République Tunisienne
Ministère de l'Enseignement Supérieur,
de la Recherche Scientifique

Université de Sfax
École Nationale d'Ingénieurs de Sfax

Ecole Doctorale
Sciences et Technologies

Thèse de *DOCTORAT*

Ingénierie des Systèmes Informatiques
N° d'ordre: 101/2016

THESE

Présentée à

L'École Nationale d'Ingénieurs de Sfax

En vue de l'obtention du

DOCTORAT

Dans la discipline Informatique
Ingénierie des Systèmes Informatiques

Par

Wafa GABSI MASMOUDI

(Mastère en Nouvelles Technologies des Systèmes Informatiques Dédiés)

Tolérance aux pannes pour les systèmes temps-réel distribués: de la modélisation à l'implantation

Soutenue le 04 Avril 2017, devant le jury composé de :

M.	**Faiez GARGOURI** (Professeur)	Président
M.	**Mohamed KAANICHE** (Directeur de recherche)	Rapporteur
M.	**Adel MAHFOUDHI** (Maître de conférences)	Rapporteur
M.	**Mohamed ABID** (Professeur)	Examinateur
M.	**Mohamed JMAIEL** (Professeur)	Directeur de thèse

Remerciements

C'est avec un grand plaisir que je réserve cette page en signe de gratitude et de profonde reconnaissance à tous ceux qui ont bien voulu apporter l'assistance nécessaire au bon déroulement de ce travail. Je tiens à remercier tous les membres du jury, pour l'intérêt qu'ils ont apporté à mes travaux de thèse et d'accepter d'être membre de son Jury.

— Faiez Gargouri, professeur à l'*Institut supérieur d'informatique et de multimédia de Sfax*, qui m'a fait l'honneur de présider ce jury.

— Mohamed Abid, professeur à l'*Ecole Nationale d'Ingénieurs de Sfax* (ENIS), qui m'a fait le grand plaisir d'être examinateur de cette thèse.

— Mohamed Kaâniche, directeur de recherche au *Laboratoire d'Analyse et d'Architecture des Systèmes de Toulouse*, (Toulouse, France).

— Adel Mhafoudhi, maître de conférences à la *Faculté des Sciences de Sfax*.

— Mohamed Jmaiel, professeur à l'ENIS et directeur général du *Centre de Recherche en Numérique de Sfax*

— Bechir Zalila, maître assistant à l'ENIS.

Je remercie tout particulièrement Monsieur Mohamed Kaaniche et Monsieur Adel Mahfoudhi pour l'honneur qu'ils m'ont fait en acceptant d'être rapporteur de ma thèse. Je les remercie vivement pour leurs commentaires avisés.

J'exprime ma profonde reconnaissance à Monsieur Mohamed Jmaiel et Monsieur Bechir Zalila pour avoir dirigé mes travaux de thèse, pour leur patience, leurs conseils pour m'orienter à entreprendre les bonnes décisions et notamment pour le temps qu'ils ont consacré pour moi. J'espère être à la hauteur de leur confiance. Qu'ils trouvent dans ce travail le fruit de leurs efforts et l'expression de ma profonde gratitude.

J'adresse mes remerciements aussi à toutes mes collègues qui ont contribué au développement du compilateur *AspectAda*. Je remercie vivement Rahma Bouaziz, Hana Mkaouar, Aida Boukhriss et Fatma Kacem pour leurs efforts.

Je remercie sincèrement tous les membres du laboratoire ReDCAD, permanents et doctorants. 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 laboratoire a été extrêmement enrichissante. Il m'est particulièrement

agréable de remercier mes collègues, avec qui j'ai partagé des moments inoubliables, pour leur disponibilité et leur aide précieuse. Ma pensée va également vers les doctorants plus anciens et les nouveaux doctorants parmi lesquels mes étudiants.

Mes remerciements vont également à Monsieur Jérôme Hugues, maître de conférences à l'*Institut Supérieur de l'Aéronautique et de l'Espace* de Toulouse, France, pour m'avoir accueilli pendant un mois. Je le remercie sincèrement pour les échanges enrichissants que nous avons eus.

Il est indispensable de ne pas rater cette occasion, dans ces moments inoubliables de ma vie, pour exprimer ma reconnaissance à ma famille et mes amis pour leur encouragement permanent, leur patience et leur compréhension.

Dédicaces

A mon père Samir et ma mère Salwa,
A mon mari Majdi et mes deux anges Molka et Mahdi,
A mon beau-père Mohamed et ma belle mère Fahima,
A mes frères Sofien, Lotfi et Mohamed,
A l'âme de ma sœur,
A ma grand-mère Zoubaida,
A toute ma famille Gabsi,
A toute ma famille Masmoudi,
A toute ma famille Belguith,
A mes amis,
A tous ceux qui comptent pour moi,
Je dédie ce mémoire,

Wafa Gabsi Ep Masmoudi

Table des matières

LISTE DES FIGURES

Liste des tableaux

LISTE DES LISTINGS

Liste des algorithmes

Introduction générale

Les systèmes temps-réel distribués sont présents dans plusieurs domaines d'application, tels que l'aéronautique, la télécommunication, l'automobile, etc. Devant assurer plus de fonctionnalités, ces systèmes deviennent de plus en plus complexes. Les outils et les techniques de développement logiciels ont changé considérablement et ils continuent toujours à changer et à évoluer [CtK02]. Ceci est du à l'apparition de nouvelles contraintes liées à l'allocation des ressources mémoires, les contraintes temps-réel et les besoins en sûreté de fonctionnement qui deviennent plus exigeants en vue de satisfaire les besoins des utilisateurs.

Dans ce contexte, la sûreté de fonctionnement des systèmes logiciels devient un défis majeur dans le domaine informatique. Dans ce type de systèmes, quelles que soient les précautions prises, l'occurrence de pannes est parfois inévitable à cause de conditions internes ou externes comme les erreurs humaines, la malveillance, le vieillissement du matériel, etc. Une faute peut entraîner de graves conséquences telles que la perte de temps, d'argent ou même de vies humaines. D'où la naissance de besoins de techniques garantissant la sûreté de fonctionnement *(Dependability)* tout au long du cycle de développement. Les techniques de tolérance aux pannes permettent de construire des systèmes robustes préservant la continuité du fonctionnement malgré l'occurrence de pannes. En revanche, mêmes s'ils sont fondamentaux, les besoins de tolérance aux pannes sont orthogonaux aux préoccupations fonctionnelles des applications. Pour des raisons d'efficacité et de réduction des coûts, les solutions de tolérance aux pannes s'orientent vers une conception et implantation indépendantes des préoccupations fonctionnelles.

Le développement orienté aspect de logiciel prône la séparation de préoccupations. En se basant sur une telle séparation, la programmation orientée aspect est un paradigme qui cherche à rendre le processus d'évolution logicielle plus simple. Elle préconise la modularisation et la réutilisation des préoccupations non-fonctionnelles. Le succès de ce paradigme dans divers domaines d'application, a encouragé son utilisation dans d'autres domaines plus complexes notamment les applications temps-réel critiques. Par ailleurs, ces applications présentent des contraintes temporelles et des exigences de ressources et de vérifications plus difficiles à satisfaire. De ce fait, la garantie du bon fonctionnement et le respect des contraintes fonctionnelles et non-fonctionnelles deviennent des défis majeurs pour les

applications temps-réel critiques.

Ce travail de thèse s'articule autour du développement des systèmes temps-réel distribués tolérants aux pannes. Plus précisément, l'objectif est de définir un processus de développement pour ces types de systèmes allant de la modélisation jusqu'à l'implantation. Ce processus devra permettre également l'utilisation des techniques de vérifications et de preuves tôt dans le cycle de vie, d'une part, et le respect des contraintes temporelles, d'autre part, pour garantir le bon fonctionnement de ces systèmes.

Dans ce qui suit, nous définissons les quatre problématiques de cette thèse, les objectifs à atteindre, et enfin l'approche proposée pour les réaliser.

1 Problématiques

En explorant la littérature dans le domaine de conception et de développement des systèmes temps-réel et l'intégration de la tolérance aux pannes dans ce domaine, nous avons dégagé certaines problématiques.

1.1 Développement des systèmes temps-réel

De nos jours, la complexité et le développement des systèmes informatiques deviennent de plus en plus importants en termes de besoins temps-réel, de distribution, de dynamisme et de sûreté de fonctionnement. La tolérance aux pannes nécessite d'appliquer certaines actions de reconfiguration dynamique au niveau de l'exécution pour le rétablissement du système. Ceci rend l'estimation du temps d'exécution difficilement prévisible et donc entraîner le non déterminisme dans le cas des systèmes temps-réel, dont l'exactitude des résultats ne dépend pas uniquement des résultats obtenus mais aussi du temps de leurs production. Par ailleurs, la manière dont les mécanismes de tolérance aux pannes sont implantés peut compromettre le bon fonctionnement des systèmes temps-réel à savoir l'introduction de l'indéterminisme. En explorant la littérature, nous avons constaté l'absence de travaux offrant un processus de développement de bout en bout des systèmes temps-réel distribués tolérants aux pannes. Les travaux existants se concentrent plutôt sur l'une des phases du processus indépendamment des autres sans séparer la tolérance aux pannes des autres préoccupations. Ils se sont intéressés à la modélisation de la tolérance aux pannes, la vérification de certaines propriétés pour la mesure des attributs de la sûreté de fonctionnement ou à son implantation sans lier les différentes phases.

1.2 Modélisation manuelle de la réplication

La tolérance aux pannes est un moyen de la sûreté de fonctionnement, dont la mise en place est basée sur une technique de redondance et de multiplicité des composants, appelée réplication. Les langages permettant la modélisation des mécanismes de la tolérance aux

pannes existants, reposent sur une réplication manuelle et explicite de composants. En cas d'un grand nombre de répliques ou des composants à répliquer, le modèle devient complexe et requiert un effort considérable de la part du concepteur. En outre, une telle réplication manuelle engendre un risque d'erreurs et augmente significativement le temps de conception. De plus, les approches à base de réplication adoptent généralement un seul style de réplication, la réplication active ou passive. Cependant, une application bien déterminée pourra requérir les deux styles à la fois.

1.3 Génération de code

La production des systèmes tolérants aux pannes revient à implanter les mécanismes nécessaires pour la détection des pannes au cours de l'exécution et le rétablissement de ces systèmes à partir de leurs spécifications. Une grande partie du code pourra être générée automatiquement à partir du modèle. Toutefois, les travaux existants ne s'intéressent pas à la génération automatique du code. Ils se concentrent sur l'implantation manuelle et dispersée de la tolérance aux pannes indépendamment du modèle. Ceci rend la tâche d'implantation beaucoup plus complexe et risque d'engendrer une incohérence entre le système en cours d'exécution et sa spécification.

1.4 Violation des contraintes temps-réel

Du fait des contraintes temps-réel exigeantes, le développement des systèmes temps-réel tolérants aux pannes n'est pas une tâche triviale. Ce type de systèmes requiert un code respectant certaines restrictions afin d'éviter l'apparition de constructions jugées dangereuses d'après la littérature. Parmi les restrictions les plus connues, nous citons l'interdiction de l'allocation dynamique de mémoire puisqu'il est fort probable qu'elle induit au non déterminisme [Pua02]. Particulièrement, l'intégration de la tolérance aux pannes, qui nécessite des actions de reconfiguration dynamique, dans l'implantation des systèmes temps-réel peut compromettre le déterminisme. Il s'agit d'insérer dans le code de l'application des constructions pouvant violer les contraintes temps-réel et échappant à tout contrôle du développeur.

2 Objectifs

Pour remédier aux problèmes soulevés dans la section précédente, nous proposons dans le cadre de ce travail de thèse de guider la conception et le développement de la tolérance aux pannes dans les systèmes temps-réel distribués dynamiquement reconfigurables. Nous visons ainsi proposer un processus de développement des systèmes, respectant les contraintes temps-réel et préservant des propriétés temporelles, qui assure la séparation de la tolérance aux pannes des autres préoccupations. Plus précisément, il s'agit, d'intégrer dès la phase de

modélisation les éléments de la tolérance aux pannes nécessaires et de permettre la génération séparée de code.

2.1 Proposition d'un processus de développement rigoureux

Il s'agit de définir un processus de conception et de développement de la tolérance aux pannes pour les applications temps-réel distribuées. Ce processus inclut la modélisation, la vérification et la génération de code des préoccupations fonctionnelles aussi bien que celles non fonctionnelles, en particulier la tolérance aux pannes. Cette génération aide le développeur pour aboutir à une application prête à être exécutée sans violer les contraintes temps-réel et les contraintes fonctionnelles.

L'ingénierie dirigée par les modèles (IDM [Bro04]), proposée par l'Object Management Group (OMG) [1], est une démarche qui se concentre sur la notion de modèle. L'IDM propose de séparer les spécifications fonctionnelles d'un système des spécifications liées à la plate-forme d'implantation. La mise en œuvre de cette démarche repose sur les raffinements et les transformations entre des modèles. C'est à la base de cette technique que le concept de modélisation avec différents niveaux d'abstraction a vu le jour. Visant l'interopérabilité entre les différents systèmes, la réduction du coût de développement et l'augmentation de l'évolutivité, nous proposons l'adoption de cette démarche pour la définition de notre processus de développement allant de la modélisation, passant par une phase d'analyse et de vérification pour arriver à une génération automatique du code. Pour garantir une meilleure modularité et réutilisation et faciliter la maintenabilité de ces systèmes sur tous les niveaux, nous envisageons séparer la tolérance aux pannes des préoccupations métiers dès la phase de modélisation tirant profit des techniques de preuves et de vérification possibles à ce stade.

2.2 Modélisation automatisée et séparée de la réplication

L'introduction d'une technique de modélisation automatisée de la réplication est le second objectif de cette thèse. Vis-à-vis des problèmes notés suite à la réplication manuelle, nous visons assister le concepteur pour faciliter la modélisation de la réplication surtout dans le cas d'un grand nombre de répliques ou de composants à répliquer. Toujours dans le but de la séparation des préoccupations, nous souhaitons concevoir la réplication d'une façon modulaire indépendamment des préoccupations fonctionnelles. Il s'agit donc de guider le concepteur pour encapsuler les paramètres de réplication afin d'appliquer une approche générique régissant la gestion des répliques.

1. Object Management Group, www.omg.org

2.3 Génération automatique de code

La génération automatique et massive de code permet de réduire non seulement le risque d'erreur par le développeur mais aussi le temps de développement de l'application. Une grande partie du code peut être générée automatiquement. C'est à la base des intergiciels que la génération de code est mise en œuvre dans le développement des systèmes temps-réel. Outre ses fonctionnalités, un intergiciel doit satisfaire à son tour certaines contraintes. Il doit offrir des routines assurant, d'une part, la reconfiguration dynamique et, d'autre part, la supervision et la cohérence [Zal08]. Pour répondre aux besoins des systèmes temps-réel, les intergiciels dédiés sont conçus pour respecter ces contraintes, plus particulièrement le déterminisme. Dans ce contexte, nous souhaitons utiliser un intergiciel dédié pour la génération automatique de code pour les systèmes temps-réel. Nous souhaitons également l'adapter ou l'étendre pour la génération du code tolérant aux pannes tout en préservant la séparation des préoccupations et le respect des contraintes temps-réel.

2.4 Respect des contraintes temps-réel

Afin de garantir le respect des contraintes temps-réel, nous souhaitons utiliser un langage de programmation qui est dédié pour le développement temps-réel. Pour intégrer des éléments de tolérance aux pannes au niveau du code, nous devons éviter d'insérer, dans le code de l'application, toutes constructions pouvant violer les contraintes temps-réel et échappant à tout contrôle du développeur. Ainsi, nous préservons le déterminisme des applications produites à l'aide de notre processus de développement proposé.

3 Contributions

Pour atteindre nos objectifs, nos contributions reposent sur quatre parties qui sont brièvement décrites dans cette section.

3.1 Processus de développement

La première contribution de cette thèse consiste à la définition d'un processus de développement pour les applications temps-réel distribuées tolérantes aux pannes. Suivant la démarche *Model Driven Approach (MDA)*, ce processus inclut la modélisation (erreurs, propagation, rétablissement), la vérification (mesures de la sûreté de fonctionnement et de propriétés non-fonctionnelles) et la génération du code des applications en se basant sur des raffinements et des transformations appliquées sur des modèles. Étant donné que la tolérance aux pannes est une préoccupation non-fonctionnelle, nous proposons sa séparation des préoccupations métiers durant toutes les phases du processus. Ainsi, partant d'un modèle architectural, suivant des étapes de raffinements et de transformations de modèles, nous

arrivons à aider le développeur pour aboutir à une application prête à être exécutée sans violer les contraintes temps-réel et les contraintes fonctionnelles.

Nous avons choisi le langage *Architecture Analysis & Design Language (AADL)* [SAE12]), un langage de description d'architecture concret qui a prouvé sa puissance lors de la modélisation des systèmes temps-réel critiques comme dans le domaine aéronautique. Parmi les avantages de ce langage, c'est son caractère extensible à travers des propriétés ou des annexes. En particulier, l'annexe *Error Model Annex (EMA)* [SAE15], permet la description d'un modèle d'erreur au niveau architectural. Cette annexe permet non seulement la modélisation des différents types d'erreurs mais aussi la détection, la propagation et le recouvrement de ces erreurs. De plus, il existe des travaux qui sont basés sur cette annexe pour effectuer des mesures et des analyses de sûreté de fonctionnement [Rug08] et d'autres qui permettent de générer du code à partir des modèles AADL. Pour ces raisons, nous avons adopté le langage AADL et son annexe EMA pour la modélisation de la tolérance aux pannes dans les systèmes temps-réel.

3.2 Modélisation de la réplication

Pour gérer automatiquement la réplication, nous avons proposé une approche basée sur les transformations de modèles permettant le passage d'un modèle AADL réduit vers un modèle AADL enrichi avec les répliques. Il s'agit d'élever le niveau d'abstraction de notre modèle et d'automatiser la gestion de la réplication. Cette approche tire profit des extensions offertes par le langage AADL. Elle permet la diminution et la gestion de la complexité du système et la réduction du temps de conception en se basant sur la séparation des préoccupations. Pour ce faire, nous avons défini un ensemble de propriétés à savoir la description de la réplication, le style de réplication et le nombre de répliques. Puis, en se basant sur un ensemble de règles de transformation, nous générons un modèle cohérent intégrant les différentes répliques en établissant les connexions et en générant les comportements pour toutes les répliques.

3.3 Génération de code

Pour la génération automatique de code, nous avons adopté la démarche MDA non seulement pour la génération du code fonctionnel mais aussi du code tolérant aux pannes pour les systèmes temps-réel. À partir du modèle AADL, nous proposons l'utilisation des intergiciels dédiés pour la production du code fonctionnel. Quant à la génération du code tolérant aux pannes, nous adoptons la programmation orientée aspect dès le niveau modèle pour sa séparation des préoccupations fonctionnelles. À partir de l'annexe EMA, nous raffinons le modèle tolérant aux pannes par l'ajout d'aspects architecturaux donnant ainsi lieu à la génération d'un modèle intermédiaire indépendant de la plate-forme. Il s'agit d'un modèle AADL enrichi avec des aspects *Aspect Oriented Extension For AADL*

(AO4AADL) [LKZJ13], une extension d'aspect pour le langage AADL. Ce dernier sera à son tour transformé pour avoir le code final spécifique à la plate-forme. Afin de garder la séparation des préoccupations, le code final sera produit en un langage d'aspect pour être tissé avec le code fonctionnel.

3.4 Adaptation d'un langage d'aspect pour les systèmes temps-réel

Pour la génération du code des systèmes tolérants aux pannes, nous avons proposé d'intégrer la programmation orientée aspect dans le but de séparer les préoccupations. En particulier, nous avons proposé l'utilisation d'un langage d'aspect pour la production du code tolérant aux pannes. Précisément, nous avons sélectionné le langage Ada et son extension d'aspect AspectAda pour la génération du code fonctionnel et tolérant aux pannes respectivement. Afin de garantir le respect des contraintes temps-réel, nous avons étudié le langage AspectAda [PC05] existant et testé en particulier le fonctionnement de son compilateur prototype et le déroulement de l'opération de tissage toujours par rapport aux systèmes temps-réel. Cette étude de l'existant nous a amené à explorer les limites de ce langage, discuter les solutions possibles et finalement proposer une nouvelle architecture du compilateur.

4 Organisation du document

Ce document est organisé selon le plan suivant.

Le chapitre 1 présente, en premier lieu, les notions de base et une terminologie des techniques abordées dans notre domaine. Nous présentons d'abord les systèmes temps-réel, leurs caractéristiques et leurs contraintes. Puis, nous détaillons la tolérance aux pannes étant un des moyens de la sûreté de fonctionnement. En particulier, nous présentons ses menaces et ses principes. Ensuite, nous nous intéressons aux architectures logicielles et plus particulièrement à la démarche MDA et les Architecture Description Languages (ADLs). Enfin, nous décrivons le paradigme de programmation orientée aspect, ses concepts de base et ses avantages.

Dans le chapitre 2, nous donnons un état de l'art des technologies traitant les thématiques de notre travail de thèse. Tout d'abord, nous faisons un tour d'horizon sur les différents langages de modélisation dédiés pour les systèmes temps-réel distribués. Ensuite, nous nous focalisons sur les travaux visant la modélisation et l'implantation des techniques de la tolérance aux pannes notamment la réplication et le consensus. Puis, nous nous intéressons aux approches proposées dans le but d'intégrer la programmation orientée aspect dans le développement des systèmes temps-réel. Enfin, nous exposons les différentes approches connexes à nos travaux. À partir de cette étude, nous positionnons nos travaux.

Le chapitre 3 est consacré à détailler les grandes lignes de la solution que nous proposons dans ce travail de thèse. Nous définissons un processus de développement générique dé-

dié aux systèmes temps-réel distribués tolérants aux pannes. Nous détaillons les différentes étapes du processus allant de la modélisation jusqu'à l'implantation. Puis, nous justifions le choix du langage AADL pour la modélisation des préoccupations fonctionnelles ainsi que celle non fonctionnelles.

Le chapitre 4 détaillera les concepts de base du langage AADL et particulièrement ses aspects dont nous avons besoin pour la spécification, la configuration et l'analyse d'applications temps-réel distribuées tolérantes aux pannes. Il donne par la suite une vue d'ensemble sur les annexes utilisées dans le suite de ce mémoire.

Le chapitre 5 est consacré par la suite à introduire les raffinements qui s'articulent autour du processus générique DP4FTRTS défini dans le chapitre 3. Ces raffinements sont effectués suite au choix du langage AADL comme langage de description d'architecture adopté et son annexe EMA pour la modélisation de tolérance aux pannes. Par la suite, nous donnons une vue d'ensemble sur l'extension proposée dans le cadre de notre thèse pour le support de la réplication automatique. Puis, nous expliquons brièvement l'approche proposée pour la génération du code fonctionnel et tolérant aux pannes et les choix des langages dédiés. Finalement, nous nous focalisons sur les langages choisis pour l'implantation des systèmes temps-réel et de la tolérance aux pannes et les contraintes qui doivent être respectées à ce niveau. Ceci nous mènera vers le besoin d'un langage d'aspect adapté pour le développement temps-réel.

Le chapitre 6 se focalise sur notre approche proposée pour la modélisation et la gestion automatique de la réplication. Il s'agit de détailler la démarche à suivre suivie des différentes propriétés définies. Puis, nous présentons les règles de transformation définies pour générer le modèle enrichi par les répliques. Ensuite, nous nous intéressons à l'extension de la suite d'outils Ocarina pour supporter notre approche.

Le chapitre 7 présente les étapes suivies pour mener à bien la génération du code des deux préoccupations fonctionnelles et transversales, en particulier, la tolérance aux pannes. Nous détaillons d'abord les générateurs existants utilisés pour la génération du code fonctionnel. Ensuite, nous décrivons notre approche proposée pour la génération du code tolérant aux pannes à partir du modèle d'erreur. Puis, nous nous focalisons sur les règles de transformations régissant la génération de code.

Le chapitre 8 présente l'adaptation du langage AspectAda existant pour le développement temps-réel. Nous commençons par présenter les limites et les défauts du langage notamment par rapport aux besoins de la tolérance aux pannes et aux contraintes des systèmes temps-réel. Ensuite, nous nous attardons sur les solutions proposées pour remédier à ces problèmes. Enfin, nous présentons les différentes parties de la nouvelle architecture proposée pour le compilateur et l'implantation de chacune d'entre elles.

Deux cas d'étude sont donnés dans le chapitre 9 pour vérifier l'adéquation de nos solutions avec les problématiques posées. La première est consacrée à l'illustration des étapes du processus proposé et la seconde pour valider l'approche d'extension et d'adaptation du

langage AspectAda. Enfin, nous concluons ce mémoire de thèse dans le dernier chapitre et nous présentons les perspectives possibles d'amélioration et d'extension de notre travail.

Première partie

Étude Théorique

1

Terminologie et Notions de base

1.1 Introduction

Les besoins en tolérance aux pannes ne cessent d'accroître du fait de l'utilisation croissante des systèmes informatiques. La livraison de services corrects dans les brefs délais devient un défi dans tous les domaines et en particulier dans le domaine temps-réel critique. De plus, face aux exigences sélectives et continues dans ce domaine, la description des architectures logicielles reste toujours une opération complexe pour le concepteur. Dans le but de faciliter la conception des systèmes logiciels critiques et de réduire les coûts de maintenance, le développement orienté aspect basé sur la séparation des préoccupations a été introduit sur les deux niveaux de conception et de codage.

Dans ce chapitre, nous présentons les concepts de base liés à nos contributions. Nous commençons, tout d'abord, par la présentation des systèmes temps-réel en citant quelques définitions de ces systèmes suivie par l'énumération de leurs caractéristiques les plus importantes et les contraintes de développement qui y sont liées. Ensuite, nous nous intéressons à la définition de la tolérance aux pannes comme un moyen de sûreté de fonctionnement, ses menaces et ses principes. Puis, nous détaillons les défis de l'architecture logicielle, la démarche MDA et nous présentons les ADLs. Enfin, nous définissons la programmation orientée aspect, ses notions de base suivies par les apports de ce paradigme qui nous ont encouragé à l'utiliser pour implanter la tolérance aux pannes.

1.2 Systèmes temps-réel

Les systèmes temps-réel connaissent de nos jours un essor considérable et dans différents domaines d'application tels que le transport, l'avionique, la télécommunication, etc. Dans

cette section, nous donnons une définition claire des systèmes temps-réel en exposant leurs caractéristiques majeures et nous présentons les contraintes associées à ce type de systèmes.

1.2.1 Définitions

Un système temps-réel [CI99] est défini comme un système dont le résultat est considéré correct si et seulement s'il satisfait les attentes fonctionnelles en effectuant correctement les traitements demandés et s'il est délivré sans dépasser les échéances fixées. Ainsi, un système temps-réel doit fournir une réponse logiquement correcte avant le dépassement des échéances. Afin de répondre à ses besoins, un système temps-réel est souvent composé de plusieurs sous-systèmes respectant des contraintes temporelles plus ou moins strictes. Dans ce contexte, les systèmes temps-réel sont classés en deux catégories dans la mesure de tolérer ou non certains retards.

— **Système temps-réel dûr** (*Eng. Hard real-time*)
Un système est dit temps-réel dûr (ou critique) s'il doit obligatoirement respecter les échéances prévues. Pour ce type de systèmes, un retard peut provoquer de graves conséquences financières, environnementales ou même humaines [Rus94].
Exemples : contrôleur de vitesse pour voiture, contrôleur du train d'atterrissage d'un avion...

— **Système temps-réel mou** (*Eng. Soft real-time*)
Un système est dit temps-réel mou (ou non critique) s'il accepte à l'avance certains retards par rapport aux échéances prévues toujours dans la mesure du possible. *Exemples :* systèmes de visioconférence, encodeur/décodeur MPEG, ...

Ces deux types de systèmes temps-réel admettent des caractéristiques et des contraintes particulières influençant leurs comportements.

1.2.2 Caractéristiques des systèmes temps-réel

Outre la fiabilité, les systèmes temps-réel doivent satisfaire certaines propriétés comme le déterminisme et la prévisibilité de façon à respecter les exigences temporelles spécifiées.

— **Le déterminisme :**
Un système temps-réel est dit déterministe si et seulement si les temps d'exécution de ses tâches sont bornés [CI99]. On peut donc déterminer si le système est déterministe avant sa construction. Cette propriété doit être satisfaite dans le contexte temps-réel.

— **La prévisibilité :**
Cette propriété permet de déterminer à l'avance les paramètres liés aux contraintes temporelles des tâches à savoir les priorités, les temps d'exécution, les pires temps d'exécution, les périodes, le taux d'utilisation du processeur, etc [CI99]. Ces paramètres

aident par la suite au choix de l'algorithme d'ordonnancement le plus adéquat dans le but d'avoir un comportement prévisible du système.

— **La fiabilité :**
Elle consiste à offrir un service dont nous pouvons faire confiance [ALRL04]. C'est la capacité du système de répondre aux exigences fonctionnelles et temporelles souhaitées. La fiabilité est fortement liée à la disponibilité, l'intégrité et la sécurité.

1.2.3 Contraintes des systèmes temps-réel

De nos jours, les systèmes temps-réel couvrent différents domaines qualifiés critiques tels que les applications médicales, spatiales et aéronautiques. Plusieurs facteurs internes ou externes aux systèmes peuvent dans ce cas invoquer la violation de leurs contraintes et donc engendrer l'indéterminisme ou l'occurrence de fautes. Parmi ces facteurs, nous citons la variation des temps d'exécution des tâches, les pannes des dispositifs matériels, les fautes de développement ou encore les interruptions auxquelles un système doit rapidement réagir. Dans ce contexte, le développement de ces types de systèmes demande beaucoup plus de précision et de prudence par rapport à ceux classiques. Le développeur doit prendre en considération ces facteurs pour éviter très tôt dans le processus de développement toute sorte d'indéterminisme.
Certaines études approfondies dans le développement temps-réel ont dégagé un ensemble de concepts de programmation nuisant au déterminisme comme l'allocation dynamique de mémoire ou le polymorphisme.

1. **Allocation dynamique de mémoire**

   Cette technique consiste à la fragmentation de la mémoire causée par la dés-allocation et la ré-allocation. Puisque, l'estimation du temps de recherche d'un espace libre dans une mémoire fragmentée est pratiquement imprévisible, l'allocation dynamique peut alors engendrer des problèmes lors des calculs des pires temps d'exécution (*Worst Case Execution Time (WCET)*) des tâches et donc compromettre le déterminisme. Comme la manipulation de la mémoire est assez délicate, alors le programmeur peut facilement commettre des erreurs engendrant des fuites de mémoire. Ces fuites peuvent à leurs tours provoquer le gaspillage des ressources en mémoire qui sont généralement limitées.

   Pour ces raisons, l'allocation dynamique de mémoire est interdite dans le contexte des systèmes temps-réel [Pua02].

2. **Aiguillage dynamique et polymorphisme**

   Dans [HH03], les auteurs discutent la compatibilité du paradigme orienté-objet avec les systèmes temps-réel. Basé sur le mécanisme d'aiguillage dynamique, le paradigme orienté objet introduit les concepts d'héritage et de polymorphisme. Il s'agit de

construire automatiquement des tables appelées *tables d'aiguillage* à travers les compilateurs orientés objet. L'analyse de ces tables n'est pas supportée et donc elles échappent à tout contrôle du développeur ce qui rend le flux de contrôle non connu à l'avance. Ceci compromet fortement le déterminisme de l'application requis dans les systèmes temps-réel. Afin de remédier à ces problèmes, les auteurs de [HH03] interdisent l'utilisation du polymorphisme et de l'aiguillage dynamique en les considérant non compatibles avec les systèmes temps-réel.

1.3 Sûreté de fonctionnement et tolérance aux pannes

Pour faire face aux pannes pouvant survenir et menacer le bon fonctionnement des systèmes, la sûreté de fonctionnement définit certaines techniques pour agir lors de l'occurrence de l'une de ses menaces. Dans cette section, nous définissons la sûreté de fonctionnement, ses attributs et ses entraves classées selon divers critères. Aussi, nous nous intéressons aux différents moyens de la sûreté de fonctionnement et en particulier la tolérance aux pannes.

1.3.1 Définitions et notions de base

La **sûreté de fonctionnement** (*Eng, Dependability*) d'un service se définit par la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre [ALRL04]. C'est la capacité d'éviter les pannes des services dont la fréquence et le danger ne sont pas acceptables.

Différents attributs caractérisent la sûreté de fonctionnement. Ces attributs expriment les propriétés qui sont attendues à partir d'un système répondant à des besoins de la sûreté de fonctionnement [Dub13]. Parmi ces attributs, trois sont considérés comme principaux :

— **Disponibilité** : un système est dit disponible s'il est prêt à fournir un service correct. Dans de nombreux cas, nous nous sommes intéressés, non seulement à la probabilité de défaillance, mais aussi aux nombres d'échecs possibles et en particulier, au temps nécessaire pour réparer un système en panne. La disponibilité est l'attribut permettant d'exprimer l'intervalle de temps durant lequel le système fonctionne correctement.

— **Fiabilité :** c'est le fait de fournir un service correct d'une manière continue. Une haute fiabilité est nécessaire pour les systèmes devant fonctionner sans interruption, comme dans le cas d'un stimulateur cardiaque, ou lorsque la maintenance ne peut être effectuée à cause de consultation ou d'accès impossible du système comme dans le cas des engins spatiaux. Dans ce type de systèmes, on prévoit le temps nécessaire pour achever la mission demandée. La manière dont le temps est spécifié varie sensiblement en fonction de la nature du système en question. Pour ces raisons, la fiabilité est aussi considérée une fonction du temps.

— **Sécurité innocuité** : c'est l'absence de conséquences catastrophiques sur l'utilisateur et l'environnement (*Eng. Safety*). Dans ce type de système, une défaillance peut entraîner des désastres environnementaux ou des pertes de vies humaines.

Outre ces attributs, nous citons la confidentialité, l'intégrité et la maintenabilité :

— **Confidentialité** : c'est le faite de ne pas dévoiler des informations sans autorisation.

— **Intégrité** : c'est l'absence d'altérations inappropriées de l'état du système.

— **Maintenabilité :** c'est la capacité de supporter des modifications et des réparations.

Selon le type de l'application, un ou plusieurs de ces attributs doivent être satisfaits pour assurer l'évaluation adéquate du comportement du système. Par exemple, dans un distributeur automatique de billet (DAB), la proportion de temps où le système est capable de fournir le service demandé (c'est-à-dire la disponibilité du système) est une mesure importante. Pour un patient doté d'un stimulateur cardiaque, le fonctionnement continu de l'appareil est une question de vie ou de mort. Dans ce cas, la fiabilité du système est cruciale. Toutefois, dans une centrale nucléaire, la sécurité et la fiabilité deviennent toutes les deux d'une importance supérieure.

1.3.2 Menaces de la sûreté de fonctionnement

Un service est délivré correct lorsqu'il assure les fonctionnalités demandées du système [ALRL04]. Cependant, le service peut dévier par rapport à sa spécification suite à l'occurrence des menaces de la sûreté de fonctionnement qui sont les fautes, les erreurs et les défaillances. Une caractéristique commune des trois entraves est qu'elles nous donnent un message indiquant le mal fonctionnement du système. Toutefois, chacune de ces menaces survient à un niveau différent.

— **La faute :** elle consiste à un défaut physique ou logique du matériel ou du logiciel. C'est toute cause, événement, action ou circonstance pouvant provoquer une dégradation du service.

— **L'erreur :** elle se traduit par une valeur incorrecte dans le système susceptible de causer la défaillance.

— **La défaillance ou la panne :** c'est la déviation du système par rapport à sa spécification pour une période de temps spécifiée. Un système est ainsi défaillant s'il est mal spécifié c'est-à-dire sa spécification ne décrit pas convenablement son fonctionnement ou s'il ne fonctionne pas conformément à sa spécification.

Les fautes causent l'apparition des erreurs qui sont à leur tours des raisons de défaillances. Considérons par exemple, une centrale électrique dans laquelle un système contrôlé par ordinateur est responsable de la surveillance des températures différentes de

plantes, des pressions et d'autres caractéristiques physiques. La rupture du capteur rapportant la vitesse à laquelle la turbine principale tourne, amène le système à envoyer plus de vapeur par rapport à celle requise. Cette erreur cause la sur-vitesse de la turbine. Dans ce cas, le système de sécurité mécanique, arrête la turbine pour éviter qu'elle soit endommagée. Ainsi, il ne développe plus de puissance causant sa défaillance.
La propagation des fautes présente un risque majeur de la sûreté de fonctionnement. En fait, l'erreur est la manifestation de la faute activée sur le système et la défaillance est l'effet d'une erreur sur le service. De plus, étant donné qu'un système informatique est souvent composé de plusieurs sous-systèmes, la défaillance d'un sous-système peut créer ou activer une faute dans un autre sous-système ou dans le système global.

1.3.3 Classification des fautes

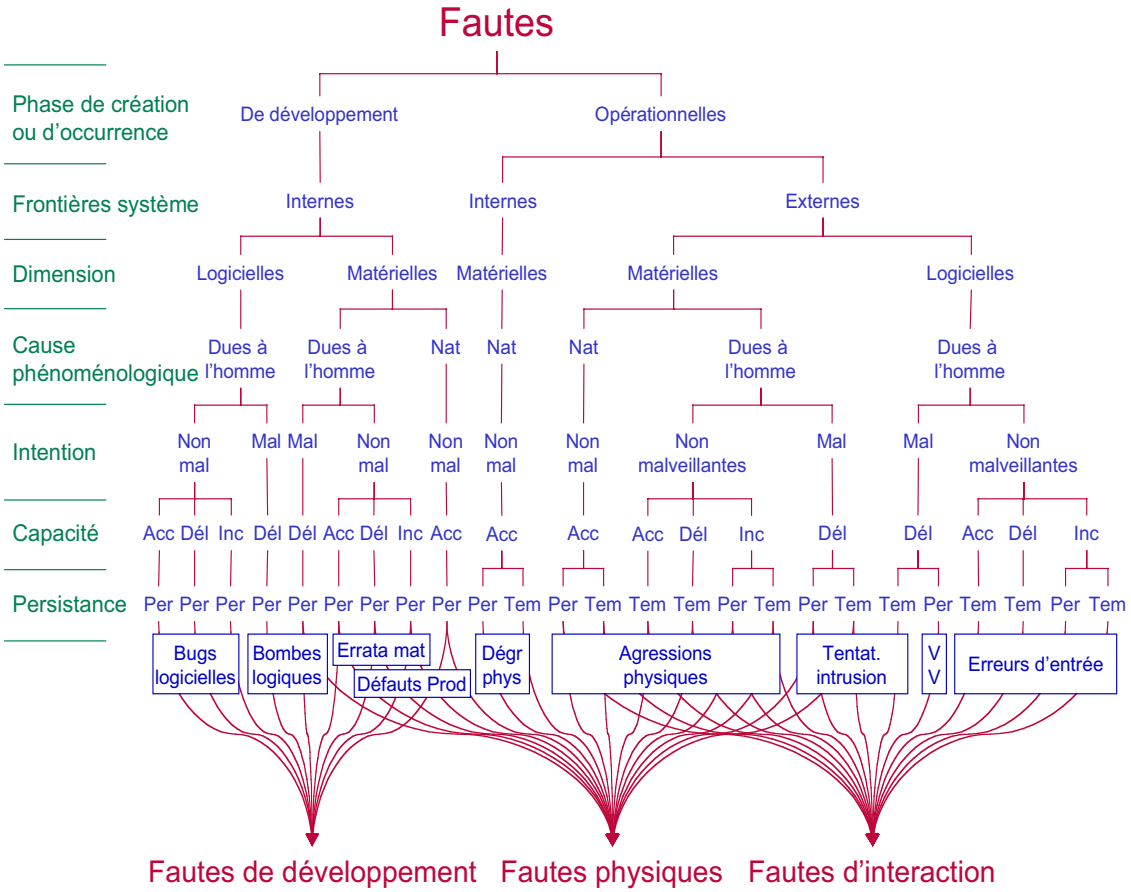

FIGURE 1.1 – Classification des fautes [ALRL04]

Les fautes susceptibles d'affecter un système sont classées d'une manière orthogonale selon plusieurs critères comme le montre la figure 1.1. Selon [ALRL04], les critères de classi-

fication adoptés sont les suivants :

1. Phase de création ou d'occurrence : une faute est considérée de développement si elle est créée durant le développement du système, y compris la génération de code. Si le système a commencé son exécution, une faute est alors considérée opérationnelle puisqu'elle survient durant la fourniture du service au cours de la vie opérationnelle.
2. Frontières de système : une faute est soit externe soit interne localisée à l'intérieur des frontières du système.
3. Dimension : une faute est matérielle si elle se manifeste dans le matériel. Par contre, si la faute affecte le logiciel, les programmes ou les données, elle est considérée logicielle.
4. Causes phénoménologiques : selon ce critère, une faute est soit physique due à des phénomènes naturels, sans intervention humaine directe soit humaine.
5. Intention : une faute est de nature intentionnelle ou non. La faute intentionnelle (dite encore malveillante) est introduite par un humain dans le but de causer des dommages au système.
6. Capacité : une faute est délibérée si elle est due à des mauvaises décisions, à savoir les actions prévues qui sont fausses et qui causent des fautes. Au contraire, une faute est considérée non délibérée si elle est due à des actions involontaires à propos desquelles le développeur ou le concepteur est inconscient.
7. Persistance : une faute transitoire est une faute dont la présence est temporellement bornée contrairement aux fautes permanentes dont la présence est continue.

Tous ces types de fautes peuvent être évitées ou détectées par divers moyens acceptant la présence ou éliminant l'occurrence de fautes.

1.3.4 Moyens de la sûreté de fonctionnement

Les différents attributs de la sûreté de fonctionnement sont garantis par l'élimination des erreurs de l'état du système voire la prévention avant l'occurrence de quelques types en se basant sur différents moyens. Ces moyens sont regroupés en quatre catégories :

— **La prévention de fautes (Fault Prevention) :** elle concerne toutes les parties de l'ingénierie et couvre des aspects qui sont d'intérêt direct en ce qui concerne la fiabilité et la sécurité. Il s'agit d'empêcher l'occurrence ou l'introduction de fautes [ALRL04]. A titre d'exemple, la prévention des fautes de développement est un objectif évident pour les méthodologies de développement, pour les logiciels (la modularisation, l'utilisation des langages de programmation fortement typés...) aussi bien que pour le matériel (les règles de conception).

— **La tolérance aux fautes (Fault Tolerance) :** il s'agit d'assurer la fourniture du service malgré l'occurrence de fautes [ALRL04]. La tolérance a pour objectif d'éviter les pannes à travers la détection d'erreurs et la récupération ou le recouvrement du système.

— **La prévision de fautes (*Fault forecasting)* :** elle consiste à estimer la présence, la création et les conséquences des fautes [ALRL04]. Il s'agit d'évaluer le comportement du système par rapport à l'apparition ou l'activation de fautes. L'évaluation comporte deux aspects. Le premier aspect est l'évaluation qualitative qui sert à identifier et classer les modes de défaillance ou les combinaisons d'événements (par exemple les défaillances des composants) qui conduiraient à des défaillances du système. Le deuxième aspect est l'évaluation de la mesure de satisfaction de certains attributs en termes de probabilités. Elle est pour cela qualifiée d'évaluation quantitative ou probabiliste.

— **L'élimination de fautes (*Fault Removal)* :** elle consiste à détecter et supprimer les fautes avant la production d'erreur [ALRL04]. Pendant la phase de développement du cycle de vie d'un système, l'élimination de fautes se fait en trois étapes : la vérification, le diagnostic et la correction. Cependant, lors de l'utilisation d'un système, l'élimination de fautes se base sur la maintenance corrective ou préventive. La maintenance corrective a pour objectif d'éliminer les fautes qui ont été signalées après avoir produit une ou plusieurs erreurs. Quant à la maintenance préventive, elle vise à découvrir et enlever les fautes avant qu'elles puissent provoquer des erreurs durant le fonctionnement normal du système.

La prévention et la tolérance aux fautes rendent le système capable de fournir un service dont on peut faire confiance. Tandis que l'élimination et la prévision visent à atteindre la confiance dans cette capacité en justifiant que les spécifications fonctionnelles, de la sécurité et de la sûreté de fonctionnement sont adéquates et que le système est susceptible de les reconnaître. La tolérance et la prévision de fautes sont deux moyens qui acceptent la présence de fautes contrairement à la prévention et l'élimination qui empêchent leurs occurrences.

Dans le reste de ce mémoire, nous nous intéressons uniquement à la tolérance aux pannes qui est définie par la capacité d'un système à continuer de fournir un service acceptable malgré la présence d'erreurs.

1.3.5 Principes de la tolérance aux fautes

Dans le cadre de cette thèse, nous nous intéressons à la tolérance aux pannes, un des moyens de la sûreté de fonctionnement. Les techniques de la tolérance aux pannes présentent un moyen essentiel pour bâtir des services mis en jeu dans le cadre des applications critiques. Combinée avec les autres moyens de la sûreté de fonctionnement dans chaque phase du cycle de développement (conception et implantation), la tolérance aux pannes peut mieux conduire à la construction d'un système fiable et sûr en se basant sur deux principes : la détection et le rétablissement [ACD+06].

1. **La détection** de l'erreur consiste à l'identification de sa présence. La détection est soit **concomitante** réalisée lors de l'exécution normale du service ou **préemptive** effectuée

lors de l'interruption du service.

2. **Le rétablissement ou le recouvrement** consiste à transformer l'état erroné du système en un état dispensé de l'erreur détectée et des fautes qui peuvent être activées de nouveau. La tolérance aux pannes est distinguée de la maintenance par le fait que cette dernière requiert la participation d'un agent externe. Le rétablissement se fait par :

   (a) **Le traitement de l'erreur** à travers l'élimination des erreurs de l'état du système. Ce traitement est suivi par une maintenance corrective visant à éliminer les fautes qui ont été déjà isolées. Après la détection d'erreur, un rétablissement par reprise ou par poursuite est invoqué à la demande. Tandis que le traitement par compensation peut être appliqué soit à la demande ou de manière systématique, à des moments ou des événements prédéterminés, indépendamment de la présence ou l'absence de l'erreur (détectée).

      i. **Rétablissement par reprise (*Rollback*) :** il s'agit de sauvegarder au fur et à mesure des états du système appelés points de reprise. En cas d'erreur, le système est ramené alors à un point de reprise survenu avant la détection de panne (figure 1.2, (a)).

      ii. **Rétablissement par poursuite (*Forward*) :** le système doit trouver un nouvel état stable exempt de l'erreur détectée (figure 1.2, (b)).

      iii. **Compensation ou masquage :** dès la conception du système, une redondance des composants est mise en place afin de masquer les erreurs détectées (figure 1.2, (c)).

   (b) **Le traitement de fautes** qui a pour but la prévention d'une nouvelle activation de fautes à travers divers mécanismes :

      i. **Le diagnostic :** permet d'identifier et d'enregistrer la ou les causes de l'erreur détectée en termes de localisation et de temps.

      ii. **La passivation :** vise à assurer l'exclusion logique du ou des composants fautifs du processus d'exécution c'est à dire rendre la faute dormante.

      iii. **La reconfiguration :** réaffecte les tâches aux composants non défaillants ou intègre des composants en réserve.

      iv. **La réutilisation :** contrôle, met à jour et enregistre la nouvelle configuration, les tables et les données du système.

1.3.6 Redondance, Réplication et diversification

Un système tolérant aux pannes, doit continuer à fonctionner sans interruption pendant le processus de réparation. Il doit dans tous les cas éviter les points de défaillances uniques.

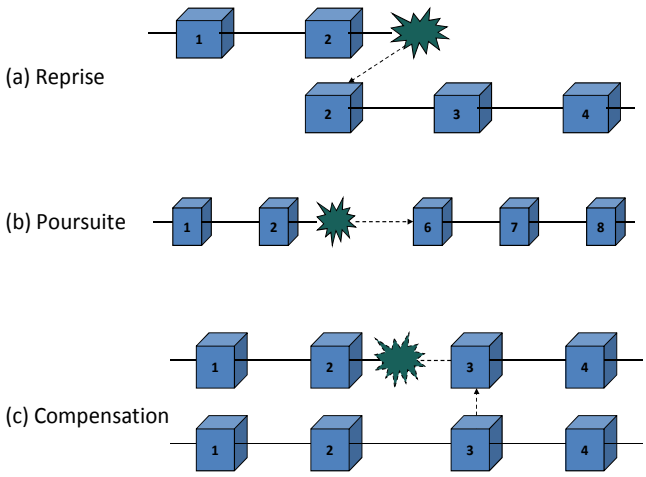

FIGURE 1.2 – Techniques de recouvrement

Parmi les techniques les plus utilisées pour assurer le recouvrement sans interruption en cas de pannes, nous citons la redondance, la réplication et la diversification sur lesquelles les principes de tolérance aux pannes sont basés.

La redondance

Elle consiste à la multiplicité et la répétition des composants logiciels ou matériels. La tolérance aux pannes d'un système fait appel nécessairement à une ou plusieurs formes de redondance affectant chacune une phase de la tolérance aux pannes.

En effet, la redondance peut être classée de plusieurs manières. Selon [ACMF00], la redondance est classée en trois catégories :

— La **redondance temporelle** est utilisée dans l'objectif de recouvrement d'erreur. Dans des cas extrêmes, on peut se baser sur ce type de redondance pour la détection des erreurs.

— La **redondance spatiale** est appliquée dans le but de la détection des erreurs, d'élimination des fautes (identification) ou de recouvrement d'erreur.

— La **redondance d'information**, elle est utilisée pour la détection d'erreur, l'évaluation des dommages, et la récupération d'erreur.

Selon [SRG95], trois autres formes de redondance sont distinguées : la redondance analytique, la redondance fonctionnelle et la réplication. La réplication consiste à exécuter exactement des traitements identiques contrairement à la redondance fonctionnelle visant à exécuter des traitements différents pour l'obtention des mêmes résultats. Quant à la redondance

analytique, elle accepte des résultats différents dans la mesure d'une certaine cohérence. Ainsi, parmi ces trois formes, il faut choisir au moins une pour mettre en place un système tolérant aux pannes [GÖ9].

La réplication

La réplication est une technique très connue et utilisée pour la tolérance aux pannes. Elle est définie par la redondance des composants logiciels, matériels ou hybrides. Le déploiement des mêmes unités logicielles sur différentes unités matérielles permet d'atténuer les effets de défaillances des composants uniques et de créer ainsi un système plus fiable et améliorer sa disponibilité. Ainsi, les composants matériels ou logiciels critiques, ou même des systèmes entiers, peuvent être répliqués. Deux stratégies principales de réplication sont distinguées [PVW04] :

— **Réplication active** : pour ce type de réplication, toutes les copies (répliques) jouent un rôle identique. Elles reçoivent les mêmes requêtes, les traitent, et émettent la même séquence de réponses. Dans une architecture client/serveur, les serveurs sont répliqués et le client choisit enfin l'une des réponses fournies (voir figure 1.3). Le protocole de choix de la réponse peut être, dans certains cas, personnalisé. A titre d'exemple, il s'agit d'appliquer un mécanisme de vote majoritaire ou de choisir la moyenne des valeurs retournées. Le mécanisme de vote est mis en œuvre de différentes façons selon le besoin de l'application.

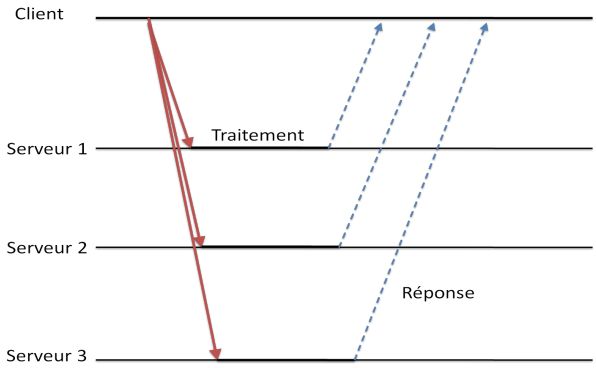

FIGURE 1.3 – Réplication active

— **Réplication Passive** : elle distingue deux comportements différents des composants répliqués. Une réplique est considérée primaire et les autres sont secondaires. Uniquement la copie primaire est chargée de recevoir les différentes requêtes, de les traiter et d'émettre les réponses. En cas d'erreur, les copies secondaires prennent le relais. Dans ce cas, en se basant sur un algorithme d'élection, une copie secondaire est élue pour remplacer la copie primaire défaillante (voir figure 1.4).

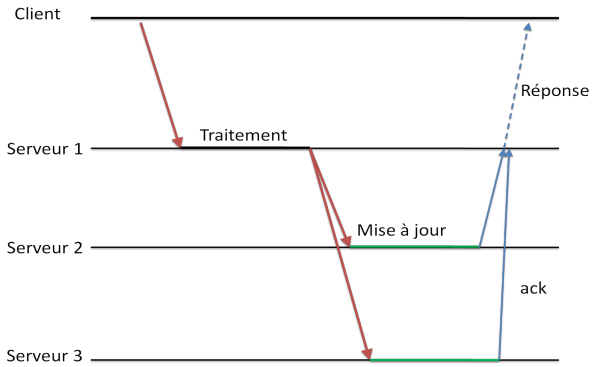

FIGURE 1.4 – Réplication passive

Le choix du style de réplication dépend fortement des besoins de l'application en terme de sûreté de fonctionnement, des ressources disponibles et des contraintes environnementales. Un compromis s'impose entre les différents styles. Pour le cas de la réplication active, toutes les répliques exécutent le même traitement sachant qu'elles admettent le même état interne. Dans ce cas l'application est déterministe. On peut facilement calculer le temps de réponse. En ce qui concerne la réplication passive, il s'agit d'un grand nombre de messages envoyés entre les copies secondaires et la copie primaire. De même, en cas de défaillance, l'élection de la copie secondaire et le replacement de celle primaire nécessite un temps de recouvrement qui introduit dans certains cas l'indéterminisme. Il est à noter que ce style de réplication peut être amélioré pour assurer une synchronisation périodique des états des répliques. Il existe des travaux visant l'optimisation de la période de synchronisation dans le but de garantir une cohérence entre les répliques en consommant moins de ressources. Toutefois, les deux styles de réplication, active et passive, ne peuvent pas répondre à tous les besoins de réplication vis-à-vis des contraintes de ressources, d'environnement et de temps fixées pour l'application. Notons que d'autres styles appelés intermédiaires (semi) ont été proposés pour satisfaire mieux ses besoins comme la réplication semi-passive [DSS98] et semi-active [DTT99].

La réplication **semi-active** est un style de réplication considéré à mis-chemin entre la réplication active et passive. Une ou plusieurs répliques appelées copies secondaires suivent la copie primaire appelée guide. De façon similaire à la réplication active, toutes les répliques reçoivent les mêmes entrées et peuvent les traiter. Cependant, similairement à la réplication passive, seule la réplique primaire est responsable du traitement d'une demande et de la prise des décisions. La réplique primaire traite une demande dès qu'elle la reçoit, alors que les suiveurs attendent la notification émise par le guide pour le traitement d'une demande.

La diversification

La diversification est développée spécifiquement pour tolérer les fautes logicielles de

conception qui découlent des spécifications erronées ou d'un codage incorrect. Elle vise à fournir le même service par le biais d'un modèle ou d'une implantation distincte. Les techniques les plus populaires qui sont basées sur la diversification de la conception sont les blocs de recouvrement (*Recovery Block (RB)*), les blocs de recouvrement distribués (*Distributed Recovery Block (DRB)*) et la programmation en N-Versions (*N-Version Programming (NVP)*).

