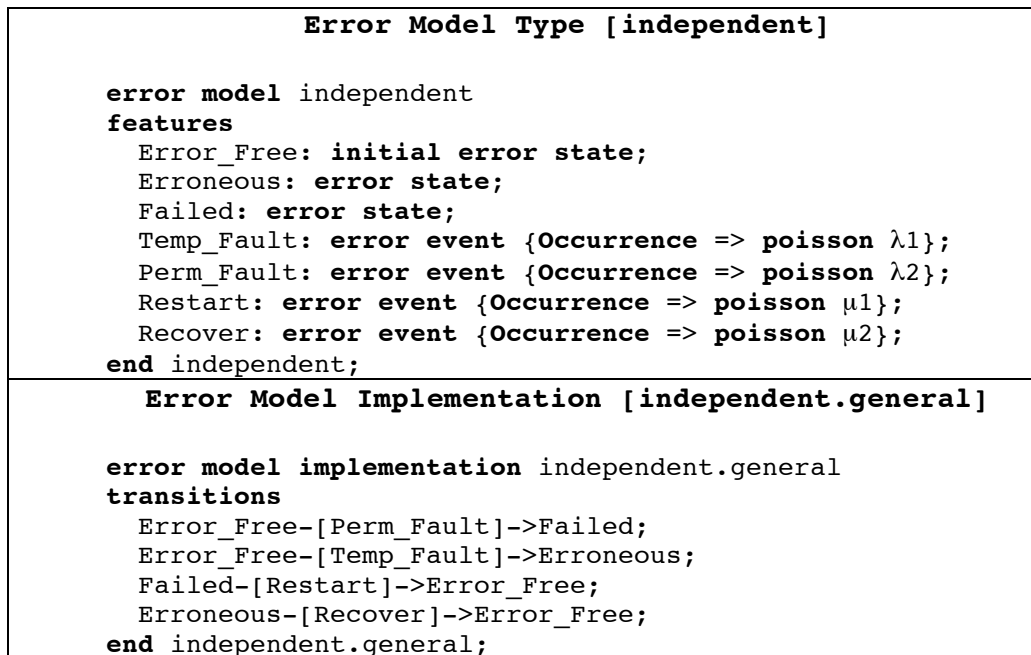
---

<sup>23</sup> Error events and error propagations are semantically different from architectural events raised in the architecture of the system. Error events and error propagations do not represent event communications through ports. By default, they do not manifest themselves as architectural events that may cause mode transitions and thread dispatches.

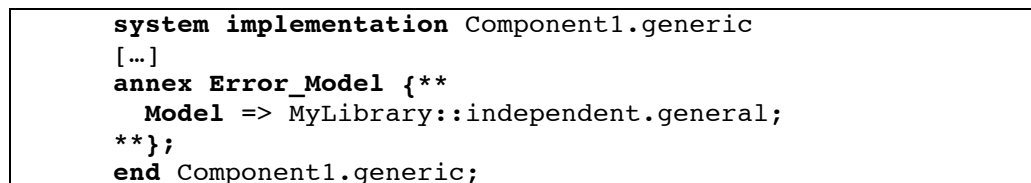
<sup>24</sup> Further on, error events and propagations whose Occurrence properties are Poisson distributions are referred to as *timed events and propagations*. Also, error events and propagations whose Occurrence properties are fixed probabilities are referred to as *immediate events and propagations*.

<sup>25</sup> Note that **error states** can model error-free states, **error events** can model repair events and **error propagations** can model all kinds of notifications. Thus, we will refer to **error states**, **error events**, **error propagations** and **error transitions** without the qualifying term **error** in contexts where the meaning is unambiguous.

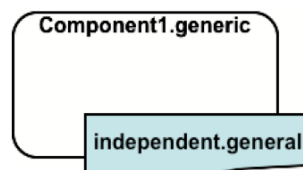
In order to give a global view of an AADL architectural model with associated error models, we chose a (non-standardized) convention to represent the association of an error model instance with a component represented in the AADL graphical syntax. Our graphical convention, that is used throughout this dissertation, is depicted in Figure II-7.



**Figure II-5.** Error model example for independent component



**Figure II-6.** Error model instance association



**Figure II-7.** Graphical notation for error model instance

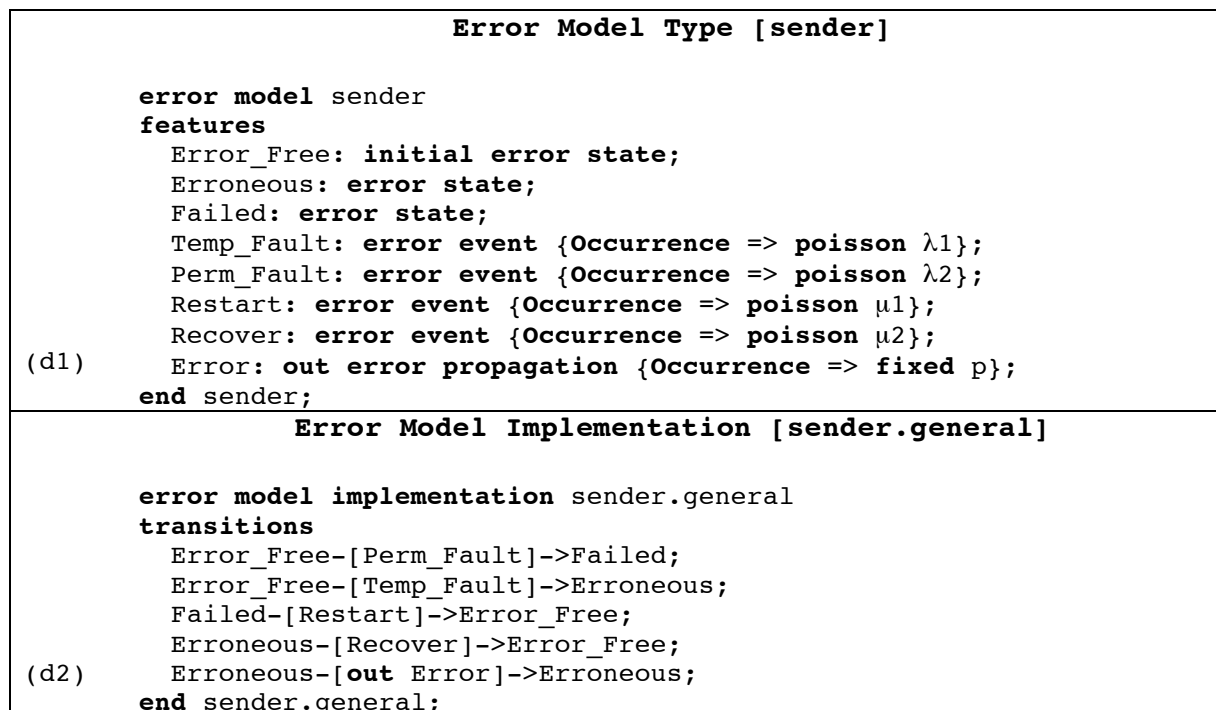
### II.1.2.2 Error model for dependent components

Interacting components at the architectural level may influence each other's behaviors through propagations. In order to describe dependencies between interacting components, the user must explicitly declare directional error propagations in the error models associated with the components involved in the dependency. An **out** error propagation occurs spontaneously and randomly according to the specified occurrence probability (or distribution of probability)

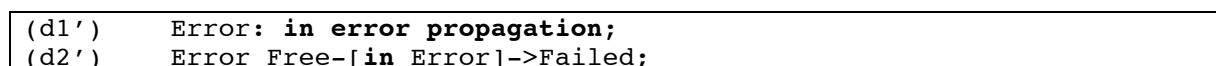
when it is named in a transition and the current state of the component is the origin state of the transition. It is broadcasted out of a component through all features connecting it to other components. By default, an **out** propagation has an impact on any receiving component that declares an **in** propagation with the same name. Note that declaring an **in out** propagation is equivalent to declaring an **in** and an **out** propagation with the same name.

The error model of Figure II-8 is an extension of the one of Figure II-5 (corresponding to an independent component). It takes into account a dependency on the sender-side, i.e., when the component with which it is associated is in an erroneous state, it can influence the behavior of components that depend on it. The error model of Figure II-8 declares an **out** propagation *Error* in the type (see line d1) and an AADL transition triggered by the **out** propagation in the implementation (see line d2). The propagation *Error* occurs with a given probability *p*.

In order to specify a recipient-side dependency, i.e., to describe the fact that a component can be influenced by *Error* propagations from other components, it is necessary to replace lines d1 and d2 of Figure II-8 by lines d1' and d2' of Figure II-9. **In** propagations are the consequences of **out** propagations from other components; therefore they do not need Occurrence properties.



**Figure II-8.** Error model example for component with interactions (sender-side)



**Figure II-9.** Error propagation (recipient-side)

It is possible to describe both a sender and a recipient-side dependency involving a unique propagation name in the same error model by declaring an **in out** propagation *Error* (by merging lines d1 and d1'). In this case, the Occurrence property only applies to the **out**

propagation *Error*. In the error model implementation, line d2 describes the sender-side dependency while line d2' describes the recipient-side dependency.

Propagation of errors between error models associated with components and connections are determined by the interactions in the architectural model. Besides a few special cases, most of the interactions fall into the three following categories:

- 1) They may be due to the fact that application components run on top of platform components. For example, **out** propagations declared in an error model associated with a processor are visible in all threads bound to that processor.
- 2) They may be due to the fact that application components interact through connections, accesses to shared data and calls to services provided by other components. For example, **out** propagations declared in an error model associated with a component can impact all components reachable through connections.
- 3) They may be due to the fact that platform components are connected to each other through shared access to buses. For example, **out** propagations declared in an error model associated with a bus arrive to all components accessing the bus.

The exhaustive list of *dependency rules* is presented in the AADL Error Model Annex, §3.5.2.

### II.1.2.3 Propagation filtering and masking mechanisms

In some cases, it is desirable to model how error propagations from multiple sources are handled. This is modeled by specifying component-specific filters and masking conditions for propagations by using **Guard\_In** and **Guard\_Out** properties associated with its features. In the remainder of this section, the **Guard\_In** and the **Guard\_Out** properties are successively presented in subsections II.1.2.3.1 and II.1.2.3.2. Finally, the two properties are compared in subsection II.1.2.3.3.

#### II.1.2.3.1 **Guard\_In**

A **Guard\_In** property allows the user to conditionally map an incoming set of propagations and error states from other components into a set of **in** propagations that may affect the receiving component. In other words, a **Guard\_In** property can specify filters for error propagations from other components used to determine if and how the state of the impacted component should be changed.

A **Guard\_In** property consists of a set of logic rules for incoming propagations. Each rule is defined as a Boolean expression referring to one or more outgoing (**out** or **in out**) propagations as well as to states from error models of components that can impact the component declaring the **Guard\_In** property. When a Boolean expression evaluates to **TRUE**, it is either:

- Mapped to an incoming (**in** or **in out**) propagation that occurs in the component declaring the **Guard\_In** property and that may trigger a state change in its error model.
- Or it is masked (if the rule is labeled with the **mask** keyword).

In addition to outgoing propagations and states of components that can impact the given component, a Boolean expression can refer, using the keyword **self**, to states of the component that declares the **Guard\_In**.



The syntax of the **Guard\_In** property in Backus-Naur form (BNF) is given in Figure II-10, without developing the Boolean expression. It is noteworthy that this property applies to **in** features of components.

```
Guard_In ::= mapping_rule {, mapping_rule}* applies to inFeature;
mapping_rule ::= (InProp_id | mask) when boolean_expr
```

Figure II-10. *Guard\_In* property syntax

Figure II-11 shows an example of **Guard\_In** property associated with an **in** port of a system component. The **Guard\_In** property (lines gi1-gi5) specifies that the component *Component1.generic* perceives:

- an incoming propagation *inError1* when an **out** propagation *Error* is visible through the **in** port *inp1* while the component connected to *inp2* is *Failed*.
- an incoming propagation *inError2* when an **out** propagation *Error* is not visible through the **in** port *inp1* while the component connected to *inp2* is *Failed*.

The other error propagation and state configurations are masked.

The **in** propagations *inError1* and *inError2* must be declared in the error model type associated with the component (named here *receiver*). The occurrences of *inError1* or *inError2* may trigger state changes in the associated error model implementation (named here *receiver.general*).

```
system Component1
features
  inp1: in data port;
  inp2: in data port;
end Component1;

system implementation Component1.generic
[...]
annex Error_Model {**
  Model => MyLibrary::receiver.general;
  Guard_In =>
gi1      inError1 when (inp1[Error] and inp2[Failed]),
gi2      inError2 when (not inp1[Error] and inp2[Failed]),
gi3      mask when others
gi4      mask when others
gi5      applies to inp1;
  **};
end Component1.generic;
```

Figure II-11. *Guard\_In* property example

Figure II-12 uses the graphical syntax to show a global view of the application of the **Guard\_In** property presented above in an architectural model. The **Guard\_In** property allows one to specify how the internal behavior of the *Component1* is influenced by the propagations and error states of the two components that communicate with it through data connections.

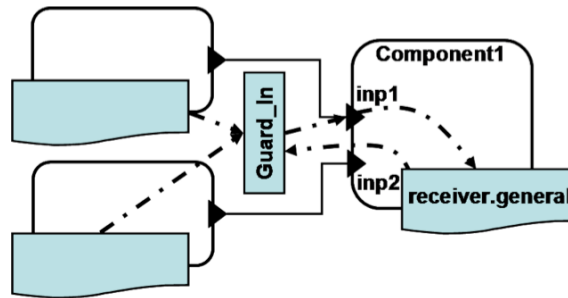


Figure II-12. Architectural view of a **Guard\_In** property

### II.1.2.3.2 **Guard\_Out**

A **Guard\_Out** property allows the user to conditionally pass through an incoming set of propagations and states as an outgoing propagation of the error model associated with the component whose implementation contains the **Guard\_Out** property.

Similarly to a **Guard\_In** property, a **Guard\_Out** property consists of a set of logic rules. Each rule is defined as a Boolean expression having the same structure as in the case of a **Guard\_In** property. When a Boolean expression evaluates to **TRUE**, it is either:

- Passed through as an outgoing (**out** or **in out**) propagation sent out of the component through the out feature to which applies the **Guard\_Out** property.
- Or it is masked (if the rule is labeled with the **mask** keyword).

The syntax of the **Guard\_Out** property in Backus-Naur form is given in Figure II-13, without developing the Boolean expression. The property applies to **out** features.

```
Guard_Out ::= passThrough_rule {, passThrough_rule}*
           applies to outFeature;
passThrough_rule ::= (OutProp_id | mask) when Boolean_expr
```

Figure II-13. **Guard\_Out** property syntax

Figure II-14 shows an example of **Guard\_Out** property associated with an **out** port of a system component. The **Guard\_Out** property (lines go1-go6) specifies that the component *Component2.generic* propagates out:

- *outError1* when two conditions hold: 1) an **out** propagation *Error* is visible through the **in** port *inp1* and 2) the component connected to *inp2* is *Failed*.
- *outError2* when only one of the two conditions stated above holds.

Any other error propagation and state configuration is not passed through as **out** propagation. If a Boolean expression of the **Guard\_Out** pass-through rule evaluates to **TRUE**, the local state machine of the component is not influenced but an **out** propagation is generated. The **out** propagations *outError1* and *outError2* must be declared in the error model type associated with the component (named here *dependent*).

Figure II-15 shows an architectural view of the application of the **Guard\_Out** property presented above. The **Guard\_Out** property allows specifying that *Component2* sends **out** propagations to a particular recipient component by associating a **Guard\_Out** property to a particular **out** port. This allows modeling Byzantine behavior (i.e., the component fails to

behave consistently when interacting with multiple other components). This is the case, for example, when a component propagates an error through one `out` port and masks it for another `out` port.

```

system Component2
features
  inp1: in data port;
  inp2: in data port;
  outp1: out data port;
  outp2: out data port;
end Component2;

system implementation Component2.generic
[...]
annex Error_Model {**
  Model => MyLibrary::dependent.general;
go1  Guard_Out =>
go2    outError1 when (inp1[Error] and inp2[Failed]),
go3    outError2 when (not inp1[Error] and inp2[Failed]) or
go4      (inp1[Error] and not inp2[Failed]),
go5    mask when others
go6      applies to outp1;
**};
end Component2.generic;

```

Figure II-14. *Guard\_Out* property example

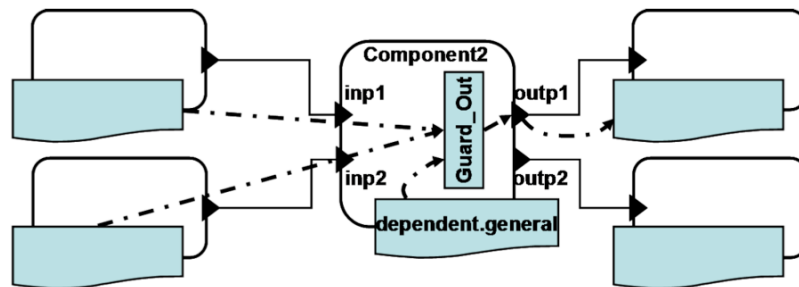


Figure II-15. Architectural view of a *Guard\_Out* property

### II.1.2.3.3 Comparison between *Guard\_In* and *Guard\_Out*

Table II-1 shows the similarities and differences of *Guard\_In* and *Guard\_Out* properties both from a functional point of view and from an input/output point of view.

Table II-1. *Symmetry and asymmetry between Guard\_In and Guard\_Out*

|                          | <b>Guard_In</b>  | <b>Guard_Out</b>  |
|--------------------------|--|---|
| <i>Functional View</i>   | Applies to <b>incoming</b> features.   | Applies to <b>outgoing</b> features.  |
|                          | Its evaluation result has an <b>impact on the internal behavior</b> of the component that declares it.               | Its evaluation result has an <b>impact on components that depend on</b> the component that declares it.                             |
| <i>Input/Output View</i> | <b>Input:</b> propagations from the component's environment.<br><b>Output:</b> propagations to the component itself. | <b>Input:</b> propagations from the component's environment.<br><b>Output:</b> propagations to the <b>component's environment</b> . |

One might consider using a **Guard\_In** if the decision-making layer (filtering) is placed at the input interface (i.e., on the incoming features) of a component. Instead, a **Guard\_Out** may be appropriate if the decision-making functionality exists as a stand-alone component.

#### II.1.2.3.4 Interacting **Guard\_In** and **Guard\_Out** properties

An **out** propagation, which occurs as a result of a transition in an error model or as a result of a pass-through rule of a **Guard\_Out** property, may be named in (1) pass-through rules of **Guard\_Out** properties or in (2) mapping rules of **Guard\_In** properties. This leads to two kinds of interactions between **Guard** properties:

- 1) from a **Guard\_Out** (whose result is an **out** propagation) to a **Guard\_In** (whose mapping rules refer to that **out** propagation).
- 2) from a **Guard\_Out** (whose result is an **out** propagation) to a **Guard\_Out** (whose pass-through rules refer to that **out** propagation).

These two kinds of interactions are illustrated successively hereafter.

##### II.1.2.3.4.1 Cascading **Guard\_Out** - **Guard\_In**

Figure II-16-a presents an architectural model example in which a **Guard\_In** property applying to **in** ports of *Component3* refers to an **out** propagation that occurs as a result of a **Guard\_Out** property applying to the **out** port of *Component1*. The **in** propagations occurring as a result of the **Guard\_In** property trigger AADL transitions in *Component3*. The **Guard\_Out** property associated with the **out** port of *Component1* is given in Figure II-16-b. The **Guard\_In** property associated with the **in** ports of *Component3* is given in Figure II-16-c.

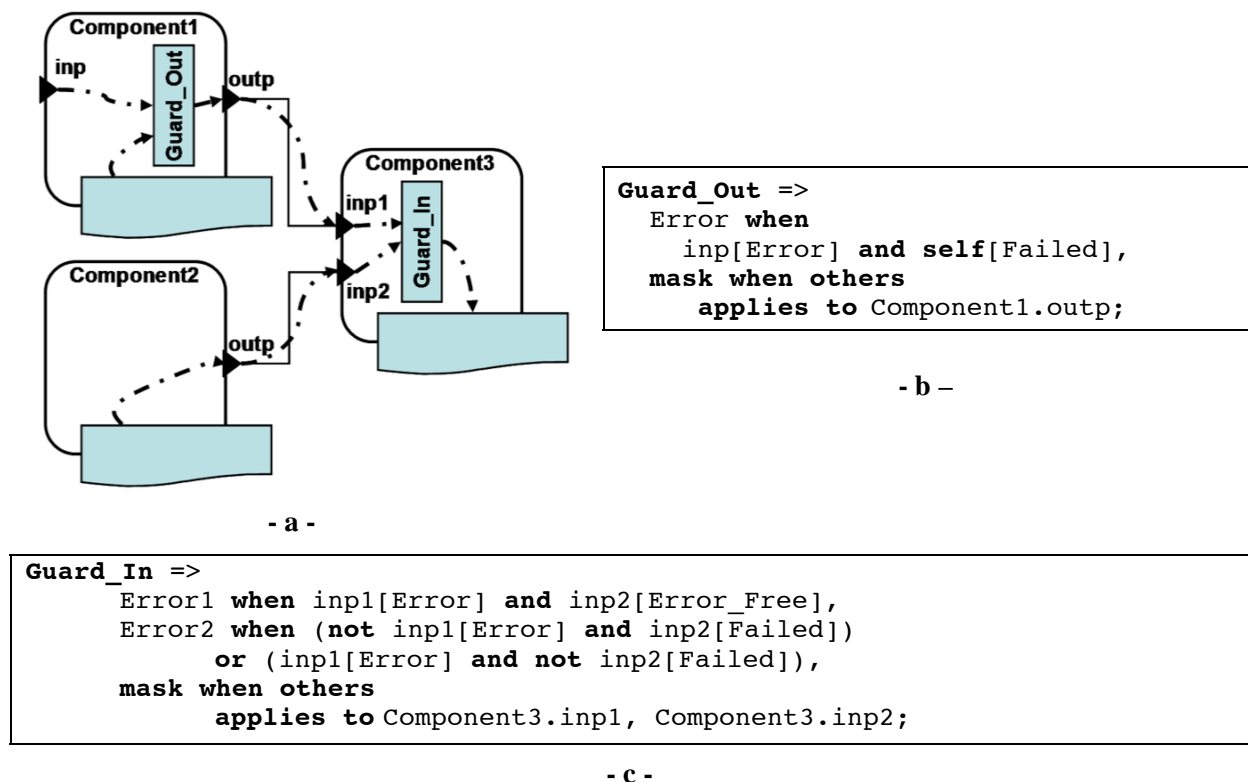
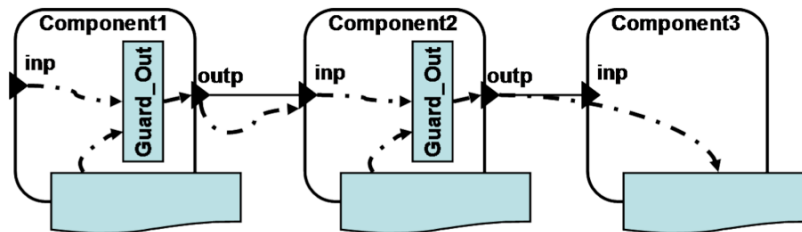


Figure II-16. Cascading **Guard\_Out** - **Guard\_In** properties

### II.1.2.3.4.2 Cascading *Guard\_Out* - *Guard\_Out*

Figure II-17-a presents an architectural model example in which a **Guard\_Out** property applying to an **out** port of *Component2* refers to an **out** propagation that occurs as a result of a **Guard\_Out** property applying to an **out** port of *Component1*. The error model associated with *Component3* is impacted by the propagations resulting from the **Guard\_Out** property of *Component2*. The **Guard\_Out** property associated with the **out** port of *Component1* is given in Figure II-17-b while the one associated with the **out** port of *Component2* is given in Figure II-17-c.



- a -

```
Guard_Out =>
  Error when
    inp[Error] and self[Failed],
  mask when others
  applies to Component1.outp;
```

- b -

```
Guard_Out =>
  Error1 when
    inp[Error] and self[Failed],
  Error2 when
    (not inp1[Error] and self [Failed])
  or (inp1[Error] and not self[Failed]),
  mask when others
  applies to Component2.outp;
```

- c -

Figure II-17. Cascading *Guard\_Out* - *Guard\_Out* properties

## II.1.2.4 Mechanisms for connecting error states to modes

AADL allows modeling error states and operational modes of a component separately. It also allows establishing connections between the error states and the operational modes. There are three mechanisms to specify such connections:

- 1) **Guard\_Event** properties model the generation of architectural events depending on the behavior described in the error model instances.
- 2) **Guard\_Transition** properties constrain mode transitions depending on logic expressions referring to occurrences of several architectural events. Used together with **Guard\_Event** properties, **Guard\_Transition** properties provide advanced decision mechanisms for modeling fault-tolerance reconfiguration strategies.
- 3) **activate/deactivate** transitions that allow the description of different component behaviors in the presence of faults depending on whether the component is active or inactive in a particular mode.

These mechanisms are presented successively in the three following subsections.

### II.1.2.4.1 *Guard\_Event*

**Guard\_Event** properties translate error states and propagations into actions (under the form of architectural events) on the running system. They map error state and propagation configurations into architectural events that are associated to:

- 1) **out** event ports. An **out** event port named in a mode transition of the enclosing component causes the occurrence of that mode transition when the event occurs (i.e., when the Boolean condition of the **Guard\_Event** property is true).
- 2) local event ports. A local event port causes a mode transition in the component that owns it, if it is named in that mode transition.

If an event connection has its source in the **out** port or local port, then the event is routed through that connection and may affect the behavior of a recipient component by triggering a mode transition. An architectural event passing through a port named in a mode transition unconditionally triggers that mode transition. The Boolean condition of a **Guard\_Event** property is similar to Boolean expressions of **Guard\_In** and **Guard\_Out** properties. It names outgoing propagations as well as states of the components that impact the given component. In addition, it may refer to states of the component that declares the **Guard\_Event**. Thus, the generated event can reflect an error state and propagation configuration.

The syntax of the **Guard\_Event** property in Backus-Naur form is given in Figure II-18.

```
Guard_Event ::= boolean_expr applies to EventPort;
EventPort ::= outEventPort | localEventPort
```

Figure II-18. *Guard\_Event* property syntax

Figure II-19 shows an example of **Guard\_Event** property associated with an **out** event port of a system component.

```
system Component3
  features
    inp1: in data port;
    inp2: in data port;
    outp: out event port;
  end Component3;

system implementation Component3.generic
  [...]
  annex Error_Model {**
    Model => MyLibrary::dependent.general;
ge1    Guard_Event =>
ge2      (not inp1[Error] and inp2[Failed])
ge3      or  (inp1[Error] and not inp2[Failed]),
ge4      applies to outp;
  **};
end Component3.generic;
```

Figure II-19. *Guard\_Event* property example

The **Guard\_Event** property (lines ge1-ge4) of Figure II-19 specifies that the component *Component3.generic* generates an architectural event, which is released through the **out** event port *outp*, when one of the following two conditions holds:

- 1) an **out** propagation *Error* is visible through its **in** port *inp1*.
- 2) the component connected to *inp2* is *Failed*.

Any other error propagation and state configuration does not result in an architectural event generation.

Figure II-20 shows an architectural view of the application of the **Guard\_Event** property presented above. Events sent out through the **out** event port *outp* of *Component3* are routed through a connection to the **in** event port *inp* of a *Component4* and trigger a mode transition from *m1* to *m2* in *Component4*.

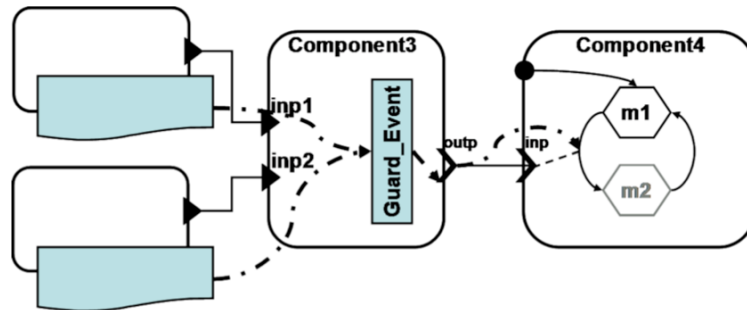


Figure II-20. Architectural view of a **Guard\_Event** property

#### II.1.2.4.2 Guard\_Transition

AADL mode transition declarations name one or more event ports whose events trigger the mode transition. By default, an event through any of the named ports triggers the mode transition. When modeling fault-tolerance, it is desirable to constrain a mode transition in a system component to reflect specific conditions such as a voting protocol to decide on fault handling. This can be achieved through the use of **Guard\_Transition** properties associated with mode transitions and specifying mode transition logic expressions overriding the default **or** condition on events arriving through ports named in the mode transition.

In the AADL standard v1.0, mode transitions do not have names. Thus, the standardized AADL Error Model Annex specifies that a **Guard\_Transition** property is associated with the event ports named in a mode transition. This raises a problem in the case where several mode transitions name the same set of event ports and assuming that each one of the mode transitions must be triggered by a different Boolean condition involving those event ports. An erratum to the AADL standard allows naming mode transitions. We use this facility to identify the mode transition to which applies a **Guard\_Transition** property.

The **Guard\_Transition** property supports:

- 1) The specification of mode transition conditions other than the default **or** on architectural events occurring through ports named in mode transitions.
- 2) The specification of conditions for error propagations and states to trigger mode transitions without explicitly mapping them into an architectural event via a **Guard\_Event** property.

In [Feiler & Rugina 2007], we strongly recommend that **Guard\_Transition** properties be only used for specifying architectural event-based conditions for mode transitions to avoid inconsistent or conflicting **Guard\_Event** and **Guard\_Transition** specifications. Therefore, we focus on the first usage of **Guard\_Transition** properties in the remainder of the dissertation. The syntax of the **Guard\_Transition** property in Backus-Naur form is given in Figure II-21.

```
Guard_Transition ::= boolean_expr applies to modeTransitionName;
```

Figure II-21. *Guard\_Transition property syntax*

Figure II-22 shows an example of **Guard\_Transition** property associated with a mode transition, named M1toM2, of a system component (see line gt). The **Guard\_Transition** property specifies that the mode transition from *m1* to *m2* occurs when events arrive both through ports *inp1* and *inp2*. Without this property association, the mode transition would occur at the occurrence of an event through any of the two ports.

```
system Component4
  features
    inp1: in event port;
    inp2: in event port;
  end Component4;

system implementation Component4.generic
  modes
    m1: initial mode;
    m2: mode;
    M1toM2: m1-[inp1, inp2]->m2;
    M2toM1: m2-[inp1, inp2]->m1;
  annex Error_Model {**
(gt)    Guard_Transition => inp1 and inp2 applies to M1toM2;
  **};
end Component4.generic;
```

Figure II-22. *Guard\_Transition property example*

Figure II-23 shows an architectural view of the application of the **Guard\_Transition** property presented above. Events occurring according to **Guard\_Event** properties associated with the **out** event ports *outp* of *Component31* and *Component32* are routed through connections to the **in** event ports *inp1* and *inp2* of a *Component4* that trigger a mode transition from *m1* to *m2* in *Component4*.

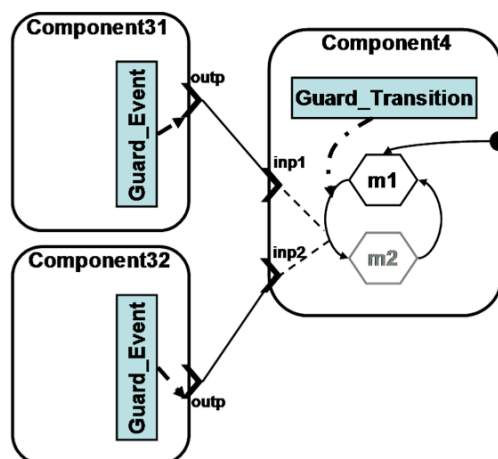


Figure II-23. *Architectural view of Guard\_Transition property*



### II.1.2.4.3 Activate / Deactivate transitions

AADL allows modeling the fact that a component is active in particular modes and inactive in other modes. This is of particular interest when a component behaves differently in the presence of faults when it is active or inactive. To describe such behaviors, the AADL Error Model Annex allows the user to declare (optionally) an **initial inactive** state, in addition to the initial state, in an error model type. This initial inactive state is the initial state of the component if the component is inactive in the initial mode of the system.

To express that the error model characteristics change when the system configuration changes, i.e., when the component is activated or deactivated, one may declare transitions labeled **activate** or **deactivate** in an error model implementation, in addition to transitions triggered by (error) events and propagations. A transition labeled **activate** occurs when the component is activated at a mode switch while a transition labeled **deactivate** occurs when the component is deactivated at a mode switch. If a component is activated or deactivated at a mode switch but no transitions from the current state are labeled respectively **activate** and **deactivate**, its state does not change.

Figure II-24 shows an example of error model declaring an initial inactive state and **activate** / **deactivate** transitions (see lines *d* and *a*).

| <b>Error Model Type [modal]</b>  |            |
|--|------------|
| <pre> <b>error model</b> modal <b>features</b>   ON_Error_Free: <b>initial error state</b>;   OFF_Error_Free: <b>initial inactive error state</b>;   ON_Failed: <b>error state</b>;   ON_Fail: <b>error event</b> {Occurrence =&gt; poisson λ1};   ON_Restart: <b>error event</b> {Occurrence =&gt; poisson μ}; <b>end</b> modal; </pre> |            |
| <b>Error Model Implementation [modal.general]</b>  |            |
| <pre> <b>error model implementation</b> modal.general <b>transitions</b> (d)   ON_Error_Free-[<b>deactivate</b>]-&gt;OFF_Error_Free; (a)   OFF_Error_Free-[<b>activate</b>]-&gt;ON_Error_Free;       ON_Error_Free-[ON_Fail]-&gt;ON_Failed;       ON_Failed-[ON_Restart]-&gt;ON_Error_Free; <b>end</b> modal.general; </pre>             | (d)<br>(a) |

**Figure II-24. Activate/deactivate transitions**

The initial inactive state is an error-free state, similarly to the initial (active) state. We make the assumption that an inactive component does not fail. A component that is error-free when activated at a mode transition moves to an active error-free state and may fail while the component is active. An active failed component may be restarted to regain its active error-free state.

### II.1.2.5 Error model abstractions

An AADL architectural model is hierarchical when components contain subcomponents. The level of detail of the architectural model depends on the stage of the design and development process. For example, a system may initially be modeled as a partial model to

the level of subsystems and later completed to the level of threads. Both partial and complete models can be instantiated to produce system instance models for system analysis.

The user chooses the level of the component hierarchy to which error models are associated. For example, an error model can be associated with the root-level system component to represent an abstracted error model of the system instance. An instance of this error model represents the system instance error model as a finite state stochastic automaton.

Similarly, error models can be associated with each of the leaf components in the system hierarchy, i.e., individual application threads and individual hardware components. In this case, the system instance error model consists of the set of component error model instances and connection error model instances, if declared. The system instance error model represents a set of concurrent finite state stochastic automata.

Error models can be associated with several levels of the system hierarchy at the same time. For example, an error model may be associated with an application thread, with an enclosing application (sub)system, and with the system as a whole. In this case the error model higher in the system hierarchy is an abstraction of the contained error models. The AADL Error Model Annex offers two approaches for representing error model abstractions:

- 1) an *abstract error model* represents the behavior of a component in the presence of faults in terms of states and events inherent to the component, propagations from and to components this component interacts with, and transitions that are triggered by intrinsic events or incoming propagations and initiate outgoing propagations. The behavior of a component in the presence of faults is defined without referring to any subcomponent.
- 2) a *derived error model* represents the behavior of a component in the presence of faults in terms of global states as a *logic expression* (**Derived\_State\_Mapping**) of the states of its subcomponents. The definition of the **Derived\_State\_Mapping** expression in Backus-Naur form is given in Figure II-25 without developing the Boolean expression. The Boolean expression refers to states of subcomponents and connections that are part of the component declaring the derived error model.

```

Derived_State_Mapping ::= stateMapping_rule {, stateMapping_rule}*
stateMapping_rule ::= globalState_id when boolean_expr

```

Figure II-25. *Derived\_State\_Mapping* expression definition

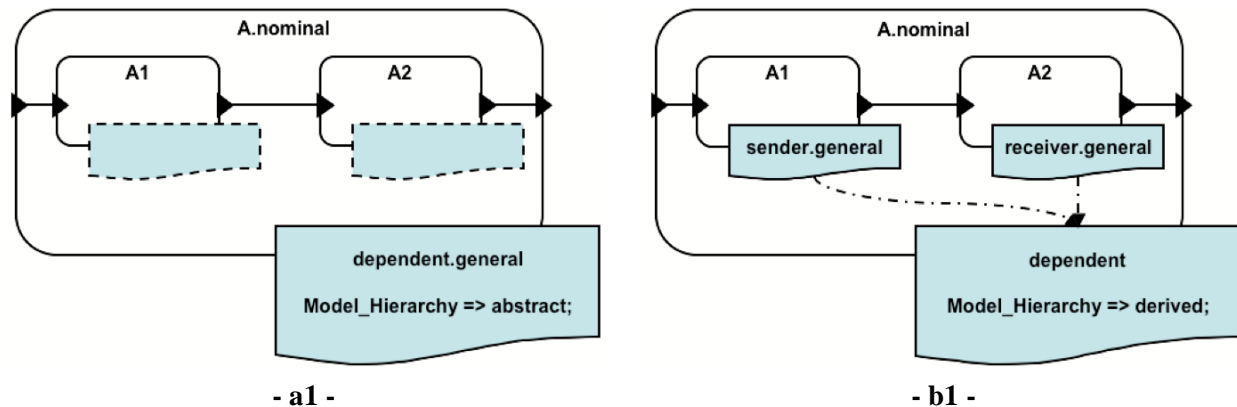
In Figure II-26, we use the same AADL architectural model to illustrate the two error model abstractions. The system *A* contains two components, *A1* and *A2*, which have error models.

In Figure II-26-a1, *A* has an *abstract error model*, indicated by the value **abstract** for the **Model\_Hierarchy** property. Figure II-26-a2 describes this system in textual AADL. The behavior of the system in the presence of faults is considered to be the one described in the error model implementation dependent.general. The error models of subcomponents are ignored (thus we did not name them in Figure II-26-a1).

In Figure II-26-b1, *A* has a *derived error model*, indicated by the value **derived** for the **Model\_Hierarchy** property. When the **Model\_Hierarchy** property is defined as **derived**, it is necessary to declare a **Derived\_State\_Mapping** expression to specify the global states of the component or system (declared in the type indicated by the **Model** property) as a logic expression of the states of its subcomponents. Figure II-26-b2 describes the system in textual AADL. The **Derived\_State\_Mapping** expression specifies that *A* is

*Error\_Free* if both *A1* and *A2* are *Error\_Free*, and *Failed* otherwise. *Error\_Free* and *Failed* must be states declared in the error model type *dependent*.

To evaluate dependability measures, the user must specify state classes for the overall system. For example, if the user wishes to evaluate reliability or availability, it is necessary to specify the system states that are to be considered as failed states. If in addition, the user wishes to evaluate safety, it is necessary to specify the failed system states that are considered as catastrophic. In AADL, such state classes are declared by means of a *derived error model* for the overall system.



- a1 -

- b1 -

```

system implementation A.nominal
subcomponents
  A1: system sw.nominal;
  A2: system sw.nominal;
annex Error_Model {**
  Model => dependent.general;
  Model_Hierarchy => abstract;
  **};
end A.nominal;

```

- a2 -

```

system implementation A.nominal
subcomponents
  A1: system sw.nominal;
  A2: system sw.nominal;
annex Error_Model {**
  Model => dependent;
  Model_Hierarchy => derived;
  Derived_State_Mapping =>
    Error_Free when (A1[Error_Free]
      and A2[Error_Free]),
    Failed when others;
  **};
end A.nominal;

```

- b2 -

Figure II-26. Architectural view of error model abstractions

## II.2 Petri nets

This section presents the characteristics and terminology of classical place/transitions Petri nets (PN) before showing particularities of Generalized Stochastic Petri nets (GSPNs), which are an extension of PNs aimed at supporting performance and dependability analyses.

### II.2.1 Place/Transitions Petri nets

PNs are directed graphs with two types of nodes: places drawn as circles and transitions drawn as boxes [Peterson 1981]. Places model partial states (conditions that may hold at given times). Transitions correspond to actions or events that may induce a state change.

## CHAPTER II

Places can contain tokens drawn as dots. The state corresponding to a PN is called a marking and represents the number of tokens in each place. The initial marking represents the initial state.

The arcs of the graph connect places with transitions. There are two types of arcs: *normal* (arrow-headed) and *inhibitor* (circle-headed). A place  $P_i$  is considered an *input place* of a transition  $t$  if there is an arc from  $P_i$  to  $t$ . A place  $P_i$  is considered an *output place* of a transition  $t$  if there is an arc from  $t$  to  $P_i$ . An arc is annotated with a positive number called *weight* (or *multiplicity*). By default the weight of an arc is 1. A transition is enabled when each of its input places has a number of tokens greater or equal to the weight of the corresponding input arc if that arc is normal and each of its input places is empty of tokens if the corresponding input arc is inhibitor. Several transitions can be enabled in a marking. Any enabled transition can fire leading to a new marking. The initial marking together with the structure of the PN define its state space. The *reachability set* contains all markings reachable from the initial marking. The *reachability graph* contains nodes corresponding to all markings contained in the *reachability set*. Nodes are connected through arcs corresponding to the firing of transitions leading from one marking to another one.

Two transitions are considered to be in *conflict* if they are both enabled and the firing of one of them disables the other one.

A PN is characterized by a set of properties. The most important are as follows.

- 1) *Bounded* = The reachability set is bounded. In other words, all its places are *k-bounded*. A place is said to be *k-bounded* when the number of tokens in that place never exceeds a constant  $k$ .
- 2) *Live* = All its transitions are *live*. A transition is said to be *live* if a path can be found in the reachability graph starting from any marking  $M_i$  such that a marking  $M_j$  is reached in which that transition is enabled.
- 3) *Reversible* = From any marking, a path can be found so that the initial marking  $M_0$  is reached.

Figure II-27 shows two examples of PN with their respective reachability graphs. They have the same initial marking ( $P_1, P_4$ ), which represents the initial state.

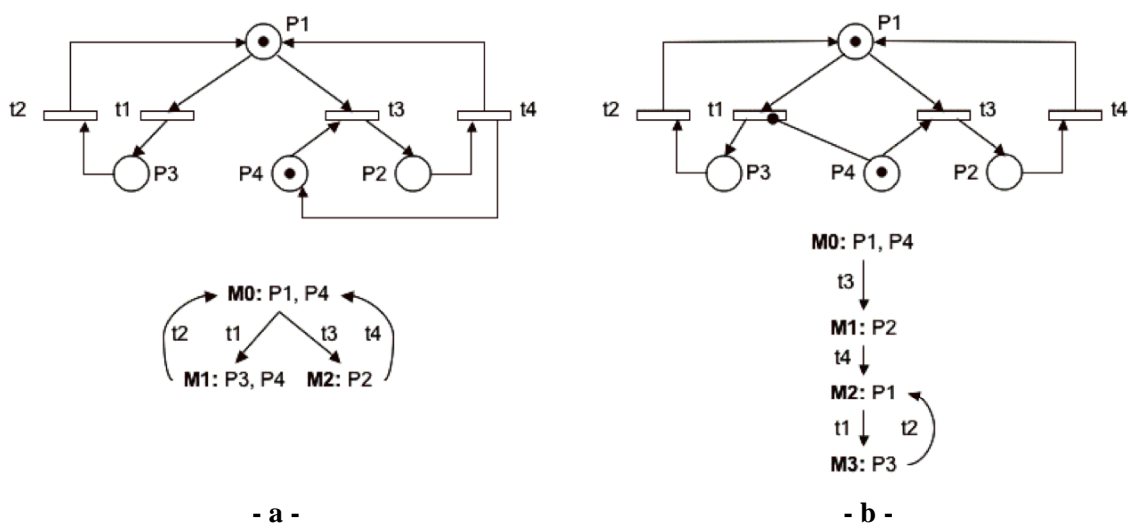


Figure II-27. Examples of PN

In the PN of Figure II-27-a, transitions  $t1$  and  $t3$  are in conflict (if  $t1$  fires, then  $t3$  is disabled). This PN is bounded (all places are 1-bounded), live (from each marking, there is a path allowing to fire any transition) and reversible (the initial marking can be reached from the other markings).

In the PN of Figure II-27-b, the only transition enabled in the initial marking is  $t3$ . This PN is bounded (all places are 1-bounded). It is not live (for example from marking M1, there is no path leading to a marking allowing to fire  $t3$ ) and it is not reversible (the initial marking cannot be reached from the other markings).

Formally, a PN is defined as a six-tuple:

$PN = \{P, T, I, O, H, M_0\}$  where

- $P$  is the set of places,
- $T$  is the set of transitions,  $T \cap P = \emptyset$ ,
- $I, O, H: T \rightarrow \text{Bag}(P)$ , are the input, output and inhibitor functions,
- $M_0$  is the initial marking: a function that associates with a place a natural number corresponding to the number of tokens initially in that place.

Given a transition  $t \in T$ , we denote<sup>26</sup> with:

$\vec{t}$  = the set of input places connected by unidirectional input arcs to a transition  $t$ ,

$\leftarrow t$  = the set of output places connected by unidirectional output arcs from a transition  $t$ ,

$^?t$  = the set of input places connected by inhibitor arcs to a transition  $t$ .

$A_t = \vec{t} \cup \leftarrow t \cup ^?t$  (the set of arcs connected to  $t$ )

## II.2.2 Generalized Stochastic Petri nets

The definition of GSPNs is based on that of the classical PNs in which some transitions are *timed* while others are *immediate* [Ajmone Marsan *et al.* 1995]. Random, exponentially distributed firing delays are associated with timed transitions while the firing of immediate transitions takes no time. Immediate transitions have priority over timed transitions, i.e., if an immediate and a timed transition are both enabled in a marking, the immediate transition is fired first. The selection of the next transition to fire among several enabled immediate transitions is made through firing probabilities. The selection of the next transition to fire among several enabled timed transitions is made according to a race policy, i.e., the transition that fires is the one with the minimum firing delay.

In a GSPN, the sojourn time for markings that enable timed transitions only is non-null and exponentially distributed. Conversely, the sojourn time for markings that enable immediate transitions is null. Markings of the former type are considered as *tangible* while markings of the latter type are considered to be *vanishing*.

GSPNs do not preserve the reachability graph of the underlying PN because of the priority of immediate transitions over timed ones and of the mixing of exponentially distributed and null firing delays. Figure II-28 shows two examples of GSPN. Their underlying PNs are identical. However, their reachability graphs are different based on which transitions are timed and which transitions are immediate. In the example of Figure II-28-a, only  $t1$  is

---

<sup>26</sup> The notation used here is slightly different of the one used in [Peterson 1981].

timed<sup>27</sup>.  $t1$  and  $t3$  are both enabled in  $M_0$  but only  $t3$  can be fired as  $t3$  is immediate. Thus, this GSPN is not live ( $t1$  and  $t2$  never fire): it contains an infinite loop over a set of vanishing markings. All reachable markings are vanishing. In the example of Figure II-28-b,  $t2$  and  $t4$  are timed. From  $M_0$ ,  $t1$  fires with probability  $p$  and  $t3$  fires with probability  $1-p$ .  $M_0$  is vanishing.

For the analysis of GSPNs, a reduced reachability set is of interest. It is formed of all tangible markings and is isomorphic to a Markov chain. The reduced reachability set for the example of Figure II-28-a is empty while the reduced reachability set for the example of Figure II-28-b contains  $M_1$  and  $M_2$ .

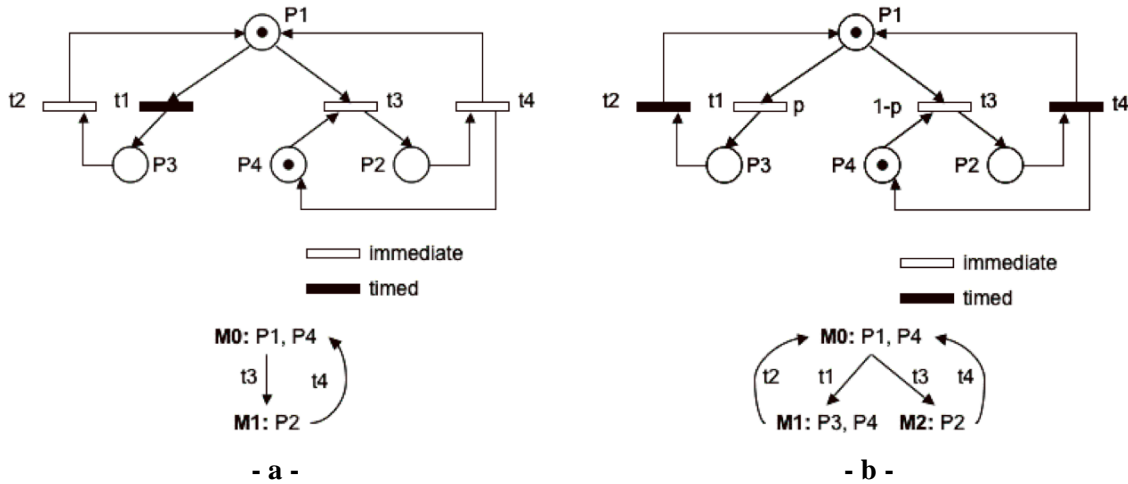


Figure II-28. Examples of GSPN

In order to evaluate performance or dependability measures from a GSPN, the GSPN must be bounded and free of infinite loops over sets of vanishing markings.

Formally, a GSPN is defined as a eight-tuple [Ajmone Marsan *et al.* 1995]:

GSPN =  $\{P, T, I, O, H, M_0, \pi, \Lambda\}$  where

- The first six elements are the same as for PNs (see section II.2.1)
- $\pi: T \rightarrow \{0, 1\}$ , is a function that maps an immediate transition into the number 1 and a timed one into the number 0, meaning that immediate transitions have a priority 1 (higher) than timed transitions (priority 0),
- $\Lambda: T \rightarrow \mathfrak{R}^+$ , defines the stochastic properties of transitions (the rate for timed transitions and the probability for immediate ones).

### II.3 Conclusion

In this chapter, we introduced the AADL support for dependability analyses and we gave a brief overview of GSPNs, compared to classical PNs.

<sup>27</sup> The representation for immediate and timed transitions that we use here is different from the one used in [Ajmone Marsan *et al.* 1995]. An immediate transition is represented by an empty box while a timed one is represented by a full box. Intuitively, it takes no time to pass through an empty box and it takes time to pass through a full box.

The AADL language support for dependability analyses is formed of two interacting languages: the core AADL and the AADL Error Model Annex. The core AADL is aimed at being the backbone of a system's development process. It allows the description of component and connection-based architectures (formed of software and hardware) and architecture configurations through the use of operational modes. The AADL architectural model acts as a support for analysis-specific information, which is modeled through the use of annexes and properties. The Error Model Annex represents language support for modeling dependability-related information through the use of error models (which are separate of the AADL architectural model, favoring reusability) and error model annex subclauses for associating the dependability-related information to the architectural model.

We presented the main concepts related to GSPNs, as well as the properties they must have to be analyzable. We illustrated them on examples that are very close in terms of structure, but that do not share properties. In particular, it is noteworthy that GSPNs do not necessarily preserve the properties of their underlying PNs.

To obtain a dependability-oriented GSPN reflecting the behavior represented in the AADL architectural model enriched with error model annex constructs, we must transform the error model annex constructs into GSPN subnets. The connections between the subnets are dictated by the architectural model that hosts the error model annex constructs. Our modeling approach, and especially the AADL to GSPN transformation, is based on the language support presented in this chapter. However, the AADL language is still evolving. We present in Appendix A our evolution proposals for the Error Model Annex, that have been submitted to the SAE AADL standardization committee.

The terminology related to GSPNs and their formalization presented in this Chapter is further used in Chapter IV, to describe the AADL to GSPN transformation rules.

---

## III AADL Dependability Modeling: Guidelines and Patterns

---

Our iterative AADL-based dependability modeling approach introduced in Chapter I, Figure I-1, is based on identifying the components and connections of interest for the dependability analysis, followed by the identification of dependencies between them. First, the components' behaviors in the presence of faults are modeled as if they were independent. Then, dependencies are added progressively. In this Chapter, we focus on the AADL dependability model construction. We first give general AADL modeling guidelines for independent components and for the different types of dependencies. Then, we show that the development of patterns is very useful to facilitate the modeling of fault-tolerant systems and to enhance the reusability of the models. Reusability contributes to reducing the cost of building the AADL dependability model. For example, to build models for alternative architectural solutions, several variants of one dependency may be added to the same intermediary AADL dependability model that already includes a set of common dependencies.

Due to the separation of concerns between the architectural model and the dependability-related information, model reusability is achieved at several levels:

- 1) **The AADL architectural models of components are reusable:** a component declaration may be instantiated in several system models. A component model may represent a complex configuration of components and connections. This allows the definition of *architectural patterns* that may be the basis for building architectural models for candidate architectures. An architectural pattern may be used unmodified or may be customized by using the inheritance and refinement mechanisms provided by AADL. In particular, these mechanisms allow adding features, or making some of the subcomponent declarations modal, i.e., describing the fact that some of the subcomponents are active in particular operational modes and inactive in others.
- 2) **The AADL error models are reusable:** instances of an error model may be associated with several components and connections. It is possible to customize error model instances by defining component-specific Occurrence properties for events and **out** propagations declared in the error model instantiated by the **Model** property.
- 3) **The AADL dependability models (AADL architectural model + AADL error models) are reusable:** a component having an associated error model annex subclause may be instantiated in several system instance models, allowing the definition of *dependability-oriented patterns* that reduce the effort necessary for building AADL dependability models of candidate architectures for a given system specification. Dependability-oriented patterns take advantage of the two levels of reusability exposed above. Several dependability-oriented patterns may be based on the same architectural pattern (only the dependability-related information differs between them). Also, they may be based on customized architectural patterns, i.e., on AADL dependability models that extend the patterns through the AADL inheritance and refinement mechanisms.



Since our ultimate goal is to obtain dependability measures from AADL models, this Chapter focuses on the third level of reusability, related to *dependability-oriented patterns*. For conciseness, we will refer to them simply as *patterns*. Patterns are useful if they represent common behaviors that do not need substantial customization. Otherwise, their reusability is questionable.

This chapter is organized as follows. Section III.1 gives recommendations for modeling independent components. Section III.2 is dedicated to modeling dependencies. Section III.3 presents reusable and customizable patterns for fault-tolerance policies and Section III.4 concludes this Chapter.

### III.1 Modeling independent components

The behavior of each component of an architectural model that is considered for the dependability analysis is first described as resulting from its own faults and repair events. To this end, the user associates with each component a generic error model representing a state machine. An error model representing an isolated component is formed of states and events. Events are considered inherent to the component. The only effect the occurrence of an event may have is a state change in the component itself. In our modeling framework, we require that the source and the destination of an AADL transition triggered by an event be different states. In this way, an event always has a physical meaning.

A transition triggered from an origin state by an immediate event  $i$ , whose Occurrence property is  $p_i$ , occurs with a probability  $p_i / \sum_{j=1}^n p_j$ , where  $n$  is the number of events that trigger transitions out of the same origin state.

It is noteworthy that qualitative dependability analyses do not require the definition of Occurrence properties for events. As we focus on quantitative evaluation of dependability based on AADL models, we do require the definition of Occurrence properties for all events. When a generic error model is associated with a component, it may be customized by specifying component-specific Occurrence properties for its events. Component-specific Occurrence properties override the default values defined in the generic error model.

Generally, if the targeted dependability analysis requires the derivation of a state space model (such as a GSPN or a Markov chain), it is recommended to avoid the declaration of immediate events, as they may lead to state space explosion that may cause problems in processing the model. From a practical point of view, only stable states, i.e., states that are sources of transitions triggered by timed events, have a physical meaning. Immediate propagations will be used for the synchronization of effects on other components, as explained in subsection III.2.1.

### III.2 Modeling dependencies

The dependencies between different components of the AADL dependability model are described using named **out** and **in** propagations to model respectively which propagations coming from other components are known within the component by the specified name and which parts of the behavior of the component are made visible to the interacting components (see Section II.1.2.2). By default, the mapping of an **out** propagation occurring in a

component to an **in** propagation of another component is determined by name matching. The user may override this rule by explicitly specifying such mappings as **Guard\_In** and **Guard\_Out** properties.

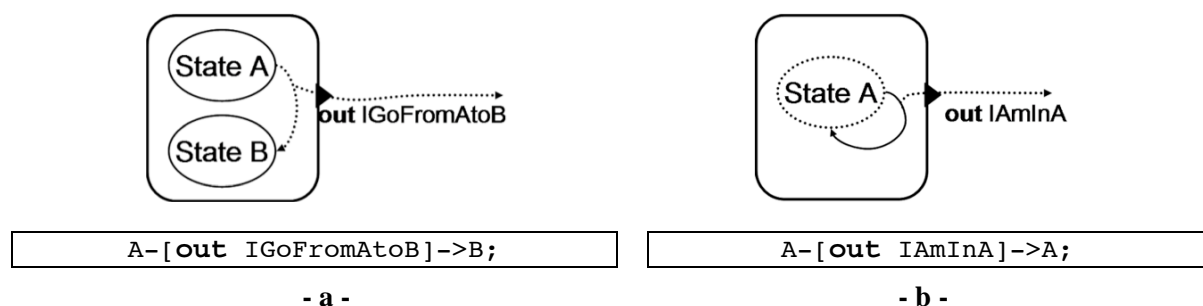
Subsection III.2.1 gives guidelines on the use of **out** and **in** propagations, which are the main way to describe a dependency. The subsequent subsections (III.2.2, III.2.3 and III.2.4) guide the description of structural and functional dependencies, those related to maintenance and those related to fault-tolerance.

### III.2.1 On the use of **out** and **in** propagations

The occurrence of an **out** propagation may or may not trigger a state change in the component that declares it. From a semantic point of view, if it triggers a state change, then the transition from the origin state to the destination state is made visible to other components. Alternatively, the propagation makes this state visible to other components (i.e., when the propagation occurs, other components are notified of the fact that this component is in this state). In other words, an **out** propagation triggering a state change is either consumed immediately or it does not have any effect on its recipient component and connections. An **out** propagation that does not trigger a state change is visible while the component is in that state. These two cases are shown in Figure III-1. In Figure III-1-a, the **out** propagation *IGoFromAtoB* triggers an AADL transition from state A to state B. This transition, represented by a dotted line, is made visible to other components. In Figure III-1-b, the **out** propagation *IAmInA* does not trigger a state change, i.e., the AADL transition it triggers has the same state as origin and destination. The fact that the component is in state A, represented by a dotted line, is made visible to other components. In the first case, the transition is always made visible from outside. In the second case, the state may be made visible with a given probability or with a delay characterizing the Occurrence of the **out** propagation.

We suggest using this second case whenever possible, as it favors model readability: the propagation assumption is separate from the internal event, whereas, in the first case, the propagation assumption is built in. Removing the dependency requires the transformation of the propagation declaration into an event declaration in the first case.

We require that the source and the destination of an AADL transition triggered by an **in** propagation be different states. This prevents infinite loops over a set of vanishing<sup>28</sup> states.



**Figure III-1:** Transition and state visible from outside

<sup>28</sup> A state is said to be *vanishing* if all the AADL transitions having it as a source are triggered by events or propagations characterized by Occurrence properties which are fixed probabilities.

Due to the fact that vanishing states are not physical states, we warn the user about the following modeling configurations and suggest their avoidance:

- An **in** propagation triggering a transition out of a vanishing state never occurs.
- A timed **out** propagation triggering a transition out from a vanishing state never occurs.

Also, if from a stable state  $S$ , the only exit is through an AADL transition triggered by an immediate **out** propagation whose probability  $p$  is different of 1, the component remains blocked in that state  $S$  with probability  $1-p$ .

### III.2.2 Modeling structural and functional dependencies

Structural and functional dependencies are supported by the architectural model and must be modeled in the error models associated with dependent components, usually by specifying respectively outgoing and incoming propagations and their impact on the corresponding error models. An example of structural dependency is shown in Figure III-2. Figure III-2-a presents the AADL architectural model: a thread is bound to a processor. Figure III-2-b shows the corresponding dependency block diagram (the behavior of the thread depends on the behavior of the processor). Figure III-2-c presents the AADL dependability model where error models are associated both with the thread and with the processor to describe the dependency.

The error model associated with the processor takes into account the sender-side dependency from the processor to the thread. The error model associated with the thread takes into account the recipient-side dependency. When the processor is in the erroneous state, it sends a propagation through the binding of the thread to the processor. As a consequence, the incoming propagation *Error* causes the failure of the thread. The *Error* propagation is visible while the processor remains in the erroneous state, since the AADL transition triggered by the **out** propagation *Error* in the processor has the same state as origin and destination.

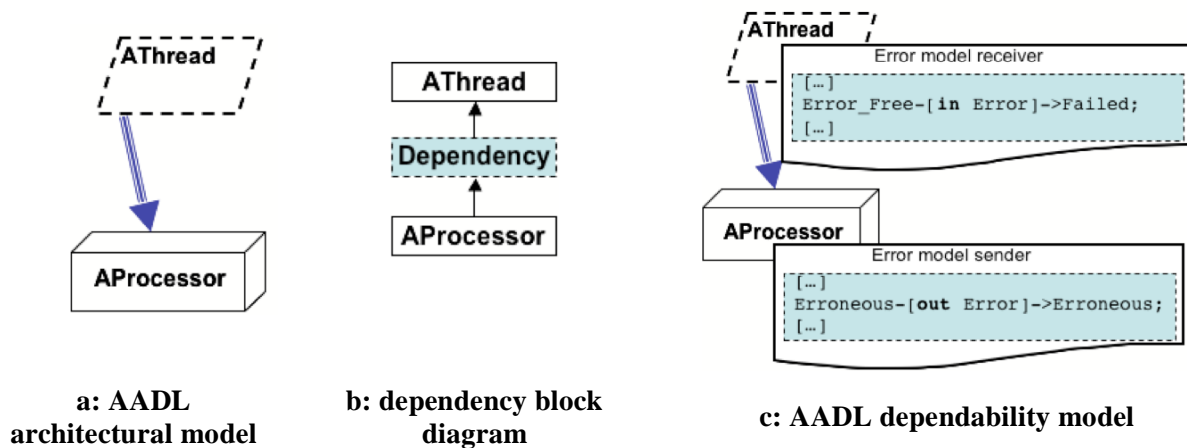
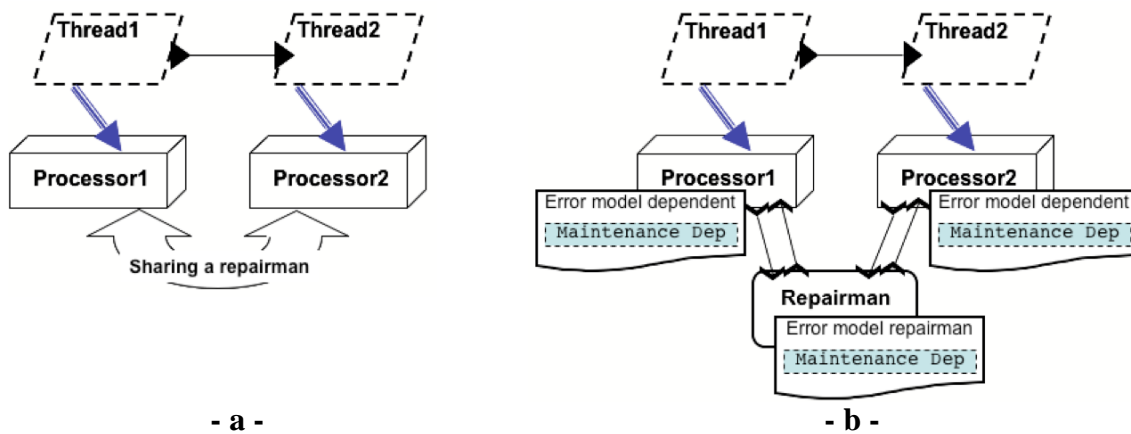


Figure III-2. Structural dependency example

### III.2.3 Modeling maintenance dependencies

Maintenance dependencies need to be described when repair and restoration facilities are shared between components or when the maintenance activity of some components has to be carried out according to a given order or a specified strategy.

Components that are not dependent at the architectural level may become dependent due to the fact that they share maintenance facilities or to the synchronization of the maintenance activities. Thus, the architectural model might need some adjustments to support the description of dependencies related to the maintenance policy. As error models interact only via propagations through architectural features (i.e., connections, bindings), the maintenance dependency between components' error models must also be supported by the architectural model. This means that besides the system architecture components, we may need to add components representing the shared repair facilities (or repairmen) to model the maintenance dependencies. These additional components do not have any role at the design level (except for being an indication that maintenance is foreseen for the system), but they support dependability modeling. Figure III-3-a shows an architectural model example where *Processor1* and *Processor2* do not interact (there is no architectural dependency between them and they only interact with the threads bound to them). However, if we assume that they share one repairman, it is necessary to represent the repairman at the level of the architectural model, as shown in Figure III-3-b, which represents the AADL dependability model taking into account the maintenance dependency between the two processors.



**Figure III-3.** Maintenance dependency example

Also, the error models of dependent components might need some adjustments. For example, to represent the fact that *Processor1* must be repaired only if *Processor2* is not failed, it is necessary to decompose the failed state of *Processor1* to distinguish between a state where *Processor1* waits to be repaired and a state where it is being repaired. In the next two subsections, we give two concrete examples of maintenance dependencies: the first one models a shared maintenance facility. The second one extends the first one to consider the priority of one of the components to repair.

### III.2.3.1 Shared maintenance facility

It is assumed that the maintenance facility (or repairman) is a shared resource between two components. Thus, if both components fail one after the other, the second failed component must wait until the maintenance facility has been released by the first failed component. It is straightforward to generalize the pattern for several components.

We consider the system of Figure III-3-b, in which *Processor1* and *Processor2* share a *repairman*. Instances of the same error model, *dependent.general*, are associated with *Processor1* and *Processor2*. The *repairman* also has an error model. The maintenance and recovery policy is described in the error models. Figure III-4 shows the error model

*dependent.general*. We need to distinguish between the failed state where the component does not use the repairman and the state where it uses it. To this end, the error model declares the following two states: *Failed*, *InRepair* (see lines f1, f2). We also need to declare a state meaning that the component has been repaired (see line f3), so that the *repairman* knows when it has been released<sup>29</sup>. When the component fails, it notifies the *repairman* (see lines o1, o1'). It expects an authorization to be repaired (see lines i1, i1'). When it has been repaired, it releases the *repairman* (see lines o2, o2').

| <b>Error Model Type [dependent]</b>  |  |
|--|--|
| <pre> <b>error model</b> dependent <b>features</b>   Error_Free: <b>initial error state</b>;   Erroneous: <b>error state</b>; (f1)  Failed: <b>error state</b>; (f2)  InRepair: <b>error state</b>; (f3)  Repaired: <b>error state</b>;       Temp_Fault: <b>error event</b> {Occurrence =&gt; poisson <math>\lambda</math>1};       Perm_Fault: <b>error event</b> {Occurrence =&gt; poisson <math>\lambda</math>2};       Restart: <b>error event</b> {Occurrence =&gt; poisson <math>\mu</math>1};       Recover: <b>error event</b> {Occurrence =&gt; poisson <math>\mu</math>2};       Error: <b>out error propagation</b> {Occurrence =&gt; fixed p}; (o1)  FailedVisible: <b>out error propagation</b> {Occurrence =&gt; fixed q}; (o2)  IRepaired: <b>out error propagation</b> {Occurrence =&gt; fixed 1}; (i1)  IRepair: <b>in error propagation</b>; <b>end</b> dependent; </pre> |  |
| <b>Error Model Implementation [dependent.general]</b>  |  |
| <pre> <b>error model implementation</b> dependent.general <b>transitions</b>   Error_Free-[Perm_Fault]-&gt;Failed;   Error_Free-[Temp_Fault]-&gt;Erroneous;   Erroneous-[Recover]-&gt;Error_Free;   Erroneous-[<b>out</b> Error]-&gt;Erroneous; (o1')  Failed-[<b>out</b> FailedVisible]-&gt;Failed; (i1')  Failed-[<b>in</b> IRepair]-&gt;InRepair;       InRepair-[Restart]-&gt;Repaired; (o2')  Repaired-[<b>out</b> IRepaired]-&gt;Error_Free; <b>end</b> dependent.general; </pre>  |  |

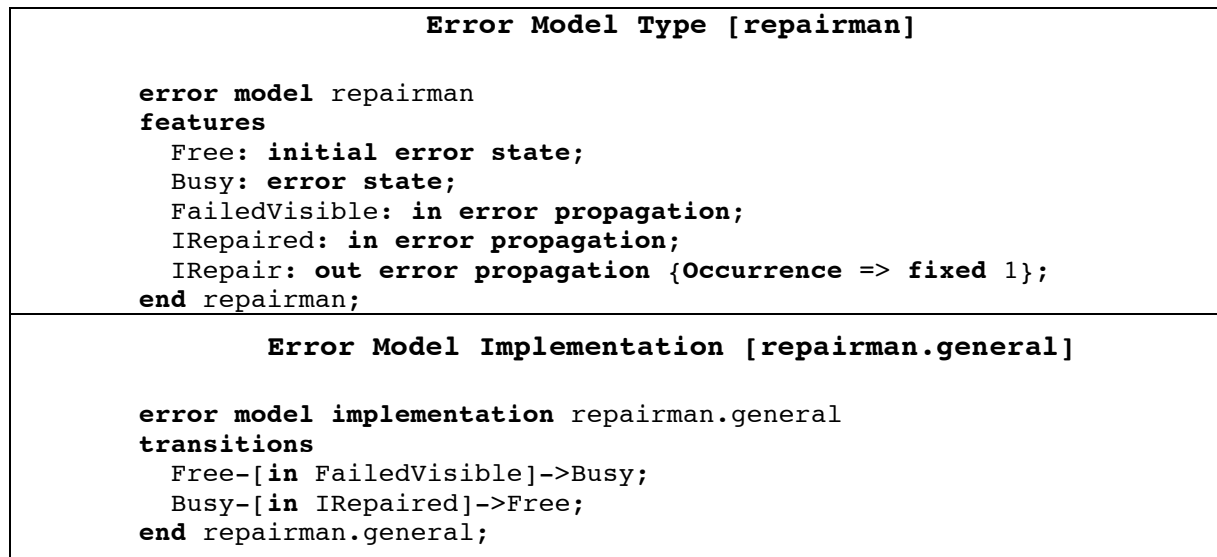
**Figure III-4.** Error model for component with maintenance dependency

Figure III-5 shows the error model associated with the *repairman*. The error model *repairman.general* declares two states: *Free* (initial state) and *Busy*. The transitions between them are triggered by *in* propagations (coming from *Processor1* or *Processor2*). Assuming that the *repairman* is *Free*, if it receives a *FailedVisible* propagation from one of the components, it goes to state *Busy* and sends a *IRepair* propagation to the component that sent the *FailedVisible* propagation, to allow it to go to state *InRepair*. At the end of the recovery procedure, the component sends an *IRepaired* propagation allowing the *repairman* to go back to state *Free*.

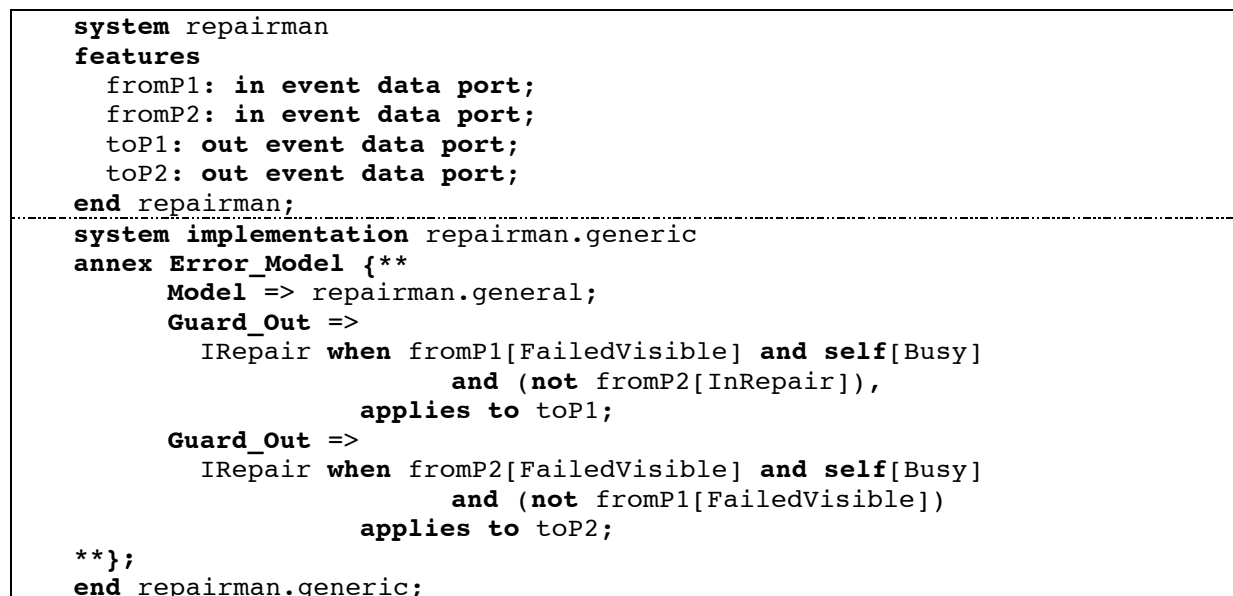
The *IRepair* propagation must be sent only to the component that sent the *FailedVisible* propagation. Otherwise, if both components fail one after the other, both receive the *IRepair*

<sup>29</sup> We cannot use the *Error\_Free* state to this end, as the repairman could be released if the other processor is *Error\_Free* (the processor receives propagations from both processors and cannot distinguish between propagations having the same name and coming from different components).

propagation and are repaired simultaneously. The **Guard\_Out** properties of Figure III-6 are used in order to send the *IRepair* propagation only to the concerned component. They are associated with the ports *toP1* and *toP2*, which are the origins for connections respectively to *Processor1* and *Processor2*.



**Figure III-5.** Error model for shared maintenance facility



**Figure III-6.** Textual AADL dependability model – shared maintenance facility

### III.2.3.2 Priority to a component's maintenance

As in the previous subsection, it is assumed that the maintenance facility (or repairman) is a shared resource between two components. In addition, it is assumed that one of the components has a priority to repair. The AADL dependability model of Figure III-3-b applies to this policy too. An adjustment is necessary in the error model *dependent.general*, to bring the component back from state *InRepair* to state *Failed*, if the other component failed in the

meanwhile and has a priority to repair. Figure III-7 shows what needs to be added (line i2 in the error model type and line i2' in the error model implementation). *StopRepair* must also be declared as an *out* propagation in the error model associated with the repairman.

|       |   |
|-------|---|
| (i2)  | <code>StopRepair: in error propagation;</code>    |
| (i2') | <code>InRepair-[in StopRepair]-&gt;Failed;</code> |

**Figure III-7.** *Refinement of dependent.general*

Let us assume that *Processor1* has a priority to repair. We model the behavior related to the priority to repair by using two different *Guard\_Out* properties associated with the ports *toP1* and *toP2*, as shown in Figure III-8.

The *Guard\_Out* property associated with *toP1* specifies that the repairman sends the *IRepair* propagation through *toP1* as soon as it perceives *FailedVisible* at the port *fromP1* (it does not matter if *Processor2* failed, as *Processor1* has a priority). The *Guard\_Out* property associated with *toP2* specifies that the repairman sends the *IRepair* propagation through *toP2* as soon as it perceives *FailedVisible* at the port *fromP2* only if *FailedVisible* is not perceived at the port *fromP1* too. Also, a *StopRepair* propagation is sent through *toP2*, when *Processor1* fails while *Processor2* is being repaired, to make *Processor2* return to the *Failed* state.

```

system implementation repairman.generic
annex Error_Model {**
  Model => repairman.general;
  Guard_Out =>
    IRepair when fromP1[FailedVisible] and self[Busy],
      applies to toP1;
  Guard_Out =>
    IRepair when fromP2[FailedVisible]
      and (not fromP1[FailedVisible]) and self[Busy],
    StopRepair when fromP1[FailedVisible]
      and (not fromP2[InRepair]) and self[Busy]
      applies to toP2;
**};
end repairman.generic;

```

**Figure III-8.** *Textual AADL dependability model – maintenance with priority*

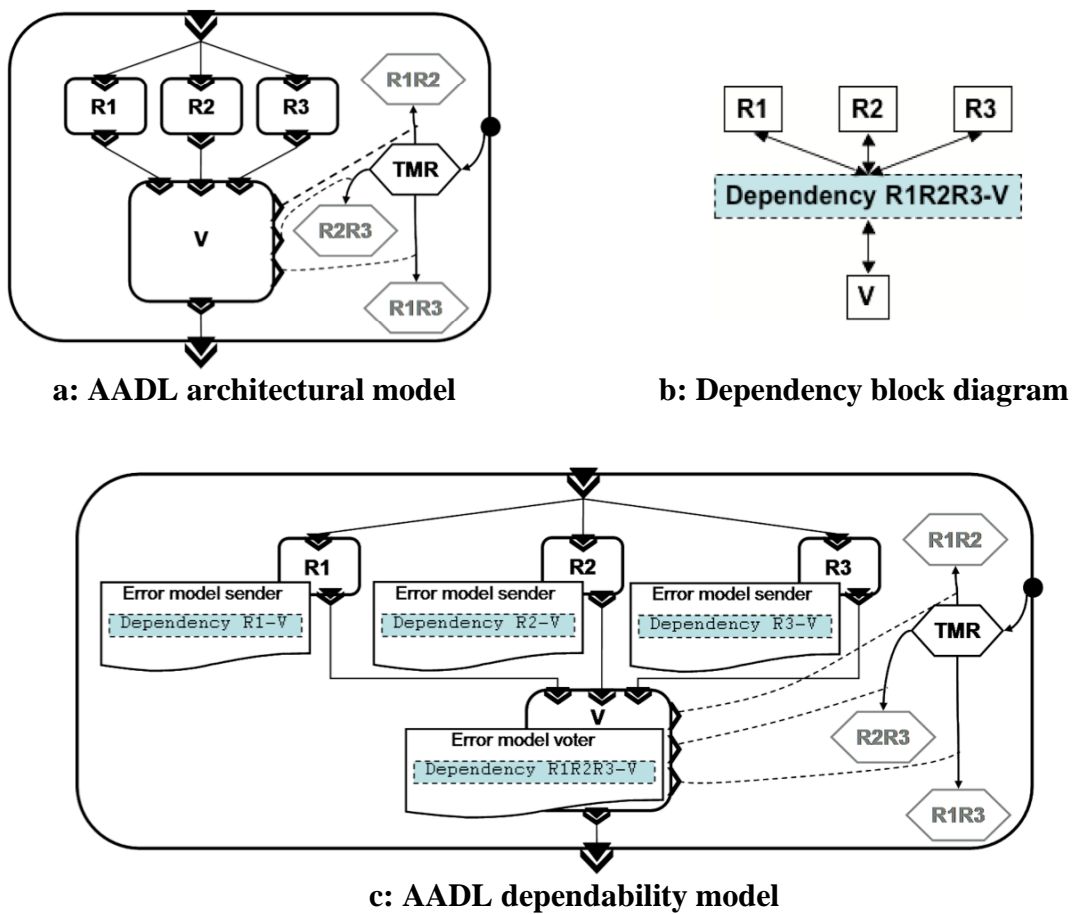
### III.2.4 Modeling fault-tolerance dependencies

Modeling fault-tolerance policies requires the description of architectural reconfigurations. Such configurations are modeled by AADL modes. Mode-specific fault tolerant system configurations reflect the fault-tolerance policy chosen for the system or for particular parts of the system. For example, a fault tolerant system formed of three components may have a nominal operational mode corresponding to a triple-redundancy configuration and degraded operational modes corresponding to dual-redundant configurations.

The architectural model may need adjustments to reflect the structure and mode dynamics required by the considered fault-tolerance policy. In fact, the architectural model depends very much on the fault-tolerance policy, since the fault-tolerance policy determines the number of replicas, the decision-making components (e.g., a voter), the connections and fault tolerant modes. Once the architectural model reflects the structure of the fault tolerant system, the dependency related to the configuration changes due to the fault-tolerance policy must be modeled at the error model level. An example is shown in Figure III-9. Figure III-9-a presents

the AADL architectural model that already reflects the considered fault-tolerance policy: triple redundancy with degraded operational modes corresponding to dual-redundant configurations. Figure III-9-b shows the corresponding dependency block diagram, in which the behavior of the voter  $V$  depends on the data input received from the three replicas:  $R1$ ,  $R2$  and  $R3$ . The behavior of the replicas depends on the decision of the voter: for example, if it decides that  $R1$  is faulty, then it deactivates  $R1$ . In other words, the triple modular redundant system moves from mode  $TMR$  to mode  $R2R3$ . Figure III-9-c presents the AADL dependability model where error model instances are associated with each replica and with the voter, to describe the fault-tolerance dependency between the replicas and the voter.

Both modes and error states represent states of a system. The difference between them lies primarily in their semantics. Error states result from occurrences of error events (faults, repair events) while modes represent operational states of the system that may be totally independent of the occurrence of error events.



**Figure III-9.** *Fault-tolerance dependency example*

Error states may influence the mode dynamics through the use of **Guard\_Event** and **Guard\_Transition** properties. Modes may influence the error state dynamics through the use of **activate** / **deactivate** transitions. These mechanisms have been presented in Section II.1.2.4. Concrete examples of their use for modeling fault tolerant dependencies are given in Section III.3 (dedicated to fault-tolerance patterns).



Error states referred to in **Guard\_Event** properties must be stable<sup>30</sup>, to ensure that the generated events always occur if the corresponding Boolean expression evaluates to **TRUE**.

For a component having distinct behaviors when it is active and inactive, source states of **activate** and **deactivate** transitions must be stable. Otherwise, they may never occur, as the component would move in zero time to a stable state.

Modes are also used for modeling modes of operation in phased-mission systems. *Operational modes in phased-mission systems* model configurations representative of different phases in a mission. For example, in the case of an aircraft model, one may distinguish between the takeoff, cruise and landing phases. During each of the three phases, the system would have a particular configuration with active components and connections. Also, the behavior in the presence of faults would be different during each phase. Different types of faults may affect the system in different phases.

Usually, phased-mission systems also need modes to represent fault-tolerance mechanisms. In AADL, this nesting of modes is captured by phased-mission modes in a component, which is a subcomponent of a system component whose modes represent alternative configuration of its redundant subcomponents.

### III.3 Fault-tolerance patterns

The two following subsections present successively patterns for classical hardware and software fault-tolerance policies that tolerate a single fault. Preliminary versions of the patterns have been presented in [Rugina *et al.* 2006a]. For hardware, we considered N-modular redundancy (exemplified on a triple modular redundancy scheme), and several duplex schemes: cold, warm, hot standby and active dynamic redundancy with self-checking components. For software, we considered N-version programming (exemplified on 3-version programming), the recovery block and N self-checking programming (both exemplified on duplex architectures). We adopt the terminology of [Laprie *et al.* 1990].

All patterns presented in the rest of this Section instantiate error models presented previously in sections II.1.2.2 (*sender.general* - for dependent components) and II.1.2.4.3 (*modal.general* - for **activate** / **deactivate** transitions). We add to these error models an **out** propagation *FailedVisible* that makes visible the failure, as shown in Figure III-10.

```
(d1'')    FailedVisible: out error propagation {Occurrence => fixed q};
(d2'')    Failed-[out FailedVisible]->Failed;
```

**Figure III-10.** Failure propagation in error model *sender.general*

#### III.3.1.1 Hardware fault-tolerance

Each of the subsequent subsections first presents the specifications of a fault-tolerance policy and then it shows the corresponding AADL pattern.

<sup>30</sup> A *stable* state is a non-vanishing state, i.e., there is at least one AADL transition having it as a source and triggered by a timed event or propagation.

## III.3.1.1.1 N-modular redundancy

N-modular redundancy assumes that the fault tolerant system is formed of N active replicas and a decider which acts as a *voter*. The outputs of all replicas are compared by the voter, which decides which output to be delivered. The result given by the majority is delivered as result of the system. If more than (N-2) replica failed, then the system is considered to have failed. The voter may also fail. If so, the system is considered to have failed.

The N-modular redundancy pattern is modeled in AADL in Figure III-11 by considering three replicas (i.e., Triple Modular Redundancy). The system A is formed of three replicas (*R1*, *R2* and *R3*) and a *decider*. The error model *sender.general* is associated with *R1*, *R2*, *R3* and the *decider*. The three replicas are connected through event data connections to the *decider*. Based on the data received and on its own state, the decider/voter decides which result is to be delivered: if at least two replicas outputs are erroneous, an error is propagated.

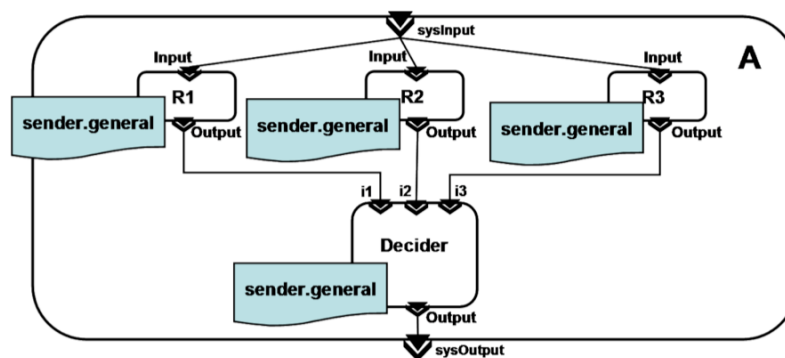


Figure III-11. N-modular redundancy pattern

The AADL dependability models of the replicas are identical: an instance of the error model *sender.general* is associated with each replica. Note that it is possible to customize this generic error model to the components by setting component-specific Occurrence properties for events and propagations. Figure III-12 shows only the AADL dependability model of the decider. A **Guard\_Out** property is associated with the *decider*'s *out* port *Output*. It specifies that the system propagates out *FailedVisible* when the decider failed or when at least two replicas failures have been detected by the decider.

```

system decider
features
  i1: in event data port;
  i2: in event data port;
  i3: in event data port;
  Output: out event data port;
end decider;
-----
system implementation decider.generic
annex Error_Model {**
  Model => sender.general;
  Guard_Out =>
    FailedVisible when two ormore (i1[FailedVisible],
      i2[FailedVisible], i3[FailedVisible]) or self[Failed]
    applies to Output;
  **};
end decider.generic;

```

Figure III-12. Textual AADL dependability model – N-modular redundancy

## III.3.1.1.2 Cold standby sparing

The cold standby sparing fault tolerant system is formed of two *replicas*. One replica is active while the other one is powered off. A *decider* (representing a physical switch) monitors the active replica. If it detects its failure, it powers off the active replica and powers on the spare, which continues to provide the service. If the decider detects the failure of the second replica, it powers it off and the system is considered as *Failed*. It is assumed that an inactive replica does not fail. However, the pattern can be extended to include such failures.

The cold standby sparing pattern is modeled in AADL in Figure III-13. For this pattern, we show the textual AADL dependability model of the system in Figure III-14.

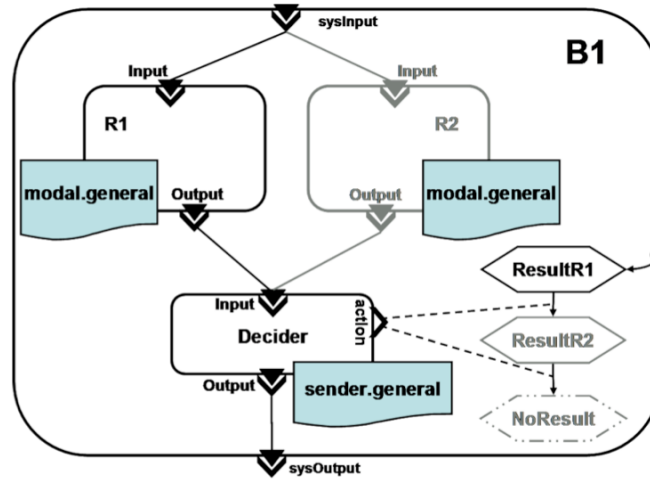


Figure III-13. Cold standby sparing pattern

According to the specification, the system *B1* is formed of two replicas (*R1* and *R2*) and a *decider* (see Figure III-13). The behavior of the replicas is described by the error model *modal.general* (they do not fail when they are inactive). Each replica, together with the connections to and from it, is active in one operational mode. In the mode *NoResult*, none of the replicas is active. The behavior of the *decider* is described by the error model *sender.general*. The *decider's* *out* event port *action* triggers the mode transitions so that to activate / deactivate *R1* and *R2*.

The replicas have the same component type and implementation with an associated instance of the error model *modal.general*, which is not shown in Figure III-14. The upper part of Figure III-14 shows the component type and implementation of the *decider* while the lower part corresponds to the whole system type and implementation.

A **Guard\_Event** property is associated with the *decider's* *out* event port *action*, to specify the mode switching conditions. The system switches from *ResultR1* to *ResultR2* when the *decider* detects the failure of *R1* (*R1* is active and a *FailedVisible* propagation is perceived at the *decider's* *Input* port). It switches from *ResultR2* to *NoResult* when the *decider* detects the failure of *R2* (*R2* is active and a *FailedVisible* propagation is detected at the *decider's* *Input* port). If the *decider* fails, it does not order a mode switch.

```

system decider
features
  Input: in event data port;
  Output: out event data port;
  action: out event port;
end decider;
-----
system implementation decider.generic
annex Error_Model {**
  Model => sender.general;
  Guard_Event => Input[FailedVisible] and self[Error_Free]
  applies to action;
**};
end decider.generic;
-----
system ColdStandBy
features
  sysInput: in event data port;
  sysOutput: out event data port;
end ColdStandBy;
-----
system implementation ColdStandBy.generic
subcomponents
(c1) R1: system replica.generic in modes ResultR1;
(c2) R2: system replica.generic in modes ResultR2;
  decider: system decider.generic;
connections
  event data port sysInput->R1.Input in modes ResultR1;
  event data port sysInput->R2.Input in modes ResultR2;
  event data port R1.Output->decider.Input in modes ResultR1;
  event data port R2.Output->decider.Input in modes ResultR2;
  event data port decider.Output->sysOutput;
modes
  ResultR1: initial mode;
  ResultR2, NoResult: mode;
  ResultR1-[decider.action]->ResultR2;
  ResultR2-[decider.action]->NoResult;
end ColdStandBy.generic;

```

Figure III-14. Textual AADL dependability model – cold standby sparing

### III.3.1.1.3 Warm standby sparing

Similarly to the cold standby sparing, the warm standby sparing fault tolerant system is formed of two *replicas*. Unlike the cold standby sparing policy, the warm standby sparing policy assumes that both replicas are always powered on and only a single replica's output is active. Based on the input from the replica whose output is active, a *decider* attempts to detect failures, so as to activate the output of the other replica.

The warm standby sparing pattern is modeled in AADL in Figure III-15. Compared to the AADL model of the cold standby sparing of Figure III-13, the only difference is that both *R1* and *R2* are active in modes *ResultR1* and *ResultR2* of *B2*. Only the connections are modal.

To obtain the textual AADL model of the warm standby sparing, based on the model of the cold standby sparing, we replace lines *c1*, *c2* of Figure III-14 with lines *w1*, *w2* shown in Figure III-16. Note that associating the error model *sender.general* instead of *modal.general* with the replicas does not change anything in the system behavior.

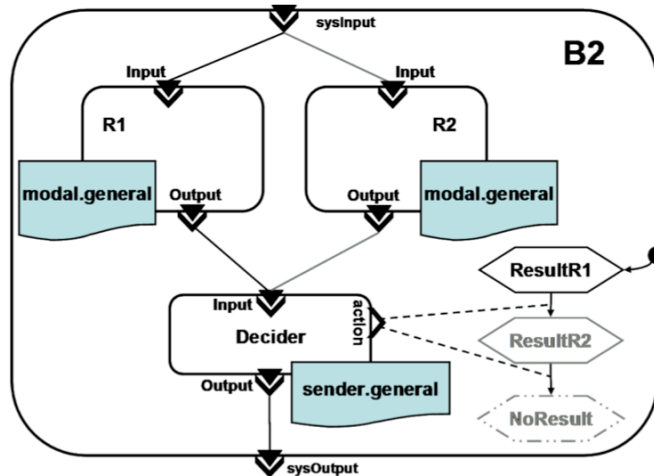


Figure III-15. Warm standby sparing pattern

|      |   |
|------|---|
|      | <b>System implementation</b> WarmStandBy.generic                      |
|      | <b>subcomponents</b>  |
| (w1) | R1: <b>system</b> replica.generic <b>in modes</b> ResultR1, ResultR2; |
| (w2) | R2: <b>system</b> replica.generic <b>in modes</b> ResultR1, ResultR2; |
|      | decider: <b>system</b> decider.generic;                               |
|      | [...]   |
|      | <b>end</b> WarmStandBy.generic;                                       |

Figure III-16. Textual AADL dependability model – warm standby sparing

#### III.3.1.1.4 Hot standby sparing

The hot standby sparing fault tolerant system is formed of two active replicas: a *primary* and a *backup*. A *decider* monitors the two replicas. If it detects the failure of the primary, it orders the backup to take over so as to continue to provide the service. If both replicas fail one after the other, the first one recovered becomes the primary. In this Section, we present successively two ways of modeling this specification in AADL.

The AADL dependability model of Figure III-17 represents a first model variant of the hot standby sparing pattern. The system *C1* is formed of two replicas (*R1* and *R2*) and a *decider*. The error model *sender.general* is associated with *R1*, *R2*, and the *decider*. Each replica can be in one of two modes: *primary* and *backup*. When a component is in *primary* mode, it provides the service expected from the redundant system. If the *decider* detects the failure of the replica in *primary* mode, it initiates a mode switch in each component, so that the one that is *Error\_Free* continues to provide the service. In the case of failure of both components, it waits until one of them becomes operational and orders it to go to *primary* mode.

Figure III-18 shows the textual AADL dependability model of the component initially in *primary* mode (upper part of the figure) and of the decider (lower part of the figure).

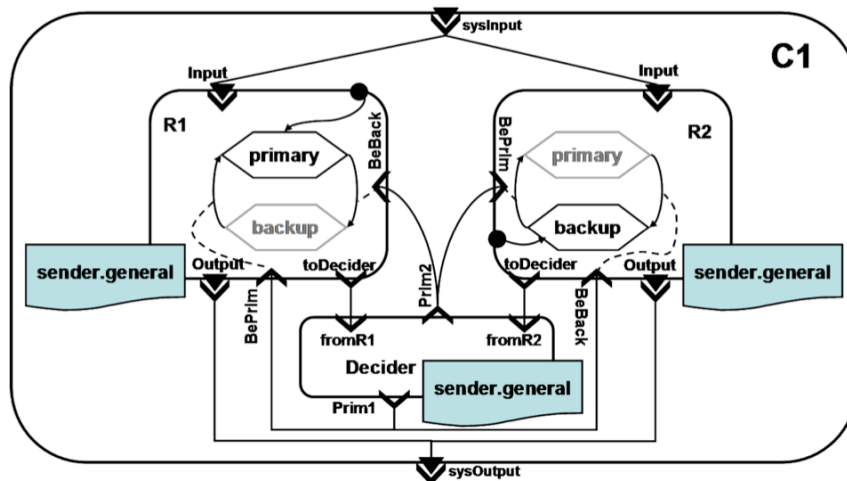


Figure III-17. AADL architectural model of the hot standby sparing pattern (variant 1)

```

system replica
features
  Input: in event data port;
  Output: out event data port;
  BePrim, BeBack: in event port;
  toDecider: out event port;
end replica;
-----
system implementation replica.primary
modes
  primary: initial mode;
  backup: mode;
  primary-[BeBack]->backup;
  backup-[BePrim]->primary;
annex Error_Model {**
  Model => sender.general;
**};
end replica.primary;
-----
system decider
features
  fromR1, fromR2: in event port;
  Prim1, Prim2: out event port;
end decider;
-----
system implementation decider.generic
annex Error_Model {**
  Model => sender.general;
  Guard_Event => fromR1[FailedVisible] and fromR2[Error_Free]
    and self[Error_Free] applies to Prim2;
  Guard_Event => fromR2[FailedVisible] and fromR1[Error_Free] and
    self[Error_Free] applies to Prim1;
**};
end decider.generic;

```

Figure III-18. Textual AADL dependability model – hot standby sparing (variant 1)

As shown in Figure III-18, **Guard\_Event** properties are associated with the **out** ports *Prim1* and *Prim2* of the *decider* to specify that a mode switch from *primary* to *backup* is to be performed in one of the components when the *decider* detects the failure of the other component. At the same time, a mode switch from *backup* to *primary* is to be performed in the other component, i.e., one event triggers two mode transitions, one in each replica. If the *decider* fails, it stops sending switch orders until it is restarted.

The AADL dependability model of Figure III-19 represents a second model variant of the hot standby sparing pattern. As the system  $C1$ , the system  $C1'$  is formed of two replicas ( $R1$  and  $R2$ ) and a *decider*. The error model *sender.general* is associated all components.