— **Programmation en N-Versions :** ce concept est similaire à l'approche N-Modular Programming (NMR) appliquée dans le contexte de la tolérance aux pannes matérielles. Cette technique consiste à exécuter en parallèle N (N>=2) programmes fonctionnellement identiques et générés de façon indépendante qui sont appelés *versions* [LA95]. Un vote logique majoritaire est utilisé pour comparer les résultats obtenus par l'ensemble des versions et signaler l'un des résultats qui est probablement correct. La capacité à tolérer des fautes dépend du degré d'indépendance des différentes versions.

— **Blocs de recouvrement (RB) :** cette approche est fondée sur plusieurs variantes d'un logiciel qui sont fonctionnellement équivalentes et déployées selon le modèle de redondance temporelle [PKJ11]. Un test d'acceptation est utilisé pour tester la validité du résultat produit par la version primaire. Si, d'une part, le résultat de la version primaire passe le test d'acceptation, ce résultat est rapporté et l'exécution s'arrête. Si, d'autre part, le résultat de la version primaire échoue au test d'acceptation, une autre version parmi les versions multiples est appelée et le résultat obtenu est vérifié à son tour par le test d'acceptation.

— **Blocs de recouvrement distribués :** c'est une technique de conception typique (sous forme de traitements uniformes parallèles et distribués) pour des blocs de recouvrement fournissant ainsi une tolérance à la fois de certaines pannes logicielles et matérielles pour les systèmes temps-réel [kK95].

Selon le contexte et les pannes à tolérer, l'une des formes de diversification ou la combinaison de deux formes peut être appliquée. Dans certains cas, la tolérance aux pannes est basée plutôt sur la réplication. Tout dépend des besoins en tolérance aux pannes.

1.3.7 Besoins en tolérance aux pannes

Les besoins de tolérance aux pannes ne cessent d'accroître du fait de l'utilisation croissante des systèmes informatiques. La livraison de services corrects dans les brefs délais devient un besoin primordial dans tous les domaines et en particulier dans le domaine informatique. L'occurrence de pannes peut entraîner la non satisfaction des utilisateurs, des dangers menaçant les utilisateurs ou mêmes de graves conséquences telles que la perte d'argent très coûteuse ou la perte de vies humaines. Dans ce contexte, il est impossible d'éviter toutes défaillances. Les communautés de sûreté de fonctionnement cherchent plutôt à minimiser l'occurrence. Malgré tout effort, il y a toujours des défaillances. À titre d'exemple,

nous pouvons citer les chutes et les accidents aériens qui surviennent encore comme celle de *Malaysia Airlines* le 8 mars 2014 et *Boeing 777-200* le 17 juillet 2014 provoquant la perte de plus de 200 passagers.
Les niveaux d'exignence en termes de sûreté de fonctionnement et plus particulièrement de la tolérance aux pannes varient selon le domaine et le type de l'application. Dans le domaine aéronautique, les exigences sont beaucoup plus fortes que celles dans le domaine de télécommunications par exemple. Les dégradations des services critiques engendrent des résultats inacceptables. Des missions critiques peuvent échouer à cause de l'insuffisance ou l'absence de la tolérance aux pannes. Dans d'autres cas, il s'est avéré qu'il est impossible de procéder aux tentatives de réparation et de maintenance en temps-réel. Pour cela, la mise en place de mécanisme de la tolérance aux pannes dès la phase de conception aide énormément à réduire la probabilité d'occurrence de pannes en se basant essentiellement sur la redondance. Cependant, une redondance mal adaptée peut conduire à son tour à des défaillances inattendues ou aux dégradations des performances du système. Une redondance doit être donc appliquée d'une manière adéquate. Une même faute ne doit pas causer la défaillance de tous les composants redondants.

De plus, étant donné que la tolérance aux pannes est basée sur une certaine redondance, elle implique un coût supplémentaire généralement non négligeable surtout lorsqu'il s'agit d'un grand nombre de composants redondants. Un compromis s'impose : si on diminue le risque, on augmente le coût. Il faut donc déterminer les degrés de la sûreté de fonctionnement convenables aux systèmes mis en jeu.

1.4 Architecture logicielle

L'architecture logicielle [KOS06, Cle97, PW92] reflète la structure globale d'un système informatique du point de vue de sa composition en termes de composants et des diverses interactions entre eux. Elle décrit aussi pour chacun de ses composants, son comportement à un haut niveau d'abstraction sans considérer les détails d'implantation. L'architecture logicielle permet également de placer le système dans son environnement extérieur comme l'environnement technologique tout en mettant l'accent sur les relations entre les deux. Cette discipline spécifie la manière dont le système est organisé dans le but d'atteindre ses objectifs et de satisfaire ses besoins. Ainsi, elle est mieux adaptée dans le contexte des systèmes complexes en les décomposant en un ensemble de composants et de sous-composants élémentaires et en spécifiant les interactions existantes entre eux.
L'architecture logicielle intervient aux différents niveaux. Pour décrire l'organisation du système, elle intervient au niveau conceptuel et afin de concevoir son implantation, elle intervient au niveau du développement. Cependant, le système doit être capable d'agir vis-à-vis de certains besoins de changements. À titre d'exemple, nous citons la traduction de nouvelles exigences, la reconfiguration comportementale ou architecturale ou la gestion des

erreurs détectées au niveau de la conception ou l'implantation. Différents modèles de l'architecture logicielle ont été mis en place dans le but de couvrir tous les besoins des systèmes mis en jeu à savoir les architectures orientées services, les architectures n-tiers ou les calculs distribués.

1.4.1 Défis de l'architecture logicielle

Visant à améliorer la qualité de production logicielle, la description d'une architecture logicielle devient un travail assez complexe en matière des objectifs visés et des défis soulignés. Suivant le cycle de développement, l'architecture logicielle prétend éliminer les erreurs de conception ainsi que celles d'implantation partant d'une meilleure conception. De ce fait, la production et l'architecture logicielle partagent un ensemble de défis :

— **La construction incrémentale :** la description d'une architecture logicielle doit permettre aux concepteurs de construire leurs systèmes suivant un processus incrémental partant initialement des préoccupations basiques et considérant au fur et à mesure des préoccupations techniques.

— **La modularité :** surmonter ce défi rend possible la répartition modulaire du développement d'une application sur plusieurs groupes de personnes. Il s'agit de décomposer un système en un ensemble de composants interconnectés permettant ainsi la séparation de la mise en œuvre de chacun d'eux.

— **La réutilisation :** la description de l'architecture logicielle revient à décrire l'ensemble de composants et de connexions établies entre eux via leurs interfaces. Afin de permettre l'utilisation de ces composants dans plusieurs applications, la définition des interfaces associées doit mettre en faveur la contrainte de réutilisation.

— **L'anticipation des changements :** l'architecture logicielle des systèmes à base de composants prend en compte les changements causés par l'environnement externe ou par le système lui-même. Elle doit donc prévoir l'évolution du système et préciser sa variabilité pour effectuer les adaptations nécessaires.

— **L'abstraction :** pour décrire un système indépendant de toute plateforme ou technologie, l'architecture dirigée par les modèles (*MDA*) [Bro04] se base sur la manipulation des modèles définis avec un certain niveau d'abstraction tout au long du cycle de développement.

1.4.2 Démarche MDA

L'architecture dirigée par les modèles, connue encore avec l'acronyme Ingénierie Dirigée par les Modèles (*Model Driven Engineering (MDE)*) [Bro04], proposée par l'OMG [2], est une

2. www.omg.org

démarche qui se concentre sur la notion de modèle pour l'abstraction de toute architecture. Dans ce qui suit, nous définissons cette démarche puis nous expliquons les différentes étapes qu'elles propose.

Définition

Visant l'interopérabilité entre les différents systèmes, la réduction du coût de développement et l'augmentation de l'évolutivité, la MDA propose de séparer les spécifications fonctionnelles d'un système des spécifications liées à la plateforme d'implantation. La mise en œuvre de cette démarche est à la base de modèles et de transformations entre différents modèles. Il s'agit de :

1. Définir une architecture structurée à base de modèles indépendants de toutes plateformes, de systèmes d'exploitation et même d'intergiciels.
2. Effectuer des transformations successives jusqu'à arriver à une architecture spécifique à une plateforme donnée.

Cette approche permet en effet d'assurer toutes les étapes de développement et de standardiser les transitions d'une étape à l'autre. Elle permet donc d'épargner l'effort effectué pendant les deux étapes d'analyse et de conception.

Mise en œuvre de la démarche MDA

L'approche MDA repose sur la définition de modèles et de transitions entre eux. On peut la décomposer en quatre étapes comme l'indique la figure 1.5 :

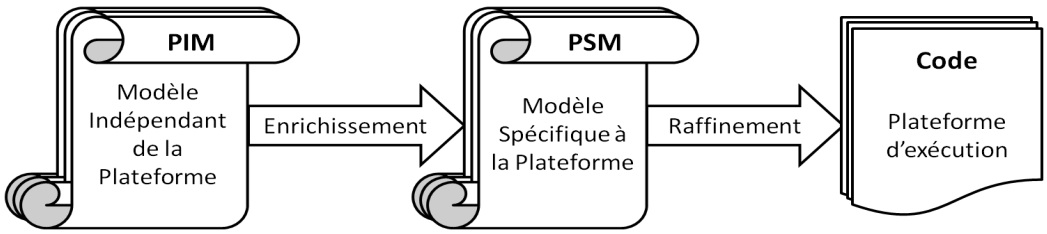

FIGURE 1.5 – Mise en œuvre de la démarche MDA

— **Platform Independent Model (PIM) :** la première étape réside dans la création d'un modèle indépendant de toute plateforme et technologie. Ce modèle, exprimé en Unified Modelling Language (UML), offre la description de la logique métier et du comportement du système sans détailler son déploiement sur une plateforme. Il décrit le fonctionnement des différents composants. Il doit être clair, complet et correct dans le but de faciliter sa compréhension et sa validation par des experts du domaine. Il domine les deux phases d'analyse et de conception.

— **Enrichissement :** la deuxième étape consiste à enrichir, détailler et filtrer le PIM par des informations non relatives à la plateforme. Donc le PIM doit être assez enrichi pour qu'on puisse le spécialiser vers une plateforme. On peut répéter cette étape un nombre indéterminé de fois afin d'améliorer le passage entre modèles.

— **Platform Specific Model (PSM) :** la troisième étape repose sur la sélection d'une plateforme technique et la production du modèle spécifique correspondant PSM. Une fois que le PIM est suffisamment enrichi, on peut alors le transformer en PSM. Cette transformation est basée sur l'ajout de nouvelles informations liées à la plateforme. C'est à ce niveau qu'on passe de la spécification indépendante à celle dépendante.

— **Raffinement :** Il s'agit de raffiner le PSM jusqu'à obtenir une implantation exécutable. On cherche à générer le code adéquat spécifique à la plateforme choisie, et ceci en ajoutant les informations propres à l'exécution et les données de configuration. Cette étape peut être répétée infiniment dans le but d'amélioration et de précision du PSM et donc du code généré.

1.4.3 Langages de description d'architecture

L'architecture logicielle permet de décrire un système comme un ensemble de composants interconnectés à un haut niveau d'abstraction indépendamment de la plateforme technique. Un composant peut à son tour contenir un ensemble de sous-composants qui interagissent. Chacun de ces composants alloue les ressources matérielles nécessaires pour son fonctionnement. Pour couvrir tous ces besoins, les langages de description d'architecture (*ADL*) ont vu le jour. Ces langages offrent un formalisme possédant une syntaxe et une sémantique offrant des abstractions et des mécanismes adaptés à la description de l'architecture logicielle.

Définition d'ADL

Proposés dans les années 90, les ADLs forment une famille de langages visant à décrire une architecture logicielle. Chacun de ces ADLs admet ses propres fonctions, ses objectifs et ses propres caractéristiques. Il offre aux concepteurs un support de modélisation qui prend en considération l'organisation des différentes entités. Généralement, les ADLs permettent à la fois une représentation textuelle et une autre graphique disposant d'outils adjoints. Ils reposent sur un lexique commun et approprié pour tous les acteurs participant à la description architecturale à savoir les concepteurs, les développeurs et les testeurs. Ce sont des formalismes à travers lesquels, nous jouissons de l'avantage de faciliter le contrôle d'application et la réutilisation des composants et d'éliminer toute sorte d'ambiguïté [MT00].

Concept d'ADL

Un ADL est un support de modélisation qui définit l'hiérarchie entre les composants pour décrire l'organisation entre les entités. Les ADLs sont basés sur trois concepts fondamentaux qui sont : les composants, les connecteurs et les configurations [MT00]. Dans la suite, nous décrivons brièvement chacun d'eux.

FIGURE 1.6 – Composants et connecteurs [Zal08]

— **Composant** : C'est une entité matérielle ou logicielle possédant un état et un type jouant le rôle d'une unité de calcul ou d'un dépôt d'une donnée. Cette entité constitue la base d'une description architecturale. À travers ses interfaces, un composant peut exprimer les liens avec les autres composants. De même, il est capable de décrire son organisation interne en termes de sous composants reliés les uns aux autres.

— **Connecteur** : C'est un élément architectural qui joue le rôle explicite de connecteur entre deux composants. En définissant des règles de spécification des interactions, un connecteur permet de modéliser l'interconnexion entre composants. Un composant peut interagir avec l'environnement externe uniquement à travers ses interfaces et des connecteurs. Ce principe est illustré dans la figure 1.6 donnant un exemple d'un connecteur qui relie deux composants. Via l'interface du composant, des services nécessaires pour son fonctionnement sont offerts comme des messages, des variables ou des opérations. Selon l'ADL choisi, la définition de l'interface change. Parmi les ADLs, il y a ceux qui considèrent chaque point de l'interface comme un port et d'autres qui considèrent un port en tant que toute l'interface. Dès lors, différents types d'interactions peuvent être décrits par les connecteurs. Ces interactions sont soit simples comme l'appel de procédure ou l'accès à une variable partagée soit complexes telles que l'échange de données sous forme de flux.

— **Configuration** : Un ensemble de composants liés à travers les connecteurs forme une configuration. Cette dernière décrit un état d'une description architecturale du système. Une telle représentation vise à vérifier si les différentes entités (composants-connecteurs) sont convenablement liées et donc à confirmer si le comportement du système souhaité est bien respecté.

Types d'ADLs

Selon la littérature, trois catégories des ADLs sont distinguées :

— **Les ADLs formels :** comme leur nom l'indique, les ADLs formels servent à décrire théoriquement un système d'une manière formelle. Sur le plan pratique, ce type

d'ADLs définit une abstraction des composants et des connecteurs sans les lier à leurs correspondants concrets. À cet égard, ces ADLs sont généralement dédiés pour faire l'analyse des systèmes mais ils sont mal adaptés pour la génération de code. Parmi les ADLs formels, nous citons Rapide [Luc96] et ACME [GMW00].

— **Les ADLs restreints :** dans ce type de langage, l'ensemble des composants logiciels sont décrits sans considérer leur sémantique opérationnelle. Fractal [CS06] est l'un de ces ADLs.

— **Les ADLs concrets :** par rapport aux ADLs formels, ce type d'ADLs est mieux adapté pour refléter la nature du monde réel. Il s'agit de raffiner le concept de composant par l'établissement de plusieurs séries correspondant chacune à une variabilité matérielle ou logicielle. Cette catégorie de langages est ainsi mieux adaptée pour la génération automatique de code à partir des modèles. Les ADLs concrets les plus connus sont AADL (*Architecture Analysis & Design Language* [SAE12]) et UML (*Unified Modeling Language* [OMG09]).

1.5 Développement orienté aspect de logiciel

Avec l'évolution technologique et la complexité croissante des systèmes temps-réel distribués en vue d'être plus fiables et plus sûrs, des nouvelles exigences doivent être prises en compte. Parmi ces exigences, nous citons la tolérance aux pannes, la sécurité ou la journalisation qui représentent des préoccupations transversales (*Crosscutting concerns*). L'intégration de ces préoccupations est faite au détriment d'une mauvaise lisibilité du code induisant à la difficulté de la maintenance. Pour pallier ces problèmes, le paradigme de développement orienté aspect de logiciels, (*Aspect Oriented Software Development (AOSD)*) [FECA05]) est apparu. La programmation orientée aspect, POA (AOP : *Aspect Oriented Programming*) assure la séparation des préoccupations tout au long du cycle de développement logiciel. Ainsi, elle facilite la maintenance des logiciels. Dans cette section, nous commençons d'abord par une introduction à la POA [KLM+97]. Puis, nous détaillons ses concepts de base. Enfin, nous clôturons par les avantages apportés.

1.5.1 Définition de la POA

Tout système logiciel peut nécessiter l'implantation des préoccupations métiers appelées aussi préoccupations fonctionnelles et d'autres techniques connues sous le nom de préoccupations non fonctionnelles ou encore transversales. Intégrer manuellement les traitements décrivant ces préoccupations transversales, dans le code métier de l'application, cause certains problèmes tels que la dispersion (*code scattering*) et l'enchevêtrement du code (*code tangling*). La dispersion de code consiste à la propagation de la portée d'une préoccupation transversale sur la totalité de l'application. L'exemple très connu de gestion de traces

présente clairement ce problème. Si le développeur souhaite afficher une trace après l'exécution de tous les sous-programmes, il sera obligé d'ajouter une ligne de code à la fin du traitement de chacun d'eux. Ainsi, pour supporter la gestion de traces, cette méthode de développement engendre une redondance dans la totalité du code de l'application. Pour l'enchevêtrement du code, c'est le résultat du traitement simultané de plusieurs préoccupations dans un même module tels que le traitement de la logique métier, de la journalisation et de la sécurité. L'enchevêtrement du code aussi bien que sa dispersion, présentent des problèmes fondamentaux influant sur la qualité, la réutilisation, la maintenabilité et l'évolution du code.
Fondée sur la séparation de la programmation des différentes préoccupations, la POA a émergé dans le but de pallier ces divers problèmes et de simplifier le processus de développement logiciel (voir figure 1.7). Il s'agit de décrire le comportement d'une préoccupation transversale dans une unité modulaire autonome nommée `Aspect` en fournissant des techniques de séparation du code fonctionnel.

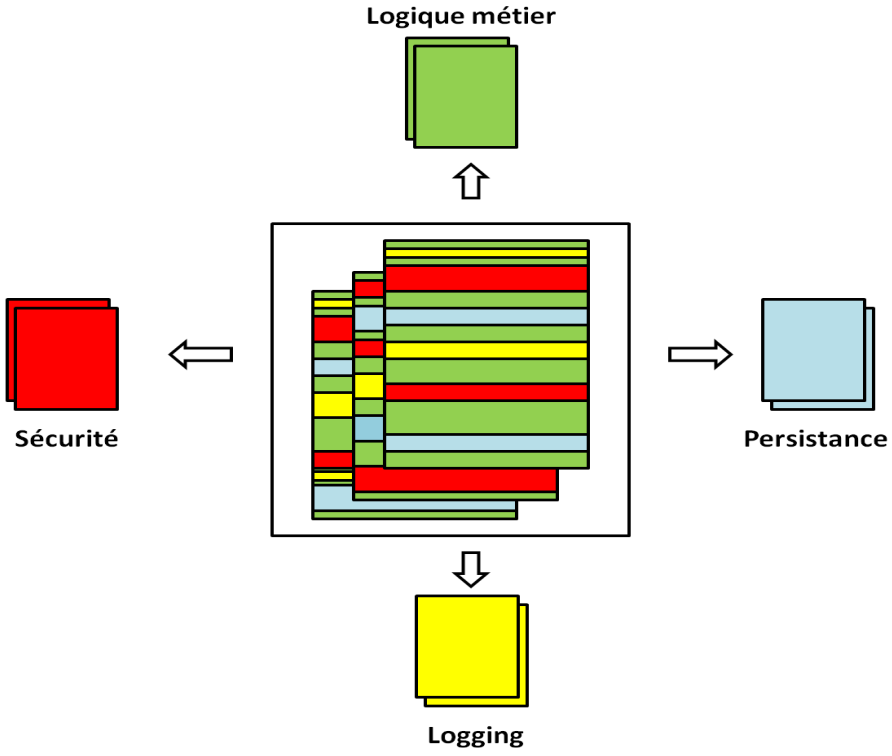

FIGURE 1.7 – Séparation des préoccupations en modules

Au début de son apparition, l'AOSD était adapté uniquement pour la phase d'implantation. Mais après une prise de conscience de ses bienfaits et de ses avantages, il a été étendu pour couvrir toutes les phases du cycle de développement logiciel [FECA05]. Afin de bien faire la liaison entre les préoccupations transversales et les descriptions métiers, l'AOSD

offre des nouveaux concepts (*pointcuts*, *joinpoints*, *advices*) et mécanismes (interception, tissage) détaillés dans la section 1.5.2. Dans ce contexte, différents langages de programmation d'aspects ont été définis afin de supporter le paradigme de programmation orientée aspect par l'extension des langages existants par le concept d'aspect. Une telle extension est possible pour des langages orientés objet comme java (AspectJ [KHH+01], JAC (*Java Aspect Components*, etc) [PDFS02]) aussi bien que pour ceux procéduraux comme le langage C (AspectC [CKFS01], AspectC++ [LBS04]) ou Ada(AspectAda [PC05]) .

1.5.2 Concepts de base de la POA

Comme indiqué précédemment, l'intégration du paradigme orienté aspect a été fondée sur la séparation entre les préoccupations fonctionnelles et transversales. De cette manière, l'application est divisée en deux parties : l'application de base implantant les fonctionnalités métiers et les modules orientés aspect, implantant les préoccupations transversales. Pour la définition et l'association des propriétés non-fonctionnelles dans le code fonctionnel, la POA repose sur certains mécanismes et définit les concepts suivants :

— **Points de jonction** (*joinpoints*) : ils sont appelés aussi des points d'insertion, ils servent à établir la liaison entre le code métier de l'application et les critères transversaux. Ces points sont définis dans le but d'indiquer des instants bien précis lors de l'exécution de l'application dans lesquels on souhaite insérer le comportement transversal.

— **Point de coupure** (*Pointcut*) : ces points regroupent un ensemble de points de jonction.

— **Code de l'advice** (*Advice*) : c'est une portion de code définissant le comportement transversal associé à un *pointcut*. Il est exécuté chaque fois quand l'exécution atteint un point de jonction défini dans le point de coupure correspondant. Il peut être exécuté avant (*Before*), après (*After*) ou autour (*Around*) d'un joinpoint.

— **Aspect** : c'est un module dans lequel une préoccupation spécifique est encapsulée[FECA05].

À la base de ces concepts, la POA fait la liaison entre les deux parties du code par le biais des deux opérations suivantes :

— **Interception** : c'est l'action d'intercepter un moment précis de l'exécution de l'application désigné par un joinpoint pour insérer un comportement spécifique.

— **Tissage** (*weaving*) : une entité spécifique appelée tisseur (*weaver*) permet d'intégrer le code des advices dans le code métier de l'application aux joinpoints définis par leurs pointcuts correspondants. Selon le temps de tissage, deux types de tisseurs sont distingués : les tisseurs statiques où le tissage des aspects est réalisé au moment de la compilation et ceux dynamiques qui font le tissage au moment de l'exécution.

La figure 1.8 illustre le principe de la POA.

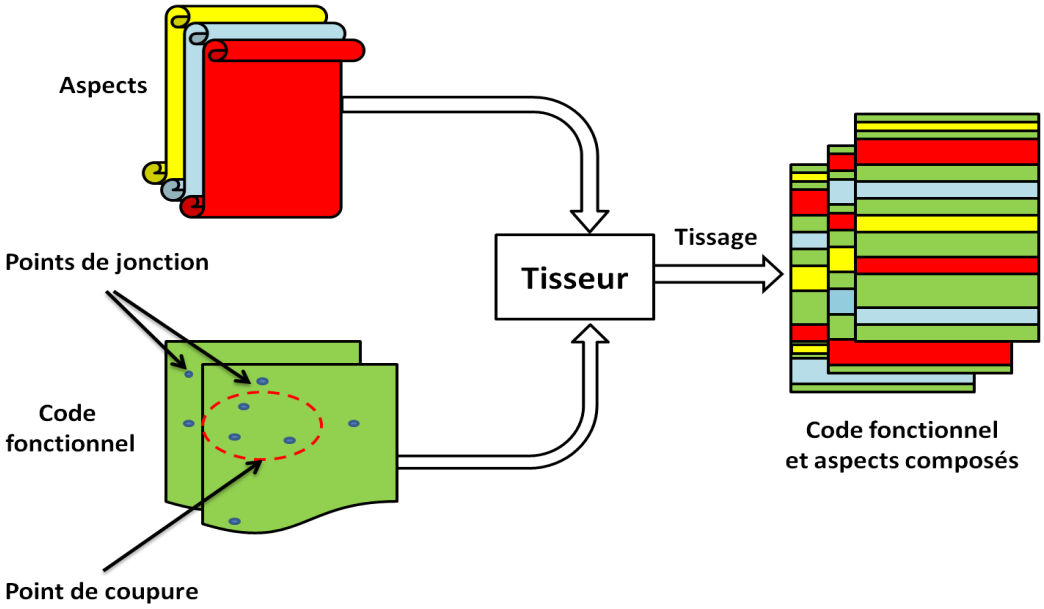

FIGURE 1.8 – Principe de la POA

1.5.3 Apports de la POA

La POA a pu résoudre les problèmes de dispersion et d'enchevêtrement de code grâce à la séparation entre les préoccupations fonctionnelles et transversales. De cette façon, la POA n'a pas remplacé les paradigmes actuellement utilisés mais elle a pu les améliorer. Par rapport aux paradigmes traditionnels, la POA offre plusieurs avantages :

— Meilleure réutilisation : la POA met en faveur l'utilisation des modules. Chacun de ces modules implante une seule préoccupation transversale. De ce fait, un tel module peut être facilement réutilisé. En outre, ajouter une nouvelle préoccupation revient à ajouter tout simplement un nouvel aspect. Ainsi, la POA favorise une meilleure réutilisation. Comme exemple, un aspect produisant la journalisation peut être utilisé dans plusieurs applications sans modifier le code correspondant.

— Amélioration de la qualité du code : la POA repose sur la séparation des préoccupations ce qui évite la duplication de code. Dès lors, on atteint une meilleure qualité du code et on garantit plus de simplicité, lisibilité et compréhensibilité.

— Gain de productivité : la POA rend possible et flexible le travail de plusieurs programmeurs en équipe. Chacun de ces programmeurs n'est chargé que du développement de l'aspect qui l'intéresse. Son travail est alors plus familier, commode et simple.

— Maintenance aisée : chaque préoccupation, encapsulée dans un module technique séparé du code métier, peut être modifiée ou maintenue aisément sans toucher les autres.

Pour l'ensemble des programmeurs au sein d'une même équipe, tous ces avantages les encouragent à profiter de la POA pour bénéficier de la parallélisation de l'implantation.

1.6 Conclusion

Dans ce chapitre, nous avons introduit les concepts de bases aux alentours de nos travaux de thèse. Nous avons commencé par présenter les systèmes temps-réel, leurs caractéristiques et leurs contraintes. Ensuite, nous avons présenté une vue d'ensemble sur les techniques de la sûreté de fonctionnement et en particulier la tolérance aux pannes. À ce niveau, nous avons détaillé les principes, les besoins et les entraves de la tolérance aux pannes. Puis, nous avons introduit les architectures logicielles et les langages de leur description. Enfin, nous avons introduit la POA, vu qu'il est le paradigme que nous avons adapté.

Dans le chapitre suivant, nous présentons le contexte de notre travail ainsi que les différents travaux qui s'orientent vers ce contexte.

2

État de l'art

2.1 Introduction

L'objectif de ce chapitre est de positionner les contributions de cette thèse face aux travaux existants. Il est pour cela composé de trois parties. Dans la première partie, nous discutons les langages de description d'architectures dédiés pour la modélisation des systèmes temps-réel en mettant l'accent sur les concepts mis à contribution dans la suite de cette étude et en particulier les techniques de modélisation de la tolérance aux pannes. Cette partie a pour but de mettre en faveur le choix du langage AADL et ses annexes. La deuxième partie de ce chapitre s'intéresse à la notion de POA et traite en particulier l'intégration de ce paradigme dans le domaine du temps-réel et de la tolérance aux pannes sur les deux niveaux, conceptuel et exécutif. Cette partie présente une synthèse permettant de classifier les travaux de recherche qui font appel à la POA pour la modélisation ou l'implantation des mécanismes de la tolérance aux pannes. Étant donné la spécificité des systèmes temps-réel et de leurs fortes contraintes devant être satisfaites, les langages de POA ne sont pas évalués de la même manière dans les domaines qualifiés de temps-réel critiques que les autres domaines. Pour cela, dans la troisième partie, nous proposons une synthèse permettant de classifier les travaux de recherche qui concernent l'évaluation des langages d'aspects pour le développement des systèmes temps-réel. Ainsi, nous distinguons les langages fortement recommandés dans tels domaines comme le domaine aéronautique. Nous nous intéressons, en particulier, à la considération du langage Ada et son extension par les aspects. Au terme de ce chapitre, nous serons alors en mesure de définir les grandes lignes de nos contributions.

2.2 ADLs modélisant les systèmes temps-réel tolérant aux pannes

Cette section permet de discuter un ensemble de langages de description d'architecture parmi les plus utilisés dans le domaine temps-réel comme l'automobile et l'avionique. Parmi ces langages, nous nous focalisons sur *UML*, *Electronic Architecture and Software Technology Architecture Description Language (EAST-ADL)* et le langage *AADL*. Pour chacun d'eux, nous mettons l'accent sur les extensions possibles dédiées pour paramétrer ou décrire les mécanismes de tolérance aux pannes vue son importance dans les domaines visés par ce travail de thèse.

2.2.1 UML et ses profils

UML est un langage de modélisation standard unifié de l'OMG à usage général. Il permet de modéliser non seulement l'architecture des applications, leurs comportements, leurs structures mais aussi les structures de données. Ce langage était sujet de plusieurs travaux de recherche visant son extension pour divers domaines. Dans le but de mettre en place des patrons de redondance, il a subi une application du paradigme orienté aspect au niveau de la conception. Dans [DM05], les auteurs se sont focalisés sur la séparation de la modélisation des préoccupations fonctionnelles et non fonctionnelles. Ils ont enrichi un modèle d'architecture UML original par un modèle aspect pour la conception des modules de redondance associés et des mécanismes de tolérance aux pannes à travers un modèle tisseur. Ainsi, la réutilisation de ces concepts était possible en se basant sur un ensemble de bibliothèques jouant le rôle de patrons de conception dans ce contexte.

La séparation des préoccupations a été à son tour la base d'extensions du langage UML par une variété de profils. En effet, ce langage est extensible par des profils adaptés à des domaines spécifiques ou des technologies particulières. Dans notre contexte, nous nous sommes intéressés particulièrement au profil *Modeling and Analysis of Real-Time Embedded systems (MARTE)*, *MARTE-Dependability Analysis and Modeling (MARTE-DAM)* et *Quality Of Service and Fault Tolerance Characteristics & Mechanisms (QFTP)*).

Profil MARTE

Le profil MARTE [OMG08] d'UML est un standard de l'OMG supportant la modélisation et l'analyse des systèmes embarqués temps-réel (*Real-Time Embedded Systems (RTES)*). Il enrichit le langage UML par des concepts relatifs à l'ingénierie dirigée par les modèles et vient remplacer le profil SPT (*UML Profile for Schedulability, Performance and Time (SPT)*) [OMG05]. Ce profil permet la spécification, la conception, la vérification et la validation des RTES. Pour la modélisation de ces types de système, UML-MARTE supporte non seulement les propriétés fonctionnelles mais aussi celles non fonctionnelles à savoir la spécification du temps d'exécution ou de l'empreinte mémoire. Ce profil supporte aussi plusieurs niveaux d'abstraction en se basant sur une variété de paquetages qui permettent à leurs tours une

séparation entre les deux niveaux logiciel et matériel du système. En outre, MARTE supporte la reconfiguration dynamique des RTES en se basant sur les modes et les transitions entre modes qui modifient les caractéristiques internes des composants déclenchées par des événements. Les modes et les transitions sont regroupés pour former une machine à états décrivant la reconfiguration comportementale dynamique du système.

Profil MARTE-DAM

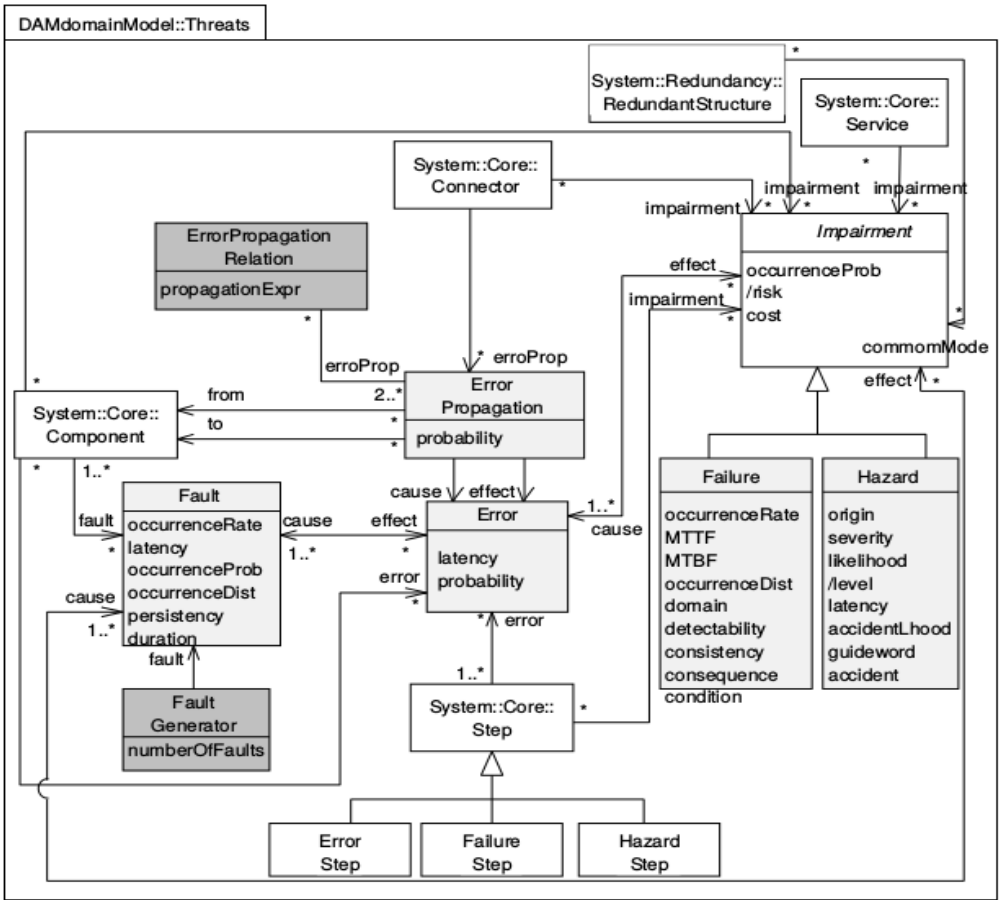

FIGURE 2.1 – Modèle de domaine du profil MARTE-DAM (Threats Package) [BMP11]

Comparé aux domaines de la performance et d'analyse d'ordonnançabilité, qui sont supportés par les profils UML standards tels que SPT et MARTE, un autre profil similaire a été standardisé pour la modélisation et l'analyse de la sûreté de fonctionnement pour les systèmes temps-réel embarqués. Il s'agit du profil MARTE-DAM [BMP11]. Son objectif principal était l'analyse quantitative de la fiabilité des systèmes logiciels modélisés avec UML en mettant l'accent sur des attributs particuliers de la sûreté de fonctionnement qui sont la fiabilité, la disponibilité, la maintenabilité et la sécurité. Ce profit vient unifier la terminologie et les concepts relatifs à ce domaine dans un modèle de domaine commun. Ce dernier

couvre différents aspects de la sûreté de fonctionnement, et réutilise et unifie les concepts de travaux antérieurs jugés comme bonnes pratiques dans ce domaine. À titre d'exemple, nous présentons dans la figure 2.1, le modèle du domaine proposé par le profil MARTE-DAM pour décrire l'ensemble de ses menaces.

Ces concepts sont par la suite traduits vers des stéréotypes du profil UML. Les nouveaux stéréotypes proposés étendent les méta-classes du langage en se basent sur des spécifications de ceux du profil MARTE. Par contre, les stéréotypes de définition des attributs se basent sur les concepts spécifiques définis dans la bibliothèque DAM. Un tel profil est ainsi considéré comme un (*Domain Specific Language (DSL)*) pour le domaine d'analyse de la sûreté de fonctionnement.

Profil QFTP

Le profil *QFTP* [CP04] est une extension spécifique du langage UML pour la description des concepts relatifs à la qualité de service et la tolérance aux pannes. L'extension de ce profil est basée sur deux frameworks généraux : un framework pour la modélisation de la qualité de service (QOS, *Quality Of Service Modeling Framework*) et un autre pour la modélisation des concepts de la tolérance aux pannes (FT, *Fault Tolerance Modeling Framework*). En particulier, le framework FT comprend des notations permettant l'analyse des risques tout en mettant l'accent sur la description des dangers, des risques et de leurs traitements. Ce framework supporte aussi la description architecturale de la tolérance aux pannes à base de réplication. Dans la figure 2.2, nous présentons le méta-modèle spécifique pour décrire les différents styles de réplication comme la réplication active et la réplication passive.

Pour l'analyse des risques, le profil QFTP accorde une attention particulière à une variété de risques, les relations entre ses différents types et leur traitements à travers des sous-profils qui dépendent fortement de la QOS. Ce profil est destiné pour les systèmes distribués complexes avec des exigences de haute fiabilité tout en considérant la fiabilité comme étant une qualité de service pouvant être évaluée de plusieurs manières. Dans ce contexte, le profil QFTP fournit des solutions pour la modélisation de la réplication, la détection des pannes et le rétablissement. À titre d'exemple, l'utilisateur de ce profil peut fixer un nombre minimal de répliques ou différents styles de monitoring dans le but de détection des pannes.

Discussion

Afin de modéliser les systèmes temps-réel distribués dynamiquement reconfigurables, UML est enrichi par différents profils visant le support des concepts relatifs aux propriétés temporelles, fonctionnelles et non fonctionnelles ainsi que la reconfiguration dynamique. D'une part, UML-MARTE permet la spécification des configurations comportementales des systèmes temps-réel à la base des machines d'états. Il permet uniquement de modéliser un nombre prédéfini de reconfigurations comportementales mais pas de reconfigurations archi-

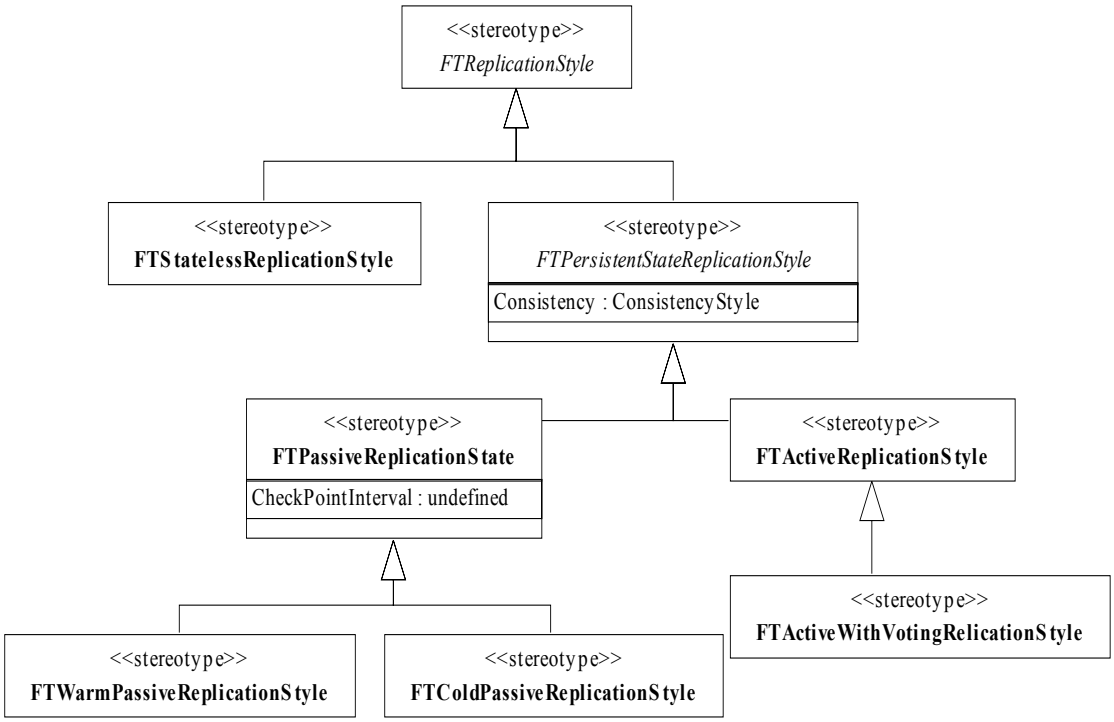

FIGURE 2.2 – Méta-modèle (stéréotypes) du profil QFTP pour les styles de réplication [CP04]

tecturales. Ainsi, il ne répond pas aux besoins des systèmes distribués. Pour cela, des travaux focalisant sur cet aspect ont été proposés pour étendre UML afin de supporter les reconfiguration architecturales encapsulées dans un seul méta-modèle pour éviter l'énumération de toutes les configurations possibles. Dans ce contexte, un nouveau profil a été proposé dans le cadre des travaux de Krichen *et al* [KGHZ12]. D'autre part, UML-MARTE ne fournit pas les capacités suffisantes pour une modélisation complète et rigoureuse de toutes préoccupations fondamentales de la sûreté de fonctionnement. Cependant, le profil MARTE-DAM est riche en termes de ces concepts à savoir l'analyse des risques et la gestion de la réplication en spécifiant un modèle du domaine considérable et précis. À titre d'exemple, à des fins d'évaluation de la performance et d'analyse de la fiabilité des systèmes tolérants aux pannes, [BMP11] ont défini un processus de transformation des annotations UML-MARTE vers une description formelle en réseaux de Petri Déterministes Stochastiques (*Deterministic Stochastic Petri Nets (DSPN)*).

Plus particulièrement pour la modélisation des systèmes tolérants aux pannes, le profil QFTP a été utilisé dans le but de concevoir des systèmes distribués tolérants aux pannes. Les auteurs de [HRV+08] et [HRL+08] ont été basés sur les deux profils UML QFTP et *CORBA Component Model (CCM)* pour définir un processus de modélisation des systèmes distribués à base de composants tolérants aux pannes. Il s'agit d'étendre le framework *Fault-Tolerant*

CORBA Services (FT-CORBA) visant la conception et le développement des mécanismes de la tolérance aux pannes à savoir la détection des pannes et la redondance.

La combinaison de plusieurs profils UML visant la couverture des différents besoins temps-réel, de dynamisme, de la sûreté de fonctionnement et finalement de la reconfiguration dynamique rend son utilisation très complexe et fastidieuse dans ce domaine.

2.2.2 EASt-ADL

EAST-ADL [CFJ+10] est un langage de description d'architecture initialement défini dans le cadre du projet ITEA EAST-EEA [3] et ensuite raffiné et aligné avec la norme *AUTomotive Open System ARchitecture (AUTOSAR)* [4]. Actuellement, il est maintenu et évolué par l'association EAST-ADL [5]. Ce langage décrit les systèmes électroniques automobiles à travers un modèle d'information complet sous une forme normalisée. Les aspects couverts incluent les caractéristiques des véhicules, les fonctions, les exigences, la variabilité, les composants logiciels, les composants matériels et la communication entre eux.

Un modèle EAST-ADL est structuré en différents niveaux d'abstractions. Chaque sous-modèle représente à son tour un système embarqué complet avec un niveau d'abstraction approprié. Ses niveaux d'abstractions correspondent aux niveaux d'abstraction de la norme ISO 26262. L'approche est basée sur la représentation AUTOSAR des architectures logicielles et des détails d'implantation matérielle (voir figure 2.3).

La définition EAST-ADL d'une structure d'un système (par exemple, en terme de composants-connecteurs) est administré selon plusieurs niveaux d'abstraction pour faire face à la gestion de la complexité, la sécurité, la fiabilité, l'amélioration des coûts et le développement efficace à base de modèles. Ces modèles respectent les fondements du cycle de vie d'un système et gardent une traçabilité complète entre les différents niveaux allant du plus haut niveau jusqu'au niveau le plus détaillé [KEM+13].

Modèle d'erreur : HIP-HOPS

EAST-ADL permet la modélisation des propriétés non fonctionnelles telles que les propriétés temporelles et la sûreté de fonctionnement. Il offre des capacités puissantes de modélisation des fautes à la base des concepts du modèle d'erreur [WRTP+13]. Ce dernier représente une vue séparée de modélisation parallèlement aux modèles de systèmes nominaux sur chaque niveau. Le modèle d'erreur permet aux concepteurs du système de spécifier les défaillances pouvant survenir pour chacun des composants et comment ces échecs peuvent se propager à d'autres parties du système. Similaire à d'autres parties de EAST-ADL, le modèle d'erreur stocke des informations uniquement. Pour les analyser, il faut faire appel à un

3. ITEA EAST-EEA : ITEA-Project-No. 00009,http ://east-eea.test.k-k.de/. C'est un projet étudiant l'architecture électronique embarqué entre l'industrie européenne d'automobile, des fournisseurs et des groupes académiques

4. www.autosar.org

5. www.east-adl.info

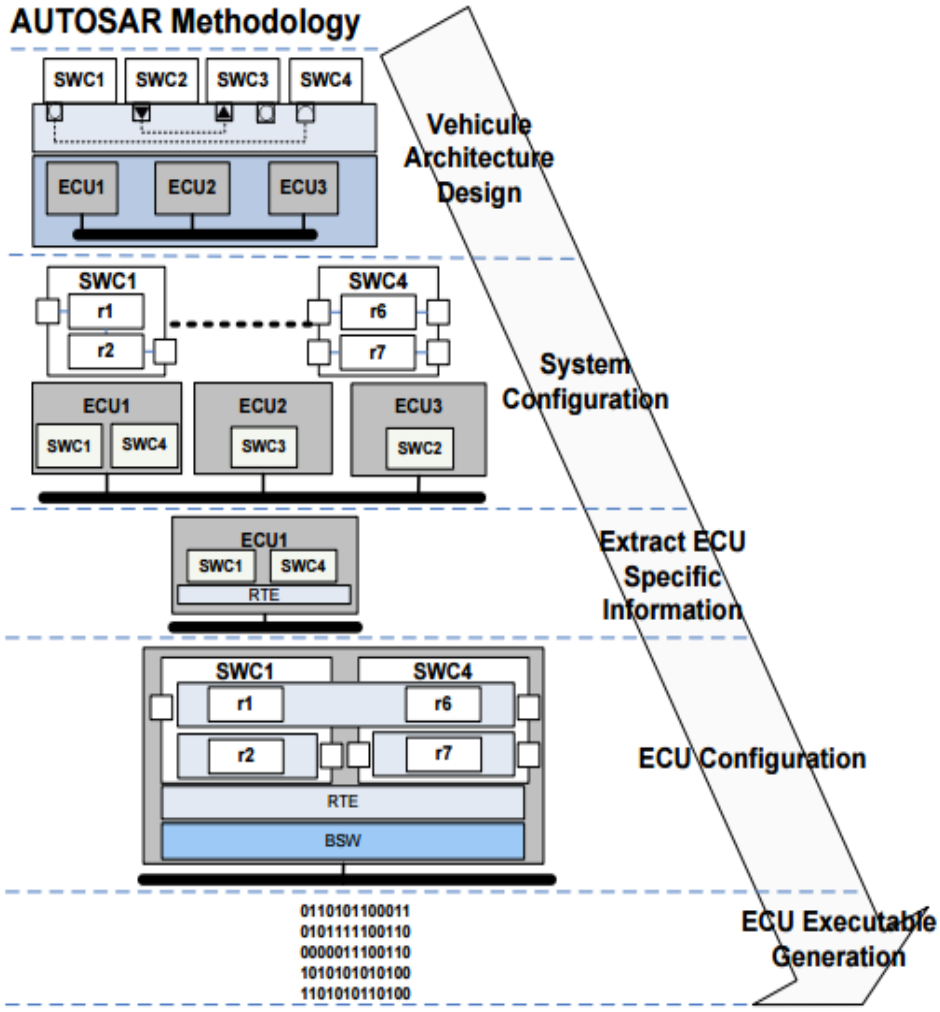

FIGURE 2.3 – Architecture multiniveaux de AUTOSAR [CFJ+10]

outil externe. Dans le cadre des deux projets ATESST2 et MAENAD, des efforts importants ont été consacrés pour mettre en place un outil d'analyse de sécurité des modèles EAST-ADL, appelé (*Hierarchically Performed Hazard Origin and Propagation Studies (HIP-HOPS)*. Cette analyse repose sur la transformation de modèle. Cela a entraîné une certaine harmonisation des concepts de modélisation d'erreur dans EAST-ADL et HIP-HOPS ayant comme résultat une capacité puissante d'analyse de la fiabilité des modèles de EAST-ADL. En effet, une analyse effectuée par l'outil HIP-HOPS se fait en trois phases [SPC+14] :

— La phase d'annotation du modèle, au cours de laquelle l'analyste annote l'architecture du système avec des descriptions des erreurs au niveau des composants.

— La phase de synthèse des arbres de fautes durant laquelle HIP-HOPS fait la synthèse automatique des arbres de fautes au niveau du système pour modéliser la propagation

des pannes à travers le système.

— La phase d'analyse, pendant laquelle HIP-HOPS effectue les deux analyses (*Fault Tree Analysis (FTA)* et *Failure Modes Effects Analysis (FMEA)*) en fonction des modèles de propagation générés.

Modèle temporel : TADL

EAST-ADL fournit un support pour la modélisation spécifique des aspects techniques, y compris les propriétés non-fonctionnelles qui sont pertinentes pour la synchronisation des systèmes automobiles. Au niveau conceptuel, les aspects temporels des systèmes temps-réel distribués peuvent être divisés en exigences temporelles et propriétés temporelles. Dans ce contexte, les propriétés temporelles d'une solution doivent effectivement répondre aux exigences temporelles spécifiées.

Pour un modèle EAST-ADL structuré et hiérarchique donné, les contraintes temporelles sont définies séparément à travers le langage *Timing Augmented Description Language (TADL)* [BHF+12] développé dans le cadre du projet TIMMO [6].

Le support de la modélisation des exigences dans EAST-ADL permet de tracer des exigences à partir des solutions modélisées au niveau structurel et aussi des vérifications. Les contraintes de TADL ajustent les exigences en termes de raffinements à base d'événements. A chaque niveau d'abstraction, on peut identifier des événements classés en deux catégories. Les événements qui provoquent une réaction, à savoir un stimulus, et des événements résultats d'une réaction, à savoir une réponse. Ces événements à leurs tours peuvent être soumis à d'autres contraintes temporelles structurées sous forme de chaîne. À titre d'exemple, on peut synchroniser deux événements ou borner le temps séparant l'occurrence de deux événements (stimulus et réponse). Pour prédire le comportement temporel des systèmes automobiles modélisés avec EAST-ADL, il existe différentes méthodes qui prennent en charge l'extraction des modèles de synchronisation à partir des modèles architecturaux. Bucaioni *et al* [BMC+15] fournissent des lignes directrices pour la sélection de la méthode la plus adaptée en ce qui concerne l'analyse temporelle du comportement d'un système embarqué.

Dans [BMCS15], les auteurs s'intéressent à l'analyse temporelle des modèles EAST-ADL à des niveaux d'abstraction plus élevés en se basant sur l'ingénierie dirigée par les modèles et l'ingénierie à base de composants. En utilisant cette méthodologie, l'architecture logicielle à un niveau supérieur est automatiquement transformée à tous les modèles au niveau de la mise en œuvre. L'analyse de synchronisation de bout en bout est effectuée sur chaque modèle de niveau application généré et les résultats d'analyse sont alimentés vers le modèle de niveau de conception. Cette activité soutient l'exploration spatiale de la conception, le modèle de raffinement ou de remodelage à des niveaux d'abstraction plus élevés pour le réglage du comportement temporel du système.

6. TIMMO Consortium : TIMMO 2 USE Project web site. http ://www.timmo-2-use.org/

Discussion

EAST-ADL a été développé spécifiquement pour la modélisation des systèmes électroniques automobiles à travers un modèle d'information complet passant par quatre niveaux d'abstraction : le niveau véhicule, le niveau analyse, le niveau conception et finalement le niveau implantation. Ce langage permet de décrire différents types de composants tout en spécifiant leurs comportements et leurs propriétés toujours en relation avec un certain niveau d'abstraction et non hiérarchique. Pour la spécification des comportements, EAST-ADL offre un support limité de description des comportements à base de modes et de transitions [JL11]. De ce fait, il est extensible par des notations externes au langage comme Simulink. Concernant les propriétés, ce langage les traite comme des entités séparées associées à un composant cible en se basant sur les principes du langage SysML. Plus particulièrement, les propriétés non-fonctionnelles, à savoir les propriétés temporelles (temps de latence), sont décrites à travers des éléments AUTOSAR. EAST-ADL fait appel à des paquetages spécifiques explicites pour la modélisation des propriétés temporelles et de la sûreté de fonctionnement vue leur importance pour les systèmes automobiles. Pour l'analyse des propriétés temporelles des modèles EAST-ADL, elle est supportée par le profil UML-MARTE à travers le plug-in Papurus [7]. En ce qui concerne la sûreté de fonctionnement, le paquetage **dependability package** dédié offre des moyens pour la modélisation de ces techniques. Il aide le concepteur à structurer les besoins de sécurité selon leur importance dans le cycle de vie du système, formaliser les besoins à travers les contraintes du système, analyser la propagation des fautes en partant des modèles d'erreurs et des structures de sécurité et d'une façon générale de spécifier et de classifier les besoins de la sûreté de fonctionnement. Ces concepts sont considérés suivant les normes du standard *Automative Safety Standard ISO/DIS 26262* et sont mis en œuvre à travers HIP-HOPS.

Similaire au langage UML, le langage EAST-ADL ne peut pas être utilisé tout seul pour la modélisation des systèmes temps-réel présentant un haut niveau de sûreté de fonctionnement. Il est extensible par différentes notations externes pour la modélisation et l'implantation des aspects spécifiques comme la reconfiguration dynamique, les propriétés temporelles ou même les besoins de sécurité. De plus, pour l'analyse de chacun de ces concepts, le concepteur a besoin d'une multitude d'outils (HIP-HOPS, TADL, Papurus,etc). Ceci est relatif aux différents niveaux d'abstraction exigés par ce langage qui le rendent approprié aux systèmes communicants et à la compréhensibilité. Contrairement au langage AADL approprié pour la modélisation des systèmes se basant sur des éléments concrets (logiciels, matériels ou hybrides), EAST-ADL laisse ainsi moins de liberté au concepteur de structurer son système et d'obtenir les fonctionnalités d'implantation.