The difference between  $C1$  and  $C1'$  is that in  $C1$  each replica has its own operational modes while in  $C1'$  the system itself can be in one of two modes:  $R1\_primary$  and  $R2\_primary$ . The *decider* initiates a mode switch from  $R1\_primary$  to  $R2\_primary$  if it detects the failure of  $R1$  while  $R2$  is *Error\_Free*. If both replicas fail, no mode change is ordered. Further on, in section III.3.1.3, we give guidance on choosing one of the two alternative patterns.

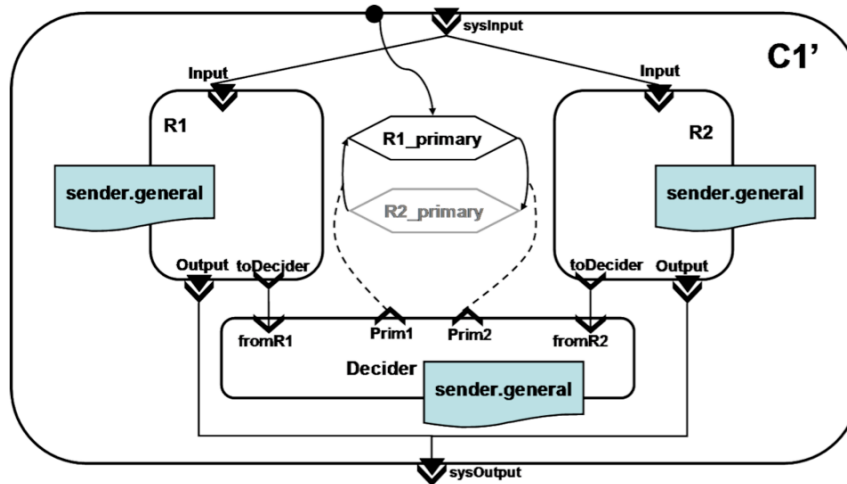


Figure III-19. AADL architectural model of the hot standby sparing pattern (variant 2)

Figure III-20 shows the complete textual AADL dependability model for the pattern  $C1'$ . The replicas have the same component type and implementation, given in the upper part of Figure III-20. The middle part shows the component type and implementation of the decider while the lower part corresponds to the whole system type and implementation. The **Guard\_Event** properties associated with the **out** event ports *Prim1* and *Prim2* of the decider are the same as those of the first variant of the pattern.

```

system replica
features
  Input: in event data port;
  Output: out event data port;
  toDecider: out event port;
end replica;
-----
system implementation replica.primary
annex Error_Model {**
  Model => sender.general;
**};
end replica.primary;
-----
system decider
features
  fromR1, fromR2: in event port;
  Prim1, Prim2: out event port;
end decider;
-----
system implementation decider.generic
annex Error_Model {**
  Model => sender.general;
  Guard_Event => fromR1[FailedVisible] and fromR2[Error_Free]
    and self[Error_Free] applies to Prim2;
  Guard_Event => fromR2[FailedVisible] and fromR1[Error_Free] and
    self[Error_Free] applies to Prim1;
**};
end decider.generic;
-----
system HotStandBy_v2
features
  sysInput: in event data port;
  sysOutput: out event data port;
end HotStandBy_v2;
-----
system implementation HotStandBy_v2.generic
subcomponents
  R1: system replica.generic;
  R2: system replica.generic;
  decider: system decider.generic;
connections
  event data port sysInput->R1.Input;
  event data port sysInput->R2.Input;
  event data port R1.Output->sysOutput;
  event data port R2.Output->sysOutput;
  event data port R1.toDecider->decider.fromR1;
  event data port R2.toDecider->decider.fromR2;
modes
  R1_primary: initial mode;
  R2_primary: mode;
  R1_primary-[decider.Prim2]->R2_primary;
  R2_primary-[decider.Prim1]->R1_primary;
end HotStandBy_v2.generic;

```

**Figure III-20.** Textual AADL dependability model – hot standby sparing (variant 2)

### III.3.1.1.5 Active dynamic redundancy

We consider active dynamic redundancy with two self-checking replicas. One replica plays the *primary* role while the other one plays the *backup* role. It is assumed that each replica is able to detect its own failure and to notify the other replica about its failure. If the primary fails, it notifies the backup, which takes over. If both replicas fail one after the other, the first one recovered becomes the primary. In this Section, we present successively two ways of modeling this pattern in AADL.



The AADL dependability model of Figure III-21 represents a first model variant of the pattern. The system *C2* is formed of two replicas (*R1* and *R2*). The error model *sender.general* is associated with *R1* and *R2*. Each replica can be in one of these three modes: *primary*, *backup* and *recovery*<sup>31</sup>. At the beginning, one component is in *primary* mode while the other one is in *backup* mode. When a replica is in *primary* mode, it provides the service expected from the redundant system. The two replicas are connected through data connections. Based on the data received and on its own state, each component decides whether it must be the sender of output. When a failure occurs in a component, the component goes to *recovery* mode. If the failed component was in *primary* mode, the other component takes over.

In the previous patterns, the mode transitions of a component or system are controlled by a separate *decider* component. In this pattern, we use self-managing components that control their own mode transitions. This is modeled by local event ports (represented as dotted ovals) triggering the mode transitions.

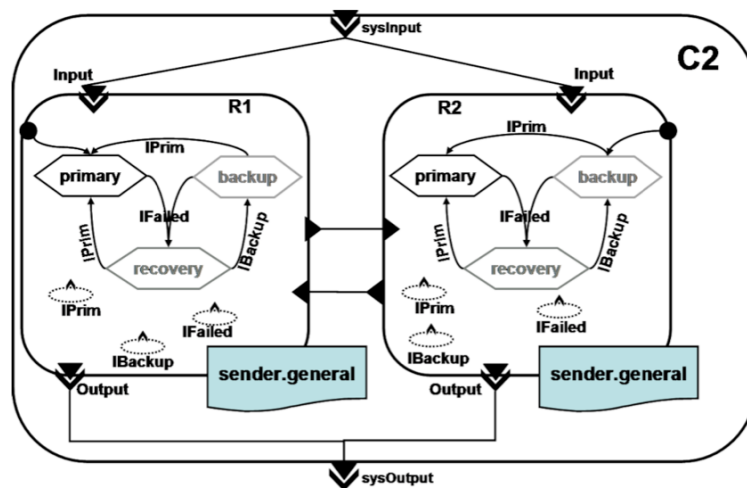


Figure III-21. Active dynamic redundancy pattern with self-checking replicas (variant 1)

The two components' architectural models are identical except for their initial modes, i.e., one is initially in *primary* mode while the other one is in *recovery* mode. Thus, Figure III-22 shows only the textual AADL dependability model of the component that is initially in *primary* mode. **Guard\_Event** properties are associated with all internal ports (expressed by *self.eventname*) named in mode transitions. For example, the first declared **Guard\_Event** property specifies that the component moves to *recovery* mode when it fails.

<sup>31</sup> Here we chose to model explicitly the recovery of each replica through a mode. However, it is possible to consider only two modes: *primary* and *backup*, as in the previously presented models.

```

system replica
features
  Input: in event data port;
  Output: out event data port;
  toReplica: out data port;
  fromReplica: in data port;
end replica;
-----
system implementation replica.primary
modes
  primary: initial mode;
  backup, recovery: mode;
  primary-[self.IFailed]->recovery;
  backup-[self.IFailed]->recovery;
  recovery-[self.IPrim]->primary;
  recovery-[self.IBackup]->backup;
  backup-[self.IPrim]->primary;
annex Error_Model {**
  Model => sender.general;
  Guard_Event => self[Failed] applies to self.IFailed;
  Guard_Event => fromReplica[FailedVisible] and self[Error_Free]
    applies to self.IPrim;
  Guard_Event => fromReplica[Error_Free] and self[Error_Free]
    applies to self.IBackup;
  **};
end replica.primary;

```

Figure III-22. Textual AADL dependability model – active dynamic redundancy (variant 1)

The AADL dependability model of Figure III-23 represents a second model variant of the pattern.

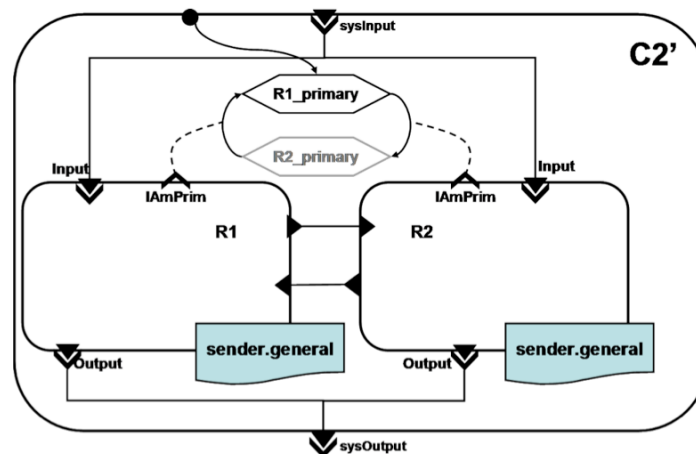


Figure III-23. Active dynamic redundancy pattern with self-checking replicas (variant 2)

Similarly to *C2*, the system *C2'* is formed of two replicas (*R1* and *R2*). The error model *sender.general* is associated with *R1* and *R2*. Unlike *C2*, the system *C2'* has operational modes at the system level instead of having them at the replicas' level. The initial mode is *R1\_primary*, meaning that *R1* plays the role of primary. Based on the data received from the other replica and on its own state, the replica acting as backup can decide to take over by initiating a mode transition. Thus, the transition from *R1\_primary* to *R2\_primary* is triggered by the **out** event port *IAmPrim* of *R2* and the transition from *R2\_primary* to *R1\_primary* is

triggered by the **out** event port *IAmPrim* of *R1*. Further on, in section III.3.1.3, we give guidance on choosing one of the two alternative patterns.

Figure III-24 shows the complete textual AADL dependability model for the *C2'* pattern.

```

system replica
features
  Input: in event data port;
  Output: out event data port;
  toReplica: out data port;
  fromReplica: in data port;
  IAmPrim: out event port;
end replica;
-----
system implementation replica.primary
annex Error_Model {**
  Model => sender.general;
  Guard_Event => fromReplica[FailedVisible] and self[Error_Free]
    applies to IAmPrim;
**};
end replica.primary;
-----
system ADR_v2
features
  sysInput: in event data port;
  sysOutput: out event data port;
end ADR_v2;
-----
system implementation ADR_v2.generic
subcomponents
  R1: system replica.generic;
  R2: system replica.generic;
connections
  event data port sysInput->R1.Input;
  event data port sysInput->R2.Input;
  event data port R1.Output->sysOutput;
  event data port R2.Output->sysOutput;
  data port R1.toReplica->R2.fromReplica;
  data port R2.toReplica->R1.fromReplica;
modes
  R1_primary: initial mode;
  R2_primary: mode;
  R1_primary-[R2.IAmPrim]->R2_primary;
  R2_primary-[R1.IAmPrim]->R1_primary;
end ADR_v2.generic;

```

**Figure III-24.** Textual AADL dependability model - active dynamic redundancy (variant 2)

The replicas have the same component type and implementation, given in the upper part of the figure. The lower part of the figure corresponds to the whole system type and implementation. The **Guard\_Event** property associated with the **out** event port *IAmPrim* models the mode switch condition.

### III.3.1.2 Software fault-tolerance

Each of the subsequent subsections first presents the specifications of a fault-tolerance policy and then comments on the suitable AADL pattern to be used.

### III.3.1.2.1 N-version programming

In the N-version programming policy [Avizienis 1995], the system is formed of N diversified replicas and a *voter* whose role is to detect disagreements between replicas. If one replica fails, the voter detects a disagreement between the failed replicas and the error-free replicas. If  $N=3$ , one fault is tolerated, as the voter provides the “good” result as output of the fault tolerant system. If more than  $(N-2)$  replicas fail or if the voter fails, the system is considered as failed.

The AADL dependability model of this policy, tolerating one fault, is formed of three replicas that receive the same input and of a voter that receives all replicas’ outputs. Based on the results received and on its own state (we assume that the voter is self-checking), the voter decides which result to send out or whether the system failed. Notice that the AADL dependability model built for the N-modular redundancy policy captures this behavior (see Figure III-21). One may associate component-dependent Occurrence properties for events and propagations, to model the asymmetric behavior of the versions.

### III.3.1.2.2 Recovery block

In the recovery block policy [Randell & Xu 1995], the system is formed of several replicas, called *alternates*, and a decider, which is an *acceptance test*. The acceptance test is applied sequentially to the results of the alternates. If the result of one alternate does not satisfy the test, the next alternate executes. The system is considered as failed if none of the alternates provides an acceptable result or if the acceptance test failed.

The AADL dependability model of this policy is formed of two replicas. One replica executes and sends its result to the decider that applies the acceptance test. If the acceptance test fails, i.e., if the decider detected a failure in the active replica, the second replica executes. Notice that the AADL dependability model built for the cold standby sparing policy captures this behavior (see Figure III-13). It is possible to customize the pattern with software-specific Occurrence properties of events and propagations.

### III.3.1.2.3 N self-checking programming

In the N self-checking programming policy [Laprie *et al.* 1990], software components are able to verify their correct operation [Yau & Cheung 1975]. This is achieved by introducing redundancy in the component. After the detection of an error, a recovery procedure can be attempted to correct the abnormal behavior. Tolerance of a single fault is achieved by the parallel execution of two self-checking software replicas. One replica acts as *primary* and the other one acts as *backup*. The two replicas communicate one with the other. If the replica that acts as primary detects its failure, it notifies the backup, which becomes primary.

The AADL dependability model of this policy is formed of two self-checking replicas that communicate one with the other. Each replica detects its own failure. Based on the internal detection mechanisms and on the data received from the other replica, it decides whether to be the primary or the backup. The AADL dependability model built for the active dynamic redundancy policy with self-checking components captures this behavior (see Figure III-21).

### III.3.1.3 Summary and observations

Patterns are reusable AADL dependability models that may be instantiated directly in a system instance model or that may be extended to form other patterns. Table III-1 summarizes our patterns for fault-tolerance policies.

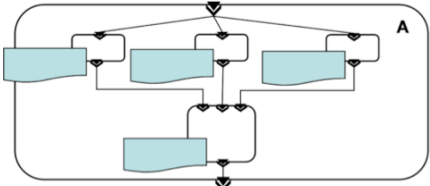
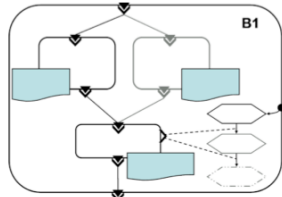
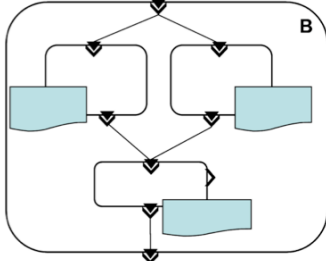
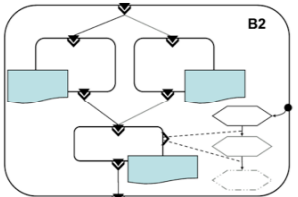
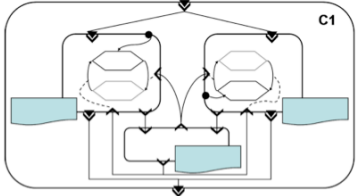
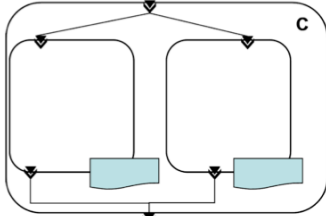
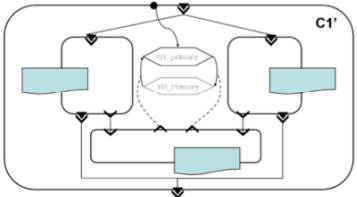
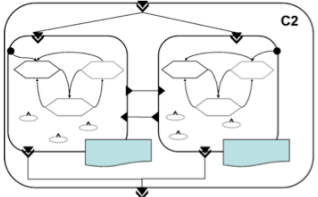
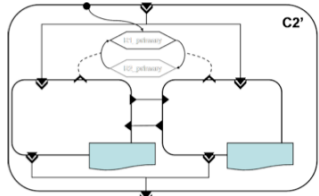
The first two columns of Table III-1 represent fault-tolerance policies for hardware and software. The third column shows the corresponding patterns while the last column shows higher-level patterns, which are *ancestors* of groups of patterns. An ancestor is defined as a general model that does not model a particular fault-tolerance policy but that can be refined to obtain patterns for several strategies. Patterns *B1* and *B2* extend pattern *B* (i.e., the three components and the corresponding connections). *B1* and *B2* customize this model by adding modes and by making components / connections modal. Patterns *C1*, *C1'* and *C2*, *C2'* extend pattern *C*, which models the two replicas and their connections to the input and output of the system. *C1* and *C2* customize this model by adding modes to replicas, while *C1'* and *C2'* add modes at the system level. Other components and connections are added as well. The advantage of modeling operational modes at the system level is that the pattern can be refined to consider that connections or components inside the system are active in one of the modes and inactive in the other. The advantage of modeling operational modes in the components is that the components may be refined during the development cycle, to represent their subcomponents that may be active in one mode and inactive in the other.

Other patterns may be defined on the same basis, to reflect other mechanisms related to fault detection and recovery actions.

An AADL architectural pattern of duplex master-slave redundancy was presented in [Feiler *et al.* 2004]. Its goal is to facilitate the understanding of the functional architecture by clearly showing what is replicated in the architectural model and what the active system components are. Our patterns additionally include:

- 1) Information about the fault detection and decision-making mechanisms (expressed mainly through **Guard** properties).
- 2) A customizable layer of dependability-related information (error/failure and recovery behavior) and of dynamics necessary for evaluating dependability measures. This layer can be easily abstracted away in order not to clutter the higher-level view of the architecture.

**Table III-1. Pattern summary**

| HW FT  | SW FT                              | Pattern  | Ancestor pattern  |
|--|------------------------------------|--|---|
| N-Modular Redundancy (3 replicas)            | N-Version Programming (3 replicas) |    |   |
| Cold Standby Sparing                         | Recovery Block                     |     |    |
| Warm Standby Sparing                         |                                    |    |   |
| Hot Standby Sparing                          |                                    |  |  |
|  |                                    |  |   |
| Active Dynamic with Self-Checking Components | N Self-Checking Programming        |   |   |
|  |                                    |  |   |

### ***III.4 Conclusion***

In this chapter, we first gave general guidance for modeling independent components and various types of dependencies (including maintenance strategies) in AADL. Then, we presented reusable patterns for common fault tolerance schemes. For genericity, we have used components of type system to define the patterns. AADL v2.0 will provide enhanced support for defining patterns in AADL. A new and generic component category will be introduced to define patterns that are then customized to a particular component category. Moreover, the user will be able to explicitly specify parameters for the pattern, e.g., whether it is formed of identical or different components.

We showed that reusability is achieved at several modeling levels: the level of error models, the level of architectural models and the level of dependability models. Patterns may be stored in libraries and instantiated in order to be used in particular models. The OSATE toolset supports the definition of such libraries.

In our iterative modeling approach, presented in Section I.4, each dependency is modeled separately. The order for introducing dependencies does not impact the final AADL dependability model. However, it may impact the reusability of parts of the model. Thus, the order may be chosen according to the context of the targeted analysis. The AADL dependability model may be progressively validated based on its associated GSPN. The automated transformation from AADL to GSPN is based on the rules given further on, in the following Chapter.

---

## IV AADL to GSPN Model Transformation

---

This Chapter presents the complete set of transformation rules corresponding to the AADL Error Model Annex constructs presented in Section II.1.2. We exemplify the transformation rules on AADL Error Model Annex construct examples already presented in Section II.1.2. The only difference lies in the names used for the states and propagations. In the current Chapter, we chose to use generic names, such as *outProp* for **out** propagations, *inProp* for **in** propagations and *s* for states, in order to give a very general view of the model transformation. All transformation rules are first presented and then formalized using the notations of section II.2, related to Petri nets. All rules are defined to ensure that the obtained GSPN is correct by construction (bounded and free of infinite loops over sets of vanishing markings) under the assumption that the AADL model has been built following the guidelines presented in Chapter III. A small set of the transformation rules presented in this Chapter has been published in [Rugina *et al.* 2007].

This Chapter is structured into nine sections. Section IV.1 gives an overview of the whole transformation process. Sections IV.2 to IV.7 are devoted to the transformation rules to be used to transform an AADL dependability model into a GSPN. Section IV.8 discusses issues related to the scalability of the transformation while Section IV.9 concludes the Chapter.

### IV.1 Overview of the transformation

As presented in Section I.4.3, the GSPN obtained by transforming the AADL dependability model is structured as a set of subnets: *component subnets* that model the behavior of components in the presence of their own faults and repair events, and *dependency subnets* that model the behavior associated with dependencies. This subsection gives an overview of the transformation rules that are detailed later on in the current Chapter.

Section IV.2 presents the rules for transforming independent components, to create the component subnets. Components' error models are processed by taking into account their states and transitions triggered by events. Propagations are ignored.

Section IV.3 deals with dependencies described by name-matching **in** - **out** propagations. First, **out** propagations are identified. For each **out** propagation, the AADL architectural model is traversed following the dependency rules defined in the AADL Error Model Annex (see Section II.1.2.2) in order to find **in** propagations that occur as effects of the **out** propagation. The name-matching **in** - **out** propagations are then transformed, according to the given rule, into dependency subnets that are connected to the component subnets.

Section IV.4 is devoted to the rules for transforming propagation filtering and masking mechanisms, i.e., **Guard\_In** and **Guard\_Out** properties. For each such property found in the AADL dependability model, we search for the components owning the propagations and states named in the Boolean expressions of the property. The resulting dependency subnets are connected to the component subnets.

Section IV.5 presents the rules targeting mechanisms for connecting error states to modes, i.e., **Guard\_Event** and **Guard\_Transition** properties, and **activate** / **deactivate**



transitions. **Guard\_Event** and **Guard\_Transition** properties are dealt with in a similar way to **Guard\_In** and **Guard\_Out** properties. First, the AADL dependability model is traversed to identify the components or connections owning the elements named in the Boolean expressions. The transformation rules are then applied in order to obtain a dependency subnet to be connected to the component nets.

Section IV.6 focuses on the rule for transforming hierarchical AADL dependability models. In particular, derived error models must be transformed up to the system level.

Section IV.7 concerns the customization of the existing GSPN subnets, to take into account architecture configurations with operational modes (e.g., error propagations are only broadcasted out of an active component).






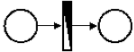
In the presentation of the transformation rules of AADL Error Model Annex constructs that use Boolean expressions, we assume that Boolean expressions are in disjunctive normal form (DNF). It is noteworthy that the user is free to use any legal Boolean expression. Thus, before performing the transformation rules, the tool must transform the Boolean expressions into DNF. A Boolean expression is in DNF if it is a *disjunction* (sequence of **ORs**) consisting of one or more disjuncts, each of which is a *conjunction* (**AND**) of one or more variables and negations of variables. A variable is a propagation name or a state name.

The following six subsections present successively the transformation rules, without detailing the AADL dependability model traversal to search for dependent components. This traversal is implementation-dependent.

## IV.2 Transforming error models of independent components

In the case of an independent component or in the case of a set of independent components, the AADL to GSPN transformation is rather straightforward, as an error model represents a stochastic automaton. Table IV-1 shows the basic transformation rules.

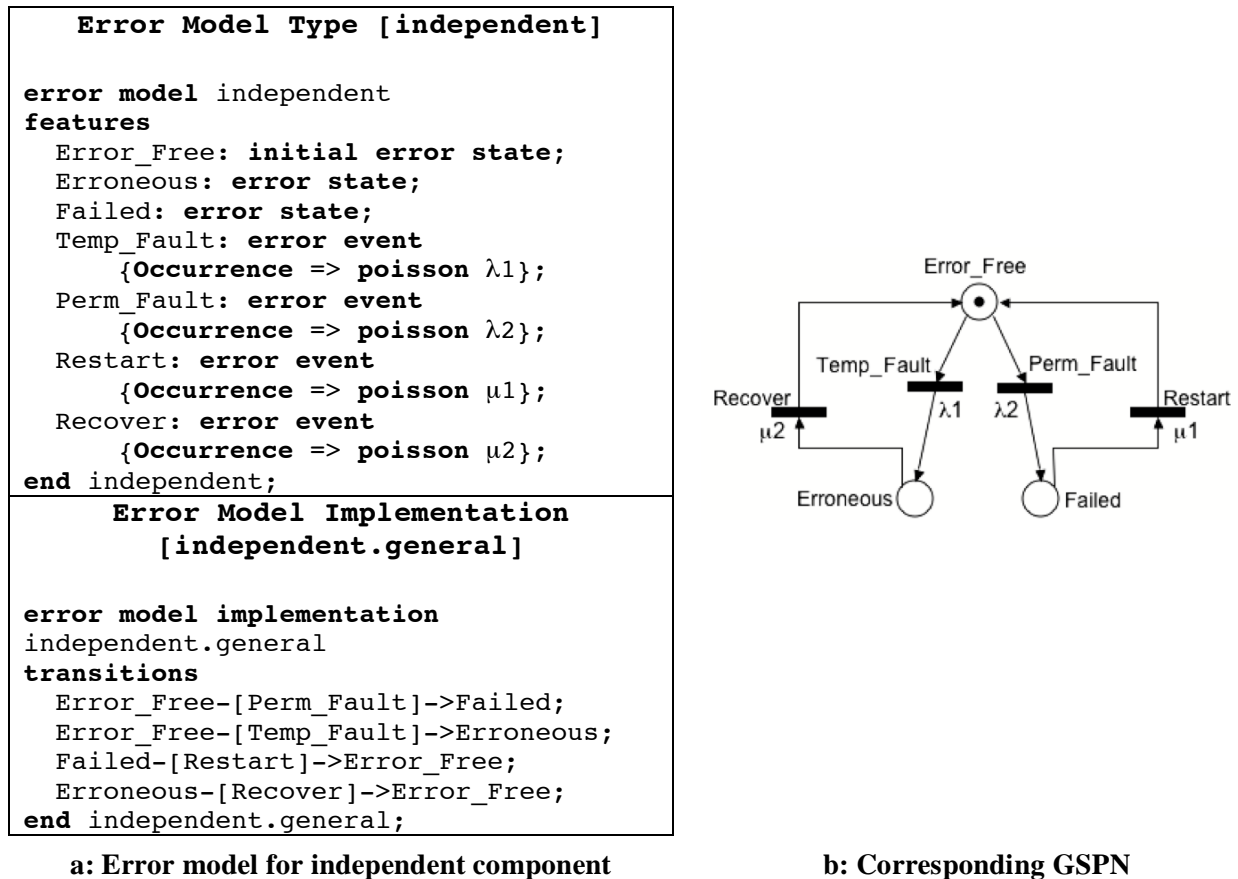
**Table IV-1.** Basic AADL error model to GSPN transformation rules

| AADL error model construct                           | GSPN element   |   |
|--|--|---|
| State  | Place  |            |
| Initial state  | Place with token   |            |
| Event  | GSPN transition (timed or immediate)   |            |
| Occurrence property <sup>32</sup>                    | Distribution or probability, as specified by the AADL Occurrence property, characterizing the occurrence of the associated GSPN transition |  Timed     |
|  |  |  Immediate |
| AADL transition<br>(Src_State-[Event] -> Dest_State) | Arcs connecting places (corresponding to AADL Src_State and Dest_State) via GSPN transition (corresponding to AADL Event)                  |            |

<sup>32</sup> AADL allows specifying a fixed probability, a Poisson distribution or another (non-standard) distribution. Since we use GSPNs, AADL Occurrence properties must be either fixed probabilities or Poisson distributions.

An error model of an independent component is transformed into a component subnet. As stated before, propagations are not considered in this rule, as they should not be used in models for independent components.

The number of tokens in a component subnet is always one, as a component can only be in one state. By applying the transformation rules presented in Table IV-1 to the error model example for an independent component, shown in Figure IV-1-a, we obtain the GSPN of Figure IV-1-b.



**Figure IV-1.** Illustration of the transformation rule for independent components

### IV.3 Transformation of basic dependency elements

All transformation rules for dependent components are based on the choice of the transformation rules for **out** and **in** propagations, as **out** and **in** propagations are the basic mechanisms for representing interactions between AADL components (as stated in Section III.2). Consequently, it is important that the rules for **out** and **in** propagations be generic, to simplify the definition of the other rules and to favor the modularity of the GSPN. To this end, we first reason about **out** propagations and **in** propagations in a dissociated manner despite the fact that they are strongly connected from a semantic point of view (i.e., an **in** propagation in a receiver component or connection only occurs as a consequence of an **out** propagation or of a set of **out** propagations). Then, we show how to connect the GSPN

subnets of **out** and **in** propagations through the use of name-matching subnets. Finally, we discuss our choice of the transformation rule for name-matching **in** - **out** propagations.

### IV.3.1 out propagations

Figure IV-2 presents a general example of AADL transition triggered by an **out** propagation named *Prop*. The source *Out\_src* and the destination *Out\_dst* of the propagation may or may not be the same state. We distinguish two cases with respect to the type of Occurrence property for the **out** propagation:

- Case a: Poisson distribution with parameter  $\sigma$  (Figure IV-2-a);
- Case b: fixed probability  $p$  (Figure IV-2-b). It is assumed that the component does not change its state with probability  $1-p$ .

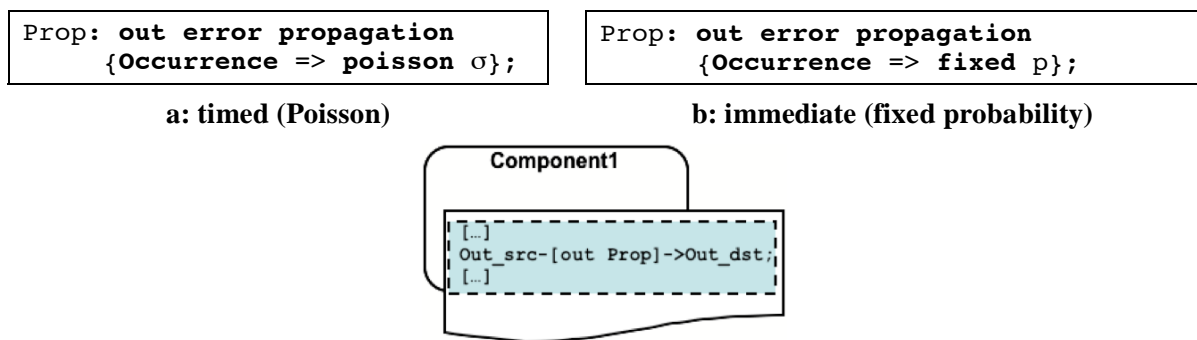


Figure IV-2. AADL transition triggered by an **out** propagation

In Figure IV-2, *Case a* represents a delayed propagation that will not be perceived immediately by its receivers. *Case b* represents an immediate propagation.

#### IV.3.1.1 Rule presentation

The transformation rule is slightly different for these two cases. Figure IV-3-a presents the GSPN corresponding to the transformation rule for *Case a* while Figure IV-3-b presents the one for *Case b*.

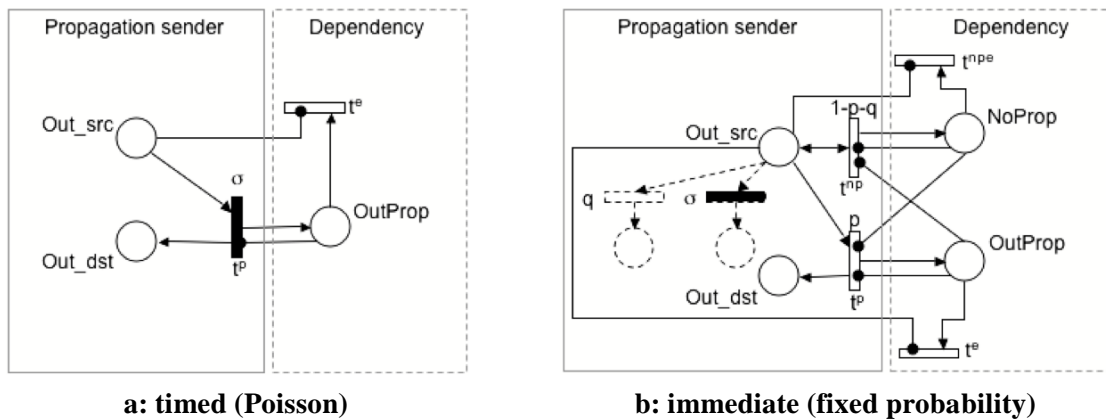


Figure IV-3. Transformation rule for AADL transition triggered by an **out** propagation

In both cases, a GSPN transition  $t^p$  links the place  $Out\_src$  to the place  $Out\_dst$  (corresponding to states  $Out\_src$  and  $Out\_dst$ ).  $t^p$  is characterized by the Occurrence property of the **out** propagation. When firing this transition, a token is created in a place named  $OutProp$ , memorizing the **out** propagation. The place  $OutProp$  is emptied through the immediate transition  $t^e$  when the place  $Out\_src$  is empty, i.e., the **out** propagation is visible while the component is in state  $Out\_src$ , from which the propagation has been generated.

In *Case b*, the place  $NoProp$  models the situation where the propagation does not occur when the component is in  $Out\_src$  state. If the propagation does not occur, the immediate transition  $t^{np}$  is fired. Its associated probability must take into account the sum of probabilities of all events and propagations that trigger transitions from  $Out\_src$ . In Figure IV-3-b the probability associated with  $t^{np}$  is  $(1-p-q)$  because it is assumed that there is an event or a propagation that occurs with probability  $q$  and triggers a transition from the source state  $Out\_src$ . The transition  $t^{npe}$  empties the place  $NoProp$ , with probability 1, when the propagation sender component has moved from  $Out\_src$ .

Also in *Case b*, the subnet formed of the place  $NoProp$  and the transitions  $t^{np}$  and  $t^{npe}$  is not necessary if the error model declares an error event or an **out** propagation having its source in  $Out\_src$  and occurring with a probability  $1-p$  (i.e.,  $q = 1-p$  in Figure IV-3-b). If the subnet is created in this case,  $t^{np}$  would never be fired as its probability would be equal to zero.

GSPN transitions  $t^p$  and  $t^{np}$  are part of the sender component subnet, as an **out** propagation occurs in the sender component independently according to its Occurrence property and has an impact on the sender's state machine. Places  $OutProp$  and  $NoProp$  are part of one or more dependency nets, as they are interfaces between the sender component and receiver components of the architectural model.  $t^e$  and  $t^{npe}$  are also part of one or more dependency nets, as the emptying of  $OutProp$  and  $NoProp$  has to be synchronized to occur after the occurrence of the effects of the propagation on receiver components.

In general, a named **out** propagation could trigger  $n$  AADL transitions in an error model (e.g., an **out** propagation *Failed* could be propagated out both from a *FailStopped* and a *FailRandom* states, each of these states representing different failure modes). A GSPN subnet as one of those presented in Figure IV-3 is created for each one of the  $n$  AADL transitions. Thus, an **out** propagation is active when there is a token in one of its corresponding places.

### IV.3.1.2 Rule formalization

Let  $P^{src}$ ,  $P^{pt}$  and  $P^{pi}$  be the sets of places, and  $T^t$  (for a timed propagation),  $T^i$  (for an immediate propagation) be the sets of GSPN transitions:

$$P^{src} = \{(Out\_src, Out\_dst) \in \text{sender component net}\}$$

$$P^{pt} = \{OutProp \in \text{dependency net}\}$$

$$P^{pi} = \{OutProp, NoProp \in \text{dependency net}\}$$

$$T^t = \{t^p: \text{timed transition} \in \text{sender component net}, t^e \in \text{dependency net}\}$$

$$T^i = \{t^p, t^{np} \in \text{sender component net}, t^e, t^{npe} \in \text{dependency net}\}$$

The necessary arcs are as follows.

$$\rightarrow t^p = \{Out\_src\} \qquad \rightarrow t^e = \{OutProp\}$$

$$\leftarrow t^p = \{Out\_dst, OutProp\} \qquad \leftarrow t^e = \emptyset$$

$$\circ t^p = \{NoProp \text{ (if } \exists NoProp), OutProp\} \qquad \circ t^e = \{Out\_src\}$$

$$\rightarrow t^{np} = \{Out\_src\} \qquad \rightarrow t^{npe} = \{NoProp\}$$

$$\begin{aligned} \leftarrow t^{np} &= \{Out\_src, NoProp\} \\ \circ t^{np} &= \{NoProp, OutProp\} \end{aligned}$$

$$\begin{aligned} \leftarrow t^{npe} &= \emptyset \\ \circ t^{npe} &= \{Out\_src\} \end{aligned}$$

Let us define the sets of arcs necessary for the subnets describing respectively a timed and an immediate **out** propagation, as follows.

$$A_{outPropag\_timed} = A_{t^p} \cup A_{t^e}$$

$$A_{outPropag\_immediate} = A_{t^p} \cup A_{t^e} \cup A_{t^{np}} \cup A_{t^{npe}}$$

The GSPN subnet  $PN_{1\_outPropag}$  describing the AADL transition triggered by the **out** propagation is defined as follows.

Case a:  $PN_{1\_outPropag} = P^{src} \cup P^{pt} \cup T^t \cup A_{outPropag\_timed}$  (propagation with Poisson distribution)

Case b:  $PN_{1\_outPropag} = P^{src} \cup P^{pi} \cup T^i \cup A_{outPropag\_immediate}$  (propagation with fixed probability)

Let  $n$  be the number of AADL transitions triggered by the **out** propagation  $Prop$  in a propagation sender component. The GSPN subnet describing all AADL transitions triggered by one same **out** propagation is defined as follows.

$$PN_{n\_outPropag} = \bigcup_{i=1}^n PN_{1\_outPropag_i}$$

### IV.3.2 In propagations

Figure IV-4-a presents an example of AADL transition triggered by an **in** propagation named  $Prop$ . Note that  $In\_src$  and  $In\_dst$  are different states.

#### IV.3.2.1 Rule presentation

A GSPN transition  $InProp$  of probability 1 (it certainly occurs when the cause of the **in** propagation occurs) links the place  $In\_src$  to the place  $In\_dst$  as shown in Figure IV-4-b.

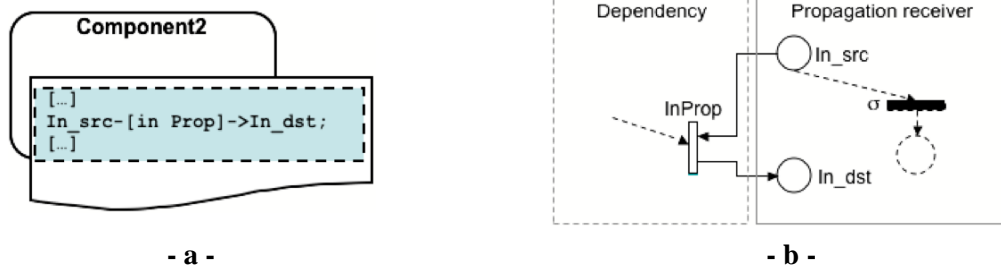


Figure IV-4. Transformation rule for **in** propagation

In general, a named **in** propagation could trigger  $m$  AADL transitions in an error model (i.e., an error propagation may affect its receiver when the latter is in any one of a set of states). An AADL **in** propagation occurs as a consequence of an **out** propagation or of a set of **out** propagations, denoted by the term *cause* further on.  $m$  GSPN transitions are created for each cause of the **in** propagation. Thus, the number of GSPN transitions  $InProp$  is unknown before the complete analysis of dependencies in the architectural model. GSPN transitions  $InProp$  are part of a dependency net and are authorized to fire only when one of their causes occurs.

### IV.3.2.2 Rule formalization

Let  $P^{dst}$  be the set of places and  $T^{in}$  be the set of GSPN transitions:

$$P^{dst} = \{(In\_src_j, In\_dst_j) \in \text{receiver component net}, In\_src \neq In\_dst, j = 1..m\}$$

$$T^{in} = \{InProp_j: \text{immediate transitions} \in \text{dependency net}, j = 1..m\}$$

The necessary arcs are as follows.

$$\rightarrow InProp_j = \{In\_src_j\}$$

$$\leftarrow InProp_j = \{In\_dst_j\}$$

Let us define respectively the set of arcs connected to all GSPN transitions  $InProp_j, j=1..m$ , as follows.

$$A\_InProp = \bigcup_{j=1}^m A\_InProp_j$$

The GSPN subnet  $PN_{inPropag}$  describing the AADL transitions triggered by the **in** propagation  $Prop$  is defined as follows.

$$PN_{inPropag} = P^{dst} \cup T^{in} \cup A\_InProp$$

We presented the case where a named **in** propagation has only one cause. For the general case of several causes for the same **in** propagation, it is sufficient to clone the description above for each cause.

### IV.3.3 Name-matching in – out propagations

For didactical reasons, Sections IV.3.1 and IV.3.2 presented the building blocks representing respectively **out** and **in** propagations, even though **in** propagations are dependent on **out** propagations and have no meaning alone. In the current section, we show how the two subnets are connected together to model a name-matching **in – out** propagation representing a dependency between two components.

Figure IV-5 shows an example of a pair of **in – out** name-matching propagations declared in two connected components. *Component1* plays the role of the propagation sender and it sends propagations named  $Prop$  through the connection that arrives at *Component2*. *Component2* plays the role of a receiver. If it receives a propagation named  $Prop$ , it moves from  $In\_src$  to  $In\_dst$  state. We illustrate name-matching propagations using an AADL architectural model formed of two components connected through data port connections. The result is identical if the architectural model involves components interacting differently (e.g., through bindings or shared data).

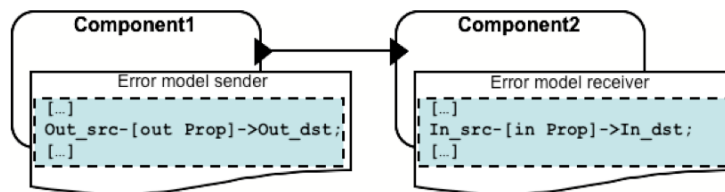


Figure IV-5. Sender and Receiver – **in-out** name matching propagations

### IV.3.3.1 Rule presentation

The transformation rule consists in linking the intermediary place *OutProp* to the transitions *InProp* by bidirectional arcs and in adding inhibitor arcs from the place *In\_src* to *t<sub>e</sub>*, as shown in Figure IV-6. The latter allows avoiding a concurrency situation between *t<sub>e</sub>* and *InProp*. The place *OutProp* must not be emptied before the enabled transition *InProp* is fired. In order not to encumber the figure, we assume that the Occurrence property of the **out** propagation *Prop* is a Poisson distribution. This changes nothing in the way the subnets corresponding respectively to the **out** and to the **in** propagation are connected.

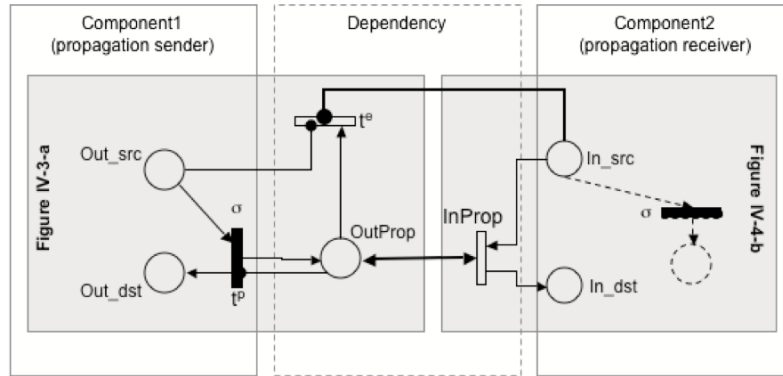


Figure IV-6. GSPN modeling a propagation from a sender to a receiver

Note that, if *Out\_src* and *Out\_dst* are two different states, the **out** propagation is propagated out once and has an immediate effect on receiver components. The place *OutProp* is emptied immediately after. If *Out\_src* and *Out\_dst* are one same state, the **out** propagation is propagated out and has an immediate effect on receiver components if their current states are *In\_src* states. The token remains in the place *OutProp* as long as the propagation sender remains in the state from which the **out** propagation was generated. Thus, the effect of the **out** propagation may be delayed in case the receiver component moves in the meantime in the place *In\_src*. The **in** propagation may even occur several times while the **out** propagation is visible. This behavior is compliant with the semantics of AADL propagations.

### IV.3.3.2 Rule formalization

Let us define the following arcs:

$$\begin{aligned} \vec{InProp}' &= \{OutProp\} & \circ t^e &= \{In\_src\} \\ \leftarrow InProp' &= \{OutProp\} \end{aligned}$$

The GSPN subnet  $PN_{in\_outPropag}$  describing the name-matching AADL transitions is defined as follows.

$$PN_{in\_outPropag} = PN_{outPropag} \cup PN_{inPropag} \cup \vec{InProp}'_x \cup \leftarrow InProp'_x \cup \circ t^e$$

## IV.3.4 Generalization to multiple receivers

Generally, an **out** propagation could trigger  $n$  AADL transitions in its sender component. Name-matching **in** propagations could be declared in  $r \geq 1$  propagation receiver components and trigger  $m_k$  AADL transitions in each  $k$  ( $k = 1 \dots r$ ) receiver component. **In** propagations

are consequences of **out** propagations. This means that in each receiver component  $k$ , each AADL transition triggered by the **in** propagation (among the  $m_k$  AADL transitions) has  $n$  causes corresponding to AADL transitions triggered by the **out** propagation in the sender component.

For each receiver  $j$ ,  $m_j$  GSPN transitions are created for each one of the  $n$  causes of the **in** propagation. This leads to the creation of  $InProp_{ij}$ ,  $i = 1..n, j = 1..m_k$  for each receiver component (as many GSPN transitions for one AADL transition triggered by the **in** propagation as causes of the **in** propagation).

The number of GSPN transitions ( $N_{tr}$ ) necessary to describe the **in** propagations as effects of  $n$  **out** propagations of a sender component on  $r$  receiver components is given by:

$$N_{tr} = n * \sum_{k=1}^r m_k \tag{1}$$

Figure IV-7-a shows an example of an AADL dependability model with one sender propagating out the propagation  $Prop$  from two distinct error states and two receivers, each declaring one AADL transition triggered by the **in** propagation  $Prop$ . It is transformed into the GSPN of Figure IV-7-b. Note that  $N_{tr} = 4$  ( $n = 2, m_1 = 1, m_2 = 1$ ).

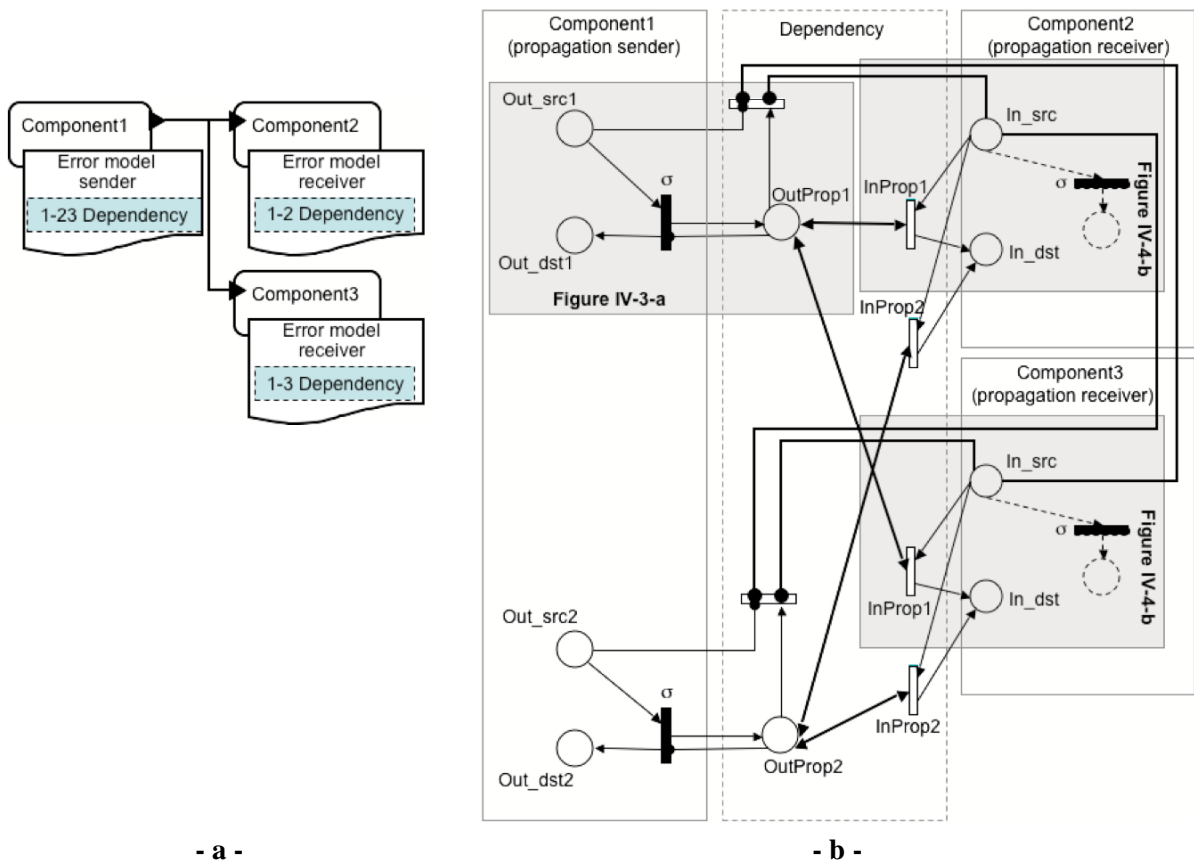


Figure IV-7. Propagations from one sender to two receivers



### IV.3.5 On the choice of transformation rules of **in** – **out** propagations

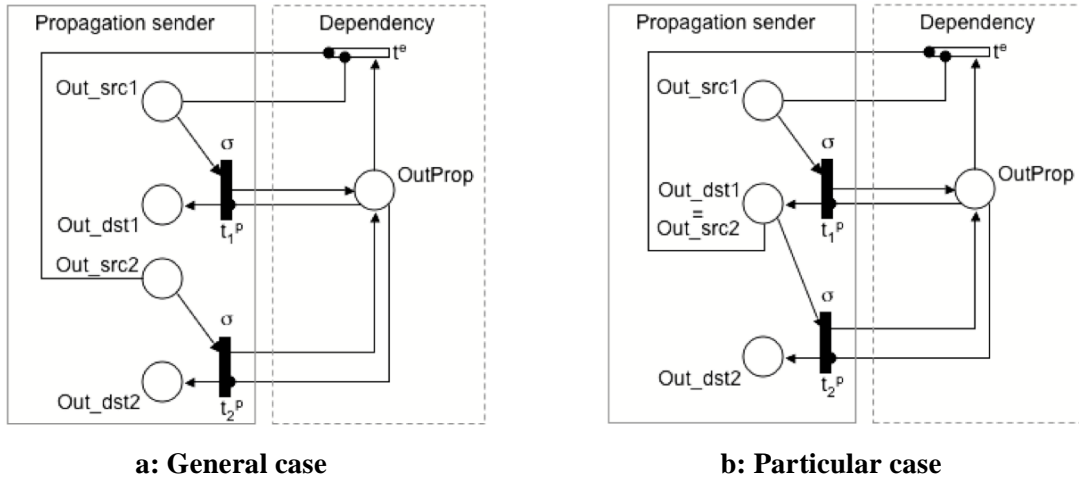
We identified and analyzed several transformation rules for the same AADL specification of **out** propagations and their name-matching **in** propagations. Some of the rules are convenient when an **out** propagation has only one receiver. On the other hand, these rules do not favor subnet reusability and are hard to automate in case of several receivers (e.g., the **in** propagation is declared in several components' error models) for the same **out** propagation. Also, the choice of a transformation rule for **out** propagations and their name-matching **in** propagations impacts the transformation rules for propagation filtering and masking mechanisms and for mechanisms for connecting error states to modes. The transformation rules for **out** propagations, presented in section IV.3.1 and for **in** – **out** name-matching propagations, presented in section IV.3.3 are very well adapted for the case where an **out** propagation has several receivers that declare name-matching **in** propagations or **Guard** properties. They also simplify the definition of the other transformation rules, as the same GSPN propagation place is connected to all dependency subnets on the receivers' side.

To give an example, we show an alternative to the transformation rule presented in sections IV.3.1 and IV.3.3 for **out** propagations. The alternative rule seemed more natural to us at a first glance, as the GSPN obtained is smaller. After further analysis, it turned out that the state spaces are equal in size for the two rules, but the alternative rule requires restricting the AADL modeling power in order to obtain a GSPN without deadlocks.

Let us remind here the general case of  $n$  AADL transitions triggered by the same **out** propagation in an error model. In sections IV.3.1 and IV.3.3 we presented a transformation rule that processes independently the  $n$  AADL transitions, creating a GSPN propagation place for each one of them. Thus, the named **out** propagation is considered to be active when there is a token in one of these propagation places.

The alternative rule consists in creating only one propagation place for a named **out** propagation and in connecting it to the  $n$  GSPN transitions that model the occurrence of the **out** propagation. A GSPN transition  $t_i^p$ ,  $i = 1..n$ , is created for each one of the  $n$  AADL transitions. All  $t_i^p$  are connected to the same place *OutProp*. Figure IV-8-a shows an example with two AADL transitions triggered by the same timed **out** propagation *Prop*. This alternative rule requires restricting the AADL modeling power by forbidding the declaration of transitions triggered by the same **out** propagation from consecutive states. Otherwise the GSPN obtained would not be guaranteed to be free of deadlocks.

Let us assume that *Out\_dst1* and *Out\_src2* are the same place, i.e., the **out** propagation occurs from two consecutive error states. This case is presented in Figure IV-8-b. In this case, the GSPN behavior is as follows. The transition  $t_1^p$  is fired, i.e., the **out** propagation occurs from state *Out\_src1*. A token is created in place *OutProp*, i.e., the propagation becomes active.  $t_1^p$  cannot fire, as the condition to empty *OutProp* is not true.  $t_2^p$  cannot fire either, as *OutProp* is not empty (notice that the inhibitor arc from *OutProp* to  $t_1^p$  is necessary in order to bound the place *OutProp*). Consequently, this GSPN may have a deadlock if no other transition has *Out\_src2* as a source state. The same problem is revealed if *Out\_src1* and *Out\_dst1* are one same place (the source state is the same as the destination state of the **out** propagation) and *Out\_dst1* and *Out\_src2* are two consecutive states.



**a: General case** **b: Particular case**  
**Figure IV-8.** *Alternative transformation rule for out propagation*

## IV.4 Transforming propagation filtering and masking mechanisms

This Section presents the transformation rules for propagation filtering and masking mechanisms, in the form of **Guard\_In** and **Guard\_Out** properties. **Guard\_In** subnets connect **out** and **in** propagation subnets together, while **Guard\_Out** subnets connect **out** propagation subnets together. We also show the transformation of examples of cascading **Guard\_In** and **Guard\_Out** properties.

### IV.4.1 Guard\_In

As stated in Section II.1.2.3.1, a **Guard\_In** property allows the user to conditionally map an incoming set of propagations and error states from other components into a set of in propagations that may affect the receiving component. Figure IV-9 gives the Backus-Naur syntax definition of the **Guard\_In** property, already shown in Section II.1.2.3.1, and develops the definition of the Boolean expression in DNF. This definition is further used to present the general transformation rule before showing a concrete example.

```

Guard_In ::= mapping_rule {, mapping_rule}* applies to inFeature;
mapping_rule ::= (InProp_id | mask) when boolean_expr_DNF
boolean_expr_DNF ::= conjunction | boolean_expr_DNF OR boolean_expr_DNF
conjunction ::= variable | conjunction AND conjunction
variable ::= StateOrPropagation | NOT StateOrPropagation
    
```

**Figure IV-9.** *Guard\_In property syntax*

#### IV.4.1.1 Rule presentation

**Masked mapping\_rules** are ignored, as their Boolean conditions do not impact the component that declares the **Guard\_In** property.

Each non-**masked mapping\_rule** specifies that **in** propagations **InProp\_id** occur as consequences of each **conjunction** of **boolean\_expr\_DNF**. Thus, one GSPN transition

$InProp\_id_{kj}$  is created for each pair ( $conjunction_k, t_j$ ) with  $t_j$  representing an AADL transition triggered by the **in** propagation  $InProp\_id$  in the component that declares the **Guard\_In** property.  $InProp\_id_{kj}$  links the place  $In\_src_j$  to the place  $In\_dst_j$  (corresponding to a source and a destination state of an AADL transition triggered by the **in** propagation  $InProp\_id$ ).  $InProp\_id_{kj}$  is connected through:

- bi-directional arcs to places corresponding to **variables** of **boolean\_expr\_DNF** (states or propagations).
- inhibitor arcs to places corresponding to negated **variables** of **boolean\_expr\_DNF** (**NOT StateOrPropagation**).

If a  $InProp\_id_{kj}$  and a transition that empties a place **StateOrPropagation** are both enabled,  $InProp\_id_{kj}$  is fired before emptying that place. More explicitly, a place corresponding to a **StateOrPropagation** representing a propagation is only emptied when all GSPN transitions connected to it (through non-inhibitor arcs) are disabled.

Figure IV-10-a shows an example of a **Guard\_In** property associated with an **in** port named  $inp1$ . This example and the one shown in Figure II-11 are identical except for the names of the propagations and states referred to in the Boolean expressions, as stressed in the introduction of this chapter.

The **Guard\_In** property is formed of three **mapping\_rules**, one of them being a **mask**. Figure IV-10-b shows a component (*Component1*) having the **Guard\_In** property on its port  $inp1$ . An architectural view of this property is given in Figure II-12. *Component1* has an associated error model with AADL transitions triggered by the **in** propagations named in the **Guard\_In** mapping rules.  $InProp1$  triggers one while  $InProp2$  triggers two AADL transitions.

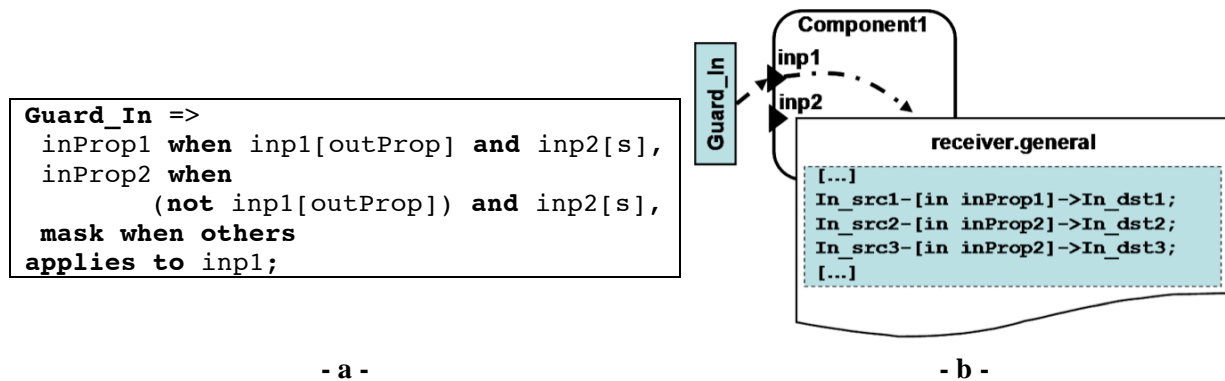


Figure IV-10. *Guard\_In* property

Figure IV-11-a presents the GSPN corresponding to the **Guard\_In** property of Figure IV-10. Each **conjunction** is transformed into a number of immediate GSPN transitions equal to the number of AADL transitions triggered by the **in** propagation named in **mapping\_rule**. Thus, one GSPN transition is created for the first **mapping\_rule** while two GSPN transitions are created for the second one.

For the sake of clarity, the GSPN transitions that empty the place *OutProp* are shown separately in Figure IV-11-b. The place *OutProp* can be emptied when the transition  $InProp1$  is disabled. Note that the transitions  $InProp2_{12}$  and  $InProp2_{22}$  are not considered, as the place *OutProp* is connected to them through inhibitor arcs.  $InProp1$  is disabled either when (1) place  $In\_src1$  is empty, i.e.,  $InProp1$  has been fired or the receiver component was in a

different state at the receipt of the *in* propagation *InProp1*, or when (2) *s* is empty, i.e., none of the Boolean expressions that refer to *OutProp* is true.

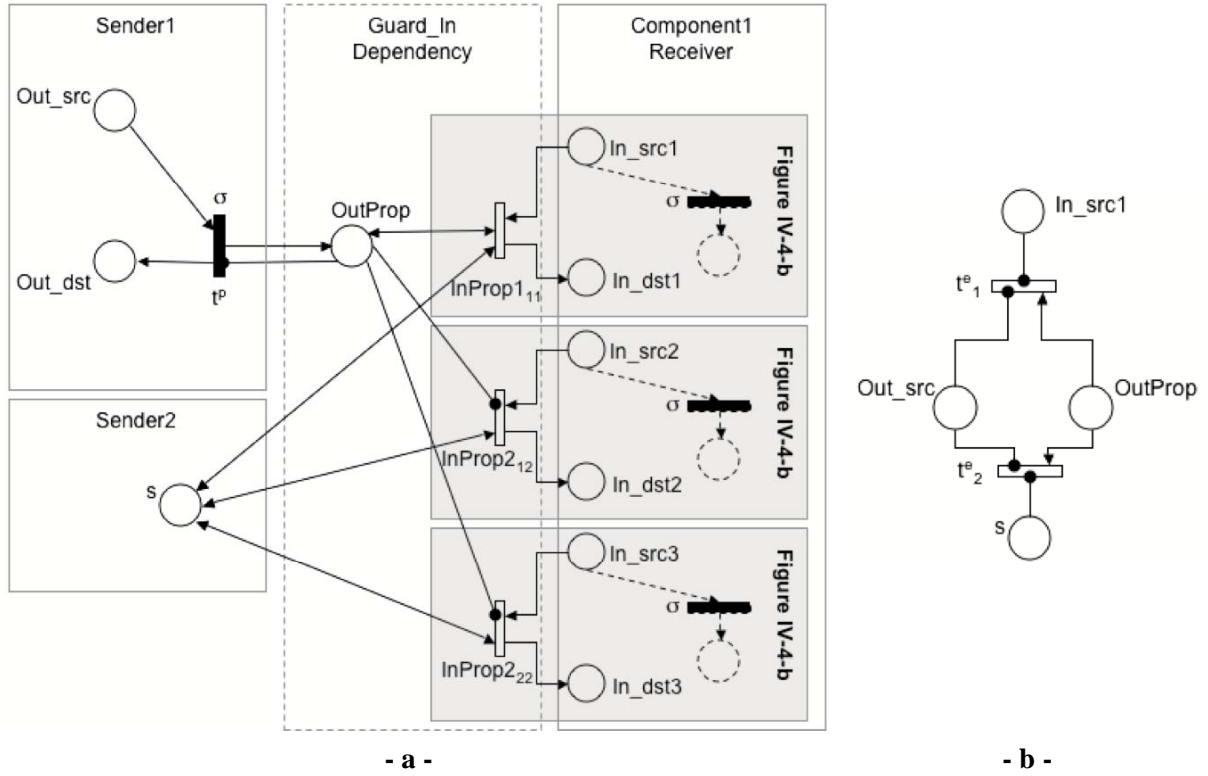


Figure IV-11. Transformation rule for *Guard\_In* property

#### IV.4.1.2 Rule formalization

We assume that states and propagations (*StateOrPropagation*) referred to in the *boolean\_expr\_DNF* have already been transformed into Petri net places. We formalize the transformation rule for a *mapping\_rule* of the *Guard\_In* property without considering the GSPN transitions that empty places *StateOrPropagation* corresponding to propagations. The general rule for emptying propagation places is detailed in Appendix B. We use the following notations:

$m$  = the number of AADL transitions triggered by *InProp\_id* in the component that declares the *Guard\_In* property.

$c$  = the number of *conjunctions* in *mapping\_rule*

Let  $P^{GI}_{\text{mapping\_rule}}$  and  $P^{dst}_{\text{mapping\_rule}}$  be the sets of places, and  $InProp\_id_{kj}$  be the set of GSPN transitions.

$$P^{GI} = \{\text{StateOrPropagation} \in \cup \text{conjunction}_{\in \text{mapping\_rule}}\}$$

$$P^{dst} = \{In\_src_j, In\_dst_j \in \text{receiver component net}, In\_src_j \neq In\_dst_j, j = 1..m\}$$

$$InProp\_id = \{InProp\_id_{kj}; \text{immediate transitions} \in \text{dependency net}, k = 1..c, j = 1..m\}$$

We define the following arcs:

$$\vec{InProp}_{kj} = \{In\_src_j\} \cup \{\text{StateOrPropagation} \in \text{conjunction}_k / \text{variable} = \text{StateOrPropagation}\}$$

$$\begin{aligned} \overleftarrow{InProp}_{kj} &= \{In\_dst_j\} \cup \{ \mathbf{StateOrPropagation} \in \mathbf{conjunction}_k / \\ &\quad \mathbf{variable} = \mathbf{StateOrPropagation} \} \\ \overrightarrow{InProp}_{kj} &= \{ \mathbf{StateOrPropagation} \in \mathbf{conjunction}_k / \\ &\quad \mathbf{variable} = \mathbf{NOT StateOrPropagation} \} \end{aligned}$$

The necessary set of arcs connected to all GSPN transitions  $Inprop$  is as follows.

$$A\_InProp = \bigcup_{k=1}^c \bigcup_{j=1}^m A\_InProp_{kj}$$

The GSPN subnet  $PN_{mapping\_rule}$  describing a **mapping\_rule** of the **Guard\_In** property is defined as follows.

$$PN_{mapping\_rule} = P^{GI} \cup P^{dst} \cup Inprop\_id \cup A\_InProp \cup PN_{InPropag}$$

All **mapping\_rules** (except the one labeled **mask**) are transformed according to the above rule.

Let us consider separately the GSPN transitions that empty a place **StateOrPropagation** corresponding to a propagation. We define the subset  $effects_{StateOrPropagation}$  of the union of  $InProp\_id$  corresponding to all **mapping\_rules** containing all  $InProp$  that have incoming arcs from **StateOrPropagation**.

$$effects_{StateOrPropagation} = \{ InProp \subset (\cup InProp\_id) / \mathbf{StateOrPropagation} \subset \overrightarrow{InProp} \}$$

The cause of **StateOrPropagation** is a  $t^p$  GSPN transition.

**StateOrPropagation** can be emptied if all GSPN transitions connected to it are disabled. This condition is expressed in the following Boolean expression.

$$\neg t^p \wedge \neg effects_{StateOrPropagation}$$

After transforming this expression in DNF, a GSPN transition  $t^e$  is created for each conjunction.

#### IV.4.2 Guard\_Out

As stated in Section II.1.2.3.2, a **Guard\_Out** property allows the user to conditionally pass through an incoming set of propagations and states as an outgoing propagation of the error model associated with the component declaring the **Guard\_Out** property. Figure IV-12 shows the Backus-Naur syntax definition of the **Guard\_Out** property including the Boolean expression in DNF form (which has the same definition as the **Guard\_In** property). We use this definition to present the general transformation rule before showing a concrete example.

```

Guard_Out ::= passThrough_rule {, passThrough_rule}*
               applies to outFeature;
passThrough_rule ::= (OutProp_id | mask) when boolean_expr_DNF
boolean_expr_DNF ::= conjunction | boolean_expr_DNF OR boolean_expr_DNF
conjunction ::= variable | conjunction AND conjunction
variable ::= StateOrPropagation | NOT StateOrPropagation

```

Figure IV-12. *Guard\_Out* property syntax

### IV.4.2.1 Rule presentation

As in the case of **Guard\_In** properties, **masked mapping\_rules** are ignored. When their Boolean conditions are true, no propagation is generated.

One place *OutProp\_id\_outFeature* corresponding to **OutProp\_id** is created for each non-**masked mapping\_rule**. This place models the fact that an **out** propagation **OutProp\_id** occurs as a consequence of each **conjunction** of **boolean\_expr\_DNF** and is sent out through the **outFeature** named in the **applies to** clause. One GSPN transition *OutProp\_id<sub>k</sub>* is created for each **conjunction<sub>k</sub>** of the **mapping\_rule**. *OutProp\_id<sub>k</sub>* has an outgoing arc to the place *OutProp\_id\_outFeature* and is also connected through:

- bi-directional arcs to places corresponding to **variables** of **boolean\_expr\_DNF** (states or propagations).
- inhibitor arcs to places corresponding to negated **variables** of **boolean\_expr\_DNF** (**NOT StateOrPropagation**).

A place **StateOrPropagation** corresponding to a propagation can be emptied when all GSPN transitions connected to it are disabled. Similarly, *OutProp\_id\_outFeature* can be emptied when all GSPN transitions connected to it are disabled (its causes are inactive, i.e.,  $\neg$ **boolean\_expr\_DNF** evaluates to TRUE and all GSPN transitions modeling effects of the **out** propagation are disabled). Appendix B generalizes this rule for emptying places corresponding to **out** propagations resulting from AADL transitions or **Guard\_Out** properties.

Figure IV-13-a shows an example of a **Guard\_Out** property associated with an **out** port named *outp1*. This example and the one shown in Figure II-14 are identical except for the names of the propagations and states referred to in the Boolean expressions. Figure IV-13-b shows *Component1* having the **Guard\_Out** property on port *outp*. An architectural view of this property is given in Figure II-15.

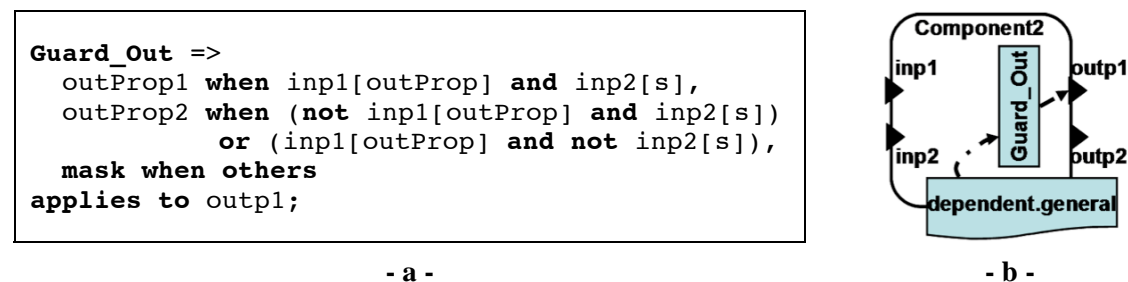


Figure IV-13. *Guard\_Out* property

The **Guard\_Out** property of Figure IV-13 is formed of two non-masked **passThrough\_rules** which are transformed into two places. The **boolean\_expr\_DNF** of the first **passThrough\_rule** is formed of one **conjunction** transformed into one GSPN transition. The **boolean\_expr\_DNF** of the second **passThrough\_rule** is formed of two **conjunctions**, each one transformed into one GSPN transition. Figure IV-14 presents the GSPN corresponding to the **Guard\_Out** property of Figure IV-13 without considering the GSPN transitions that empty the places corresponding to **out** propagations. The places corresponding to **out** propagations occurring as a result of a **Guard\_Out** property are to be interfaced with name-matching **in** propagations subnets and with **Guard\_In** subnets on the receiver side.

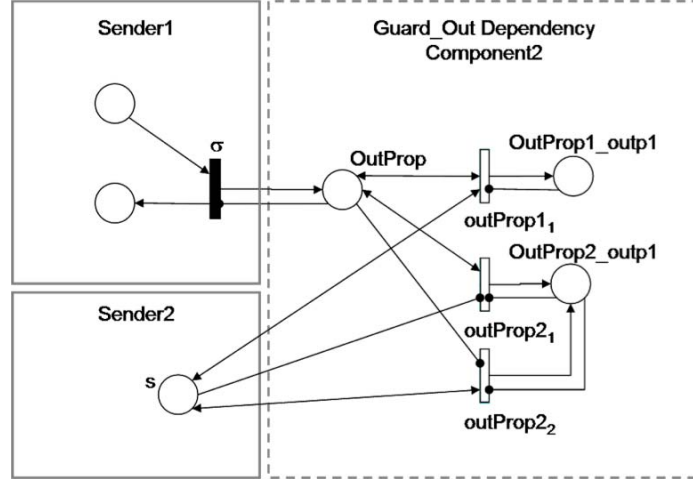


Figure IV-14. Transformation rule for **Guard\_Out** property

#### IV.4.2.2 Rule formalization

We assume that states and propagations (**StateOrPropagation**) referred to in the **boolean\_expr\_DNF** have already been transformed into Petri net places. We formalize the transformation rule for a **mapping\_rule** of the **Guard\_Out** property without considering the GSPN transitions that empty places **StateOrPropagation** corresponding to propagations. The latter rule is presented in Appendix B.

We use the following notations:

$c$  = the number of **conjunctions** in **mapping\_rule**

Let  $P_{\text{mapping\_rule}}^{GO}$  and  $P_{\text{mapping\_rule}}^{gen}$  be the sets of places, and  $OutProp\_id_k$  be the set of GSPN transitions.

$$P^{GO} = \{ \text{StateOrPropagation} \in \cup \text{conjunction}_{\in \text{mapping\_rule}} \}$$

$$P^{gen} = \{ OutProp\_id\_outFeature \in \text{dependency net} \}$$

$$OutProp\_id = \{ OutProp\_id_k: \text{immediate transitions} \in \text{dependency net}, k = 1.. c \}$$

The necessary arcs are as follows.

$$\rightarrow OutProp_k = \{ \text{StateOrPropagation} \in \text{conjunction}_k / \text{variable} = \text{StateOrPropagation} \}$$

$$\leftarrow OutProp_k = \{ OutProp\_id \} \cup \{ \text{StateOrPropagation} \in \text{conjunction}_k / \text{variable} = \text{StateOrPropagation} \}$$

$$\circ OutProp_k = \{ OutProp\_id \} \cup \{ \text{StateOrPropagation} \in \text{conjunction}_k / \text{variable} = \text{NOT StateOrPropagation} \}$$

The set of arcs connected to all GSPN transitions  $OutProp_k$  is defined as follows.

$$A\_OutProp = \bigcup_{k=1}^c A\_OutProp_k$$

The GSPN subnet  $PN_{\text{mapping\_rule}}$  describing a **mapping\_rule** of the **Guard\_In** property is defined as follows.

$$PN_{mapping\_rule} = P^{GO} \cup P^{gen} \cup OutProp\_id \cup A\_OutProp$$

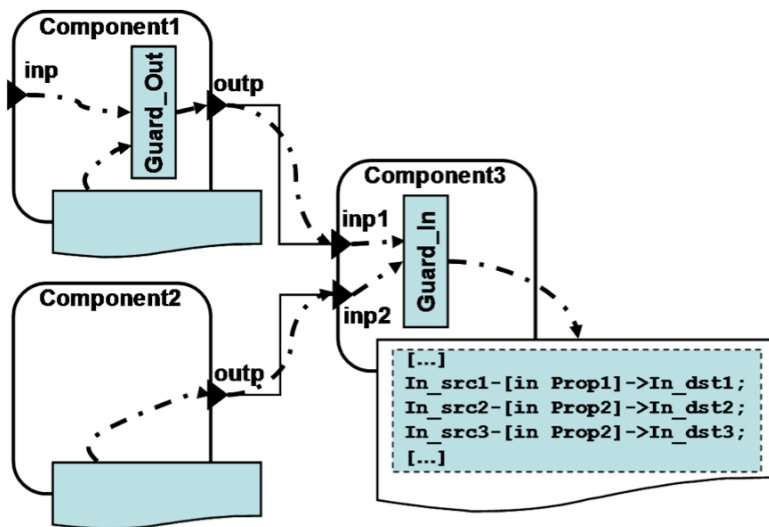
All **mapping\_rules** (except the one labeled **mask**) are transformed according to the rule.

### IV.4.3 Interacting Guard\_In and Guard\_Out properties

This section illustrates the use of the transformation rules presented in Sections IV.4.1 and IV.4.2 in the case of interacting **Guard\_In** and **Guard\_Out** properties. The AADL architectural models and the **Guard** properties used for illustration have already been shown in Section II.1.2.3.4.

#### IV.4.3.1 Cascading Guard\_Out - Guard\_In

Figure IV-15-a presents an architectural model example (the same as in Figure II-16) in which a **Guard\_In** property applying to **in** ports of *Component3* refers to an **out** propagation that occurs as a result of a **Guard\_Out** property applying to the **out** port of *Component1*. The **in** propagations occurring as a result of the **Guard\_In** property trigger AADL transitions in the error model associated with *Component3*. The **Guard\_Out** property associated with the **out** port of *Component1* is given in Figure IV-15-b. The **Guard\_In** property associated with the **in** ports of *Component3* is given in Figure IV-15-c.



- a -

```
Guard_Out =>
  outProp1 when
    inp[outProp] and self[s1],
  mask when others
  applies to Component1.outp;
```

- b -

```
Guard_In =>
  Prop1 when inp1[outProp1] and inp2[s2],
  Prop2 when
    (not inp1[outProp1] and inp2[s2])
    or (inp1[outProp1] and not inp2[s2]),
  mask when others
  applies to Component3.inp1,
  Component3.inp2;
```

- c -

Figure IV-15. Cascading **Guard\_Out** - **Guard\_In** properties

Figure IV-16 shows the GSPN corresponding to the AADL model of Figure IV-15 without taking into account the immediate GSPN transitions that empty propagation places.



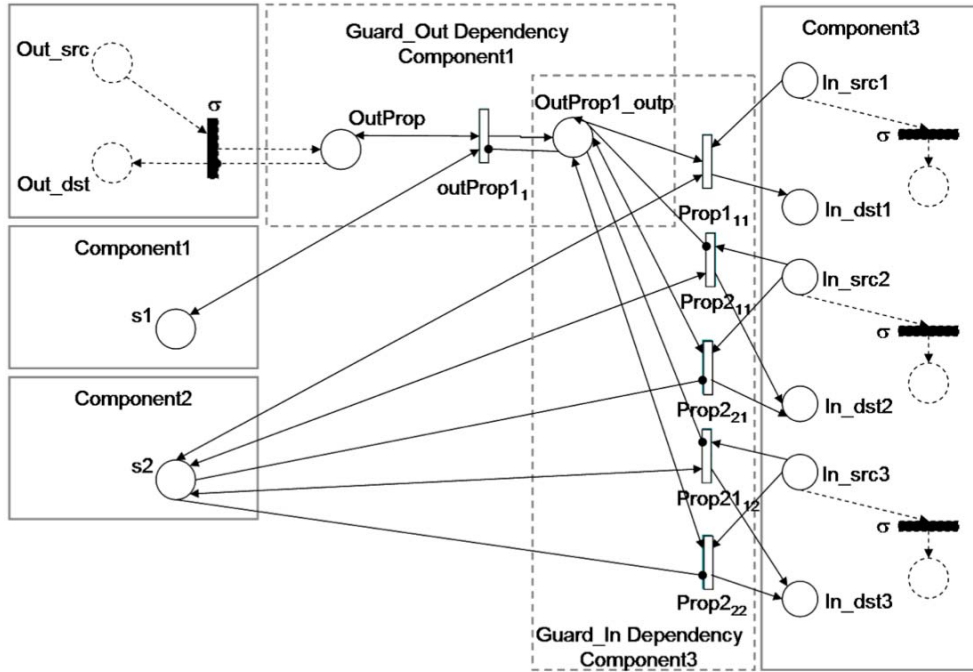


Figure IV-16. GSPN modeling of cascading **Guard\_Out** - **Guard\_In** properties

**Out** propagations that result from **Guard\_Out** properties are represented in the GSPN as places, in the same way as **out** propagations generated by AADL transitions. Thus, the connection between any of the two types of **out** propagation and a **Guard\_In** subnet is performed similarly.

#### IV.4.3.2 Cascading **Guard\_Out** - **Guard\_Out**

Figure IV-17-a presents an architectural model example (the same as in Figure II-17) in which a **Guard\_Out** property applying to an **out** port of *Component2* refers to an **out** propagation that occurs as a result of a **Guard\_Out** property applying to an **out** port of *Component1*. The error model associated with *Component3* declares AADL transitions triggered by **in** propagations name-matching those resulting from the **Guard\_Out** property of *Component2*. The **Guard\_Out** property associated with the **out** port of *Component1* is given in Figure IV-17-b while the one associated with the **out** port of *Component2* is given in Figure IV-17-c.

Places corresponding to **out** propagations that result from **Guard\_Out** properties are connected to other **Guard\_Out** properties in the same way as **out** propagations generated by AADL transitions. Figure IV-18 shows the GSPN corresponding to the AADL model of Figure IV-17 without taking into account the immediate GSPN transitions that empty propagation places.

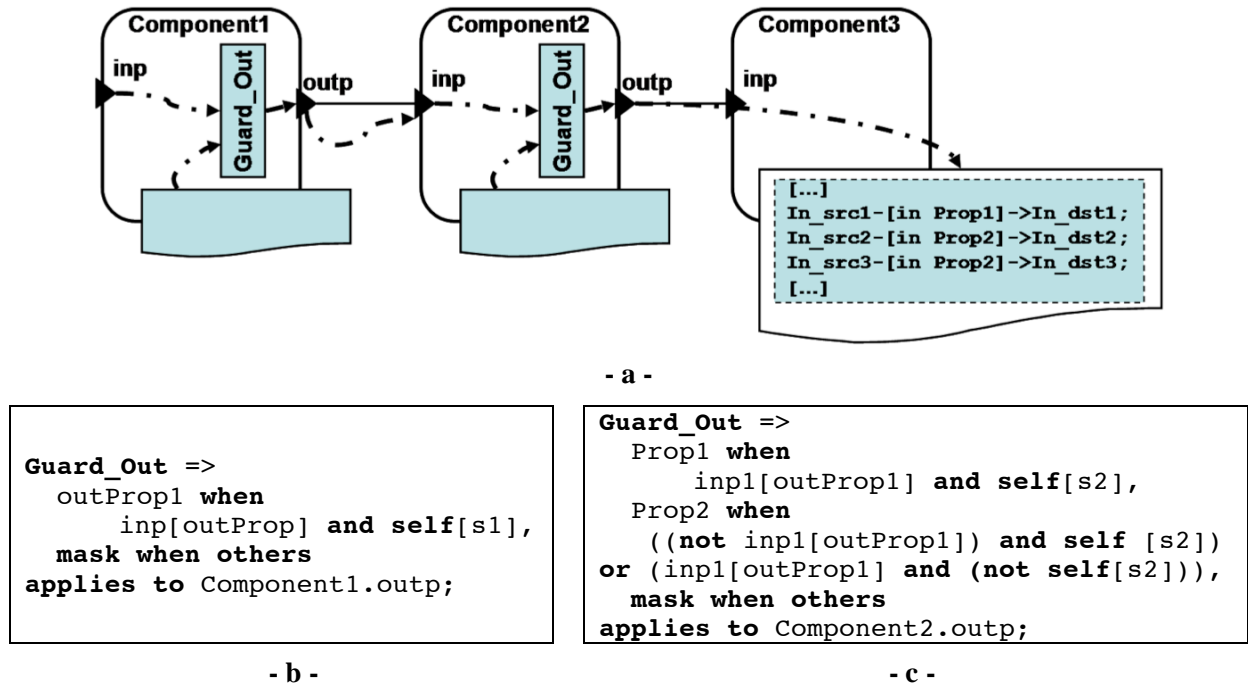


Figure IV-17. Cascading *Guard\_Out* - *Guard\_Out* properties

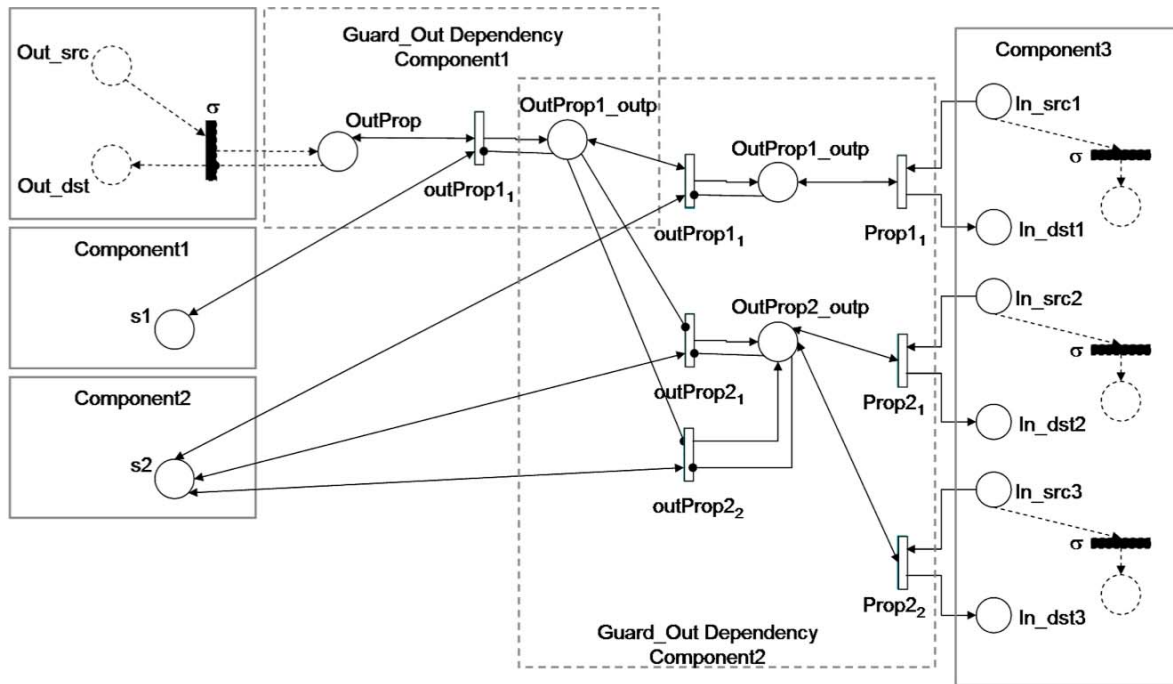


Figure IV-18. GSPN modeling of cascading *Guard\_Out* - *Guard\_Out* properties

### IV.5 Mechanisms for connecting error states to modes

Modes that are of interest in the dependability analysis (i.e, which are involved in mode transitions triggered according to *Guard\_Event* and *Guard\_Transition* properties) are transformed into Petri net places.

In the next subsections we present successively AADL to GSPN transformation rules for (1) **Guard\_Event** properties, (2) **Guard\_Transition** properties and (3) **activate / deactivate** transitions.

### IV.5.1 Guard\_Event

Figure IV-19 shows the Backus-Naur syntax definition of the **Guard\_Event** property including the Boolean expression in DNF.

```

Guard_Event ::= boolean_expr_DNF applies to outEventPort;
boolean_expr_DNF ::= conjunction | boolean_expr_DNF OR boolean_expr_DNF
conjunction ::= variable | conjunction AND conjunction
variable ::= StateOrPropagation | NOT StateOrPropagation

```

Figure IV-19. *Guard\_Event* property syntax

#### IV.5.1.1 Rule presentation

The main difference between a **Guard\_Out** and a **Guard\_Event** property is that the **Guard\_Out** property has several path-through rules while the **Guard\_Event** property only has one rule that maps a Boolean expression to an architectural event. Thus, the transformation rule for a **Guard\_Event** property is similar to the one applied to a path-through rule of a **Guard\_Out** property.

One place  $e\_outEventPort$  is created for a **Guard\_Event** property associated with **outEventPort**. This place models the fact that an architectural event occurs as a consequence of each **conjunction** of **boolean\_expr\_DNF** and is sent out through the **outEventPort** named in the **applies to** clause. One GSPN transition  $ev_i$  is created for each **conjunction<sub>i</sub>** of **boolean\_expr\_DNF**.  $ev_i$  has an outgoing arc to the place  $e\_outEventPort$  and is also connected through:

- bi-directional arcs to places corresponding to **variables** of **boolean\_expr\_DNF** (states or propagations).
- inhibitor arcs to places corresponding to negated **variables** of **boolean\_expr\_DNF** (**NOT StateOrPropagation**).

It is noteworthy that a place corresponding to a propagation may have been created when transforming an AADL transition triggered by the **out** propagation or when transforming a **Guard\_Out** property.

An immediate GSPN transition  $ev\_m_jm_k$  is created for each mode transition that may be triggered by the event occurring as a result of the **Guard\_Event** property. It is connected through:

- an incoming arc from the place corresponding to the source mode.
- an outgoing arc to the place corresponding to the destination mode.
- a bi-directional arc to the place  $e\_outEventPort$ .

The place  $e\_outEventPort$  can be emptied when all GSPN transitions connected to it are disabled (its causes are inactive, i.e.,  $\neg$ **boolean\_expr\_DNF** evaluates to **TRUE** and all GSPN transitions modeling effects of the architectural event are disabled). The general rule presented in Appendix B also applies to places that model architectural events.

Figure IV-20-a shows the same example of **Guard\_Event** property as Figure II-19. It is associated with an **out** port named *outp*. Figure IV-20-b shows *Component3* having the **Guard\_Event** property on its **out** event port *outp*. Events sent out through the port *outp* are routed to the **in** port *inp* of *Component4* and trigger mode transitions from mode *m1* to *m2*.

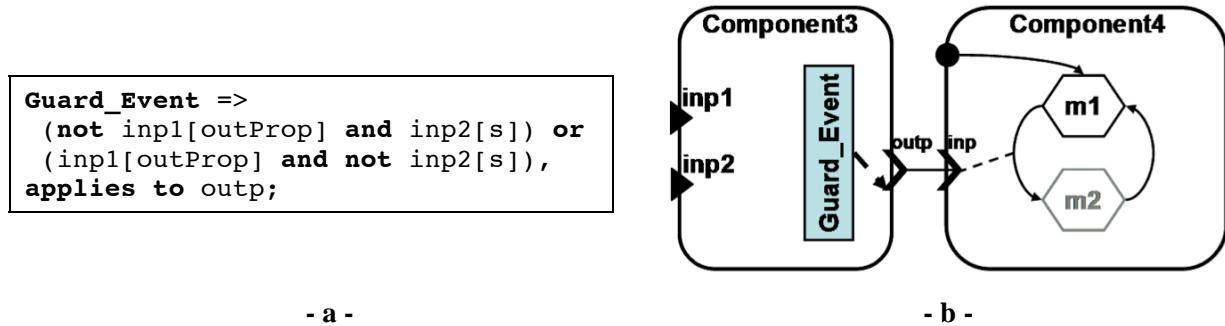


Figure IV-20. *Guard\_Event* property

The **Guard\_Event** property of Figure IV-20 is transformed into a place *ev\_outp*. Its **boolean\_expr\_DNF** is formed of two **conjunctions**, each one transformed into one GSPN transition. Figure IV-21 presents the GSPN corresponding to the **Guard\_Event** property of Figure IV-20 without considering the two immediate GSPN transitions that empty *ev\_outp*. The existence of a token in the place *ev\_outp* leads to transferring the token from place *m1* to place *m2*.

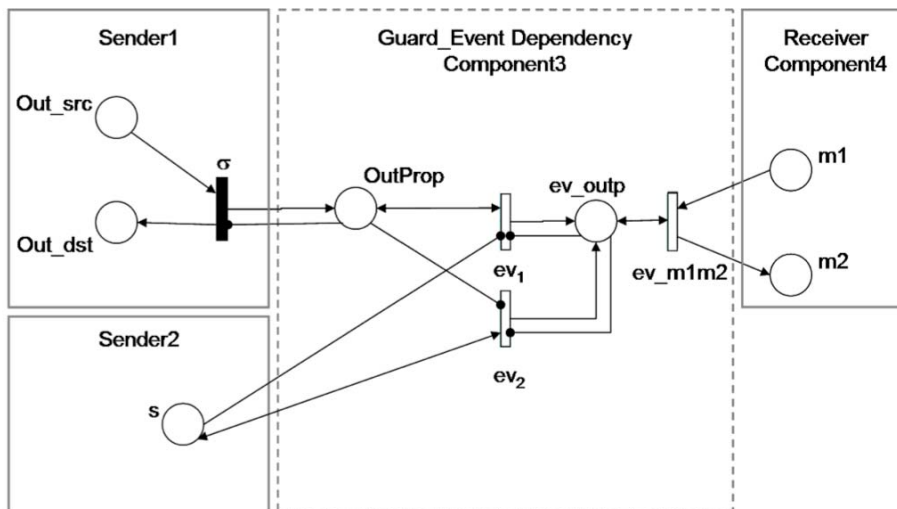


Figure IV-21. Transformation rule for *Guard\_Event* property

### IV.5.1.2 Rule formalization

We assume that states and propagations (**StateOrPropagation**) referred to in the **boolean\_expr\_DNF** have already been transformed into Petri net places. We formalize the transformation rule for a **Guard\_Event** property without considering the GSPN transitions that empty places **StateOrPropagation** corresponding to propagations. We use the following notation:

*c* = the number of **conjunctions** in **boolean\_expr\_DNF**

The place  $e\_outEventPort$  corresponds to the event resulting from the **Guard\_Event** property. We define the sets of places  $P^{GE}$  (places corresponding to states and propagations referred to in **boolean\_expr\_DNF**) and  $P^{modes}$  (places corresponding to modes that are sources or destinations of mode transitions triggered by the event resulting from the **Guard\_Event** property) and the sets of GSPN transitions  $ev$  (corresponding to **conjunctions** of **boolean\_expr\_DNF**) and  $ev\_m$  (corresponding to mode transitions triggered by the event).

$$\begin{aligned}
P^{GE} &= \{\text{StateOrPropagation} \in \cup \text{conjunction}_{\in \text{boolean\_expr\_DNF}}\} \\
P^{modes} &= \{\text{mode} \in \text{dependency net}\} \\
ev &= \{ev_i: \text{immediate transitions} \in \text{dependency net}, i = 1..c\} \\
ev\_m &= \{ev\_m_j m_k: \text{immediate transitions} \in \text{dependency net}, j \neq k / m_j, m_k \in P^{modes}\}
\end{aligned}$$

The necessary arcs are as follows.

$$\begin{aligned}
\rightarrow ev_i &= \{\text{StateOrPropagation} \in \text{conjunction}_i / \\
&\quad \text{variable} = \text{StateOrPropagation}\} \\
\leftarrow ev_i &= \{e\_outEventPort\} \cup \{\text{StateOrPropagation} \in \text{conjunction}_i / \\
&\quad \text{variable} = \text{StateOrPropagation}\} \\
\circ ev_i &= \{e\_outEventPort\} \cup \{\text{StateOrPropagation} \in \text{conjunction}_i / \\
&\quad \text{variable} = \text{NOT StateOrPropagation}\} \\
\rightarrow ev\_m_j m_k &= \{e\_outEventPort\} \cup \{\text{mode}_j\} \\
\leftarrow ev\_m_j m_k &= \{e\_outEventPort\} \cup \{\text{mode}_k\} \\
\circ ev\_m_j m_k &= \emptyset
\end{aligned}$$

Let us define the sets of arcs connected respectively to all GSPN transitions of the set  $ev$  and to all GSPN transitions of the set  $ev\_m$  as follows.

$$A\_ev = \bigcup_{i=1}^c A\_ev_i$$

$$A\_ev\_m_j m_k = \bigcup A\_ev\_m_j m_k$$

The GSPN subnet  $PN_{Guard\_Event}$  describing the **Guard\_Event** property is defined as follows.

$$PN_{Guard\_Event} = P^{GE} \cup P^{modes} \cup ev \cup ev\_m \cup A\_ev \cup A\_ev\_m_j m_k \cup \{e\_outEventPort\}$$

## IV.5.2 Guard\_Transition

**Guard\_Transition** properties associated with mode transitions specify mode transition logic expressions overriding the default **or** condition on events arriving through ports named in the mode transition. Figure IV-22 shows the Backus-Naur syntax definition of the **Guard\_Transition** property, already shown in section II.1.2.4.2, and develops the definition of the Boolean expression in DNF. This definition is further used to present the general transformation rule before showing a concrete example.

```

Guard_Transition ::= boolean_expr_DNF applies to modeTransitionName;
boolean_expr_DNF ::= conjunction | boolean_expr_DNF OR boolean_expr_DNF
conjunction ::= variable | conjunction AND conjunction
variable ::= EventPort | NOT EventPort
EventPort ::= outEventPortOfSubcomp | inEventPortOfComp
    
```

Figure IV-22. *Guard\_Transition* property syntax

### IV.5.2.1 Rule presentation

The main difference between a **Guard\_In** and a **Guard\_Transition** property is that the **Guard\_In** property has several propagation mapping rules while the **Guard\_Transition** property only has one event mapping rule. Thus, the transformation rule for a **Guard\_Transition** property is similar to the one applied to a mapping rule of a **Guard\_In** property. A **Guard\_Transition** subnet is connected on the one hand to places corresponding to events that result from **Guard\_Event** properties and on the other hand to places corresponding to modes involved in the mode transition with which the **Guard\_Transition** property is associated.

The unique event-mapping rule of the **Guard\_Transition** property specifies that the mode transition occurs when one of the **conjunctions** of **boolean\_expr\_DNF** evaluates to **TRUE**. Thus, one GSPN transition  $guard\_tr_i$  is created for each  $conjunction_i$ .  $guard\_tr_i$  links the place  $m_j$  that corresponds to the source mode of the transition to the place  $m_k$  that corresponds to its destination mode.  $guard\_tr_i$  is connected through:

- bi-directional arcs to places corresponding to **variables** of  $conjunction_i$  (events).
- inhibitor arcs to places corresponding to negated **variables** of  $conjunction_i$  (**NOT EventPort**).

Figure IV-23-a reminds the example of Figure II-22: a **Guard\_Transition** property associated with a mode transition named M1toM2, from mode  $m1$  to  $m2$ . This property specifies that the mode transition must occur only when two events arrive simultaneously through ports  $inp1$  and  $inp2$ . Figure IV-23-b shows the architectural view of the property.