7. http ://www.eclipse.org/papyrus/

2.2.3 AADL

AADL [SAE12] est un standard qui offre à la fois une description graphique et textuelle visant toutes deux à décrire l'exécution sémantique des systèmes logiciels embarqués temps-réel. Ce langage a prouvé sa puissance dans le domaine temps-réel, spatial et aéronautique. AADL est un langage concret pouvant modéliser tout un système hétérogène comportant des composants logiciels, matériels ou hybrides tout en spécifiant les connexions qui les relient. Cependant, pour les propriétés non fonctionnelles, ce langage peut être étendu par des annexes ou des propriétés pour des descriptions spécifiques. Les propriétés et les annexes utilisées sont soit standards soit personnalisées. Parmi les annexes standards qui touchent notre domaine, nous citons l'annexe d'erreurs EMA [SAE15] et l'annexe comportementale (*Beahavioal Annex (BA)*) [SAE06]. L'annexe d'erreurs EMA permet de décrire tout un modèle d'erreur architectural en spécifiant les types d'erreur pouvant survenir, les points de propagation de telles erreurs, les techniques de détection et même de recouvrement. Il offre ainsi un support de la tolérance aux pannes et plus généralement de la sûreté de fonctionnement.

Une introduction au langage AADL et de quelques annexes dont nous avons besoin, est présentée dans le chapitre 4.

Discussion

AADL est un standard dédié pour la spécification des systèmes temps-réel embarqués distribués. Il est utilisé dans le domaine de l'aéronautique et de l'espace. AADL est un langage concret permettant à la fois de représenter les composants logiciels, matériels et hybrides donnant ainsi lieu à un modèle hiérarchique. Chacun des composants au niveau du modèle peut être enrichi par des propriétés ou des annexes afin de décrire leur comportement. Le modèle architectural ainsi que le modèle comportemental peuvent subir des changements suite à l'apparition de nouveaux besoins (décrits par le concepteur) ou suite à des évolutions dans l'environnement. Ces changements peuvent être pris en compte à travers la reconfiguration dynamique. Pour le langage AADL, il permet de spécifier la reconfiguration dynamique en utilisant les modes et les transitions comme machine à états. Toutefois, ces reconfigurations doivent être énumérées d'avance. Dans le contexte de la sûreté de fonctionnement, le recouvrement consiste à passer d'un état erroné à un état exempt d'erreur. Ceci peut être garanti en utilisant les modes du langage AADL. Plus particulièrement, dans le contexte des systèmes critiques, toute panne ainsi que les mécanismes de réparation associés doivent être prévus très tôt dans le cycle de développement. Pour cela, une annexe EMA permettant l'extension des modèles AADL par des modèles d'erreurs a été proposée. Cette annexe permet non seulement de représenter les types de pannes pouvant attaquer le système, mais aussi la manière de les détecter, de les réparer et même de les analyser. De cette façon, les besoins de la sûreté de fonctionnement peuvent être décrite séparément en utilisant l'annexe EMA. Dans ce contexte, A. Rugina [Rug08] a utilisé ce langage pour définir un processus de

modélisation et de génération des modèles analytiques de sûreté de fonctionnement. Ceci est dans le but de faciliter l'évaluation des mesures de sûreté de fonctionnement comme la disponibilité et la fiabilité. Le travail a défini un nouveau modèle à base du langage AADL, son annexe d'erreurs et un ensemble de règles de transformation vers des réseaux de Petri stochastiques généraux (*General Stochastic Petri Nets (GSPN)*). L'analyse des GSPN générés permettent de valider les modèles AADL étendus par l'annexe d'erreurs EMA vis-à-vis de leur spécification. Cependant, cette approche ne s'intéresse pas à la génération du code des applications sûr par construction, mais plutôt à l'analyse et la vérification de ces systèmes au niveau modèle.

2.2.4 Synthèse

Face à la grande importance donnée durant les dernières années à la sûreté de fonctionnement des systèmes temps-réel embarqués, plusieurs travaux dans ce contexte se sont focalisés sur les approches d'ingénierie guidée par des modèles afin de modéliser, analyser et vérifier ces systèmes. Etant donné que la sûreté de fonctionnement des systèmes temps-réel embarqués devient un besoin fondamental dans plusieurs domaines à savoir l'avionique, le domaine médical, l'espace et même la production d'énergie, nous avons noté une variété des langages de modélisation et des techniques de mesures de la sûreté de fonctionnement. Nous avons récapitulé un ensemble de ses travaux qui sont les plus proches de notre travail dans le tableau 2.1.

Des efforts significatifs [RKK08, JVB07, MBP+15, hHZ11] ont ciblé la transformation des modèles AADL et ses annexes en des modèles d'analyses. Ces travaux ont été basés sur le standard AADL pour la modélisation de leurs systèmes en l'enrichissant par l'annexe EMA pour la spécification des modèles d'erreurs. À base de transformation de modèles, chacun de ces travaux a effectué l'analyse de sûreté de fonctionnement. En effet, une transformation des modèles AADL des systèmes tolérants aux pannes en GSPN a été proposée par [RKK08]. Ces modèles générés aident le concepteur à analyser son système dans le but d'obtenir des mesures de la sûreté de fonctionnement. Toutefois, en prenant en compte les dépendances entre les composants, les chaînes de Markov sont plus adéquates que les réseaux de Petri pour l'analyse de la sûreté de fonctionnement. D'ailleurs, ces réseaux constituent des formalismes de plus haut niveau permettant de vérifier structurellement le modèle avant de générer les chaînes de Markov. Pour ces raisons, cette démarche a été adoptée par [hHZ11]. Pour [JVB07], ils se sont basés sur la génération et l'analyse des arbres de fautes statiques (*Static Fault Tree (SFT)*) afin d'analyser et de vérifier les propriétés temporelles des systèmes temps-réel critiques. Par contre, [MBP+15] ont transformé les modèles AADL enrichis par l'annexe EMA en des spécifications HIP-HOPS qui à leur tour sont transformées en arbres de faute afin d'effectuer des analyses des effets et des modes de défaillances (FMEA). Une autre contribution plus récente [ZWL16] a enrichi les modèles AADL non seulement par l'annexe

EMA mais aussi par l'annexe ARINC et ce dans le but d'analyser les propriétés temporelles du modèle. Pour l'analyse, il s'agit d'appliquer des règles de transformation sur le modèle pour générer des chaînes de Markov temporisées continues (*Continued Time Markov chain (CTMC)*).

Tous ces travaux ont manipulé le langage AADL enrichi avec son annexe EMA dans le but de faire des analyses, des vérifications et des mesures de certaines métriques de la sûreté de fonctionnement. Mais, ils n'ont pas procédé à la génération du code applicatif de leur système. De même, les travaux qui ont ciblé le langage UML et ses profils occupent aussi une position importante dans le domaine de la sûreté de fonctionnement de point de vue modélisation aussi bien qu'analyse mais pas à des fins de génération de code. Nous citons, [HRV+08] et [BMP11]. La première contribution, visant à modéliser et analyser les systèmes à base d'architecture Common Object Request Broker Architecture (CORBA), a utilisé des modèles AADL étendus avec le profil QFTP pour donner lieu à une génération des fichiers descripteurs CCM [OMG06]. Ces fichiers sont considérés pour l'analyse de la sûreté de fonctionnement aussi bien que pour la génération de code. Concernant le profil MARTE-DAM, extension du profil UML MARTE, il enrichit le langage UML par des concepts fondamentaux de la sûreté de fonctionnement. D'autres travaux se sont concentrés sur ce profil pour modéliser et analyser des systèmes exigeant un très haut niveau de fiabilité. À cet égard, [BMP11] ont défini des règles de transformation pour générer des réseaux de Petri stochastiques à travers une description réalisée avec MARTE-DAM toujours visant l'analyse des modèles et pas la génération de code.

Dans le domaine de l'ingénierie des systèmes embarqués automobiles, le langage EAST-ADL a fait ses preuves pour une modélisation spécifique s'appuyant sur une variété de niveaux d'abstractions couvrant tout le cycle de développement. Allant des exigences jusqu'à atteindre des architectures spécifiques des systèmes, EAST-ADL raffine l'ensemble des composants logiciels ou matériels du système. Même à des fins de sûreté de fonctionnement, EAST-ADL supporte des extensions sur tous les niveaux d'abstraction pour décrire et analyser les modèles d'erreur en parallèle avec l'élaboration du modèle du système de base. Dans ce contexte, certains travaux [CMW+13, SPC+14] ont visé la modélisation et l'analyse des systèmes automobiles à travers HIP-HOPS qui effectue automatiquement les deux types d'analyses FTA et FMEA sur la base d'un modèle de système. Ce modèle peut être décrit au moyen de différents ADLs à savoir EAST-ADL et AADL. Pour [CMW+13], ils ont considéré uniquement l'analyse des arbres de fautes. Par contre, [MBP+15] ont parti depuis un modèle AADL enrichi par l'annexe EMA, et ont défini des règles de transformation de l'annexe EMA vers HIP-HOPS pour en profiter des deux types d'analyses supportés.

Cette étude autour des ADLs dans le domaine temps-réel embarqué a montré que, malgré la variété des langages de modélisation utilisés et la variété des techniques d'analyse quantitative et qualitative des attributs de la sûreté de fonctionnement, il y a un manque de support de la génération de code relatif à la sûreté de fonctionnement. Pour cela, la deuxième

partie de ce chapitre est consacrée pour aller plus loin et étudier les techniques d'implantation de la sûreté de fonctionnement et plus particulièrement la tolérance aux pannes.

TABLE 2.1 – Récapitulation des travaux visant la modélisation et l'analyse des systèmes fiables

Approche	**Techniques de Modélisation**	**Transformation de modèles**	**Types de systèmes**	**Génération de code**	**Analyse et Vérification**	**Temps-réel**
Rugina *et al*, [RKK08]	AADL& EMA	GSPN	Systèmes embarqués, distribués et reconfigurables	non	oui	oui
Joshi *et al*, [JVB07]	AADL & EMA	SFT	Systèmes fiables et critiques	non	oui	oui
Hamid *et al*, [HRL$^+$08]	UML & QFTP	Fichiers de description CCM	Architectures CORBA Systèmes distribués	C, Java, C++	oui	non
HU Jun-hua, [hHZ11]	AADL & EMA	SPN & CM	Systèmes temps-réel fiables	non	oui	oui
Bernardi *et al*, [BMP11]	UML MARTE-DAM	DSPN	Systèmes temps-réel embarqués	non	oui	oui
Chen *et al*, [CMW$^+$13]	EAST-ADL & HIP-HOPS	Fault Tree (FT)	Systèmes embarqués automobiles	non	oui	oui
Mian *et al*, [MBP$^+$15]	AADL & EMA	HIP-HOPS FT & FMEA	Systèmes fiables	non	oui	oui
zhang *et al*, [ZWL16]	AADL & EMA & ARINC	CTMC	Systèmes embarqués avioniques	non	oui	non

2.3 Modélisation et implantation de la tolérance aux pannes avec la POA

En explorant la littérature, nous avons constaté que les techniques de tolérance aux pannes utilisées pour gérer les erreurs logicielles et matérielles sont nombreuses et diversifiées. De ce fait, les auteurs de [AAM09] donnent un aperçu sur des techniques logicielles pour gérer les pannes logicielles développées dans les domaines de la sûreté de fonctionnement. Comme toutes ces techniques exploitent une certaine forme de redondance, ils ont considéré l'impact de la réplication sur l'architecture logicielle. Dans le but de comparer et de classer les mécanismes pour gérer les pannes logicielles, ils ont proposé une taxonomie fondée sur la nature et l'utilisation de la redondance dans tels types de systèmes. Á titre d'exemple, nous citons [CDP+10] où les auteurs ont proposé une approche dédiée pour les systèmes distribués tolérants aux pannes reposant sur la réplication et basée sur des données de surveillance. Cette approche utilise les deux stratégies de réplication active et passive pour mettre en place un protocole de réplication optimisé à travers un service intermédiaire proxy.

L'ajout de la tolérance aux pannes à la sécurité des applications temps-réel critiques introduit une complexité supplémentaire à l'application de base. L'une des solutions pour réduire cette complexité est la séparation et la modularisation des préoccupations transversales des préoccupations fonctionnelles sur le niveau conceptuel et le niveau applicatif. Du point de vue génie logiciel, ce paradigme a conduit à des améliorations significatives. Il permet également d'éviter un nombre important d'erreurs commises au niveau du code et de simplifier la tâche de vérification au programmeur [RPM05]. Dans ce contexte, certains considèrent la tolérance aux pannes comme une préoccupation transversale et donc la mettent en œuvre en utilisant le paradigme orienté aspect.

2.3.1 Utilisation de la POA pour l'implantation de la tolérance aux pannes

Jalilian *et al* [JSKT11] ont implanté différentes techniques de tolérance aux pannes en utilisant le langage d'aspect AspectC++, une extension d'aspect pour le langage C. Ce travail a mis en œuvre les techniques de détection et de correction d'erreur (*Error Detection and Correction (EDC)*), de test d'admission (*Acceptance Test (AT)*), l'exécution redondante (*Time Redundancy Execution (TRE)*) et la *Control Flow Checking (CFC)*. Selon cette expérience, il a prouvé les fortes capacités de la POA dans l'implantation de la POA d'une manière transparente. De plus, il a prouvé que les coûts de la POA ne sont pas assez élevés comme il est le cas pour la méta-programmation.

Alexandersson *et al* [AK11] ont proposé une étude sur l'injection de fautes permettant d'estimer le recouvrement de fautes logicielles. Ils ont implanté les deux mécanismes (*Double Signature Control Flow Checking (DSCFC)*) et (*Triple Time Redundant Execution with*

Forward Recovery (TTRFR)) en utilisant la programmation orientée aspect. Pour tester l'effet de la POA et les coûts supplémentaires qui en résultent, ces auteurs ont utilisé différents langages de programmation et ont prouvé que la POA permet de réduire d'une manière significative ces coûts. Toutefois, cette approche présente certains inconvénients. D'une part, la POA n'est pas bien adaptée pour les systèmes dynamiquement reconfigurables. D'autre part, à travers du tisseur, la POA génère un code non visible pour les programmeurs ce qui le rend vulnérable aux fautes.

Dans le même contexte, Hameed *et al* [HWS09] ont proposé un framework pour la détection de pannes et les mécanismes de recouvrement qui y sont nécessaires. Ce framework permet la conception et la gestion des exceptions s'appuyant sur des tests de vraisemblance et des capteurs *best-effort*. Les patrons de conception orientés aspect provoqués par ce framework offrent plus d'avantages comme la localisation du code responsable de la gestion d'erreur en termes de définition, d'initialisation et d'implantation. Ainsi, ce framework a prouvé sa portabilité et réutilisation grâce à l'implantation séparée du traitement des fautes via l'injection des fautes permanentes. Néanmoins, cette approche n'est pas adaptée ni aux systèmes embarqués ni aux systèmes temps-réel puisqu'elle se base sur des capteurs *best-effort* pour la détection des erreurs.

D'autres travaux s'orientent vers l'implantation séparée des mécanismes de détection d'erreurs toujours visant l'amélioration des performances de la tolérance aux pannes. Szentivanyi *et al* [SN04] implantent des services de tolérance aux pannes (FT-CORBA) pour des architectures CORBA en utilisant la POA. Effectuant une évaluation quantitative des gains de performance, ils assument que la POA a permis une baisse remarquable des coûts supplémentaires lors de l'exécution en la comparant à d'autres implantations étudiées. Cette approche offre une implantation séparée des deux mécanismes journalisation (traçage) et synchronisation. Elle apporte également une optimisation des intercepteurs dans l'infrastructure FT-CORBA pour la gestion des exceptions au niveau de l'application. Cependant, cette approche, similairement à celle qui la précède, ne peut pas être appliquée dans le domaine temps-réel embarqué à cause des besoins très importants en termes de ressources et d'empreinte mémoire demandées par l'intergiciel FT-CORBA mis en œuvre. De plus, l'intégration des aspects au niveau de l'intergiciel FT-CORBA peut provoquer une dégradation du niveau de sécurité et une difficulté de la maintenance.

En contre partie, Afonso *et al* [ASB+08] ont proposé une solution dédiée pour les systèmes temps-réel embarqués. Il s'agit d'implanter et d'intégrer des stratégies de la tolérance aux pannes dans un framework supportant ce type de systèmes. En se basant sur la POA, cette approche considère les techniques de détection et de recouvrement suivantes : les blocs de recouvrements simples [PKJ11] et distribués [kK95]) basés tous les deux sur le recouvrement par reprise et la programmation en N-versions [LA95] basée à son tour sur le masquage d'erreur et le vote majoritaire. Comme déjà mentionné au niveau de la section 1.3.6, ces techniques permettent la détection des erreurs logicielles à travers une

conception variée appelée diversité. De ce fait, cette approche est prometteuse par rapport aux autres vue qu'elle est adaptée non seulement aux systèmes temps-réel embarqués mais aussi aux systèmes distribués dynamiquement reconfigurables. Ceci revient à la communication transparente entre les nœuds grâce au modèle Publier/Souscrire [MFP06]. Toutefois, cette approche avec toutes qui la précèdent, permettent uniquement l'implantation des stratégies de la tolérance aux pannes mais ne supportent pas la modélisation de telles stratégies dès le niveau conception. De cette manière, l'implantation en utilisant la POA est indépendante du modèle de conception. En effet, ces travaux se concentrent sur l'intégration des préoccupations de tolérance aux pannes au niveau développement en utilisant la POA. En outre, ces travaux ne prennent pas en considération l'effet de l'opération de tissage sur l'ordonnancement et le déterminisme du système.

Dans notre contexte, nous nous intéressons à la définition des techniques ou des stratégies de tolérance aux pannes dès le niveau modèle comme la diversité ou la réplication. Dans cette orientation, nous avons discuté les langages de modélisation de la tolérance aux pannes et plus généralement de la sûreté de fonctionnement comme indiqué dans la section 2.2. Dans ce qui suit, nous soulignons plus particulièrement les travaux focalisant sur l'intégration de la POA dès le niveau modèle par l'extension des ADLs déjà vus.

2.3.2 Modélisation de la tolérance aux pannes avec la POA

Visant la réutilisabilité, la transparence et la capacité d'adaptation dynamique des politiques de réplication, Sánchez *et al* [STH01] se sont basés sur la POA pour la définition d'un modèle de réplication appelé *JReplica*. Ce modèle permet l'implantation séparée des politiques de réplication du comportement fonctionnel des systèmes orientés objets sur les deux niveau conception et implantation. Ils ont ainsi garanti un degré élevé de transparence grâce à la génération automatique de répliques. De plus, ils offrent la possibilité pour les programmeurs d'introduire un nouveau comportement pour spécifier des exigences autres que la tolérance aux panne. L'aspect encapsulant la réplication a été introduit lors de la phase de conception. *JReplica* est un langage à base de Java qui est dédié pour exécuter la réplication passive sous forme d'aspect à l'aide d'un nouveau stéréotype appelé *Replication*. Pour cela, UML a été étendu par ce stéréotype afin de considérer ces politiques de réplication séparément lors de la conception des systèmes tolérants aux pannes.

Dans cette vision aussi, Domokos *et al* [DM05] ont appliqué des patrons de conception pour la réplication au niveau architectural en se basant sur le paradigme orienté aspect. Ils se sont focalisés sur le tissage des modèles de réplication dans un modèle d'architecture original (non-redondant) relié. Cette approche vise la séparation des préoccupations fonctionnelles de celles non fonctionnelles dés la conception. Sur le plan pratique, ils ont utilisé le langage UML pour concevoir les besoins fonctionnelles de leurs systèmes. Puis,

ils ont intégré un modèle d'aspect au modèle UML de base à l'aide d'un modèle tisseur. Ainsi, des mécanismes de tolérance aux panneset de gestion de redondance ainsi que leurs sous-modèles spécifiques d'analyse ont été proposés pour être réutilisés par le biais d'une bibliothèque de patrons de conception.

Dans le même contexte, Bernardi *et al* [BMP11] ont proposé un profil MARTE-DAM couvrant différents aspects de sûreté de fonctionnement grâce à des modèles de domaine riches. En particulier, un modèle de redondance a introduit des composants tolérants aux pannes qui peuvent fournir une structure redondante, comme les variantes, les arbitres (*Adjudicators*) et les stratégies de tolérance aux pannes.

Niz *et al* [NF09] ont proposé une approche pour la modélisation et la vérification formelle des patrons de réplication pour le langage AADL. A la base de ces patrons, ce travail a analysé potentiellement les comportements inattendus du système. Cette approche est basée sur la séparation de l'architecture redondante du reste du modèle. Il s'agit de concevoir deux modèles séparés : le premier définit les comportements imprévus via des séquences d'appels synchronisées et le deuxième décrit l'architecture de réplication. Le style de réplication choisi est la réplication passive modélisée avec les concepts de modes et de transitions entre les modes offerts par le langage AADL. Le travail a proposé deux répliques : une primaire et une autre secondaire. La transition entre les deux modes se fait à travers le déclenchement d'un événement via un *event port*. Ce dernier est connecté à son tour à une unité de contrôle de transitions qui est représenté soit par un être humain soit par un module de détection de pannes.

Un autre travail pour la modélisation de la réplication passive en utilisant AADL a été proposé dans [LRPK10]. Ce travail a illustré par un exemple la stratégie de réplication passive modélisée à travers le langage AADL et son annexe comportementale BA [FBF+07]. Il a modélisé le système de base en utilisant les composants AADL interconnectés via les interfaces. Les processus légers sont synchronisés à travers des événements déclenchés. Puis, en se basant sur l'annexe comportementale BA, ce travail a modélisé un automate décrivant les différents états du blocage du système pour décrire les séquences d'appel exécutées pour chacun de processus légers. À travers cet exemple, [LRPK10] ont prouvé aussi que l'annexe BA offre des stratégies intéressantes permettant la définition des régions critiques. Toutefois, en appliquant cette approche, les concepteurs modélisent difficilement les mécanismes de synchronisation complexes fréquemment utilisés dans les systèmes temps-réel distribués à savoir l'exclusion mutuelle. En outre, ils doivent spécifier manuellement le modèle AADL de base ainsi que les patrons de réplication. Cette approche ne fournit pas des tâches automatiques qui aident à générer un modèle consistant.

Ping *et al* [PP12] ont proposé un modèle de tolérance non invasif pour mettre en faveur le tissage dynamique des techniques de la tolérance aux pannes dans un système existant sur les deux niveaux conceptuel et exécutif en se basant sur la POA et la réplication. Pour l'utilisation des aspects au niveau modèle, ils ont été basés sur une extension du langage

UML [SY99] et le langage OCL (*Object Constraint Language*) pour mettre en œuvre une application J2EE. Le tisseur dynamique des aspects a été développé en compilant tout d'abord le code source en des fichiers (*.class* avec un compilateur Java régulier) qui sont par la suite utilisés par le compilateur d'aspect (compilateur AspectJ) responsable du tissage. Une évaluation a été par la suite établie en calculant différents indicateurs pour estimer l'impact de l'encapsulation de la tolérance aux pannes dans un aspect sur l'application originale et en particulier d'évaluer quantitativement le niveau d'invasion. À titre d'exemple, l'indicateur le plus important était ($R_{LOC\text{-}FT}$) calculant la proportion des lignes de code concernant la tolérance aux pannes par rapport au nombre total des lignes de code de tout le système. Cet indicateur mesure le degré d'invasion. Les résultats expérimentaux montrent que le modèle orienté aspect peut effectivement favoriser la productivité du développement et la maintenabilité de la tolérance aux pannes.

2.3.3 Synthèse

Le tableau 2.2 récapitule les travaux, visant l'implantation des techniques de la tolérance aux pannes avec la séparation des préoccupations, décrits dans cette section. Ces travaux sont comparés selon différents critères. La comparaison porte tout d'abord sur les techniques de tolérance aux pannes utilisées ainsi que les types de fautes traités. Nous nous limitons dans cette comparaison aux techniques basées essentiellement sur la réplication et la diversité étant donné l'importance de ces deux techniques. En particulier, nous nous intéressons aux techniques de réplication avec ces divers styles. Parmi les styles de réplication, nous distinguons la réplication active et passive, les deux techniques les plus utilisées. Toutefois, il y a peu de travaux focalisant sur les deux ensembles [CDP+10]. La plupart s'intéresse soit à l'un soit à l'autre en se basant sur des concepts disjoints. Dans note contexte, nous visons proposer une approche supportant les deux styles de réplication active et passive et en se basant sur les mêmes concepts. La comparaison porte aussi sur les techniques d'évaluation des mesures de la tolérance aux pannes mettant en valeur les performances du système et la consommation des ressources. Vu que nous nous intéressons au domaine temps-réel embarqué critique, la consommation des ressources influe énormément sur le bon fonctionnement des systèmes en question.

Les langages d'aspects utilisés sont aussi considérés parmi les critères de comparaison des approches permettant d'implanter les préoccupations de la tolérance aux pannes en utilisant le paradigme orienté aspect. Le langage d'aspect dépend du langage fonctionnel avec lequel l'application de base est implantée. Nous remarquons que le langage AspectC++ est le langage le plus utilisé dans ce contexte. Dans le contexte des système temps-réel critiques, le langage C/C++ ou le langage Java sont sélectionnés pour coder les préoccupations fonctionnelles. Dans ce cas, les préoccupations transversales sont implantées respectivement en AspectC/C++ ou en AspectJ.

TABLE 2.2 – Récapitulation des travaux visant la modélisation ou l'implantation des techniques de la tolérance aux pannes avec la séparation des préoccupations

Approche	**Techniques de TAP adopté**	**Fautes traitées**	**Langage utilisé**	**Evaluation**	**Temps-réel**	**Dynamisme**	**Embarqué**
Jalilian *et al* [JSKT11]	EDAC, TRE CFC, AT	Fautes transitoires	AspectC++	-	oui	non	oui
Alexanderrson *et al* [AK11]	Détection et masquage d'erreurs	Fautes transitoire et intermittente	AspectC++ Ext/Opt	Injection de fautes	oui	non	oui
Costan *et al* [CDP+10]	Réplications active et passive	Fautes de communication et les retards de transmission	-	NCIT testbed	oui	oui	non
Hameed *et al* [HWS09]	Gestion des exceptions et contrôle de vraisemblance	Fautes Permanentes	AspectC++	Injection de fautes	non	oui	non
Afonso *et al* [ASB+08]	Techniques de diversité RB, DRB, NVP	Fautes logicielles et matérielles	AspectC++	Test d'utilisation du CPU et de mémoire	oui	oui	oui
Szentivanyi *et al* [SN04]	Réplication passive dans une architecture CORBA	Fautes logicielles	AspectJ	évaluation quantitative des gains de performance	non	non	non

Après une étude approfondie, il s'est avéré que le langage Ada est aussi classé parmi les meilleurs langages permettant le développement des systèmes critiques. Toutefois, son extension d'aspect AspectAda n'est jamais testée dans ce domaine. Pour cela, nous nous intéressons dans la section suivante à l'étude des différents langages d'aspects pour le développement temps-réel.

2.4 Langages orientés aspects pour le temps-réel

De nombreux langages de programmation, qu'ils soient procéduraux ou orientés objet, ont été étendus par les aspects pour éviter la dispersion et l'enchevêtrement du code et donc améliorer la réutilisation et la modularisation. La programmation orientée aspect est un ensemble d'approches et de technologies apparues dans le but d'implanter séparément les préoccupations transversales. Sa méthode de programmation met en faveur la modularité et la ré-utilisabilité en séparant l'implantation du code fonctionnel des aspects et en spécifiant leur composition et coordination en se basant sur un ensemble de règles. L'opération de tissage vise à intégrer automatiquement les aspects créés séparément dans le code fonctionnel. Étant donné que la conception et l'implantation des systèmes temps-réel est considérablement différentes des systèmes classiques, l'utilisation du paradigme orienté aspect dans le contexte des systèmes temps-réel est devenu un grand défi à cause des fortes contraintes devant être satisfaites pour garantir un niveau de fiabilité et de sécurité souhaité. Dans notre contexte, nous nous focalisons sur l'intégration des aspects dans le domaine temps-réel. En effet, l'opération de tissage consiste à insérer des bouts de codes appelés greffons (*Advice*) dans le code métier d'une application. Ces bouts de code insérés automatiquement peuvent compromettre le déterminisme d'un système temps-réel et violer les contraintes temps-réel puisqu'ils ne subissent aucune analyse et échappent à tout contrôle du développeur.

Dans le reste de ce chapitre, nous discutons des travaux visant l'intégration de la POA dans le domaine temps-réel en se basant sur différents langages d'aspects.

Pour le développement logiciel, il conviendrait d'accorder plus d'attention aux systèmes temps-réel critiques. Le code de l'application devra respecter certaines restrictions pour l'insertion des instructions considérées dangereuses pour le temps-réel. Une des plus connues restrictions est l'interdiction de l'allocation dynamique de mémoire pouvant être source de non déterminisme [Pua02]. En outre, la compatibilité de la programmation orientée objet avec les systèmes temps-réel reste un sujet à discuter. Pour remédier à ces problèmes, [HH03] ont proposé un ensemble de recommandations pour bénéficier des avantages de la POO lors du développement des systèmes temps-réel comme l'héritage et l'encapsulation et ont interdit certains concepts comme le polymorphisme ou l'acheminement dynamique des données (*dynamic dispatch*). Ces restrictions doivent être évitées quelque soit le type du langage : procédural, orienté objet ou même orienté aspect.

Pour ces raisons, les deux langages C++ et Java sont considérés mal adaptés pour le développement temps-réel. Toutefois, le langage Java a été étendu par une spécification dédiée pour le développement temps-réel (*Real Time Specification for Java*).

2.4.1 RTSJ et AspectJ pour le développement temps-réel

La spécification *Real-Time Specification for Java (RTSJ)* [JG00] a été proposée par *Real-Time for Java Expert Group (RTJEG)* dans le but d'étendre les deux spécifications du langage Java et de la JVM (*Java Virtual Machine*) afin de supporter les systèmes temps-réel à travers d'une API qui permettra la création, la vérification, l'analyse, l'exécution et la gestion des threads Java. Ces threads connus sous le nom *real-time threads* prennent en compte les contraintes temporelles. Cette spécification tourne autour de sept concepts qui sont :

— Ordonnancement et dispatch des threads

— Gestion de mémoire

— Synchronisation et partage de ressources

— Gestion des événements asynchrones

— Transfert de contrôle asynchrone

— Terminaison de thread asynchrone

— Accès à la mémoire physique

Chacun de ces concepts a été détaillé au niveau de la spécification pour mieux considérer les systèmes temps-réel développés avec le langage Java. À titre d'exemple, pour l'allocation dynamique de mémoire, la spécification RTSJ a été conçue d'une manière indépendante de tout algorithme de ramasse-miettes (*Garbage Collector (GC)*) particulier. De plus, elle permet au programme de caractériser précisément l'effet de l'algorithme GC mis en œuvre sur le temps d'exécution, la préemption, et l'exécution des threads temps-réel et de garantir que l'allocation et la désallocation des objets se fait en dehors de toute inférence avec un algorithme de GC.

En se basant sur la comparaison des deux paradigmes orienté objet et orienté aspect avec le langage Java, Tsang *et al* [TCB04] ont effectué une évaluation de la POA pour les systèmes embarqués temps-réel. Ils ont comparé les deux approches à travers un simulateur sensible au trafic choisi comme cas d'étude. Ils ont appliqué en premier lieu l'approche orientée objet pour le développement du simulateur qu'ils ont appelé OOSim en utilisant la spécification RTSJ. En deuxième lieu, le même simulateur appelé AOSim a été développé à la base de la POA. Les deux approches ont été par la suite combinées dans le but d'évaluer les propriétés du système comme la fiabilité. Les résultats prouvent que la POA a amélioré la modularité grâce à l'utilisation des wildcards. Ceci est du réellement à la réduction du couplage et de la cohésion entre les sept domaines améliorés de RTSJ. Les auteurs ont également prouvé que

retarder l'encapsulation des préoccupations temps-réel Java en des aspects permettra de réduire l'impact négatif sur la compréhensibilité et la maintenabilité puisque l'encapsulation des préoccupations non transversales affecte négativement ces deux métriques. Toutefois, AspectJ est une extension d'aspect pour le langage Java. Pour les systèmes temps-réel critiques et complexes, il faut fournir un grand effort pour implanter les concepts temps-réel et les utiliser au niveau des aspects (AspectJ).

2.4.2 C++ et son extension d'aspect pour le développement temps-réel

Une telle comparaison a été de même établie pour évaluer l'effet de la POA sur les systèmes temps-réel en se basant sur le langage C++. Kartal *et al* [KS07] ont utilisé le langage C++ pour implanter les préoccupations fonctionnelles d'un commutateur audio sélectionné comme un système embarqué temps-réel. Puis, ils se sont basés sur les deux approches orientée objet (avec C++) et orientée aspect (avec AspectC++) pour enrichir ce système par des préoccupations transversales comme l'enregistrement et le timing. Pour la comparaison des deux approches, ils ont évalué les deux approches à l'égard des métriques de performance caractérisant les systèmes embarqués temps-réel. En tête de la liste, nous citons la consommation mémoire et l'utilisation CPU. Les résultats ont prouvé que la POA a amélioré les performances du système non seulement au niveau des métriques de performance mais aussi au niveau des attributs de la qualité du logiciel. Cette amélioration revient à la suppression des appels de messages et des créations d'instances de classes. Ainsi, les auteurs ont prouvé l'efficacité du langage AspectC++ pour le développement temps-réel.

À la base du langage C++, Gal *et al* [GSSP02] ont affirmé aussi que la POA est à considérer une technique bien adaptée pour la conception et le développement même des systèmes temps-réel distribués fiables. Ils attestent que la POA permet d'éviter l'enchevêtrement du code source des composants temps-réel et de diminuer les coûts supplémentaires (*overhead*) si ces composants sont utilisés en dehors de ce domaine. Ces résultats sont établis à travers différents exemples dont les aspects sont intégrés afin de suivre la trace d'exécution sans avoir recours à une mise à jour manuelle du code et donc minimiser le risque d'erreur. Parmi lesquels, nous citons le système de surveillance du temps d'exécution et celui du robot mobile dont les aspects sont implantés dans le but de superviser les messages envoyés par le robot. Comme celui qui le précède, ce travail a utilisé AspectC++ pour l'implantation des aspects et le langage C++ pour l'implantation des préoccupations fonctionnelles. Les programmeurs montrent à ce niveau la possibilité d'étendre facilement des aspects en se basant sur le concept d'héritage d'aspect. Enfin, un troisième exemple d'accès client à distance a été mis en place via l'intergiciel CORBA, dans le but de prouver que la POA peut supporter plusieurs plates-formes différentes en mettant en œuvre les aspects dédiés pour chaque intergiciel.

Toutefois, le langage C++, le langage de base de AspectC++, reste toujours considéré

comme un langage bas-niveau puisqu'il permet un accès direct aux composants matériels. En outre, les deux langages C++ et son extension d'aspect AspectC++, peuvent être utilisés pour implanter les systèmes temps-réel sous réserve de les enrichir par des bibliothèques spécifiques supportant les concepts temps-réel comme la synchronisation, le timing, l'ordonnancement et les accès concurrents.

2.4.3 Synthèse

Les approches précitées ont prouvé l'efficacité de l'utilisation de la POA dans le domaine temps-réel embarqué en utilisant les deux langages AspectJ et AspectC++ extensions respectives des langages de programmation Java et C++. Cependant, ces langages ne sont ni idéals pour tous types d'applications embarquées temps-réel critiques ni utilisés pour développer les systèmes complexes qui sont présents dans notre vie quotidienne comme le langage Ada.

Pour toutes ces raisons, nous affirmons qu'une extension d'aspect pour ce langage sera sans doute bien adaptée pour implanter et ordonnancer les systèmes temps-réel embarqués critiques. Dans ce contexte, Pederson *et al* [PC05] ont proposé le langage AspectAda, extension du langage Ada95 par les concepts d'aspects, et son compilateur prototype afin d'améliorer l'adaptabilité et la concurrence des systèmes temps-réel critiques par la séparation et la modularisation des préoccupations. Pour cela, nous avons étudié ce langage afin de tester sa conformité avec le développement temps-réel. Les résultats sont discutés au niveau du chapitre 8.

2.5 Conclusion

Dans ce chapitre, nous avons présenté les travaux connexes à nos travaux et au fur à mesure nous avons fixé les objectifs de cette thèse. Nous avons également abordé la problématique de modélisation et de développement des mécanismes de tolérance aux pannes pour les systèmes temps-réel qui sont des systèmes particuliers en termes de contraintes et de consommation de ressources. Á travers une étude comparative de certains ADLs existants visant la modélisation de tels systèmes, nous avons choisi le langage AADL pour la modélisation de notre système et de l'étendre pour supporter la réplication automatique des composants AADL. Par ailleurs, nous avons étudié dans ce chapitre, les travaux proposés dans la littérature pour intégrer la POA au niveau modèle aussi bien qu'au niveau code à des fins de modularisation et de séparation des préoccupations considérant la tolérance aux pannes comme une préoccupation transversale. La synthèse de ces travaux nous a permis de conclure que l'utilisation de la POA a un impact positif sur la qualité du code, la modularisation et la maintenance même pour les systèmes temps-réel. Ceci est prouvé pour différents langages d'aspect. Toutefois, le caractère critique de ces systèmes interdit l'utilisation de certains langages ou plus particulièrement de certaines constructions pouvant provoquer le non

déterminisme. Dans ce contexte, puisque le langage Ada est bien adapté pour le développement temps-réel critique, nous avons décidé d'évaluer l'efficacité du langage AspectAda pour le domaine temps-réel et son impact sur les contraintes temps-réel.

Cette thèse vise alors à proposer des solutions pour cette problématique. Nous proposons tout un processus de modélisation et de développement des systèmes temps-réel distribués dynamiquement reconfigurables tolérant aux pannes basé sur la séparation des préoccupations sur les deux niveaux. Tout d'abord, nous proposons d'adopter le langage AADL pour la conception des systèmes temps-réel distribués dynamiquement reconfigurables. Nous adoptons également l'annexe EMA pour la modélisation des mécanismes de la tolérance aux pannes. Pour supporter la réplication automatique des composants AADL, nous définissons notre propre approche basée sur l'extension du langage AADL par les propriétés. Cette contribution permettra d'éviter les risques d'erreurs et la perte de temps de conception. Ensuite, en nous basant sur cette première contribution, nous introduisons un processus de génération de code pour les préoccupations fonctionnelles et celles transversales. Nous nous intéressons plus particulièrement à la tolérance aux pannes dont nous générons le code correspondant en AspectAda en suivant la démarche MDA. Finalement, nous évaluons le langage AspectAda existant et nous proposons une adaptation et extension pour le respect des contraintes temps-réel.

3

Proposition d'une approche de tolérance aux pannes pour les systèmes temps-réel distribués

3.1 Introduction

L'un des principaux objectifs des langages de description d'architecture est de permettre des vérifications de modèles très tôt dans le processus de développement. La spécification et la vérification du comportement de l'application est sans doute un défi majeur pour les systèmes critiques. Bien que certains aspects comportementaux peuvent être décrits avec le noyau des ADLs, d'autres comptent sur l'utilisation des extensions comme les annexes ou les profils. Dans l'état de l'art, nous avons présenté certains ADLs et leurs extensions qui proposent de modéliser la sûreté de fonctionnement pour les systèmes temps-réel distribués. Nous avons aussi noté un manque d'outils permettant de générer le code séparé correspondant à partir de ces ADLs et extensions. En particulier, nous avons mis l'accent sur les techniques de tolérance aux pannes mises en œuvre comme la réplication. La plupart des travaux existants supportent un seul style réplication gérée manuellement. Enfin, nous avons aussi étudié l'implantation de la tolérance aux pannes avec des langages d'aspects et la compatibilité de ces derniers avec les systèmes temps-réel.

Dans ce chapitre, nous donnons un aperçu global de notre approche générique permettant la modélisation et l'implantation séparées de la tolérance aux pannes pour les systèmes temps-réel distribués. Nous commençons par expliquer notre processus de développement proposé que nous appelons DP4FTRTS (*Development Process For Fault Tolerant Real-Time Systems*) pour la prise en compte explicite de la tolérance aux pannes séparément au niveau de

toutes les phases de développement. Dans le but d'intégrer dès la phase de modélisation des éléments de tolérance aux pannes, comme la réplication, la détection et le recouvrement, nous utilisons un langage de description d'architecture dédié pour la modélisation des systèmes temps-réel. Nous proposons d'étendre le langage AADL par les aspects de tolérance aux pannes visant d'une part la génération de code et d'autre part l'analyse de la sûreté de fonctionnement. Étant donné que la tolérance aux pannes est une préoccupation transversale, nous nous basons sur la programmation orientée aspect pour l'implantation de ses différents concepts.

Ce chapitre est structuré comme suit. La première section explique notre processus proposé DP4FTRTS qui explique l'enchaînement chronologique des contributions détaillées après. La deuxième section est consacrée pour donner un aperçu sur les concepts de base devant figurer au niveau du modèle tolérant aux pannes. Ensuite, nous décrivons notre approche proposée pour la génération de code. Puis, nous expliquons les motivations pour l'adaptation du langage d'aspect pour le développement temps-réel. Enfin, nous justifions le choix du langage AADL dont nous détaillerons ces concepts de base dans le chapitre suivant.

3.2 Processus DP4FTRTS

À l'égard de l'évolution technologique et de la complexité des systèmes émergeant dans de nombreux domaines de notre vie quotidienne, nous remarquons toujours l'apparition de nouveaux besoins visant répondre aux exigences fonctionnelles aussi bien que celles non fonctionnelles comme la sécurité et la tolérance aux pannes. Les systèmes temps-réel critiques tolérants aux pannes sont de plus en plus omniprésents dans notre vie quotidienne dans les domaines de télécommunication, le transport et le domaine médical. Les menaces de la sûreté de fonctionnement qui sont les fautes, les erreurs et les pannes, ont des conséquences potentiellement catastrophiques sur différentes échelles. Elles peuvent causer la perte de temps, d'argent et même la perte de vie humaine dans le pire des cas. Les fautes susceptibles d'affecter un système sont classées selon plusieurs critères d'une manière orthogonale comme déjà mentionnée au niveau de la section 1.3.3. Dans notre contexte, nous nous limitons à un sous-ensemble de fautes que nous envisageons traiter.

3.2.1 Sous-ensemble de fautes considéré

Parmi les différentes classes de fautes définies par la littérature, nous sélectionnons un sous-ensemble afin d'offrir les moyens et techniques pour sa détection et faire face en appliquant les mécanismes de recouvrement adéquats. Dans cette section, nous rappelons les critères de classification tout en mettant l'accent sur les types sélectionnés. Nous classifions les fautes selon :

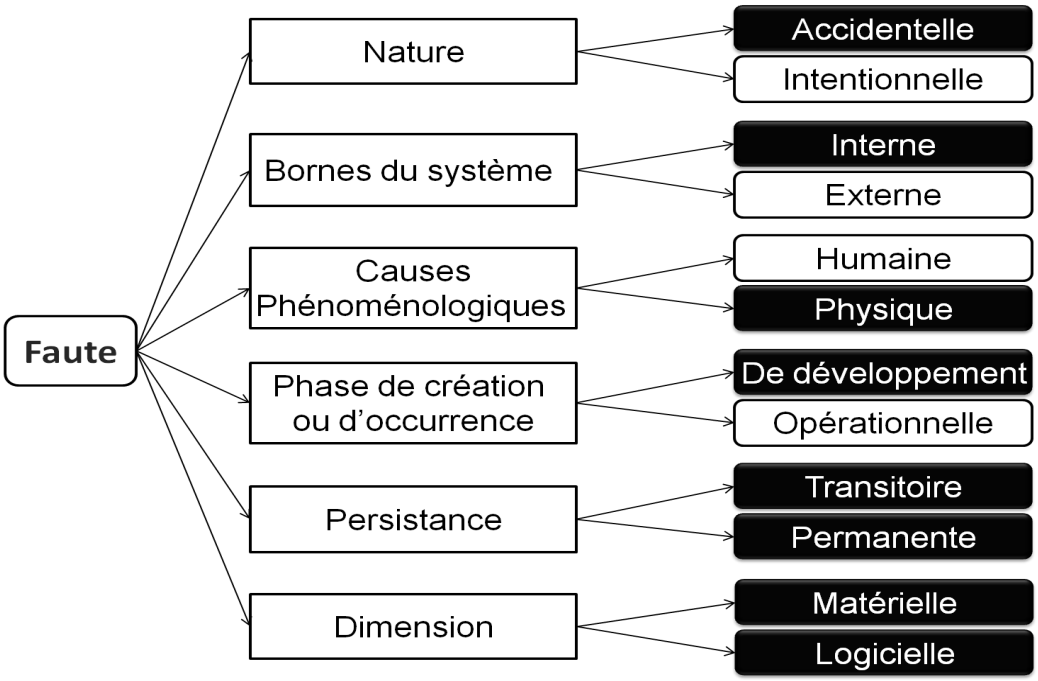

FIGURE 3.1 – Classification des fautes

1. Nature : Une faute est de nature accidentelle ou intentionnelle. La faute intentionnelle (malveillante) est introduite par un être humain dans le but de causer des dommages au système.
2. Origine : Selon l'origine, les fautes sont classées selon trois autres sous-classes.
   (a) Selon les bornes de système, une faute est soit externe soit interne localisée à l'intérieur des frontières du système.
   (b) Si on se concentre sur les causes phénoménologiques, une faute est soit physique due à des phénomènes naturels, sans intervention humaine directe soit humaine.
   (c) Phase de création ou d'occurrence : Une faute est considérée de développement si elle est créée durant le développement du système, y compris la génération de code. Si le système a commencé son exécution, une faute est alors considérée opérationnelle puisqu'elle survient durant la fourniture du service au cours de la vie opérationnelle.
3. Dimension : Une faute est matérielle si elle se manifeste dans le matériel. Par contre, si la faute affecte le logiciel, les programmes ou les données, elle est considérée logicielle.
4. Persistance : Une faute transitoire est une faute dont la présence est temporellement bornée contrairement au faute permanente dont sa présence est continue.

Dans notre contexte, nous avons sélectionné un sous-ensemble de fautes que nous envisageons traiter comme l'indique la figure 3.1. Nous nous intéressons à la tolérance aux

pannes, donc nous considérons uniquement les fautes accidentelles puisque celles intentionnelles sont plutôt liées à la sécurité. Nous adoptons les systèmes embarqués dynamiquement reconfigurables donc nous traitons les fautes interne et physique. Nous adoptons aussi les systèmes temps-réel, donc nous envisageons les deux types de fautes, transitoire et permanente puisque le service demandé doit être délivré à temps dans le cadre de ces systèmes. Nous traitons également les fautes opérationnelles visant à offrir des mécanismes de rétablissement au cours de l'exécution en se basant sur la reconfiguration dynamique. Finalement, nous nous intéressons aux fautes logicielles affectant le logiciel, les programmes ou les données et les fautes matérielles afin de détecter les défaillances des capteurs et donc éviter la collecte des données erronées.

Vis à vis de la variété des types de fautes pouvant affecter les systèmes critiques, ces derniers doivent répondre au maximum aux exigences de tolérance aux pannes et plus généralement de sûreté de fonctionnement. Pour faire face à cette problématique, ces exigences doivent être considérées tout au long du cycle de développement. Dans cette orientation, nous avons proposé un processus de développement des systèmes temps-réel tolérants aux pannes décrivant les différentes étapes à suivre allant de la modélisation jusqu'à la génération de code en se basant sur la séparation des préoccupations sur tous les niveaux.

3.2.2 Vue d'ensemble du processus DP4FTRTS

Comme la conception de systèmes temps-réel distribués est loin d'être triviale, plusieurs travaux ont ciblé la modélisation de ses systèmes à des fins d'analyse et de vérification. En contre partie, pour la génération du code tolérant aux pannes, il y a peu de travaux focalisant sur la génération à partir du modèle. D'autres travaux se concentrent sur l'implantation des mécanismes de tolérance aux pannes en utilisant la POA sans avoir recours au modèle. Pour cela, nous avons proposé un processus de développement des systèmes temps-réel distribués dynamiquement reconfigurables tolérants aux pannes. Ce processus appelé, DP4FTRTS (*Development Process For Fault Tolerant Real-Time Systems*) est basé sur la séparation des préoccupations sur tous les niveaux en faisant appel à la démarche MDA. La figure 3.2 met en évidence les étapes du processus de production auxquelles nous nous intéressons dans ce chapitre. Il s'agit d'un ensemble d'étapes allant de la modélisation du système de base, passant par des étapes intermédiaires visant la conception et l'implantation des préoccupations non fonctionnelles jusqu'à la génération du code applicatif pour tout le système.

Comme première étape, le concepteur utilise un ADL connu pour la modélisation du système. Il s'agit de décrire l'architecture globale du système en spécifiant les différents composants qui le constituent et les connexions qui les relient. A ce niveau, plusieurs ADLs peuvent être choisis comme le langage UML ou AADL. Dans une seconde étape, ce modèle architectural est étendu par une extension séparée à des fins de tolérance aux pannes. Cette dernière est considérée comme une préoccupation transversale ce qui a favorisé sa sépara-

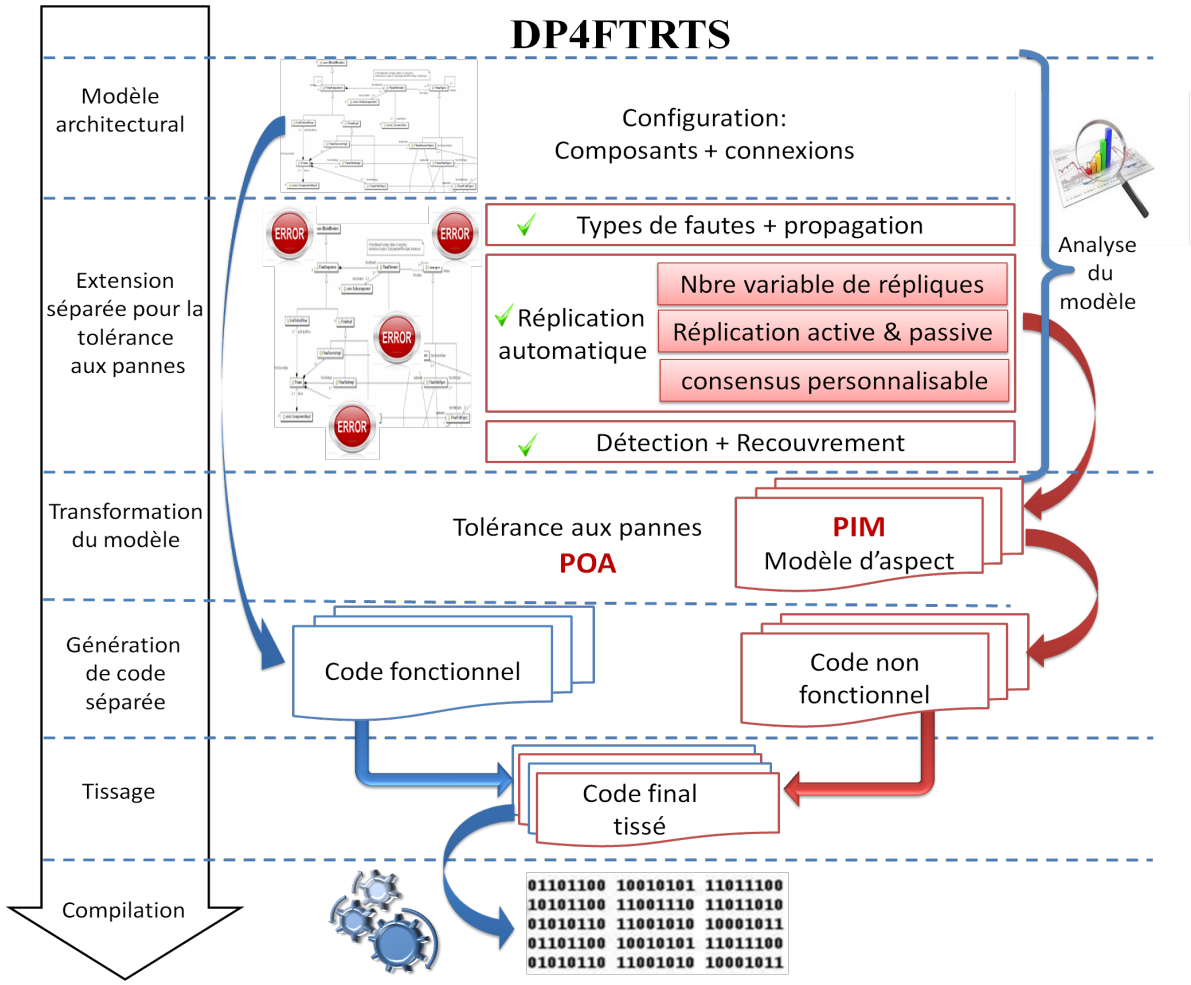

FIGURE 3.2 – Processus de développement proposé

tion des préoccupations fonctionnelles durant le processus de développement. Pour cette raison, nous envisageons étendre le modèle architectural par un modèle d'erreurs ajoutant séparément des concepts relatifs à la tolérance aux pannes. L'idée est inspirée des extensions fournies pour les ADLs standards. Nous citons à titre d'exemples le profil MARTE-MARTE-DAM, EMA et HIP-HOPS extensions respectives des ADLs UML, AADL et EAST-ADL. Ce modèle est crée dans le but de donner des détails sur la tolérance aux pannes et plus généralement sur la sûreté de fonctionnement à savoir les types de fautes pouvant affecter le système et leurs modes de propagation à travers ses composants. Le concepteur doit spécifier à ce niveau les composants concernés par des types spécifiques de fautes, les techniques de détection et de recouvrement dédiées pour chacun de ces types. Parmi les limites des extensions ci-citées, nous avons noté la réplication manuelle et explicite des composants ce qui requiert un effort considérable surtout dans le cas d'un grand nombre de répliques ou de composants à répliquer. De même, une telle réplication peut engendrer une perte de temps

de conception et un grand risque d'erreur. Pour remédier à ces problèmes, nous proposons supporter la réplication automatique des composants. Nous visons aider le concepteur pour générer un modèle supportant la réplication automatique de composants tout en considérant un nombre variable de répliques, deux styles de réplication et un algorithme de consensus prédéfini ou personnalisé selon le besoin. Ainsi, le modèle architectural étendu peut faire l'objet des analyses ou des vérifications à des fins de sûreté de fonctionnement. À partir de ce modèle, le concepteur peut utiliser des outils ou des techniques existants pour faire des mesures des attributs de la sûreté de fonctionnement en passant par des formalismes intermédiaires tels que les réseaux de Petri ou les arbres de fautes.

Quant à la génération du code à partir du modèle, nous proposons suivre la démarche MDA basée sur le raffinement des modèles. Il s'agit d'appliquer des règles de transformation sur le modèle tolérant aux pannes séparé pour en générer un modèle d'aspect indépendant de la plate-forme (PIM). Visant garder la séparation de la tolérance aux pannes des préoccupations fonctionnelles dans toutes les phases du processus, nous avons proposé de générer le code tolérant aux pannes à partir du modèle tolérant aux pannes en profitant de la programmation orientée aspect. Ainsi, le modèle PIM sera un modèle d'aspect enrichissant le modèle architectural par des aspects. Pour assurer la tolérance aux pannes, ce modèle permet d'intercepter les composants pouvant être affectés par les fautes déclarées ou propagées via ces interfaces. La détection des pannes déclenche des actions de reconfiguration dans le but de rétablissement. Le modèle PIM joue le rôle d'un modèle intermédiaire donnant lieu par la suite à une génération de code d'aspect dépendamment du langage choisi pour le code fonctionnel. Après la génération du code tolérant aux pannes vers le langage d'aspect adéquat, il sera automatiquement tissé avec le code fonctionnel généré aussi automatiquement. Enfin, le code applicatif tissé est prêt à être exécuté après avoir être compilé.