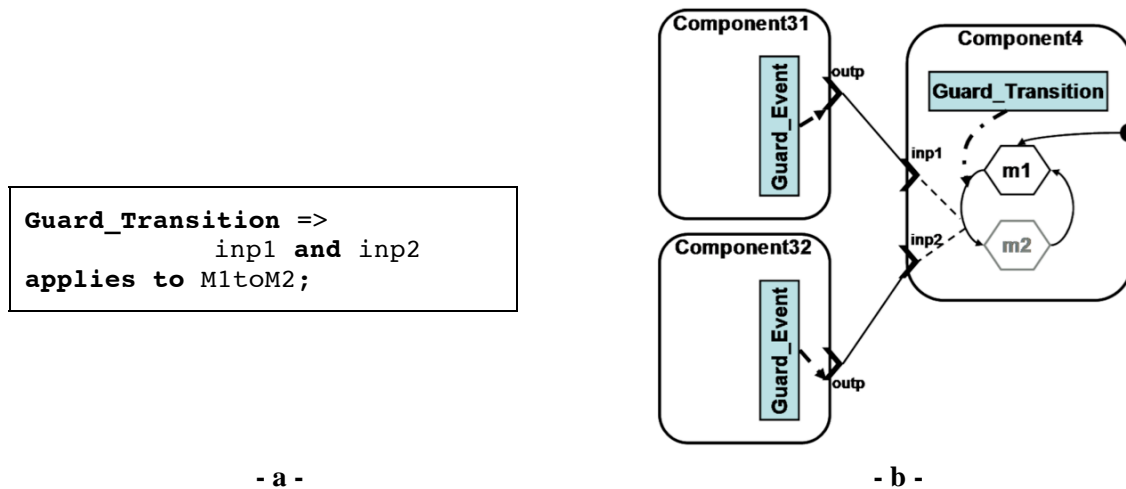


Figure IV-23. *Guard\_Transition* property

For the sake of clarity, Figure IV-24 presents the GSPN corresponding to the **Guard\_Transition** property of Figure IV-23 without considering the GSPN transitions that empty the places corresponding to events.

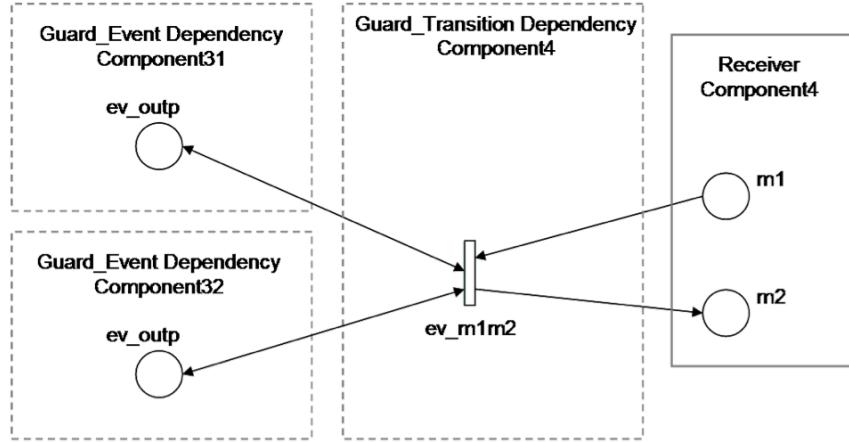


Figure IV-24. Transformation rule for **Guard\_Transition** property

The **boolean\_expr\_DNF** is formed of one **conjunction** that is transformed into an immediate GSPN transition. The condition necessary to empty these places is the same as that for emptying places corresponding to propagations (see Appendix B).

#### IV.5.2.2 Rule formalization

We assume that events occurring as a result of a **Guard\_Event** property and referred to in the **boolean\_expr\_DNF** of the **Guard\_Transition** property have already been transformed into Petri net places (i.e., the **Guard\_Event** properties must be transformed into GSPN before the **Guard\_Transition** properties). We formalize the transformation rule for the **Guard\_Transition** property without considering the GSPN transitions that empty places corresponding to events. We use the following notation:

$c$  = the number of **conjunctions** in **mapping\_rule**

Let us define the sets of places  $P^{GT}$  (corresponding to events occurring through the event ports named in the **applies to** clause of the **Guard\_Transition** property) and  $P^{modes}$  (places corresponding to modes that are sources or destinations of the mode transition triggered by the event ports of the **applies to** clause of the **Guard\_Transition** property) and the set of GSPN transitions  $guard\_tr$  (corresponding to **conjunctions** of **boolean\_expr\_DNF**).

$$P^{GT} = \{\mathbf{EventPort} \in \cup \mathbf{conjunction}\}$$

$$P^{modes} = \{\mathit{mode}_j, \mathit{mode}_k \in \text{dependency net} / j \neq k\}$$

$$guard\_tr = \{guard\_tr_i; \text{immediate transitions} \in \text{dependency net}, i = 1.. c\}$$

The necessary arcs are as follows.

$$\rightarrow guard\_tr_i = \{\mathit{mode}_j\} \cup \{\mathbf{EventPort} \in \mathbf{conjunction}_i / \mathbf{variable} = \mathbf{EventPort}\}$$

$$\leftarrow guard\_tr_i = \{\mathit{mode}_k\} \cup \{\mathbf{EventPort} \in \mathbf{conjunction}_i / \mathbf{variable} = \mathbf{EventPort}\}$$

$$\circ guard\_tr_i = \{\mathbf{EventPort} \in \mathbf{conjunction}_i / \mathbf{variable} = \mathbf{NOT EventPort}\}$$

Let us define the set of arcs connected to all GSPN transitions of set  $guard\_tr$  as follows.

$$A\_guard\_tr = \bigcup_{i=1}^c A\_guard\_tr_i$$

The GSPN subnet  $PN_{Guard\_Transition}$  describing a **Guard\_Transition** property is defined as follows.

$$PN_{Guard\_Transition} = P^{GT} \cup P^{modes} \cup guard\_tr \cup A\_guard\_tr$$

### IV.5.3 Activate / deactivate transitions

Error models declaring **activate / deactivate** transitions must also declare an **initial inactive error state**, in addition to the **initial error state**. As any error state, the **initial inactive error state** is transformed into a GSPN place.

If the component or connection with which the error model is associated is active in the initial operational mode of the system, then a token is placed in the **initial error state**. Otherwise a token is placed in the **initial inactive error state**.

#### IV.5.3.1 Rule presentation

Let us denote by  $m_{current}$  the current operational mode of a system.  $m_{current}$  is a set of operational modes of the system's components that are of interest for the dependability analysis, i.e., they have been transformed into GSPN places when processing **Guard\_Event** and **Guard\_Transition** properties. A component or a connection is active in  $m_{current}$  if it is active in one of the modes of  $m_{current}$  and if all its enclosing components are also active.

We create places  $CompOrConn_{active}$  to model the fact that  $CompOrConn$  is active in  $m_{current}$  and connect them to places corresponding to modes as follows.

- A Boolean expression representing the condition for  $CompOrConn$  to be active is built. It refers to modes of  $CompOrConn$  (in which  $CompOrConn$  is active) and to modes in which its enclosing components are active. This Boolean expression is brought to DNF.
- An immediate GSPN transition is created for each conjunction of the Boolean expression. It is connected through
  - bi-directional arcs to places corresponding to modes referred to in the conjunction.
  - an output arc to the place  $CompOrConn_{active}$ . The place is 1-bounded by using inhibitor arcs to all immediate GSPN transitions that represent conjunctions of the Boolean expression.

**Activate** and **deactivate** transitions are transformed into immediate GSPN transitions that link the place corresponding to the source state of the AADL transition to the place corresponding to the destination state of the AADL transition.

Each GSPN transition corresponding to an **activate** AADL transition is connected through a bi-directional arc to  $CurrentCompOrConn_{active}$ , i.e., it is enabled immediately after the activation of the component or connection with which it is associated.

Each GSPN transition corresponding to a **deactivate** AADL transition has an inhibitor arc from the place  $CurrentCompOrConn_{active}$ .

Places  $CompOrConn_{active}$  must be emptied, similarly to propagation places: all GSPN transitions connected to the place must be disabled.



Figure IV-25 shows an example of AADL dependability model in which *Component11* (which is a subcomponent of *Component1*) has an error model declaring **activate** / **deactivate** transitions. The whole system modelled here has already been presented in Figure II-4. It is formed of three components: *Component1*, *Component2* and *Component3*. Here, we refine *Component1*. It has two subcomponents: *Component11* and *Component12*, and two operational modes. *Component11* is only active in mode *m11*.

The operational modes of the system and of *Component1* have been transformed into places when transforming the **Guard\_Event** properties associated with the ports triggering them. *Component11* is active when the places corresponding to *m1* and *m11* contain tokens.

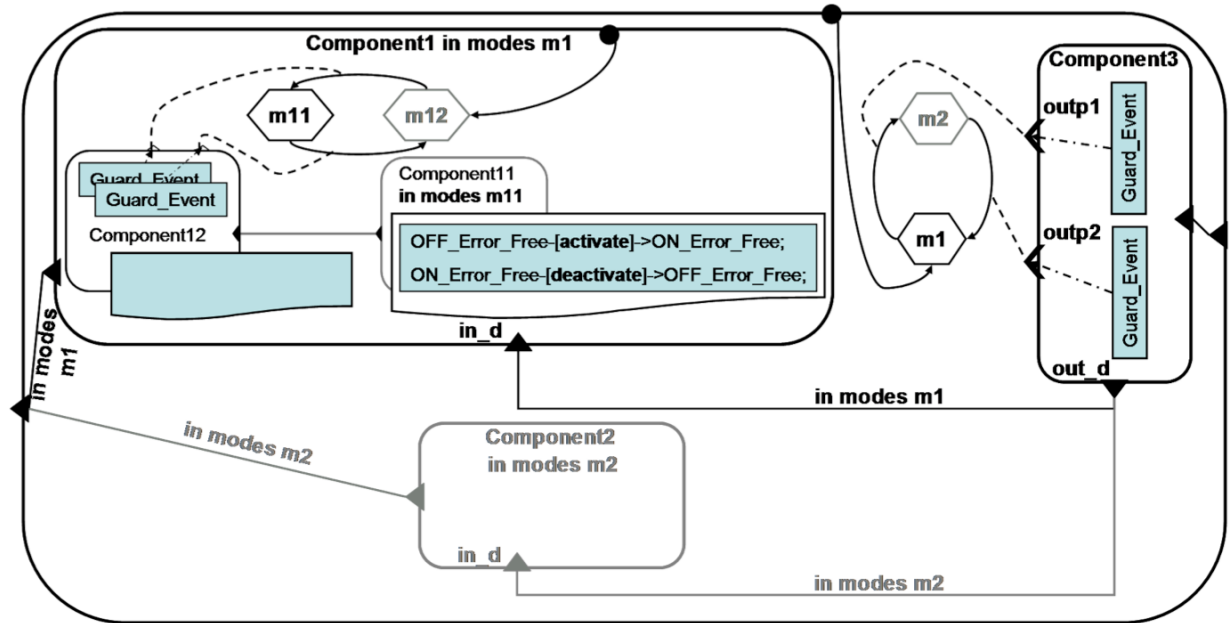


Figure IV-25. Architectural view of **activate/deactivate** transitions

Figure IV-26 presents the GSPN corresponding to the **activate** / **deactivate** transitions.

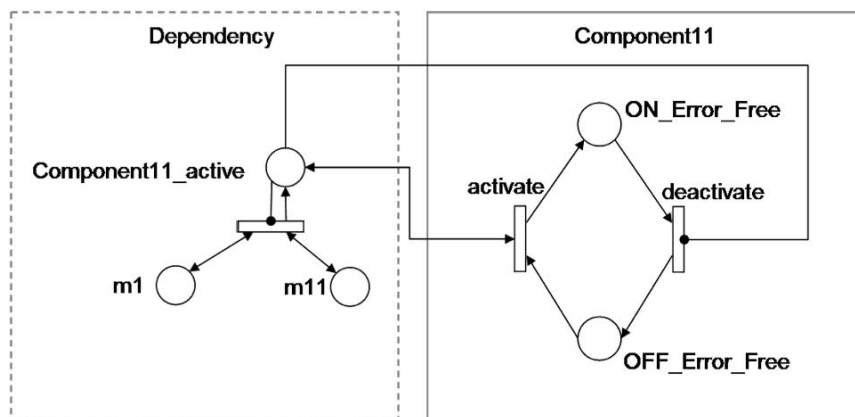


Figure IV-26. **Activate/deactivate** transitions - Transformation rule

The **activate** transition has a bi-directional arc from the place *Component11\_active*. The **deactivate** transition has an inhibitor arc from the place *Component11\_active*. The place

contains a token if places  $m1$  and  $m11$  contain tokens, i.e., both  $Component1$  and  $Component11$  are active.

### IV.5.3.2 Rule formalization

A Boolean expression denoted by  $CompOrConn_{activate}$ , representing the condition for  $CompOrConn$  to be active is built. It refers to modes in which  $CompOrConn$  is active and to modes in which its enclosing components are active. We consider the DNF form of  $CompOrConn_{activate}$ . We use the following notation:

$c$  = the number of *conjunctions* in  $CompOrConn_{activate}$

Let us define the sets of places  $P^{CompOrConn_{active}}$  (places corresponding to components and connections being active) and the sets of transitions  $T^{activate}$  (corresponding to conditions of activation of  $CompOrConn$ ) and  $T^{incoming}$  (corresponding to **in** propagations into  $CompOrConn$  or to **Guard\_transition** properties defined in  $CompOrConn$ ).

Let us define the sets of places  $P^{AD}$  (corresponding to places corresponding to source and destination states of **activate** and **deactivate** transitions),  $P^{config}$  (places corresponding to modes of  $CompOrConn$  and of its enclosing components), and  $P^{CompOrConn_{active}}$  (places corresponding to components and connections being active) and the sets of GSPN transitions  $T^{AD}$  (corresponding to **activate** and **deactivate** transitions) and  $T^{activate}$  (corresponding to conditions of activation of  $CompOrConn$ ).

$$P^{AD} = \{(src_i, dst_i) \in \text{component net}\}$$

$$P^{config} = \{mode \in \text{dependency net}\}$$

$$P^{CompOrConn_{active}} = \{CompOrConn_{active} \in \text{dependency net}^{33}\}$$

$$CurrentCompOrConn_{active} \in P^{CompOrConn_{active}}$$

$$T^{AD} = \{t_i^{activate}, t_i^{deactivate} \in \text{component net}\}$$

$$T^{activate\_comp} = \{t_i^{activate\_comp} \in \text{dependency net} / i = 1..c\}$$

The necessary arcs are as follows.

$$\rightarrow t_i^{activate} = \{src_i\} \cup \{CurrentCompOrConn_{active} \in P^{CompOrConn_{active}}\}$$

$$\leftarrow t_i^{activate} = \{dst_i\} \cup \{CurrentCompOrConn_{active} \in P^{CompOrConn_{active}}\}$$

$$o_i^{activate} = \emptyset$$

$$\rightarrow t_i^{deactivate} = \{src_i\}$$

$$\leftarrow t_i^{deactivate} = \{dst_i\}$$

$$o_i^{deactivate} = \{CurrentCompOrConn_{active} \in P^{CompOrConn_{active}}\}$$

$$\rightarrow t_i^{activate\_comp} = \{mode \in (\text{conjunction}_i \in CompOrConn_{activate})\}$$

$$\leftarrow t_i^{activate\_comp} = \{mode \in (\text{conjunction}_i \in CompOrConn_{activate})\} \cup \{CurrentCompOrConn_{active} \in P^{CompOrConn_{active}}\}$$

$$o_i^{activate\_comp} = \{CurrentCompOrConn_{active} \in P^{CompOrConn_{active}}\}$$

---

<sup>33</sup> These places are part of dependency nets, as they are only necessary if the components and connections represented as active are not independent.

Let us define the set of arcs connected respectively to all GSPN transitions of set  $T^{AD}$  and  $T^{activate\_comp}$  as follows.

$$A_{T^{AD}} = \bigcup_i (A_{t_i^{activate}} \cup A_{t_i^{deactivate}})$$

$$A_{T^{activate\_comp}} = \bigcup_{i=1}^c A_{t_i^{activate\_comp}}$$

The GSPN subnet  $PN_{AD}$  describing **activate** and **deactivate** transitions is defined as follows.

$$PN_{AD} = P^{AD} \cup P^{config} \cup P^{CompOrConn\_active} \cup T^{AD} \cup T^{activate\_comp} \cup A_{T^{AD}} \cup A_{T^{activate\_comp}}$$

## IV.6 Transforming error model abstractions

The AADL Error Model Annex specifies two error model abstractions: *abstract error models* representing the behavior of a component in the presence of faults in terms of states and events inherent to the component and propagations from and to components this component interacts with, and *derived error models* representing the behavior of a component in terms of global states as a logic expression of the states of its subcomponents. These two mechanisms are considered for transformation in the two following subsections.

### IV.6.1 Transforming abstract error models

Abstract error models are transformed according to the rules presented in the preceding sections. The subcomponents of a component having an abstract error model are not transformed. From a practical point of view, the architectural model hierarchy is traversed from top to down to search the components and connections whose error models and **Guard** properties are to be transformed. When a component having an abstract error model is found, the traversal stops and that error model is considered as part of the set of error models to be transformed.

### IV.6.2 Transforming derived error models

Derived error models use Boolean expressions only referring to states (and not to propagations). These expressions determine the state of the derived error model (i.e., the global states). Figure IV-27 reminds the Backus-Naur syntax definition for the **Derived\_State\_Mapping** expression.

```
Derived_State_Mapping ::= stateMapping_rule {, stateMapping_rule}*
stateMapping_rule ::= globalState_id when boolean_expr_DNF
```

Figure IV-27. *Derived\_State\_Mapping* definition

**IV.6.2.1 Rule presentation**

Global states (*globalstate\_id*) of the system correspond to places in the GSPN. Each *boolean\_expr\_DNF* is transformed into a set of immediate GSPN transitions. These transitions are connected through arcs (or inhibitor arcs in case of negations) to places that correspond to states of the subcomponents. Only one place corresponding to a *globalstate\_id* can be marked at a given time. Thus, the number of GSPN transitions corresponding to a *conjunction* is equal to  $n-1$  ( $n$  being the number of places corresponding to global states). Each GSPN transition has an input arc coming from a place corresponding to a global state (which is emptied when the transition is fired) and an exit arc going to the place corresponding to the *globalstate\_id* named in the current *stateMapping\_rule*. Initially, a token is placed in the place that corresponds to the global initial state of the system. This global initial state is determined from the initial states of the system's subcomponents.

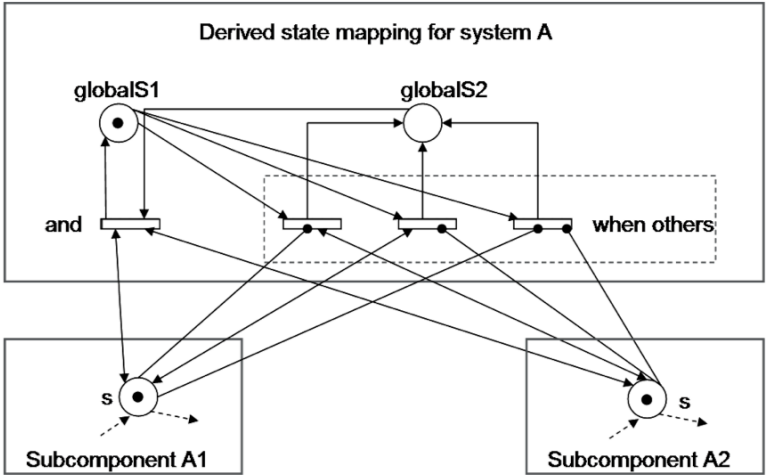
An example is given in Figure IV-28, which shows the implementation of an AADL system *A* with two components named *A1* and *A2* (this example has been already shown in Figure II-26-b2 with other names for the global states and the states of subcomponents). The *Derived\_State\_Mapping* expression specifies that the system is in state *globalS1* if both its subcomponents are in state *s*, and in state *globalS2* otherwise.

```

system implementation A.nominal
subcomponents
  A1: system sw.nominal;
  A2: system sw.nominal;
annex Error_Model {**
  Model => forA;
  Derived_State_Mapping =>
    globalS1 when (A1[s] and A2[s]),
    globalS2 when others;
  **};
end A.nominal;
  
```

**Figure IV-28.** Example of derived error model

Figure IV-29 shows the GSPN obtained after transformation of this derived error model.



**Figure IV-29.** GSPN modeling of the derived error model

The place *globalS1* is marked as we consider *s* states for *A1* and *A2* initial states. If at least one of the two places that correspond to *s* states for *A1* and *A2* is not marked, the place *globalS2* of the derived error model is marked. Note that, *A* cannot be simultaneously in states *globalS1* and *globalS2*. One GSPN transition corresponds to each **conjunction** of each **stateMapping\_rule** (in our example the expression **when others** is formed of three conjunctions).

#### IV.6.2.2 Rule formalization

We assume that the subcomponents' error states have already been transformed in Petri net places. We use the following notations:

$n$  = the number of global states

$c_i$  = the number of **conjunctions** in **mapping\_rule<sub>i</sub>**,  $i = 1..n$

Let us define the sets of places  $P^{global}$  (corresponding to global states) and  $P^{subcomp}$  (corresponding to states of subcomponents) and the sets of GSPN transitions  $T_i^{derived}$  (corresponding to **stateMapping\_rule<sub>i</sub>**) and  $T^{derived}$  (corresponding to all **stateMapping\_rules**).

$$P^{global} = \{globalS_i \in \text{derived dependency net}, i = 1..n\}$$

$$P^{subcomp} = \{subcompS \in \text{subcomponent net} /$$

$$\text{subcompState} \in \text{Uconjunction}_{\in \text{stateMapping\_rule}}\}$$

$$T_i^{derived} = \{t_{jk} : \text{immediate transitions} \in \text{derived dependency net}, j = 1..(n-1), k = 1..c_i\}$$

$$T^{derived} = \bigcup_{i=1}^n T_i^{derived}$$

The necessary arcs are as follows.

$$\rightarrow t_{jk} = \{globalS_j\} \cup \{subcompS / \text{subcompState} \in \text{Uconjunction}_{\in \text{stateMapping\_rule}} \text{ and } \text{variable} = \text{subcompState}\}$$

$$\leftarrow t_{jk} = \{globalS_i\} \cup \{subcompS / \text{subcompState} \in \text{Uconjunction}_{\in \text{stateMapping\_rule}} \text{ and } \text{variable} = \text{subcompState}\}$$

$$o_{t_{jk}} = \{subcompS / \text{subcompState} \in \text{Uconjunction}_{\in \text{stateMapping\_rule}} \text{ and } \text{variable} = \text{NOT subcompState}\}$$

Let us define the set of arcs connected all GSPN transitions of set  $T^{derived}$  as follows.

$$A\_t = \bigcup_{i=1}^n \bigcup_{j=1}^{n-1} \bigcup_{k=1}^{c_i} A\_t_{jk}$$

The GSPN subnet  $PN_{derived}$  describing a **Derived\_State\_Mapping** expression is defined as follows.

$$PN_{derived} = P^{global} \cup P^{subcomp} \cup T^{derived} \cup A\_t$$

### IV.7 Taking into account architecture configurations

This Section provides the rule for taking into account architecture configurations with operational modes in the transformation. Components and connections may be defined as

inactive in particular operation modes, according to Section II.1.1.2. Since inactive components and connections do not communicate with the rest of the system, incoming or outgoing propagations are not possible. Similarly, an inactive component cannot send or receive architectural events if it is inactive. Also, an inactive connection or binding cannot represent a propagation path.

### IV.7.1 Rule presentation

The transformation rule we present hereafter assumes that all the transformation rules presented earlier in this chapter (for name-matching propagations and **Guard** properties) have already been performed. For each component or connection *CompOrConn* having an error model, the rule consists in two steps:

- 1) Creating places *CompOrConn<sub>active</sub>*, if they have not been created earlier. These places model the fact that *CompOrConn* is active.
- 2) Adding bi-directional test arcs from each *CompOrConn<sub>active</sub>* place to each GSPN transition modeling an **in** propagation or a **Guard\_Transition** property. These arcs allow enabling the GSPN transitions when all the following conditions are fulfilled:
  - *CompOrConn* is active in *m<sub>current</sub>*;
  - Possible senders of propagation or architectural event are active in *m<sub>current</sub>*;
  - The propagation path(s) from sender(s) to *CompOrConn* is active in *m<sub>current</sub>* (a place *path<sub>active</sub>* is created in step 1 to represent the fact that a propagation path is active).

Figure IV-30 presents the same example of AADL dependability model with modal architecture configurations used in Figure IV-25 to illustrate the transformation of **activate** and **deactivate** transitions. *Component11* and *Component12* declare name-matching **in - out** propagations.

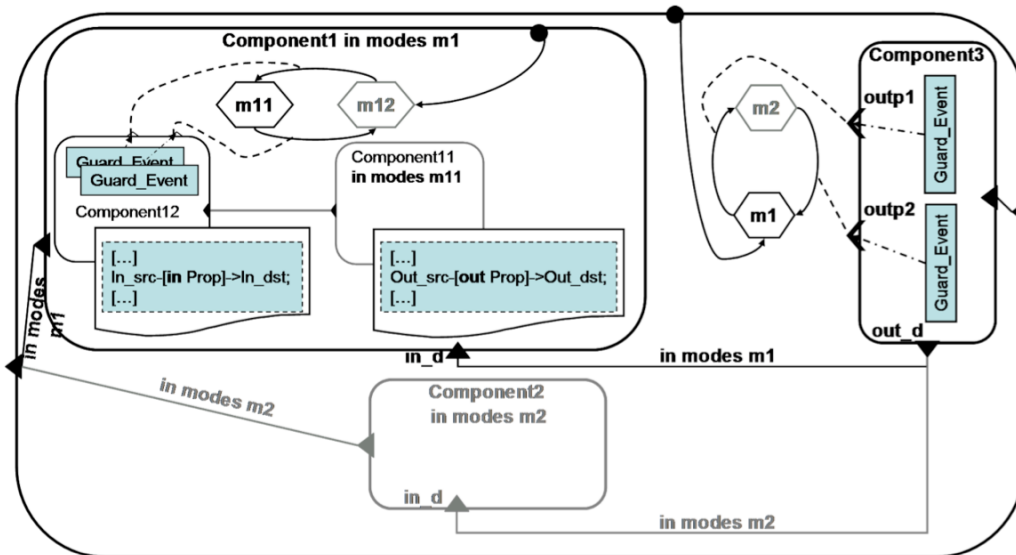


Figure IV-30. System with modal architecture configurations

Figure IV-31 presents the GSPN corresponding to the name-matching dependency enriched to take into account the architecture configurations. We apply the rule for architecture configurations to the name-matching dependency, whose transformation to GSPN has been presented in Figure IV-6. The operational modes of the system and of *Component1* have

already been transformed into places. *Component11* is active when the places corresponding to modes *m1* and *m11* contain tokens. *Component12* and the connection from *Component11* to *Component12* are active when there is a token in place *m1*. Thus, places representing the fact that they are active are not necessary, as they are always active when *Component11* is active.

The rule for emptying the places *CompOrConn\_active* is the same as the one presented in Appendix B for emptying GSPN propagation places. We do not represent here the two immediate GSPN transitions that empty the place *Component11\_active*. They have inhibitor arcs respectively from places *In\_src* and *m11* and from *In\_src* and *m1*.

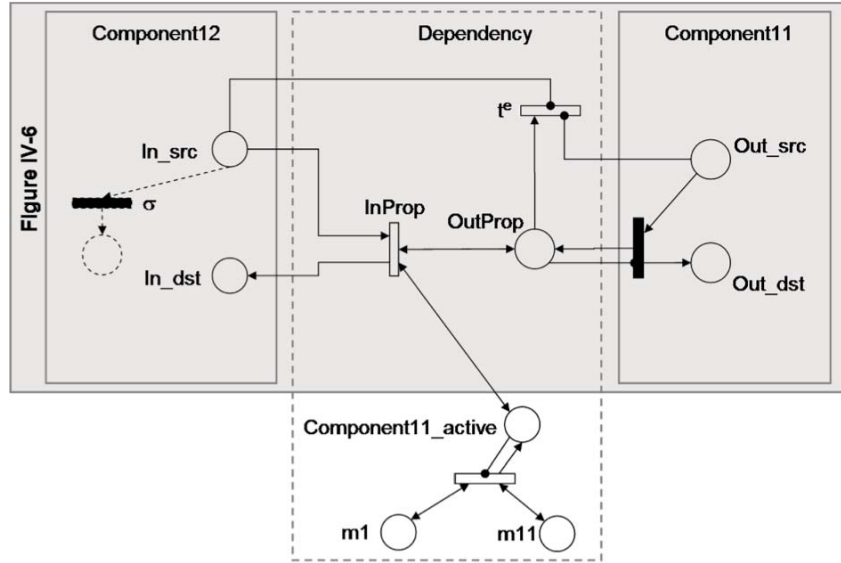


Figure IV-31. Modal configuration - Transformation rule

## IV.7.2 Rule formalization

The modes of interest for the dependability analysis have already been transformed into Petri net places. We formalize the transformation rule for a component or connection *CurrentCompOrConn* active in a specified architecture configuration without considering the GSPN transitions that empty places corresponding to the fact that a component or a connection is active.

Let us define the set of places  $P^{CompOrConn\_active}$  (places corresponding to components and connections being active) and the set of transitions  $T^{incoming}$  (corresponding to **in** propagations into *CompOrConn* or to **Guard\_Transition** properties defined in *CompOrConn*).

$$P^{CompOrConn\_active} = \{CompOrConn\_active \in \text{dependency net}^{34}\}$$

$$T^{incoming} = \{t_j^{incoming} \in InProp\_id \vee t_j^{incoming} \in T^{in} \vee t_j^{incoming} \in guard\_tr\}$$

The necessary arcs are as follows.

$$\rightarrow t_j^{incoming} = \{CurrentCompOrConn\_active \in P^{CompOrConn\_active}\} \cup \{path\_active \in P^{CompOrConn\_active}\} \cup \{sender\_active \in P^{CompOrConn\_active}\}$$

<sup>34</sup> These places are part of dependency nets, as they are only necessary if the components and connections represented as active are not isolated.

$$\begin{aligned} \leftarrow t_j^{incoming} &= \{ \text{CurrentCompOrConn}_{active} \in P^{CompOrConn\_active} \} \cup \{ \text{path}_{active} \in P^{CompOrConn\_active} \} \cup \\ &\quad \{ \text{sender}_{active} \in P^{CompOrConn\_active} \} \\ o_{t_j}^{incoming} &= \emptyset \end{aligned}$$

Let us define the sets of arcs connected to all GSPN transitions of the set  $T^{incoming}$  as follows.

$$A_{T^{incoming}} = \bigcup A_{t_j^{incoming}}$$

The GSPN subnet  $PN_{config}$  describing the architecture configuration under which the component *CurrentCompOrConn* is active is defined as follows.

$$PN_{config} = P^{CompOrConn\_active} \cup T^{incoming} \cup A_{T^{incoming}}$$

## IV.8 Scalability analysis

Naturally, when transforming large architectural models formed of many components, the size of the corresponding GSPN increases. This may lead to problems related to the processing of the GSPN and its underlying Markov chain. The state space size depends on the number of components and on the dependencies between them. In most of the cases, the more loosely coupled components are, the larger the state space gets. Intuitively, for independent components, the reachability set includes all combinations of states of all components (i.e., their Cartesian product). In general, the existence of strong dependencies in the AADL dependability model causes the removal of some of these combinations from the reachability set.

We have analyzed the evolution of the state space (both the reachability set and the Markov chain) for the case of components declaring name-matching propagations. Figure IV-32-a represents the variation of the state space size as a function of the number  $n$  of AADL transitions triggered by the same **out** propagation considering one receiver component ( $r=1$ ) that declares one AADL transition triggered by the name-matching **in** propagation ( $m=1$ ). Figure IV-32-b represents the variation of the state space size as a function of the number  $m$  of AADL transitions triggered by an **in** propagation name-matching an **out** propagation declared in one sender component ( $r=1$ ). The out propagation triggers one AADL transition in the sender component.

We observe that the state space increases linearly with the growth of the number of AADL transitions triggered by the **out** propagation in the sender component. On the other hand, it decreases linearly (but more slowly) with the growth of the number of AADL transitions triggered by the **in** propagation in the receiver component. The reason for this is that increasing the number of AADL transitions triggered by the **in** propagation in the receiver component means enforcing the coupling between the state machine of the sender component and the one of the receiver component.



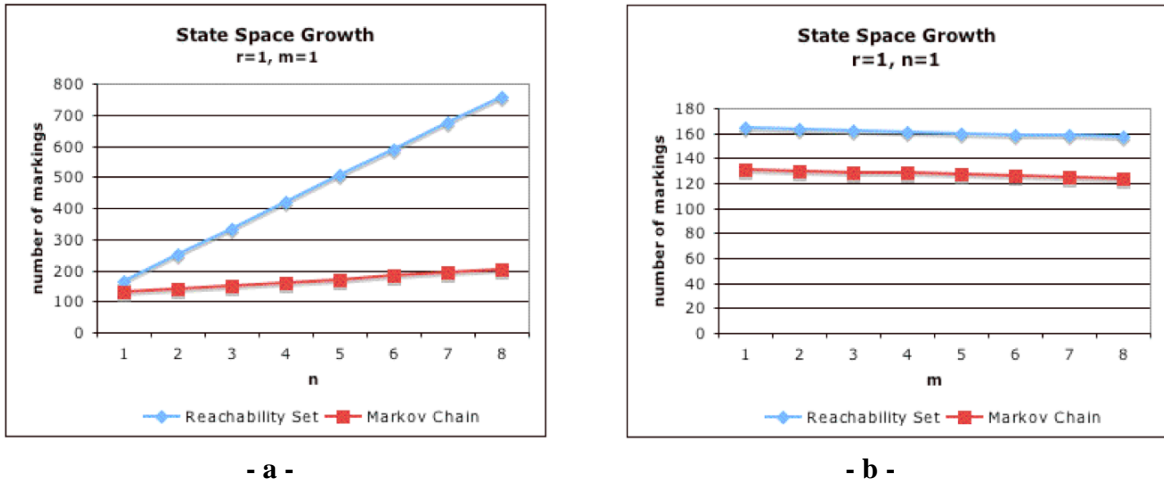


Figure IV-32. State space analysis

In the case of several identical receiver components ( $r$  is variable) for one sender component, the state space grows exponentially. This is the main limitation of this model transformation approach and represents a future research direction. This identified limitation is directly related to the fact that, in AADL, all components are represented separately (even if they have identical error models and are in identical topologies) and thus, the GSPN obtained by direct model transformation is not compact. Currently, to address these problems, GSPN reduction methods categorized as largeness tolerance techniques, such as those mentioned in section I.2.3, may be efficiently used before computing the marking graph.

## IV.9 Conclusion

In this Chapter, we have defined transformation rules for all AADL Error Model Annex constructs presented in Section II.1.2. This set of transformation rules is necessary and sufficient for obtaining GSPNs modeling complex systems formed of many components with several types of dependencies between them, as discussed in Section I.4.

The transformation rules have been defined to facilitate their implementation and thus to favor the transformation automation. In order to show the feasibility of the automation, we have implemented the model transformation tool ADAPT (*from AADL Architectural models to stochastic Petri nets through model Transformation*). It is built as a set of plug-ins, developed in the Java programming language, on top of the open-source Eclipse platform and on top of the OSATE plug-ins supporting the AADL and AADL Error Model Annex. The output of the tool is a GSPN represented in two forms: XML and tool-specific complying with the input format of the dependability evaluation tool Surf-2, developed at LAAS-CNRS. The model transformation tool is further detailed in Appendix C.

At the end of this Chapter, we have analyzed the evolution of the size of the state space underlying the GSPN when a varying number of dependencies are modeled. We have identified the limitations of our approach, which represent future research directions.

---

## V Case Study: Subsystem of the Air Traffic Control System

---

In this Chapter, we show how our modeling approach is used to compare the availability of candidate architectures of a subsystem of the French Air Traffic Control System (CAUTRA). The subsystem is local to the French Area Control Centers and has been previously studied in [Kanoun *et al.* 1999] using only GSPNs. Here we focus on modeling this system in AADL and on transforming the AADL dependability model into GSPN. The AADL dependability model of the system is built by the user according to the iterative approach described in Section I.4. The dependency block diagram is built based on the high level AADL architectural model and on the assumptions related to maintenance and fault-tolerance policies. The components are first modeled as if they were independent, following the guidelines of Section III.1. Dependencies are integrated iteratively in the AADL dependability model, following the guidelines of Section III.2. The fault-tolerance patterns presented in Section III.3 are instantiated and customized to describe a particular fault-tolerance policy for the system. The AADL dependability model is transformed into GSPN at the end of each iteration. The model transformation is transparent to the user, as it is entirely performed by a tool implementing the rules of Chapter IV. The final GSPN has been analyzed with the dependability evaluation tool Surf-2 [Béounes *et al.* 1993] in order to compare the dependability (e.g., availability, reliability) of candidate architectures that differ through the mapping of software on hardware, or through their maintenance and fault-tolerance policies.

This Chapter is structured as follows. Section V.1 gives an overview of the subsystem of CAUTRA to be analyzed. Section V.2 is dedicated to the AADL modeling and to the AADL to GSPN transformation iterations. In particular, we show how the GSPN of the system is enriched by the automated transformation at the end of each iteration. Section V.3 gives two examples of analyses based on the evaluation of the unavailability of candidate architectures. Section V.4 concludes this Chapter.

### V.1 System description

We consider a subsystem formed of two pairs of software components and two hardware platforms. Each of the two pairs of software components forms a fault-tolerant software unit. One of them is in charge of processing flight plans (FP) and the other one is in charge of processing radar data (RD). FP and RD exchange information and are connected through dedicated local networks. They run on a bi-processor hardware architecture.

RD is a real-time software system that processes radar data enriched with information concerning flight plans, received from FP. Its goal is to provide a map of the airspace to the air traffic controllers. FP is a real-time software system that processes flight plans and provides all the necessary information regarding flights to air traffic controllers. In particular, it updates flight plans according to input given by air traffic controllers. RD also provides FP with information related to aircrafts' position, speed and altitude. Based on this information, FP updates aircrafts' flight plans while they fly in the controlled airspace. Also, the map

provided by RD to air traffic controllers contains information regarding the aircrafts detected by radar. RD receives this information from FP. A broken connection from FP to RD causes the failure of RD.

RD and FP are two fault tolerant distributed software units. Each of them is formed of two replicas: one replica provides the service while the other one acts as a backup.

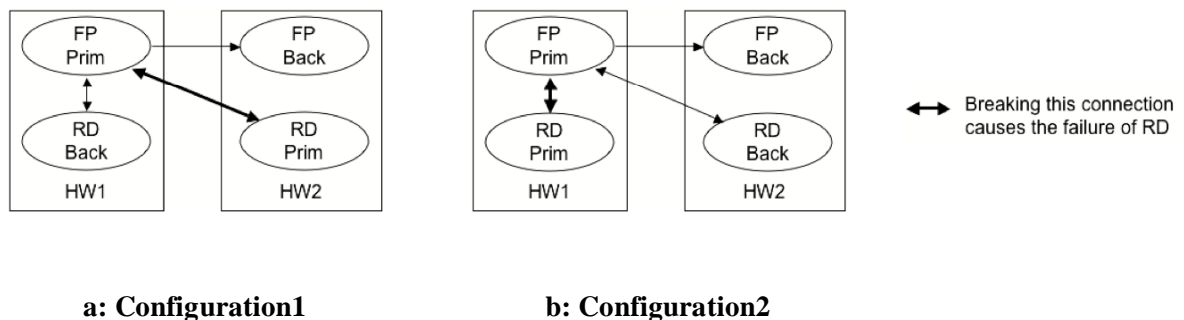
For FP, we investigate two fault-tolerance policies:

- 1) When the backup replica detects the failure of the primary replica, it becomes the primary. The two replicas keep their roles after the restoration of the failed replica.
- 2) When the backup replica detects the failure of the primary replica, it becomes the primary. The two replicas switch roles after the restoration of the failed replica.

For RD, we consider the first policy that we considered for FP. The error detection mechanisms (e.g., self-checking replicas or separate monitor) are not set at this stage.

The two replicas of one software unit are bound to separate hardware units (so as not to lose both replicas if one processor fails) and they are implemented differently, thus common mode failures are assumed very unlikely, and thus ignored. The primary FP replica sends data to the backup FP replica while the RD replicas do not communicate one with the other.

In this Chapter, we consider two candidate architectures of the CAUTRA subsystem, referred to as *Configuration1* and *Configuration2*. They are based on the same software and hardware components, but differ by the mapping between the software and the hardware components. In *Configuration1*, the initially primary replicas of FP and RD run on separate hardware units, as shown in Figure V-1-a, while in *Configuration2* they run on the same hardware unit, as shown in Figure V-1-b.



**Figure V-1.** Candidate architectures

Subsection V.1.1 presents the high level AADL architectural models obtained from the above specifications. Subsection V.1.2 is dedicated to the dependency analysis based on the high level AADL architectural models.

### V.1.1 AADL architectural models

The high level AADL architectural models of *Configuration1* and *Configuration2* are presented respectively in Figure V-2 and Figure V-3. In both configurations, each of the two software units (FP and RD) is formed of two replicas modeled as processes: *FP\_Comp1*, *FP\_Comp2* and *RD\_Comp1*, *RD\_Comp2*. *FP\_Comp1* and *FP\_Comp2* are connected through data connections used for sending data from the primary process to the backup process. FP and RD communicate through event data connections. Both candidate architectures use two hardware units modeled as processors: *Processor1* and *Processor2*.

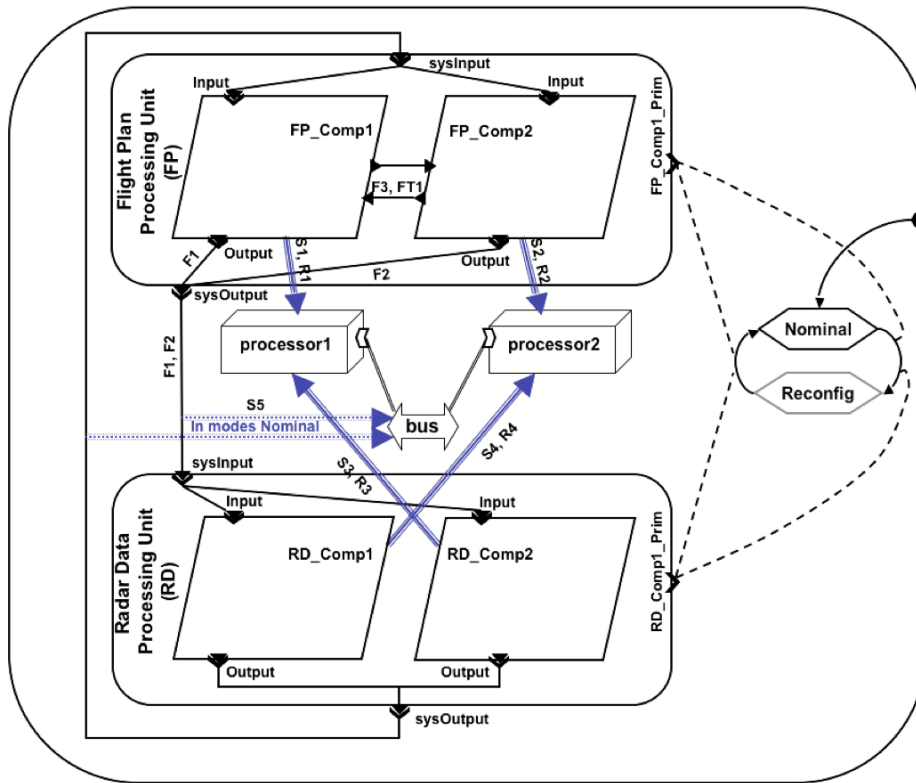


Figure V-2. AADL architectural model of Configuration1

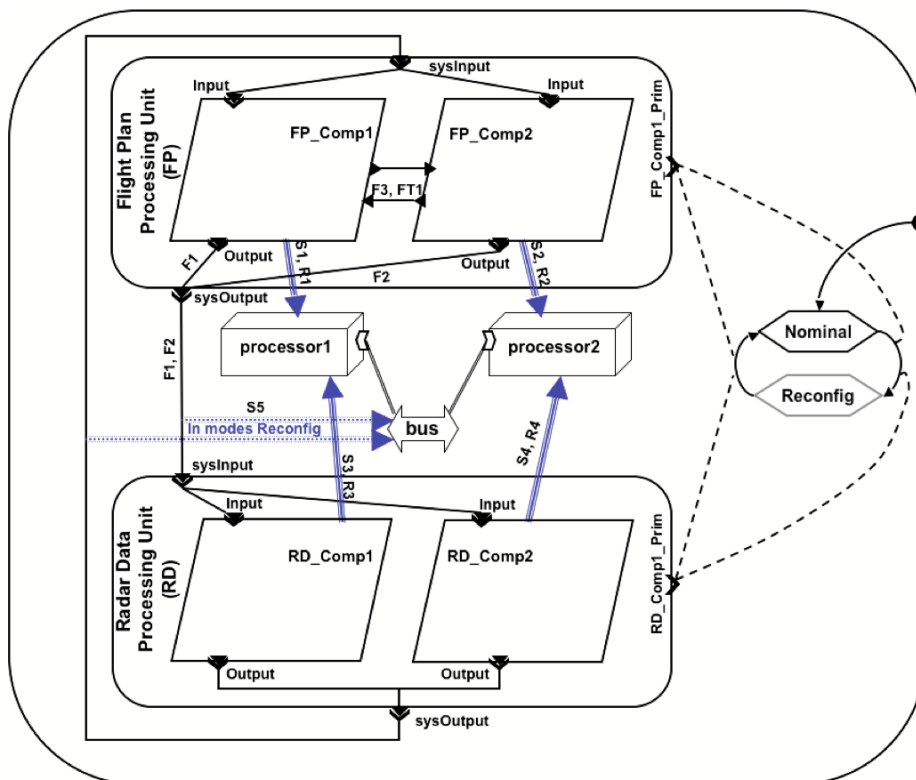


Figure V-3. AADL architectural model of Configuration2

The main difference between the two models lies in the processes' bindings to processors. In *Configuration1*, *FP\_Comp1* and *RD\_Comp1* (the initially primary replicas of FP and RD) are bound to separate processors: *FP\_Comp1* is bound to *Processor1* and *RD\_Comp1* to *Processor2*. In *Configuration2*, *FP\_Comp1* and *RD\_Comp1* are both bound to *Processor1*. In both configurations, the subsystem has two operational modes, *Nominal* and *Reconfig*, with different meanings according to the configuration. In *Configuration1*, the *Nominal* mode is the one in which the primary replicas of FP and RD are bound to separate processors while they are bound to the same processor in *Configuration2*. When the primary replicas are bound to separate processors, the connections between FP and RD are bound to a bus. Thus, the connection bindings to the bus depend on the operational mode of the subsystem. They are active in mode *Nominal* for *Configuration1* and in mode *Reconfig* for *Configuration2*.

The high level AADL architectural models presented in this Section will be further refined when detailing fault-tolerance and maintenance policies.

### V.1.2 Dependency analysis

The various interactions between this subsystem's components induce architectural dependencies between them:

- Structural dependency between each processor and the processes that run on top of it. We assume that hardware faults can propagate and influence the software running on top of it. These dependencies (S1, S2, S3 and S4 in Figure V-2 and Figure V-3) result from the architectural bindings of processes to processors.
- Recovery dependency between each processor and the processes that run on top of it. If a process fails, it cannot be restarted if the processor on top of which it runs is in a failed state. These dependencies (R1, R2, R3 and R4 in Figure V-2 and Figure V-3) are supported by the architectural bindings of processes to processors.
- Structural dependency between the bus and the processes of RD. If the bus fails, the broken connections bound to it make RD fail in mode *Nominal* of *Configuration1* and in mode *Reconfig* of *Configuration2*. This dependency (S5 in Figure V-2 and Figure V-3) is supported by the binding of the connection from FP to RD to the bus.
- Functional dependencies between FP and RD. The primary FP process may propagate errors to both RD processes. These dependencies (F1 and F2 in Figure V-2 and Figure V-3) are supported by the connections of the FP replicas to the RD replicas. Note that we consider that RD errors do not propagate to FP even though there is a connection from RD to FP.
- Functional dependency between the primary FP process and the backup FP process. This dependency (F3 in Figure V-2 and Figure V-3) results from the connections between the FP replicas. We consider that only the primary process can propagate errors to the backup. Thus, the architectural models of Figure V-2 and Figure V-3 must be refined when integrating this dependency, to disable the connection from the backup to the primary replica.
- Fault-tolerance dependency between the two FP processes and the two RD processes. If the replica that delivers the service fails but the other one is error free, the two software replicas switch roles. Then, the failed replica is restarted. We investigate two policies:
  - FT1, which assumes that the two replicas keep their roles after the restoration of the failed replica,
  - FT1', which assumes that they switch roles after the restoration of the failed replica.

The fault-tolerance dependency between the FP replicas (FT1/FT1' in Figure V-2 and Figure V-3) is partially supported by the connections between *FP\_Comp1* and *FP\_Comp2*. We assume there is a fault-tolerance dependency between the RD replicas, named FT2 further on. This dependency is not completely supported by the initial architectural models of Figure V-2 and Figure V-3. Thus, FT2 requires a refinement of these models.

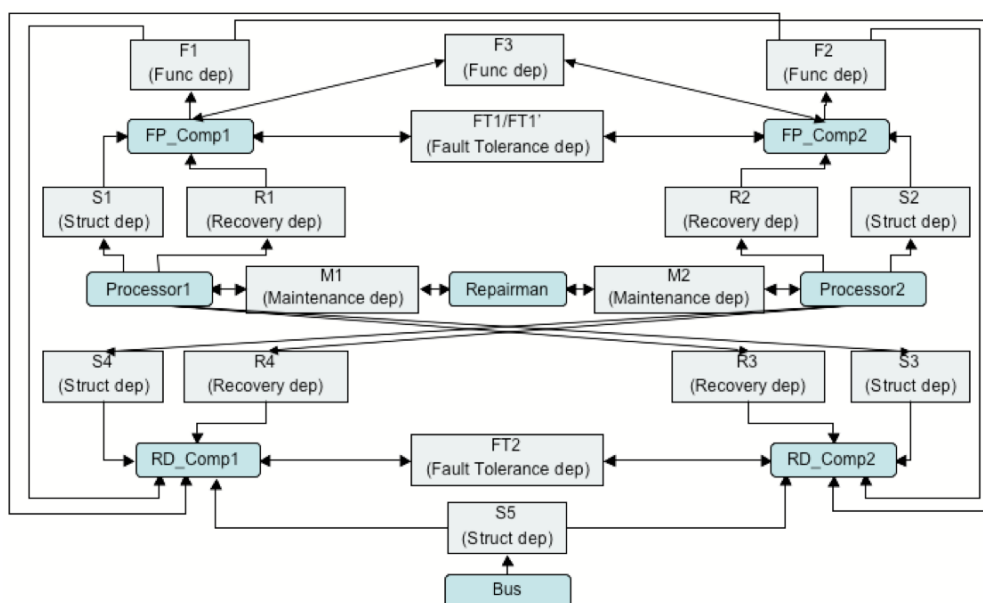
In addition to the architectural dependencies, we take into account:

- Maintenance dependency between the two processors that share a repairman that is not simultaneously available for the two components. This maintenance dependency is not visible on the architectural models of Figure V-2 and Figure V-3.

Based on the above analysis, we build the dependency block diagrams of the two candidate architectures. They provide a global view of the system components and interactions that are taken into account for the dependability analysis, as stated in Section I.4.2.

Figure V-4 shows the dependency block diagram describing the dependencies between the components in *Configuration1*. The block *Repairman* does not correspond to a component of the AADL architectural models of Figure V-2 and Figure V-3. It represents a component to be added during the refinement of these models, in order to take into account the maintenance dependency between the processors. This diagram is formed of eight component blocks (one for each of the four software components, one for each of the two hardware components, one for the bus and one for the repairman) and sixteen dependency blocks. Among the dependency blocks, some correspond to identical AADL and GSPN models. For example, S1-4 are identical, since they represent the same dependency specification between identical components. In the following Sections, we point out the dependencies modeled simultaneously because of this reason.

The dependency block diagram for *Configuration2* is similar, except for the links between the processor blocks and the blocks of structural and recovery dependencies. Concretely, in *Configuration2*, *Processor1* is linked to *RD\_Comp1* via blocks S4 and R4 and *Processor2* is linked to *RD\_Comp2* via blocks S3 and R3.



**Figure V-4.** Dependency Block Diagram of *Configuration1*

In this Chapter, the purpose is to compare, with respect to their availability, four alternatives resulting from the two architecture candidates *Configuration1* and *Configuration2* and from the two fault-tolerance policies FT1 and FT1’.

## V.2 AADL dependability model and transformation to GSPN

This section presents the AADL dependability model construction iterations together with the corresponding transformation iterations. Based on the system description and on the dependency block diagrams, we decide to associate error models with hardware (processors and bus) and software components (processes). We integrate successively in these error models the dependencies, in the order of their presentation in Section V.1.2: first those that do not require architectural refinements, in the end, those requiring architectural refinements: F3, FT1, FT2, M1 and M2 of Figure V-4. For the fault-tolerance dependencies, we use the patterns presented in Section III.3. We do not take into account faults in the connections. We declare one error model implementation for each iteration.

### V.2.1 Iteration 1: independent components

We start the construction of AADL error models as if the components with which they are associated were independent. No AADL propagations are defined at this stage: we model the behaviour of the processors and the bus, then the behaviour of the processes in the presence of their own faults and repair events.

#### V.2.1.1 Assumptions

We consider the following behavior for the processors:

- Initially the component is in *Error\_Free* state.
- Temporary faults are activated with rate  $\lambda_{h1}$ .
- Permanent faults are activated with rate  $\lambda_{h2}$ .
- Errors caused by temporary faults disappear with a rate  $\mu_{h1}$ .
- Errors caused by permanent faults are either detected (detection rate  $\tau_{h1}$ ), or non-detected (non detection rate  $\tau_{h2}$ ). In both cases the component moves to a *Failed* state. If the error has been detected, the hardware component is repaired. If not, the failure is perceived with a rate  $\xi_h$ . Then the component is repaired with a repair rate  $\mu_{h2}$ .

We consider the following behavior for the bus:

- Initially the component is in *Error\_Free* state.
- Faults are activated with a rate  $\lambda_b$ .
- The bus is repaired after a failure with a repair rate  $\mu_b$ .

We consider the following behavior for the processes:

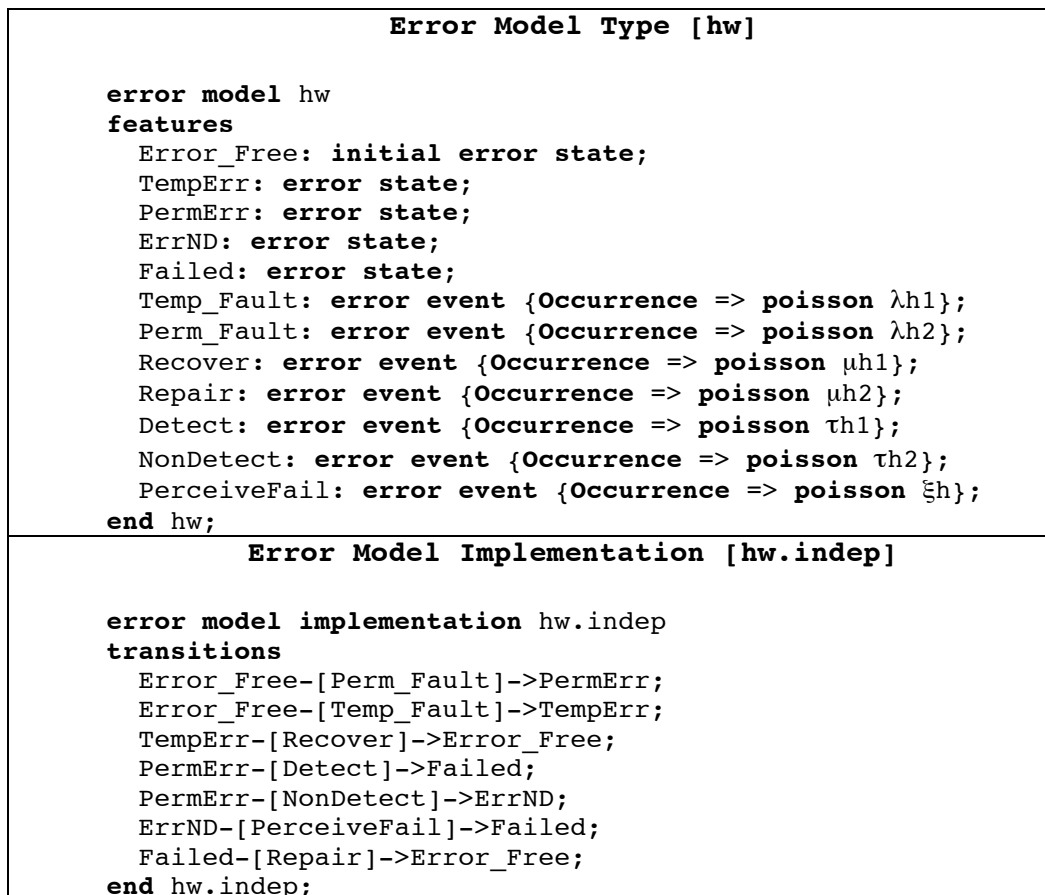
- Initially the component is in *Error\_Free* state.
- Faults are activated with a rate  $\lambda_s$ .
- An error is either detected (detection rate  $\tau_{s1}$ ), or not detected (non detection rate  $\tau_{s2}$ ).

- The effects of a detected error caused by the activation of a temporary fault are eliminated by the error detection mechanisms with a rate  $\pi_{s1}$ . As a consequence, the component moves to the *Error\_Free* state. A detected error caused by a permanent fault causes the failure of the process with a rate  $\pi_{s2}$ . As a consequence, the process needs to be restarted to eliminate the effects of the error. The restart rate is  $\nu_s$ .
- Effects of a non-detected error may disappear (rate  $\delta_s$ ) or may be perceived (rate  $\xi_s$ ).

Note that the difference between the processors' and processes' behaviours is that for the processors, temporary and permanent faults are distinguished by their respective consequences, following their activation, whereas for the processes they are distinguished after specific processing.

### V.2.1.2 AADL dependability model

Figure V-5, Figure V-7 and Figure V-6 show respectively the error models *hw.indep*, *bus.indep* and *sw.indep* describing the behaviors of an independent processor, bus and process as resulting from their own failures and repairs.



**Figure V-5.** Error model for independent processor



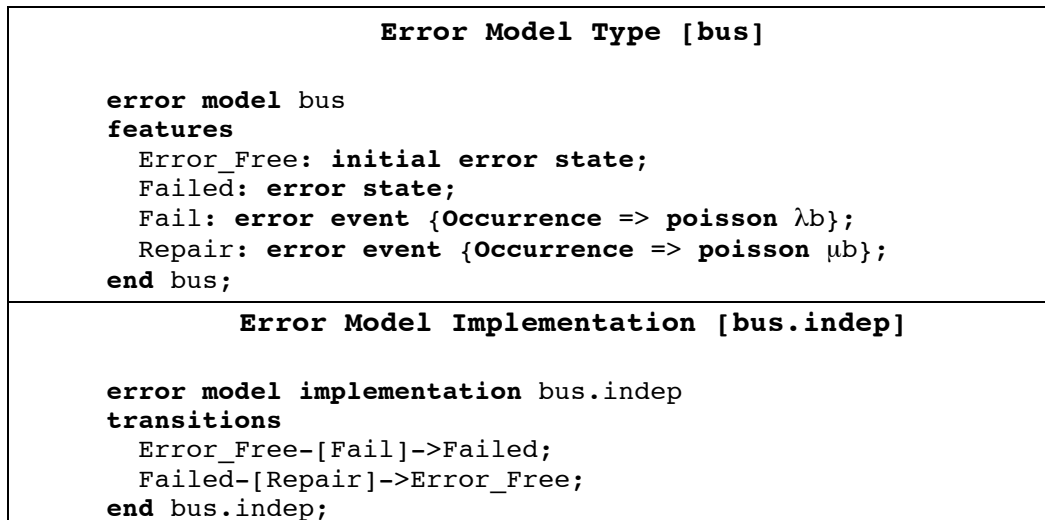


Figure V-6. Error model for independent bus

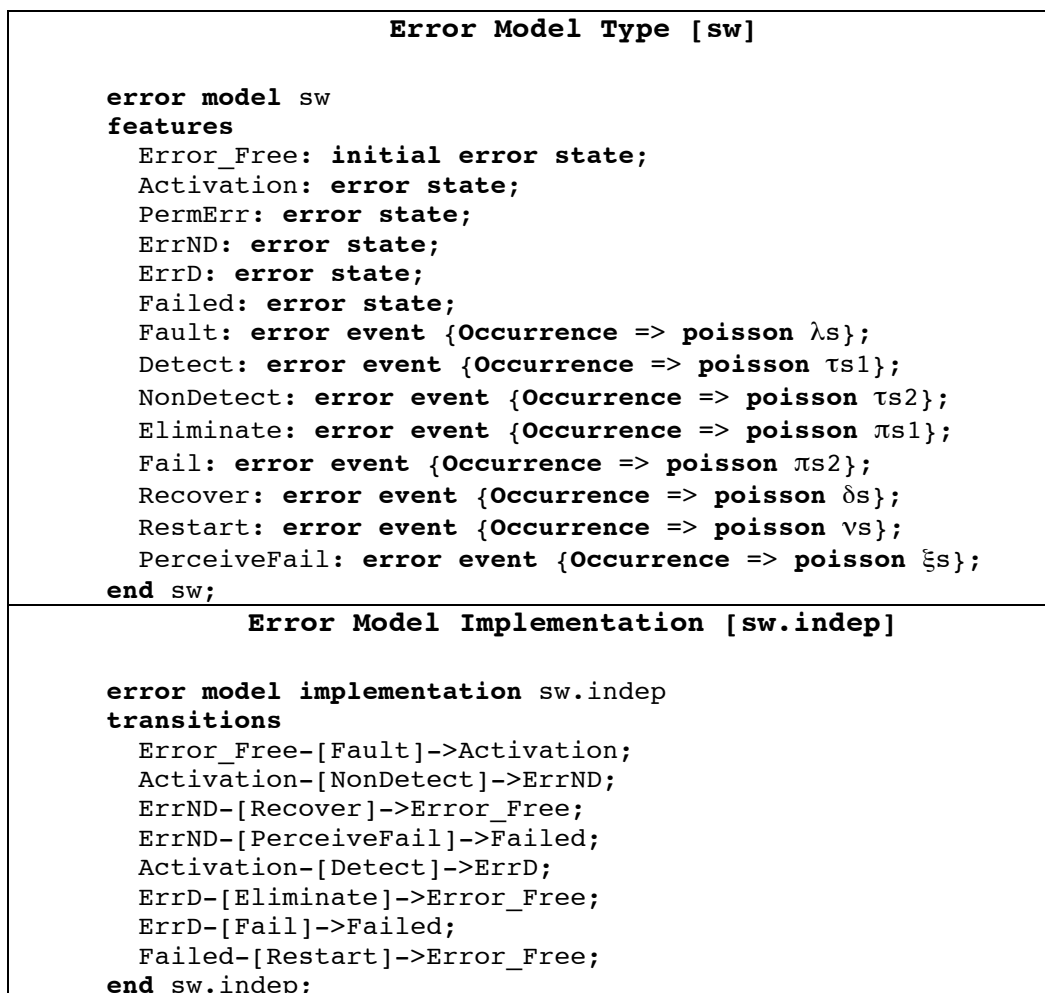


Figure V-7. Error model for independent process

The above error models are associated respectively with the AADL processor implementation *processor.cautra*, process *process.cautra* and bus *bus.cautra*. As an example, we show in Figure V-8 the association of the error model *hw.indep* with *processor.cautra*. *Processor1* and *Processor2* are two instances of *processor.cautra*.

```

system implementation processor.cautra
[...]
annex Error_Model {**
    Model => hw.indep;
**};
end processor.cautra;
    
```

Figure V-8. Textual AADL dependability model - processor

### V.2.1.3 AADL to GSPN model transformation

Figure V-9, Figure V-11 and Figure V-10 show respectively the GSPNs obtained for an independent processor, bus and process, by applying the model transformation rule for independent components (see Section IV.2).

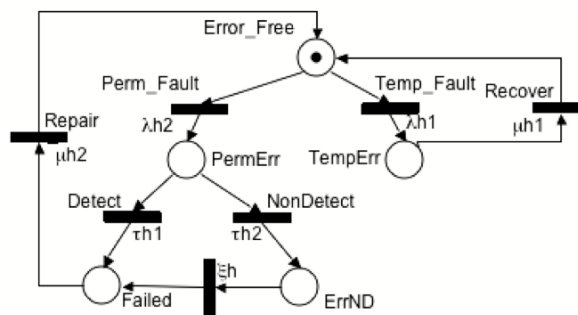


Figure V-9. GSPN modeling an independent processor

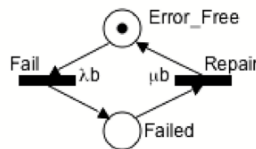


Figure V-10. GSPN modeling an independent bus

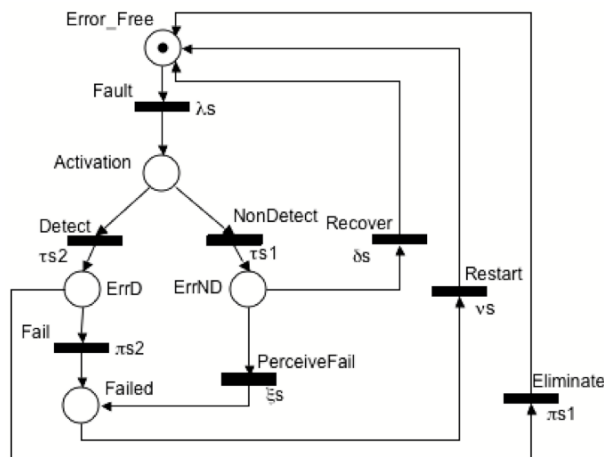


Figure V-11. GSPN modeling an independent process

## V.2.2 Iteration 2: structural dependency from processor to process

### V.2.2.1 Assumptions

Because of the fact that processes run on processors, errors in a processor may impact the processes running on top of it, as follows:

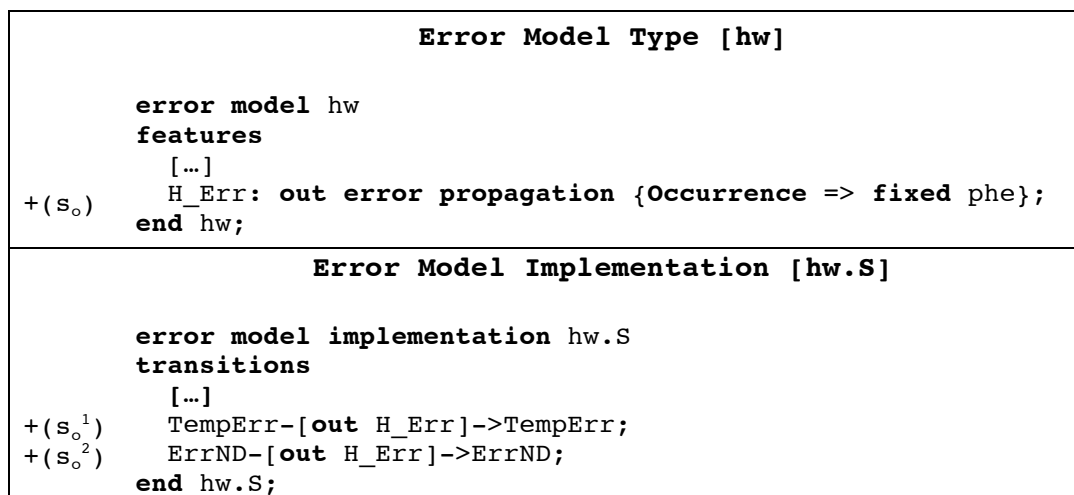
- A temporary fault in the hardware may cause error transmissions to the processes.
- A non detected permanent fault may cause error transmissions to the processes.

The error propagation from the processor to the processes occurs with a probability,  $p_{he}$ .

If the processes are *Error\_Free*, the error transmission from the processor leads them in the state where a fault has been activated. Then, the error is processed in the same way as internal ones. We consider that internal software errors followed by hardware error transmissions are very unlikely to happen, thus we neglect them.

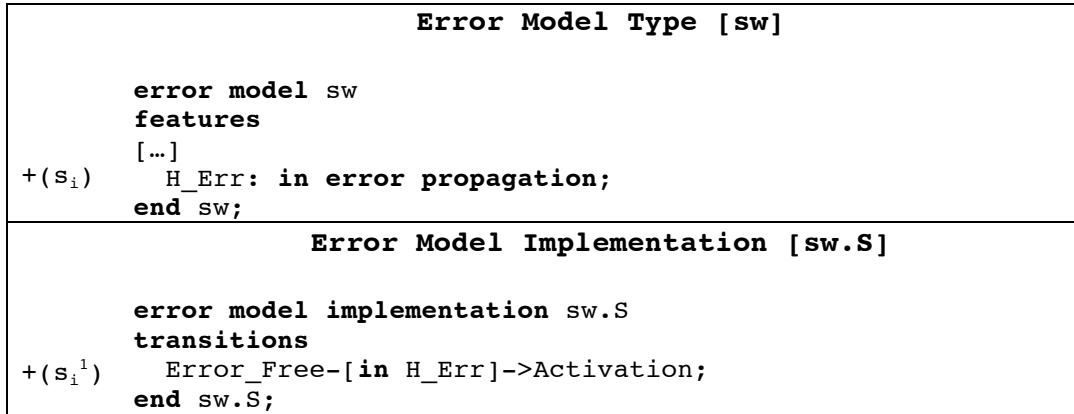
### V.2.2.2 AADL dependability model

Figure V-12 only shows what is added to the error model for an independent processor (Figure V-5) in order to describe the structural dependency. The error model type for processors, *hw*, is completed with line  $s_o$  in order to include an **out** error propagation declaration *H\_Err*. The error model implementation *hw.S* takes into account the sender side of the structural dependency: it declares two transitions triggered by the newly introduced **out** propagation (see lines  $s_o^1$  and  $s_o^2$  of Figure V-12). Note that the destination state is the same as the source state for the two transitions. Thus, if the **out** propagation occurs, it remains visible until the processor leaves this state.



**Figure V-12.** Error model for structural dependency (sender side)

Figure V-13 only shows what is added to the error model for an independent process (Figure V-7) in order to describe the structural dependency. The error model type *sw* is completed with line  $s_i$  in order to include an **in** propagation declaration *H\_Err*. Its name matches the name of the **out** propagation declared in the error model type *hw*. The error model implementation *sw.S* takes into account the recipient side of the structural dependency by declaring a transition triggered by the **in** propagation *H\_Err* (see line  $s_i^1$  of Figure V-13) and leading the process from state *Error\_Free* to *Activation*.

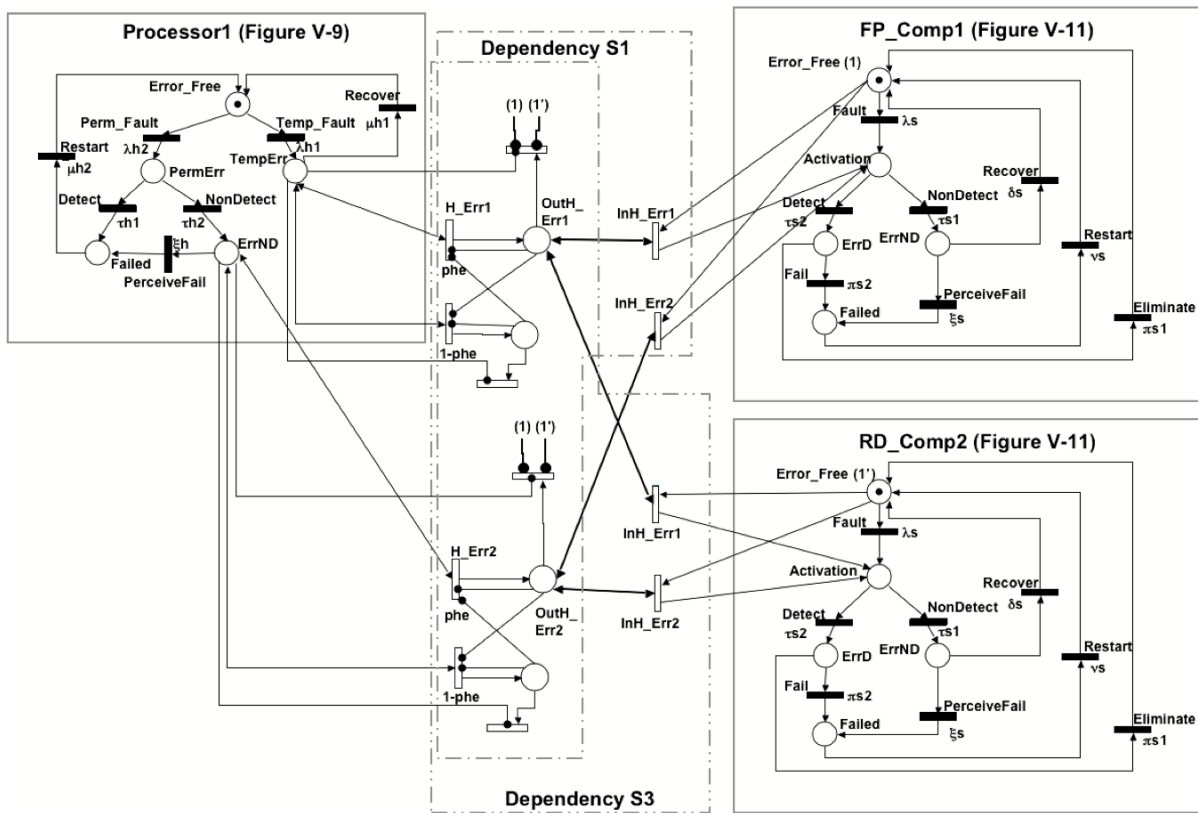


**Figure V-13.** Error model for structural dependency (recipient side)

In both our candidate architectures, each processor hosts two processes. Thus, associating the error model *sw.S* with all four processes and the error model *hw.S* with both processors, we model the four structural dependencies denoted S1-4 in Figure V-4.

### V.2.2.3 AADL to GSPN model transformation

Figure V-14 shows the GSPN obtained when transforming the AADL dependability model corresponding to two processes bound to one processor, and taking into account the structural dependency from the processor to the processes.



**Figure V-14.** GSPN modeling a processor and two processes with structural dependencies

To obtain the GSPN of Figure V-14, we apply the model transformation rules presented respectively in Section IV.2 (for independent components) and in Section IV.3.3 (for name-matching **in** – **out** propagations). The transformation is applied to *Configuration1*, in which *FP\_Comp1* and *RD\_Comp2* are bound to *Processor1*. In this same configuration, we obtain a similar GSPN for *Processor2*, *FP\_Comp2* and *RD\_Comp1*. Note that the component net named *Processor1* in Figure V-14 has already been shown in Figure V-9, while the two component nets *FP\_Comp1* and *RD\_Comp2* are identical to the one shown in Figure V-11.

## V.2.3 Iteration 3: recovery dependency from processor to process

### V.2.3.1 Assumptions

Because of the fact that processes run on processors, we assume that errors in a processor may impact the processes running on top of it, as follows:

- Detected permanent faults in a processor cause failures that require repairing the processor. During repair, the processes running on top of it are stopped.
- The recovery of the processes must be synchronized with the repair of the processor, i.e., a process may not be restarted before the processor has been repaired.

### V.2.3.2 AADL dependability model

Figure V-15 only shows what is added to the error model associated with the processor in order to describe the recovery dependency. The error model type for processors, *hw*, is completed with lines  $r_{o1}$  and  $r_{o2}$  in order to include **out** error propagation declarations *H\_FailedVisible* and *H\_OK*. Their Occurrence properties are fixed probabilities of 1, i.e., the propagations occur certainly and immediately. *H\_FailedVisible* causes the processes' failures while *H\_OK* is used to synchronize the repair of the processor with the restart of the process. The error model implementation *hw.SR* takes into account the sender side of the recovery dependency: it declares one transition triggered by each of the two newly introduced **out** propagations (see lines  $r_o^1$  and  $r_o^2$  of Figure V-15). When one of the **out** propagations occurs, the processor remains in the same state (i.e., *Failed* or *Error\_Free*) and the propagation remains visible until the processor leaves this state.

Figure V-16 shows what is added to the error model associated with a process in order to describe the recovery dependency. The error model type *sw* is completed with lines  $r_{i1}$ ,  $r_{i2}$ ,  $r_{i3}$ :

- line  $r_{i1}$  declares an additional state in which the process is allowed to restart. This state is used in the synchronization between the repair of the processor and the restart of the process.
- line  $r_{i2}$  declares an **in** propagation *H\_FailedVisible*. Its name matches the name of the **out** propagation declared in the error model type *hw* (see Figure V-15).
- line  $r_{i3}$  declares an **in** propagation *H\_OK*. Its name matches the name of the **out** propagation declared in the error model type *hw* (see Figure V-15).

The error model implementation *sw.SR* takes into account the recipient side of the recovery dependency by declaring four transitions triggered by the **in** propagation *H\_FailedVisible* (see lines  $r_i^1$ ,  $r_i^2$ ,  $r_i^3$ ,  $r_i^4$  of Figure V-13) and leading the process from each state (other than *Failed*) to the *Failed* state. Also, the process is authorized to move from the *Failed* state to *InRestart* only when it receives the *HW\_OK* propagation (see line  $r_i^5$  of Figure V-13). Line  $r_i^6$

replaces line  $r_i^7$ , as the restart procedure is now engaged from state *InRestart*, after authorization from the processor.

In both our candidate architectures, each processor hosts two processes. Thus, associating the error model *sw.SR* to all four processes and the error model *hw.SR* to both processors, we model the four recovery dependencies denoted R1-4 in Figure V-4.

| <b>Error Model Type [hw]</b>   |  |
|--|--|
| <pre> <b>error model</b> hw <b>features</b>   [...] +(r<sub>o1</sub>)   H_FailedVisible: <b>out error propagation</b> {Occurrence =&gt; fixed 1}; +(r<sub>o2</sub>)   H_OK: <b>out error propagation</b> {Occurrence =&gt; fixed 1}; <b>end</b> hw; </pre>   |  |
| <b>Error Model Implementation [hw.SR]</b>  |  |
| <pre> <b>error model implementation</b> hw.SR <b>transitions</b>   [...] +(r<sub>o<sup>1</sup></sub>)   Failed-[<b>out</b> H_FailedVisible]-&gt;Failed; +(r<sub>o<sup>2</sup></sub>)   Error_Free-[<b>out</b> H_OK]-&gt;Error_Free; <b>end</b> hw.SR; </pre> |  |

Figure V-15. Error model for recovery dependency (sender side)

| <b>Error Model Type [sw]</b>  |  |
|---|--|
| <pre> <b>error model</b> sw <b>features</b>   [...] +(r<sub>i1</sub>)   InRestart: <b>error state</b>; +(r<sub>i2</sub>)   H_FailedVisible: <b>in error propagation</b>; +(r<sub>i3</sub>)   H_OK: <b>in error propagation</b>; <b>end</b> sw; </pre>   |  |
| <b>Error Model Implementation [sw.SR]</b>   |  |
| <pre> <b>error model implementation</b> sw.SR <b>transitions</b> +(r<sub>i<sup>1</sup></sub>)   Error_Free-[<b>in</b> H_FailedVisible]-&gt;Failed; +(r<sub>i<sup>2</sup></sub>)   Activation-[<b>in</b> H_FailedVisible]-&gt;Failed; +(r<sub>i<sup>3</sup></sub>)   ErrND-[<b>in</b> H_FailedVisible]-&gt;Failed; +(r<sub>i<sup>4</sup></sub>)   ErrD-[<b>in</b> H_FailedVisible]-&gt;Failed; +(r<sub>i<sup>5</sup></sub>)   Failed-[<b>in</b> H_OK]-&gt;InRestart; +(r<sub>i<sup>6</sup></sub>)   InRestart-[Restart]-&gt;Error_Free; -(r<sub>i<sup>7</sup></sub>)   Failed-[Restart]-&gt;Error_Free; <b>end</b> sw.SR; </pre> |  |

Figure V-16. Error model for recovery dependency (recipient side)

### V.2.3.3 AADL to GSPN model transformation

Figure V-17 shows the GSPN obtained when transforming the AADL model corresponding to two processes bound to one processor, taking into account the recovery dependencies

between the processor and the processes. We apply the model transformation rules presented respectively in Section IV.2 (for independent components) and in Section IV.3.3 (for name-matching *in* – *out* propagations). As in the case of the structural dependency, we assume here that the transformation is applied to *Configuration1*, in which *FP\_Comp1* and *RD\_Comp2* are bound to *Processor1*. Note that the component net named *Processor1* in Figure V-17 has already been shown in Figure V-9. The two component nets *FP\_Comp1* and *RD\_Comp2* differ of the one of Figure V-11, as we have added the state *InRestart* in the error model associated with processes.

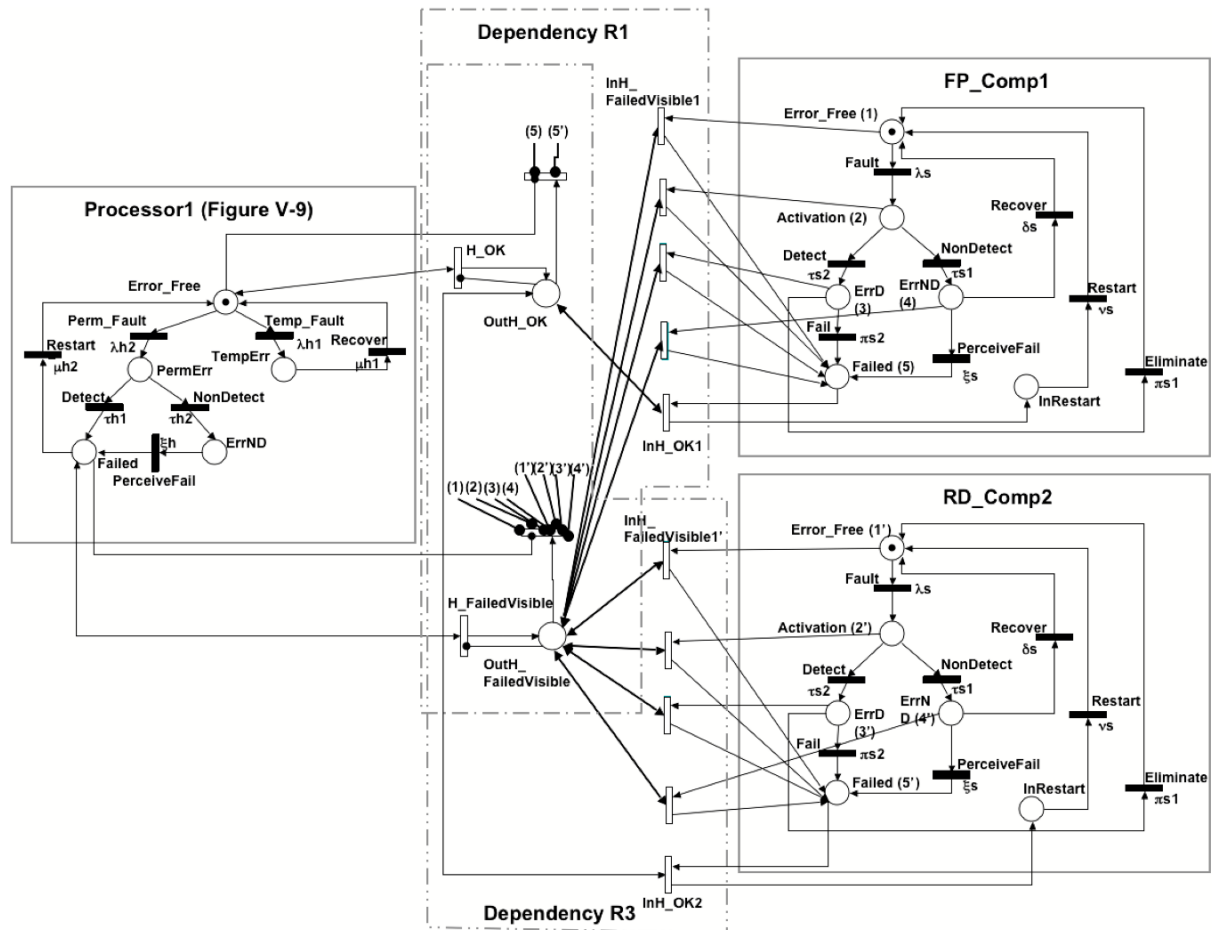


Figure V-17. GSPN modeling a processor and two processes with recovery dependencies

## V.2.4 Iteration 4: structural dependency from bus to RD processes

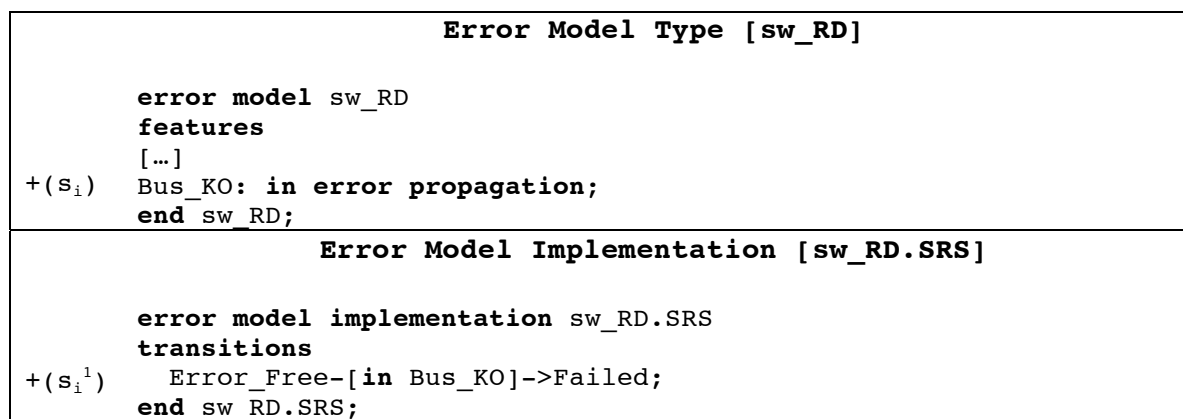
### V.2.4.1 Assumptions

According to the system description, broken connections from FP to RD cause the failure of the RD processes. We assume that this connection may be broken only if it is bound to a bus which fails:

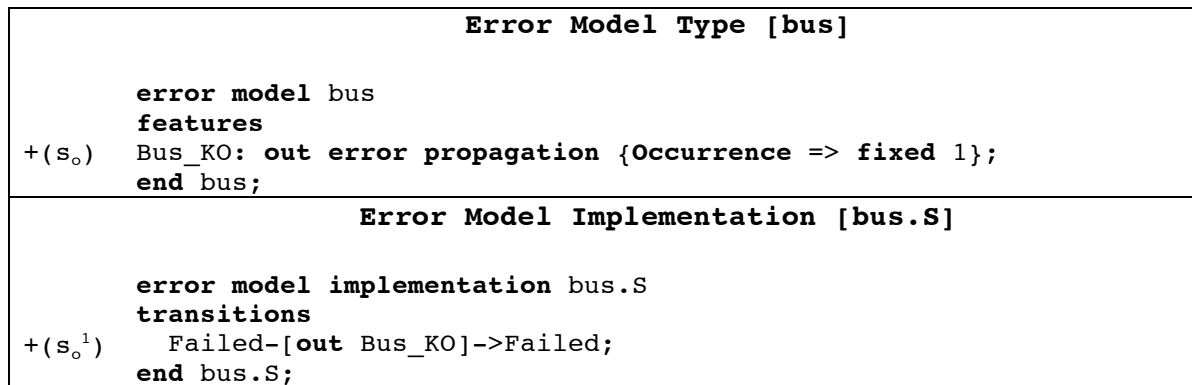
- In *Configuration1*, the failure of the bus makes the RD processes fail in mode *Nominal* (as the connection is bound to the bus in mode *Nominal*).
- In *Configuration2*, the failure of the bus makes the RD processes fail in mode *Reconfig* (as the connection is bound to the bus in mode *Reconfig*).

### V.2.4.2 AADL dependability model

When taking into account the previous dependencies, we considered that the four processes of our candidate architectures have the same behavior. Now, we consider that the RD and the FP processes behave differently when the bus fails, i.e., the RD processes fail, whereas the FP processes are not affected. Thus, we customize only the error models associated with RD processes. This is achieved by adding an **in** propagation declaration in the error model type (see line  $s_i$  of Figure V-18) and a transition triggered by it in the error model implementation (see line  $s_i^1$  of Figure V-18). We assume that it is very unlikely for a bus failure to occur immediately after fault activation in a process, thus we consider that the process is affected by a bus failure only when it is *Error\_Free*. Figure V-19 shows the lines added in the error model associated with the bus, to describe the sender side of the dependency. The **out** propagation *Bus\_KO* occurs immediately when the component arrives in state *Failed*.



**Figure V-18.** Error model for structural dependency bus – RD process (recipient side)



**Figure V-19.** Error model for structural dependency bus – RD process (sender side)

### V.2.4.3 AADL to GSPN model transformation

Figure V-20 and Figure V-21 show respectively the GSPNs obtained for *Configuration1* and *Configuration2* when transforming the AADL model corresponding to the two RD processes and to the bus, taking into account the structural dependency from the bus to the processes.



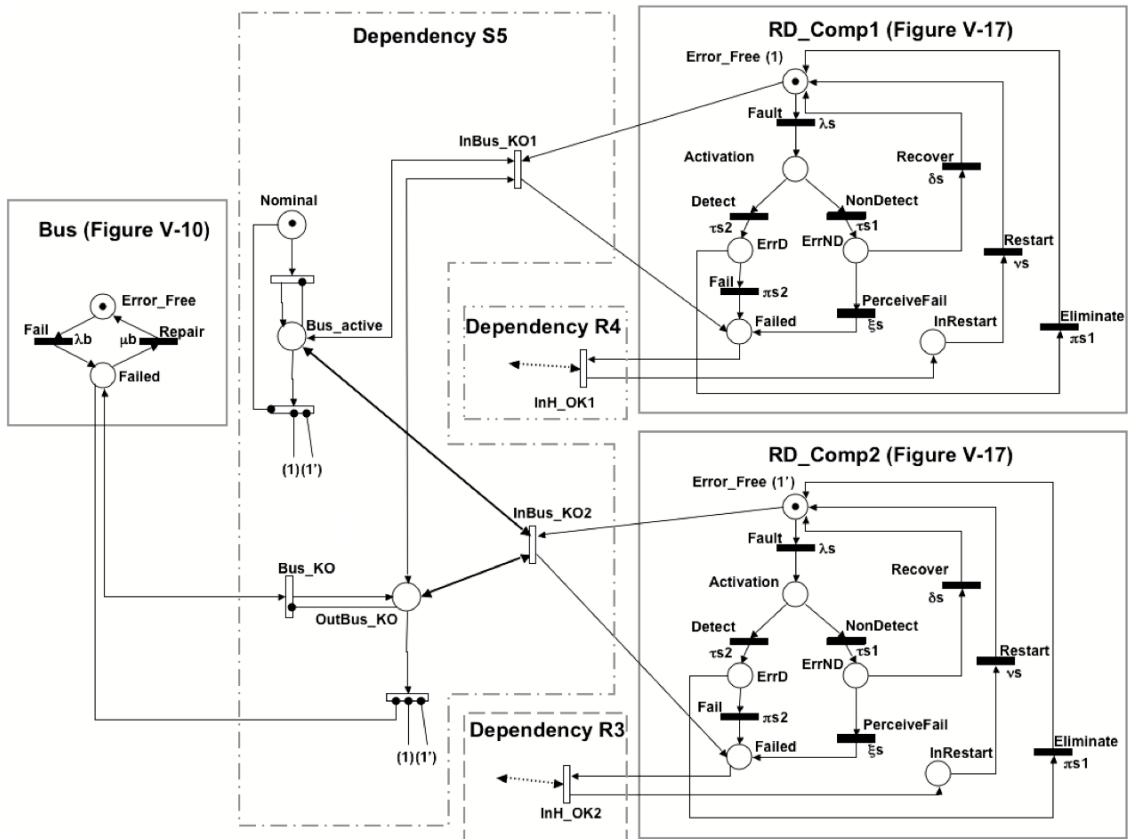


Figure V-20. GSPN of the bus and two processes with structural dependency (Configuration 1)

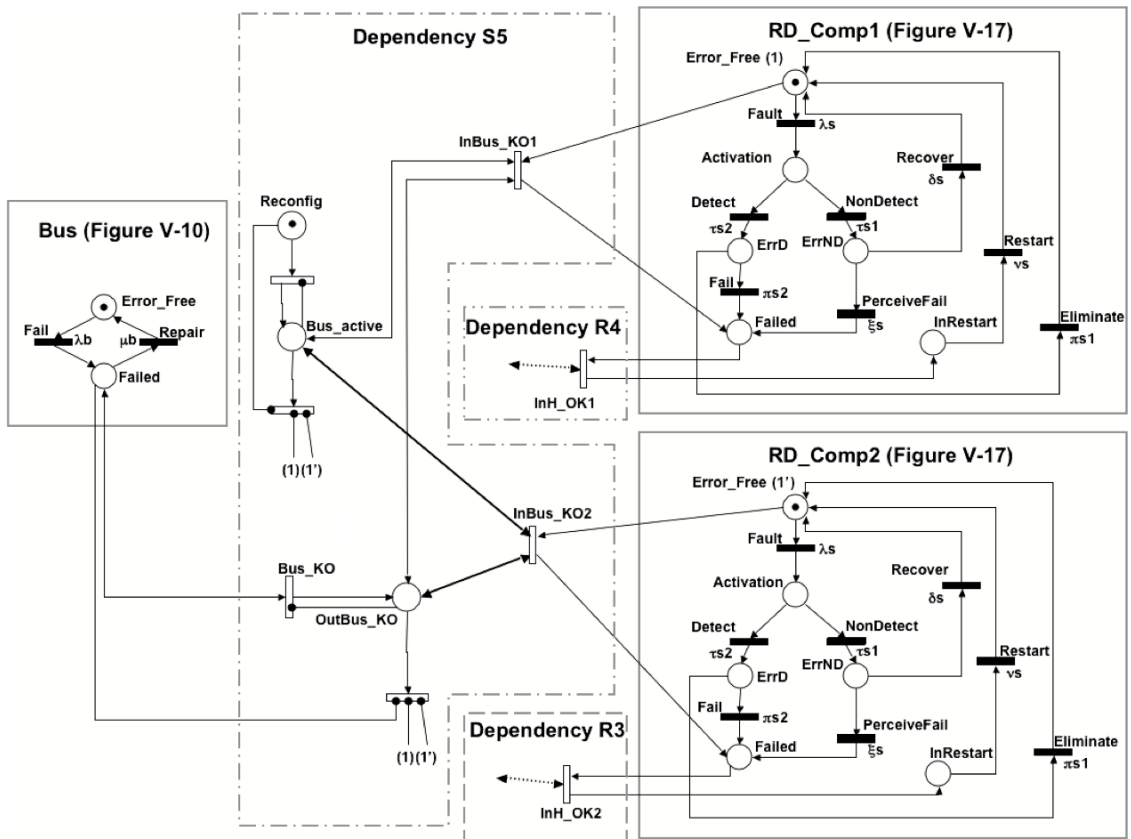


Figure V-21. GSPN of the bus and two processes with structural dependency (Configuration 2)

To obtain the GSPNs of Figure V-20 and Figure V-21, we apply the transformation rules presented in Section IV.2 (for independent components), Section IV.3.3 (for name-matching *in – out* propagations) and Section IV.7 (for architecture configurations). The GSPNs obtained for *Configuration1* and *Configuration2* are identical, except for the place that models the operational mode in which the *in* propagation *Bus\_KO* is authorized. The place *Bus\_active* has a token when the bus is active, i.e., when there is a token in the place corresponding to the mode *Nominal* for *Configuration1* (*Reconfig* for *Configuration2*). This place results from applying the rule for architecture configurations (see Section IV.7).

The component nets *RD\_Comp1* and *RD\_Comp2* in Figure V-20 and Figure V-21 are the same as those of Figure V-17. The component net *Bus* is same as the one of Figure V-10.

## V.2.5 Iteration 5: functional dependency from FP to RD processes

### V.2.5.1 Assumptions

According to the system description, the primary FP process may propagate errors to both RD processes. We assume that only non-detected errors propagate with a given probability  $p_{md}$ . Even though there is a connection from RD to FP, we assume that RD replicas do not propagate errors to FP.

### V.2.5.2 AADL dependability model

Only the primary FP process propagates errors to RD processes and each of the two FP processes may be primary or backup. Thus, the AADL architectural model needs to be refined in order to describe the fact that the connection from one of the FP processes to the port *sysOutput* of FP is active only if that process is the primary. Figure V-22 shows the refined AADL architectural model of *Configuration1* (the refinement is identical for *Configuration2*).