En fonction des nouveaux besoins de l'application et des nouvelles exigences de son environnement, soit le modèle architectural soit le modèle tolérant aux pannes sera modifié sans affecter l'autre. Grâce à la séparation des préoccupations, même le processus de génération de code sera partiellement mis à jour, c'est-à-dire uniquement le modèle modifié sera la source de génération.

Dans ce qui suit, nous détaillons les besoins auxquels le modèle tolérant aux pannes doit répondre.

3.3 Modèle tolérant aux pannes

Un modèle architectural dédié à un système temps-réel distribué décrit uniquement les composants qui le constituent et les connexions qui les relient. Si le modèle architectural est riche, il comporte des propriétés décrivant ses caractéristiques temporelles ou ses données. Le concepteur peut bénéficier également de certaines propriétés des ADLs permettant la description des caractéristiques d'un système temps-réel comme les périodes des processus

légers, leurs priorités, les algorithmes d'ordonnancements, etc. En ce qui concerne le modèle tolérant aux pannes, il doit fournir des informations supplémentaires pour la détection des pannes prévues et le rétablissement du système sans toucher l'hiérarchie ou l'organisation du modèle architectural. Ce modèle doit être capable de décrire divers concepts primordiaux pour la tolérance aux pannes tels que la description des menaces de la sûreté de fonctionnement, leur propagation et leur détection, le rétablissement du système, la gestion de réplication et l'analyse et la vérification du système. Toutes ces exigences sont détaillées dans les sections qui suivent.

3.3.1 Description et propagation des menaces de la tolérance aux pannes

L'extension du modèle architectural d'un système par un modèle tolérant aux pannes est effectuée dans le but de le rendre apte de détecter les menaces qui provoquent une dégradation du service ou une déviation du système par rapport à sa spécification. De ce fait, ces menaces devront être identifiées et prévues même avant l'exécution du système. Bien que plusieurs modèles d'erreurs liés à des ADLs standards permettent la déclaration des types ou des ensembles de fautes pouvant affecter les composants logiciels ou matériels du système et éventuellement ses relations (propagation), chacun d'eux définit un sous ensemble particuliers et restreints de fautes prédéfinies comme celles relatives aux valeurs, à la réplication de composants, au temps de livraison, etc. Pour cette raison, le modèle d'erreurs proposé permettra de définir de nouveaux types d'erreurs d'une manière abstraite ou concrète dans certains cas. Ces types de fautes déclarées serviront par la suite dans l'étude de la propagation des erreurs entre composants via les interfaces ou dans la spécification des comportements des composants en cas de détection de telles erreurs qui seront référencées par leur types.

3.3.2 Détection des menaces et rétablissement du système

Après avoir défini les menaces prévues de la sûreté de fonctionnement du système sous forme de types et des ensembles d'erreurs, le système sera capable de les détecter à travers divers mécanismes. Au niveau modèle, le concepteur décrit les mécanismes de détection souhaités pour chaque composant en question et relativement aux types d'erreurs désignés. La plupart des modèles d'erreurs existants se sont basés sur l'évaluation logique des conditions pour la détection des erreurs. Le modèle tolérant aux pannes doit donc offrir les moyens nécessaires pour la description et l'évaluation des conditions donnant lieu à une détection des erreurs déclarées au niveau des composants. Suite à la détection des erreurs, le système tolérant aux pannes exige des actions souvent de reconfiguration pour le rétablissement du système en cours d'exécution. Ses actions reflètent des changements anticipés en cas d'occurrence de pannes particulières décrites dès le niveau modèle. Les modèles d'erreurs permettent la planification des actions de reconfiguration permettant à leur

tour d'éliminer les états erronés du système. Ces actions sont déclenchées suite à la détection des erreurs prévues pour donner lieu à une transition entre des configurations qui sont planifiées à l'avance. Pour la mise en place des actions de reconfiguration, les modèles d'erreurs font appel aux machines à états pouvant ainsi fixer les configurations possibles et les comportements du système en cas d'erreur.

3.3.3 Gestion de réplication

Une technique très connue et utilisée pour mettre en œuvre les systèmes tolérants aux pannes est la réplication. Cette technique est gérée au niveau modèle d'une manière explicite ou implicite selon le type de l'ADL choisi et son modèle d'erreurs ou tolérant aux pannes correspondant. La réplication dépend du type du composant répliqué (logiciel ou matériel), du nombre de répliques demandé, de la stratégie de réplication (active ou passive ou hybride) et de l'algorithme de consensus.
Comme il est déjà discuté dans la section 2.3.2, certains modèles d'erreurs supportent une réplication manuelle avec un nombre fixe de répliques et un style particulier de réplication. Une telle réplication requiert un effort considérable de la part du concepteur, une perte du temps de conception et un risque d'erreur dès que le nombre de répliques ou de composants à répliquer devient très important. D'autres modèles tolérants aux pannes considèrent plusieurs styles de réplication dans un même modèle mais pour différents composants. Pour offrir un modèle tolérant au pannes riche en termes de réplication, il est devenu indispensable de supporter une réplication automatique des composants avec un nombre variable de répliques et les deux principaux styles de réplication au sein d'un même modèle. Dans cette orientation, considérer différents algorithmes de consensus sera d'une importance cruciale si nous souhaitons effectuer une réplication automatique avec divers styles. Le concepteur dans ce cas peut bénéficier des algorithmes prédéfinis comme le vote majoritaire ou définir d'autres selon le besoin. Ces différentes propriétés seront considérées à travers un ensemble de propriétés enrichissant le modèle architectural. À partir de ces propriétés, le modèle tolérant aux pannes, contenant les répliques et les composants de votes ou d'élection, sera automatiquement généré tout en établissant les connexions qui les relient. La génération automatique du modèle de réplication prend en considération plusieurs facteurs. Nous citons en tête de la liste de composants répliqués, le style de réplication et le nombre de répliques. Pour cette raison, un ensemble d'algorithmes devrait être établi pour la génération automatique du modèle de répliques tenant en compte de tous ces facteurs.

3.3.4 Analyse et vérification

La définition d'un modèle tolérant aux pannes séparé de celui architectural a pour objectif de séparer les préoccupations non fonctionnelles des autres fonctionnelles durant tout le cycle de développement. Ceci est dans le but d'automatiser la génération de code à partir

de modèles, d'une part, et de faciliter l'analyse des propriétés non fonctionnelles du logiciel en cours de développement sur la base de ses modèles, d'autre part [BMP12]. Parmi les propriétés non fonctionnelles, nous distinguons la fiabilité, la disponibilité, l'intégrité et la maintenabilité qui touchent le domaine de la tolérance aux pannes. Dans le contexte de la démarche MDA, une approche visant l'analyse des différentes propriétés non fonctionnelles consiste à suivre les étapes suivantes : (a) étendre le modèle avec des annotations décrivant la propriété qui nous intéresse ; (b) transformer automatiquement le modèle annoté en un autre formalisme choisi pour l'analyse des propriétés non fonctionnelles ; (c) analyser le modèle formel en utilisant des outils existants et (d) évaluer le système en se basant sur les résultats et donner des informations de retour (*feedback*) pour les concepteurs. La manière dont un modèle est annoté dépend fortement de l'ADL utilisé pour sa conception qui influence, à son tour, le choix du formalisme dédié à la vérification. Pour l'évaluation de la tolérance aux pannes, divers formalismes ont été utilisés comme les réseaux de Petri ou les arbres de fautes générés à partir des modèles d'erreurs. Un modèle tolérant aux pannes fournit certaines propriétés relatives à l'analyse quantitative (probabiliste) ou qualitative de la sûreté de fonctionnement des systèmes mis en jeu à savoir la probabilité de défaillance du système complet ou de certains de ces composants.

3.4 Génération de code

La génération de code est un processus visant la production du code à partir d'un modèle. Plusieurs travaux se sont focalisés sur l'abstraction du modèle de conception dans l'objectif de fournir pour un seul modèle abstrait décrivant les fonctionnalités de l'application, plusieurs sorties correspondantes [Her03]. Les techniques de génération de code mettent l'accent sur l'automatisation de la transformation à partir d'un niveau supérieur de modèles décrits avec des ADLs. En se basant sur l'approche MDA, quatre types de transformations successives peuvent être appliquées tout au long du cycle de développement allant d'un modèle abstrait, passant par des modèles plus concrets (spécifiques à la plate-forme) jusqu'à l'obtention du code. Les modèles intermédiaires appelés PIM, décrivent le système indépendamment de toute plate-forme technique et de toute technologie utilisée pour déployer l'application. Puis, une transformation appliquée sur ce modèle donne naissance à un nouveau modèle spécifique à la plate-forme servant à générer le code exécutable pour des plate-formes techniques particulières.

Dans ce contexte, pour faciliter le développement des systèmes temps-réel distribués, nous nous basons pour la génération de code, sur un intergiciel aidant à générer une grande partie du code. Il s'agit de générer les composants invariants d'une application (des services communs pour toutes les applications) et de générer automatiquement le reste des composants à partir d'une description précise. Pour la génération des services de tolérance aux pannes pour les systèmes temps-réel, certains intergiciels comme [NDP+05]

ont été conçus pour faciliter la génération des systèmes temps-réel tolérants aux pannes. Ces intergiciels doivent satisfaire un ensemble de propriétés comme la modularisation, la flexibilité, le support de la reconfiguration dynamique afin de configurer et personnaliser finement les différents constituants générés. Ainsi, Un générateur de code par cible doit être construit au sein de l'intergiciel dédié.

À cet égard, nous adoptons l'approche MDA pour la mise en place de notre processus de génération du code tolérant aux pannes. A partir du modèle d'erreurs abstrait, nous générons un modèle indépendant de la plate-forme détaillant les mécanismes de tolérance aux pannes pris en compte. Nous avons adopté la programmation orientée aspect pour la génération séparée des concepts de la tolérance aux pannes. À partir du modèle d'erreurs, nous générons un modèle d'aspect indépendant de la plate-forme et jouant le rôle d'un modèle intermédiaire pour la génération du code tolérant aux pannes en langage d'aspect afin d'être tissé avec le code fonctionnel. Le code fonctionnel cible sera produit en se basant sur l'utilisation d'intergiciel existant dédié pour la génération des applications temps-réel.

3.5 Adaptation d'un langage d'aspect pour le respect des contraintes temps-réel

Comme nous l'avons noté au niveau de la section précédente, nous avons proposé l'intégration du paradigme orienté aspect dans notre processus de génération de code dans le but de séparer la tolérance aux pannes des préoccupations techniques. Dans notre contexte, nous nous intéressons au développement des systèmes temps-réel critiques. Considérant la spécificité de ces systèmes vus leur criticité et le nombre de contraintes qu'ils doivent respecter pour satisfaire les exigences temps-réel et le dynamisme rend le développement de ces systèmes une tâche fastidieuse. En effet, le code doit respecter certaines restrictions pour éviter que des constructions jugées dangereuses apparaissent. Typiquement, une des restrictions les plus connues pour les systèmes embarqués critiques est l'interdiction de l'allocation dynamique de mémoire puisqu'elle peut causer l'indéterminisme [Pua02]. Cette construction peut également être source de fuites de mémoire dues à des erreurs du programmeur ou parfois même du compilateur. Étant donné que la durée d'exécution de ce genre de fuites est très longue et que ces ressources en mémoire sont limitées, ces fuites ne sont pas tolérées dans le contexte des systèmes embarqués. De plus, certains concepts du paradigme orienté objet sont interdits pour les systèmes temps-réel comme le polymorphisme et l'aiguillage dynamique.

Notamment, nous avons discuté au niveau de la section 2.4, la compatibilité des langages d'aspects existants avec les systèmes temps-réel. En explorant la littérature, nous avons remarqué la violation de certaines contraintes pour des langages spécifiques. Il est donc indis-

pensable de vérifier si le langage d'aspect utilisé est bien adapté pour le développement des systèmes temps-réel pour éviter la violation de ces contraintes étant donnée la nature spécifique de ces types de systèmes en termes de caractéristiques et de contraintes de ressources.

Si le langage d'aspect est bien approprié pour le développement temps-réel, le code d'aspect généré tissé avec le code fonctionnel généré avec un intergiciel dédié pour les systèmes temps-réel respecte sans doute ses contraintes particulières. Ce code final tissé sera par la suite compilé pour être exécuté. De cette manière, la sûreté de fonctionnement de l'application ainsi que sa performance sont testées et garanties avant la construction du système et à partir des premières phases du cycle de développement.

En particulier, nous sélectionnons le langage Ada pour la génération du code fonctionnel puisqu'il est adapté et testé pour le développement des systèmes temps-réel critiques. Pour le langage d'aspect correspondant, AspectAda n'a pas été testé ni évalué pour développer de tels systèmes. Afin de vérifier son adéquation avec les contraintes temps-réel, nous avons effectué une étude du langage AspectAda existant dans le cadre de ce travail.

3.6 Choix du langage de description d'architecture

L'approche proposée pour la modélisation, l'implantation et la génération de code des systèmes temps-réel considère la tolérance aux pannes comme préoccupation fondamentale pour toutes les phases du cycle de développement. Étant donné qu'elle est considérée comme préoccupation transversale, nous nous sommes focalisés sur sa séparation des préoccupations techniques dès les premières phases du processus de développement, notamment la phase de modélisation. Le modèle sera donc extensible par d'autres concepts décrivant les types de fautes pouvant survenir, la propagation, les comportements des composants en cas d'erreur, la réplication, etc. Ainsi, l'apparition d'un nouveau besoin fonctionnel peut être modélisé sans toucher les exigences de tolérance aux pannes modélisées séparément et inversement. Cette approche exige aussi une phase d'analyse pour la mesure des attributs de la sûreté de fonctionnement et la vérification du système.

Nous considérons les systèmes temps-réel critiques ayant des exigences particulières devront être spécifiés dès le niveau modèle. Nous citons comme exemples la description des caractéristiques des tâches (priorité, période, etc), les plate-formes d'exécution (mémoire, processeur, bus, etc) et les types de données. Pour ce type de système, il est aussi indispensable d'avoir une syntaxe bien claire et rigoureuse pour servir les analyses demandées comme l'analyse d'ordonnancement.

Donc, la modélisation avec un ADL approprié met en évidence la spécification des exigences fonctionnelles et non fonctionnelles comme l'analyse du système au niveau modèle et la description des propriétés temps-réel. De plus, en tirant profil des intergiciels dédiés pour les systèmes temps-réel distribués, une large partie du code sera générée automatiquement tout en prenant en compte les contraintes de ressources. Pour le faire, nous avons besoin d'un

ADL pour décrire l'architecture détaillée du système temps-réel distribué et d'exprimer les exigences influant son analyse, sa configuration et la génération de code adéquate.

Par ailleurs, nous nous mettons l'accent durant le processus de développement non seulement sur les besoins fonctionnels mais aussi sur la tolérance aux pannes. À cet effet, le formalisme qui décrit l'application doit être capable de spécifier séparément les besoins en tolérance aux pannes à travers des extensions ou des propriétés donnant lieu à une analyse de la sûreté de fonctionnement dès le niveau modèle et aussi à une génération de code séparée de ces exigences. Parmi les ADLs adéquats pour modéliser toutes ces exigences, nous choisissons AADL.

Nous avons sélectionné le langage AADL parce qu'il possède toutes les caractéristiques dont nous avons besoin pour spécifier et générer automatiquement les systèmes temps-réel distribués tolérants aux pannes. En effet, ce langage a prouvé sa capacité pour le développement des systèmes embarqués temps-réel critiques. Testé dans différents domaines et surtout dans le domaine avionique, AADL est jugé comme un langage de description d'architecture pour les systèmes temps-réel de première classe. Il est classé le premier selon la compagnie aérienne AIRBUS [MLM+13]. De plus, AADL est un langage concret permettant de décrire toute une architecture d'un système grâce à la définition de trois types de composants : matériels, logiciels et hybrides. Son caractère extensible le rend aussi parmi les langages les plus utilisés. En effet, ce langage peut être enrichi par des propriétés s'appliquant aux entités constituant l'architecture afin de leur associer des contraintes ou des caractéristiques données. Ces propriétés sont standards prédéfinies dans un ensemble appelé *Property set* ou personnalisées définies par le concepteur.

Pour modéliser les préoccupations non fonctionnelles comme la tolérance aux pannes, le concepteur aura le choix parmi les propriétés ou les annexes. Ces dernières permettent d'étendre un modèle AADL avec des déclarations non architecturales exprimées avec un autre langage que AADL comme l'annexe d'erreurs EMA. Nous utilisons cette annexe dans le but de décrire le modèle d'erreurs architectural. Il s'agit de décrire des types de pannes pouvant affecter le système, la manière de détection de telles pannes et même les techniques de rétablissement utilisées. Cette annexe supporte plusieurs concepts de la tolérance aux pannes et plus généralement de la sûreté de fonctionnement. Elle admet une syntaxe bien définie pour effectuer des analyses de la sûreté de fonctionnement en se basant sur des transformations en arbre de fautes ou en réseaux de Petri.

3.7 Conclusion

Dans ce chapitre, nous avons présenté le processus de développement DP4FTRTS que nous avons défini pour la modélisation et l'implantation des systèmes temps-réel distribués tolérants aux pannes. Nous avons décrit d'une manière générique toutes les étapes du processus. En partant d'un modèle architectural décrit avec un ADL et enrichi avec des concepts

de la tolérance aux pannes, nous procédons non seulement à l'analyse de la sûreté de fonctionnement mais aussi à la génération de code de chacune des exigences fonctionnelles et non fonctionnelles en suivant la démarche MDA. Après avoir généré un modèle intermédiaire indépendant de la plate-forme, nous donnons lieu à une génération de code séparée de la tolérance aux pannes des autres préoccupations en profitant des avantages de la programmation orientée aspect.

Dans le chapitre suivant, nous présentons, en détail, les constructions du langage AADL nécessaires pour la modélisation et la configuration de nos systèmes tout en mettant l'accent sur les annexes utilisées durant notre processus de développement.

4

Introduction au langage AADL

4.1 Introduction

Dans le chapitre précédent, nous avons défini une approche générique de modélisation, d'analyse et d'implantation de la tolérance aux pannes pour les systèmes temps-réel distribués. Il s'agit d'un processus de développement qui renferme des étapes ordonnées afin de diminuer les risques d'erreurs et de masquer la complexité de la production. La première étape de ce processus est la modélisation de ces systèmes avec le langage AADL. Le choix de ce langage a été effectué suite à une étude comparative de plusieurs ADLs comme il est justifié dans la section 3.6.

Ce chapitre donne une introduction au langage AADL et particulièrement à ses aspects dont nous avons besoin pour la spécification, la configuration et l'analyse d'applications temps-réel distribués tolérants aux pannes pour faciliter la compréhension de la suite de ce mémoire.

La suite de ce chapitre est organisé comme suit : dans la section 4.2, nous introduisons le langage AADL et ses concepts de base. Nous distinguons les différentes catégories de composants, l'hiérarchie en sous-composants et appels, les interfaces et les connexions reliant les composants, les propriétés et les annexes et finalement les modes et les transitions. La section 4.3 est consacrée pour décrire les différentes annexes du langage AADL auxquelles nous avons besoin pour comprendre le reste de ce mémoire.

4.2 AADL

AADL (*Avionics Architecture Description Language*) est un ADL conçu initialement pour répondre aux besoins des systèmes avioniques. Il a été par la suite standardisé et publié par la SAE (*Society of Automotive Engineers*) pour satisfaire les besoins de tous systèmes temps-réel

embarqués. Pour cette raison, son acronyme a été modifié en *Architecture Analysis& Design Language* pour mettre en évidence cette considération. AADL est publié en deux versions. AADL 1.0 est la première version publiée en Octobre 2004 [SAE04] et la deuxième AADLV2 est publiée en Janvier 2009 [SAE12]. Les deux versions sont dédiées à l'analyse et la conception d'architectures logicielles et matérielles des systèmes fondés sur la notion de composants. Dés lors, AADL, similaire aux autres ADLs, permet de décrire un système comme un ensemble de composants interconnectés. La spécificité de ce langage réside dans la variété des éléments qu'il peut modéliser. En effet, il décrit non seulement ceux architecturaux mais aussi il est capable de décrire les éléments non architecturaux liés aux métriques de performances critiques telles que les exigences temporelles. Outre la représentation graphique, AADL offre une représentation textuelle et une autre sous format XML (*eXtensible Markup Language*). Un concepteur peut adapter à son travail l'une de ces représentations selon ses besoins.

AADL est un langage concret permettant de décrire une architecture à la fois matérielle et logicielle d'un système. Pour ce faire, il définit diverses catégories de composants, classées en trois familles : les composants matériels, les composants logiciels et les composants hybrides. Dans ce qui suit, nous détaillons chacune de ces catégories ainsi que les différents composants qu'ils contiennent.

4.2.1 Catégories de composants

Pour décrire toute une architecture d'un système englobant les entités matérielles et logicielles, AADL distingue trois types des composants qui sont les composants matériels, logiciels et hybrides. Dans ce qui suit, nous détaillons chacune des catégories en indiquant les différents composants qui y appartiennent.

Composants matériels

L'architecture matérielle d'un système modélise les éléments de la plate-forme d'exécution pouvant contenir quatre types de composants :

— *Processor* : Ce composant décrit les processeurs pour dans le but de spécifier un système d'exploitation. Les processeurs représentent des ensembles constitués d'un microprocesseur accompagné d'un ordonnanceur pour la gestion des fils d'exécution qu'il exécute. Ce composant peut contenir uniquement des mémoires et accéder à un bus via son interface.

— *Virtual Processor* : Ce composant représente une ressource logique qui est capable d'ordonnancer et d'exécuter des *threads* ou d'autres processeurs virtuels qui lui sont liés. Il peut être déclaré comme un sous-composant d'un composant processeur ou d'un autre composant processeur virtuel.

— *Memory* : Dans le but de modéliser les mémoires accessibles par les processus légers (threads) en cours d'exécution, ce composant représente tous les dispositifs de stockage (mémoire vive, disque dur, etc). Il permet de stocker des données ainsi que des programmes d'une application.

— *Device* : Ce composant permet de décrire les périphériques modélisant une large variété de matériel (les actionneurs, les capteurs, les appareils, etc) représentent ainsi l'environnement extérieur. AADL ne permet pas de détailler la structure interne d'un *device* mais plutôt il le considère comme une boite noire. Seulement l'interface et les caractéristiques externes du périphérique sont visibles par les autres composants.

— *Bus* : Afin de transporter les informations entre les processeurs, les mémoires et les autres périphériques *device*, le composant *bus* permet de faire la liaison entre eux.

— *Virtual Bus* : Ce composant représente une abstraction de bus logique tel qu'un canal virtuel ou un protocole de communication. Ce type de bus peut être lié à des bus, des bus virtuels, des processeurs, des processeurs virtuels, des devices ou des mémoires.

Composants logiciels

Cette catégorie de composants AADL est dédiée à la définition des éléments applicatifs de l'architecture d'un système formé par des entités logicielles. Ils sont au nombre de cinq :

— *Thread* : les processus légers (tâches) sont modélisés grâce aux threads. Ces derniers modélisent les fils d'exécution qui constituent la partie active de l'application.

— *Process* : Destiné à la modélisation des processus lourds de l'application, ce composant représente un espace d'adressage mémoire dans lequel s'exécute un ensemble de threads. Il peut contenir aussi des données.

— *Thread group* : Afin d'éviter la duplication de code dans le cas où de nombreux threads d'un système partageant un nombre de propriétés, AADL a introduit la notion de groupes de processus légers (*thread group*).

— *Subprogram* : Ce composant est utilisé pour modéliser les sous-programmes représentant un fragment de code séquentiel exécutable avec ou sans paramètres.

— *Subprogram group* : Ce composant représente une bibliothèque de sous-programmes. Il est accessible pour d'autres composants grâce à ses interfaces d'accès (*subprogram group access)*. Les composants peuvent déclarer qu'ils ont besoin d'accéder à des sous-programmes ou à des groupes de sous-programmes. Les composants peuvent donner accès à leurs sous-programmes ou groupes de sous-programmes.

— *Data* : Ce composant décrit soit des types de données lorsqu'elles sont déclarées sous la forme de composants soit des variables partagées lorsqu'elles sont instanciées sous la forme de sous-composants.

Composants hybrides

Cette catégorie de composants rassemble différents composants en entités logiques pour former des blocs logiques d'entités afin de décrire la structure de l'architecture globale. Un composant hybride est modélisé à l'aide du composant *system*. Contrairement aux autres catégories, cette catégorie de composants ne décrit pas une entité concrète. Elle consiste au composant composite contenant un ensemble d'entités logicielles et matérielles.

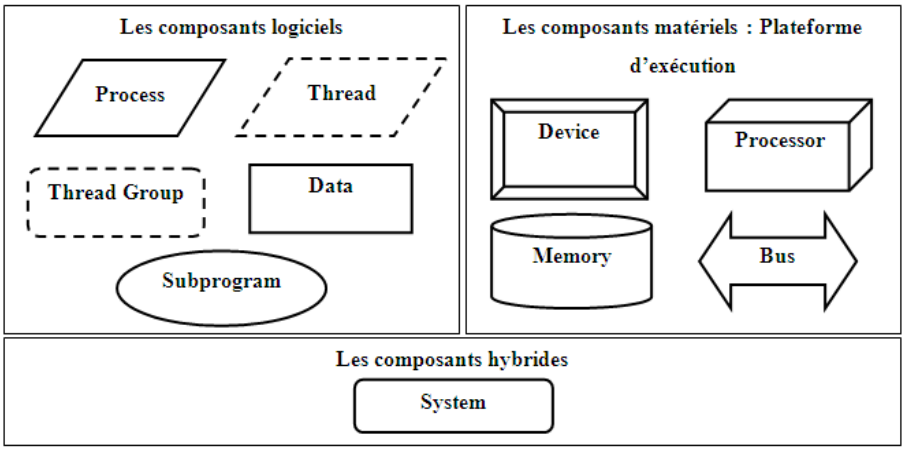

FIGURE 4.1 – Représentation graphique des composants AADL

La figure 4.1 schématise les différentes catégories des composants AADL graphiquement représentés.

Outre ces catégories, la catégorie abstraite de composants représente des composants abstrait utilisés pour décrire des composants du modèle. Un composant *abstract* peut contenir ou être contenu dans n'importe quel type de composant pour être par la suite raffiné vers une catégorie concrète de composants : n'importe quel composant logiciel, matériel ou hybride.

4.2.2 Sous composants et appels

La racine d'une description architecturale en AADL est le composant système. En effet, une description architecturale est structurée comme une arborescence d'instances de composants qui interagissent. La plupart de ses composants peut contenir à son tour un sous-composant qui représente une instance de la déclaration d'un composant. Toutefois, la logique de l'architecture de l'application interdit certains composants de contenir des sous composants. Le tableau 4.1, extrait du standard AADL 2.0 [SAE12] détaille les règles de contenance des sous-composants AADL.

Pour certains composants logiciels, il est interdit de contenir des sous-composants mais il est possible de les appeler. L'implantation d'un processus léger ou d'un sous-programme peut contenir des séquences d'appels à d'autres sous-programmes décrivant ainsi un flot

TABLE 4.1 – Règles de contenance des sous-composants dans AADL 2.0 [SAE12]

Composant	**Peut contenir**
system	data, subprogram, subprogram group, process, virtual processor, processor, memory, bus, virtual bus, device, system et abstract
process	data, subprogram, subprogram group, thread, thread group et abstract
thread	data, subprogram, subprogram group et abstract
thread group	data, subprogram, subprogram group, thread, thread group et abstract
subprogram	data et abstract
subprogram group	subprogram et abstract
data	data, subprogram et abstract
processor	memory, bus, virtual processor, virtual bus et abstract
virtual processor	virtual processor, virtual bus et abstract
memory	memory, bus, et abstract
bus	virtual bus et abstract
virtual bus	virtual bus et abstract
device	bus, virtual bus et abstract
abstract	data, subprogram, subprogram group, thread, thread group, process, processor, virtual processor, memory, bus, virtual bus, device, system, abstract

d'exécution. Seules ces catégories de composants peuvent contenir de tels appels [Zal08]. Le listing 4.1 illustre un exemple d'appel d'un sous-programme. Le programme `Hello_Spg` est appelé par le thread `Task.impl`.

Listing 4.1 – Appel à un sous programme

```
1  subprogram Hello_Spg
2  end Hello_Spg;
3
4  thread Task
5  end Task;
6
7  thread implementation Task.impl
8  calls    {
9           P_Spg : subprogram Hello_Spg;
10          };
11 end Task.impl;
```

4.2.3 Interfaces et connexions

Les composants AADL sont reliés les uns aux autres par le biais de connexions. Ces connexions sont établies entre les interfaces de ces composants. Plus précisément, les sous-composants d'un composant peuvent communiquer les uns avec les autres ou avec leur composant parent en connectant leurs éléments d'interface respectifs. Ces derniers permettent de décrire les flots des flots de contrôles et de données entre composant. Il existe trois types

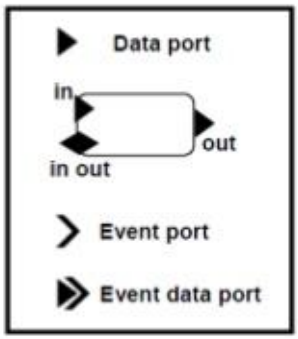

FIGURE 4.2 – Différents types des ports AADL

de ports (voir figure 4.2) : les ports de type donnée (`data port`), de type événement (`event port`) ou de type événement donnée (`event data port`). Le premier type sert avec les paramètres des sous programmes à l'échange des données. Quant au deuxième type, il permet l'échange des signaux. Le troisième type de ports sert à échanger des événements associés à des données. Une connexion est la liaison entre deux sous-composants ou entre un sous-composant et son composant parent via ces ports.

Listing 4.2 – Connexion entre un process et son sous-composant par le biais des ports

```
1  thread Q.Impl
2  features
3          Data_Sink: in event data port;
4  end Q.Ipml;
5
6  process implementation B.Impl
7  subcomponents
8          Ping_Me : thread Q.Impl;
9  connections
10         event data port In_Port -> Ping_Me.Data_Source;
11 end B.Impl;
```

4.2.4 Propriétés et annexes

Une description architecturale en AADL peut être enrichie par des annexes et des propriétés permettant toutes les deux de l'étendre avec des caractéristiques non architecturales. Dans ce qui suit, nous détaillons chacune de ces extensions possibles.

Propriétés

Les propriétés se sont des attributs s'appliquant aux entités constituant l'architecture. Elles servent à associer des caractéristiques ou des contraintes à un composant AADL donné. Ces propriétés sont soit standards prédéfinies par le langage AADL soit personnalisées définies spécifiquement à une application donnée. Nous citons à titre d'exemple, le composant

thread pouvant être enrichi par une propriété décrivant sa nature (sporadique, périodique ou hybride) ou la propriété décrivant son pire temps d'exécution (*WCET*).

Le listing 4.3 illustre l'enrichissement d'un sous-programme Hello_Spg_1 par un ensemble de propriétés. Notons que son implantation s'appelle user_Hello_Spg_1 et qu'elle en Ada.

Listing 4.3 – Propriétés d'un sous-programme

```
1  subprogram Hello_Spg_1
2
3  properties
4          source_language => Ada95;
5          source_name     => "user_Hello_Spg_1";
6  end Hello_Spg_1;
```

Annexes

Outre les propriétés, les annexes servent aussi à enrichir un modèle AADL. Ces annexes permettent d'étendre tel modèle avec des déclarations non architecturales exprimées avec un autre langage que AADL comme l'annexe comportementale (*Behavioral Annex*) ou le langage de requête (OCL : *Object Constraint Language*) exprimant des contraintes.

Listing 4.4 – Exemple d'utilisation de l'annexe OCL dans un modèle AADL [SAE04]

```
1  data Sample
2  end Sample ;
3  thread Collect_Samples
4  features
5  Input_Sample : in data port Sample ;
6  Output_Average : out data port Sample ;
7  annex OCL {**
8              pre : 0 < Input_Sample < maxValue ;
9              post : 0 < Output_Sample < maxValue ;
10            **} ;
11 end Collect_Samples ;
```

Le listing 4.4 montre un *thread* enrichi par l'annexe OCL dans le but de spécifier une pré-condition sur le port d'entrée du composant `Collect_Samples` et une post-condition sur la valeur de son port en sortie. Cet exemple est issu du standard AADL [SAE04].

4.2.5 Modes et transitions entre modes

Pour supporter la reconfiguration dynamique, AADL définit les concepts de modes et de transitions entre modes. Un mode décrit une configuration qui définit explicitement des composants AADL, les connexions entre eux et les valeurs des propriétés associées. Les modes représentent des états opérationnels alternatifs d'un système ou d'un composant. Plus particulièrement, un mode peut spécifier différentes séquences d'appels à utiliser dans

un thread ou un sous-programme. Il peut également représenter les différents états logiques d'un composant pour lequel la valeur des différentes propriétés sont fixées selon le mode actif. Par exemple, sous différents modes, un thread peut avoir des temps d'exécution pour représenter un algorithme qui peut s'exécuter avec des niveaux de précision distincts. Les modes peuvent également représenter différentes configurations matérielles comme les processeurs qui sont actives à un moment donné.

Les différents modes sont représentés comme des états au sein d'une machine à états. Chaque configuration distincte d'un composant est identifiée comme étant un mode (état) dans la machine abstraite du composant. Le passage d'un mode à un autre, c-à-d d'une configuration à une autre, est spécifié à travers une transition qui peut être déclenchée par des événements. La configuration qui définit chaque mode et les événements qui provoquent les transitions dans le comportement du composant doit être spécifié. Chaque machine d'état modal doit avoir au moins deux modes, dont un doit être déclaré en tant que mode initial pour le composant. A travers ces deux concepts, AADL est capable de gérer la reconfiguration dynamique d'une manière claire et explicite sous réserve de considérer uniquement des états prévus à l'avance de l'exécution.

4.2.6 Synthèse

AADL est un langage considéré de première classe dans les domaines temps-réel embarqué critique comme la télécommunication, l'avionique et la médecine. Ce langage est très connu et puissant grâce à sa capacité de décrire l'architecture logicielle et matérielle et de les enrichir par des propriétés et aussi des annexes. Il est facilement extensible pour décrire de nouvelles contraintes ou même des comportements spécifiques à travers d'autres langages. En outre, vis-à-vis le dynamisme, ce langage offre également un support de la reconfiguration dynamique à travers les concepts de modes et de transitions entre modes définissant ainsi des machine à états. C'est la raison pour laquelle nous avons adopté le langage AADL et ses annexes pour la modélisation non seulement des préoccupations fonctionnelles mais aussi celles transversales. Pour modéliser les aspects de la tolérance aux pannes tels que la description, la détection, la propagation et la réparation des erreurs nous utilisons l'annexe d'erreur du langage AADL. Cette annexe offre un support solide de la tolérance aux pannes et plus généralement de la sûreté de fonctionnement. Quant à la réplication, il s'agit d'une redondance manuelle et explicite des composants, des connexions et des comportements. Dans le cas d'un grand nombre de répliques ou de composants à répliquer, la réplication manuelle peut engendrer la complexité de la modélisation des composants et des connexions, la perte de temps de conception et le risque d'erreurs. Pour cela, nous avons défini notre propre solution pour la gestion automatique de réplication en se basant sur les propriétés enrichissant un modèle AADL.

4.3 Modélisation des systèmes temps-réel tolérants aux pannes

Suivant le processus défini, le concepteur commence par modéliser les préoccupations métiers de son système en considérant le langage AADL de base. Aussi, il se charge de l'enrichir selon les exigences du système par des propriétés prédéfinies ou personnalisées ou des annexes appropriées. Dans notre contexte, pour modéliser le comportement des composants ou le consensus nous utilisons l'annexe comportementale et pour modéliser la détection, la propagation et la réparation des erreurs nous utilisons l'annexe d'erreur. Cette annexe offre un support solide de la tolérance aux pannes et plus généralement de la sûreté de fonctionnement. Quant à la réplication, il s'agit d'une redondance manuelle et explicite des composants, des connexions et des comportements. Dans le cas d'un grand nombre de répliques ou de composants à répliquer, la réplication manuelle peut engendrer la complexité de la modélisation des composants et des connexions, la perte de temps de conception et le risque d'erreurs. Pour cela, nous avons défini notre propre solution pour la gestion automatique de réplication. Dans cette section, nous commençons par détailler l'annexe comportementale, l'annexe d'erreurs puis nous décrivons brièvement l'approche de réplication proposée.

4.3.1 Annexe comportementale (Behavioral Annex : BA)

Cette annexe comportementale [SAE06] étend le langage AADL pour raffiner les aspects comportementaux exprimés par le biais d'un automate à états-transitions. Le but de cette annexe est de compléter le langage AADL par d'autres définitions sémantiques et syntaxiques pour exprimer le comportement interne des composants. Il s'agit de spécifier le comportement logiciel du système pour enrichir notamment la description des composants threads et sous-programmes. Il est question d'en fournir principalement des types de données sophistiqués et une façon d'exprimer les parties exécutables des tâches (thread) et des sous-programmes d'une manière portable en utilisant le langage *Behavior_Specification* sans avoir besoin d'un autre langage impératif. La sémantique de cette annexe dépend étroitement du modèle d'exécution AADL et elle est décrite comme suit :

— Lorsque l'exécution d'un composant est lancée, on parle de l'état *initial*.

— Un thread termine son exécution s'il atteint un état *complet* et exécute la partie action de la transition. Les exécutions suivantes vont démarrer à partir de cet état.

— Un état peut être déclaré comme *composite*. Un sous-automate est attaché à un tel état. Une transition peut quitter un sous-état et admet comme cible un descendant de n'importe quel ancêtre de l'état composite auquel il appartient.

— L'annexe accède au contenu des ports d'entrée au moment de l'exécution de la transition. Les données reçues après ne sont pas accessibles.

— Les données écrites sur les ports de sortie de données (*Out data port*) sont transmises après l'accomplissement de l'exécution.

— Les événements (*event*) et les données-événements (*data event*) sont explicitement envoyés par des actions spécifiques.

— L'asynchronisme est décrit par des événements, des accès aux données partagées via des interfaces AADL de types *data access* et les appels distants de procédures.

— Les aspects temps-réel sont soulignés au moyen de trois primitives :

  — *delay (min, max)* permet de spécifier une suspension durant un intervalle de temps non déterministe.

  — *computation (min, max)* résume un calcul par sa consommation non-déterministe de CPU.

  — Un délai d'attente (*timeout*) peut être fixé par une condition.

Le listing 4.5 donne un exemple d'utilisation de l'annexe comportementale pour spécifier le comportement d'un sous-programme dont la déclaration est appelée *addition* et l'implantation est appelée *addition.default*.

Listing 4.5 – Exemple d'utilisation de l'annexe comportementale dans un modèle AADL

```
1  subprogram addition
2  features
3          x   : in parameter Behavior::integer;
4          y   : in parameter Behavior::integer;
5          r   : out parameter Behavior::integer;
6          ovf : out parameter Behavior::boolean;
7  end addition;
8
9  subprogram implementation addition.default
10 annex behavior_specification {**
11         states
12                 s0 : initial state;
13                 s1 : return state;
14         transitions
15                 normal : s0 -[]-> s1 {r := x+y ; ovf:= false;};
16                 overflow : s0 -[]-> s1 {r := 0; ovf:= true;};
17 **};
18 end addition.default;
```

Cette annexe permet de décrire avec détails le comportement des composants AADL. En particulier, nous mettons l'accent sur les composants `device` et `thread` au niveau desquels nous pouvons intégrer des clauses de cette annexe pour décrire leur comportement dans le but de reconfiguration ou de rétablissement en cas de détection d'erreur. Pour les sous-programmes, nous faisons aussi appel à cette annexe pour spécifier dans certains cas le comportement d'un voteur.

4.3.2 Annexe d'erreurs du langage AADL

Malgré la précision, la richesse et la rigueur de AADL, ce langage ne permet pas la description des éléments de la sûreté de fonctionnement dans le but de vérification, d'analyse ou de génération de code. L'annexe des modèles d'erreur *EMA* a vu le jour pour remédier aux limites de AADL et d'automatiser les méthodes d'analyse de sûreté de fonctionnement. Cette annexe est un standard visant à étendre le langage AADL par la définition des modèles d'erreurs au niveau architectural dans le but d'analyser quantitativement et qualitativement la sûreté de fonctionnement des systèmes en se basant sur des bibliothèques [SAE15]. Cette annexe permet également d'assurer la conformité de la conception du système et sa mise en œuvre vis à vis des stratégies d'atténuation des fautes déclarées au niveau modèle architectural.

Dans cette section, nous détaillons les concepts de base de cette annexe puis nous nous intéressons à ces différents niveaux d'abstraction.

Concepts de base

Se référant aux définitions de la FMECA [8], l'annexe des modèles d'erreur traite les concepts suivants :

— Les propagations d'erreurs entre les composants du système et l'environnement.

— Les erreurs, les modes de défaillance et les comportements lors d'une panne pour les composants.

— Les comportements du système ou de ses composants lors de l'occurence d'une panne.

L'annexe d'erreurs permet ainsi de faciliter la modélisation des comportements des erreurs qui peuvent être réutilisées dans plusieurs activités de conception, d'analyse ou de vérification. Contrairement aux composants AADL qui peuvent être déclarés en utilisant la notation graphique ou textuelle, les modèles d'erreurs sont décrits uniquement textuellement sous deux formes :

1. **Bibliothèques de modèles d'erreurs** : Ces bibliothèques contiennent des déclarations réutilisables par référencement dans des sous-clauses des modèles d'erreur. Il s'agit d'un ensemble de types d'erreur et des comportements associés spécifiés sous forme de machines d'état. L'annexe d'erreurs déclare une bibliothèque d'erreurs prédéfinies organisées sous une forme hiérarchique. L'ensemble des types prédéfinies couvre une ontologie de domaine d'erreurs commune qui peuvent survenir dans divers domaines à savoir les erreurs d'omission, les erreurs de valeurs et les erreurs liées aux temps de livraison. Toutefois, cette ontologie peut être personnalisée ou étendue. Les utilisateurs peuvent soit introduire leurs propres types d'erreur spécifiques aux composants ou aux domaines soit adapter les types pré-déclarés en définissant des alias qui

8. *Failure Modes Effects and Criticality Analysis* : la combinaison de FMEA et l'Analyse de Criticité (*CA*)

leurs sont plus significatifs. Un type d'erreur prédéfini ou personnalisé peut avoir plusieurs sous-types. Les types d'erreurs qui font partie du même niveau hiérarchique de types ne peuvent pas survenir simultanément pour un même composant AADL. A titre d'exemple, la fourniture d'un service ne peut pas être précoce et tardive en même temps. Dans ce contexte, l'annexe d'erreurs définit un type *ServiceTimingError* avec deux sous-types qui sont *EarlyService* et *DelayedService*.
Ces bibliothèques permettent également d'introduire des correspondances (*mappings*) ou des transformations qui peuvent être appliquées sur l'ensemble de types d'erreurs définies dans le but de les réutiliser ultérieurement au niveau des sous-clauses du modèle. Les correspondances permettent la spécification des ensembles réutilisables de règles de correspondance entre les types sources et cibles. Les règles de correspondances expliquent comment un type d'erreur peut être projeté sur une instance d'un autre type notamment dans le cadre des chemins de flux d'erreurs.

Listing 4.6 – Extrait de la grammaire décrivant les bibliothèques de modèles d'erreur

```
1 error_model_library ::= annex EMV2 ( ({** error_model_library_constructs **}) | none )
      ;
2 error_model_library_constructs ::=  [ error_type_library ]
3                                     { error_behavior_state_machine }
4                                     { error_type_mappings } *
5                                     { error_type_transformations } *
6 error_model_library_reference ::= package_or_package_alias_identifier
```

Les bibliothèques de modèles d'erreur sont déclarées dans des paquetages. Elles sont réutilisables puisque les concepteurs peuvent les référencer dans les sous-clauses de l'annexe en les qualifiant avec le nom du paquetage correspondant.

2. **Sous-clauses du modèle d'erreurs** : Se sont des clauses de l'annexe relatives à un composant type ou implantation. Comme le détaille le listing 4.7, il s'agit de modéliser les propagations d'erreur au sein du composant, les comportements du composant en présence d'erreur, les comportements d'erreurs composites et les comportements des connexions. Il s'agit aussi de décrire certaines propriétés relatives au composant en question. Dans ce qui suit, nous détaillons brièvement chacune de ces notions.

Listing 4.7 – Extrait de la grammaire décrivant les sous-clauses du modèle d'erreurs

```
1 error_model_subclause ::=
2 annex EMV2 ( ( {** error_model_component_constructs **} ) | none ) [ in_modes ] ;
3 error_model_component_constructs ::=
4 [ use types error_type_library_list ; ]
5 [ use type equivalence error_type_mappings_reference ; ]
6 [ use mappings error_type_mappings_reference ; ]
7 [ use behavior error_behavior_state_machine_reference ; ]
8 [ error_propagations ]
9 [ component_error_behavior ]
```

```
10  [ composite_error_behavior ]
11  [ connection_error_behavior ]
12  [ propagation_paths ]
13  [ EMV2_properties_section ]
```

— *Propagations d'erreurs* : cette section permet de décrire les flux et les propagations d'erreurs. Pour chaque composant, on spécifie, d'une part, les types d'erreurs propagées à travers ses interfaces (*features*) et ses liaisons (*bindings*) à travers la clause *Error propagation* et, d'autre part, les types d'erreurs qui peuvent survenir au sein du composant par le moyen de la clause *error containment*. En outre, on peut décrire le flux de propagation de l'erreur (*error flow*). Ainsi, le composant peut être une source de l'erreur, un puits de l'erreur ou un point par lequel passe le chemin d'erreur (*error path*).

— *Comportement d'un composant* : Le composant AADL est considéré comme étant une boite noire dont on ne connaît pas sa structure interne au niveau de l'annexe d'erreurs. Cette section décrit les occurrences possibles d'erreurs et les modes de défaillances résultants. Les spécifications du comportement du composant en présence d'erreur sont décrites à l'aide d'une machine à état. De plus, elles spécifient les conditions qui déclenchent les transitions d'un état à un autre suite à un évènement d'erreur (*error event*), de recouvrement (*recover event*) ou de réparation (*repair event*), ou par un flux entrant de propagation de l'erreur et les conditions de propagation de l'erreur en sortie.

  Le comportement du composant détaille aussi le processus de détection d'erreur basé sur l'évaluation logique des conditions. Toutefois, ils existent des types d'erreurs qui sont non détectables de cette manière. Notamment, afin d'être détectées, il est indispensable de faire appel aux mécanismes de réplication pour les composants concernés.

— *Comportement d'une connexion* : Les connexions assurent des liaisons physiques et logiques entre les composants permettant le transfert de données d'une source bien déterminée. Un comportement d'erreur d'un composant matériel durant le transfert peut entraîner la propagation de l'erreur dans la connexion jusqu'à la cible de cette connexion. Cette section permet de détailler ce comportement et les conditions de son occurrence, ainsi que, les transformations des types d'erreurs durant le transfert.

— *Comportement d'erreur composite* : Le comportement d'erreur d'un composant est présenté en termes des états d'erreurs de ses sous-composants et de leurs interactions et en termes du flux de propagation d'erreurs entrant.

— *Chemins de propagation* : Ces chemins permettent de décrire les flux de propagation entre les composants en suivant les connexions et les liaisons établies pour les composants AADL modélisés. Le concepteur du modèle d'erreurs peut étendre ces liaisons par d'autres appelées des points de propagation (*propagation points*) ou des chemins de propagations (*propagation paths*).

— *Propriétés* : L'annexe d'erreurs offre un ensemble de propriétés pré-déclarées définies pour être référencées par les modèles d'erreur de la même manière que les propriétés AADL. Ces propriétés servent à décrire profondément les impactes d'erreurs en vue d'analyses de propagations d'erreurs. Certains travaux ont été basés sur ces propriétés pour la génération des arbres de fautes afin d'évaluer des mesures de la sûreté de fonctionnement.

Niveaux d'abstraction

L'EMA met l'accent sur la modélisation des architectures d'erreur en trois niveaux d'abstraction [SAE15] :

1. **Propagation et flux de propagation d'erreurs :** Pour chaque composant, le concepteur indique des types spécifiques d'erreurs qui peuvent être propagées depuis ou vers ses interfaces, ses liaisons ou ses connexions. Le flux passant du point source de propagation (*incoming*) vers le point cible de propagation (*outgoing*) peut être de même désigné par le concepteur pour spécifier s'il s'agit d'une source d'erreur (*error source*), d'une destination (*error sink*) ou d'un chemin d'erreur (*error path*).
   La figure 4.3 illustre la modélisation de la propagation et les flux de propagation d'erreur au sein d'un composant *Process*.

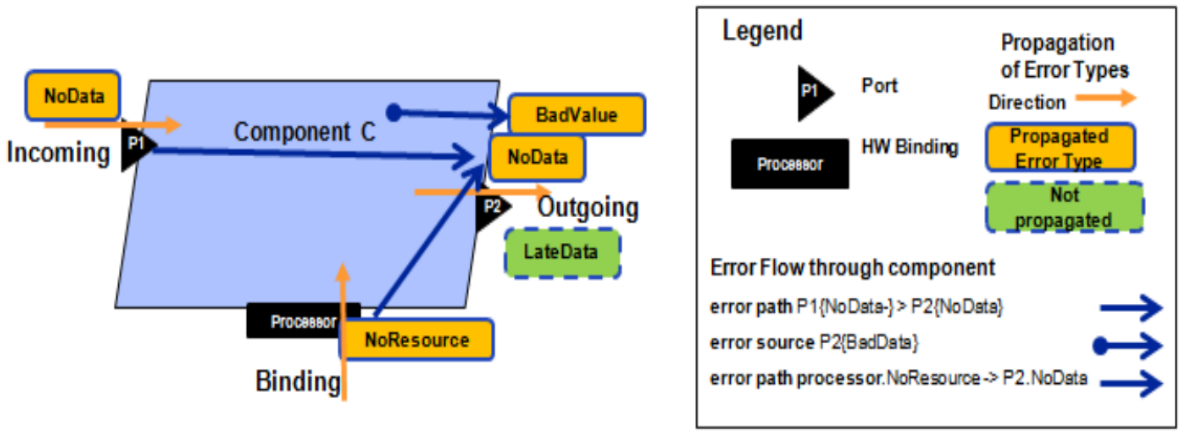

FIGURE 4.3 – Propagation d'erreur et flux de propagation d'erreur [SAE15]

2. **Comportement d'erreurs d'un composant :** Pour chaque composant AADL (type ou implantation), le concepteur peut lui associer une machine d'état décrivant le comportement d'erreur. Il s'agit d'un ensemble d'états et de transitions qui surviennent dans des conditions particulières. Les transitions d'un comportement d'erreur vers un autre sont déclenchées par une erreur détectée sur une source de propagation, des événements de détection d'erreur, des événements de recouvrement (*recover event* ou de réparation (*repair event*) à différents points de propagation d'erreur.
   Les événements d'erreur (*error event*) et les propagations d'erreur sont sémantiquement différents des événements architecturaux déclenchés au niveau architectural du système. Ils ne représentent pas la communication événementielle à travers les ports. Par défaut, ils ne se manifestent pas comme des événements architecturaux qui peuvent causer des transitions de mode ou l'exécution des threads.

Listing 4.8 – Extrait de la grammaire décrivant la machine à états du comportement d'erreur

```
error_behavior_state_machine ::=
error behavior defining_state_machine_identifier
[ use types error_type_library_list ; ]
[ use transformations error_type_transformation_set_reference ; ]
[ events { error_behavior_event } + ]
[ states { error_behavior_state } + ]
[ transitions { error_behavior_transition } + ]
[ properties { error_behavior_state_machine_emv2_contained_property_association } +
]
end behavior ;
```

3. **Comportement d'erreurs composites :** Il s'agit de faire la correspondance entre la spécification abstraite du comportement d'erreur composite du système de base et la spécification du comportement des composants de ses sous-systèmes. C'est le mapping entre le comportement d'erreur d'un composant et le comportement d'erreur du système qui l'englobe.

Listing 4.9 – Extrait de la grammaire décrivant les états composites

```
composite_state_expression ::=
composite_state_element
| (composite_state_expression )
| composite_state_expression and composite_state_expression
| composite_state_expression or composite_state_expression
| numeric_literal ormore
(composite_state_element { , composite_state_element } + )
| numeric_literal orless
(composite_state_element { , composite_state_element } + )
```

Toutes les notions discutées dans cette section, partant de la description des types de fautes, passant par la propagation des fautes entre les composants, et arrivant jusqu'à la

détection et le recouvrement des pannes détectées, prouvent sans doute la puissance et l'efficacité de cette annexe touchant la sûreté de fonctionnement et notamment la tolérance aux pannes. Pourtant, cette annexe ne prend pas en considération la réplication automatique des composants AADL. Ceci peut causer des risques d'erreur et la perte de temps de conception qui reviennent principalement à une redondance manuelle d'un nombre très important de composants et de connexions. Pour faire face à ce problème, nous avons proposé notre propre approche de réplication automatique décrite dans la section suivante.

4.3.3 Annexe d'aspect AO4AADL

AO4AADL [LKZJ13] est une annexe orientée aspect du langage AADL permettant la spécification des aspects architecturaux afin de modéliser les préoccupations transversales au niveau architectural. Trois types d'aspects AO4AADL peuvent être modélisées.

1. Aspect comportemental : C'est un aspect qui décrit le comportement d'une propriété non fonctionnelle transversale à exécuter si un point de jonction au niveau du code fonctionnel est atteint.
2. Aspect de précédence : Il permet de spécifier l'ordre d'exécution souhaité des aspects comportementaux. L'intérêt de cet aspect réside lorsque deux aspects différents sont associés pour le même point de jonction.
3. Aspect de composants affectés : C'est un aspect pour définir la liste des composants AADL affectés lors de l'exécution d'un aspect comportemental.

Ainsi, AO4AADL permet d'intercepter l'appel ou l'exécution des sous programmes, les ports (in, out et inout) et les types et les valeurs de données envoyées/reçues par les ports. Ce langage offre la possibilité d'exécuter différentes actions possibles au niveau de l'advice. Il s'agit non seulement des déclarations, des initialisations et des affectations des variables mais aussi des actions de communications comme l'envoi et la réception de données via les ports et la spécification des contraintes temporelles (*delay, behavior time*).

Listing 4.10 – Extrait de la grammaire du langage annexe AO4AADL

```
1  Action ::= Basic_Action
2           | If_Statement
3           | For_Statement
4           | While_Statement
5  Basic_Action ::= Assignment
6                  | Communication
7                  | Timed_Actions
8                  | Proceed_Action
9  Timed_Actions ::= computation ( Behavior_Time [, Behavior_Time] ) ;
10                 | delay ( Behavior_Time [ , Behavior_Time] ) ;
```

Une spécification d'un aspect AO4AADL comportemental est décrite en deux parties :

— **Spécification Pointcut** : Cette partie identifie les conditions dans lesquelles l'aspect est invoquée par les composants AADL fonctionnels correspondants. Les points de jonction architecturaux sont explicitement exprimés comme les endroits où le code de l'advice peut s'exécuter. Elle consiste à intercepter des sous-programmes AADL ainsi que des données sortantes de (resp. entrantes vers) des composants AADL.