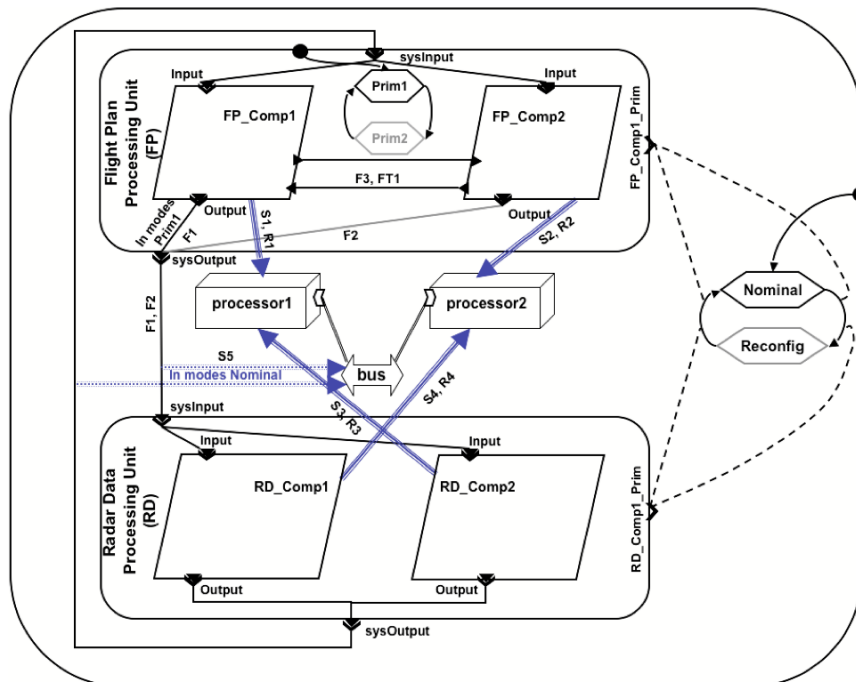
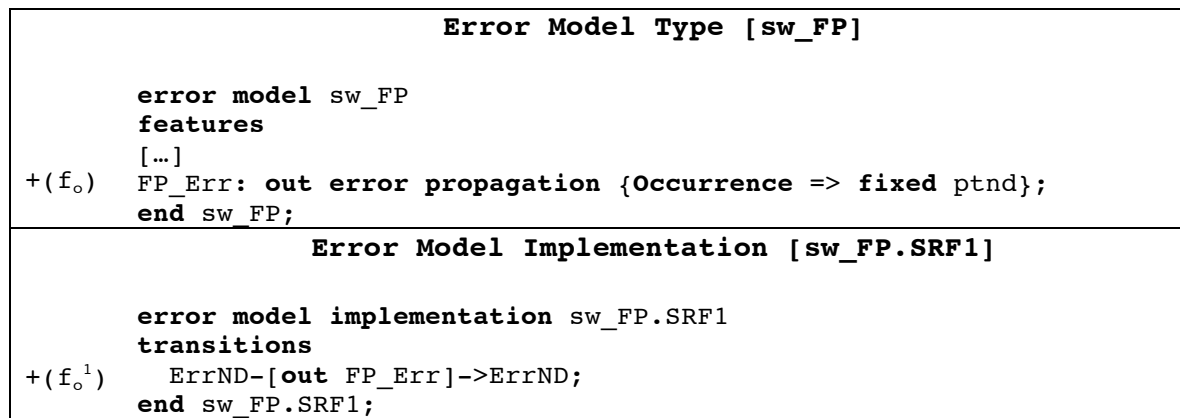


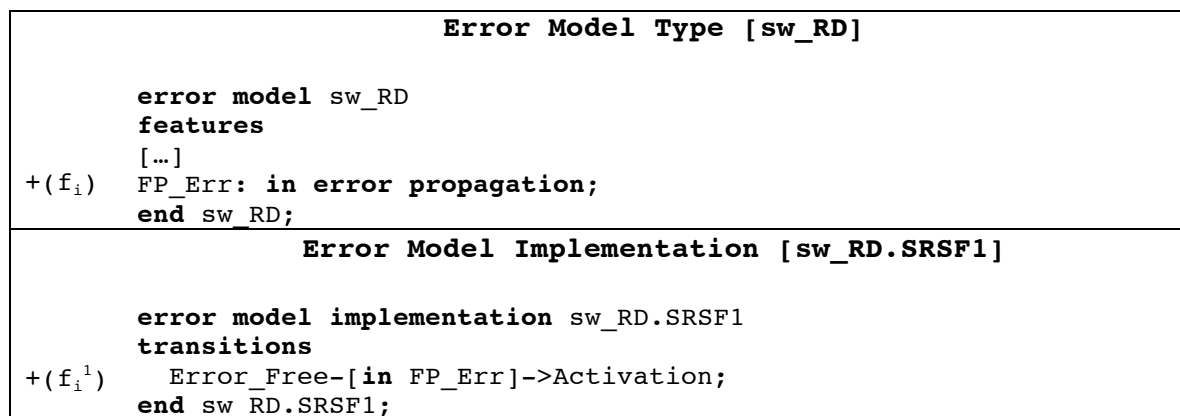
Figure V-22. AADL architectural model of Configuration1 – refined with modal connections

The refinement of Figure V-22 consists in declaring two modes in the FP subsystem (*Prim1*, in which *FP\_Comp1* is the primary and *Prim2* in which *FP\_Comp2* is the primary) and in making the connections from the port *Output* of each process to the port *sysOutput* of FP active in the mode in which the process is active. The connection from the port *Output* of *FP\_Comp1* is active in mode *Prim1*. Thus, when FP is in mode *Prim2*, *FP\_Comp1* cannot propagate errors outside FP. The conditions for switching modes are dictated by the chosen fault-tolerance policy. Thus, the mode dynamics is not modeled at this stage.

Since RD processes do not propagate errors to FP processes, we customize separately the error models associated respectively with RD and FP replicas. This is done by adding an **out** propagation declaration in the error model type associated with FP processes (see line  $f_o$  of Figure V-23) and a transition triggered by it in the corresponding error model implementation (see line  $f_o^1$  of Figure V-23). Also, a name-matching **in** propagation is declared in the error model type associated with RD processes (see line  $f_i$  of Figure V-24) and a transition triggered by it in the corresponding error model implementation (see line  $f_i^1$  of Figure V-24).



**Figure V-23.** Error model for functional dependency FP process – RD process (sender side)



**Figure V-24.** Error model for functional dependency FP process – RD process (recipient side)

### V.2.5.3 AADL to GSPN model transformation

Figure V-25 shows the GSPN obtained when transforming the AADL model of one FP process and the two RD processes, taking into account the functional dependency from the FP process to the RD processes. The GSPN is the same for *Configuration1* and *Configuration2*.

We apply the model transformation rules presented in Section IV.2 (for independent components), Section IV.3.3 (for name-matching *in* – *out* propagations) and Section IV.7 (for architecture configurations). The component nets *FP\_Comp1*, *RD\_Comp1* and *RD\_Comp2* in Figure V-25 are the same as those of Figure V-17.

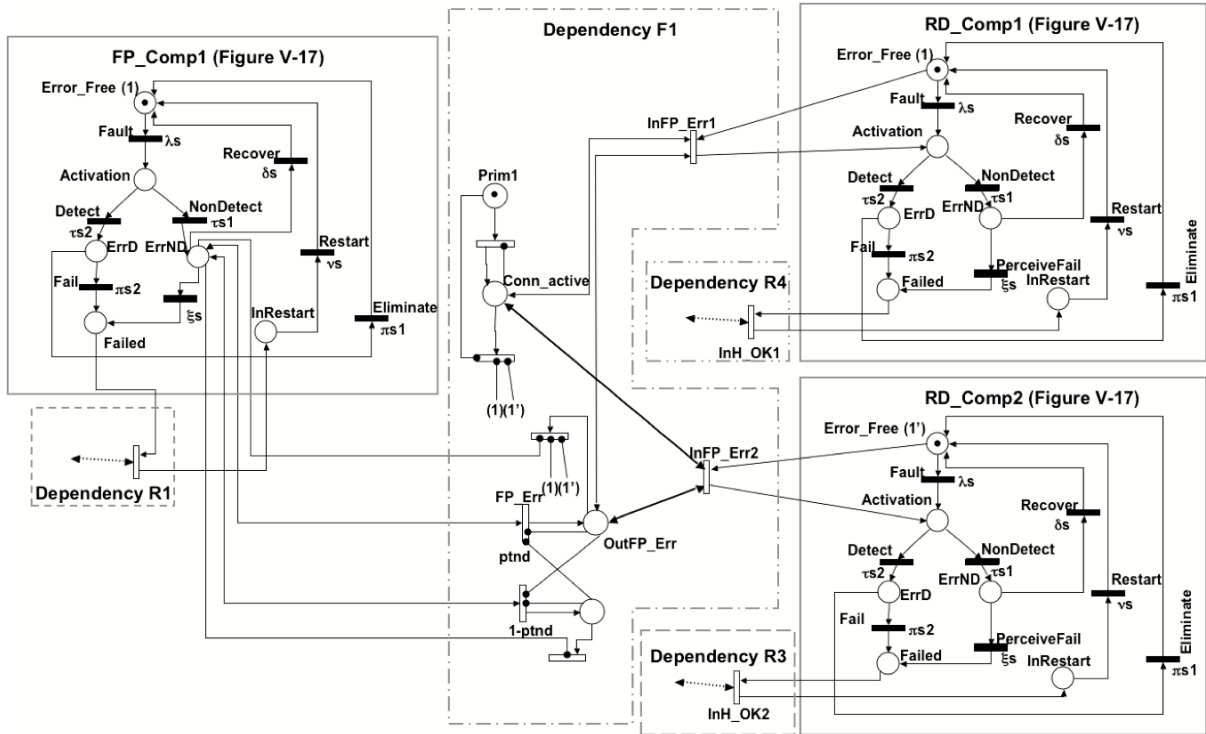


Figure V-25. GSPN modeling a FP process and the two RD processes with functional dependency

## V.2.6 Iteration 6: functional dependency between FP processes

### V.2.6.1 Assumptions

According to the system description, the primary FP process may propagate errors to the backup FP process. We assume that only non-detected errors propagate with a given probability  $p_{md}$ . Even though there is a connection from RD to FP, we assume that RD replicas do not propagate errors to FP.

### V.2.6.2 AADL dependability model

Since the backup FP process does not propagate errors to the primary FP process, the data connection from the backup to the primary process must be deactivated. The AADL architectural model is refined by making the connection from *FP\_Comp1* to *FP\_Comp2* active in mode *Prim1* and the connection from *FP\_Comp2* to *FP\_Comp1* active in mode *Prim2*.

The error model associated with the two FP processes is customized to receive the *FP\_Err* propagation introduced in the previous iteration. This is achieved by adding the identifier *in* to the propagation *FP\_Err*, which was already declared (see line  $f_o$  of Figure V-26) and a transition triggered by this *in* propagation in the corresponding error model implementation (see line  $f_i^2$  of Figure V-26).

| <b>Error Model Type [sw_FP]</b>  |  |
|--|--|
| <pre> <b>error model</b> sw_FP <b>features</b> [...] *(f<sub>o</sub>) FP_Err: <b>in out error propagation</b> {Occurrence =&gt; fixed ptnd}; <b>end</b> sw_FP; </pre>              |  |
| <b>Error Model Implementation [sw_FP.SRF1F3]</b>   |  |
| <pre> <b>error model implementation</b> sw_FP.SRF1F3 <b>transitions</b> +(f<sub>o</sub><sup>2</sup>) Error_Free-[<b>in</b> FP_Err]-&gt;Activation; <b>end</b> sw_FP.SRF1F3; </pre> |  |

**Figure V-26.** Error model for functional dependency between FP processes

### V.2.6.3 AADL to GSPN model transformation

The result of the transformation for this dependency is similar to the one of Figure V-25, thus we do not show it here. We apply the model transformation rules presented in Section IV.2 (for independent components), Section IV.3.3 (for name-matching **in – out** propagations) and Section IV.7 (for architecture configurations).

## V.2.7 Iteration 7: fault-tolerance dependency between FP processes

### V.2.7.1 Assumptions

The system description sets the number of replicas to two. We consider two fault-tolerance policy variants, FT1 and FT1', which have been presented in Section V.1.

FP is formed of two replicas that communicate through data connections and both replicas are connected to FP's ports *sysInput* and *sysOutput*. FP has two modes: in each mode, one replica is the primary. The closest fault-tolerance pattern to this AADL architectural model is C2' (see Section III.3.1.1.5). In this pattern, the two replicas keep their respective roles after a reconfiguration due to the failure of the primary. To model FT1, it is sufficient to instantiate the pattern C2'. To model FT1', we customize an instance of this pattern.

### V.2.7.2 AADL dependability model

We add to the AADL architectural model of FP the elements that are part of the pattern C2', including the error model annex subclause with the **Guard\_Event** property declarations. Both for FT1 and FT1', we customize the error model associated with the FP processes by declaring an **out** propagation that notifies the failure of the component (see lines *ft<sub>o</sub>* and *ft<sub>o</sub><sup>1</sup>* of Figure V-27). For FT1' only, we need to customize the **Guard\_Event** property declared in *FP\_Comp1* by adding a new mode switch condition so that *FP\_Comp1* becomes the primary again after its failure and restoration (see line *ft<sub>1</sub>'* of Figure V-28).

The refined AADL architectural model will be shown at the end of this Chapter (after having taken into account the fault-tolerance dependency in the RD subsystem and the maintenance dependency between the two processors).

|  |  |
|--|--|
| <b>Error Model Type [sw_FP]</b>                    |  |
| +(ft <sub>o</sub> )                                | <pre> <b>error model</b> sw_FP <b>features</b> [...] FP_FailedVisible: <b>out error propagation</b> {<b>Occurrence =&gt; fixed pf</b>}; <b>end</b> sw_FP; </pre> |
| <b>Error Model Implementation [sw_FP.SRF1F3FT]</b> |  |
| +(ft <sub>o</sub> <sup>1</sup> )                   | <pre> <b>error model implementation</b> sw_FP.SRF1F3FT <b>transitions</b> Failed-[<b>out</b> FP_FailedVisible]-&gt;Failed; <b>end</b> sw_FP.SRF1F3FT; </pre>     |

**Figure V-27.** Error model for FT dependency between FP processes

|                       |   |
|-----------------------|---|
| +(ft <sub>1</sub> ' ) | <pre> <b>process implementation</b> replica.primary [...] <b>annex</b> Error_Model {**   <b>Model</b> =&gt; sw_FP.SRF1F3FT;   <b>Guard_Event</b> =&gt;     fromReplica[FailedVisible] <b>and</b> self[Error_Free]     <b>or</b> fromReplica[FailedVisible] <b>and</b> self[Error_Free]   <b>applies to</b> IAmPrim; **}; <b>end</b> replica.primary; </pre> |
|-----------------------|---|

**Figure V-28.** Textual AADL dependability model for FP\_Comp1, with dependency FT1'

### V.2.7.3 AADL to GSPN model transformation

Figure V-29 shows the GSPN obtained when transforming the AADL model of the two FP processes linked by the considered fault-tolerance dependencies. The subnet FT1' is added in the case of the dependency FT1' to model the supplementary **Guard\_Event** condition. The rest of the GSPN is the same for both FT1 and FT1'. We apply the model transformation rules presented in Section IV.2 (for independent components) and Section IV.5.1 (for **Guard\_Event** properties).

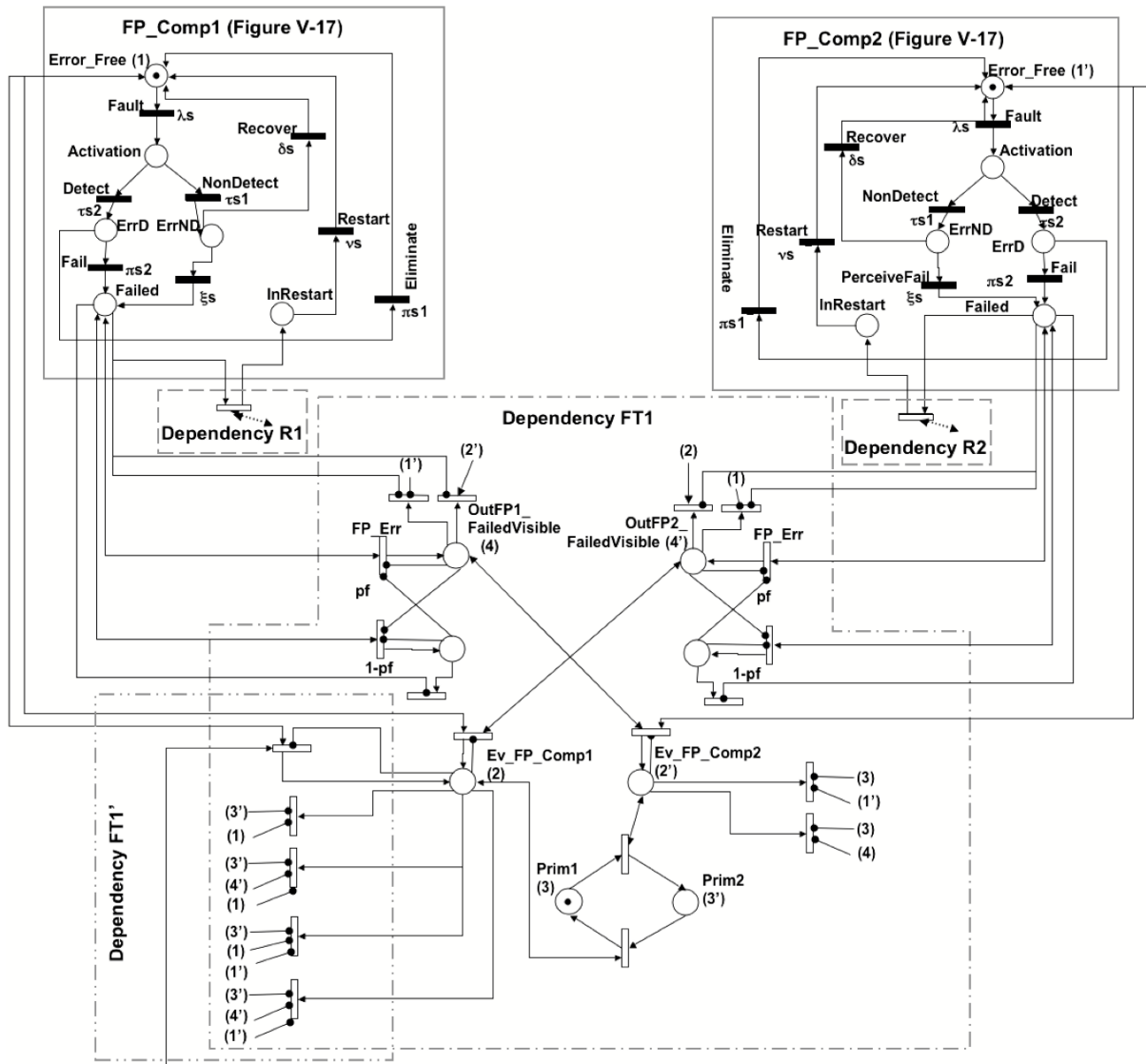


Figure V-29. GSPN modeling the two FP processes with FT dependency

## V.2.8 Iteration 8: fault-tolerance dependency between RD processes

### V.2.8.1 Assumptions

The system description sets the number of replicas to two. The two replicas do not communicate directly one with the other. The closest fault-tolerance patterns to this AADL architectural model are *CI* and *CI'* (see Section III.3.1.1.4). Here, we chose to instantiate *CI*.

### V.2.8.2 AADL dependability model

We add to the AADL architectural model of RD the elements that are part of the pattern *CI*, including the error model annex subclauses with the **Guard\_Event** property declarations. We customize the error model associated with RD processes by declaring an **out** propagation *RD\_FailedVisible* that notifies its failure, as in the case of the FP processes (see Figure V-27). The refined architectural model is shown at the end of this Chapter.

### V.2.8.3 AADL to GSPN model transformation

Figure V-30 shows the GSPN obtained when transforming the AADL model of the two RD processes with the considered fault-tolerance dependency. We apply the model transformation rules presented in Section IV.2 (for independent components) and Section IV.5.1 (for **Guard\_Event** properties). The decider component net is not entirely detailed here.

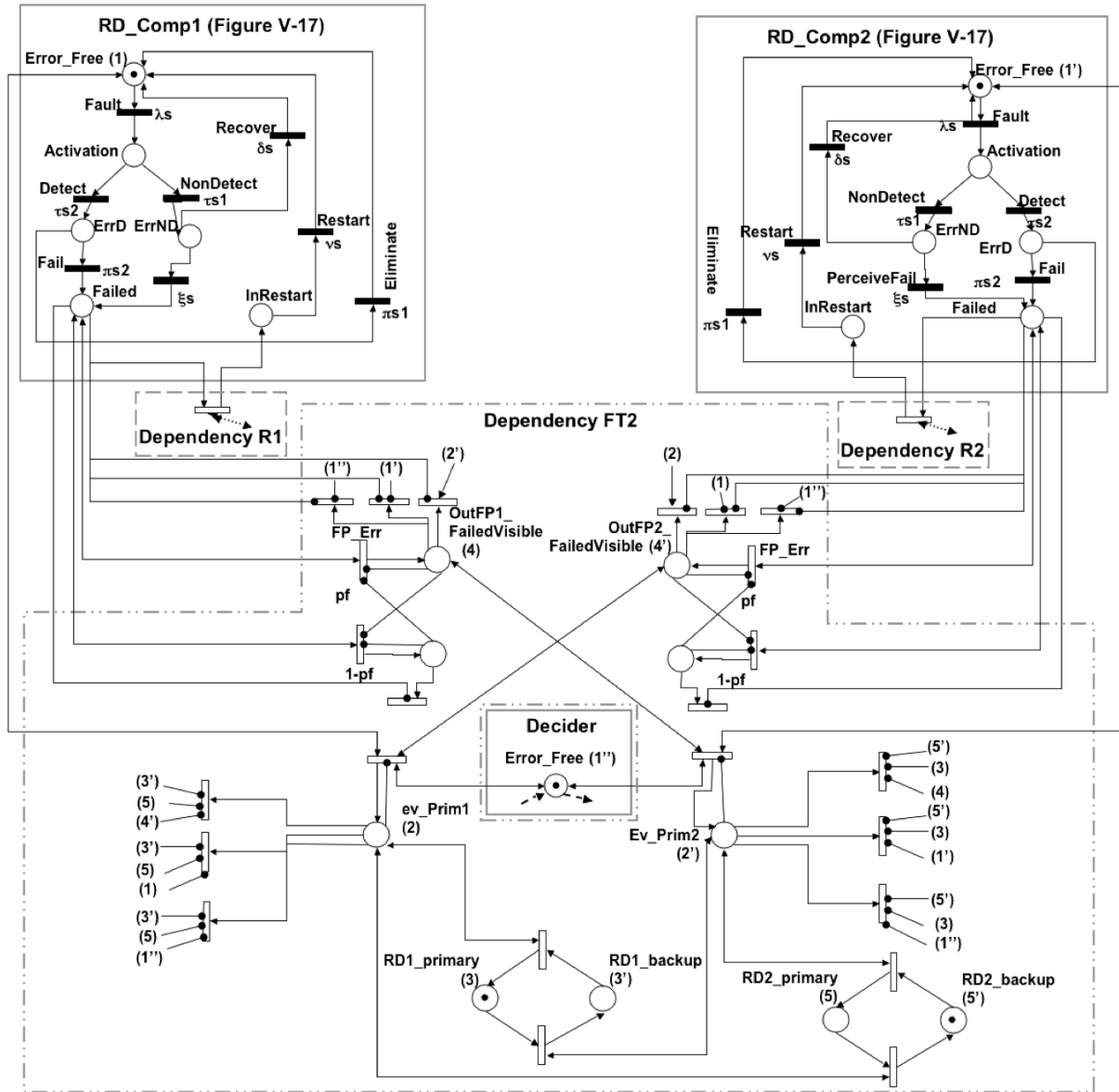


Figure V-30. GSPN modeling the two RD processes with FT dependency

## V.2.9 Iteration 8: global reconfiguration strategy

### V.2.9.1 Assumptions

The subsystem is in *Nominal* mode when both *FP\_Comp1* and *RD\_Comp1* have the primary role or the backup role. Otherwise, it is *Reconfig* mode.



### V.2.9.2 AADL dependability model

The reconfiguration strategy is modeled by associating **Guard\_Transition** properties with the two mode transitions, as shown in Figure V-31. The conditions defined by these properties override the default **or** conditions. Also, the two **out** event ports triggering the two mode transitions (*FP\_Comp1\_Primary* of component FP and *RD\_Comp1\_Primary* of component RD) must be connected to **out** event ports of subcomponents that generate events, through **Guard\_Event** properties. Concretely, *FP\_Comp1\_Primary* is connected to port *IAmPrim* of *FP\_Comp1* and *RD\_Comp1\_Primary* is connected to port *Prim1* of the *Decider* in RD. The refined architectural model is shown at the end of this Chapter.

```

system implementation areaCC.cautra
[...]
annex Error_Model {**
  Guard_Transition =>
    FP.FP_Comp1_Primary and RD.RD_Comp1_Primary
    or (not FP.FP_Comp1_Primary) and (not RD.RD_Comp1_Primary)
    applies to ReconfigToNominal;

  Guard_Transition =>
    (not FP.FP_Comp1_Primary) and RD.RD_Comp1_Primary
    or FP.FP_Comp1_Primary and (not RD.RD_Comp1_Primary)
    applies to NominalToReconfig;
**};
end areaCC.cautra;

```

Figure V-31. Modeling the global reconfiguration strategy

### V.2.9.3 AADL to GSPN model transformation

Figure V-34 shows the GSPN obtained when transforming the two **Guard\_Transition** properties. We apply the model transformation rule presented in Section IV.5.2 (for **Guard\_Transition** properties). The place *ev\_FP\_Comp1* has already been created when transforming the dependency FT1 (see Figure V-29). The place *ev\_Prim1* has already been created when transforming the dependency FT2 (see Figure V-30). They represent the occurrence of events through ports *FP\_Comp1\_Primary* and *RD\_Comp1\_Primary*.

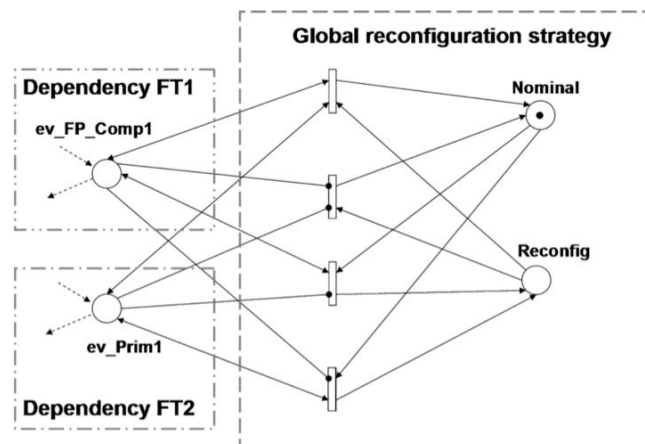


Figure V-32. GSPN modeling the two RD processes with FT dependency

## V.2.10 Iteration 10: maintenance dependency between processors

### V.2.10.1 Assumptions

We assume that the two processors share one repairman. If the repairman is busy repairing a processor while the second processor fails, this second processor must wait until the repairman finishes repairing the first processor.

### V.2.10.2 AADL dependability model

We have already shown how these assumptions are modeled in AADL in Section III.2.3.1. The AADL architectural model must be enriched to represent the repairman as a component connected with the two processors. Figure V-33 shows the refined AADL architectural model that takes into account the fault-tolerance and maintenance dependencies. The repairman's error model is the one for a shared maintenance facility, already presented in Section III.2.3.1. We do not revisit it here. The error model associated with the two processors is enriched with lines *fl-3*, *ol-2*, *il*, *ol'-2'*, *il'* of Figure III-4, to describe the maintenance dependency.

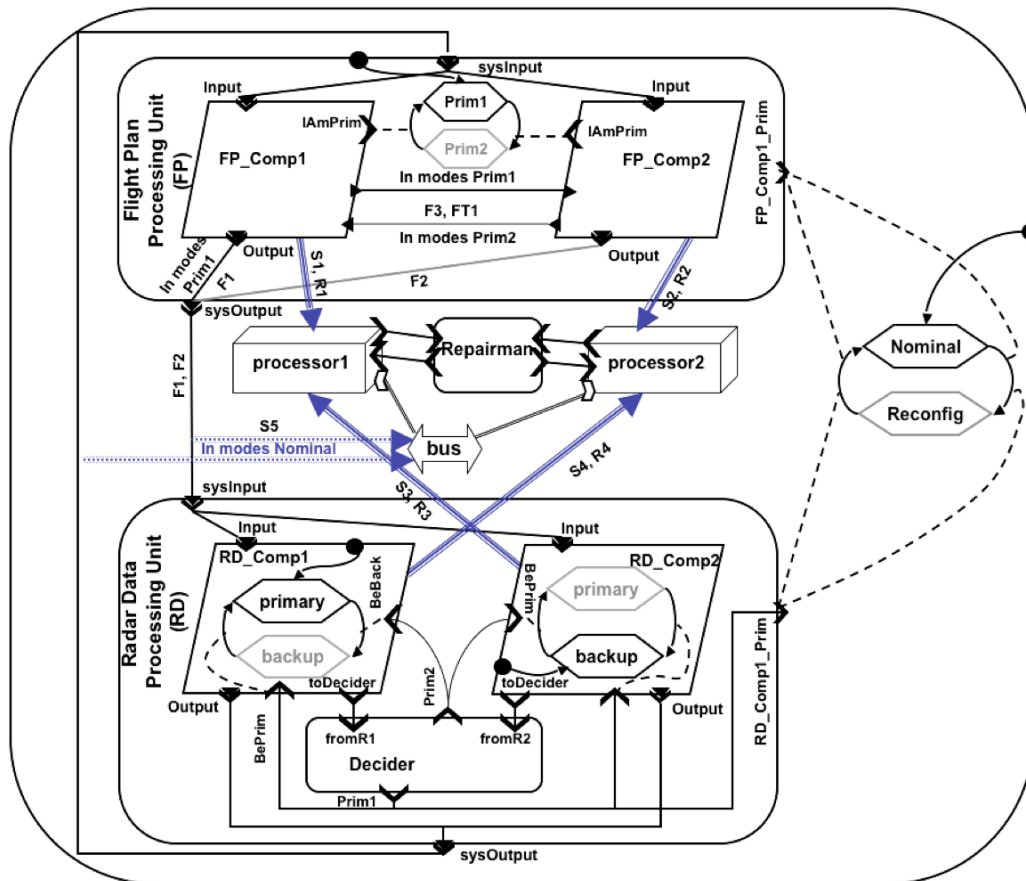


Figure V-33. Refined AADL architectural model of Configuration1

### V.2.10.3 AADL to GSPN model transformation

Figure V-34 shows the GSPN obtained when transforming the AADL model of the two processors linked by the considered maintenance dependency.

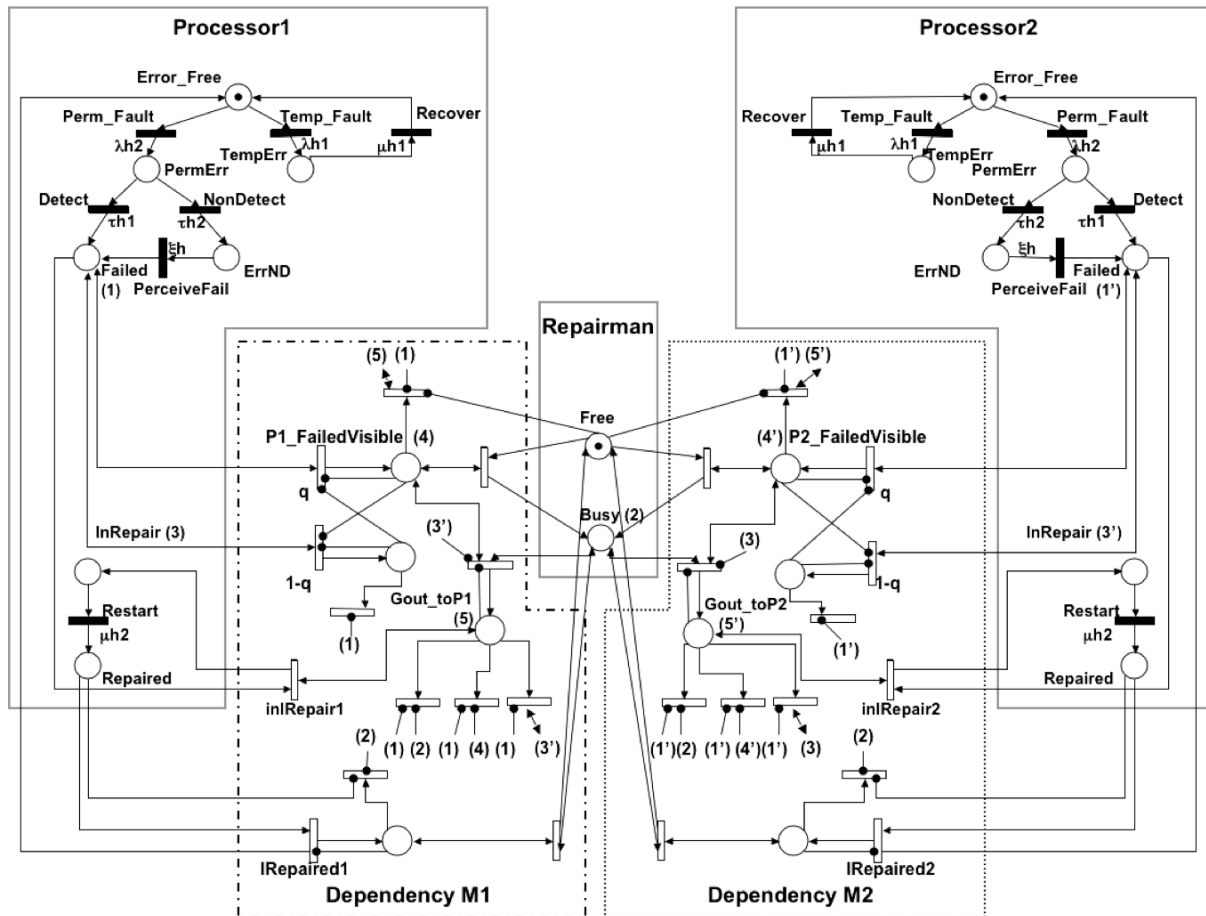


Figure V-34. GSPN modeling the two processors with maintenance dependency

### V.3 Quantitative dependability evaluation

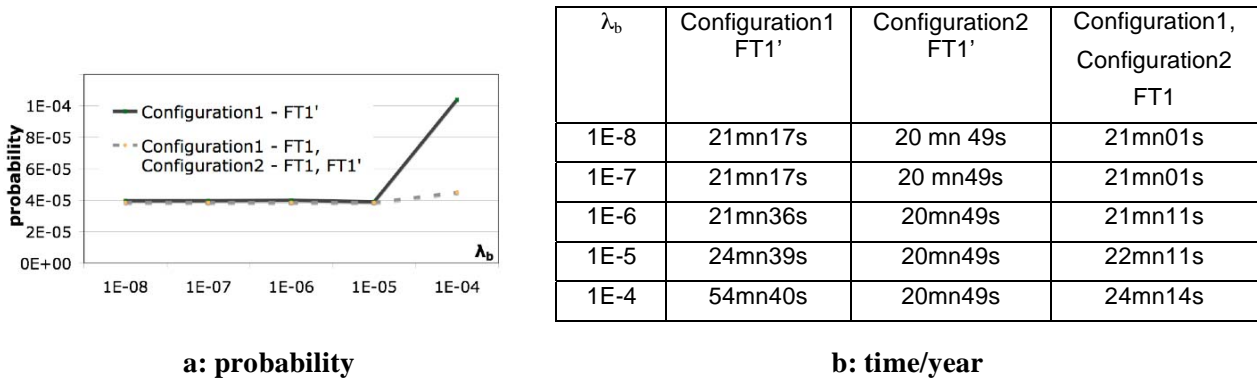
In the previous sections, we have built iteratively the AADL dependability models for two candidate architectures, by considering each time two fault-tolerance policies for the FP subsystem. Thus, we have built AADL and GSPN models allowing the analysis of four alternatives.

The purpose of the current Section is to illustrate the kind of quantitative results that can be obtained from the models built in the previous sections. In particular, we show a comparison of the four alternatives of the subsystem of CAUTRA, with respect to their availability. Subsection V.3.1 compares the four alternatives by setting all model parameters except for the failure rate of the bus,  $\lambda_b$ . Subsection V.3.2 compares them with respect to the fault-tolerance policy for the FP subsystem.

#### V.3.1 Comparison with respect to the failure rate of the bus

The unavailability of the FP subsystem is not influenced by  $\lambda_b$ , the failure rate of the bus, since a broken connection from RD to FP does not cause the failure of FP. The unavailability of FP is of 2h42mn/year. On the other hand, the unavailability of the RD subsystem is

influenced by  $\lambda_b$ , as shown in Figure V-35, as a broken connection from FP to RD causes the failure of RD.  $\lambda_b \leq 10^{-6}/h$  corresponds to a redundant bus. The impact of  $\lambda_b$  has little influence on the unavailability for *Configuration2* and FT1, because in these cases, the communication from FP to RD is not bound to the bus most of the time. For *Configuration1* and FT1', the impact is important when  $\lambda_b \geq 10^{-5}/h$ . From a practical point of view, if  $\lambda_b \geq 10^{-5}/h$ , it is recommended to use *Configuration2* or FT1.



**Figure V-35.** Unavailability of RD with respect to  $\lambda_b$

### V.3.2 Comparison with respect to the FT policy

Both for FP and RD, with the fault-tolerance policy FT1, the unavailability is not influenced by the use of *Configuration1* or *Configuration2* (see Table V-1, line FT1). Actually, if the system is not reconfigured to find its nominal configuration after the failure and restoration of a FP replica, it will spend as much time in *Configuration1* as in *Configuration2*.

For RD in *Configuration1*, the unavailability is higher for FT1' than for FT1 while in *Configuration2*, the unavailability is higher for FT1 than for FT1' (see Table V-1-b). Thus, we may conclude that for *Configuration1* the best fault-tolerance policy is FT1 while for *Configuration2* the best is FT1'.

**Table V-1.** Unavailability with respect to the fault-tolerance policy (time/year)

|      | a: FP          |                | b: RD          |                |
|------|----------------|----------------|----------------|----------------|
|      | Configuration1 | Configuration2 | Configuration1 | Configuration2 |
| FT1  | 2h42mn         | 2h42mn         | 21mn11s        | 21mn11s        |
| FT1' | 2h42mn         | 2h41mn         | 21mn36s        | 20mn49s        |

### V.4 Conclusion

In this Chapter, we have illustrated the feasibility of the proposed modeling approach by applying it to the comparison of four alternatives, resulting from two candidate architectures

and two fault-tolerance policies, for a subsystem of the French Air Traffic Control System. This subsystem is formed of four software components that form two fault tolerant units and three hardware components: two processors and a bus.

We have started from high level AADL architectural models built according to the system description. We have refined these models during the AADL dependability model construction and we have associated error models with components. The error models reflect the dependencies taken into account and have been built iteratively. For the considered subsystem, we have built the AADL dependability model in 10 iterations. The GSPNs obtained by applying our model transformation rules have been shown at each iteration. The global GSPNs are formed of 24 subnets. Among them, eight are component subnets and sixteen are dependency nets. The two component subnets modeling the replicas of each software unit are identical. Also, the two component subnets representing the two processors are identical. Among the dependency subnets, those modeling identical dependencies between identical components are identical. We have used two fault tolerance patterns: one of them has been adapted while the other one has been used unmodified. The subsystem that we have used for the illustration of our modeling approach is part of a real-life system formed of seventeen components, described in [Borrel 1996] and analyzed using GSPN built by hand. Since we provide a structured method for building the AADL dependability model and automatic means for obtaining the GSPNs from AADL dependability models, the effort necessary to analyze the whole system based on its AADL models is minimized.

In order to better support the iterative error model construction, we have proposed to introduce inheritance mechanisms in the Error Model Annex, which do not exist in the current version of the standard. Our proposal is further detailed in Appendix A. Such mechanisms would allow both enriching the models without copying and pasting from the previous modeling phases and clearly showing the evolution phases.

---

# Conclusion

---

This dissertation presented a modeling framework allowing the evaluation of quantitative dependability measures, such as reliability and availability, from AADL models. The ultimate objective of this framework is to ease the task of evaluating dependability measures in the context of modern model-driven engineering processes based on AADL. AADL is a mature industry-standard well suited to address quality attributes. In particular, it provides precise execution semantics for modeling the architecture of software systems and their target platform and has been seriously considered in the embedded safety-critical industry.

We have achieved the above-stated objective by proposing a framework formed of:

- 1) A structured method for expressing in AADL the information related to failure, recovery, maintenance and fault-tolerance, which is necessary for dependability analysis. Such a method is necessary when dealing with large systems having many interdependencies between components. It favors model reuse and evolution.
- 2) A set of patterns modeling common fault-tolerance mechanisms. The patterns are instantiated and inserted in the AADL model of a system, with or without customization. Model reusability is an essential issue in the context of complex, critical and evolvable systems. We also believe that the use of patterns enhances the understandability and the readability of the model.
- 3) A model transformation from AADL to a classical dependability evaluation model in the form of a GSPN. The transformation can be completely automated, thus hiding the complexity of traditional analytical models from AADL users. From a user's perspective, this allows the evaluation of dependability measures directly from the AADL model. We have implemented the model transformation tool ADAPT, to show the feasibility of the automation.
- 4) A set of evolution proposals for the AADL Error Model Annex, based on our extensive experience with AADL and dependability modeling requirements. We have submitted these proposals to the AADL standardization committee, which decided to integrate some of them in the current version of the standard. The remaining ones are considered for integration in the next version of the standard.

Our framework is iterative and is based on system dependability modeling using AADL and on a model transformation from AADL to GSPNs. We have chosen to use GSPNs because they provide means for structural verification of the model and can be automatically converted to Markov chains. In addition, they allow modular modeling, which is very convenient in the context of component-based systems, and they are able to capture various functional and stochastic dependencies among components.

The AADL modeling guidelines provided in this dissertation aim to assist the user in the structured construction of the AADL dependability model. To support and trace model evolution, this approach proposes that the user builds the AADL dependability model iteratively. Components' behaviors in the presence of their own faults are modeled in the first iteration as if they were independent. Then, dependencies between system components are introduced iteratively in the AADL dependability model. The proposed fault-tolerance modeling patterns may also be used in the context of this iterative modeling approach.

The AADL dependability model is transformed into a GSPN to be processed by existing tools. The model transformation can be performed iteratively, each time the AADL dependability model is enriched. In this way, the GSPN model can be progressively validated (hence the corresponding AADL dependability model can be progressively validated too, and corrected accordingly, if required). The GSPN of the global system is structured as a set of interacting subnets, where a subnet is associated with a component or a dependency.

To be automated, the transformation is based on rigorous and systematic rules. All rules are defined to ensure that the obtained GSPN is correct by construction (bounded and free of infinite loops over sets of vanishing markings). Also, the resulting GSPN is tool-independent (i.e., we do not use tool-specific features or predicates).

We have implemented ADAPT, a model transformation tool that interfaces the Open Source AADL Environment (OSATE) on the AADL side and the dependability evaluation tool Surf-2 on the GSPN side. Besides the GSPN complying with the input format of Surf-2, our tool also generates a generic GSPN under XML/XMI form. This XML/XMI file represents a gateway to other dependability evaluation tools, as the processing techniques for XML files allow it to be easily processed in order to obtain a tool-specific GSPN.

We have illustrated the proposed framework by applying it to a subsystem of the French Air Traffic Control System. Based on the informal description of the system, we have identified the dependencies between components and we have considered maintenance and fault-tolerance policies. Then, we have shown the iterations related to the AADL dependability model construction and to the AADL to GSPN transformation. We have also provided examples of analyses based on the models that we have built. They are related in particular to the comparison of four alternatives (resulting from two candidate architectures and two fault-tolerance policies considered for the subsystem) with respect to their unavailability. The subsystem that we have used to illustrate our modeling framework is part of a larger, real-life system formed of seventeen components. The GSPN modeling the whole system has been built previously by hand and analyzed using the dependability evaluation tool Surf-2. In this dissertation, we focused on obtaining analyzable GSPNs from AADL dependability models. Since we provide a structured method for building the AADL dependability model and automatic means for obtaining a GSPN from it, the effort necessary to analyze the whole system based on its AADL model is minimized.

Our most important AADL evolution proposals are briefly presented in Appendix A and they refer to (1) parametric Occurrence properties, (2) the enforcement of the link between operational modes and Error Model annex constructs, (3) removing the **applies to** clause from the **Guard\_In** property syntax and (4) the definition of inheritance and refinement mechanisms. Proposal 1 is part of the current version of the standard. Proposal 1 is integrated in the first standardized version of the AADL Error Model Annex. Proposal 3 has been accepted by the standardization committee in April 2007. Proposals 2 and 4 have been put in standby, as it would be interesting to have these mechanisms applied to several AADL annexes.

## ***Future research directions***

Several directions can be explored to extend the contributions presented in this dissertation.

A first research direction concerns the scalability of the AADL to GSPN model transformation. The study that we have carried out showed that in the case of several identical propagation receiver components for one sender component, the state space grows

exponentially. This identified limitation is directly related to the fact that, in AADL, all identical components are represented separately and thus, the GSPN obtained by direct model transformation is not compact. Currently, it is possible to apply GSPN reduction methods in order to process a compact GSPN. However, for large models, it may be difficult to generate the non-compact GSPN. Thus, future research should aim at finding an efficient way for obtaining directly a compact GSPN from the AADL model. An idea to be explored in this context would be to find an algorithm based on a first traversal of the entire AADL model and on the identification of identical components.

Also, it is desirable to improve the model transformation prototype tool that we have implemented, so that the generic GSPN obtained as an output complies with the Petri Net Markup Language (PNML), as PNML is intended to become an extensible interchange standard for Petri nets.

More generally, it would be worth considering the elaboration of model transformations from AADL to other formalisms, allowing the evaluation of similar or complementary analyses to ours, based on the same model. For dependability, performance and performability analyses, it would be particularly interesting to define a model transformation from AADL to stochastic activity networks (SAN), which are an extension of stochastic Petri nets. SANs are supported by Möbius, a widely-used tool in the performance and dependability evaluation community [Deavours *et al.* 2002]. Also, investigating other formalisms allowing the use of general distributions would enable the use of any type of distribution for the firing delays characterizing events and propagations. The drawback of using GSPNs is that they only allow us to use fixed probabilities or exponentially distributed firing delays, while in AADL there is no constraint at this level.

Finally, it would be interesting to apply our modeling framework to other complex case studies from different application domains (e.g., aerospace, automotive), in order to study the suitability of our framework in other contexts. Such studies could also help identifying potential evolution directions for the AADL standard.





---

# Appendix A: Error Model Annex Evolution Proposals

---

Our experience in dependability modeling using AADL and the AADL Error Model Annex allowed us to propose evolutions of the standard documents. The proposals that we consider as the most important are presented in this appendix. The identified issues, classified in the order of their importance are enumerated hereafter.

- 1) Parametric Occurrence properties;
- 2) Enforcement of the link between the modes and Error Model Annex constructs;
- 3) **Guard\_In** properties without **applies to** clause;
- 4) Inheritance and refinements.

Proposals 1 and 3 are strictly specific to the Error Model Annex, as they apply to specific constructs of this annex. The others may be generalized to some extent to apply to other annexes. As stated in the conclusion of this dissertation, proposal 1 has been integrated to the standard document of the AADL Error Model Annex v1. Proposal 3 has been accepted by the AADL standardization committee in April 2007 and will be integrated with the AADL Error Model Annex Errata. Proposals 2 and 4 have been put in standby, as it would be interesting to have these mechanisms applied to several AADL annexes in a general manner.

We briefly present the proposals in the following subsections. Proposal 1, part of proposal 2 and proposal 4 have been detailed in [Arlat *et al.* 2006].

## ***A.1 Occurrence properties***

In previous AADL Error Model Annex drafts, Occurrence properties had only numeric values. Our proposal consisted in allowing the specification of symbolic values for Occurrence properties. Such parametric Occurrence properties are of interest for performing sensitivity analyses that aim for example at identifying acceptable limit values for the Occurrences of error and repair events for a given system. In addition, symbolic values for Occurrence properties allow the definition of generic error models that can be customized with component-specific Occurrence values when they are instantiated.

## ***A.2 Link between modes and the Error Model Annex constructs***

Mode transitions are influenced by the error model behavior through the use of **Guard\_Event** and **Guard\_Transition** properties. By using these properties, a user is able to express that a mode change occurs as a result of a particular error state configurations or at the arrival of particular error propagations.