— **Spécification Advice** : Cette partie encapsule le comportement de l'aspect relativement à son emplacement. Le comportement de l'advice exécute un ensemble d'instructions lorsque le point de jonction associé est atteint. Ces instructions incorporent non seulement les déclarations, l'initialisation et l'affectation de variables mais aussi des actions de communications. Ces derniers sont responsables d'envoi (ou de réception) de messages vers des ports de sorties (à travers des ports d'entrées respectivement). De plus, il est possible de spécifier certaines actions temporelles au niveau d'un advice AO4AADL. A titre d'exemple, nous pouvons introduire des restrictions temporelles à prendre en considération lors de l'exécution du code fonctionnel.

4.4 Conclusion

Dans ce chapitre, nous avons détaillé les concepts du langage AADL tout en mettant l'accent sur les aspects utilisés durant notre processus. Ce langage est utilisé dans la suite de ce mémoire non seulement pour la modélisation des architectures des applications temps-réel distribuées mais aussi pour la modélisation des concepts de la tolérance aux pannes en se basant à la fois sur les annexes et les propriétés. Pour configurer ces applications et générer le code correspondant en utilisant les intergiciels dédiés, ce langage offre aussi des constructions spécifiques comme la description matérielle dont on peut se servir pour le déploiement de telles applications.

Ceci achève la première partie de ce mémoire qui constitue l'étude du contexte, de la problématique et des solutions existantes. La partie suivante est consacrée à la description détaillée des contributions de cette thèse liées aux différentes étapes de notre processus de développement des systèmes temps-réel distribués tolérants aux pannes.

Deuxième partie

Mise en Œuvre et Validation

5

DP4FTRTS : un processus de développement des systèmes temps-réel distribués tolérants aux pannes

5.1 Introduction

Dans le chapitre 3, nous avons présenté notre approche générique pour la mise en place des systèmes temps-réel distribués tolérants aux pannes. Cette approche définit un processus, appelé DP4FTRTS, formé de certaines étapes permettant la modélisation, l'analyse, la génération de code et l'implantation de ces systèmes tout en séparant la tolérance aux pannes des préoccupations fonctionnelles durant le cycle de développement. Puis nous avons expliqué les motivations qui nous ont poussées à choisir la langage AADL pour la description architecturale de ces systèmes.

Dans ce chapitre, nous introduisons les grandes lignes de notre travail qui s'articule autour du processus générique DP4FTRTS en considérant le langage AADL comme langage de description d'architecture adopté dans la section 5.2. Dans la section 5.3, nous décrivons notre approche de gestion automatique de réplication basée sur la définition des propriétés AADL. Puis, nous nous focalisons dans la section 5.4 sur le processus de génération séparée du code fonctionnel et tolérant aux pannes grâce à la programmation orientée aspect. Enfin, vue la spécificité des systèmes temps-réel, nous nous intéressons dans la section 5.5 à l'extension et l'adaptation d'un langage d'aspect pour le support de ces systèmes.

5.2 DP4FTRTS : processus de développement raffiné

Dans le chapitre précédent, nous avons présenté une approche générique pour la production des systèmes temps-réel distribués tolérants aux pannes sous forme d'un processus de développement qui renferme différentes étapes allant de la modélisation jusqu'à l'exécution. Dans cette section, nous raffinons ce processus pour donner un nouveau processus concret en termes de langage utilisés pour la modélisation, de techniques d'analyse, des outils et des paradigmes de programmation mis en évidence. La figure 5.1 donne un aperçu global sur le processus raffiné.

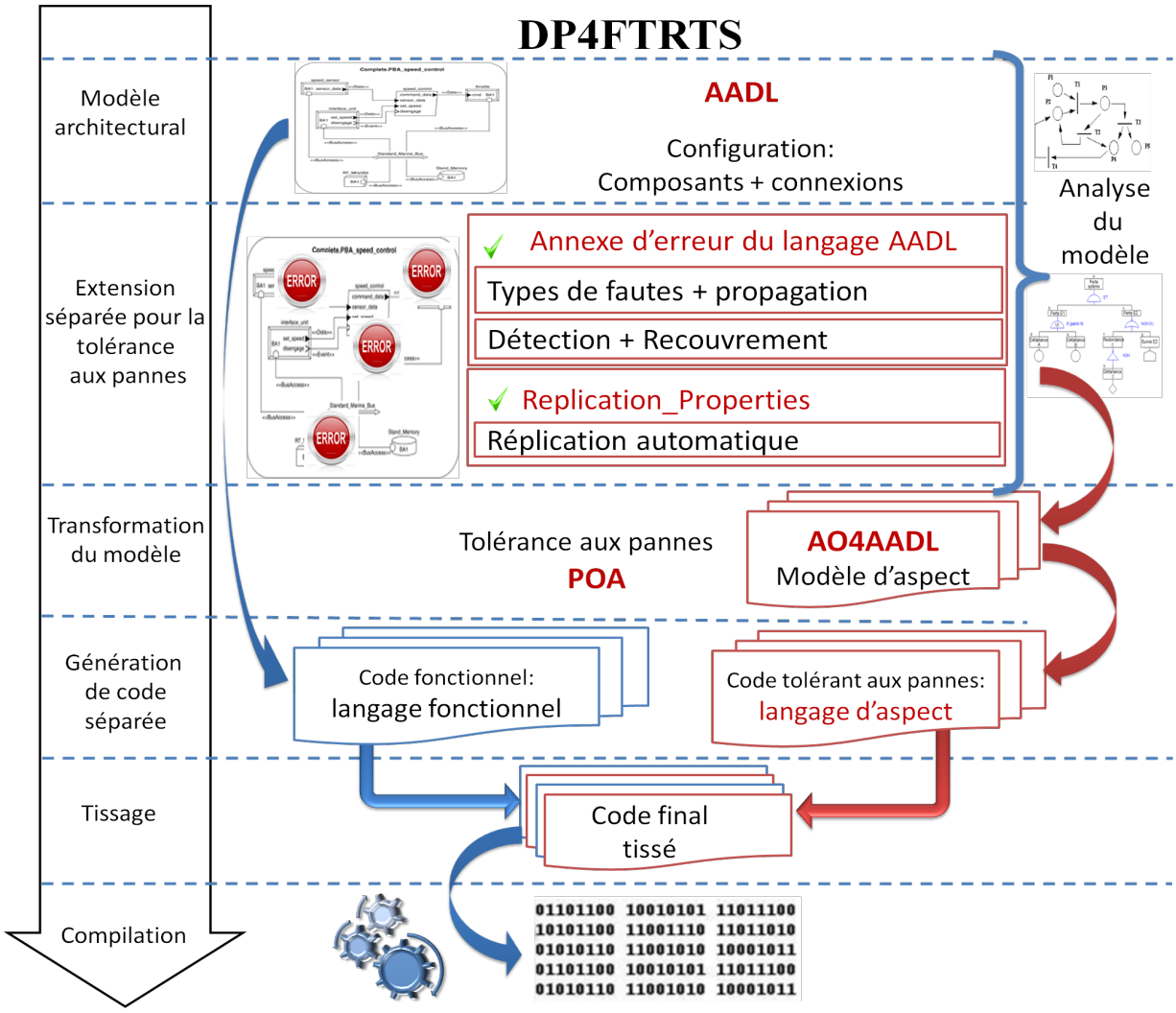

FIGURE 5.1 – Processus de développement raffiné

Comme nous l'avons déjà justifié au niveau du chapitre 3, nous avons choisi le langage AADL pour la modélisation des systèmes considérés puisqu'il s'agit d'un langage qui a fait

preuve de ses capacités dans le domaine temps-réel critique. Malgré la puissance du langage AADL dans la modélisation des systèmes temps-réel embarqués distribués, le noyau de ce langage n'est pas destiné pour la description des concepts de tolérance aux pannes. Toutefois, une annexe d'erreurs EMA a été standardisée pour s'en occuper. L'annexe EMA décrite dans la section suivante est une annexe riche en termes de concepts offerts relatifs à la tolérance aux pannes et plus généralement à la sûreté de fonctionnement. Cette annexe prend en charge la définition des modèles d'erreurs réutilisables par le biais de bibliothèques. Ces modèles d'erreurs sont représentés par des machines d'état décrivant le comportement des composants dans la présence d'erreurs, les propagations d'erreurs entre composants et les évènements de rétablissement. Le concepteur associe des modèles d'erreurs à des composants du système, des composants de la plate-forme d'exécution, ainsi que les connexions entre eux. Lorsqu'un modèle d'erreurs est associé à un composant, il est possible de le personnaliser en définissant des valeurs spécifiques aux composants pour le taux d'arrivée ou de la probabilité d'occurrence d'événements d'erreur et les propagations d'erreurs déclarées. Le concepteur peut également spécifier l'effet de propagations d'erreurs sur les composants en question par le moyen des propriétés prédéfinies donnant ainsi lieu à des analyses de la sûreté de fonctionnement. Ainsi, un modèle d'erreurs décrit avec l'annexe EMA est capable de décrire les erreurs pouvant affecter un des composants du système et les propagations qui peuvent en résulter et affecter d'autres composants. Il peut également décrire les conditions de détection et de rétablissement à base d'évènements et d'actions de reconfiguration. En outre, le concepteur peut effectuer à ce stade des analyses de la sûreté de fonctionnementen se servant des propriétés de cette annexe. Le concepteur peut profiter des travaux d'analyse qui se sont fondés sur cette annexe pour l'évaluation des mesures de la sûreté de fonctionnement à travers le passage par des formalismes intermédiaires. Notons ici les transformations des modèles AADL vers des réseaux de Petri [Rug08] ou des arbres de fautes [JVB07] à des fins d'analyse de la sûreté de fonctionnement.

Toutefois, bien que la réplication est une technique fréquemment utilisée pour la mise en place de systèmes tolérants aux pannes, cette annexe n'offre pas un support pour la gestion automatique de réplication des composants AADL. Le concepteur se charge de gérer la réplication d'une manière manuelle et explicite des composants et des connexions AADL. Cette tâche requiert en effet un effort considérable pour la gestion de la cohérence de modèle répliqué en présence de composants générés et d'autres présents dans le modèle de base. La communication entre ces différents composants prenant en compte ces catégories et l'ensemble de connexions qui les relient devient une tâche fastidieuse dès que le nombre de répliques ou de composants à répliquer devient important. Ce qui rend une telle réplication manuelle peut engendrer une perte de temps de conception et un grand risque d'erreur.

Pour remédier à ce problème, notre **première contribution** consiste à automatiser la réplication de certains composants AADL en étendant le modèle AADL architec-

tural par des propriétés AADL. Nous avons défini un ensemble de propriétés baptisé `Replication_Properties` pour la description des paramètres et des mécanismes de réplication souhaités. Il s'agit de décrire la manière, le but ou le contexte de réplication sous forme de propriété appelée `Description`. Le concepteur fixe aussi le nombre de répliques souhaité à travers une propriété appelée `Replica_Number`. Ainsi, dans un même modèle, nous pouvons supporter un nombre variable de répliques pour différents composants répliqués. Afin de bien identifier les répliques générées, nous avons défini une propriété appelé `Replica_Identifiers`. Pour supporter aussi les deux styles de réplication les plus utilisés (réplication active et passive), nous avons défini une propriété baptisée `Replica_Type` permettant de choisir l'un des deux styles. Enfin, pour décrire les algorithmes de consensus, nous avons défini trois propriétés pour couvrir tous les cas possibles pouvant désigner un sous-programme AADL. L'ensemble de toutes les propriétés doit être spécifié par le concepteur et appliqué à un composant AADL pour donner lieu à une génération automatique d'un modèle AADL v2 enrichi avec les répliques. Par la suite, ce modèle peut être l'objet de certaines analyses comme l'analyse d'ordonnancement ou la vérification de certaines propriétés, de génération de code ou même de mesure des attributs de la sûreté de fonctionnement à travers les arbres de fautes par exemple.

Notre **seconde contribution** est la génération automatique de code à partir du modèle AADL final. Visant la séparation des préoccupations même au niveau applicatif, nous avons adopté la programmation orientée aspect pour la génération du code tolérant aux pannes. Pour la génération du code fonctionnel, il existe des intergiciels comme PolyORB_HI permettant la génération de code à partir des modèles AADL pour des applications temps-réel critiques. Cet intergiciel permet de produire du code en différents langages cibles comme Ada, Java ou C selon le choix de l'utilisateur. Nous utilisons cet intergiciel pour la génération de code fonctionnel par l'intermédiaire de Ocarina, une suite d'outils dédiée pour la manipulation et l'analyse des modèles AADL [VZH06]. En ce qui concerne la génération du code tolérant aux pannes décrit avec l'annexe EMA et supportant la réplication, il s'agit d'appliquer une transformation de modèle pour générer un modèle intermédiaire indépendant de la plate-forme. Puisque nous souhaitons utiliser la programmation orientée aspect, nous avons pensé à générer un modèle AADL riche avec les aspects architecturaux grâce à l'annexe AO4AADL. Ce dernier est une extension d'aspect du modèle AADL permettant d'intercepter des composants ou des connexions. Puis, ce modèle intermédiaire va subir par la suite une transformation M2C (*Model To Code*) pour produire du code aspect. Le langage d'aspect cible doit être conforme avec le langage cible utilisé pour la génération du code fonctionnel. Pour cette raison, le code d'aspect sera généré en AspectAda, AspectJ ou AspectC pour être tissé avec le code fonctionnel généré respectivement en Ada, RTSJ ou C. Ces divers choix peuvent être appliqués avec notre processus de génération de code pour les applications temps-réel tolérants aux pannes. Ainsi, notre approche de génération offre une

degré de flexibilité à l'utilisateur pour les choix des langages fonctionnels et donc d'aspect dépendant fortement du domaine d'application. Pour cela, notre solution consiste à définir des règles génériques de transformation de l'annexe EMA vers le langage intermédiaire AO4AADL à partir duquel nous générons tous les langages d'aspects en se basant sur les même règles. Plus particulièrement, dans le cadre de cette thèse, nous nous focalisons sur le générateur Ada de l'intergiciel PolyORB_HI s'adressant au domaine temps-réel critique. Et pour garantir une conformité entre le code fonctionnel et non fonctionnel, ce dernier doit être généré en AspectAda, extension d'aspect pour le langage Ada.

La **troisième contribution** de cette thèse est fondée sur l'étude et l'adaptation du langage AspectAda [PC05] existant pour le développement temps-réel. L'exploration de la littérature a prouvé l'absence d'utilisation de ce langage dans ce domaine comme nous l'avons remarqué au niveau de la section 2.4. Pour cette raison, nous avons effectué une étude approfondie pour le test et l'évaluation de ce langage existant dans une première étape. Dans une deuxième étape nous avons opté à une adaptation et extension de ce langage pour respecter les contraintes temps-réel.

Toutes les contributions tournent autour le processus de développement proposé pour la modélisation et l'implantation des systèmes temps-réel distribués tolérant aux pannes. Ce processus montre un ensemble d'étapes partiellement ordonnées qui à partir d'un modèle d'architecture AADL et des extensions par des propriétés et des annexes, produit un code généré automatiquement en gardant la séparation des préoccupations sur tous les niveaux. Ce code généré représente le code applicatif du système temps-réel capable d'intercepter les composants et de détecter les erreurs survenues s'ils sont prévues dès la phase de conception. Dans la suite de ce chapitre, nous décrivons brièvement chacune de nos contributions pour en donner plus de détails dans les chapitres qui suivent.

5.3 Gestion de la réplication

Afin de faciliter la tâche du concepteur et de gérer la réplication des composants AADL automatiquement, nous avons défini une approche basée sur les transformations de modèles permettant le passage d'un modèle AADL réduit à un modèle AADL enrichi avec les répliques. Il s'agit d'élever le niveau d'abstraction de notre modèle et d'automatiser la gestion de la réplication. Cette approche tire profit des extensions offertes par le langage AADL. Elle est basée sur l'encapsulation des concepts de réplication, sous forme d'un ensemble de propriétés pouvant être appliquées à un composant AADL, pour le répliquer le nombre de fois que désiré. Ces propriétés permettent à leur tour de générer un autre modèle AADL riche avec les répliques. De cette façon, cette approche permet la diminution et la gestion de la complexité du système et la réduction du temps de conception en se basant sur la

séparation des préoccupations. L'approche proposée renferme trois étapes.

1. La première étape consiste à concevoir le modèle AADL spécifiant l'architecture de base du système. Ce modèle décrit l'hiérarchie des composants interconnectés.
2. Le concepteur dans une seconde étape encapsule les paramètres de la réplication sous forme de propriétés. Pour cela, nous avons défini un ensemble de propriétés que nous avons nommé `Replication_Properties` pour la description des paramètres et des mécanismes de réplication souhaités. À travers ces propriétés, le concepteur devient capable de fixer le nombre de répliques souhaités, les identifiants des répliques qui seront générées, le style de réplication et l'algorithme de consensus adopté pour appliquer le vote entre les répliques dans le cas d'une réplication active. Il est à noter que nous supportons un nombre variable de répliques et les deux styles de réplication les plus fréquents (réplication active et passive) qui peuvent être appliqué au sein d'un même modèle. Nous supportons aussi toutes les formes d'un sous programme AADL (texte, classificateur et référence). Il suffit d'appliquer les propriétés nécessaires à un composant AADL pour en servir lors de la génération automatique du modèle répliqué.
3. La troisième étape consiste à appliquer une transformation de modèles afin de générer un modèle cohérent intégrant les différentes répliques en établissant les connexions et en générant les comportements pour toutes les répliques. La transformation est basée sur l'ensemble de propriétés établies par le concepteur et s'applique en suivant les algorithmes de transformation que nous avons développés. Plusieurs algorithmes ont été mis en place pour considérer les divers types de composants et les deux styles de réplication.

Le modèle généré peut être par la suite objet d'analyse et de vérification ou de génération de code. Nous nous intéressons dans cette thèse à la génération de code automatique par le biais des techniques de transformation de modèles offertes par la démarche MDA.

La définition des propriétés de réplication ainsi que la mise en place des algorithmes de transformations seront détaillées dans le chapitre 6.

5.4 Génération de code

Une fois notre modèle AADL est conçu, considérant les différents concepts de tolérance aux pannes comme la réplication, les types d'erreurs et les techniques de recouvrement, nous aidons le concepteur à générer le code fonctionnel et tolérant aux pannes pour avoir un code applicatif exécutable. Nous avons adopté la POA pour la génération séparée des préoccupations. Pour la génération du code fonctionnel à partir du modèle AADL, nous utilisons la suite d'outils Ocarina, un outil très utilisé pour la manipulation et l'analyse des modèles AADL. Outre l'analyse lexicale, syntaxique et sémantique des modèles AADL, Ocarina permet également la vérification formelle des modèles AADL avec les Réseaux de Petri. Elle

contient un support d'analyse d'ordonnancement avec Cheddar. Aussi, cet outil permet de générer des applications distribuées en Ada, RTSJ ou C en se basant sur les intergiciels Polyorb et Polyorb_High_Integrity. Finalement, Ocarina supporte aussi la POA à travers l'annexe AO4AADL, extension d'aspects architecturaux pour le langage AADL, et permet la génération de son code en AspectJ.

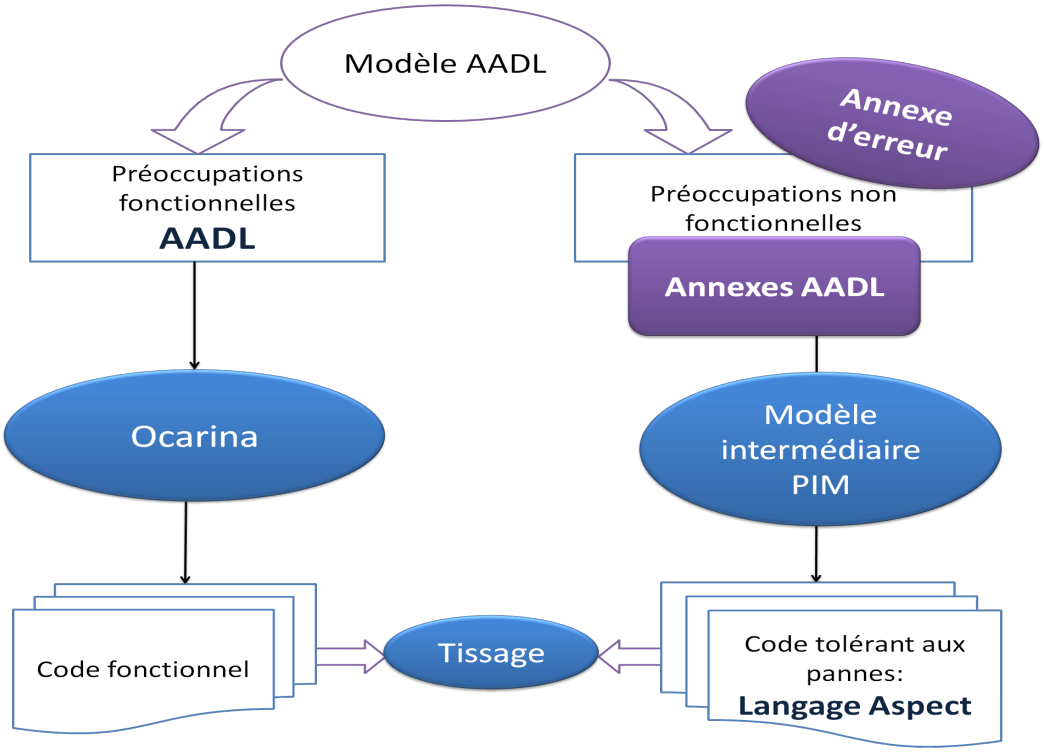

FIGURE 5.2 – Processus de génération de code proposé

Concernant le générateur de code de la suite Ocarina, il est basé sur l'intergiciel PolyORB_HI dédié pour la production du code des systèmes temps-réel réparties embarqués. Cet intergiciel repose à son tour sur des patrons de conception pour les systèmes critiques, comme le profil Ravenscar, pour éviter les comportements dynamiques incompatibles avec les exigences de ce type de systèmes [Zal08]. Pour chaque nœud du modèle AADL, des composants applicatifs et d'autres composants intergiciels sont générés conformément au profil Ravenscar ainsi qu'à un ensemble de restrictions pour les systèmes critiques. Ceci nous a encouragé à utiliser cette suite d'outils pour la génération du code fonctionnel à partir des modèles AADL. Toutefois, pour la génération du code tolérant aux pannes, nous proposons cibler un langage d'aspect pour garantir la séparation entre préoccupations tout au long du cycle de développement. Pour des raisons de conformités, le code aspect doit être généré en AspectAda, AspectJ ou AspectC pour être tissé avec le code fonctionnel généré respectivement en Ada, RTSJ ou C. Sachant que la suite d'outils Ocarina comprend une partie frontale

pour l'analyse de l'annexe AO4AADL et une partie dorsale pour la génération du code AspectJ à partir de l'annexe AO4AADL, nous avons profité de ces deux parties pour mettre en place un processus de génération de code générique. Il s'agit de passer par un modèle intermédiaire décrit en AO4AADL qui sera crée automatiquement à partir des description de l'annexe EMA en suivant des règles de transformation que nous avons défini. Ce modèle indépendant de la plate-forme sera raffiné dans le but de générer le code aspect cible dépendamment du langage fonctionnel choisi.

Étant donné que le langage Ada est considéré le mieux adapté dans le domaine temps-réel critique, nous avons considéré dans ce travail la génération du code fonctionnel en Ada. Pour des raisons de conformité, le code aspect doit être impérativement généré en AspectAda. Dans cette orientation, notre dernière contribution était dans le but d'évaluer, adapter et étendre le langage AspectAda existant pour le support des systèmes temps-réel.

Le processus de génération de code, les outils nécessaires ainsi que les règles de transformation établies pour la génération de l'annexe EMA vers le langage AO4AADL sont détaillés dans le chapitre 7. Le chapitre 8 sera consacré pour l'étude du langage AspectAda existant et la description de l'approche proposée afin de l'étendre et l'adapter pour le respect des contraintes temps-réel.

5.5 Adaptation du langage AspectAda pour le temps-réel

La programmation orientée aspect est basée sur l'opération de tissage qui consiste à insérer automatiquement des bouts de codes dans le code métier d'une application. Échappant à tout contrôle du développeur, ces bouts de code insérés puissent compromettre le déterminisme et violer les contraintes d'un système temps-réel. Dans ce contexte, les langages d'aspects doivent être évalués et adaptés pour être utilisé dans le domaine temps-réel critique. La dernière contribution de cette thèse concerne l'extension et l'adaptation du langage AspectAda pour le support des contraintes temps-réel. Ce langage vise à étendre le langage Ada avec les concepts d'aspects.

Afin de tester le langage AspectAda existant, nous avons opté à l'ingénierie inverse. Nous avons étudié ce langage par rapport à sa syntaxe et sa sémantique, son opération de tissage et de génération de code et son compilateur/tisseur. Nous avons constaté et extrait plusieurs lacunes sur tous les plans. En réalité, ce langage présente des défauts de syntaxe et de sémantique et sa bibliothèque Runtime est réduite. L'architecture de son compilateur était aussi non spécifique et mal structurée et son implantation était encore à un stade précoce. Finalement, l'opération de tissage adoptée, basée sur une allocation dynamique des ressources, viole les contraintes temps-réel.

Tenant compte de ces problèmes, nous avons pensé à choisir une parmi deux solutions. La première consiste à résoudre tous ces problèmes et maintenir le compilateur existant. La seconde consiste à définir un nouveau langage et de lui développer un compilateur propre.

À cause du manque de documentation et du code mal structuré, il était très dur de maintenir le compilateur existant. Pour cela, nous avons choisi la réécriture de la grammaire et le développement d'un nouveau compilateur après avoir étudié les problèmes dégagés profondément un à un. Nous avons proposé des solutions pour enrichir la syntaxe et la sémantique de ce langage pour adapter le langage au développement temps-réel. Quant au compilateur, nous avons défini une nouvelle architecture et nous avons développé le compilateur correspondant. Pour l'implantation du compilateur, nous avons évité toute construction interdite par le profil Ravenscar et d'autres concepts jugés d'après la littérature dangereux pour les systèmes temps-réel. Ceci est dans le but de garantir la compatibilité de ce nouveau langage avec les restrictions exigées pour le développement des systèmes temps-réel critiques.

L'étude du langage existant ainsi que la présentation des solutions proposées sont détaillées au niveau du chapitre 8.

5.6 Conclusion

Dans ce chapitre, nous avons présenté le processus de développement que nous avons défini dans le cadre de notre thèse. Ce processus renferme quatre parties. Nous commençons, d'abord, par décrire l'architecture du système en utilisant le langage AADL. A ce niveau, Nous enrichissons ce modèle par l'annexe EMA afin de mettre en place les modèles d'erreurs visant un modèle tolérant aux pannes. Ensuite, nous utilisons les propriétés pour décrire la réplication souhaitée des composants, et en utilisant notre extension, nous générons un modèle intermédiaire enrichi avec les répliques. Puis, nous produisons le code fonctionnel et celui tolérant aux pannes en se basant sur la programmation orientée aspect. Pour la génération du code aspect, nous utilisons, enfin, notre compilateur AspectAda développé aussi dans le contexte de notre thèse.

Dans les chapitres 6, 7 et 8, nous expliquons et détaillons chacune de ces étapes. Puis nous validons nos contributions dans le chapitre 9.

6

Gestion automatique de la réplication

6.1 Introduction

Dans le chapitre 3, nous avons présenté un processus de développement pour la modélisation et l'implantation séparées de la tolérance aux pannes pour les systèmes temps-réel distribués. En particulier, dans la section 3.6, nous avons justifié le choix du langage AADL pour la modélisation et la configuration de ces systèmes. Pour la modélisation de la tolérance aux pannes, nous avons adopté l'annexe d'erreurs pour la modélisation de la tolérance aux pannes. Cette annexe, décrite dans la section 4.3.2 du chapitre 4, permet la description de plusieurs aspects de la tolérance aux pannes, à savoir les erreurs, la propagation et le rétablissement. Quant à la réplication, plusieurs études ont porté sur la modélisation de la réplication avec AADL. Il s'agit d'une réplication manuelle et explicite des composants, des connexions et des comportements. Dans le cas d'un grand nombre de répliques ou de composants à répliquer, la réplication manuelle peut engendrer la complexité de la modélisation des composants et des connexions, la perte de temps de conception et le risque d'erreurs. Pour cela, nous avons défini notre propre solution pour la gestion automatique de réplication. Nous proposons une extension de la suite d'outils Ocarina pour supporter l'automatisation de la réplication des composants AADL à des fins de tolérance aux pannes. Nous aidons ainsi le concepteur à générer automatiquement un modèle AADL tolérant aux pannes et supportant la réplication.

Dans ce chapitre, nous donnons tout d'abord une vue d'ensemble sur notre approche de réplication dans la section 6.2. Ensuite, nous détaillons les différentes propriétés définies au niveau de la section 6.3. Puis, nous expliquons les règles de transformation de modèle établies dans la section 6.4. Enfin, nous décrivons les étapes d'implantation de notre contribution comme extension de la suite d'outils Ocarina dans la section 6.5.

6.2 Description de l'approche de modélisation de la réplication

Notre approche de transformation de modèles AADL [SAE12] tire profit des extensions offertes par ce langage. Elle permet la diminution et la gestion de la complexité du système et la réduction du temps de conception. Il s'agit d'enrichir un modèle AADL avec des propriétés de réplication.

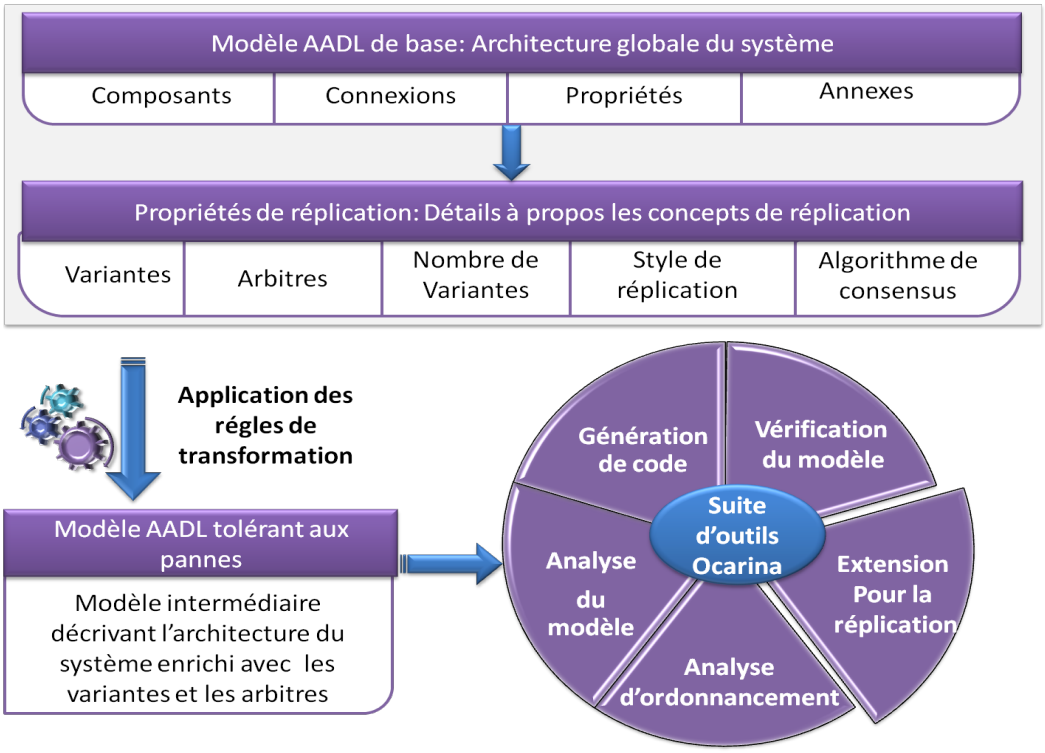

FIGURE 6.1 – Processus de réplication

Cette approche s'articule en trois phases comme l'indique la figure 6.1. Il s'agit de concevoir en première étape le système initial. A ce niveau, le concepteur décrit l'architecture du système en spécifiant les différents composants et les connexions qui les relient. Ce modèle reflète l'architecture globale du système. Il peut être enrichi par des propriétés ou des annexes. En particulier, nous nous intéressons à l'annexe d'erreurs (*Error Model Annex* [SAE15]) et l'annexe comportementale (Behavioral Annex [SAE06]). Ainsi, nous avons un modèle AADL intégrant les politiques de tolérance aux pannes. Pour appliquer la réplication de certains composants AADL, nous proposons une approche basée sur l'extension du modèle par des propriétés dédiées.

Dans une seconde étape, le concepteur enrichie le modèle par les propriétés de réplication que nous avons défini. Il s'agit de donner une description de la réplication souhaitée, fixer le nombre de répliques et la stratégie de réplication adoptée. Nous supportons les deux

styles de réplication active et passive. Le concepteur spécifie aussi les identifiants (string) des répliques générés et enfin spécifie l'algorithme de consensus choisi. Toutes ces propriétés, appliquées à un composant AADL, doivent être présentes et non redondantes. Si l'utilisation des propriétés de réplication est bien vérifiée, le concepteur dans une troisième étape, génère un modèle AADL supportant la réplication des composants en question. Cette génération se fait à la base des propriétés spécifiées et du modèle initial. Elle dépend fortement, d'une part, de la stratégie de réplication choisie et, d'autre part, du type et de l'hiérarchie du composant répliqué. Pour cela, nous avons défini un ensemble de règles et d'algorithmes de transformation du modèle pour régir le processus de génération. Ces règles définissent la façon non seulement de générer les répliques (variantes) mais aussi de générer les composants voteurs (arbitres) et de relier les composants générés aux composants du modèle initial.

Dans la section suivante, nous décrivons chacune des propriétés décrivant le mécanisme de réplication.

6.3 Description de l'ensemble des propriétés de réplication

Dans le but de gérer automatiquement la réplication des composants AADL, nous avons défini un ensemble de propriétés appelé `Replication_Properties` comportant des propriétés décrivant avec détails le mécanisme de réplication adopté. Le concepteur décrit le modèle de réplication souhaité en spécifiant les valeurs adéquates pour chacune de ces propriétés. Puis, à l'aide des règles de transformations, le modèle souhaité sera généré automatiquement. Dans cette section, nous décrivons et expliquons chacune des propriétés que nous avons définies.

6.3.1 Description de la réplication

La propriété `Description` permet de décrire le contexte de réplication. Le concepteur donne plus de détails sur le but, la manière ou les besoins de réplication. Cette propriété est représentée par une chaine de caractères (string) et appliquée à la fois aux composants logiciels (process, thread), matériels (processor, device) et hybride (system).

Listing 6.1 – Propriété Description

```
1 -- description of the context, requirements and goal of the replication
2 Description : aadlstring applies to (system, process, thread, processor, device);
```

Puisque cette propriété donne une information sur le contexte de réplication sans influencer le mécanisme de réplication, elle contribuera à la documentation du modèle et non à la génération du modèle.

6.3.2 Nombre de répliques

La propriété `Replica_Number` fixe le nombre de répliques qu'on souhaite modéliser. Ce nombre est fixé par le concepteur.

Listing 6.2 – Spécification du nombre de répliques

```
1 -- Specification of the replica number
2 Replica_Number : aadlinteger applies to ( system, process, thread, processor, device);
3
4 -- Min_Nbr_Replica: A constant that represents the minimal number of replica.
5 Min_Nbr_Replica : constant aadlinteger => 3 ;
6
7 -- Max_Nbr_Replica: A constant that represents the maximum number of replica.
8 Max_Nbr_Replica : constant aadlinteger => 5 ;
```

D'après la littérature, ce nombre doit être généralement impair supérieur ou égal à trois pour diminuer la chance de désaccord dés la première étape de vote. Ainsi le nombre minimal de répliques est fixé à travers une constante `Min_Nbr_Replica` initialisée à trois. Le nombre maximal de copies est aussi fixé à travers une constante appelée `Max_Nbr_Replica`. Pour ne pas augmenter le coût supplémentaire de la redondance (overhead), nous avons initialisé le nombre maximal de répliques à cinq. Ces deux constantes `Max_Nbr_Replica` et `Min_Nbr_Replica` représentent respectivement le nombre maximal et minimal de répliques. Afin de laisser notre modèle plus flexible, ces deux constantes puissent être paramétrées par le concepteur dans l'ensemble des propriétés `Replication_Properties`. Ainsi le nombre de répliques sera borné par les deux constantes définies par le concepteur lui même.

6.3.3 Identifiants des répliques

La propriété `Replica_Identifiers` est présentée par une liste de string dont chacun représente un identifiant d'une réplique générée.

Listing 6.3 – Description des identifiants des différentes répliques

```
1 -- Identifiers of the different generated replica
2 Replica_Identifiers : list of aadlstring applies to (system, process, thread, processor,
      device);
```

La taille de la liste doit être exactement égale au nombre `Replica_Number`. Lors de la génération du nouveau modèle, chacune des répliques admet comme identifiant l'un des éléments de cette liste.

6.3.4 Type de réplication

La propriété `Replica_Type` consiste à définir le type de réplication. Il s'agit d'une réplication active ou passive. Le concepteur choisit un des deux types selon les besoins du système.

Listing 6.4 – Description du type de réplication

```
1 --Replication type
2 -- Passive Replication: one replica has two behaviors (primary and backup behaviors)
3 -- Active Replication: all replica have the same behavior and there is a consensus algorithm
      to vote between them
4
5 Replication_Types: type enumeration (Active , Passive);
6 Replica_Type : Replication::Replication_Types applies to (system, process, thread, processor
      , device);
```

Réplication active

Toutes les copies admettent un comportement symétrique. La prise de décision concernant la sortie finale est effectuée en fonction d'un algorithme de vote qui dépend des hypothèses sur les défaillances et de la spécification du système. Afin d'appliquer la bonne stratégie de réplication, le concepteur doit décrire l'algorithme de vote convenable au système.

Réplication passive

Une seule réplique (qu'on appelle primaire) exécute le traitement demandé et les autres répliques (qu'on appelle secondaires) sont inactifs, jusqu'à la détection d'erreurs (défaillance de la copie primaire). Dans ce cas, après la détection d'erreur, on doit basculer vers une nouvelle configuration pour choisir une nouvelle copie primaire. Le choix de la nouvelle copie primaire est basé sur un algorithme d'élection d'une copie parmi N non actives.

6.3.5 Algorithme de consensus

L'algorithme de consensus ou d'accord est appelé dans deux cas. Le premier consiste à élire la nouvelle copie primaire en cas de défaillance de la copie primaire actuelle parmi deux ou plusieurs copies secondaires dans le cas d'une réplication passive. Le second consiste à voter entre les différentes répliques dans le cas de la réplication active. Dans les deux cas, cet algorithme peut être décrit par l'intermédiaire d'un sous-programme AADL. En AADL, un sous-programme peut être décrit à travers le composant subprogram. Également, l'implantation du sous-programme peut être fournie par l'utilisateur sous forme d'un fichier externe écrit avec un autre langage à savoir le langage **Ada** ou **C**. De même, dans notre contexte, nous considérons aussi les sous-programmes spécifiés avec l'annexe comportemental d'AADL.

Pour supporter tous les cas possibles, nous avons défini l'algorithme de consensus à travers trois propriétés comme il est illustré dans le listing 6.5.

— `Consensus_Algorithm_Source_Text` : pour supporter le cas où le sous-programme est implanté séparément dans un fichier source externe.

— `Consensus_Algorithm_Class` : pour supporter les composants sous-programmes décrit avec AADL à travers un *classifier*.

— `Consensus_Algorithm_Ref` : pour supporter le cas où le sous-programme implantant l'algorithme de consensus est spécifié en utilisant l'annexe AADL comportemental.

Listing 6.5 – Description de l'algorithme de consensus

```
1 -- the consensus algorithm source text correspondent to the out port
2 Consensus_Algorithm_Source_Text: aadlString applies to (port, system, processor, device,
      data access);
3
4 -- the consensus algorithm correspondent to the out port to call an existent subprogram
      implementation
5 Consensus_Algorithm_Class : classifier(subprogram) applies to (port, system, processor,
      device, data access);
6
7 -- to call a subprogram instance from a behavioral annex
8 Consensus_Algorithm_Ref : reference( subprogram) applies to (port, system, processor, device
      , data access);
```

Ces trois propriétés sont appliquées au composant port dans le cas de réplication active de composants logiciels. Ceci est dans le but de voter pour prendre la décision concernant une sortie parmi plusieurs d'un même composant. S'il s'agit d'une réplication passive d'un composant, la propriété décrivant l'algorithme de consensus doit être appliquée au composant répliqué puisqu'il s'agit en fait d'un algorithme d'élection pour le choix de la copie primaire.

6.4 Transformation de modèle

Il s'agit d'une transformation M2M (Model to Model) dans le but d'intégrer les politiques de réplication afin de raffiner le modèle initial. Cette transformation consiste à en appliquer un enrichissement basé sur les règles de transformation définies pour générer un modèle plus précis et concret. Pour cela, nous devons analyser syntaxiquement et sémantiquement le modèle réduit avant d'appliquer les transformations nécessaires et de générer le nouveau modèle enrichi qui doit être lui-même valide et cohérent. Dans cette partie, nous allons définir les règles de transformation du modèle AADL réduit vers le modèle AADL enrichi des différentes propriétés. Nous donnons aussi quelques exemples d'entités AADL enrichies avec l'ensemble de propriétés `Replication_Properties` suivie de leur transformées respectives. L'établissement et l'application de ces règles dépendent de plusieurs facteurs notamment le nombre de répliques, le type de réplication, le type du composant répliqué et l'algorithme de consensus spécifié.

6.4.1 Nombre de répliques

La propriété `Replica_Number` permet de spécifier le nombre de répliques souhaités. Ce nombre correspond exactement au nombre de composants redondants au niveau du modèle généré. Avant d'effectuer la transformation, nous vérifions si ce nombre est bien compris entre les deux constantes prédéfinies `Max_Nbr_Replica` et `Min_Nbr_Replica`. Nous vérifions aussi la propriété `Replica_Identifiers` pour s'assurer que le nombre fourni d'identifiants correspond au nombre de répliques spécifié.

6.4.2 Type de réplication

La génération du modèle AADL enrichi avec les répliques dépend fortement du type de réplication défini par la propriété `Replica_Type`. Les politiques de réplication adoptées ne sont pas les mêmes dans le cas d'une réplication active ou passive. Dans ce qui suit, nous décrivons brièvement la différence entre les deux.

Réplication active

Pour ce type de réplication, toutes les copies (répliques) jouent un rôle identique. Elles reçoivent les mêmes requêtes, les traitent, et émettent la même séquence de réponses. Dans une architecture client/serveur, les serveurs sont répliqués et le client choisit enfin l'une des réponses fournies. Le protocole de choix de la réponse peut être prédéfini ou personnalisé par le concepteur. A titre d'exemple, on peut appliquer un mécanisme de vote majoritaire ou choisir la moyenne des valeurs retournées. Le mécanisme de vote est mis en œuvre de différentes façons selon le besoin de l'application. De ce fait, dans notre contexte, le modèle généré contient `Replica_Number` répliques identiques générées au même niveau hiérarchique du composant répliqué. Chacune d'elles est connectée directement ou à distance à un thread voteur existant ou généré pour faire le choix. Ceci dépend de la propriété utilisée pour décrire l'algorithme de consensus et du type du composant répliqué.

Réplication passive

Contrairement à la réplication active, la réplication passive est basée sur la migration entre deux ou plusieurs configurations. Chaque configuration admet une architecture différente basée sur une organisation des répliques ou des copies. Une même copie peut avoir deux comportements différents dépendants de son état courant : copie primaire ou secondaire. La réception des données, leur traitement et l'émission des réponses sont des tâches à la charge de la copie primaire seulement. En cas de panne de la copie primaire, les copies secondaires détectent l'erreur. Dans ce cas, en se basant sur un algorithme d'élection, une copie secondaire est élue pour remplacer la copie primaire défaillante. Afin de suivre la logique de cette stratégie, nous envisageons générer un nombre de composants égal à `Replica_Number`

supportant la reconfiguration dynamique pour répondre aux besoins d'adaptation. Les répliques ne sont pas identiques comme il est le cas de la réplication active puisqu'elles ne sont pas traitées à pied d'égalité. Pour cela, nous avons utilisé les concepts de modes et transitions entre modes offertes par AADL pour décrire le comportement dynamique de l'architecture. Pour toujours garantir un modèle généré cohérent, nous devons satisfaire des contraintes de reconfiguration responsable de la migration entre modes. Une liste de règles de transformation est établies dans cette vision.

6.4.3 Type du composant répliqué

La nature du composant répliqué admet aussi une grande influence sur le modèle généré. En effet, la réplication d'un composant logiciel est distincte d'un composant matériel ou hybride. Cette différence est due à la hiérarchie de composants AADL. Elle concerne aussi les connexions possibles qui peuvent être établies et la reconfiguration dynamique en se basant sur la notion de modes et de transitions entre modes qui est supportée pour des composants et absente pour d'autres. Nous avons effectué une étude approfondie pour discuter la réplication possible de chaque composant AADL. Le tableau 6.1 présente un récapitulatif sur les composants AADL indiquant son support ou on de la réplication. Enfin, nous supportons la réplication des composants logiciels `thread` et `process`, des composants matériels `device` et le composant `system`. Nous avons omis en contre partie la réplication des composants `data` et `subprogram`. Nous suggérons dans ce cas l'application du mécanisme de diversité plutôt que la réplication. En effet, la diversité [AAM09] vise la fourniture du même service à travers une modélisation et implantation diverses. La réplication de sous-programmes identiques ne garantit pas une meilleure fiabilité de point de vue traitement. Ainsi, nous avons appliqué les propriétés définies dans l'ensemble de propriétés `Replication_Properties` uniquement à un sous ensemble de composants AADL que nous trouvons consistent et cohérent.

Concernant les éléments d'interfaces des différentes répliques, elles seront les mêmes du composant répliqué : les mêmes types (data access, provides/requires subprogram access, data port, event ou data event port) tout en conservant les mêmes modes (in, out, in out). Les répliques héritent aussi les mêmes propriétés du composant répliqué.

TABLE 6.1 – Réplication de composants AADL

Composants AADL					
Logiciels		***Matériels***		***Hybride***	
Process	✓	Bus	✘	System	✓
Thread	✓	Device	✓		
Data	✘	Processor	✓		
Subprogram	✘	Memory	✘		

Réplication de composants logiciels

Si le composant répliqué est un composant AADL logiciel (`data, subprogram, thread, process`), on parle donc de redondance logicielle dans le but d'éviter les pannes logicielles. En l'occurrence, le concepteur doit obligatoirement spécifier un algorithme de consensus avec l'une des propriétés `Consensus_Algorithm_Source_Text`, `Consensus_Algorithm_Class` ou `Consensus_Algorithm_Ref`. Pour cela, outre l'établissement des connexions avec les autres composants, nous devons décrire dans ce cas un `thread` qui joue le rôle de voteur et qui intègre un composant `subprogram`. Ce composant traduit l'algorithme de consensus spécifié. La nature du `thread` généré et de ces éléments d'interfaces dépend de la nature du composant répliqué et de ses interfaces de type `out`, `inout` ou `data access`. Ce thread voteur doit être aussi intégré dans un composant process afin d'être exécuté. A cet égard, la génération du voteur dépend elle même du composant répliqué pour garantir le respect de la sémantique hiérarchique du langage AADL d'une part et d'assurer un processus de vote automatique cohérent d'une autre part. Dans ce qui suit, nous discutons les différentes situations possibles cas par cas.

1. **Thread**

   Le composant `thread` de AADL modélise les processus légers. Son comportement est spécifié par différents moyens. On peut décrire le comportement du thread en utilisant les propriétés pour pointer vers du code source fourni par l'utilisateur en spécifiant les langages et les fichiers sources. De même, on peut utiliser les annexes pour décrire le comportement au sein du même modèle ou à travers l'appel de sous-programmes AADL. Afin de tolérer les pannes de traitement, nous pouvons répliquer les processus légers au sein du même modèle.

   Lors de l'établissement des règles de transformation nécessaires afin d'avoir le modèle final, nous devons prendre en considération la nature du thread objet de réplication et des types de ses connexions établies entre le composant répliqué et les autres composants du modèle. Ces deux variantes influent sur le type du thread voteur qui va être crée afin de modéliser le mécanisme de vote entre les différentes répliques.
   Pour cela, nous envisageons les différents cas suivants. L'algorithme 1 permet de déterminer le protocole du thread voteur dans le cas de réplication d'un thread.

   (a) Si le thread répliqué est sporadique alors le thread voteur est automatiquement sporadique.

   (b) Si le thread répliqué est périodique de période T ayant comme port de sortie un ou plusieurs de type `event` ou `event data`, alors le thread voteur sera hybride de période T.

   (c) Si le thread répliqué est périodique de période T ayant uniquement des ports de sortie de tye `data`, alors le thread voteur sera périodique de la même période T.

Algorithm 1 Algorithme déterminant le protocole du thread voteur en cas de réplication d'un process

```
1: SET_VOTER_THREAD_PROTOCOL ( T : Thread_Instance)
2: Protocol ← GET_PROPERTY_VALUE (T, "Dispatch_Protocol")
3: Has_Data_Port ← false
4: Has_Event_Port ← false
5: Has_Data_Access_Feature ← false
6: Voter_Protocol ← Unknown
7: Voter_Period ← GET_PROPERTY_VALUE (T, "Period")
8: Voter_Priority ← GET_PROPERTY_VALUE (T, "Priority")
9: if Voter_Priority = NIL then
10:    Voter_Priority ← Default_System_Priority
11: end if
12: for each A in FEATURES (T) do
13:    if (IS_DATA(A)) and((IS_OUT(A)) or(IS_INOUT(A))) then
14:       Has_Data_Port ← true
15:    else if ((IS_EVENT_DATA (A)) or(IS_EVENT(A)))and((IS_OUT (A)) or(IS_IN_OUT (A))) then
16:       Has_Event_Port ← true
17:    else if (IS_DATA_ACCESS (A)) then
18:       Has_Data_Access_Feature ← true
19:    else
20:       error "Cannot treat other types of features !"
21:    end if
22: end for
23: switch (Protocol)
24:    case Periodic:
25:          if (Has_Event_Port) then
26:             Voter_Protocol ← Periodic
27:          else if ((Has_Event_Port) or(Has_Data_Access_Feature)) then
28:             Voter_Protocol ← Hybrid
29:          endif
30:    end case
31:    case Sporadic:
32:       Voter_Protocol ← Sporadic
33:    end case
34:    case Hybrid:
35:       Voter_Protocol ← Hybrid
36:    end case
37:    default:
38:       error "Cannot decide about the voter thread type !"
39:    end default
40: end switch
41: if Voter_Protocol = Unknown then
42:    return NULL
43: else
44:    Voter_Th ← Add_New_Thread (Voter_Protocol, Voter_Period, Voter_Deadline, Voter_Priority)
45:    return Voter_Th
46: end if
```

Pour les éléments d'interfaces du thread voteur généré, il admet les mêmes types de ports de type `data`, `event` ou `data event port` ayant comme mode `out` ou `inout` au niveau du composant répliqué. Pour les entrées, nous devons inverser les modes des ports et les redonder un `Replica_Number` fois afin de les connecter aux différents com-

posants répliqués. Pour les sorties, nous devons conserver les mêmes ports ayant le mode `out` ou `in out`.

Dans le cas ou le composant thread répliqué admet des éléments d'interfaces accesseurs, c'est à dire il admet comme élément d'interface `requires` ou `provides data access`, on parle de partage de données. Nous distinguons les deux cas suivants :

(a) Si le thread répliqué admet une interface de type `provides data access`, il admet dans ce cas une donnée d'accès partagé comme sous composant. Puisque, la réplication du thread engendre la réplication implicite de ses sous-composants, la donnée va être par la suite répliquée au sein de toutes les copies. Nous devons alors répliquer les mêmes éléments d'interfaces au niveau des répliques tout en conservant la direction de la connexion. Par contre, cet élément sera inversé au niveau du thread voteur. Ainsi, le voteur admet autant de nombre de répliques, des interfaces de type `requires data access` dans chacun est connecté à une réplique et il accède à une donnée différente.

(b) Si le thread répliqué admet un feature de type `requires data access`, dans ce cas il nécessite l'accès à un composant qui lui est externe. On parle d'accès distant. Le traitement relatif des répliques par rapport à la donnée partagée dépend du type de l'accès spécifié à travers la propriété `Access_Right`. Si on parle de mode lecture seule (Read_Only), nous n'avons pas de problème de conflit contrairement au mode écriture seule (Write_Only) ou lecture/écriture (Read_Write). Nous devons gérer dans ce cas l'accès concurrent à la donnée partagée.

2. **Process** Dans le cas de réplication d'un composant process, les répliques sont elles mêmes des process générées au même niveau hiérarchique du process répliqué (Voir l'algorithme 3). Par la suite, pour chacun des éléments d'interfaces de type `out`, `inout` ou `data access` du process répliqué, le designer attribue un algorithme de consensus. Ce dernier, afin d'être exécuté, doit être appelé par un processus léger qui doit être à son tour exécuté dans un process. Pour cela, nous devons créer un process voteur situé au même niveau que les répliques. Ce process voteur ainsi que les différentes répliques sont liés au même processeur sur lequel est déployé le process répliqué. Cette propriété est spécifié à travers la propriété `Actual_Processor_Binding`. Le thread voteur sera par la suite crée au niveau du process voteur généré. Le type du ce thread et ces différentes propriétés dépendent dans ce cas de l'architecture interne du process répliqué. Il s'agit de déterminer le protocole, la période, l'échéance et la priorité des threads contenus dans le process répliqué et qui lui sont directement liés aux éléments d'interfaces auxquels on applique les algorithmes de vote. L'algorithme 2 permet de déterminer les différentes propriétés du thread voteur dans le cas de réplication d'un process.

Algorithm 2 Algorithme déterminant le protocole du thread voteur en cas de réplication d'un process

```
1: SET_VOTER_THREAD_PROTOCOL (Pro : Process_Instance)
2: Has_Data_Port, Has_Event_Port, Has_Data_Access_Feature ← false ;
3: Has_Periodic_Thread, Has_Sporadic_Thread, Has_Hybrid_Thread ← Unknown
4: Voter_Protocol ← Unknown
5: Voter_Period, Voter_Deadline, Voter_Priority ← Unknown
6: if Voter_Priority = NIL then
7:    Voter_Priority ← Default_System_Priority
8: end if
9: for each A in FEATURES (Pro) do
10:    if (IS_OUT (A) orIS_INOUT (A) orIS_DATA_ACCESS (A)) then
11:       if  IS_DATA (A)  then
12:          Has_Data_Port ← true
13:       else if IS_EVENT(A) orIS_EVENT_DATA(A) then
14:          Has_Event_Port ← true
15:       else
16:          Has_Data_Access_Feature ← true
17:       end if
18:       for each SUB in SUBCOMPONENTS (Pro) do
19:          Category ← GET_COMPONENT_TYPE (SUB)
20:          if ( Category = thread) and(IS_CONNECTED (SUB, A)) then
21:             Protocol ← GET_PROPERTY_VALUE (SUB, "Dispatch_Protocol")
22:             if Protocol = Periodic then
23:                Has_Periodic_Thread ← true
24:             else if Protocol = Sporadic then
25:                Has_Sporadic_Protocol ← true
26:             else if Protocol = Hybrid then
27:                Has_Hybrid_Protocol ← true
28:             end if
29:             if Voter_Period = unknown then
30:                Voter_Period ← GET_PROPERTY_VALUE (SUB, "Period")
31:             else
32:                Voter_Period ← min (Voter_Period, GET_PROPERTY_VALUE (SUB, "Period"))
33:             end if
34:             if Voter_Deadline = unknown then
35:                Voter_Deadline ← GET_PROPERTY_VALUE (SUB, "Deadline")
36:             else
37:                Voter_Deadline ← max (Voter_Deadline, GET_PROPERTY_VALUE (SUB, "Deadline"))
38:             end if
39:          end if
40:       end for
41:    end if
42: end for
43: if Has_Hybrid_Protocol = true then
44:    Voter_Protocol ← Hybrid
45: else if Has_Sporadic_Protocol = true then
46:    Voter_Protocol ← Sporadic
47: else if Has_Periodic_Protocol = true then
48:    if Event_Port = false then
49:       Voter_Protocol ← Periodic
50:    else
51:       Voter_Protocol ← Sporadic
52:    end if
53: end if
```

```
54: if Voter_Protocol = Unknown then
55:    return NULL
56: else
57:    Th ← Add_New_Thread ( Voter_Protocol, Voter_Period, Voter_Deadline, Voter_Priority)
58:    return Th
59: end if
```

3. **Data**

   Un composant AADL de type `data` représente un type de données. Ce composant peut être instancié au niveau des éléments d'interface (`features`) de type `data` ou `event data` qui se trouvent dans les `process`, les `threads` ou encore les sous-programmes. Il peut être déclaré aussi comme un paramètre de sous-programme ou comme une variable partagée entre plusieurs composants du modèle. Pour la réplication dans le but de tolérer les pannes, on n'a pas besoin de répliquer des données de type simple qui sont traduits par des variables au niveau code. En plus, les données qui sont déclarées comme sous-composants de thread ou de process, ils sont implicitement répliquées avec le composant parent qui les contiennent. Nous nous intéressons pas dans ce cadre à la réplication de données.

4. **Subprogram**

   Pour éviter les pannes de traitement, il est conseillé de coder le traitement demandé différemment afin de garantir le bon fonctionnement. Dans ce cas, on parle de diversité plutôt que réplication. La diversité consiste à offrir le même service à travers une modélisation et implantation distinctes [AK84]. La redondance de sous-programmes identiques ne peut pas garantir une meilleure fiabilité de point de vue traitement. Donc, nous visons diversifier le traitement. Contrairement, lorsque nous visons répliquer le même traitement, nous n'aurons pas besoin de répliquer le sous-programme. Il suffit de l'appeler autant de fois que l'on souhaite. Ainsi, le traitement sera répété sans avoir recours à un mécanisme de réplication. De plus, comme il est le cas pour les données, les sous-programmes qui sont déclarées comme sous-composants de thread sont implicitement répliqués avec le composant parent qui les contiennent.

Réplication de composants matériels ou hybrides

La réplication des composants matériels est totalement différente à celle des composants logiciels. Ceci revient à la spécificité de chaque type de composant en termes de sa nature, de sa manière d'organisation (hiérarchie entre les composants), des connexions possibles, des propriétés et même du support ou non de la reconfiguration dynamique. Dans cette section, nous nous intéressons à la réplication des composants matériels supportés.

Si le composant répliqué est un composant AADL matériel (system ou device), on parle donc de redondance symétrique et simple de composants identiques dans le but d'éviter une panne matérielle. En l'occurrence, les composants répliqués sont générés au même niveau hiérarchique que celui original. Quant à la génération du thread voteur, elle dépend

de l'acheminement des connexions partant des éléments d'interfaces des répliques jusqu'à arriver à un composant process. Ceci est du à la logique d'exécution des sous programmes dans un environnement logiciel.

Réplication passive

Contrairement à la réplication active, la réplication passive ne fait pas une distinction entre les différents types possibles du composant objet de réplication. Ce type de réplication, basé sur le mécanisme d'élection pour migrer entre deux configurations, impose la génération de `Replica_Number` composants identiques supportant la reconfiguration dynamique afin d'obéir au besoins d'adaptation. Nous avons résolu le problème de reconfiguration à travers le mécanisme de modes et de transitions entre modes défini par le standard AADL. L'idée de solution est basée sur les modes déclarés pour chacune des répliques. Ces modes doivent prendre en considération l'état possible d'une réplique qui est soit primaire soit secondaire. De ce fait, chaque réplique peut avoir deux modes. Au départ, une seule réplique doit avoir le mode primaire comme mode initial tandis que les autres répliques admettent le mode secondaire comme le mode initial. Pour toutes les répliques, la transition d'un mode à un autre est fait suite au déclenchement d'un événement traduit par la réception d'une donnée. A ce moment là, un algorithme d'élection est appelée pour choisir laquelle des répliques sera élue pour changer d'état.

6.4.4 Algorithme de consensus

L'algorithme de consenus décrit avec l'une des propriétés `Consensus_Algorithm_Source_Text`, `Consensus_Algorithm_Class` ou `Consensus_Algorithm_Ref` permet de spécifier l'algorithme adopté pour le vote dans le cas d'une réplication active ou pour l'élection de la copie primaire dans le cas d'une réplication passive. Nous avons défini trois propriétés différentes supportant les divers types de sous-programmes AADL pour offrir plus de flexibilité. Étant donné que les sous-programmes sont des entités passives en AADL, elles doivent être appelées par des processus légers pour s'exécuter. Ainsi, la propriété décrivant l'algorithme de consensus se transforme naturellement en sous-programme à condition d'existence d'un thread qui pourra l'exécuter. Si non, cette propriété sera transformée en un `thread` contenant un sous-composant `subprogram`. Donc la transformation de cette propriété dépend de la nature du composant répliqué. D'autre part, visant une approche générique de réplication qui supporte un nombre de répliques variable, nous avons proposé de mettre en considération cette contrainte. Pour cela, nous exigeons que le sous-programme voteur admet comme entrée un tableau de valeurs. Ces valeurs représentent les sorties sujettes de vote correspondantes à chacune des répliques. Pour cela, nous générons également un algorithme appelé *Map_Spg* qui prend comme entrée l'ensemble des sorties des différentes répliques dans le but de les organiser sous

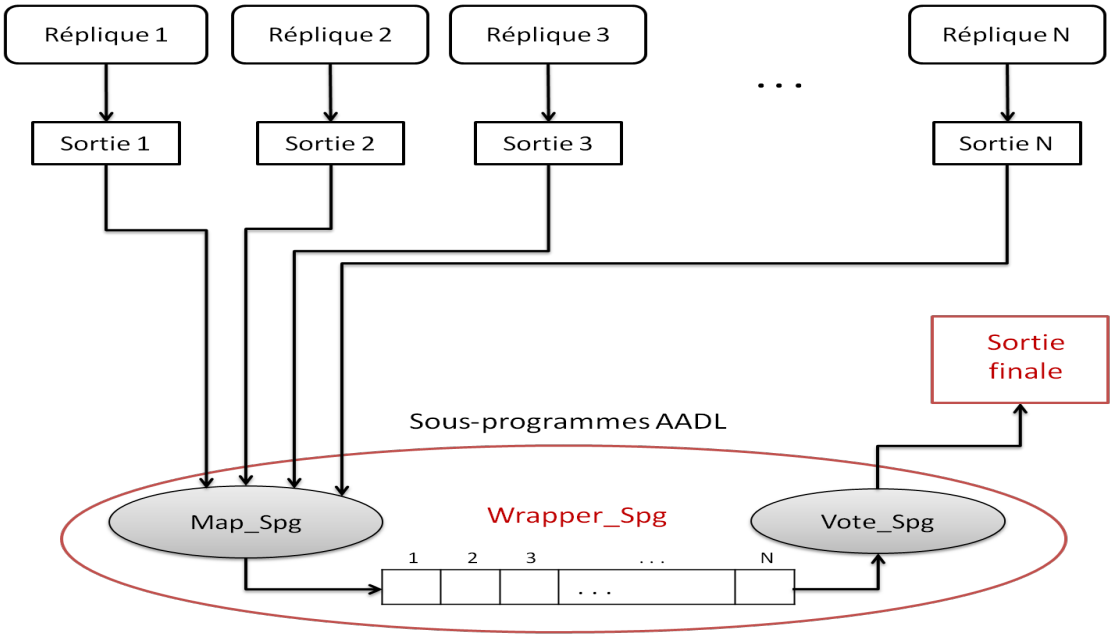

FIGURE 6.2 – Génération du sous-programme voteur

forme de tableau de valeurs qui sera à son tour l'entrée du sous programme *Vote_Spg*. Considérant les contraintes exigées par AADL pour l'insertion des sous-programmes au niveau d'un thread, un sous programme nommé *Wrapper_Spg* est généré afin d'englober les deux sous-programmes *Map_Spg* et *Vote_Spg* comme l'indique la figure 6.2.

Toutes ces contraintes imposées pour l'utilisation des propriétés de réplication, leur vérification et la génération du modèle sont prises en compte lors du développement du module de réplication. Le choix du développement de ce module comme extension de la suite d'outils Ocarina est pilotée par sa structure extensible et bien organisée et sa richesse en termes de routines développées au niveau de la bibliothèque centrale. Dans la section suivante, nous détaillons, en un premier lieu, l'architecture de base de la suite d'outils Ocarina et nous nous focalisons, en deuxième lieu, sur son extension par notre module de réplication.

6.5 Implantation du module de réplication

Dans le but de faciliter l'utilisation de notre approche, nous avons enrichi la suite d'outils Ocarina pour supporter notre extension. Cette suite de bibliothèques logicielles permet la manipulation des modèles AADL et faire plusieurs analyses et vérifications. Dans cette section, nous décrivons l'architecture globale de base de Ocarina, puis nous détaillons les étapes suivies pour l'étendre dans le but d'intégrer notre approche de réplication.

6.5.1 Architecture détaillée de Ocarina

Outre l'analyse syntaxique et sémantique des modèles AADL, Ocarina permet également la vérification formelle des modèles AADL avec les Réseau de Petri. Elle contient un support d'analyse d'ordonnancement avec Cheddar [SLNM04]. Aussi, cet outil permet de générer des applications distribuées à la base de divers supports d'exécution en Ada, RTSJ ou C/C++ en se basant sur les intergiciels `PolyORB` et `PolyORB_High_Integrity`. Ocarina supporte aussi la POA à travers l'annexe AO4AADL [LKZJ13] qui étend un modèle AADL par des aspects architecturaux.et permet la génération de son code en AspectJ. Finalement, grâce à son architecture bien organisée (Figure 6.3), Ocarina est facilement extensible. Cette architecture renferme trois parties :

— **Bibliothèque centrale :** Cette partie offre plusieurs routines qui servent à la construction et le parcours des arbres syntaxiques.

Ces routines, considérées comme abstractions de bas niveau, servent à manipuler les fichiers et les chaînes de caractères et faciliter l'accès aux nœuds et l'échange des arbres syntaxiques. Pour augmenter les performances du compilateur, les chaînes de caractères sont traitées sous forme de code de hachage aux niveaux des deux parties frontales et dorsales. Les premières permettent de construire les nœuds de ces arbres et les deuxièmes parcourent et lisent les mêmes nœuds dans le but de produire du code source ou de faire des analyses et des vérifications. Pour cette raison, ces routines sont spécifiques non seulement au langage AADL supporté dans tout le compilateur mais aussi à chacun des formalismes supportés comme le langage Ada ou C qui font partie des générateurs des parties dorsales.

— **Parties frontales :** Ces parties permettent l'analyse de la syntaxe et la sémantique des fichiers sources écrites en AADL ou en un autre langage annexe pris en charge. Outre AADL, le formalisme de la partie frontale principale de Ocarina, d'autres parties frontales secondaires sont développées dans le but de supporter la syntaxe de quelques annexes de ce langage. Suivant les compilateurs modernes, toutes ces parties renferment un analyseur lexical, syntaxique et sémantique conformément aux grammaires concernées pour l'analyse du langage en question produisant par la suite un arbre syntaxique abstrait (AST : *Abstract Syntactic Tree*). Une phase d'instanciation prend comme entrée l'AST généré dans le but de faire des analyses de certaines règles avancées. Elle consiste à produire un nouvel arbre syntaxique. Cet arbre résultant de l'application récursive, sur chaque composant, des propriétés et des sous-composants dont il hérite. La racine de cet arbre est un composant de type system contenant toute la topologie de l'application contenant l'ensemble des composants AADL du modèle dans le but de préparer l'environnement de génération de code ou d'analyse.

Au début, une seule partie analysant le formalisme AADL existe. Puis, Ocarina a été étendue par plusieurs autres parties frontales pour supporter différentes annexes telles

que l'annexe d'erreurs, l'annexe comportementale et l'annexe AO4AADL. L'ensemble de ces parties, font appel à des routines de la bibliothèque centrale de la suite Ocarina.

— **Parties dorsales :** Ces parties prennent comme entrée l'arbre syntaxique décoré généré après l'instanciation du modèle AADL. Après l'application d'une expansion, cet arbre est parcouru pour générer le code souhaité par l'utilisateur. L'étape d'expansion vise simplifier ou ajouter certaines constructions de l'arbre produit de la partie frontale dans le but de le simplifier et de l'enrichir. A la différence des langages de programmation classiques, le langage AADL, à partir d'une seule spécification AADL, produit plusieurs fichiers sources. Pour prendre en compte cette différence et pour garantir la flexibilité d'extension, la maintenance aisée et la simplicité relative de l'optimisation du code généré par les parties dorsales, la phase d'expansion est suivie d'une phase intermédiaire de transformation de l'arbre d'instance en un arbre syntaxique du langage cible qui sera à son tour traversé pour générer du code source.

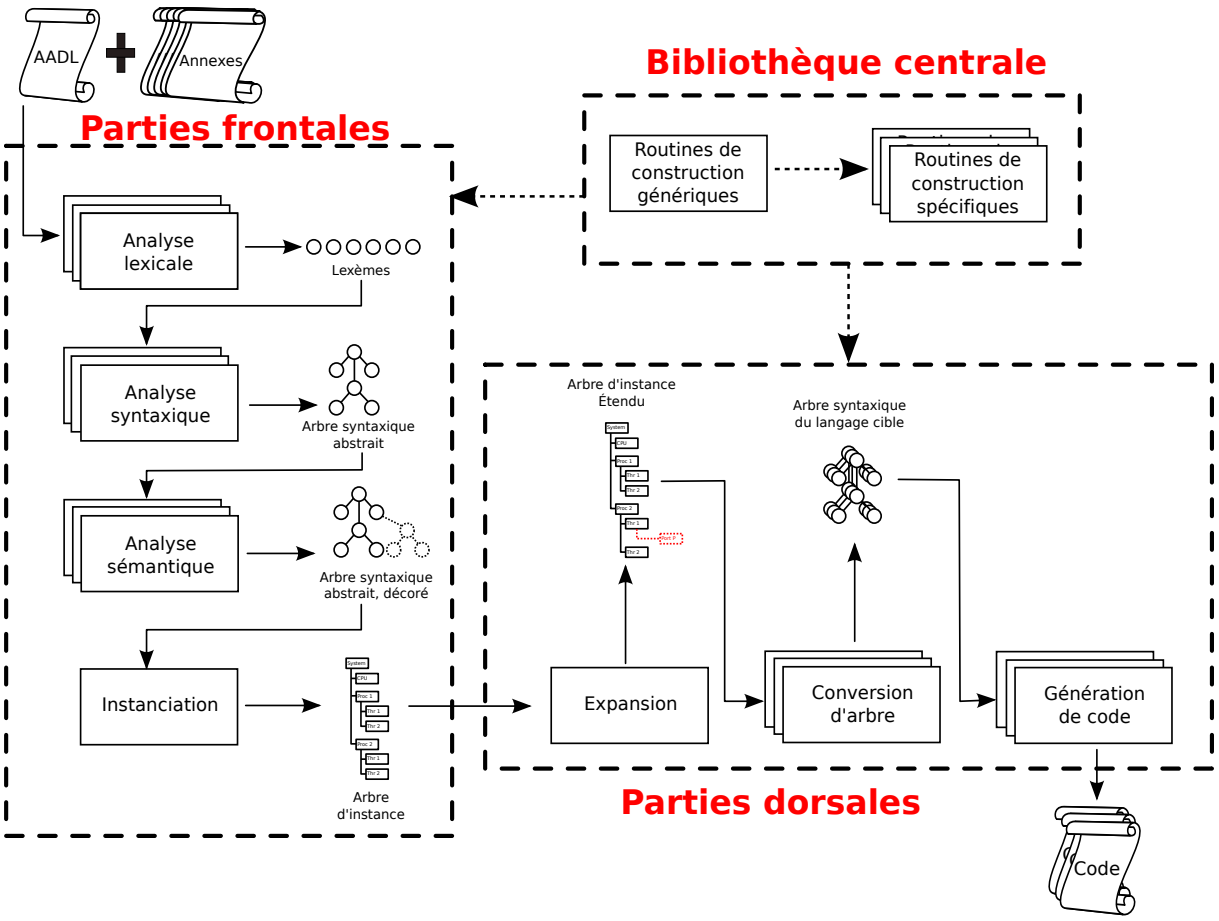

FIGURE 6.3 – Architecture de la suite d'outils Ocarina [Zal08]

Pour toute ces raisons, nous avons choisi Ocarina pour l'étendre afin de supporter notre approche de réplication. De cette manière, Ocarina peut non seulement analyser et vérifier l'utilisation des propriétés de réplication mais aussi générer le modèle intermédiaire AADL enrichi avec les différentes répliques et les arbitres. Le processus adopté pour la génération de code est détaillé dans ce qui suit.

6.5.2 Extension de la suite d'outils Ocarina

Notre extension de cette suite est établie dans le but de fournir un outil permettant d'appliquer notre approche de réplication proposée. Notre choix de Ocarina a été basé, d'une part, sur son architecture extensible et bien structurée et d'autre part sur l'absence d'un autre outil implantant une telle partie dorsale. Un autre outil très connu pour la manipulation et l'analyse des modèles AADL est Osate, un plugin Eclipse. Cet outil est très puissant au niveau de ses parties frontales visant l'analyse du langage AADL et de certaines annexes qui y sont implantées et offrant une représentation graphique de ces modèles. Toutefois, Osate ne supporte pas la génération de code et ne définit même pas les parties dorsales.

Pour implanter notre module, nous avons contribué au niveaux des trois principales parties de la suite Ocarina. Notre extension a été basée sur quatre étapes détaillées comme suit :

1. La vérification de l'utilisation des propriétés de réplication correctement. Dans cette partie, nous vérifions la validité de l'utilisation de l'ensemble des propriétés définies. Le concepteur doit spécifier le nombre de répliques qui doit être compris entre le nombre minimal et le nombre maximal de répliques fixés au niveau du fichier `Replication_Properties` et qui correspond aussi au nombre d'éléments de la liste, valeur de la propriété `Replica_Identifiers`. Le concepteur doit également spécifier la stratégie de réplication choisie qui est soit active ou passive. Nous vérifions après que l'algorithme de consensus a été bien décrit par le concepteur pour toutes les sorties des composants répliqués. Cet algorithme est soit prédéfini comme le vote majoritaire ou le calcul de la moyenne des valeurs, soit personnalisé et développé aussi par le concepteur.
2. L'extraction de la liste des propriétés de chaque composant répliqué si toutes les propriétés sont validées. Ceci est fait par la collecte de toutes les propriétés de réplication spécifiés à un composant répliqué comme un enregistrement. Par la suite, selon le type du composant répliqué et des propriétés extraites, nous appliquons le traitement approprié en se basant sur les algorithmes établies pour régir les transformations supportées.
3. L'extension du modèle AADL de base avec des répliques. Comme déjà dit, cette extension est basée sur l'ensemble de règles de transformations que nous avons établi. A

ce niveau, de point de vue traitement, il s'agit de manipulation d'arbres AADL récupérés à partir du modèle de base. Selon le type de composant répliqué et la stratégie de réplication sélectionnée, un algorithme de réplication est appliqué sur le modèle. Il s'agit d'insérer de nouveaux composants identiques au composant répliqué, d'insérer le composant arbitre et d'établir les connexions nécessaires entre les composants générés et ceux originaux.

Algorithm 3 Algorithme de réplication du composant `Process`

```
1: procedure Process_Replication (P : Replication_Info) is
2: Replicated_Comp ← Replicated_Comp (P);
3: Corresponding_comp ← Get_Corresponding_Component (Replicated_Comp);
4: Parent_Comp ← Container_Component (Replicated_C omp);
5: Replica_Number ← Replica_Number (P);
6: Replica_List ← Generate_Replica (Replicated_Comp, Identifiers (P), CC_Process);
7: Voter_Pro ← Add_New_Comp_Type (CC_Process);
8: Voter_Pro_Impl ← Add_New_Comp_Impl (Voter_Pro);
9: Voter_Pro_Subcomp ← Add_Subcomp (Parent_Comp, CC_Process, Voter_Pro_Impl);
10: Voter_Th ← Add_New_Comp_Type (CC_Thread);
11: Voter_Th_Impl ← Add_New_Comp_Impl (Voter_Th);
12: Voter_Th_Subcomp ← Add_Subcomp (Voter_Pro_Impl, CC_Thread), Voter_Th_Impl);
13: Voter_Spg ← Add_New_Comp_Type (CC_Subprogram);
14: Do_Vote_Spg ← Add_New_Comp_Type (CC_Subprogram);
15: Add_Calls (Voter_Th_Impl, Voter_Spg);
16: Add_Calls (Voter_Th_Impl, Do_Vote_Spg);
17: for each Param in PARAMETERS (Do_Vote_Spg) do
18:    Add_Parameter_Connection (Voter_Spg, Do_Vote_Spg, Param);
19: end for
20: for each F in FEATURES (Voter_Th_Subcomp) do
21:    if Is_In (F) then
22:       Add_Parameter_Connection (Voter_Th_Subcomp, Voter_Spg);
23:    else if Is_Out (F) then
24:       Add_Parameter_Connection (Do_Vote_Spg, Voter_Th_Subcomp);
25:    end if
26: end for
27: for each F in FEATURES (Voter_Pro_Subcomp) do
28:    if Is_In (F) then
29:       Add_Connection (Voter_Pro_Subcomp, Voter_Th_Subcomp);
30:    else if Is_Out (F) then
31:       Add_Connection (Voter_Th_Subcomp, Voter_Pro_Subcomp);
32:    end if
33: end for
34: for each Connection in CONNECTIONS (Parent_Comp) do
35:    if SOURCE (Connection) = Replicated_Comp then
36:       Add_Connection (Voter_Pro_Subcomp, DESTINATION (Connection));
37:    else if DESTINATION (Connection) = Replicated_Comp then
38:       for each Replica in Replica_List do
39:          Add_Connection (SOURCE (Connection), Replica);
40:       end for
41:    end if
42: end for
43: for each Replica in Replica_List do
44:    Add_Connection (Replica, Voter_Pro_Subcomp);
45: end for
46: for each Prop in PROPERTIES (Parent_Comp) do
47:    if Replicated_Comp in APPLIES_TO_PROP (Prop) then
48:       for each Replica in Replica_List do
49:          Add_Property (Replica, Prop, Parent_Comp);
50:       end for
51:    end if
52: end for
```

A titre d'exemple, l'algorithme 3 décrit les différentes étapes à suivre pour appliquer une réplication active d'un composant Process.

4. Aprés avoir intégré au niveau du modèle les différentes répliques et les arbitres, l'arbre résultant sera la base du modèle intermédiaire généré. L'arbre modifié donne lieu à une génération de ce modèle sous forme d'un fichier d'extension (`*.aadl`). Ce modèle doit être cohérent. Il doit respecter les contraintes fixées par le standard AADL comme l'hiérarchie de composant et les connexions possibles.

Les deux premières étapes sont développées au niveau de la partie frontale du compilateur Ocarina permettant d'analyser lexicalement, syntaxiquement et sémantiquement le modèle AADL enrichi par les propriétés *Replication_Properties* analysées elles mêmes durant ces phases. Quant aux autres étapes, elles agissent su niveau de la partie dorsale. La troisième étape consiste en l'expansion du modèle AADL de base permettant de générer un AST plus riche que celui généré d'après la partie frontale. Cet AST intègre les composants répliqués. La génération du code source AADL final traduit l'étape de transformation du AST intermédiaire étendu vers le code source. Ce dernier, peut être à son tour analysé et considéré comme entrée pour une autre partie dorsale ou frontale dans le but de faire d'autres analyses. Il peut être ainsi objet d'analyse et de vérification ou de génération de code (particulièrement pour des annexes).

6.6 Conclusion

Dans ce chapitre, nous avons décrit notre approche proposée pour le support de la réplication automatique des composants AADL en se basant sur les propriétés. D'abord, nous avons présenté la liste des propriétés définies. Puis, nous avons présenté l'ensemble de règles de transformations. Enfin, nous avons décrit notre extension de la suite d'outils Ocarina pour supporter notre approche de réplication en tenant compte des deux styles de réplication. Dans le chapitre suivant, nous détaillons l'approche proposée pour la génération du code tolérant aux pannes à partir de l'annexe d'erreurs.

7

Génération de code

7.1 Introduction

La dernière phase de notre processus de développement proposé pour la production des systèmes temps-réel distribués tolérants aux pannes est la génération automatique du code. Cette phase, comme celles qui la précèdent, fait la séparation entre les préoccupations fonctionnelles et transversales. En particulier, nous nous intéressons à la génération du code tolérant aux pannes. Durant tout le processus de développement, nous avons adopté la démarche MDA visant une génération automatique du code source pilotée par les modèles. Nous adoptons également cette démarche pour la génération du code fonctionnel ainsi que le code tolérant aux pannes. Pour la génération du code fonctionnel, nous partons à partir des modèles AADL de base. Quant au code tolérant aux pannes, nous partons de l'annexe d'erreurs décrivant les politiques de tolérance aux pannes pour arriver à la génération du code aspect qui sera par la suite tissé avec le code fonctionnel. Nous avons déjà mentionné que le langage Ada a été choisi comme le langage fonctionnel cible. Pour des raisons de conformité, nous choisissons le langage AspectAda.

Dans ce chapitre, nous commençons, d'abord, par décrire la démarche MDA que nous adoptons pour la génération de code. Puis, nous introduisons le langage Ada pour justifier son choix. Ensuite, nous décrivons le générateur de code fonctionnel et le processus de génération de code tolérant aux pannes. Par la suite, nous détaillons les règles établies pour la génération de l'annexe d'erreurs vers le langage AO4AADL et de ce dernier vers le langage d'aspect.

7.2 Ada pour le développement temps-réel

Ada [Ada12] est un langage de choix pour le développement des système temps-réel complexes comme les systèmes avioniques. Dans ce contexte, nous notons que la majorité des nouveaux avions, Boeing 787 Dreamliner, sont développées avec le langage

Ada [MSH11] comme il est le cas aussi pour le nouveau système de contrôle du trafic aérien en Angleterre. Ada est utilisé pour développer les systèmes critiques non seulement dans le domaine aéronautique, mais aussi dans le domaine spatial et médicale grâce0 à sa forte puissance prouvée dans le domaine temps-réel. Ada est un langage généraliste synthétisant les avantages des langages existants et les intégrant dans un ensemble cohérent normalisé à l'échelle internationale. Grâce à sa syntaxe claire et inspirée du langage Pascal, Ada permet la construction des systèmes sûrs et fiables. De plus, il offre une forte sémantique favorisant les optimisations. En outre, Ada dispose d'un compilateur puissant capable de détecter les erreurs dès les premières phases de compilation ce qui minimise la probabilité de propagation des erreurs à l'exécution. Particulièrement, ce langage offre divers moyens pour un support fort du temps-réel. Nous citons par exemple :

— Le typage statique et la généricité ;

— Le support des aspects temps-réel (tâches, objets protégés et partagés, interruptions, gestion du temps) ;

— La synchronisation, le multi-tâches, et les rendez-vous ;

— Les bibliothèques normalisées ;

Comparé aux autres langages, procéduraux ou orientés objets comme C, C++ ou Java, Ada permet non seulement de minimiser le coût de développement et d'intégration des applications, mais aussi d'offrir certaines garanties à travers les approches qui lui sont liées.

Pour pouvoir utiliser le langage Ada dans des domaines qualifiés de critiques, il doit respecter certaines exigences garantissant sa sécurité et sa sûreté de fonctionnement [Bar08]. Notamment, le déploiement et la configuration doivent être effectués statiquement pour respecter les exigences. Deux approches sont proposées pour développer des systèmes critiques en Ada nécessitant un haut niveau de fiabilité et de sécurité qui sont le profil Ravenscar et l'approche Spark détaillées ci-après.

7.2.1 Profil Ravenscar

C'est un profil [ABT04] constitué d'un ensemble de restrictions pour limiter l'usage du langage Ada aux seules constructions pour garantir l'absence d'interblocage, l'analyse statique d'ordonnancement, et l'inversion bornée temporellement de priorités entre les tâches. Certaines constructions comme les entrées multiples d'objets protégés ou partagés, les entrées dans les tâches, ou les rendez-vous peuvent causer l'augmentation des mécanismes d'interaction entre les tâches et du nombre de chemins d'exécution. Ainsi, elles rendent complexe l'analyse statique d'un système. Pour cette raison, le profil Ravenscar interdit ces types de construction.
Pour ce faire, afin de garantir que l'ensemble de tâches à analyser soit invariant, le profil exige que celles-ci ne doivent jamais se terminer et qu'aucune tâche ne sera créée après avoir

initialisé l'application (désigné par l'élaboration en Ada) [Zal08].
Pour borner l'inversion de priorité et garantir l'absence d'interblocage, le profil Ravenscar recommande l'utilisation du protocole de plafonnement de priorité (*Priority Ceiling Protocol (PCP)*) pour gérer l'accès aux objets protégés.
Visant une analyse d'ordonnancement selon RTA (*Response Time Analysis*) ou RMA (*Rate Monotonic Analysis*), Ravenscar exige l'utilisation des tâches sporadiques ou périodiques. Les premières sont des tâches activées suite à la réception d'événement. Deux événements successifs sont obligatoirement éloignés d'un temps minimal connu à l'avance. Les deuxièmes sont des tâches déclenchées par un événement temporel à des intervalles réguliers.

7.2.2 Approche Spark

L'approche Spark [Bar12] est utilisée dans les domaines critiques à savoir les chemins de fer et l'avionique grâce aux techniques offertes pour la construction des programmes Ada corrects. A l'aide de cette approche, on peut prouver certaines propriétés fonctionnelles, analyser les flux de données ou même produire dans certains cas les systèmes corrects par construction. En effet, Spark permet d'annoter des constructions du langage Ada pour exprimer des contraintes de droit d'accès ainsi que des contraintes de précédence et d'autres afin de décrire les flux de données et d'exécution. La conformité du code des annotations avec le code source Ada est vérifié avec des outils spécifiques dédiés. Ainsi, on peut avoir des programmes corrects par construction prouvés par des théorèmes.

7.3 Processus de génération de code

Pour la génération de code, nous avons adopté l'approche MDA non seulement pour le code fonctionnel mais aussi pour celui tolérant aux pannes. Pour la génération du code métier, nous avons utilisé les générateurs existants de la suite d'outils Ocarina basés sur l'intergiciel PolyORB_High_Integrity [LZPH09]. Quant à la génération de la tolérance aux pannes, nous avons appliqué les étapes de l'approche MDA, allant de PIM, passant par le PSM pour arriver finalement à générer du code aspect spécifique. Dans cette section, nous décrivons brièvement les deux générateurs de code.

7.3.1 Génération du code fonctionnel

La première phase de génération de code consiste à implanter les exigences fonctionnelles de notre système sans prendre en considération les préoccupations techniques. Une fois le modèle du système déjà conçu, le concepteur procède à la génération automatique du code fonctionnel correspondant par l'intermédiaire d'un générateur de code. Particulièrement, pour la génération du code correspondant à un modèle décrit avec le standard AADL, il existe des compilateurs permettant la traduction en différents langages à savoir

Ada [Ada12], C ou RTSJ [Aut09] qui sont offerts par la suite d'outils Ocarina fondu sur l'intergiciel PolyORB-HI [HZ07]. Ces générateurs permettent d'obtenir du code en langage de programmation impératif à partir d'une description AADL.

7.3.2 Intergiciel PolyORB-HI

Influencé par l'architecture de l'intergiciel schizophrène POLYORB [Qui03], l'intergiciel minimal supportant toutes les constructions AADL nécessaires pour produire des applications temps-réel réparties embarquées, a été nommé PolyORB-HI [Zal08]. Cet intergiciel supporte les constructions AADL suivantes :

— **Les processus légers** Divers types de tâches sont supportées dans les systèmes temps-réel répartis embarqués. L'intergiciel PolyORB-HI supporte uniquement trois types qui sont les tâches périodiques, sporadiques et hybrides. Il offre un service dit de parallélisme permettant l'instanciation pour chaque composant thread du type défini dans le modèle AADL des entités génériques correspondantes. La manière d'implantation de chacune des catégories de tâche supportée est compatible avec les restrictions du profil Ravenscar et respectent les recommandations des systèmes critiques.

— **Les éléments d'interface**

Afin de garantir la fiabilité de la communication des différentes entités de l'application distribuée au niveau du code fourni à l'utilisateur, une instance des éléments d'interfaces est créée par l'intergiciel minimal PolyORB-HI pour gérer la lecture et l'écriture dans ces éléments. Cette instanciation est effectuée pour chaque port d'un thread ou d'un sous-programme AADL permettant ainsi l'envoi et la réception des messages sans tenir compte de l'identité de la source ou de la destination, l'emplacement de la destination (locale ou distante) ou même si le port est connecté à une ou plusieurs destination. Puisqu'il s'agit d'un intergiciel dédié pour les systèmes temps-réel, les différentes actions de lecture et d'écriture sont implantées d'une manière garantissant le déterminisme et l'absence d'interblocage.

— **Les données partagées**

PolyORB-HI est conçu d'une façon à garantir la protection des données partagées entre les différentes entités de l'application. Dans le but d'implanter des méthodes d'accès aux données, cet intergiciel fournit des routines de verrouillage et de déverrouillage. Ces méthodes peuvent être appelées par le code de l'utilisateur ou au niveau du code généré. L'accessibilité des données partagées est gérée par les méthodes qu'elles exposent et la sûreté de fonctionnement en terme de la concurrence est garantie par les routines de verrouillage offertes par l'intergiciel.

La configuration de cet intergiciel se fait d'une manière automatique à partir des modèles AADL [ZPH08]. Le calcul et l'allocation de toutes les ressources nécessaires sont faits d'une

manière statique sans intervenir l'utilisateur. De plus, l'intergiciel PolyORB-HI rassemble uniquement les composants faiblement personnalisables ce qui rend sa taille extrêmement réduite. Trois versions de cet intergiciel présentant les mêmes fonctionnalités existent. Une version écrite en Ada, une en C et une autre en Java permettent de concevoir des applications temps-réel réparties embarqués respectivement en Ada, en C et Java. Dans la section suivante, nous décrivons chacune d'elles.

7.3.3 Générateurs disponibles

La première version développée de l'intergiciel minimal PolyORB-HI a été écrite en Ada supportant un sous-ensemble des restrictions du profil Ravenscar que les systèmes temps-réel doivent respecter à la différence de la deuxième version écrite en C. Des modèles AADL sont ainsi manipulés et analysés pour déduire et générer la configuration complète de l'intergiciel minimal. Ainsi, Une application décrite en AADL peut être générée automatiquement pour être exécutée grâce à cet intergiciel qui, à la base duquel, l'outil ocarina offre différents générateurs. Ocarina permet à l'utilisateur de générer du code à partir d'une description de l'architecture AADL vers une application Ada, C ou Java en cours d'exécution sur la base de PolyORB-HI.

— **Générateur Ada**

C'est avec Ada qu'a été développée la première version de l'intergiciel POLYORB-HI. À chaque entité de l'application est générée une hiérarchie de paquetages qui comporte toutes les déclarations de types de données utilisés ainsi qu'un ensemble de fonctions et de procédures assurant la traduction des sous-programmes nécessaires pour chaque nœud. Une description plus détaillée est disponible dans [Zal08].

— **Générateur C** La version en langage C de POLYORB_HI offre une génération des composants similaire à celle du langage Ada. En fait, dans cette version les sous programmes sont traduits en fonctions et les mêmes types de données que le langage Ada sont traités. Toutefois, ces deux générateurs ne sont pas totalement similaires. Le langage C est procédural, ainsi la traduction en C d'un modèle AADL ne fait pas intervenir des instructions orientées objet comme Ada. Aussi, le langage C n'introduit pas la notion de paquetage. Les sous-programmes AADL sont traduits en C par des fonctions dont le type de retour est `void`. Les accès aux composants de donnée ainsi qu'aux sous-programmes se font grâce aux pointeurs.

— **Générateur RTSJ** Un troisième générateur de code offert par Ocarina introduit le concept de la programmation orientée objet en utilisant le langage Java Temps réel nommé RTSJ. Ce générateur est à base d'un ensemble de règles de transformations des composants AADL en RTSJ.

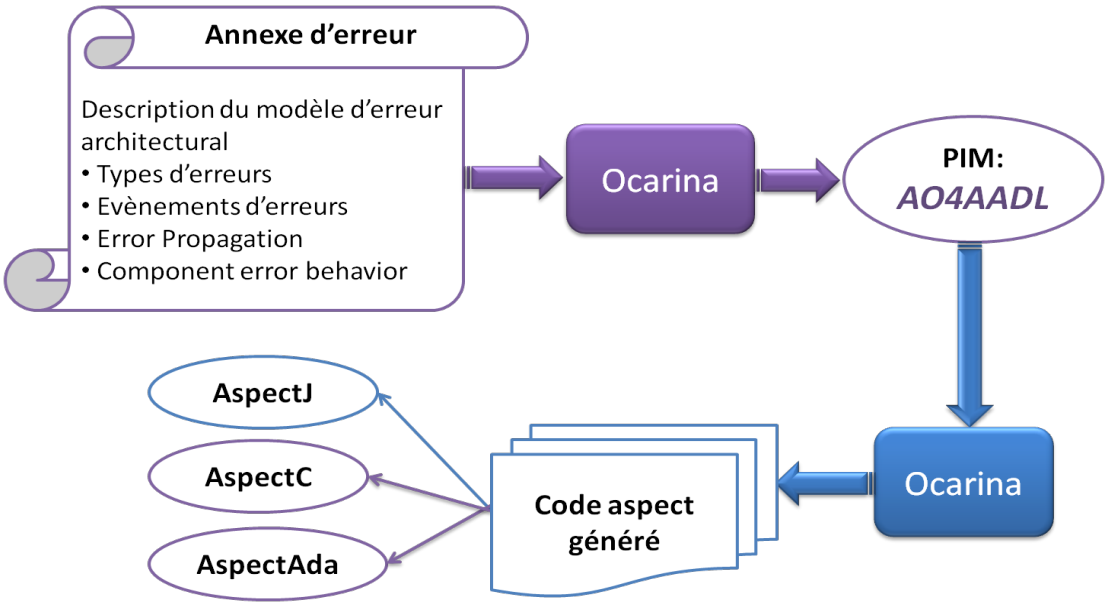

FIGURE 7.1 – Approche de génération de code

7.3.4 Génération du code aspect

En ce qui concerne les préoccupations transversales, et en particulier la tolérance aux pannes, nous utilisons la programmation orientée aspect pour garantir la séparation des préoccupations sur les deux niveaux conceptuel et exécutif. Puisque Ocarina offre plusieurs générateurs de code, comme les générateurs de code fonctionnel à partir des modèles AADL et un générateur d'aspect d'AO4AADL vers AspectJ, nous avons défini une méthode générique supportant plusieurs langages d'aspects comme ceux fonctionnels basée sur la démarche MDA. Nous générons à partir de l'annexe d'erreurs un modèle intermédiaire en AO4AADL, extension du langage AADL par des aspects architecturaux . Puis à partir du modèle AO4AADL, nous générons du code aspect en se basant sur des règles de génération que nous avons défini. Ainsi, nous offrons plus de liberté au concepteur pour choisir le langage d'aspect cible. Pour la génération du code de l'annexe d'erreur vers les langages d'aspects, nous proposons deux solutions. La première consiste à établir des règles de générations pour supporter les différents langages d'aspects et implanter les différents générateurs correspondants. Et l'autre, consiste à définir des règles génériques de transformation de EMA vers AO4AADL et donc développer uniquement le générateur de EMA vers AO4AADL. Nous adoptons la deuxième solution pour minimiser les coûts de développement et profiter des générateurs existants.

7.4 Génération du code tolérant aux pannes

Dans le but de séparer l'implantation de la tolérance aux pannes de celle des préoccupations métiers, nous avons défini une processus de génération séparée de code tolérant aux pannes comme le montre la figure 7.2.

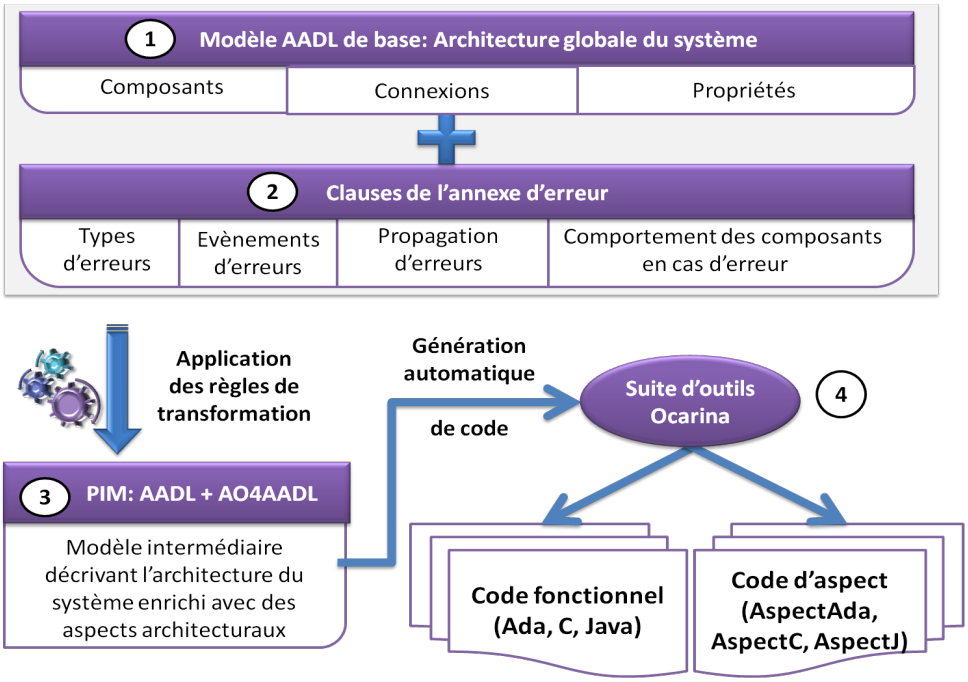

FIGURE 7.2 – Approche de génération de code

Cette séparation est basée sur le paradigme de développement orienté aspect. Cette génération, à partir des descriptions de l'annexe d'erreurs, va être par la suite intégrée automatiquement dans le code fonctionnel grâce à l'opération de tissage.

Afin de garantir une flexibilité pour les choix des langages de programmation, nous avons visé une approche générique pour la génération du code aspect tolérant aux pannes à partir des modèles d'erreurs décrits par l'annexe d'erreurs du langage AADL, EMA. Dans une première étape, nous optons à une transformation de modèle. Le modèle AADL de base décrivant l'architecture globale du système ne sera pas touché. En suivant la démarche MDA, nous générons un modèle indépendant de la plate-forme (PIM) décrit en AO4AADL. Ce modèle AADL intermédiaire, enrichi avec les aspects AO4AADL, permet d'intercepter les composants et les éléments d'interfaces qui sont enrichis par l'annexe d'erreurs pour décrire leurs comportements en cas d'occurrence de fautes. Ce modèle est généré automatiquement et d'une manière transparente à l'utilisateur.

Dans une deuxième étape, le modèle intermédiaire indépendant de la plate-forme sera raffiné pour avoir finalement un code d'aspect spécifique à la plate-forme choisie. A partir de

l'annexe d'erreur du langage AADL, nous générons des aspects en AO4AADL en utilisant Ocarina. Relativement à la plate-forme choisie pour la génération du code fonctionnel, un processus de génération de code à partir du modèle AADL et son annexe AO4AADL est établi dans le but de générer le code tolérant aux pannes en AspectJ, AspectAda ou AspectC pour être tissé au niveau du code fonctionnel généré respectivement en RTSJ, Ada ou C.

Ces deux étapes de génération sont pilotées par des règles de génération qui sont implantées au niveau de la suite d'outils Ocarina pour deux raisons : D'une part, cet outil offre des analyseurs pour les langages mis en question comme AADL et l'annexe AO4AADL. Pour chacun des langages, une partie frontale est implantée pour effectuer les différentes analyses et permet de générer un AST décoré pour en servir au niveau de la partie dorsale en cas de besoin. D'autre part, pour chacun de ces langages, il existe des générateurs de code implantés au niveau de cet outil. Pour le langage AADL, nous venons de décrire les différents générateurs de code dédiés pour les applications temps-réel embarqués critiques au niveau de la section 7.3.3. En ce qui concerne l'annexe AO4AADL, un générateur de code vers le langage d'aspect AspectJ a été conçu et développé dans le cadre des travaux de Loukil et al [LKZJ13]. De ce fait, il nous reste à développer la partie frontale dédiée pour l'annexe d'erreurs et la partie dorsale pour la génération du code aspect à partir des descriptions AO4AADL en AspectAda et AspectC.

7.4.1 Génération de code de l'annexe EMA vers AO4AADL

La première étape pour la génération du code tolérant aux pannes consiste à générer un modèle indépendant de la plate-forme décrit en AADL et son annexe AO4AADL. Ce modèle joue le rôle d'un modèle intermédiaire afin d'intercepter le modèle AADL de base pour détecter et tolérer des erreurs dont l'occurrence est prévue à l'avance. Les types d'erreurs prévues ainsi que les techniques de détection sont décrites à travers l'annexe d'erreurs. Pour cela, nous avons défini un ensemble de règles de génération de l'annexe d'erreur vers le langage AO4AADL, considéré comme langage pivot 7.1. La définition des règles de génération ne couvre pas tous les concepts présents dans l'annexe d'erreurs. Nous avons plutôt sélectionné un sous-ensemble des clauses de l'annexe d'erreurs servant à appliquer une tolérance aux pannes attendues. Nous avons mis l'accent sur les parties de l'annexe EMA visant à définir les différents types et ensembles d'erreurs prévues, les composants mis en jeu pour la détection de ces erreurs et les comportements spécifiés de ces composants.

Dans la section suivante, nous détaillons un sous-ensemble des règles établies pour la génération des aspects architecturaux à partir des clauses de l'annexe EMA qui seront greffés au niveau du modèle AADL correspondant.

7.4.2 Règles de génération

Dans cette section, nous présentons un ensemble de règles régissant le processus de transformation du modèle AADL et son annexe d'erreurs vers un modèle AADL enrichi avec l'annexe AO4AADL. Comme déjà indiqué, ces règles de transformations couvrent uniquement un sous ensemble de l'annexe d'erreurs. Nous avons sélectionné les concepts de base visant la tolérance aux pannes et nous avons omis d'autres concepts reliés à la sûreté de fonctionnementd'une manière générale.

Définition des types d'erreur Pour distinguer les types, l'hiérarchie et l'ensemble d'erreurs définies au niveau de l'annexe EMA, nous avons défini un nouveau type `error_type` de type string. Le tableau 7.1 montre la règle de transformation correspondante. Une variable de type `error_type` indique un type de faute activé, une erreur propagée entre des composants AADL, ou bien une erreur détectée considérée comme un déclencheur d'une transition de l'état d'un composant AADL particulier dans le but de décrire son comportement.
Pour chaque type d'erreur déclarée au niveau de l'annexe EMA, une variable `error_type` est instanciée contenant le même identifiant et admettant comme valeur la conversion de son identifiant en un string. Selon l'annexe d'erreurs, des types d'erreurs peuvent être organisés au sein d'une hiérarchie de type ou sous forme d'un ensemble. Pour rester fidèles à ces définitions, nous avons supporté les deux organisations comme l'indique la règle. L'hiérarchie est décrite à travers la concaténation de l'ensemble des niveaux d'hiérarchie séparé par " : :". La collecte d'un ensemble d'erreur se traduit par un tableau d'erreur dans chaque valeur est de type `error_type`.

TABLE 7.1 – Règle de transformations des types, ensemble et hiérarchie d'erreurs

Spécification EMA :
error_types
defining_error_type_identifier : **type** [**extends** error_type_reference]
defining_error_type_set_identifier : **type set** error_type_set
Spécification AO4AADL :
type error_types **is new string**
defining_error_type_identifier : **error_types** := [error_type_reference : :]defining_error_type_identifier
defining_error_type_set_identifier : **array of error_type** := {error_type_reference}*

Selon le type d'erreur, s'il s'agit tout simplement d'un identifiant ou d'un type déjà défini au niveau de l'annexe d'erreurs standardisée, le traitement qui lui est accordé sera par la suite analysé.

Propagation des erreurs La propagation d'une erreur, définie au niveau de l'annexe d'erreurs, spécifie que ce type d'erreur peut être propagée à travers l'intérieur ou vers l'extérieur

d'un composant par le biais de ces éléments d'interfaces.

Afin de considérer les types d'erreurs qui peuvent être propagées avec les données, nous proposons d'étendre chaque type de données déclaré dans le modèle AADL de base par deux propriétés. La première, appelée *Possible_Errors*, sert à indiquer les erreurs possibles qui peuvent être détectées à ce niveau. La seconde, appelée *Detected_Errors*, permet de fixer la liste des erreurs activées (détectées). Ainsi, nous serons en mesure de vérifier à tout moment la liste des erreurs attendues et la liste de celles détectées propagées à travers les points de propagation.

Un point de propagation devrait être intercepté non seulement au niveau architectural mais aussi au niveau du code dans le but de détecter une telle propagation. Ainsi, un point de propagation doit être transformé sous forme d'une déclaration de joinpoint. Selon le cas, le point de propagation est considéré comme un point de propagation *in*, *out* ou *inout*. Ceci dépend, d'une part, du type d'interface déclaré au niveau du modèle AADL et, d'autre part, de sa déclaration en tant que point de propagation d'une erreur donnée au niveau de l'annexe EMA. Ainsi, nous devons intercepter ou bien la valeur d'entrée reçue par un port *in* ou bien la valeur de sortie envoyée par un port *out*. Si le port est de type *inout*, nous devons alors intercepter les deux valeurs. Pour chacune des situations citées, nous avons défini une règle correspondante :

— **Propagation *in***

D'après Loukil et al [LKZJ13], lors de l'interception d'un port de type *in*, la primitive *call* sert à intercepter l'appel au sous-programme responsable sur la réception des données. A travers cette primitive, on est en train de lire la valeur de donnée qui va être lue par le port juste avant l'exécution. Ainsi le type de joinpoint généré est *call inport* comme l'indique la règle 7.2.

TABLE 7.2 – Règle de transformation d'une propagation de type *in*

Spécification EMA :
error_propagation_point : **in propagation** error_type_set
error sink (incoming_error_propagation_point)[error_type_set]
Spécification AO4AADL :
pointcut in_ep() : **call inport** (error_propagation_point (..))
advice after : in_ep () : error_propagation_point ? possible_errors ;
if (substring (error_type_set,possible_errors) !=0) {detected_errors := error_type_set ;}

Comme la spécification d'un aspect AO4AADL comportemental est décrite en deux parties, pointcut et advice, chacune de ces parties doit être générée. Les constructions devront être exécutées, désignées par l'advice, consistent à exécuter une opération de communication qui s'agit à la lecture des données à partir du port d'entrée concerné et puis de vérifier si l'erreur prévue est détectée. La détection est faite par le moyen d'une structure conditionnelle *if* comparant la valeur interceptée lue et la liste des erreurs

prévues à travers la constante *Possible_Errors*. Si l'erreur est bien détectée, alors une mise à jour de la liste des erreurs détectées, *Detected_Errors*, est effectuée. La détection ne peut se faire qu'après la réception des données via le port ce qui exige que l'advice généré doit être de type *After*.

— **Propagation *out***

Par analogie à la propagation de type *in*, la propagation de type *out* correspond à une interception d'un port *out* avec la primitive *execution* permet d'intercepter l'exécution réelle du sous-programme responsable sur la transmission des données sur ce port (Voir règle 7.3) déclarée au niveau du pointcut.

TABLE 7.3 – Règle de transformation d'une propagation de type *out*

Spécification EMA :
error_propagation_point : **out propagation** error_type_set
error source (outgoing_error_propagation_point)[effect_error_type_set]
[**when** fault_source] [**if** fault_condition]
Spécification AO4AADL :
pointcut out_ep () : **call outport** (error_propagation_point (..))
advice before : out_ep () :
variables{possible_errors,detected_errors : String_Type ;}
initially{possible_errors :="" ;detected_errors :=""} ;
possible_errors + := error_type_set_identifier ;
if (fault_condition **or** fault_source) {detected_errors + := effect_error_type_set ;}
error_propagation_point ! possible_errors
error_propagation_point ! detected_errors

Pour éviter la propagation des fautes, la détection doit se faire avant la transmission des données via le port ce qui impose que l'advice généré doit être de type *Before*. En ce qui concerne la spécification de l'advice, nous souhaitons à ce niveau écrire le contenu des données reçues sur le port de sortie après avoir vérifié si une erreur prévue est détectée. La liste *Possible_Errors* est mise à jour en prenant en considération les erreurs prévues. Par contre, la liste *Detected_Errors* est mise à jour que s'il y a détection d'une erreur prévue. Les deux listes sont par la suite enregistrées comme propriétés des données reçues pour en servir lors du génération du code afin d'appliquer les mécanismes de recouvrement nécessaires.

— **Propagation *inout***

Outre les points de propagations des erreurs, l'annexe d'erreurs définit les concepts de flux d'erreur. Ces flux servent à déclarer le rôle d'un composant AADL dans la propagation des erreurs. Un tel composant peut agir comme une source, un puit ou un point de passage ou d'intercommunication *path* lors d'une propagation d'erreur *out*, *in* ou *in out* respectivement. Ces flux considèrent la propagation d'un seul type d'erreur ou de plusieurs avec l'application d'une transformation possible des types d'erreur.

Pour cette raison, nous interceptons tous les types de ports, *in*, *out* et *in out*.

Pour détecter une telle propagation *in out* responsable de communiquer une erreur entre deux ou plusieurs composants AADL, nous interceptons l'appel au sous-programme responsable à la fois de la réception et l'envoie des données sur le port concerné. Ainsi, le type de l'advice généré sera *Around* pour pouvoir intercepter les ports avant et après l'exécution de la réception et/ou la transmission des données.

TABLE 7.4 – Transformation rule of *path* error propagation

EMA specification :
error path (incoming_error_propagation_point)[error_type_set]
→ (outgoing_error_propagation_point)[error_type_set]
AO4AADL Specification :
pointcut path_ep () : **call inoutport** (incoming_error_propagation_point (..))
advice around : path_ep () :
incoming_error_propagation_point ? detected_errors
if (substring (detected_errors,error_type_set_identifier) != 0)
{outgoing_error_propagation_point ! detected_errors ;}

Comportement des composants Les spécifications du comportement du composant en présence d'erreur sont décrites à l'aide d'une machine à état. Différents états sont déclarés pour prendre en considération la détection ou non d'une erreur. La transition d'un état à un autre est déclenchée par des événements. Il s'agit de évènements d'erreur (error event), de recouvrement (recover event) ou de réparation (repair event), ou par un flux entrant de propagation de l'erreur et les conditions de propagation de l'erreur en sortie. Le comportement du composant détaille aussi le processus de détection d'erreur basées sur l'évaluation logique des conditions. Pour cela la traduction des comportements de ces composants est basée sur les règles de transformations établies pour les différents types de propagation. Les machines à états doivent être considérées lors de la génération du code AO4AADL pour en prendre en compte les actions de reconfigurations à des fins de tolérance aux pannes.