The error model behavior is influenced by the system's modes through the use of **activate** / **deactivate** transitions. They allow modeling different behaviors of a

component in the presence of faults depending on whether it is active or inactive. An example of such behavior is that a software component may not fail if it is inactive. However, if a component is active in all modes, it is not possible to describe different behaviors of the component in the different modes. For example, there is no mechanism for modeling a mode-specific error propagation, which occurs only if the component is in a particular mode. Such a mechanism is necessary to model e.g., the fact that a component would propagate errors only if it is in Primary mode. Also, such a mechanism would allow modeling a priority to repair for a hardware component hosting the most critical application of the system.

We consider two mechanisms to address this issue:

- 1) Allowing **in modes** statements inside Error Model Annex subclauses, to declare mode-specific constructs (e.g., **Model**, **Derived\_State\_Mapping**, **Guard**, **Occurrence** properties).
- 2) Allowing **in modes** statements inside Error Model Annex libraries, i.e., inside error model definitions, to declare mode-specific error states, events, propagations and transitions.

We believe that the first mechanism is better adapted if the component has the same error model in different modes and the only mode-specific changes concern **Occurrence** and **Guard** properties. This means that the error model is only customized for the different modes. For derived error models, it is interesting to have the possibility of declaring different **Derived\_State\_Mapping** expressions for different modes of the system. For example, this would allow expressing that the system is *Error\_Free* if it is in mode *Comp1Primary* and if *Comp1* is *Error\_Free*. On the other hand, if *Comp1* is *Error\_Free* but the system is another mode, it means that the system fails to deliver the service expected from it.

The second mechanism is useful to declare mode-specific behavior that consists in the occurrence or non-occurrence of particular events or propagations depending on the mode of the component.

### **A.3 *Guard\_In* property without *applies to clause***

A **Guard\_In** property maps an incoming set of error propagations and error states from other components into a set of **in** propagations that may affect the receiving component. The current Error Model Annex specifies that a **Guard\_In** property is associated with an **in** feature. One can apply the same **Guard\_In** property to one feature or to several features, by naming them in the **applies to** clause of the **Guard\_In** property. This permission raises two issues:

- Applying different **Guard\_In** properties that refer to the same features to each one of the features may lead to conflicts between the **Guard\_In** properties, and thus to non-determinism. An example is shown in Figure A-1, in which the first **Guard\_In** property specifies that *KO1* is perceived by the component when *outKO* propagations are received through both ports *inp1* and *inp2*, while the second **Guard\_In** property specifies that *KO2* is perceived when this same condition is true.
- Applying the same **Guard\_In** property to several features or to one feature only has the same impact on the component declaring the **Guard\_In**. This means there are several ways to express the same behavior and there are no guidelines with regards to when to apply a **Guard\_In** property to one feature only or to all features referred to in the **Guard\_In** Boolean expressions.

```

Guard_In =>
    KO1 when (inp1[outKO] and inp2[outKO])
    mask when others
applies to inp1;

Guard_In =>
    KO2 when (inp1[outKO] and inp2[outKO])
    mask when others
applies to inp2;

```

Figure A-1. *Conflicting Guard\_In properties*

On the other hand, a transition that names an **in** propagation occurs regardless of the feature through which that propagation arrives.

To address these issues, we propose to remove the **applies to** clause from the **Guard\_In** declaration, thus allowing the definition of a unique **Guard\_In** for all incoming features of a component.

#### A.4 *Inheritance and refinements*

Inheritance and refinement mechanisms are useful when dealing with an incremental modeling approach, such as the one proposed in this dissertation. This kind of mechanism would ease the modeler's task during the model evolution phases by allowing the models to be enriched without copying and pasting from the previous modeling phases. In addition, the model evolution would be clearly visible for external readers.

Our proposal is to integrate inheritance and refinements in the AADL Error Model Annex in a way similar to the one specified in the AADL core standard. These mechanisms would apply to error model types and implementations.

- A child error model type may extend only one parent error model type. The child error model type may add error states, events and propagations to the inherited ones. It may not suppress any feature declared in the parent error model type. An inherited feature may be refined into a set of features. If the initial error state is refined, only one of the replacing error states in the child error model type is declared as being *initial*. Error model types may form an hierarchy with a child error model type inheriting from its parent, which inherits from an ancestor error model type.
- A child error model implementation may extend only one parent error model implementation. By default, the child error model implementation inherits the error model type of the parent error model implementation. The modeler can also associate it explicitly with an error model type that inherits the one of its parent error model implementation. The child error model implementation inherits the set of transitions declared in its parent and adds new transitions to this set. It may also refine a transition, i.e., replace it with a sequence of transitions starting from the source state of the refined transition and ending to the destination state of the refined transition.



## Appendix B: General Rule for Emptying GSPN Propagation Places

The rule we present hereafter ensures that the behavior modeled by the GSPN is semantically identical to the behavior modeled through propagations in the AADL model. In addition, this rule ensures the boundedness of the GSPN: any token created in a dependency net and representing a propagation must be absorbed when leaving the dependency net.

There are two types of GSPN propagation places: those generated from AADL transitions triggered by **out** propagations and those generated from **Guard\_Out** pass-through rules. The rule presented hereafter applies to both of these types.

Generally, a token is created in a place *OutProp* by firing a GSPN transition corresponding to a *cause* of the corresponding **out** propagation. For an **out** propagation occurring as a result of an AADL transition triggered by the **out** propagation, the cause is unique. For an **out** propagation occurring as a result of a **Guard\_Out** pass-through rule, there are as many causes as conjunctions in the Boolean expression of the pass-through rule. The occurrence of an **out** propagation (i.e., the existence of a token in the place *OutProp*) has one or more *effects* represented as GSPN transitions to which the place *OutProp* is linked through bi-directional arcs. Figure B-1 presents a GSPN representing an **out** propagation with its causes and effects.

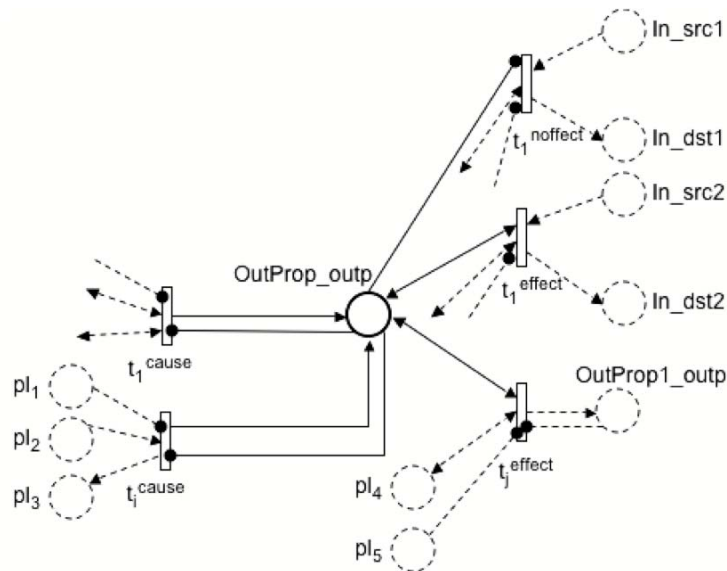


Figure B-1. General GSPN for **out** propagation

The place *outProp\_out* models the **out** propagation. This place can be emptied when none of the GSPN transitions connected to it (corresponding to causes and effects) is enabled. The Boolean condition to empty *OutProp* is as follows.

$[\bigwedge_{i=1}^n (\neg t_i^{cause})] \wedge [\bigwedge_{j=1}^m (\neg t_j^{effect})]$ , where  $\neg t$  represents the Boolean condition necessary for  $t$  to be disabled (OR Boolean condition on the marking of places connected to  $t$ ).

According to Figure B-1:

$$\neg t_i^{cause} = pl_1 \vee (\neg pl_2) \quad \text{and} \quad \neg t_j^{effect} = (\neg pl_4) \vee pl_5 \vee (\neg outProp1\_outp)$$

If the place *OutProp* is linked through inhibitor arcs to other transitions, (e.g.,  $t_1^{noeffect}$  in the figure), they do not have any impact on emptying the place *OutProp*.

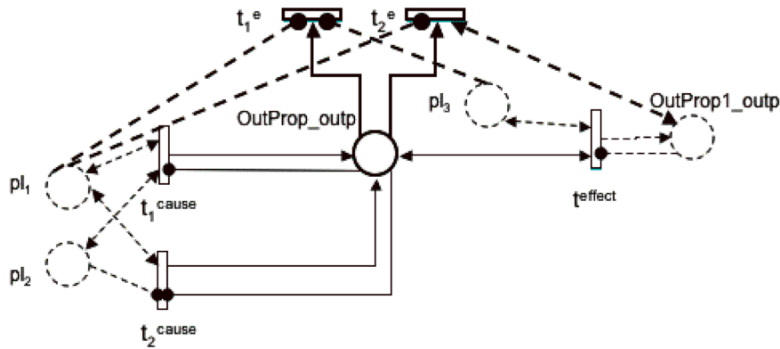
The Boolean condition to empty *OutProp* is transformed in DNF and minimized so that the number of GSPN transitions created to empty the place *OutProp* is minimized. An immediate GSPN transition is created for each conjunction of the Boolean condition. It has a uni-directional arc from the place *OutProp* that needs to be emptied. It has inhibitor arcs coming from places negated in the conjunction and bi-directional arcs connecting it to places that are not negated. An example is shown in Figure B-2. The place *OutProp\_outp* has two GSPN cause transitions and one GSPN effect transition. To empty the place *OutProp\_outp*, the following Boolean condition must be true.

$$(\neg t_1^{cause}) \wedge (\neg t_2^{cause}) \wedge (\neg t^{effect}) = [(\neg pl_1) \vee (\neg pl_2)] \wedge [(\neg pl_1) \vee (pl_2)] \wedge [(\neg pl_3) \vee OutProp1\_outp]$$

This expression in DNF form and minimized is as follows.

$$[(\neg pl_1) \wedge (\neg pl_3)] \vee [(\neg pl_1) \wedge OutProp1\_outp]$$

This leads to creating two immediate transitions that empty *OutProp\_outp*.



**Figure B-2.** Emptying an *out* propagation place - Example

# Appendix C: Model Transformation Tool

The model transformation rules presented in Chapter IV have been implemented in a tool prototype, ADAPT (*from AADL Architectural models to stochastic Petri nets through model Transformation*), that interfaces the Open Source AADL Tool Environment (OSATE<sup>35</sup>) on the AADL side and Surf-2 [Béounes *et al.* 1993] on the GSPN side. OSATE is the most used AADL modeling tool. From a developer's point of view, OSATE provides useful methods for traversing and processing the AADL architectural model. In addition to OSATE, we also base our tool on the set of prototype plug-ins developed at the Carnegie Mellon Software Engineering Institute to support the Error Model Annex.

Section C1 presents the model transformation tool from a developer's perspective while Section C2 presents it from a user's perspective.

## C.1 A developer's perspective

Figure C-1 presents the general overview of our tool: its structure and interfaces with AADL and GSPN tools respectively.

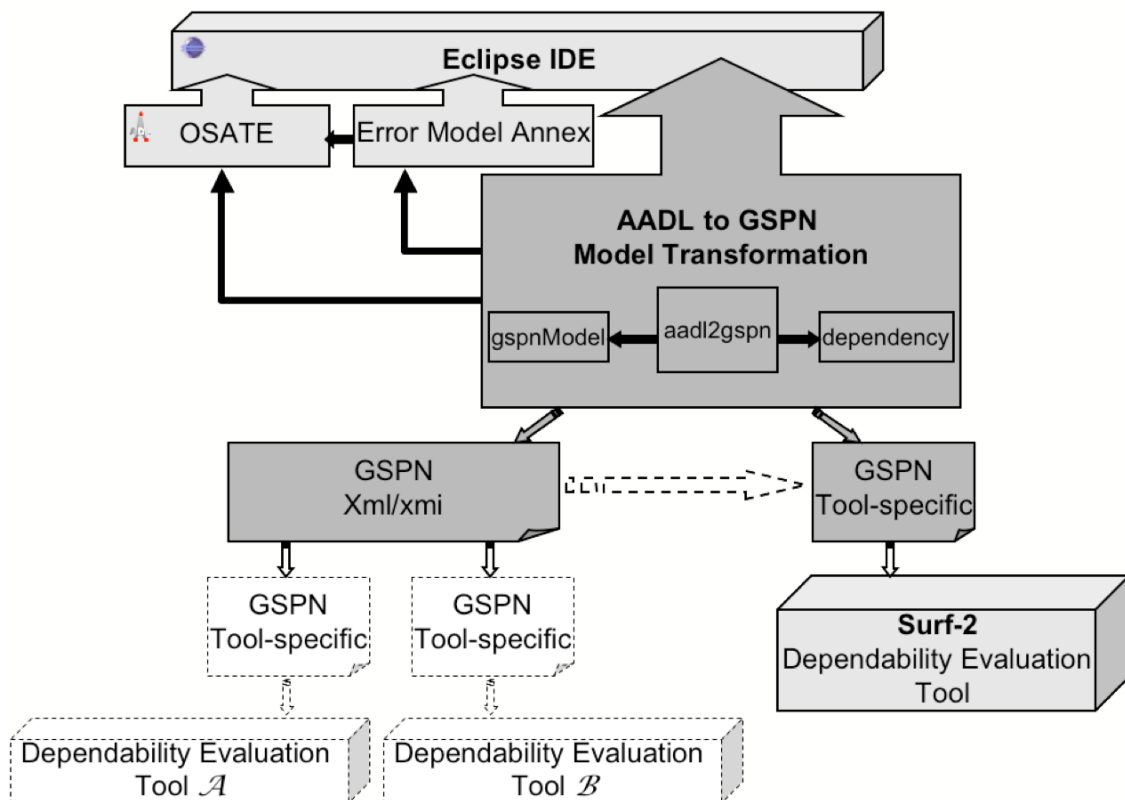


Figure C-1. Overview of the model transformation tool

<sup>35</sup> <http://www.aadl.info/OpenSourceAADLToolEnvironment.html>



Our model transformation tool, depicted in dark gray together with its outputs, is built in the Java programming language on top of the Eclipse IDE<sup>36</sup> (integrated development environment). This implementation choice is due to the fact that we interfaced our tool with OSATE.

Our tool consists of 10 kilo lines of code, half of which are automatically generated from an Ecore<sup>37</sup> metamodel using the Eclipse Modeling Framework [Budinsky *et al.* 2004]. It is structured as a set of three Eclipse plug-ins: *gspnModel* (containing methods for the creation and customization of GSPN elements), *dependency* (implementing the dependency rules defined in the AADL Error Model Annex) and *aadl2gspn* (implementing our transformation rules). The AADL and GSPN tools it interfaces are shown in light gray. The generated GSPN is saved in two forms: a generic XML/XMI file and a tool-specific file complying with the file format of the dependability evaluation Surf-2. Both files are obtained from the same GSPN object model, internal to our tool. The tool-specific file may also be obtained directly from the XML/XMI file. Possible interfaces with other GSPN-based dependability evaluation tools are represented with dotted lines.

The three plug-ins forming our model transformation tool are described successively in subsections C.1.1, C.1.2 and C.1.3.

### C.1.1 *gspnModel*: Ecore metamodel

This plug-in offers all the methods necessary for creating and customizing GSPN elements (places, transitions and arcs) and for traversing a GSPN model. The code of this plug-in has been automatically generated from an Ecore metamodel of GSPN using the Eclipse Modeling Framework (EMF). An XML/XMI schema is also generated from the Ecore metamodel. The GSPN saved under XML/XMI format is compliant with this schema.

Figure C-2 shows the Ecore metamodel that we defined and that served as a basis for the generation of code.

A *PetriNet* object contains several *Arcs* and several *PlaceOrTransition* elements. *Arcs* are described by a weight while *PlaceOrTransition* elements are identified by names. *Arcs* and *PlaceOrTransition* elements cannot be instantiated directly (they are abstract). Concrete arcs of types *PlaceToTransition* and *TransitionToPlace* can be instantiated and inherit from the *Arc* elements. *Place* and *Transition* elements inherit from the *PlaceorTransition* elements. A *Place* is characterized by an initial marking. A *Transition* is characterized by an Occurrence type and a parameter. Associations are established between the *TransitionToPlace* / *PlaceToTransition* arcs and *Place* and *Transition* elements.

One of the perspectives of this work is to use the Petri Net Markup Language (PNML) [ISO/IEC 2005] instead of the rather simple meta model for GSPN illustrated in Figure C-2. PNML is intended to become an extensible interchange standard for Petri nets.

---

<sup>36</sup> <http://www.eclipse.org/>

<sup>37</sup> Ecore is a small and simplified subset of UML, used in the Eclipse Modeling Framework.

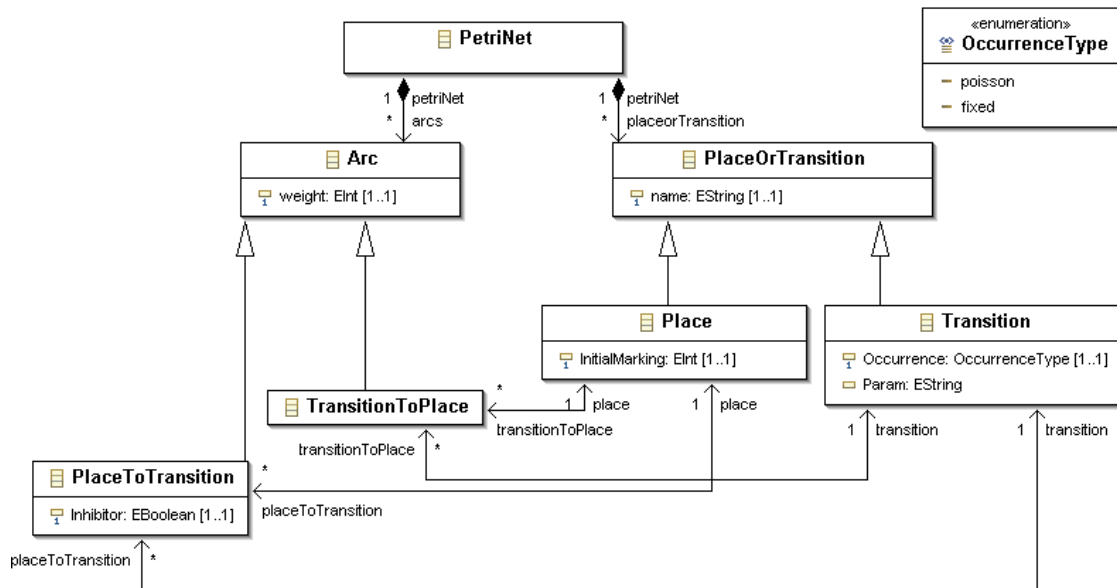


Figure C-2. Ecore metamodel for GSPN

### C.1.2 dependency

This plug-in is a library of methods implementing the dependency rules specified by the AADL Error Model Annex. These dependency rules determine the propagations between error models associated with components and connections. They allow finding receiver and sender components or connections for a given error propagation declared in an error model associated with a component or connection of the system instance. The details of this library are presented in [Bruffa & Rugina 2007].

### C.1.3 aadl2gspn

This is the main plug-in of our tool. It performs the transformation of error model elements into GSPN elements. It uses the *gspnModel* plug-in for the creation of GSPN objects and the *dependency* plug-in to determine the interactions at architectural level allowing the derivation of dependencies at error model level. The plug-in implements a metamodel-based transformation. It uses the standard metamodels of the AADL and of the AADL Error Model Annex [SAE-AS5506/1 2006a] as a source and of the GSPN as a target. We have chosen to implement the tool in Java in order to take advantage of the support offered by OSATE. Other implementation alternatives are recent metamodel-based transformation languages such as ATL [Jouault & Kurtev 2005], MOLA [Kalnins *et al.* 2005], MTL<sup>38</sup> or GReAT [Agrawal *et al.* 2003]. Model transformation techniques are compared in [Czarnecki & Helsen 2003].

In the current prototype, all transformation rules presented in Chapter IV are implemented, except for the rules for **activate** / **deactivate** transitions and derived error models. Architecture configurations are not taken into account.

<sup>38</sup> <http://modelware.inria.fr/article66.html>

## C.2 A user's perspective

A OSATE user installs our tool as an Eclipse feature and a set of plug-ins. Our tool requires that the Error Model Annex support plug-ins, provided by the Carnegie Mellon Software Engineering Institute, be installed too. In order to run the AADL to GSPN transformation tool, the user must instantiate an AADL system model and select the resulting system instance. The system instance must have an associated **Derived\_State\_Mapping** expression that we use to derive the state partitions necessary to the dependability evaluation tool, to evaluate measures. The **Derived\_State\_Mapping** expression must explicitly define the *Failed* global state of the system instance as a Boolean expression of states of its components. If safety is among the targeted measures, a *Catastrophic* global state must also be defined. As stated in Chapter III, the aim of our framework is the evaluation of quantitative dependability measures. Thus, it requires that all events and propagations have Occurrence properties. If an event or a propagation does not have an Occurrence property, our tool assumes it is immediate of probability 1.

The GSPN obtained after transformation is saved in two files with different formats:

- a generic XML/XMI file complying with a schema automatically generated from the metamodel that we defined for GSPN. The XML/XMI format of the GSPN does not comply with the input file format of a particular tool. However, it is useful for potential users intending to obtain a tool-specific GSPN, as XML/XMI files are easy to process. This file represents a gateway for interfacing other dependability evaluation tools with a minimum amount of effort. It is placed by default in the *aaxl* folder of the AADL project containing the instance model that has been transformed. It has the same name as the system instance file, followed by the *pn* extension.
- a tool-specific file complying with the input file format of the dependability evaluation tool Surf-2. The user chooses its name and location. This file can be imported in Surf-2. Surf-2 allows the user to customize the model, i.e., by giving particular values or value ranges to model parameters corresponding to symbolic Occurrence properties coming from the AADL model. The user also chooses the measures of interest.

---

## References

---

- [Agrawal et al. 2003] A. Agrawal, G. Karsai and F. Shi, Graph Transformations on Domain-Specific Models, Institute for Software Integrated Systems, Vanderbilt University, Technical Report, 2003.
- [Ajmone Marsan et al. 1995] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli and G. Franceschinis, Modelling With Generalized Stochastic Petri Nets, Wiley Series in Parallel Computing, 301p., John Wiley & Sons, 1995.
- [Arlat et al. 2006] J. Arlat, M. R. Barone, Y. Crouzet, J.-C. Fabre, J. Favaro, M. Kaâniche, K. Kanoun, S. Puri, T. Robert, M. Roy, A. E. Rugina, N. Salatge and H. Waeselyneck, Dependability Framework: Evaluation, Testing and Wrapping, Integrated Project IST - 004033 ASSERT Deliverable, N°D-32(345)-1 N°LAAS-CNRS Research Report n°06132, February 2006.
- [Avizienis 1995] A. Avizienis, “The Methodology of N-Version Programming”, in Software Fault-tolerance (M. R. Lyu, Ed.), pp.23-46, John Wiley & Sons Ltd., 1995.
- [Avizienis et al. 2004] A. Avizienis, J.-C. Laprie and B. Randell, “Dependability and its Threats: A Taxonomy”, in 18th IFIP World Computer Congress, pp.91-120, 2004.
- [Balbo et al. 1988] G. Balbo, S. C. Bruell and S. Ghanta, “Combining Queuing Networks and GSPNs for the Solution of Complex Models of System Behaviour”, IEEE Transactions on Computers, 37, pp.1251-1268, 1988.
- [Baresi 2002] L. Baresi, “Some preliminary hints on formalizing UML with Object Petri Nets”, in Integrated Design Process Technology (IDPT'02), 2002.
- [Barlow et al. 1975] R. E. Barlow, J. B. Fussel and N. D. Singpurwalla, Reliability and Fault Tree Analysis, Society for Industrial and Applied Mathematics, 1975.
- [Bechta Dugan et al. 1992] J. Bechta Dugan, S. J. Bavuso and M. A. Boyd, “Dynamic Fault-Tree Models for Fault-Tolerant Computer Systems”, IEEE Trans. on Reliability, 41 (3), pp.363-377, September 1992.
- [Bechta Dugan & Lyu 1995] J. Bechta Dugan and M. R. Lyu, “Dependability Modeling for Fault-Tolerant Software and Systems”, in Software Fault-tolerance (M. R. Lyu, Ed.), pp.47-80, John Wiley & Sons Ltd., 1995.
- [Beder et al. 2000] D. M. Beder, A. Romanovsky, B. Randell, C. R. Snow and R. J. Stroud, “An Application of Fault-tolerance Patterns and Coordinated Atomic Actions to a Problem in Railway Scheduling”, ACM SIGOPS Operating Systems Review, 34 (4), pp.21-31, 2000.
- [Béounes et al. 1993] 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”, in 23rd IEEE Int. Symposium on Fault Tolerant Computing, (Toulouse, France), pp.668-673, 1993.
- [Bernardi et al. 2001] S. Bernardi, C. Bertinello, S. Donatelli, G. Franceschinis, R. Gaeta, M. Gribaudo and A. Horvath, “GreatSPN in the new millenium”, in Tool Session of 9th Int. Workshop on Petri Nets and Performance Models, (Aachen, Germany), 2001.
- [Bernardi & Donatelli 2003] S. Bernardi and S. Donatelli, “Building Petri Net Scenarios for Dependable Automation Systems”, in 10th Int. Workshop on Petri Nets and Performance Models, (Urbana, IL, USA), pp.72-83, IEEE CS-Press, 2003.

- [Bernardi et al. 2002] S. Bernardi, S. Donatelli and J. Merseguer, "From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models", in 3rd Int. Workshop on Software and Performance, (Rome, Italy), pp.35-45, ACM Press, 2002.
- [Bernardi & Merseguer 2007] S. Bernardi and J. Merseguer, "A UML Profile for Dependability analysis of Real-Time Embedded Systems", in Workshop on Software and Performance, (Buenos Aires, Argentina), pp.115-124, 2007.
- [Bézivin 2006] J. Bézivin, "Model-driven Engineering: An Emerging Technical Space", in Generative and Transformational Techniques in Software Engineering (R. Lammel, J. Saraiva and J. Visser, Eds.), 4143/2006, LNCS, pp.36-64, Springer-Verlag Berlin / Heidelberg, 2006.
- [Binns & Vestal 2004] P. Binns and S. Vestal, "Hierarchical composition and abstraction in architecture models", in 18th IFIP World Computer Congress, ADL Workshop, (Toulouse, France), pp.43-52, 2004.
- [Blakemore 1989] A. Blakemore, "The Cost of Eliminating Vanishing Markings from Generalized Stochastic Petri Nets." in 3rd Int. Workshop on Petri Nets and Performance Models, (Kyoto, Japan), IEEE-CS Press, 1989.
- [Blanquart et al. 2006] J.-P. Blanquart, A. Rossignol and D. Thomas, "Toward Model-Based Engineering for Space Embedded Systems and Software", in 3rd European Congress on Embedded Real Time Software, (Toulouse, France), 2006.
- [Bobbio & Trivedi 1986] A. Bobbio and K. S. Trivedi, "An Aggregation Technique for the Transient Analysis of Stiff Markov Chains", IEEE Trans. on Computers, C-35 (9), pp.803-814, 1986.
- [Bolch et al. 1998] G. Bolch, G. Greiner, H. de Meer and K. Trivedi, Queueing Networks and Markov Chains, John Wiley and Sons, 1998.
- [Bondavalli et al. 2001] 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", Int. Journal of Computer Systems - Science & Engineering, 16 (5), pp.265-275, 2001.
- [Bondavalli et al. 1999a] A. Bondavalli, M. D. Cin, D. Latella and A. Pataricza, "High-level Integrated Design Environment for Dependability (HIDE)", in 5th International Workshop on Object-oriented Real-time Dependable Systems, pp.87-92, 1999a.
- [Bondavalli et al. 1999b] A. Bondavalli, I. Mura and K. S. Trivedi, "Dependability Modelling and Sensitivity Analysis of Scheduled Maintenance Systems", in 3rd European Dependable Computing Conference (EDCC-3), (A. Pataricza, J. Hlavicka and E. Maehle, Eds.), (Prague, Czech Republic), pp.7-23, Springer, 1999b.
- [Borrel 1996] M. Borrel, Interactions entre composants matériel et logiciel de systèmes tolérant aux fautes - Caractérisation - Formalisation - Modélisation - Application à la sûreté de fonctionnement du CAUTRA, LAAS-CNRS, Thèse de doctorat, N°96001, 1996.
- [Brown 2004] A. Brown, An Introduction to Model-driven Architecture, <http://www.ibm.com/developerworks/rational/library/3100.html>, IBM, February 2004.
- [Brown & Wallnau 1996] A. Brown and K. Wallnau, "Engineering of Component-Based Systems", in 2nd IEEE Int. Conf. on Engineering of Complex Computer Systems, (Montreal, Canada), pp.414-422, 1996.
- [Bruffa & Rugina 2007] M. Bruffa and A. E. Rugina, A Library Implementing Propagation Rules defined in the AADL Error Model Annex LAAS-CNRS, N°07001, February 2007.
- [Budinsky et al. 2004] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick and T. Grose, Eclipse Modeling Framework, Eclipse Series, 680p., Addison-Wesley, 2004.
- [Canals et al. 2002] A. Canals, Y. Cassaing, J. A., L. Pomies and E. Roblet, "How to Use the NEPTUNE Technology in the Modelling Process?" in Conference on Ata System In Aerospace, (Dublin, Ireland), 2002.

- [Chen et al. 2007] D. Chen, M. Törngren and L. H., Advancing Traffic Efficiency and Safety through Software Technology (ATESST) Deliverable D.2.2.1 - Elicitation of Representative and Relevant Analysis and V&V Techniques ATESST Contract Number 2004-026976, 2007.
- [Chiola & Donatelli 1991] G. Chiola and S. Donatelli, "GSPNs versus SPNs: What is the Actual Role of Immediate Transitions?" in 4th Int. Workshop on Petri Nets and Performance Models, (Los Alamitos, CA, USA), pp.20-30, IEEE-CS Press, 1991.
- [Ciardo & Miner 1999] G. Ciardo and A. Miner, "A Data Structure for the Efficient Kroneker Solution of GSPNs", in 8th Int. Workshop on Petri Nets and Performance Models, (Zaragoza, Spain), pp.22-31, IEEE Computer Society Press, 1999.
- [Ciardo & Trivedi 1993a] G. Ciardo and K. S. Trivedi, "Decomposition Approach to Stochastic Reward Net Models", *Performance Evaluation*, 18 (1), pp.37-59, 1993a.
- [Ciardo & Trivedi 1993b] G. Ciardo and K. S. Trivedi, "SPNP: The Stochastic Petri Net Package (Version 3.1)", in 1st Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'93), (San Diego, CA, USA), pp.390-391, 1993b.
- [Czarnecki & Helsen 2003] K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches", in Workshop on Generative Techniques in the Context of Model-Driven Architecture of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, (Anaheim, CA, USA), 2003.
- [de Lemos 2006] R. de Lemos, "Idealised Fault Tolerant Architectural Element", in Int. Conf. on Dependable Systems and Networks, Workshop on Architecting Dependable Systems, (Philadelphia, PA, USA), pp.76-81, 2006.
- [Deavours et al. 2002] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders and P. G. Webster, "The Mobius Framework and its Implementation", *IEEE Transactions on Software Engineering*, 28 (10), pp.956-969, October 2002 2002.
- [Debruyne et al. 2004] 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.
- [Delanote et al. 2007] D. Delanote, S. Van Baelen, W. Joosen and Y. Berbers, "Using AADL in Model-driven Development", in IEEE-SEE Int. Workshop on UML and AADL, Int. Conf. on Engineering Complex Computer Systems, (Auckland, New Zealand), 2007.
- [Elkoutbi et al. 2002] M. Elkoutbi, M. Bennani, R. K. Keller and M. Boulmalef, "Real-time system specifications based on UML Scenarios and Timed Petri Nets", in the 2nd IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'02), (Morocco), pp.362-366, 2002.
- [Elkoutbi & Keller 1998] M. Elkoutbi and R. K. Keller, "Modeling Interactive Systems with Hierarchical Colored Petri Nets", in Conference on High Performance Computing, (Boston, USA), 1998.
- [Farines et al. 2003] 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", in 27th IFAC/IFIP/IEEE Workshop on Real Time Programming, (Zielona Gora, Poland), 2003.
- [Feiler et al. 2004] P. H. Feiler, D. P. Gluch, J. J. Hudak and B. A. Lewis, "Pattern-Based Analysis of an Embedded Real-time System Architecture", in 18th IFIP World Computer Congress, ADL Workshop, (Toulouse, France), pp.83-91, 2004.
- [Feiler et al. 2006] P. H. Feiler, B. A. Lewis and S. Vestal, "The SAE Architecture Analysis & Design Language (AADL), A Standard for Engineering Performance Critical Systems", in IEEE Conf. on Computer Aided Control System Design, (Munich, Germany), pp.1206-1211, 2006.

- [Feiler & Rugina 2007] P. H. Feiler and A. E. Rugina, Dependability Modeling with the Architecture Analysis and Design Language (AADL), Carnegie Mellon Software Engineering Institute, N°CMU/SEI-2007-TN-043, 2007.
- [Fernandez Briones et al. 2006] J. Fernandez Briones, M. de Miguel, J. P. Silva and A. Alonso, “Integration of Safety Analysis and Software Development Methods”, in 1st Int. Conf. on System Safety Engineering pp.275-284, 2006.
- [Fota et al. 1999] N. Fota, M. Kâaniche and K. Kanoun, “Incremental Approach for Building Stochastic Petri Nets for Dependability Modeling”, in Statistical and Probabilistic Models in Reliability (D. C. Ionescu and N. Limnios, Eds.), pp.321-335, Birkhäuser, 1999.
- [Giese et al. 2004] H. Giese, M. Tichy and D. Schilling, “Compositional Hazard Analysis of UML Component and Deployment Models”, in SAFECOMP, pp.166-179, 2004.
- [Goseva Popstojanova & Trivedi 2001] K. Goseva Popstojanova and K. Trivedi, “Architecture-based Approach to Reliability Assessment of Software Systems”, Performance Evaluation, 45 (2-3), pp.179-204, 2001.
- [Hirel et al. 2000] C. Hirel, R. Sahner, X. Zang and K. Trivedi, “Reliability and performability modeling using SHARPE 2000”, in 11th Int. Conf. on Computer Performance Evaluation: Modelling Techniques and Tools, (Schaumburg, IL, USA), pp.345-349, Springer-Verlag, 2000.
- [Hugues et al. 2007] 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.
- [Huszerl et al. 2002] G. Huszerl, I. Majzik, A. Pataricza, K. Kosmidis and M. D. Cin, “Quantitative Analysis of UML Statechart Models of Dependable Systems”, The Computer Journal, 45 (3), pp.260-277, 2002.
- [Islam & Devarakonda 1996] N. Islam and M. Devarakonda, “An Essential Design Pattern for Fault-Tolerant Distributed State Sharing”, Communications of the ACM, 39 (10), pp.65-74, 1996.
- [ISO/IEC 2005] ISO/IEC, Software and Systems Engineering - High-level Petri Nets, Part 2: Transfer Format, International Standard 15909-2 WD Version 0.9.0, June 2005.
- [Jarraya et al. 2007] Y. Jarraya, A. Soeanu and M. Debbabi, “Automatic Verification and Performance Analysis of Time-Constrained SysML Activity Diagrams”, in 14th IEEE Conf. and Workshops on the Engineering of Computer-Based Systems, (Tucson, AZ, U.S.A.), pp.515-522, 2007.
- [Joshi et al. 2007] A. Joshi, S. Vestal and P. Binns, “Automatic Generation of Static Fault Trees”, in Workshop on Architecting Dependable Systems of The 37th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks, (Edinburgh, UK), 2007.
- [Jouault & Kurtev 2005] F. Jouault and I. Kurtev, “Transforming Models with ATL”, in Model Transformaion in Practice Workshop at ACM/IEEE International Conference on Model-driven Engineering Languages and Systems (Montego Bay, Jamaica), 2005.
- [Kalnins et al. 2005] A. Kalnins, J. Barzdins and E. Celms, “Model Transformation Language MOLA”, in Model Diven Architecture (U. Asmann, M. Aksit and A. Rensink, Eds.), 3599/2005, LNCS, pp.62-76, Springer, 2005.
- [Kanoun & Borrel 2000] 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.
- [Kanoun et al. 1999] 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.

- [Kazman et al. 1999] R. Kazman, M. Barbacci, M. Klein, J. Carriere and S. G. Woods, "Experience with Performing Architecture Tradeoff Analysis", in 21st Int. Conf. on Software Engineering (Los Angeles, CA, USA), pp.54-63, 1999.
- [Kehren et al. 2004] C. Kehren, C. Seguin, P. Bieber, C. Castel, C. Bournol, J.-P. Heckmann and S. Metge, "Architecture Patterns for Safe Design", in Int. Complex and Safe Systems Engineering, (Arcachon, France), 2004.
- [King & Pooley 1999] P. King and R. Pooley, "Using UML to Derive Stochastic Petri Net Models", in 15th annual UK Performance Engineering Workshop, pp.45-56, 1999.
- [Klein et al. 1999] M. H. Klein, R. Kazman, R. Bass, J. Carriere, M. Barbacci and H. Lipson, "Attribute-Based Architecture Styles", in 1st Working IFIP Conf. on Software Architecture, (San Antonio, TX, USA), (P. Donohe, Ed.), pp.225-244, 1999.
- [Kwiatkowska et al. 2005] M. Kwiatkowska, G. Norman and D. Parker, "Quantitative Analysis with the Probabilistic Model Checker PRISM", *Electronic Notes in Theoretical Computer Science*, 153 (2), pp.5-31, 2005.
- [Laprie et al. 1990] J.-C. Laprie, J. Arlat, C. Béounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures", *IEEE Computer*, 23 (7), pp.39-51, 1990.
- [Laprie et al. 1995] J.-C. Laprie, J. Arlat, C. Béounes and K. Kanoun, "Architectural Issues in Software Fault-tolerance", in *Software Fault-tolerance* (M. R. Lyu, Ed.), pp.47-80, John Wiley & Sons Ltd., 1995.
- [Laprie et al. 1996] J.-C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac and P. Thévenod, *Guide de la Sûreté de Fonctionnement*, Cépaduès Editions, Toulouse, 1996.
- [Laprie & Kanoun 1996] J.-C. Laprie and K. Kanoun, "Handbook of Software Reliability and System Reliability", in *Software Reliability Engineering* (M. R. Lyu, Ed.), pp.27-69, Computing McGraw-Hill, 1996.
- [López-Grao et al. 2002] J. P. López-Grao, J. Merseguer and J. Campos, "Performance Engineering based on UML & SPN's: A software performance tool", in 17th Int. Symposium on Computer and Information Sciences, (Orlando, Florida, USA), 2002.
- [Lu et al. 2005] S. Lu, W. A. Halang, H. W. Schmidt and R. Gumzej, "A Component-Based Approach to Specify Hazards in the Design of Safety-Critical Systems", in 3rd IEEE Int. Conf. on Industrial Informatics, (Perth, Australia), pp.680-685, 2005.
- [Majzik & Bondavalli 1998a] I. Majzik and A. Bondavalli, "Automatic Dependability Modeling of Systems Described in UML", in *International Symposium on Software Reliability Engineering (ISSRE)*, 1998a.
- [Majzik & Bondavalli 1998b] I. Majzik and A. Bondavalli, "Automatic Dependability Modeling of Systems Described in UML", in *Int. Symposium on Software Reliability Engineering (ISSRE)*, pp.29-30, 1998b.
- [Majzik & Bondavalli 1998c] I. Majzik and A. Bondavalli, "On high-level dependability modeling in HIDE", N°HIDE/T1.2/PDCC/4/v1, 1998c.
- [Majzik et al. 2003] I. Majzik, A. Pataricza and A. Bondavalli, "Stochastic dependability analysis of system architecture based on uml models", in *Architecting Dependable Systems, LNCS 2677, Lecture Notes in Computer Science* (C. G. R. De Lemos, and A. Romanovsky, Ed.), pp.219-244, Springer-Verlag, Berlin, Heidelberg, New York, 2003.
- [Medvidovic & Taylor 2000] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, 26 (1), pp.70-93, 2000.



- [Merseguer & Campos 2004] J. Merseguer and J. Campos, “Software Performance Modeling using UML and Petri Nets”, in *Lecture Notes in Computer Science* 2965, pp.265-289, 2004.
- [Mitton & Holton 2000] P. Mitton and R. Holton, “PEPA Performability Modelling using UML statecharts”, in *16th UK Performance Engineering Workshop*, (N. T. J. Bradley, Ed.), pp.19-33, 2000.
- [OMG 2007a] OMG, SysML Specification, <http://www.omg.org>, April 2007a.
- [OMG 2007b] OMG, Unified Modelling Language Specification: version 2.1.1, <http://www.omg.org>, February 2007b.
- [Pai & Bechta Dugan 2002] G. J. Pai and J. Bechta Dugan, “Automatic Synthesis of Dynamic Fault Trees from UML System Models”, in *13th Int. Symposium on Software Reliability Engineering*, (Annapolis, USA), pp.243-254, 2002.
- [Papadopoulos & McDermid 1999] Y. Papadopoulos and J. A. McDermid, “Hierarchically Performed Hazard Origin and Propagation Studies”, in *SAFECOMP*, pp.139-152, 1999.
- [Peterson 1981] J. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, Upper Saddle River, NJ, 1981.
- [Radjenovic & Paige 2006] A. Radjenovic and R. Paige, “Architecture Description Languages for High-Integrity Real-Time Systems”, *IEEE Software*, 23 (2), pp.71-79, March/April 2006.
- [Randell & Xu 1995] B. Randell and J. Xu, “The Evolution of the Recovery Block Concept”, in *Software Fault-tolerance* (M. R. Lyu, Ed.), pp.1-21, John Wiley & Sons Ltd., 1995.
- [ReSIST 2006] ReSIST, D12: Resilience-building Technologies: State of Knowledge, ReSIST: Resilience for Survivability in IST, A European Network of Excellence, Contract number 026764, 2006.
- [Rugina et al. 2006a] A. E. Rugina, P. H. Feiler, K. Kanoun and M. Kaâniche, *Software Dependability Modeling Using An Industry-Standard Architecture Description Language*, LAAS-CNRS Research Report, N°06558, 2006a.
- [Rugina et al. 2006b] A. E. Rugina, K. Kanoun and M. Kaâniche, “An Architecture-based Dependability Modeling Framework using AADL”, in *10th IASTED Int. Conf. on Software Engineering and Applications*, (Dallas, U.S.A.), pp.222-227, 2006b.
- [Rugina et al. 2006c] A. E. Rugina, K. Kanoun and M. Kaâniche, “Modélisation de la sûreté de fonctionnement à partir du langage AADL”, in *15ème Congrès de Maîtrise des Risques et de Sûreté de Fonctionnement*, (Lille, France), 2006c. This article was awarded the best research paper award and was also published in the Special Issue Publication of the French Institute for Risk Management and Dependability, 2006.
- [Rugina et al. 2007] A. E. Rugina, K. Kanoun and M. Kaâniche, “A System Dependability Modeling Framework using AADL and GSPNs”, in *Architecting Dependable Systems IV* (R. de Lemos, C. Gacek and A. Romanovsky, Eds.), 4615, LNCS, pp.14-38, Springer-Verlag, 2007.
- [SAE-AS5506 2004] SAE-AS5506, *SAE Architecture Analysis and Design Language (AADL)*, International Society of Automotive Engineers, November 2004.
- [SAE-AS5506/1 2006a] SAE-AS5506/1, *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1, Annex C: AADL Meta-Model and Interchange Formats*, International Society of Automotive Engineers, June 2006a.
- [SAE-AS5506/1 2006b] SAE-AS5506/1, *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1, Annex E: Error Model Annex*, International Society of Automotive Engineers, June 2006b.
- [Saldhana & Shatz 2000] J. A. Saldhana and S. M. Shatz, “UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis”, in *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, (Chicago), pp.103-110, 2000.

- [Shaw & Garlan 1994] M. Shaw and D. Garlan, Characteristics of Higher-Level Languages for Software Architecture, Carnegie Mellon University, N° Technical Report CMU-CS-94-210, 1994.
- [Singhoff et al. 2005] F. Singhoff, J. Legrand, L. Nana and L. Marcé, “Scheduling and Memory Requirements Analysis with AADL”, in SIGAda Int. Conf. on Ada, (Atlanta, GE, USA), pp.1-10, 2005.
- [Slaby & Baker 2006] J. Slaby and S. Baker, “Domain-Specific Languages for Enterprise DRE System QoS”, IEEE Computer February 2006.
- [Sokolsky et al. 2006] O. Sokolsky, I. Lee and D. Clarke, “Scedulability Analysis of AADL Models”, in 20th Parallel and Distributed Processing Symposium, (Rhodes Island, Greece), 2006.
- [Tichy et al. 2004] M. Tichy, D. Schilling and H. Giese, “Design of Self-Managing Dependable Systems with UML and Fault-tolerance Patterns”, in Workshop on Self-healing Systems, 1st SIGSOFT Workshop on Self-managed Systems, (Newport Beach, CA, USA), pp.105-109, 2004.
- [Trivedi et al. 1994] K. S. Trivedi, B. R. Haverkort, A. Rindos and V. Mainkar, “Techniques and Tools for Reliability and Performance Evaluation: Problems and Perspectives”, in 7th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, (L. N. i. C. Sciences, Ed.), pp.1-24, Springer, 1994.
- [Vestal 1998] S. Vestal, MetaH User's Manual, Honeywell Technology Center, 1998.
- [Viehl et al. 2006] A. Viehl, T. Schönwald, O. Bringmann and W. Rosenstiel, “Formal Performance Analysis and Simulation of UML/SysML Models for ESL Design”, in Conf. on Design, Automation and Test in Europe, (Munich, Germany), pp.1-6, 2006.
- [Waddington & Lardieri 2006] D. Waddington and P. Lardieri, “Model-Centric Software Development”, IEEE Computer February 2006.
- [Wallace 2005] M. Wallace, “Modular Architectural Representation and Analysis of Fault Propagation and Transformation”, in Formal foundations of Embedded Systems and Component-Based Software Architectures Workshop, (Edinburgh), 2005.
- [Yau & Cheung 1975] S. S. Yau and R. C. Cheung, “Design of Self-Checking Software”, in Int. Conf. Reliable Software, (Los Angeles, U.S.A.), pp.450-457, 1975.
- [Zarras & Issarny 2000] A. Zarras and V. Issarny, “Assessing Software Reliability at the Architectural Level”, in 4th International Software Architecture Workshop, 2000.
- [Zarras et al. 2004] A. Zarras, P. Vassiliadis and V. Issarny, “Model-Driven Dependability Analysis of Web Services”, in 6th International Symposium on Distributed Objects and Applications (DOA 2004), 2004.



# Dependability modeling and evaluation – From AADL to stochastic Petri nets

## ABSTRACT

Performing dependability evaluation along with other analyses at architectural level allows both predicting the effects of architectural decisions on the dependability of a system and making tradeoffs. Thus, both industry and academia focus on defining model driven engineering (MDE) approaches and on integrating several analyses in the development process. AADL (Architecture Analysis and Design Language) has proved to be efficient for architectural modeling and is considered by industry in the context presented above. Our contribution is 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. In this context, we also provide a set of reusable modeling patterns for fault tolerant architectures. The AADL dependability model is transformed into a GSPN (Generalized Stochastic Petri Net) by applying model transformation rules. We have implemented an automatic model transformation tool. The resulting GSPN can be processed by existing tools to obtain dependability measures. The modeling approach is illustrated on a subsystem of the French Air Traffic Control System.

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



**AUTEUR :** Ana-Elena RUGINA

**TITRE :** Modélisation et évaluation de la sûreté de fonctionnement –  
De AADL vers les réseaux de Petri stochastiques

**DIRECTEUR DE THESE :** Mme. Karama KANOUN

**LIEU ET DATE DE SOUTENANCE :** Toulouse, le 19 novembre 2007

---

## RÉSUMÉ

Conduire des analyses de sûreté de fonctionnement conjointement avec d'autres analyses au niveau architectural permet à la fois de prédire les effets des décisions architecturales sur la sûreté de fonctionnement du système et de faire des compromis. Par conséquent, les industriels et les universitaires se concentrent sur la définition d'approches d'ingénierie guidées par des modèles (MDE) et sur l'intégration de diverses analyses dans le processus de développement. AADL (Architecture Analysis and Design Language) a prouvé son aptitude pour la modélisation d'architectures et ce langage est actuellement jugé efficace par les industriels dans de telles approches. Notre contribution est 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'évaluation de mesures de sûreté de fonctionnement comme la fiabilité et la disponibilité. Nous proposons une approche itérative de modélisation. Dans ce contexte, nous fournissons un ensemble de sous-modèles génériques réutilisables pour des architectures tolérantes aux fautes. 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. Nous avons implémenté un outil de transformation automatique. Le RdPSG résultant peut être traité par des outils existants pour obtenir des mesures de sûreté de fonctionnement. L'approche est illustrée sur un ensemble du Système Informatique Français de Contrôle de Trafic Aérien.

---

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

---

**DISCIPLINE ADMINISTRATIVE :** Systèmes informatiques

---