7.4.3 Génération du AO4AADL vers un langage d'aspect

La génération du code du modèle AADL enrichi avec les aspects architecturaux a été développée auparavant dans le cadre des travaux de Loukil et al [LKZJ13]. Des règles de génération ont été établies puis implantées pour générer du code aspect AspectJ. Toutefois, les règles fournies au niveau du travail [Lou10] sont décrites d'une manière générique pouvant être appliquées avec d'autres langages d'aspects. Pour cette raison, nous nous sommes inspirés de l'ensemble de ces règles pour générer des spécifications AspectAda à partir du langage AO4AADL.

7.5 Conclusion

Dans ce chapitre, nous avons présenté notre approche proposée pour la génération du code tolérant aux pannes séparément à partir des modèles AADL et de son annexe d'erreurs. Notre approche a été définie d'une manière générique pour pouvoir l'appliquer avec différentes plate-formes. À la base de la démarche MDA, un modèle intermédiaire est généré à partir du modèle AADL initial et son extension par l'annexe d'erreurs. Le modèle intermédiaire généré est un modèle AADL enrichi par des aspects architecturaux en utilisant l'annexe AO4AADL. Pour implanter ce générateur de code, nous avons défini un ensemble de règles qui consistent à l'interception au niveau architectural des différents ports visant la détection de pannes. A partir de ce modèle, le code tolérant aux pannes sera généré en un langage d'aspect spécifique pour être conforme au langage dans lequel le code fonctionnel est généré. Dans ce cadre, nous avons défini un ensemble de règles de génération du langage AO4AADL vers le langage d'aspect AspectAda en s'inspirant des règles définies pour la génération des aspects décrites en AspectJ dans des travaux existants.

Dans le but d'adapter le langage AspectAda pour le développement des systèmes temps-réel tolérants aux pannes, nous présentons dans le chapitre suivant une étude du langage existant et des solutions pour son extension et son adaptation.

8

Adaptation d'un langage d'aspect pour le temps-réel

8.1 Introduction

Dans le chapitre précédent, nous avons présenté notre approche proposée pour la génération de code à partir des modèles AADL et ses annexes. Nous nous intéressons dans notre contexte aux systèmes temps-réel critiques soumis à des contraintes temporelles et de ressources particulières et des exigences de dynamisme spécifiques. Pour cela, ces systèmes exigent des restrictions interdisant l'utilisation des constructions jugées compromettantes à savoir l'allocation dynamique de mémoire, les nombres à virgules flottantes et l'aiguillage dynamique. Par conséquent, nous avons conclu que le code produit doit être écrit avec un langage précis, rigoureux et garantissant le respect de ces restrictions afin de respecter les contraintes temps-réel. Nous avons aussi conclu que Ada peut être considéré comme excellent candidat pour développer ce genre de systèmes pour les raisons expliquées dans la section 7.2. Ada a été pour cela choisi comme le langage adéquat pour la production du code fonctionnel à partir des modèles AADL. Quant à la génération du code tolérant aux pannes, le langage AspectAda a été sélectionné pour des raisons de conformité. Toutefois, en explorant la littérature (Voir section 2.4), ce langage n'a été nulle part testé pour le domaine temps-réel critique. Ce chapitre est donc consacré pour l'étude et le critique du langage existant vis-à-vis de son adéquation pour le développement temps-réel et les besoins d'adaptation.

Dans la section 8.2, nous présentons l'étude du langage AspectAda existant. Nous nous intéressons particulièrement à sa syntaxe, sa sémantique, ses règles de tissage et de génération de code et enfin à l'implantation de son compilateur. Cette étude a montré certaines lacunes relativement par rapport aux besoins en tolérance aux pannes et des constructions

pouvant provoquer la violation des contraintes temps-réel. Nous proposons des solutions à ses problèmes au niveau de la section 8.3. Finalement, nous donnons une vue d'ensemble sur l'architecture et l'implantation du nouveau compilateur dans la section 8.4.

8.2 AspectAda : Étude de l'existant

AspectAda [PC05], est une extension d'aspect pour le langage Ada95, proposé dans le cadre d'un projet de recherche débuté en 2003. La première version de ce langage a été publiée en 2005 par son équipe productrice. Inspiré du langage AspectJ, extension du langage Java par les aspects, AspectAda contient toutes les constructions de Ada95 en adoptant des concepts spécifiques à la POA à savoir : les aspects, les *advices*, les *joinpoints* et les *pointcuts* tout en traduisant tous ces concepts à des notions plus proches au langage Ada.

Deux nouvelles unités de programmes sont ajoutées au langage Ada pour définir les aspects :

— ***Aspect*** : paramétrée par au moins un *pointcut* et mélangée avec du code Ada, cette unité de programme contient les types des aspects et les codes des *advices*.

— ***Weaver*** : pour associer les *pointcuts* aux aspects et *advices* correspondants, cette unité de programme permet d'instancier les aspects après avoir défini les pointcuts permettant d'intercepter un ensemble de *joinpoint*.

Fondé sur ces deux concepts, le langage AspectAda a une syntaxe particulière définie par sa grammaire. Suite à une étude approfondie de cette grammaire et de la version publiée du langage, nous avons constaté qu'elle présente plusieurs défauts non seulement au niveau de sa syntaxe et sa sémantique, mais aussi au niveau de l'opération de tissage et de génération de code et du compilateur développé. Cette section présente une étude critique de l'existant relativement par rapport au respect des contraintes temps-réel et aux besoins de la tolérance aux pannes. Plus de détails techniques sur l'étude de ce langage ainsi que des exemples illustratifs sont données dans [GBZ13].

8.2.1 Syntaxe et sémantique du langage AspectAda

Joinpoints

Un joinpoint distingue des moments bien définis de l'exécution de l'application ou le code de l'advice sera inséré. Selon [PC05], le langage AspectAda supporte les joinpoints interceptant l'exécution ou l'appel d'une entrée ou d'un sous-programme, l'élaboration d'un paquetage ou l'initialisation et la finalisation de types contrôlés (*controlled types*). Toutefois, selon la grammaire, les joinpoints servent uniquement à intercepter les exécutions des sous-programmes d'un paquetage donné comme le montre le listing 8.1. Par ailleurs, l'interception de l'appel et de l'exécution des sous-programmes ou des entrées est nécessaire pour la

détection des propagations des erreurs `in` et `out` entre les composants comme déjà expliqué au niveau des règles de génération décrites dans la section 7.4.2. Ainsi, comparé au langage AspectJ, le modèle des *joinpoints* offert par le langage AspectAda ne suffit pas pour couvrir la description et la détection de la propagation des erreurs entre les composants au niveau du code généré.

Listing 8.1 – Règle de grammaire décrivant l'expression du pointcut

```
1  basic_pointcut_expr ::= (pointcut_expr)
2          | execution (method_pattern)
3          | identifier
```

Il est à noter ici que le listing 8.1 montre aussi que la grammaire de AspectAda est aussi influencée par celle du langage AspectJ. Ceci est déduit à partir de l'attribut *method_pattern* qu'elle déclare malgré l'absence de la notion de méthode dans le langage Ada. Cette grammaire pourra être plutôt plus spécifique au langage Ada en supportant l'appel ou l'exécution des entrées sur les objets protégés (*protected objects*) et sur les tâches (*tasks*) ou bien la création des tâches qui sont des concepts fondamentaux des systèmes temps-réel critiques. Pour ces raisons, nous proposons des extensions pour le modèle de joinpoint détaillées dans la section 8.3.1.

Pointcuts

Un ensemble de pointcuts encapsulés dans une même unité appelée *weaver*, permet de définir une collection de joinpoints à travers son langage d'expression particulier. Ce dernier offre une syntaxe simple pour spécifier les patrons de recherche visant le filtrage des identifiants et des types de paramètres des sous-programmes et des paquetages. Afin de rassembler un groupe de *joinpoints*, un *pointcut* fait appel à la quantification permettant ainsi à un aspect d'intercepter divers points dans le programme. Cette quantification introduit les wildcards (*, +, ..) et les opérateurs logiques (*and*, *or*, *xor* et *not*) comme l'indique le listing 8.2.

Listing 8.2 – Règles des patrons d'expressions des types des paramètres

```
1  ...
2  type_pattern_expr ::= or_type_pattern_expr
3          | type_pattern_expr and or_type_pattern_expr
4
5  or_type_pattern_expr ::= unary_type_pattern_expr
6          | or_type_pattern_expr or unary_type_pattern_expr
7          | or_type_pattern_expr xor unary_type_pattern_expr
8
9  unary_type_pattern_expr ::= basic_type_pattern
10         |  not unary_type_pattern_expr
11 ...
```

Suite à une étude de la grammaire exprimant les *pointcuts*, nous avons déduit qu'elle est fortement influencée par le langage d'expression des *pointcuts* d'AspectJ ce qui engendre une contradiction entre les deux langages Ada et son extension par les aspects, AspectAda.

D'une part, tous les sous-programmes sont considérés par AspectAda comme des méthodes Java. Toutefois, Ada supporte non seulement les fonctions mais aussi les procédures et les entrées, dont la catégorie devrait être spécifiée au niveau du *joinpoint*. En outre, il y a certaines règles de la grammaire qui prouvent qu'elle est définie d'une manière douteuse. D'après les lignes 2 et 3 du listing 8.2, un paramètre d'un sous-programme peut être à la fois du type T1 et T2 spécifié en utilisant l'opérateur logique *and* ce qui est logiquement contestable. De plus, cette grammaire néglige certaines constructions jugées très utiles (d'après les autres langages d'aspect, en particulier AspectJ) dans les expressions des *pointcuts* pour la recherche des *joinpoints*. Nous citons à titre d'exemple les modes des paramètres (*in*, *out*, ou *inout*) et les types de retour d'une fonction. Dans notre contexte, nous nous intéressons plus particulièrement, dans le contexte de la détection et la propagation des pannes, au modes des paramètres influant les chemins de propagation d'une erreur et donc sa détection. Nous envisageons dans notre solution, proposer des extensions pour ces patrons afin de supporter ces constructions manquantes.

Advice *around* et instruction *proceed*

Pour associer un advice à un pointcut dépendamment du son moment d'exécution, trois types d'advices sont définis.

1. Advice *before* exécuté avant le pointcut.
2. Advice *after* exécuté après le pointcut.
3. Advice *around* exécuté à la place d'un pointcut en offrant à ce dernier la possibilité de reprendre son exécution à travers l'instruction *proceed*.

Un advice around peut remplacer une fonction ou une procédure Ada. Mais, selon la syntaxe du langage AspectAda, ce type d'advice permet de remplacer uniquement une procédure. Pour remédier à ce problème, il faut supporter le retour d'une valeur (par le moyen d'une construction *return*) au niveau de la syntaxe d'un advice around.

De plus, selon [PC05], l'instruction *proceed* peut figurer dans un advice around afin d'exécuter le comportement du *joinpoint* courant similairement au langage AspectJ. En contre partie, cette instruction ne figure pas au niveau des règles de la grammaire et elle n'est même pas considérée comme un mot clefs du langage ni traitée lors de l'opération de tissage. Elle est considérée plutôt comme un identifiant d'un sous-programme et donc généré au niveau du code tissé telle qu'elle est. Toutefois, pour la détection d'une propagation à travers un port de type `inout`, nous avons défini un aspect interceptant le port `inout` et un advice de type `around` pour gérer les actions de recouvrement en cas de détection de pannes. Pour

cette raison, il est préférable de supporter le type `around` et son instruction `proceed` pour en se servir en cas de besoin.

8.2.2 Étude du compilateur/tisseur

L'équipe productrice du langage AspectAda 1.0 a développé un compilateur/tisseur prototype pour le tissage et la génération du code Ada tissé [PC05]. Elle a proposé une architecture du compilateur et a utilisé différents outils pour son implantation. Dans cette section, nous commençons par décrire et discuter l'architecture proposée ainsi que les outils utilisés. Ensuite, nous citons les problèmes relevés au niveau de l'opération de tissage et de génération de code après avoir testé le compilateur AspectAda prototype.

Architecture du compilateur prototype

L'architecture proposée pour le compilateur prototype du langage AspectAda (voir figure 8.1) est divisée en deux parties : un tisseur (*weaver*) et une *Runtime*. La partie *weaver* admet trois types d'entrées correspondantes chacune à un langage indépendant : le code fonctionnel en Ada, le code des aspects définissant les advices et le code des règles de composition (*weaver code*) traduisant la grammaire des pointcuts. La sortie de cette partie du compilateur est un code Ada tissé. La deuxième partie appelée *Runtime*, est une bibliothèque de fonctions offertes pour être utilisée dans le code des advices ainsi que dans le code généré après le tissage dans le but d'accéder à des informations sur le joinpoint courant. Chacune des entrées de l'unité *weaver* est analysé syntaxiquement par un parseur spécifique donnant ainsi lieu à la génération de trois arbres syntaxiques abstraits (AST) différents. Ces arbres sont par la suite traversés et interrogés par l'unité weaver pour la sélection (*selection*) des joinpoints et le tissage (*merging*). Par la suite, la compilation, l'assemblage (*binding*) et l'édition des liens (*linking*) du code Ada généré sont effectués par le compilateur standard d'Ada GNAT (*The GNU Ada Compiler*) pour générer un exécutable.

Suite à l'étude de l'architecture du compilateur/tisseur AspectAda, nous avons extrait certaines limites liées à la bibliothèque Runtime et au tisseur.

La Runtime du langage AspectAda est réduite. Elle permet l'accès à peu d'informations sur les *joinpoints* telles que le nom du joinpoint et les noms, les types et les modes des arguments. Cependant, certaines informations qui sont très nécessaires dans la spécification des advices telles que les valeurs des paramètres, la catégorie du *joinpoint* (*procedure*, *function* ou *entry*) et le type et la valeur retournée si le *joinpoint* est une fonction ont été négligées. Néanmoins, dans le contexte des systèmes tolérants aux pannes, nous avons besoins d'évaluer le types de retour d'une fonction ainsi que la valeur retournée pour l'évaluation logique des conditions dans le but de détection des erreurs. Pour cela, nous envisageons étendre la Runtime pour exporter plus d'informations.

Dans la section suivante, nous discutons les opérations de tissage et de génération de

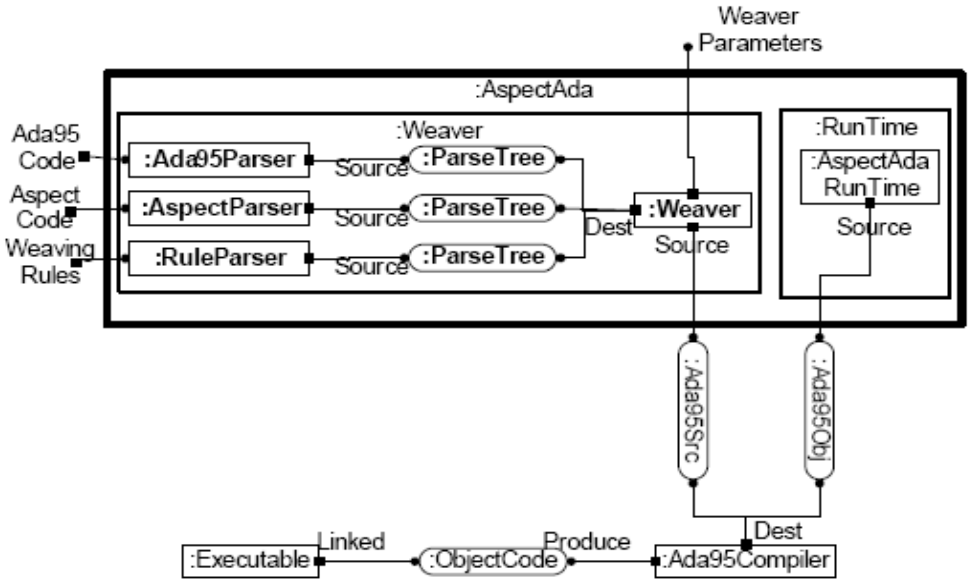

FIGURE 8.1 – Architecture du compilateur prototype [PC05]

code qui présentent à leurs tours certains défauts spécifiquement face à la spécificité des systèmes temps-réel.

Tissage et génération de code

Pour tout langage d'aspect, l'unité de programme tisseur (*weaver*) joue un rôle très important dans la définition des *pointcuts* et l'instanciation des aspects. Cette unité contient toutes les règles régissant la composition des aspects avec les *pointcuts*. Pour AspectAda, nous avons constaté l'absence de telles règles décrivant d'une manière pertinente et détaillée l'opération de tissage. Les auteurs [PC05] ont juste mentionné que le code des advices est inséré directement dans le code du *joinpoint*. Pour comprendre en détails le déroulement de l'opération de tissage et découvrir le fonctionnement interne du compilateur, nous avons étudié et analysé ce compilateur allant de plusieurs exemples pour en évaluer les résultats et analyser le code généré. Malheureusement, nous avons relevé plusieurs erreurs et défauts au niveau de l'opération de tissage. En effet, le tisseur effectue une opération simple de tissage. Il s'agit d'une insertion directe du code de l'advice dans le code du *joinpoint* en utilisant des blocs *declare* du langage Ada. Le code des advices est inséré entre *begin* et *end* du sous-programme intercepté. Toutefois, l'emplacement exacte de l'insertion (*call* ou *execution*) du code de l'advice dans celui du joinpoint doit être précisé selon le type du joinpoint et le type de l'advice (*before*, *after* ou *around*). Dans le but garantir le bon ordre d'exécution, le code du sous-programme intercepté ainsi que le code des advices sont insérés dans des blocs *declare*. Néanmoins, l'application de cette solution dans le domaine temps-réel critique est interdite pour deux raisons :

— L'utilisation des blocs *declare* est nuisible à la sûreté de fonctionnement de l'application et en particulier à son déterminisme. Ceci revient au faite que dans tel bloc, les données déclarées entre *declare* et *begin* sont allouées dynamiquement à l'exécution et automatiquement détruites à la fin du bloc.

— Toute modification faite dans le corps d'un advice de l'aspect engendre la recompilation de tout le code source de l'application.

8.2.3 Synthèse

L'étude du langage AspectAda tel qu'il était publié en 2005 a montré qu'il présente plusieurs lacunes non seulement au niveau de la syntaxe et la sémantique mais aussi au niveau du compilateur et les deux opérations de tissage et de génération de code. Ce langage, tel qu'il est publié ne peut être utilisé comme langage d'aspect pour les systèmes temps-réel tolérants aux pannes. D'une part, il contient des constructions violant les contraintes temps-réel essentiellement le déterminisme à cause de l'allocation dynamique des ressources et d'autre part ne répond pas aux besoins de la tolérance aux pannes.

Afin de remédier aux problèmes déjà cités suite à l'étude du langage AspectAda existant, nous envisageons deux solutions. La première consiste à rectifier tous ces problèmes de syntaxe et de sémantique et de maintenir le compilateur existant. Tandis que la seconde consiste à définir un nouveau langage et à développer le compilateur correspondant à partir de zéro.

A cause du manque de documentation et du code non structuré, il était très dur de maintenir le compilateur existant. C'est pour cela, que nous avons choisi la réécriture de la grammaire pour y intégrer des rectifications pour les problèmes détectés relativement par rapport à nos besoins et le développement d'un nouveau compilateur.

Dans la section suivante, nous présentons les solutions proposées pour l'enrichissement de la grammaire en traitant chacun des problèmes présentés.

8.3 Extension et adaptation du langage AspectAda

Afin de remédier aux problèmes déjà cités suite à l'étude du langage AspectAda existant, nous avons proposé diverses solutions pour chacun de ces problèmes.

8.3.1 Enrichissement du modèle des joinpoints

Dans la section 8.2, nous avons extrait un certain nombre de problèmes liés aux modèles des *joinpoints* et aux expressions des *pointcuts*. Dans cette section, nous proposons les solutions et les extensions pour améliorer chacun d'eux.

La première amélioration concerne l'ajout de la primitive *call* qui ne figurait pas au niveau des expressions des *pointcuts* de AspectAda. Nous supportons, non seulement l'exécution des *joinpoints* mais aussi leurs appels.

Le nouveau modèle des *joinpoints* que nous proposons permet aussi de distinguer entre les fonctions, les procédures et les entrées (*entry*), contrairement à l'ancien modèle qui les considère tous comme des méthodes (*method_pattern*) similairement au langage AspectJ. De plus, nous étendons le modèle des *joinpoints* de AspectAda pour définir un type de retour d'une fonction lorsqu'il s'agit d'intercepter une ou plusieurs fonctions. En outre, notre nouveau modèle permet également de spécifier les modes des arguments des *joinpoints* en plus de leurs types.

8.3.2 Extension de la Runtime

Dans la version publiée d'AspectAda, les informations de joinpoint auxquelles on peut accéder sont seulement le type, le nom et le mode de ces arguments. Ces informations sont accessibles grâce aux fonctions offertes par la Runtime. Néanmoins, les fonctions offertes ne permettent pas l'accès en lecture ou en écriture dans le code d'un advice donné, des valeurs des arguments d'un joinpoint auquel il est associé. Pour rendre possible cet accès, nous avons étendu la Runtime du langage AspectAda par certaines fonctions.

Après avoir étendu la *Runtime* pour supporter plus de fonctionnalités, nous présentons dans cette section les améliorations suggérées dans le but de manipuler la valeur d'un argument d'un joinpoint donné selon son type. En effet, pour rendre accessible la lecture des valeurs des données échangées via les ports d'entrée/sortie dans le but de détection des erreurs, nous avons besoin de récupérer le type et la valeur des arguments de joinpoint. Pour cette raison, nous avons défini dans la *Runtime* les deux types *Value_Holder* et *Value_Access*. *Value_Holder* est un type générique n'ayant aucune relation avec le type de n'importe quel argument. Pour établir une telle relation, nous avons fait appel à la conversion de type. La solution proposée évite tous type de polymorphisme de la technologie orientée objet visant le déterminisme.

Plus de détails concernant la définition de ces types et des exemples illustratifs pour chacune des solutions proposées sont présentées dans [Bou12].

8.3.3 Règles de transformation et de tissage

Le langage Ada est étendu avec de nouvelles constructions permettant de supporter les aspects ce qui donne naissance au langage AspectAda. Par conséquent, la transformation des éléments lexicaux de AspectAda tels que les identifiants (*identifiers*), les littéraux (*literals*), les constantes (*constants*), les opérateurs (*operators*) et la transformation des types des données (*types*) donne lieu à des éléments lexicaux et des types correspondants en Ada.

Dans la version publiée du langage AspectAda, l'opération de tissage est effectuée manuellement sans se contenter de règles bien définies et testées. Pour cette raison, nous avons établi les règles de tissage et de transformation de code [9] de AspectAda vers un code Ada.

9. Plus de détails sur ces règles et des exemples illustratifs pour chacune d'elles sont données dans [Bou12]

Puisque nous visons adapter ce langage pour le développement des système temps-réel, ces règles devront obéir aux contraintes des systèmes temps-réel pour ne pas dégrader leurs fonctionnements.

Règles de transformation des advices et des joinpoints

À Chaque advice défini dans le code de l'aspect correspond un pointcut défini dans le code du weaver. Un pointcut possède une expression via laquelle il sélectionne un ensemble de joinpoints. Ces derniers peuvent être des sous-programmes (des procédures et ou des fonctions) ou des entrées. La nouvelle version de AspectAda supporte deux types de joinpoints : execution joinpoints et call joinpoints.

Nous différencions deux catégories d'advice :

— Les advices dépendants des joinpoints : le comportement de certains advices requiert l'accès aux informations du joinpoint (son nom, sa catégorie, ses arguments, etc.) à travers des fonctions fournies par la *Runtime* de AspectAda. Dans ce cas , l'advice prend en paramètre le joinpoint de type *Join_Point* défini dans le paquetage *AspectAda_Types*.

— Les advices indépendants des joinpoints : d'autres advices ne nécessitent pas l'accès aux informations des joinpoints. Dans ce cas, l'advice sera sans paramètre.

La transformation et le tissage des advices dans le code métier dépendent des types des joinpoints (*execution* ou *call*), de leurs catégories (*procedure*, *function* ou *entry*), du type de l'advice lui-même (*Before*, *After* et *Around*) et de son catégorie (dépendant ou indépendant des joinpoints).

Nous avons élaboré un ensemble de règles de tissage et de génération de code en traitant les combinaisons possibles entre les types des joinpoints et les types des advices. L'élaboration de ces règles est basée sur les deux critères suivants : La garantie du bon ordre d'exécution des advices et du code métier ; et l'obtention d'un code tissé optimal, déterministe et avec un overhead optimal.

Toutes les solutions présentées pour l'amélioration de la syntaxe, la sémantique et les opérations de tissage et de génération de code du langage AspectAda ont été prises en compte lors du développement du nouveau compilateur/tisseur. L'architecture que nous avons proposée pour ce compilateur ainsi que les étapes de programmation pour sa mise en œuvre sont détaillées dans la section suivante.

8.4 Nouveau Compilateur

La construction des compilateurs est un sujet délicat et complexe au cœur de l'informatique. La nouvelle architecture du compilateur AspectAda est très semblable à l'architecture des compilateurs modernes [WM95].

8.4.1 Nouvelle architecture du compilateur AspectAda

Comme le montre la figure 8.2, la nouvelle architecture renferme trois parties principales : une collection de parties frontales, une collection de parties dorsales et une bibliothèque centrale.

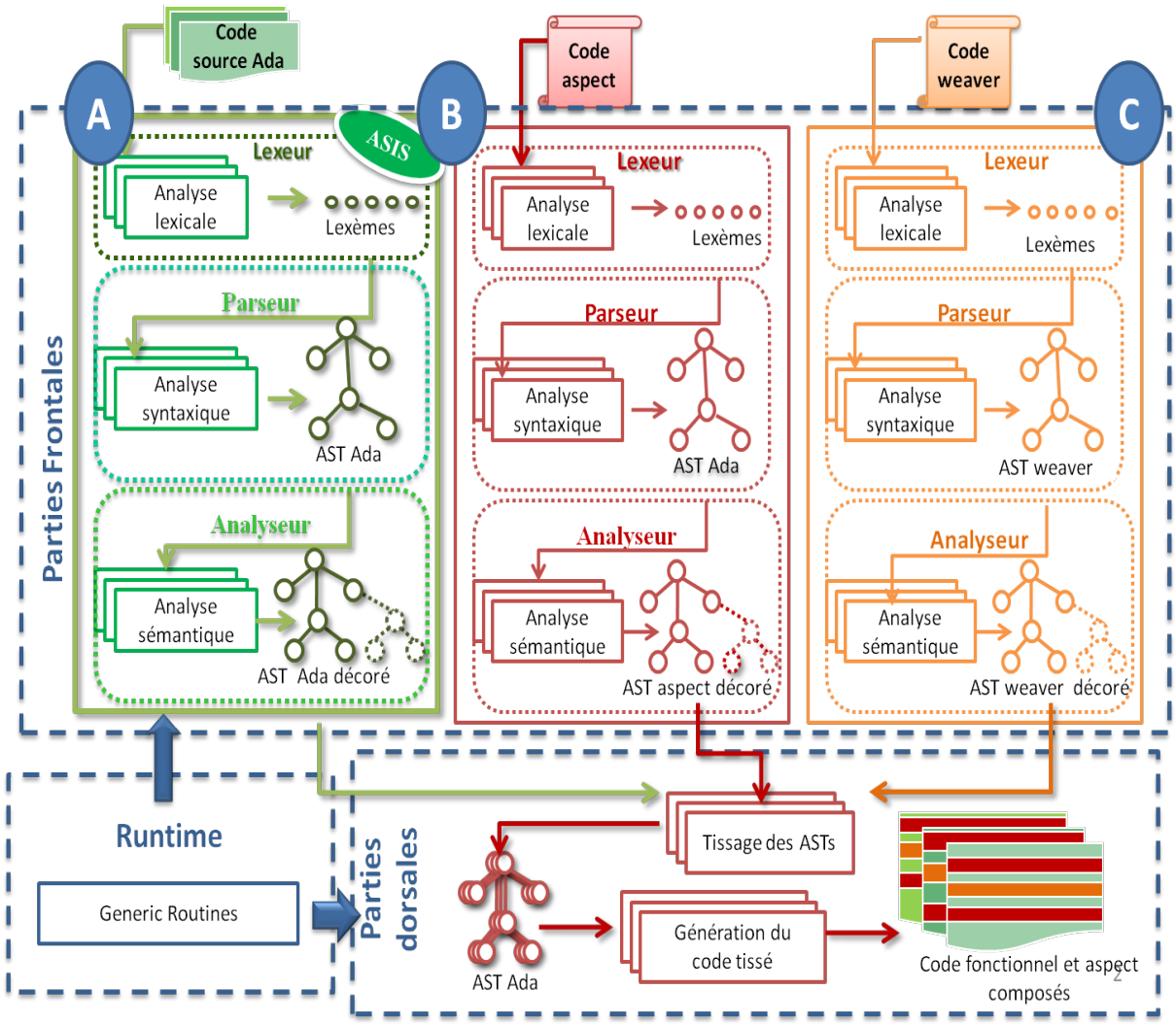

FIGURE 8.2 – Nouvelle architecture proposée pour le compilateur AspectAda

Bibliothèque centrale La bibliothèque centrale définit un ensemble de fonctions utiles pour la construction, le parcours et la manipulation des nœuds des arbres syntaxiques (AST). Elle offre de plus d'autres routines pour la manipulation des fichiers et des chaînes de caractères en utilisant les codes de hachage pour améliorer les performances.

Les parties frontales et dorsales ont besoin d'échanger des arbres syntaxiques des différents formalismes (Ada, code des aspects et code du weaver) supportés par le compilateur.

Les parties frontales construisent les nœuds des ASTs et les parties dorsales parcourent et lisent les mêmes nœuds. Il s'avère donc nécessaire que les routines relatives à la manipulation des ASTs soient situées dans un niveau supérieur à celui des parties frontales et dorsales où elles sont utilisées.

Parties frontales Les parties frontales de AspectAda permettent l'analyse lexicale, syntaxique et sémantique de trois types de code source : le code source Ada fonctionnel (figure 8.2, module A), le code des aspects (figure 8.2, module B) et le code du weaver (figure 8.2, module C). La sortie principale de chaque partie frontale est un arbre syntaxique. Pour la construction et le parcours de chacun de ces arbres, les parties frontales font appel aux routines de la bibliothèque centrale. Les sorties secondaires des parties frontales de AspectAda consistent en d'éventuels avertissements ou messages d'erreurs détectés lors des différentes analyses.

Partie dorsale La partie dorsale permet principalement le tissage et la génération du code Ada. Elle reçoit les trois arbres syntaxiques résultants des différentes parties frontales. Ensuite, elle parcourt et interroge chacun d'eux afin de produire un seul arbre Ada. Enfin, elle permet une génération de code Ada tissé par les aspects à partir de l'arbre Ada construit.

La conception de la nouvelle architecture de AspectAda d'une façon plus modulaire que l'ancienne offre plusieurs avantages. Elle permet de piloter plus simplement le processus de production tout en ayant la possibilité d'étendre ce processus en ajoutant des parties dorsales ou en enrichissant les parties déjà existantes. De plus, la bibliothèque centrale permet la manipulation des arbres syntaxiques à un haut niveau d'abstraction. Ceci permet de développer un parseur flexible, extensible et qui aide à la réparation. Cette architecture a été adoptée pour le développement du compilateur. Dans ce qui suit, nous allons détailler le processus d'implantation basé sur cette architecture modulaire.

8.4.2 Implantation

La nouvelle architecture proposée permet de gérer facilement le processus de production du code à travers sa flexibilité. Une telle architecture laisse possible l'extension du compilateur par l'ajout d'une nouvelle partie dorsale ou l'extension des parties existantes. En outre, la bibliothèque centrale permet de manipuler les différents arbres syntaxiques à un haut niveau d'abstraction. Pour plus d'informations concernant l'implantation des différentes parties du compilateur, les paquetages générés et les outils utilisés, le lecteur intéressé peut trouver plus de détails et des morceaux de code explicatifs dans [GZJ16a].

Par ailleurs, toutes les parties (frontales et dorsales) de notre compilateur sont basées sur la manipulation des arbres. Ceci est dans le but d'éviter la génération de code à la volée

et donc de minimiser le nombre d'accès répétitifs pour les même arbres syntaxiques. Dans notre travail, après avoir construit le nouveau arbre syntaxique abstrait, nous le déchargeons dans un ficher source. De plus, avant la génération de code, les ressources demandées pour le code généré sont prédictibles et les types et les structures de données sont connus à l'avance. Par conséquent, les applications générées par notre compilateur ne nécessitent jamais l'allocation dynamique de mémoire. Typiquement, toutes les ressources qui y sont demandées sont statiquement allouées.

La prédiction et l'allocation statique des ressources demandées avant la génération de code évite toute sorte de violation des contraintes temps-réel et garantie surtout le déterminisme des solutions proposées pour le développement du compilateur.

8.5 Conclusion

Dans ce chapitre, nous avons donné les résultats obtenus après une étude approfondie du langage AspectAda existant. Ces résultats montrent plusieurs défauts et lacunes de ce langage. Pour cela, nous avons proposé, d'une part, une approche pour son extension et, d'autre part, son adaptation pour le domaine temps-réel. Nous avons présenté les extensions et les corrections attribuées à la syntaxe et la sémantique de AspectAda ainsi que les fonctionnalités ajoutées à travers la Runtime et les règles de tissage et de transformation de code établies. Aussi, nous avons présenté l'architecture proposée pour le nouveau compilateur et les principes de développement afin de garantir son adéquation avec les systèmes temps-réel. Dans le chapitre suivant, nous validons les extensions proposées et implantées pour ce langage par un cas d'étude et nous illustrons tout le processus de développement proposé dans cette thèse par un deuxième cas d'étude.

9

Validation et résultats

9.1 Introduction

Dans les chapitres précédents, nous avons défini une approche de tolérance aux pannes pour les systèmes temps-réel distribués. Cette approche est traduite sous forme d'un ensemble d'étapes du processus de développement DP4FTRTS allant de la modélisation jusqu'à la génération de code. Le point de départ de ce processus est un modèle AADL enrichi avec l'annexe d'erreurs pour décrire les concepts de la tolérance aux pannes. Nous avons défini à ce niveau une approche de gestion automatique de réplication basée sur les propriétés AADL. À partir du modèle généré, l'utilisateur peut effectuer une évaluation des mesures de la sûreté de fonctionnement ou procéder à la génération de code. Visant la séparation des préoccupations même au niveau implantation, nous avons défini une approche de génération de code fonctionnel et de la tolérance aux pannes en se basant sur la programmation orientée aspect passant par un modèle intermédiaire décrit par le langage d'aspect architectural AO4AADL. Le code fonctionnel est généré en Ada et le code non fonctionnel est généré en AspectAda. Dans une dernière étape, nous avons effectué une étude du langage AspectAda pour évaluer son support des contraintes temps-réel d'une part et des concepts nécessaires pour la tolérance aux pannes d'autre part. Nous avons proposé des solutions pour les lacunes détectées et implanté un nouveau compilateur répondant à nos besoins.

Pour valider tout le processus proposé dans cette thèse, nous avons choisi comme cas d'étude un système médicale critique. Il s'agit d'une couveuse d'enfants. Cette étude de cas est inspirée des travaux de Larson et al [LHFD13] qu'ils ont appelée *Isolette*. Pour valider notre contribution par rapport au langage AspectAda, nous avons choisi un système de gestion de charge de travail issu du profil Ravenscar, permettant de valider son adaptation pour les systèmes temps-réel.

9.2 Couveuse d'enfant

La couveuse accueille un nouveau né, s'il en a besoin, dès sa naissance. Le nouveau-né, généralement prématuré, n'a pas encore la maturité nécessaire pour réguler sa température. Il a besoin d'être au chaud, c'est la fonction principale de la couveuse, également appelée incubateur. Afin d'assurer une hydrométrie optimale et se rapprocher des conditions de l'utérus maternel, la couveuse présente un milieu chaud. Pour cela, la température à l'intérieur de la couveuse doit être contrôlée et adéquate pour le nouveau-né. À travers des capteurs de surveillance, l'équipe médicale surveille la température, le rythme du cœur, la respiration, et le taux d'oxygène administré à l'enfant. Nous nous intéressons dans ce qui suit au système de règlage de la température à l'intérieur de la couveuse.

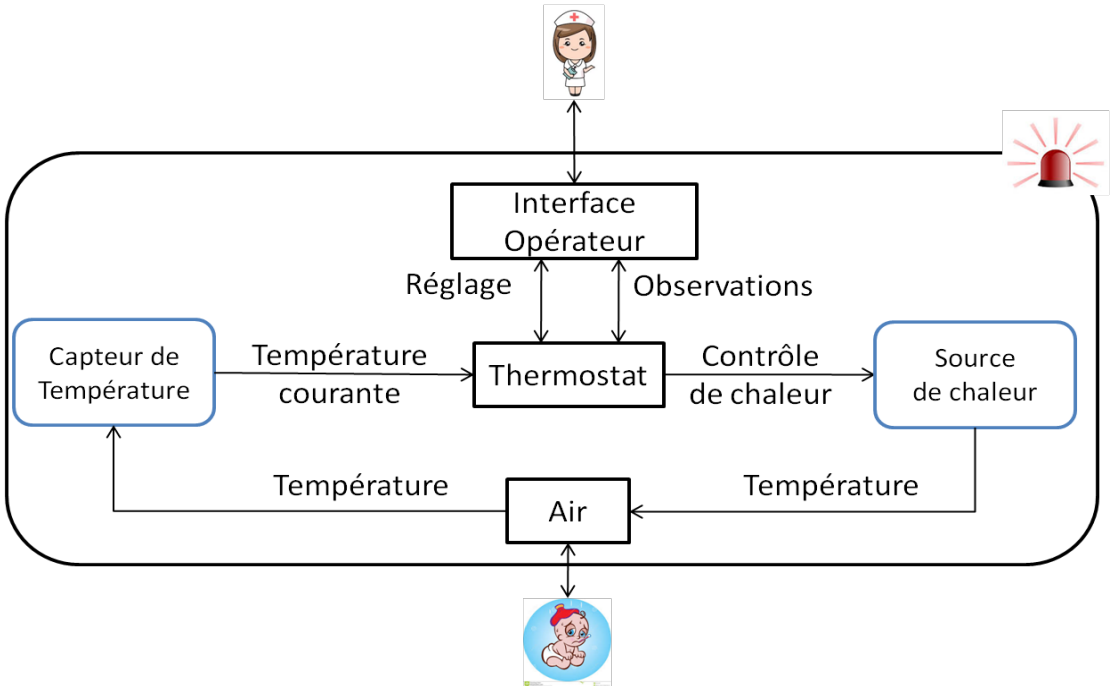

FIGURE 9.1 – Description du système Isolette

Un capteur de température prend la valeur de température de l'air et l'envoi vers le thermostat de la couveuse. Ce dernier contrôle une source de chaleur afin de produire une température de l'air dans une marge cible. Ceci est dans le but de garantir que la température de l'air à l'intérieur de l'Isolette est toujours appropriée pour éviter de nuire aux santés des nouveau-nés. Cette marge est fixée par le médecin par l'intermédiaire de l'interface opérateur. Si la température détectée est trop chaude ou trop froide, le système d'isolette émet une alarme par un sous-système externe du thermostat opérationnel. La communication entre les différents composants du système de la couveuse est illustrée sur la figure 9.1.
Dans la section 9.2.1, nous décrivons, d'abord, l'architecture du système avec le langage

AADL. Ensuite, la section 9.2.2 présente le modèle d'erreurs décrit avec l'annexe d'erreurs. La section 9.2.3 décrit les propriétés de réplication appliquées sur deux composants différents pour illustrer notre contribution. Enfin, dans la section 9.2.4, nous appliquons les règles de génération définies pour produire le modèle intermédiaire décrit avec AO4AADL et permettant d'intercepter le modèle AADL pour des fins de tolérance aux pannes.

9.2.1 Modélisation avec AADL

L'architecture globale de ce système, dont la représentation graphique est présentée dans la figure 9.2, est décrite avec le langage AADL.

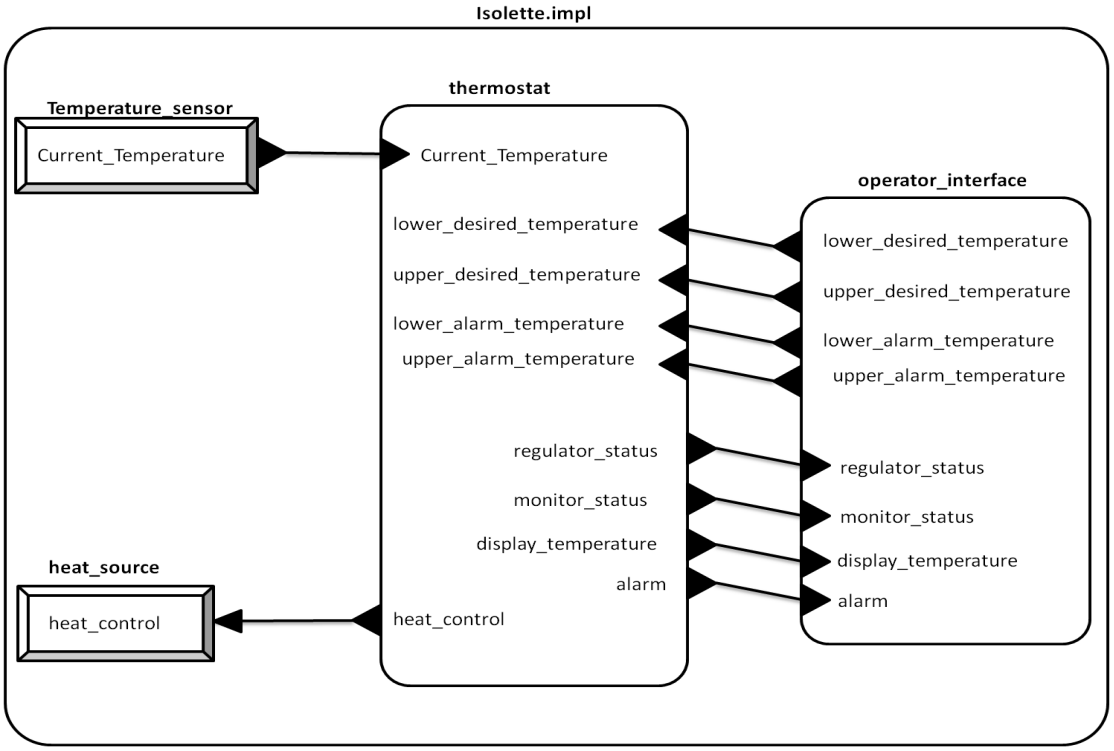

FIGURE 9.2 – Modèle AADL global correspondant au système isolette

Un composant parent de type *system* contient une variété de composants AADL dont la communication se fait à travers des connexions. Les deux capteurs, de température et la source de chaleur, sont représentés par deux composants de type *device*. Les deux contrôleurs, le thermostat et l'interface opérateur, sont décrits à ce niveau par deux composants de type *system*. L'architecture interne détaillée du composant décrivant le thermostat est présentée dans la figure 9.3. Le *thermotsat* est décrit avec un composant de type *system* qui contient comme sous-composants deux *process* appelés *monitor_temperature* et *regulate_temperature*.

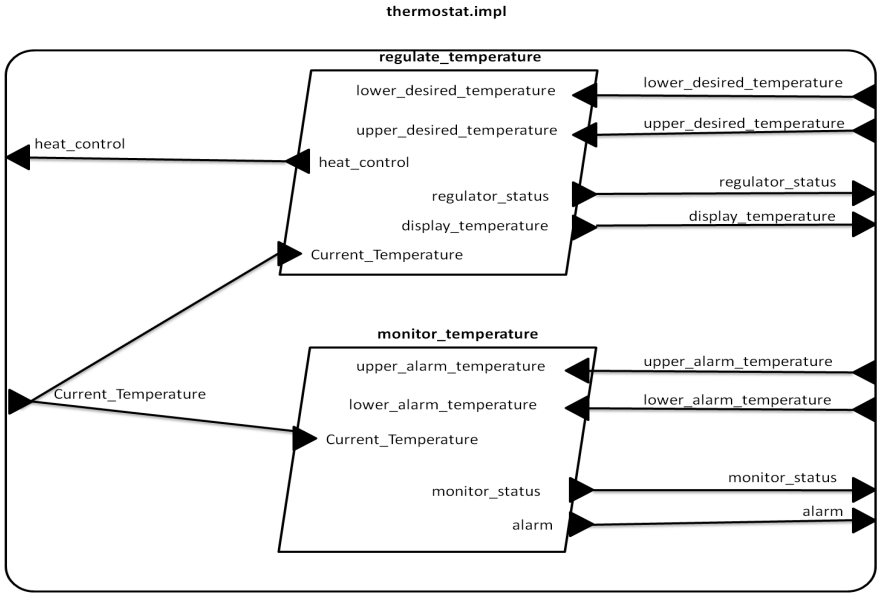

FIGURE 9.3 – Modèle AADL détaillant l'architecture interne du sous-système thermostat

9.2.2 Description du modèle tolérant aux pannes

Le listing 9.1 donne des déclarations réutilisables des types d'erreurs (lignes 3-15) pouvant survenir dans tout le système *isolette* par référencement dans des sous-clauses des modèles d'erreur. Ce listing décrit aussi une machine à état (lignes 17-25) décrivant le comportement de certains composants du système qui renferme deux états : *working* et *failed*.

Listing 9.1 – Bibliothèque du modèle tolérant aux pannes décrit avec EMA

```
1  annex EMV2
2  {**
3          error types
4            HeatControlError: type; --heater on or off inappropriately
5            AlarmError : type;              --the class of alarm errors
6            FalseAlarm : type extends AlarmError; --alarm erroneously sounded
7            MissedAlarm : type extends AlarmError; --alarm missed
8            StatusError : type; --mode and status errors
9            RegulatorStatusError : type extends StatusError; --indicated regulator status
                 wrong
10           RegulatorModeError : type extends StatusError; --regulator mode wrong
11           MonitorStatusError : type extends StatusError; --indicated monitor status wrong
12           MonitorModeError : type extends StatusError; --monitor mode wrong
13           ThreadFault renames type ErrorLibrary::EarlyServiceTermination; --thread fault
                 halts thread
14           InternalError : type; --an internal error was detected
15         end types;
16
17         error behavior FailStop
18           use types isolette;
```

```
19            events fail: error event;
20            states
21              working: initial state;
22              failed : state;
23            transitions
24              working -[fail]-> failed;
25          end behavior;
26  **};
```

Le listing 9.2 donne la description AADL du composant *temperature_sensor* et la description de la donnée *current_temperature* spécifiant la marge des valeurs de température possibles qui seront détectées par le capteur *temperature_sensor* et seront transmises vers le composant *Thermostat* via son port *current_temperature*.

Listing 9.2 – Spécification du capteur de température

```
1   data measured_temperature_range
2     properties
3       Data_Model::Real_Range => 68.0 .. 105.0;
4       Data_Model::Measurement_Unit => "Fahrenheit";
5   end measured_temperature_range;
6
7   data valid_flag
8     properties
9       Data_Model::Data_Representation => Enum;
10      Data_Model::Enumerators => ("Invalid", "Valid");
11  end valid_flag;
12
13  data current_temperature
14    properties
15      Data_Model::Data_Representation => Struct;
16      Data_Model::Element_Names => ("t", "status");
17      Data_Model::Base_Type => (classifier (measured_temperature_range), classifier (
            valid_flag));
18  end current_temperature;
19
20  device temperature_sensor_ts
21  features
22    current_temperature : out data port current_temperature;
23  annex EMV2
24    {**
25      use types ErrorLibrary;
26      error propagations
27         current_temperature: out propagation {OutOfRange,SubtleValueError};
28      flows
29         f: error source current_temperature{OutOfRange,SubtleValueError};
30      end propagations;
31    **};
32  end temperature_sensor_ts;
```

La deuxième partie de ce listing (lignes 23-30) décrit le modèle d'erreur relatif au composant *current_temperature*. Ce composant, s'il détecte une erreur, il peut la propager via sont port *current_temperature*. Les types d'erreurs qui peuvent être détectées sont : *OutO-*

fRange et *SubtleValueError* qui sont toutes les deux définies dans la bibliothèque standard d'erreur *ErrorLibrary*.

9.2.3 Gestion de la réplication

Pour éviter une action erronée suite à une fausse détection de la température et donc mettre en péril la vie des nouveaux-nés, nous avons appliqué une réplication active pour le capteur de température. Ce capteur est répliqué trois fois. Chacune des répliques donne une valeur de température détectée indépendamment des autres. Puis, un algorithme de vote majoritaire est appliqué pour décider de la bonne valeur.

La réplication du composant *temperature_sensor* est faite d'une manière automatique grâce à notre approche de réplication basée sur les propriétés. Pour cette raison, nous avons tout d'abord enrichi le modèle AADL initial par les propriétés de réplication illustrées dans le listing 9.3.

Listing 9.3 – Réplication du composant device `temperature_Sensor`

```
1 system implementation isolette.impl
2 ...
3 properties
4 Replication::Description => "Replication of the temperature sensor component" applies to
      temperature_Sensor;
5 Replication_Properties::Replica_Number => 3 applies to temperature_Sensor;
6 Replication_Properties::Replica_Type => ACTIVE applies to temperature_Sensor;
7 Replication_Properties::Replica_Identifiers => ("temp1", "temp2", "temp3") applies to
      temperature_Sensor;
8 Replication_Properties::Consensus_Algorithm_Source_Text => "Voting.Do_Vote" applies to
      temperature_Sensor.sensor_Out;
9 ...
```

Nous avons appliqué également notre approche de réplication sur un composant logiciel qui est le processus *monitor_temperature* dont les propriétés de réplication sont décrites dans le listing 9.4.

Après avoir appliqué les règles de transformations en se basant sur les propriétés de réplication définies, nous avons obtenu un nouveau modèle AADL riche avec les répliques. Ce modèle est présenté dans la figure 9.4.

À partir de cette figure, nous pouvons constaté la complexité du modèle généré par rapport au modèle initial. Cette complexité revient à la réplication des composants engendrant la génération de nouveaux composants qui constituent les répliques et les composants arbitres. Dans le tableau 9.1, nous donnons différentes statistiques sur les tailles des fichiers sources en nombre de lignes de code constituant le modèle AADL avant et après la réplication et dans la figure 9.5, un aperçu sur le nombre de composants au sein de notre modèle est donné.

Nous remarquons que la taille du code écrit par l'utilisateur est proche de la moitié de

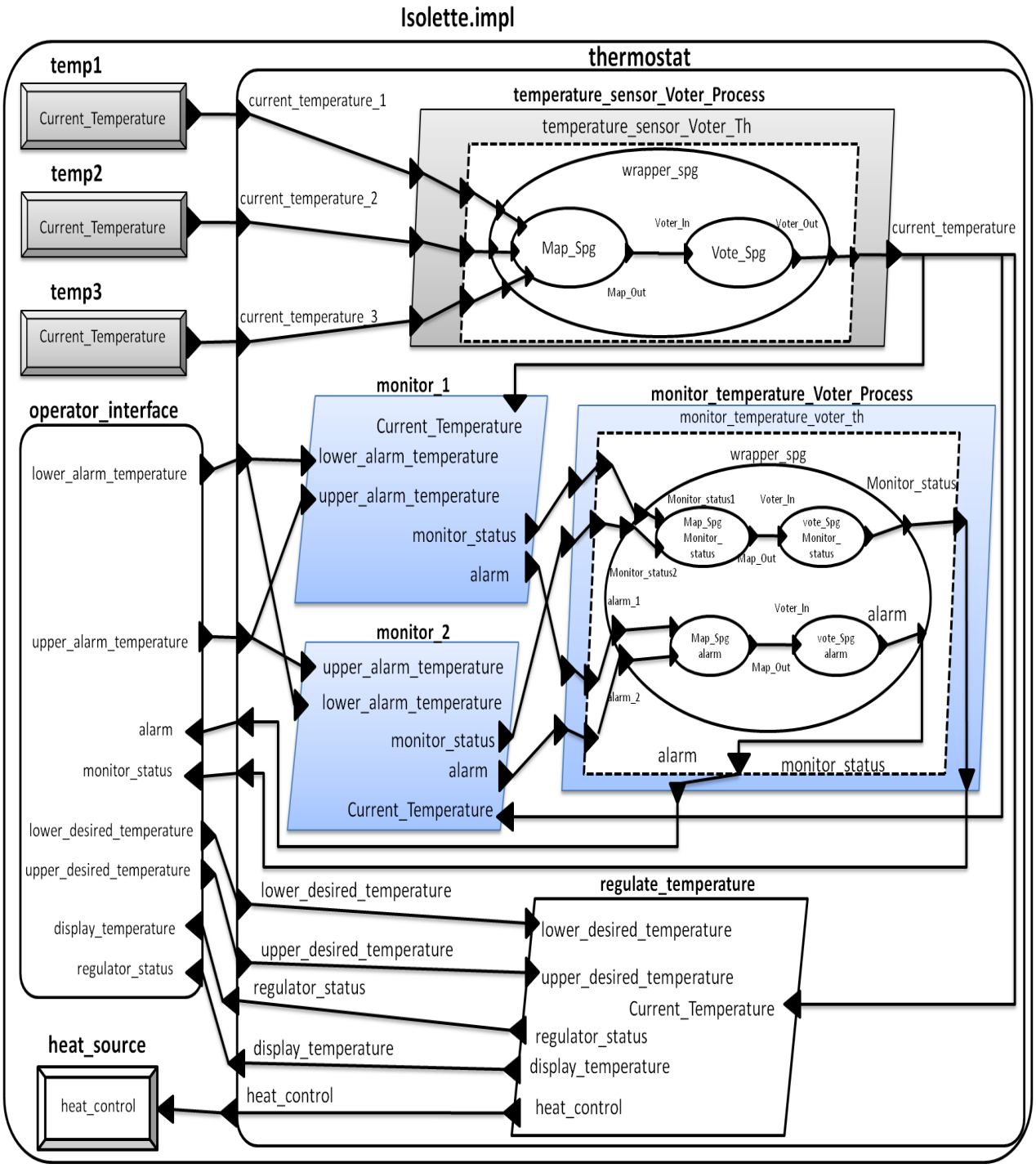

FIGURE 9.4 – Modèle AADL généré après la réplication des composants *temperature_sensor* de type device et *monitor_temperature* de type process

Listing 9.4 – Description des propriétés de réplication appliquée au composant *monitor_temperature*

```
1  system implementation thermostat_th.impl
2  subcomponents
3    regulate_temperature : process regulate_temperature_rt.impl;
4    monitor_temperature : process monitor_temperature_mt.impl;
5    detect_regulator_fail : device detect_regulator_failure.impl;
6  connections
7    ...
8  Properties
9    Replication_Properties::Description => "Replication of the process component" applies to
         monitor_temperature;
10   Replication_Properties::Replica_Number => 2 applies to monitor_temperature;
11   Replication_Properties::Replica_Type => ACTIVE applies to monitor_temperature;
12   Replication_Properties::Replica_Identifiers => ("monitor1", "monitor2") applies to
         monitor_temperature;
13   Replication_Properties::Consensus_Algorithm_Source_Text => "Voting.Do_Vote" applies to
         monitor_temperature.alarm;
14   Replication_Properties::Consensus_Algorithm_Source_Text => "Voting.Do_Vote" applies to
         monitor_temperature.monitor_status;
15 ...
```

la taille globale du modèle généré (66%). Il est à noter ici que le nombre des composants du modèle est très important (113), tandis que nous avons appliqué notre approche de réplication uniquement sur deux composants avec un nombre réduit de répliques. Avec d'autres cas d'étude présentés dans nos publications, nous avons atteint la génération automatique de 50% et même de 75% du code global du modèle (Voir [GZJ16b]). Ainsi, notre approche de gestion automatique de réplication peut aider le concepteur de générer un modèle tolérant aux pannes, tout en réduisant le risque d'erreurs et en diminuant le nombre de lignes de code d'une manière significative.

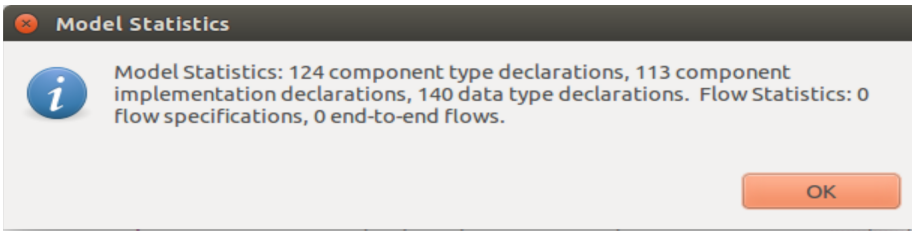

FIGURE 9.5 – Statistiques sur le modèle AADL

TABLE 9.1 – Tailles des fichiers sources (en ligne de code)

Code de l'utilisateur (Modèle AADL de base)	453
Code de l'utilisateur (Modèle d'erreur)	71
Propriétés de réplication des deux composants (device et process)	15
Modèle généré	757

Pour valider la cohérence du modèle généré, nous avons utilisé Ocarina à nouveau pour analyser et générer le code applicatif correspondant en utilisant le middleware PolyORB-HI, le compilateur GNATforLEON [MS04] et enfin le simulateur TSIM [Gei08]. Dans la section suivante, nous illustrons le processus de génération de code à travers le même cas d'étude.

9.2.4 Génération de code

Dans la section précédente, nous avons décrit le modèle AADL de base du système *Isolette*. Nous avons également enrichi ce modèle par des clauses de l'annexe EMA dans le but de décrire le modèle d'erreur d'architecture et nous avons appliqué notre approche de réplication pour donner lieu à la génération du modèle AADL final servant à la génération de code. Dans cette section nous présentons, les clauses de l'annexe AO4AADL résultant de la transformation de modèle appliquée sur l'extension du modèle AADL avec l'annexe d'erreurs.

Le capteur de température peut être source de deux types d'erreurs : l'erreur *OutOfRange* ou l'erreur *SubtleValueError* comme ils sont présentées au niveau du listing 9.2. Pour cela, la valeur de la chaîne de caractères `possible_errors` va prendre comme valeur la chaîne concaténant le nom des deux erreurs. L'erreur *OutOfRange* est détectable à partir de la valeur de température elle même détectée [SAE15] puisqu'il s'agit d'une valeur devant appartenir à une marge bien déterminée de valeurs. En contre partie, la valeur de l'erreur *SubtleValueError* ne peut être détectée qu'à travers une source additionnelle à savoir la réplication.

Ainsi, nous pouvons générer le code AO4AADL correspondant qui est capable de vérifier si la valeur de température courante est bien située dans la marge devinée des valeurs dans le but de détecter l'erreur *OutOfRange* comme le montre le listing 9.5 (lignes 19-26).

La marge des valeurs acceptée est déduite à partir de la propriété `Real_Range` associée à la donnée `measured_temperature_range` au niveau du modèle AADL (listing 9.2 (lignes 1-7)). Les clauses de l'annexe AO4AADL sont générées dans le but d'intercepter le pointcut et de propager l'erreur à travers le port de type `out` si une erreur de type *OutOfRange* est détectée. Le modèle d'aspect est généré en appliquant les règles de transformation régissant la propagation d'erreur de type *in* présenté dans le listing 7.2. Ces clauses de l'annexe AO4AADL sont générées pour étendre la spécification du capteur de température afin d'intercepter le port de sortie du dispositif correspondant puis de détecter les erreurs déclenchées.

La spécification AADL du composant thread *manage_heat_source_mhs* est aussi enrichie par des clauses de l'annexe d'erreurs pour décrire les chemins de propagation d'erreur (voir listing 9.6). En effet, une détection erronée de la température provoque un mauvais contrôle de la chaleur. La spécification AO4AADL générée pour la propagation d'erreur à l'intérieur du composant thread est présentée par le listing 9.7.

Toutes les spécifications AO4AADL décrites dans cette section sont établies dans le but

Listing 9.5 – Spécification du capteur de température avec AO4AADL

```
data Error_Types
  Data_Model::Data_Representation => String;
end Error_Types;

device Temperature_Sensor_Ts
features
  current_temperature : out data port Current_Temperature;
annex ao4aadl
  {**
    aspect Temperature_Sensor_Ts_Aspect {
    pointcut outport_current_temperature (): call outport (current_temperature (..)
            && args (temperature_value)
    advice before () : outport_current_temperature ()
    {
       variables {possible_errors: string_Type;
                  detected_errors: string_Type;}
       initially {detected_errors:="";possible_errors:="OutOfRange;SubtleValueError";}
       current_temperature ? temperature_value;
       if (temperature_value.t > 105.0 or temperature_value.t < 68.0)
       {
          detected_errors + := "Out_Of_Range";
       }
       temperature_value.possible_errors:= possible_errors;
       temperature_value.detected_errors := detected_errors;
       current_temperature ! temperature_value;
    }
  **};
end temperature_sensor_ts;
```

Listing 9.6 – Spécification du thread *manage_heat_source_mhs*

```
thread manage_heat_source_mhs
features
  heat_control : out data port Iso_Variables::on_off;
  current_temperature : in data port Iso_Variables::current_temperature;
  desired_range : in data port Iso_Variables::desired_range;
  regulator_mode : in data port Iso_Variables::regulator_mode;
annex EMV2
  {**
    use types ErrorLibrary,isolette;
  error propagations
  heat_control: out propagation {HeatControlError};
  current_temperature: in propagation {SubtleValueError};
  regulator_mode: in propagation {RegulatorModeError};
  flows
    mrmsve: error path current_temperature{SubtleValueError}
      -> heat_control{HeatControlError};
    mrmif: error path regulator_mode{RegulatorModeError}
      -> heat_control{HeatControlError};
  end propagations;
  **};
end manage_heat_source_mhs;
```

Listing 9.7 – Spécification du thread *manage_heat_source_mhs* avec AO4AADL

```
thread manage_heat_source_mhs
features
  heat_control : out data port Iso_Variables::on_off;
  current_temperature : in data port Iso_Variables::current_temperature;
  desired_range : in data port Iso_Variables::desired_range;
  regulator_mode : in data port Iso_Variables::regulator_mode;
annex ao4aadl
  {**
    aspect component_behavior{
    pointcut fail_stop_behavior : call inport
             (current_temperature (..), regulator_mode(..))
    advice after () : fail_stop_behavior ()
    {  variables{detected_errors : String_Type;
                 temperature_value : struct_type;}
       initially {detected_errors := "";}
       detected_errors := current_temperature ? temperature_value.detected_errors
       if (detected_errors!="")
       { working -> failed;
         if (temperature_value.detected_errors ="SubtleValueError")
         { heat_control.detected_errors :="HeatControlError";}
       }
       detected_errors +:= regulator_mode ? temperature_value.detected_errors
    }
    }
  **};
end manage_heat_source_mhs;
```

de détection et de recouvrement des erreurs prévues. À partir de ces spécifications, un code sera automatiquement généré en AspectAda, AspectJ ou AspectC pour être tissé avec le code fonctionnel généré respectivement en Ada, Java ou C grâce à l'intergiciel PolyORB-HI et donc intercepter le système en cours d'exécution.

9.2.5 Synthèse

Dans la première partie de ce chapitre, nous avons illustré l'étape de modélisation de l'architecture globale du système ainsi que de la tolérance aux pannes. Nous avons utilisé le langage AADL et son annexe d'erreurs pour la description respectivement des préoccupations fonctionnelles et de la tolérance aux pannes. Nous avons ensuite illustré notre contribution pour la gestion de réplication en faisant appel à l'ensemble de propriétés *Replication_Properties* que nous avons définies. Nous avons pu constater l'avantage de l'utilisation de ces propriétés non seulement pour éviter les risques d'erreurs engendrées par la réplication manuelle mais aussi pour profiter du gain en termes de réduction de la complexité (nombre de lignes de code) et de temps de conception très important. Nous avons, par la suite, appliqué notre approche de génération de code tolérant aux pannes partant du modèle AADL et son annexe d'erreurs basée sur la transformation de modèle vers un langage d'aspect. Afin d'offrir plus de flexibilité pour les choix des plate-formes cibles et donner plus de liberté au développeur en passant par un code générique indépendant de

la plate-forme, nous avons proposé une transformation intermédiaire (PIM) vers le langage AO4AADL permettant d'intégrer des aspects architecturaux visant la détection des pannes au niveau modèle. Ce code, à son tour, donne lieu à la génération en différents langages d'aspect en profitant des générateurs disponibles au niveau de l'intergiciel PolyORB-HI selon la plate-forme choisie.

Dans la suite de ce chapitre, nous nous intéressons à la validation de notre contribution relative à l'étude et l'adaptation du langage AspectAda pour le développement temps-réel.

9.3 Système de gestion de charge de travail

Pour valider notre contribution pour l'étude et l'adaptation du langage AspectAda proposée dans le chapitre 8, nous avons choisi comme cas d'étude un exemple classique d'application critique. Il s'agit d'un système de gestion de charge de travail (*Workload Manager*) inspiré du profil Ravenscar [ABT04].

Nous commençons dans la première section, par la description du système en mettant l'accent sur son architecture globale. Ensuite, nous détaillons notre extension de cette application par les concepts de la programmation orientée aspect pour l'implantation séparée de la traçabilité d'exécution. Enfin, nous interprétons les résultats obtenus.

9.3.1 Description du système

Le système de gestion de charge de travail consiste en une tâche périodique pouvant gérer des charges de travail variables : des travaux qui sont réguliers et d'autres qui sont supplémentaires. Les premiers sont effectués par une tâche périodique de haute priorité. Les seconds sont délégués à une tâche sporadique de moindre priorité. Par ailleurs, le système est apte de recevoir des interruptions venant de l'extérieur par le moyen d'une tâche de très haute priorité. Ces interruptions sont sauvegardées dans un tampon spécifique et sont traitées par une tâche sporadique de très faible priorité qui est réveillée de temps en temps par la tâche périodique principale.

Ce système est donc formé de quatre processus légers : `Regular_Producer`, `On_Call_Producer`, `Activation_Log_Reader` et `External_Event_Server` dont les caractéristiques sont définies dans le tableau 9.2 [10]. Ces différents processus partagent trois données protégées qui sont :

1. `Request_Buffer` : un tampon qui reçoit les ordres des charges de travail supplémentaires. Il est rempli par `Regular_Producer` et consulté par `On_Call_Producer`,
2. `Event_Queue` : une file d'attente des interruptions extérieures. Elle est remplie par le périphérique émettant les interruptions et consultée par `External_Event_Server`,

10. Pour les processus légers sporadiques, la colonne "Période" indique le temps minimal entre deux déclenchements.

3. `Activation_Log` : un journal des interruptions à traiter. Il est rempli par `External_Event_Server` et consulté par `Activation_Log_Reader`.

TABLE 9.2 – Description et propriétés temporelles des processus légers [Zal08]

Nom	Description	Protocole	Période (ms)	Échéance (ms)	WCET (ms)	Priorité
`Regular_ Producer`	Reçoit les interruptions extérieures et les enregistre dans un tampons spécifique	Périodique	1000	500	498	7
`On_Call_Producer`	Effectue la charge de travail régulière. Il délègue la charge de travail supplémentaire et le traitement des interruptions à d'autres processus légers	Sporadique	1000	800	250	5
`Activation_Log_Reader`	Effectue la charge de travail supplémentaire	Sporadique	1000	1000	125	3
`External_Event_Server`	Effectue une quantité de travail correspondant au traitement de la dernière interruption reçue.	Sporadique	5000	100	2	11

Le système de gestion de charge de travail contient aussi deux entités passives : La première est `Activation_Manager` qui est une entité qui assure la synchronisation des activations des tâches après l'initialisation du système. En effet, elle prévoit l'instant auquel toutes les tâches périodiques dans le système commencent leurs exécutions. La seconde est `Production_Workload` qui est une entité qui abrite l'opération *Small Whetstone* [CWS76] qui traite les travaux demandés par une des tâches.

9.3.2 Extension par les aspects

Nous avons présenté dans la section précédente l'architecture de base de notre cas d'étude : système de gestion de charge de travail. Nous nous intéressons dans cette section à l'extension de ce système par les aspects afin d'évaluer le langage AspectAda. La préoccupation transversale mise en évidence est la traçabilité (*logging*) qui consiste à sauvegarder les traces d'exécution des différentes tâches du système. Il s'agit de l'émission de messages suite au déclenchements des événements suivants : début et fin de l'exécution d'une tâche, arrivée d'une interruption et activation d'une tâche sporadique.

L'utilité des messages émis sauvegardés réside dans le suivi du fonctionnement du système en cours d'exécution et la détection des comportements anormaux s'ils y existent. En particulier, nous nous intéressons dans notre contexte aux exigences temps-réel. Grâce à la trace d'exécution, nous serons capables de vérifier le respect des contraintes temps-réel et précisément le déterminisme. Il s'agit de vérifier le respect des priorités entre les tâches et des échéances des différentes tâches.

Pour ajouter la trace d'exécution à notre application, nous avons implanté cette propriété transversale à travers des aspects décrits avec le langage AspectAda pour intercepter les événements cités et sauvegarder les messages correspondants. Le code AspectAda implanté

est composé de deux parties : code de l'aspect et code du weaver. Pour garder une trace complète de l'exécution, l'aspect `Logger_Aspect`, dont la spécification et le corps sont décrits respectivement dans les listing 9.8 et 9.9, renferme cinq advices :

Listing 9.8 – Spécification de l'aspect `Logger_Aspect`

```
1  with AspectAda_Types;
2  with Activation_Log;
3
4  generic
5     Op_Pointcut    : pointcut;
6     Signal_PC      : pointcut;
7     Start_PC       : pointcut;
8     Log_Reader_PC  : pointcut;
9  aspect Logger_Aspect is
10
11    Old_Activation_Counter : Activation_Log.Range_Counter := 0;
12
13    procedure Display_Time;
14
15    advice Before_Operation (thisJoinPoint : AspectAda_Types.Join_Point);
16    for Before_Operation'pointcut use Op_Pointcut;
17
18    advice After_Operation (thisJoinPoint : AspectAda_Types.Join_Point);
19    for After_Operation'pointcut use Op_Pointcut;
20
21    advice Before_Signal;
22    for Before_Signal'pointcut use Signal_PC;
23
24    advice After_Start (thisJoinPoint : AspectAda_Types.Join_Point);
25    for After_Start'pointcut use Start_PC;
26
27    advice Around;
28    for Around'pointcut use Log_Reader_PC;
29
30 end Logger_Aspect;
```

Listing 9.9 – Corps de l'aspect `Logger_Aspect` implanté pour `Workload_Manager`

```
1  with System.IO;
2  with Ada.Real_Time;
3  with Activation_Manager;
4  with Aspect_Ada;
5
6  aspect body Logger_Aspect is
7
8     procedure Display_Time is
9       use Ada.Real_Time;
10    begin
11      System.IO.Put( "["
12                     & Duration'Image
13                     (To_Duration
14                      (Clock - Activation_Manager.System_Start_Time))
```

```
15                      & "] ");
16    end Display_Time;
17
18    advice Before_Operation (thisJoinPoint : AspectAda_Types.Join_Point) is
19    begin
20       Display_Time;
21       if Aspect_Ada.Get_Name(thisJoinPoint) =
22                      "External_Event_Server_Parameters.Server_Operation"  then
23          System.IO.Put_Line ("External Event Server: received an external interrupt");
24       elsif Aspect_Ada.Get_Name(thisJoinPoint) =
25                      "On_Call_Producer_Parameters.On_Call_Producer_Operation" then
26          System.IO.Put_Line ("On Call Producer: doing some work.");
27       elsif Aspect_Ada.Get_Name(thisJoinPoint) =
28                      "Regular_Producer_Parameters.Regular_Producer_Operation" then
29          System.IO.Put_Line ("Regular Producer: doing some work.");
30       end if;
31    end Before_Operation;
32
33    advice After_Operation (thisJoinPoint : AspectAda_Types.Join_Point) is
34    begin
35       Display_Time;
36       if Aspect_Ada.Get_Name(thisJoinPoint) =
37                      "External_Event_Server_Parameters.Server_Operation"  then
38          System.IO.Put_Line ("External Event Server: end of sporadic activation.");
39       elsif Aspect_Ada.Get_Name(thisJoinPoint) =
40                      "Regular_Producer_Parameters.Regular_Producer_Operation" then
41          System.IO.Put_Line ("Regular Producer: end of cyclic activation");
42       elsif Aspect_Ada.Get_Name(thisJoinPoint) =
43                      "On_Call_Producer_Parameters.On_Call_Producer_Operation" then
44          System.IO.Put_Line ("On Call Producer: end of sporadic activation.");
45       end if;
46    end After_Operation;
47
48    advice Before_Signal is
49    begin
50      Display_Time;
51      Ada.Text_IO.Put_Line ("Signaling 'Activation Log Reader'");
52    end Before_Signal;
53
54    advice After_Start (thisJoinPoint : AspectAda_Types.Join_Point) is
55      Returned_Value           : Boolean;
56      Args                     : AspectAda_Types.Arg_Array;
57      Activation_Parameter_Arg : AspectAda_Types.Arg;
58      Activation_Parameter     : Positive;
59    begin
60      Returned_Value := AspectAda_Types.Extract_Boolean
61                (AspectAda_Types.Boolean_Holder(
62                            Aspect_Ada.Get_Returned_Value_Access(thisJoinPoint).all));
63      Args := Aspect_Ada.Get_Args(thisJoinPoint);
64      Activation_Parameter_Arg := Args(1);
65      Activation_Parameter := AspectAda_Types.Extract_Positive
66                (AspectAda_Types.Positive_Holder(
67                            Aspect_Ada.Get_Value_Access(Activation_Parameter_Arg).all));
68      Display_Time;
69      if Returned_Value then
```

```
70          Ada.Text_IO.Put_Line ("Sending␣extra␣work␣to␣'On_Call_Producer':␣"
71                      & Activation_Parameter'Img);
72        else
73          Ada.Text_IO.Put_Line ("Failed␣sporadic␣activation.");
74        end if;
75      end After_Start;
76
77      advice Around is
78        use Ada.Real_Time;
79      begin
80        Display_Time;
81        Ada.Text_IO.Put_Line ("Activation␣Log␣Reader:␣do␣some␣work.");
82        Proceed;
83        Display_Time;
84        if Interrupt_Arrival_Counter /= Old_Activation_Counter then
85           Ada.Text_IO.Put_Line ("Read␣external␣new␣interruption:"
86                         & Activation_Log.Range_Counter'Image (Interrupt_Arrival_Counter)
87                         & ".␣Arrived␣at␣["
88                         & Duration'Image
89                         (To_Duration
90                          (Interrupt_Arrival_Time - Activation_Manager.System_Start_Time))
91                         & "]");
92          Old_Activation_Counter := Interrupt_Arrival_Counter;
93        else
94          Ada.Text_IO.Put_Line ("Activation␣Log␣Reader:␣no␣new␣interrupts.");
95        end if;
96        --And finally we report nominal completion of current
97        --activation.
98        Display_Time;
99        Ada.Text_IO.Put_Line ("Activation␣Log␣Reader:␣end␣of␣parameterless␣sporadic"
100                     & "␣activation.");
101     end Around;
102 end Logger_Aspect;
```

— **Before_Operation** : Il s'agit d'un advice *Before* associé au pointcut `Op_Pointcut` qui intercepte l'exécution des opérations des tâches `External_Event_Server`, `Regular_Producer` et `On_call_Producer`. Cet advice *Before* enregistre le début de l'exécution de l'opération interceptée selon la nature de la tâche courante.

— **After_Operation** : Il s'agit d'un advice *After* ayant un comportement très similaire à l'advice `Before_Operation` sauf que l'advice `After_Operation` enregistre la fin de l'exécution de la tâche interceptée par le pointcut `Op_Pointcut` parmi les trois tâches (`External_Event_Server`, `Regular_Producer` et `On_call_Producer`).

— **Before_Signal** : cet advice est créé dans le but d'enregistrer les activations du thread `Activation_Log_Reader`. En effet, ce dernier est signalé par le thread `Regular_Producer` qui appelle la procédure `Signal`. Pour ce faire, l'advice `Before_Signal` est associé au *pointcut* `Signal_PC` permettant d'intercepter l'appel à la procédure `Signal`.

— **After_Start** : cet advice permet de sauvgarder les activations du thread

`On_Call_Producer`. Le thread `Regular_Producer` active le thread `On_Call_Producer` en lui envoyant la charge de travail supplémentaire. Ceci est fait à travers un appel à la fonction `Start` qui prend en paramètre la charge supplémentaire. Par ailleurs, l'advice `After_Start` est associé au *pointcut* `Start_PC` permettant d'intercepter l'appel de la fonction `Start` du thread `On_Call_Producer`.

— **Around** : cet advice est associé au *pointcut* `Log_Reader_PC` qui intercepte l'exécution de l'opération du thread `Activation_Log_Reader`. Le comportement de cet advice Around est divisé en trois parties :

(1) Avant l'exécution de l'opération du thread `Activation_Log_Reader` : Enregistrer le début de l'exécution de ce *thread*,

(2) Exécution de l'opération du thread `Activation_Log_Reader` (en faisant appel à l'instruction *proceed*),

(3) Après l'exécution de l'opération du thread `Activation_Log_Reader` : Enregistrer la nouvelle interruption lue par ce thread (évidemment s'il y a une nouvelle interruption enregistrée dans le journal `Activation_Log`). Puis sauvegarder la fin de l'exécution de ce *thread*.

À travers notre étude de cas, nous avons utilisé les différents types d'advices et de joinpoints dans le but de tester la majorité des règles de tissage et de génération de code établies dans le chapitre 8.

Listing 9.10 – Code du Weaver `Workload_Rules`

```
1  with Logger_Aspect;
2
3  weaver Workload_Rules is
4
5   Operation_PC : pointcut := execution (procedure *_Parameters.Regular_Producer_Operation)
6                                                    or
7                              execution (procedure *_Parameters.Server_Operation)
8                                                    or
9                              execution (procedure *_Parameters.On_Call_Producer_Operation);
10
11  Signal_PC    : pointcut := call (procedure Activation_Log_Reader.Signal);
12
13  Start_PC     : pointcut := call (function On_Call_Producer.Start (..) return Boolean);
14
15  Log_Reader_PC: pointcut := execution
16          (procedure Activation_Log_Reader_Parameters.Activation_Log_Reader_Operation(..));
17
18  aspect My_Logger_Aspect is new Logger_Aspect(Operation_PC,
19                                               Signal_PC,
20                                               Start_PC,
21                                               Log_Reader_PC);
22
23 end Workload_Rules;
```

Le tisseur `Workload_Rules` définit les pointcuts permettant d'intercepter les évènements concernés. Il crée une instance de l'aspect `Logger_Aspect` permettant d'associer les pointcuts aux advices. Nous présentons dans le listing 9.10 le code du tisseur `Workload_Rules.aaw`

Après avoir implanté et validé les règles de tissage et de génération de code, nous avons testé ces règles avec l'application `Workload_Manager`. L'ensemble des fichiers sources Ada produits est le suivant :

— `Logger_Aspect.ad[b|s]` : contiennent la spécification et le corps du paquetage `Logger_Aspect`. Ils projettent la spécification et le corps de l'aspect en des entités Ada.

— `AspectAda_Types.ad[b|s]` : contiennent la spécification et le corps du paquetage `AspectAda_Types`. Ils contiennent les types de données utilisés par la `Runtime` et le code généré. De plus, ils implantent les routines nécessaires pour les conversions des types.

— `Regular_Producer_Parameters.adb`, `External_Event_Server.adb` et `On_Call_Producer.adb` : contiennent respectivement les corps des paquetages `Regular_Producer_Parameters`, `External_Event_Server` et `On_Call_Producer` après le tissage des advices `Before_Operation` et `After_Operation`.

— `Tmp_Activation_Log_Reader.ad[b|s]` : contiennent la spécification et le corps du paquetage `Tmp_Activation_Log_Reader`. Ils implantent le tissage de l'advice `Before_Signal` associé à l'appel de la procédure `Signal` du paquetage `Activation_Log_Reader`.

— `Tmp_On_Call_Producer.ad[b|s]` : contiennent la spécification et le corps du paquetage `Tmp_On_Call_Producer`. Ils implantent le tissage de l'advice `After_Start` associé à l'appel de la fonction `Start` du paquetage `On_Call_Producer`.

— `Activation_Log_Reader_Parameters.ad[b|s]` : contiennent la spécification et le corps du paquetage `Activation_Log_Reader_Parameters`. Ils implantent le tissage de l'advice `Around` associé à l'exécution de la procédure `Activation_Log_Reader_Operation` du paquetage `Activation_Log_Reader_Parameters`.

9.3.3 Résultats et interprétations

Pour l'exécution de notre système de gestion de charge de travail, nous avons choisi d'envoyer aléatoirement des interruptions avec un temps d'arrivée minimal entre deux interruptions supérieur ou égal à 5 secondes. Ceci vise à respecter les contraintes de sporadicité du thread `External_Event_Server` qui gère la réception des interruptions.

Avant d'entamer l'exécution de l'application, nous avons effectué une étude théorique de l'exécution des différents threads (figure 9.6) en nous basant sur les données du tableau 9.2. Ceci est dans le but de comparer les résultats obtenus après l'exécution réelle avec les résultats de l'étude théorique.

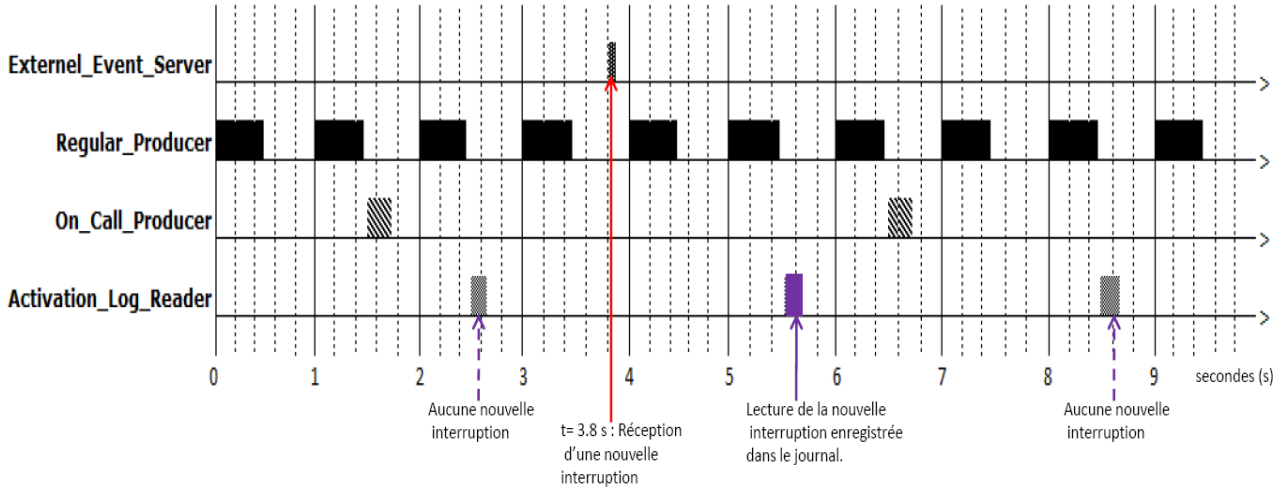

FIGURE 9.6 – Étude théorique de l'exécution

Concernant l'algorithme d'ordonnancement, nous avons choisi l'ordonnancement par priorité statique (*RMS : Rate Monotonic Scheduling* [LL73]) pour effectuer l'étude théorique. Par ailleurs, les instants des activations des threads sporadiques `Activation_Log_Reader` et `On_Call_Producer` sont déduites à partir du comportement du thread `Regular_Producer`. En effet, le thread `Regular_Producer` se charge de l'activation de ces deux threads sporadiques. Pour l'activation de chacun d'eux, `Regular_Producer` définit et vérifie une condition spécifique. Une fois cette condition est vérifiée, alors il y aura activation du thread (soit `On_Call_Producer` soit `Activation_Log_Reader`) associé à cette condition.

En examinant la figure 9.6, nous remarquons qu'à sa première activation (à l'instant 2.5 secondes), le thread `Activation Log Reader` n'a pas trouvé aucune interruption enregistrée par le thread `External Event Sever`. Par contre, à sa deuxième activation (à l'instant 5.5 secondes), il lit l'interruption reçue par le thread `External Event Sever` à l'instant 3.8 secondes.

Notre application temps-réel est compilée par le compilateur GNAT FOR LEON. Puis, elle a été exécutée à l'aide du simulateur du processeur LEON2 **Tsim**. Pour des limites liées au simulateur, nous avons modifié le thread `External_Event_Server` pour rendre interne l'envoi des interruptions. Une partie de la trace obtenue de l'exécution de l'application `workload_Manager` est présentée dans la figure 9.7. Nous remarquons que tous les travaux supplémentaires sont exécutés. Toutes les interruptions externes ont été traitées par la tâche `Activation_Log_Reader` ce qui montre que le dimensionnement du système utilisant une file d'attente de taille 1 pour les interruptions est correct puisqu'aucune interruption n'a été perdue.

Si nous comparons la trace d'exécution avec l'étude théorique effectuée, nous pouvons déduire que :

FIGURE 9.7 – Trace d'exécution du workload Manager sur tsim

1. Les tâches respectent bien leurs échéances ;
2. Les priorités entre les tâches sont aussi respectées.

Par conséquent, l'application obtenue après le tissage des aspects est déterministe. Elle respecte les contraintes temporelles de l'application d'origine décrites dans le tableau 9.2

(avant l'introduction des aspects). Ceci montre que les règles de tissage et de génération de code élaborées ainsi que l'implantation du compilateur n'affectent pas le déterminisme de l'application.

9.4 Conclusion

Dans ce chapitre, nous avons présenté deux études de cas visant la validation de toutes nos contributions.

La première étude de cas consiste à un système temps-réel critique simple issue du domaine médicale dans le but d'illustrer tout le processus de développement proposé dans cette thèse. Il s'agit d'un système de couveuse d'enfants appelé **Isolette**. Après la description abstraite du système, nous l'avons modélisé avec le langage AADL décrivant l'architecture globale du système tout en mettant l'accent sur les différentes interactions entre les composants qui le constituent. Ce système a été, par la suite, étendu par l'annexe d'erreurs dans le but de donner le modèle d'erreur architectural tout en focalisant sur les types d'erreurs pouvant survenir, les manières de détection envisagées et les mécanismes de recouvrement mis en place. Dans le but de répliquer certains composants automatiquement, nous avons appliqué les propriétés *Replicaton_Properties* que nous avons définies pour générer automatiquement un modèle supportant la réplication. Enfin, à partir du modèle généré, nous avons appliqué notre approche pour la génération de code fonctionnel et celui tolérant aux pannes à travers le passage par le langage intermédiaire AO4AADL.

La seconde étude de cas consiste en un système de gestion de charge de travail comme une application répartie pour la validation de la nouvelle version de AspectAda proposée pour combler les lacunes et les problèmes par rapport à sa compatibilité pour les systèmes temps réel.

Ainsi, nous avons validé toutes les contributions de cette thèse en passant par les différentes phases du processus DP4FTRTS proposé pour la production des systèmes temps-réel distribués tolérants aux pannes.

Conclusion générale et perspectives

Dans les chapitres précédents de ce mémoire, nous avons présenté la problématique de notre travail de thèse : proposer un processus de développement pour la mise en place des systèmes temps-réel distribués tolérants aux pannes : allant de la modélisation jusqu'à l'implantation. Aussi, nous avons proposé et implanté des solutions pour répondre à cette problématique. Pour valider ces solutions, nous avons mis en œuvre plusieurs études de cas parmi lesquelles nous avons présenté une pour illustrer le processus de développement et une autre pour valider notre contribution par rapport au langage AspectAda. Enfin, dans ce chapitre, nous rappelons les réalisations de ce travail de thèse et nous présentons les conclusions et les perspectives pour étendre le travail réalisé.

1 Rappel des contributions

La problématique principale de notre travail de thèse consistait à définir un processus de développement permettant de modéliser et de générer automatiquement les applications temps-réel distribuées tolérantes aux pannes. En explorant la littérature, nous avons discuté plusieurs travaux visant la modélisation de la sûreté de fonctionnement et en particulier de la tolérance aux pannes à des fins d'analyse dès le niveau modèle. Toutefois, nous avons remarqué un manque de travaux visant la génération du code tolérant aux pannes d'après tels modèles. Pour cela nous avons proposé nos propres solutions tout en adoptant la séparation des préoccupations durant toutes les phases du cycle de développement.

La solution que nous avons proposée consistait tout d'abord à définir un processus de développement en tenant compte de la tolérance aux pannes des systèmes temps-réel dès la phase de modélisation (Chapitre 3). Ce processus vise offrir un support pour la conception des systèmes temps-réel critiques tout en mettant l'accent sur l'aspect tolérance aux pannes. Ce processus, grâce au recours à un ADL standardisé, permet en premier lieu la modélisation de ces systèmes tout en profitant des techniques d'analyse et de vérification offertes à ce stade.

Étant donné que les approches existantes reposent sur une réplication manuelle et explicite des composants engendrant un risque d'erreurs, une perte de temps et une complexité

de la charge du concepteur, nous avons proposé, dans un second lieu, une approche de modélisation et de gestion automatiques de la réplication des composants (Chapitre 6). Nous avons défini un ensemble de propriétés encapsulant les concepts de réplication dont les valeurs seront personnalisées par le concepteur. À partir de ces propriétés enrichissant le modèle, nous générons un modèle intermédiaire étendu par les répliques. Nous avons étendu la suite d'outils Ocarina par l'implantation d'un ensemble de règles de transformation que nous avons établi pour régir le processus de génération.

À partir du modèle généré, nous aidons le développeur à la génération du code fonctionnel en plus du code tolérant aux pannes. Visant la modularité du code et la simplicité de maintenance et de réutilisation, nous avons adopté la programmation orientée aspect pour la génération des deux types de code. Afin de rendre extensible et d'offrir la liberté des choix de plate-forme, nous avons proposé une approche de génération de code à la base de la démarche MDA (Chapitre 7). Nous avons proposé le passage par un modèle intermédiaire indépendant de la plate-forme qui donne à son tour naissance à une génération de code aspect spécifique à la plate-forme selon le choix du développeur.

Finalement, visant à profiter des avantages de la programmation orientée aspect dans le domaine temps-réel, nous avons proposé l'extension et l'adaptation d'un langage d'aspect pour le support des systèmes temps-réel et le respect de leurs contraintes. Puisque les systèmes temps-réel imposent l'interdiction de certaines constructions comme l'allocation dynamique de mémoire ou le polymorphisme, nous avons étudié le langage AspectAda, extension du langage Ada par les concepts d'aspects (Chapitre 8). L'étude de ce langage a été effectuée face à sa syntaxe, sa sémantique, son opération de tissage et de génération de code, sa bibliothèque de routines Runtime et enfin son compilateur tisseur. Nous avons testé et évalué le langage par rapport à chacun de ses points dans le but de s'assurer de l'absence de violation des contraintes temps-réel. Toutefois, cette version du langage a montré plusieurs lacunes sur tous les niveaux. Pour cette raison, nous avons proposé diverses solutions pour chacun de ses problèmes tout en respectant les contraintes temps-réel.

Pour démontrer la faisabilité de cette approche, nous avons enrichi la suite d'outils Ocarina qui manipule les modèles AADL. Nous l'avons tout d'abord étendu pour supporter notre approche de réplication basée sur l'extension des modèles AADL par les propriétés de réplication que nous avons définies. Puis, nous avons étendu cette suite d'outils, pour implanter les générateurs de code de l'annexe EMA vers le langage AO4AADL et du langage AO4AADL vers le langage d'aspect AspectAda. Ce dernier a été aussi l'objet d'une implantation d'un nouveau compilateur.

2 Conclusions

Pour valider les solutions que nous avons réalisées, nous avons mis en œuvre des études de cas pour valider leur adéquation au problématiques posées au début de ce mémoire.

L'avancement de nos travaux ainsi que les résultats satisfaisants des études de cas nous ont permis aussi de contribuer à la future version du langage AspectAda.

Plusieurs études de cas ont été réalisées durant notre travail de thèse. Il s'agit, pour la plupart, d'exemples d'applications temps-réel du domaine de l'aéronautique ou de la médecine proposés par des experts du domaine. Pour ce mémoire, nous avons repris un exemple classique d'application Ravenscar critique que nous avons étendu pour par les aspects pour tester le langage AspectAda et son compilateur développé dans le cadre de cette thèse. Nous avons adapté cet exemple, issu de [ABT04], pour le langage AspectAda et nous l'avons testé à l'aide d'un simulateur pour la plate-forme TSIM for LEON. Le chapitre 9 décrit en détails cette étude de cas. Il donne aussi les résultats de l'analyse statique d'ordonnançabilité en utilisant notre compilateur et langage AspectAda après extension de la version publiée.

Dans le même chapitre, nous avons choisi une autre étude de cas du domaine médical pour mettre en évidence nos contributions liées aux différentes étapes du processus de développement. Il s'agit d'un système médical temps-réel critique simple. C'est le système de couveuse d'enfants. Après la description du fonctionnement de base du système, nous l'avons modélisé avec le langage AADL décrivant l'architecture globale du système tout en mettant l'accent sur les différentes interactions entre les composants qui le constituent. Ce système à été par la suite étendu par EMA dans le but de donner le modèle d'erreur architectural tout en focalisant sur les types d'erreurs pouvant survenir, les manières de détection envisagées et les mécanismes de recouvrement mis en place. Dans le but de répliquer certains composants automatiquement, nous avons appliqué les propriétés *Replicaton_Properties* que nous avons définies pour générer automatiquement un modèle supportant la réplication. À partir de ce modèle généré, nous avons appliqué notre approche pour la génération de code fonctionnel et celui tolérant aux pannes à travers le passage par le langage intermédiaire AO4AADL.

Nous avons donné ensuite différents résultats sur les tailles des sources de l'application avant et après l'intégration de notre approche de réplication automatique. Ceci a permis de mettre en évidence le fait que l'application de notre approche de réplication en fonction des propriétés, que nous avons défini, peut diminuer le code écrit à la main d'une manière significative (jusqu'à 75 % lorsqu'il s'agit d'un nombre très important de composants à répliquer ou de répliques).

3 Perspectives

De nombreuses pistes de recherche peuvent être envisagées pour poursuivre ce travail.

Actuellement, les outils développés dans le cadre de cette thèse sont déployés sur un serveur de gestion de versions (*subversion*), accessible aux membres de l'équipe. Nous souhaitons à court terme déployer ces outils sur le site github `https://github.com/OpenAADL/ocarina` pour être utilisés par toute la communauté AADL.

La seconde perspective réside au niveau de l'évaluation de notre extension du langage AspectAda et du développement de son compilateur. Outres les études de cas mis en œuvre pour la validation de la compatibilité du langage avec les contraintes temps-réel, d'autres métriques d'évaluation pourront être utilisées pour la validation de notre extension comme les suite de tests de *Chidamber et Kemerer* [CK91](C&K). Cette suite offre un ensemble complet et validé des mesures pour quantifier la qualité du logiciel. Il sera aussi intéressant d'évaluer la qualité du logiciel par rapport aux différents attributs de la sûreté de fonctionnement à savoir la compréhensibilité, la réutilisabilité, la maintenabilité et la testabilité.

A moyen terme, une autre piste de travail serait la vérification formelle des transformations effectuées durant le cycle de développement. Pour garantir la cohérence entre le modèle et le code généré automatiquement, il est possible de vérifier formellement les règles de génération par rapport à certaines propriétés fonctionnelles ou non fonctionnelles. A ce niveau, nous pouvons profiter des transformations possibles des modèles AADL vers des formalismes formelles comme LNT [MZHJ15], un travail en cours dans notre équipe. Sachant que les transformations effectuées dans le cadre des travaux de [Rug08] concernent la première version du langage AADL, il serait intéressant d'adapter une telle transformation pour sa deuxième version pour rendre plus sûre la conception de la tolérance aux pannes des applications temps-réel distribuées avec AADL et simplifier sa vérification formelle.

Dans le cadre du processus de développement, nous avons présenté différentes phases permettant la production des systèmes temps-réel distribués intégrant les mécanismes de tolérance aux pannes tôt dans le cycle de développement. Pour la modélisation de la réplication, nous avons encapsulé les paramètres nécessaires dans un ensemble de propriétés. Parmi ces propriétés, le concepteur doit spécifier l'algorithme de consensus qui sera décrit à travers des fichiers sources ou des sous-programmes AADL. Dans ce contexte, nous envisageons développer des algorithmes de consensus standard comme services d'un intergiciel. Plus particulièrement, nous souhaitons, à long terme, étendre l'intergiciel PolyROB-HI à des fins de la tolérance aux pannes et plus généralement de la sûreté de fonctionnement. Visant la génération des systèmes temps-réel distribués tolérants aux pannes, nous pouvons ainsi profiter des services prédéfinis de l'intergiciel PolyORB-HI tout en bénéficiant des services de tolérance aux pannes facilitant ainsi la tâche du développeur et aussi offrant différentes implantations selon la plate-forme choisie.

Acronymes

AADL Architecture Analysis & Design Language. 6–8, 30, 35, 36, 43–46, 48, 52, 58, 59, 61, 63, 64, 71–89, 92–98, 100–103, 105–109, 113–121, 123–134, 146, 148, 150, 151, 154, 156, 166, 168–170

ADL Architecture Description Language. 7, 12, 28–30, 46, 51, 58, 60, 63–66, 68, 71, 73, 74, 167

AO4AADL Aspect Oriented Extension For AADL. 6, 88, 89, 95, 96, 98, 99, 121, 126–133, 146, 148, 154, 157, 166, 168, 169

AOSD Aspect Oriented Software Development. 30, 31

AT Acceptance Test. 49, 54

AUTOSAR AUTomotive Open System ARchitecture. 40, 43

BA Beahavioal Annex. 44, 52

CCM CORBA Component Model. 39, 46, 48

CFC Control Flow Checking. 49, 54

CORBA Common Object Request Broker Architecture. 46, 48, 50, 54, 57

CTMC Continued Time Markov chain. 46, 48

DRB Distributed Recovery Block. 24, 54

DSCFC Double Signature Control Flow Checking. 49

DSL Domain Specific Language. 38

DSPN Deterministic Stochastic Petri Nets. 39, 48

EAST-ADL Electronic Architecture and Software Technology Architecture Description Language. 36, 40–43, 46, 48, 64

EDC Error Detection and Correction. 49

EMA Error Model Annex. 6, 8, 44–46, 48, 59, 71, 83, 86, 94–96, 99, 100, 126–131, 168, 169

FMEA Failure Modes Effects Analysis. 42, 45, 46, 48

FT Fault Tree. 48

FT-CORBA Fault-Tolerant CORBA Services. 39, 50

FTA Fault Tree Analysis. 42, 46

GC Garbage Collector. 56

GSPN General Stochastic Petri Nets. 45, 48

HIP-HOPS Hierarchically Performed Hazard Origin and Propagation Studies. 41–43, 45, 46, 48, 64

MARTE Modeling and Analysis of Real-Time Embedded systems. 36–39, 43, 46, 64

MARTE-DAM MARTE-Dependability Analysis and Modeling. 36–39, 46, 48, 52, 64

MDA Model Driven Approach. 5–7, 12, 26, 27, 59, 63, 65, 68, 69, 72, 97, 121, 123, 126, 127, 133, 168

MDE Model Driven Engineering. 26

NMR N-Modular Programming. 24

NVP N-Version Programming. 24, 54

OMG Object Management Group. 4, 26, 36

PCP Priority Ceiling Protocol. 123

PIM Platform Independent Model. 27, 28

PSM Platform Specific Model. 28

QFTP Quality Of Service and Fault Tolerance Characteristics & Mechanisms. 36, 38, 39, 46, 48

RB Recovery Block. 24, 54

RTES Real-Time Embedded Systems. 36, 37

RTJEG Real-Time for Java Expert Group. 56

RTSJ Real-Time Specification for Java. 56, 95, 98, 124, 125, 128

SFT Static Fault Tree. 45, 48

SPT Schedulability, Performance and Time. 36, 37

TADL Timing Augmented Description Language. 42, 43

TRE Time Redundancy Execution. 49

TTRFR Triple Time Redundant Execution with Forward Recovery. 49

UML Unified Modelling Language. 27, 30, 36–40, 43, 46, 48, 51–53, 63, 64

WCET Worst Case Execution Time. 14, 79, 158

Bibliographie

[AAM09] Carzaniga Antonio, Gorla Alessandra, and Pezzè Mauro. Handling software faults with redundancy. In *Architecting Dependable Systems VI*, ADS'09, pages 148–171, Germany, 2009. Springer Berlin Heidelberg.

[ABT04] Burns Alan, Dobbing Brian, and Vardanega Tullio. Guide for the use of the ada ravenscar profile in high integrity systems. *ACM SIGAda Ada Letters*, 24(2) :1–74, 2004.

[ACD+06] Jean Arlat, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Jean-Claude Laprie, and David Powel. Tolérance aux fautes. Technical report, Encyclopédie de l'informatique et des systèmes d'information, Paris, France, 2006.

[ACMF00] Hany H. Ammar, Bojan Cukic, Ali Mili, and Cris Fuhrman. A comparative analysis of hardware and software fault tolerance : Impact on software reliability engineering. *Annals of Software Engineering*, 10(1) :103–150, November 2000.

[Ada12] ANSI. *Reference Manual for the Ada Programming Language*, February 2012. ANSI/MIL-STD 1815A.

[AK84] Algirdas Avizienis and John. P. J. Kelly. Fault tolerance by design diversity : Concepts and experiments. *IEEE Computer*, 17(8) :67–80, August 1984.

[AK11] Ruben Alexandersson and Johan Karlsson. Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks*, DSN'11, pages 303–314, Hong Kong, June 2011. IEEE.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1) :11–33, January 2004.

[ASB+08] Francisco Afonso, Carlos Silva, Nuno Brito, Sergio Montenegro, and Adriano Tavares. Aspect-oriented fault tolerance for real-time embedded systems. In *Proceedings of the 2008 AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 1–8, New York, USA, 2008. ACM.

[Aut09] Thomas Autret. Génération de code real-time java pour systèmes temps-réel.

Master's thesis, Université Pierre & Marie Curie, ParisVI, France, September 2009.

[Bar08] Jhon Barnes. *Safe and Secure Software, an invitation to Ada 2012*, volume 20150501. AdaCore, 2008.

[Bar12] John Barnes. *Spark : The Proven Approach to High Integrity Software*. Altran Praxis, http ://www.altran.co.uk, UK, 2012.

[BHF+12] Hand Blom, Lönn Henrik, Hagl Frank, Papadopoulos Yiannis, Reiser Mark-Oliver, Sjöstedt Carl-Johan, Chen De-Jiu, and Kolagari Ramin T. East-adl - an architecture description language for automotive software-intensive systems. Technical report, The EAST-ADL 2 Consortium, 2012.

[BMC+15] Alessio Bucaioni, Saad Mubeen, Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. Comparative evaluation of timing model extraction methodologies at east-adl design level. In *2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conference on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*, pages 1110–1115, New York, USA, August 2015. IEEE.

[BMCS15] Alessio Bucaioni, Saad Mubeen, Antonio Cicchetti, and Mikael Sjödin. Exploring timing model extractions at east-adl design-level using model transformations. In *2015 12th International Conference on Information Technology - New Generations (ITNG)*, pages 595–600. IEEE, April 2015.

[BMP11] Simona Bernardi, José Merseguer, and Dorina C. Petriu. A dependability profile within marte. *Software & Systems Modeling*, 10(3) :313–336, July 2011.

[BMP12] Simona Bernardi, José Merseguer, and Dorina C. Petriu. Dependability modeling and assessment in uml-based software development. *The Scientific World Journal*, 2012(5) :11, September 2012.

[Bou12] Rahma Bouaziz. Extension et adaptation d'un langage d'aspect pour les systèmes temps réel. Master's thesis, École Nationale d'Ingénieurs de Sfax, Tunisia, July 2012.

[Bro04] Alan W. Brown. Model driven architecture : Principles and practice. *Software and System Modeling*, 3(4) :314–327, December 2004.

[CDP+10] Alexandru Costan, Ciprian Dobre, Florin Pop, Catalin Leordeanu, and Valentin Cristea. A fault tolerance approach for distributed systems using monitoring based replication. In *2010 IEEE International Conference on Intelligent Computer Communication and Processing*, ICCP'10, pages 451–458, Cluj-Napoca, Romania, August 2010. IEEE.

[CFJ+10] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. The east-adl architecture description language for automotive embedded software. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 297–307, Germany, December 2010. Springer Berlin Heidelberg.

[CI99] Bonnet Christian and Demeure Isabelle. *Introduction aux systèmes temps réel*. Collection pédagogique de télécommunications. Hermès Science Publications, 1999.

[CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 197–211, Phoenix, Arizona, USA, November 1991. ACM.

[CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European Software Engineering Conference (ESEC)*, pages 88–98, Vienna, Austria, September 2001. ACM.

[Cle97] Paul C. Clements. Coming attractions in software architecture. In *1997. Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, pages 2–9. IEEE, April 1997.

[CMW+13] DeJiu Chen, Nidhal Mahmud, Martin Walker, Lei Feng, Henrik Lönn, and Yiannis Papadopoulos. Systems modeling with east-adl for fault tree analysis through hip-hops. *IFAC Proceedings Volumes*, 46(22) :91–96, 2013.

[CP04] Vittorio Cortellessa and Antonio Pompei. Towards a uml profile for qos : A contribution in the reliability domain. *SIGSOFT Softw. Eng. Notes*, 29(1) :197–206, January 2004.

[CS06] Thierry Coupaye and Jean-Bernard Stefani. Fractal component-based software engineering. In *Object-Oriented Technology. ECOOP 2006 Workshop Reader*, ECOOP'06, pages 117–129, Nantes, France, July 2006. Springer Berlin Heidelberg.

[CtK02] Karen Church and Geoff te Kraabe. *Future of Software Development*. IRM Press, 2002.

[DM05] Péter Domokos and István Majzik. Design and analysis of fault tolerant architectures by model weaving. In *Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE)*, pages 15–24, Heidelberg, Germany, October 2005. IEEE Computer Society.

[DSS98] Xavier Défago, André Schiper, and Nicole Sergent. Semi-passive replication. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, SRDS'98, pages 43–50, Indiana, USA, October 1998. IEEE Computer Society.

[DTT99] Anne-Marie Déplanche, Pierre-Yves Théaudière, and Yvon Trinquet. Implementing a semi-active replication strategy in chorus/classix, a distributed real-time executive. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, SRDS'99, pages 90–101, Lausanne, Switzerland, October 1999. IEEE Computer Society.

[Dub13] Elena Dubrova. Fundamentals of dependability. In *Fault-Tolerant Design*, pages 5–20. Springer, New York, March 2013.

[FBF+07] Ricardo Bedin França, Jean-Paul Bodeveix, Mamoun Filali, Jean-François Rolland, David Chemouil, and Dave Thomas. The aadl behaviour annex – experiments and roadmap. In *12th IEEE International Conference on Engineering Complex Computer Systems*, ICECCS'07, pages 377–382, Auckland, New Zealand, July 2007. IEEE Computer Society.

[FECA05] Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2005.

[GÖ9] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1) :1–26, 1999.

[GBZ13] Wafa Gabsi, Rahma Bouaziz, and Bechir Zalila. Towards an aspect oriented language compliant with real time constraints. In *2013 Workshops on Enabling Technologies : Infrastructure for Collaborative Enterprises*, pages 68–73, Hammamet, Tunisia, 2013. IEEE Computer Society.

[Gei08] Sandra Geisler. *TSIM2 Simulator User's Manual*, 2008.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. *Foundations of Component-based Systems*. Cambridge University Press, 2000.

[GSSP02] Andreas Gal, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. On aspect-orientation in distributed real-time dependable systems. In *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems*, WORDS'02, pages 261–267, San Diego, CA, USA, January 2002. IEEE Computer Society.

[GZJ16a] Wafa Gabsi, Bechir Zalila, and Mohamed Jmaiel. Aspectada : An aspect oriented extension of ada for real-time systems. In *15th IEEE/ACIS International Conference on Computer and Information Science*, ICIS'16, pages 1–6, Okayama, Japan, June 2016. IEEE Computer Society.

[GZJ16b] Wafa Gabsi, Bechir Zalila, and Mohamed Jmaiel. Extension of the ocarina tool suite to support reliable replication-based fault-tolerance. In *Proceedings*

21st Ada-Europe International Conference on Reliable Software Technologies, Ada-Europe'16, pages 129–144, Pisa,Italy, June 2016. Springer.

[Her03] Jack Herrington. *Code Generation in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[HH03] Kelly J. Hayhurst and C. Michael Holloway. Considering object oriented technology in aviation applications. In *The 22nd Digital Avionics Systems Conference*, DASC'03, pages 3–8. IEEE, October 2003.

[hHZ11] Jun hua Hu and Wei Zuo. Method for modeling and analysis real-time system dependability using aadl. In *International Conference on Electric Information and Control Engineering*, ICEICE'11, pages 3356–3359, Wuhan, China, April 2011. IEEE.

[HRL+08] Brahim Hamid, Ansgar Radermacher, Agnes Lanusse, Christophe Jouvray, Sébastien Gérard, and François Terrier. Designing fault-tolerant component based applications with a model driven approach. In *6th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, SEUS'08, pages 9–20, Anacarpi, Capri Island, Italy, October 2008. Springer.

[HRV+08] Brahim Hamid, Ansgar Radermacher, Patrick Vanuxeem, Agnes Lanusse, and Sebastien Gerard. A fault-tolerance framework for distributed component systems. In *34th Euromicro Conference on Software Engineering and Advanced Applications*, Euromicro-SEAA'08, pages 84–91, Parma, Italy, September 2008. IEEE Computer Society.

[HWS09] Kashif Hameed, Rob Williams, and Jim Smith. Aspect oriented software fault tolerance. In *Proceedings of the World Congress on Engineering*, WCE'09, pages 110–117, London, UK, July 2009. IEEE.

[HZ07] Jérôme Hugues and Bechir Zalila. PolyORB High Integrity User's Guide. Technical report, École Nationale Supérieure des Télécommunications, January 2007.

[JG00] Gosling James and Bollella Greg. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[JL11] Andreas Johnsen and Kristina Lundqvist. Developing dependable software-intensive systems : Aadl vs. east-adl. In *16th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe'16, pages 103–117, Edinburgh, UK, June 2011. Springer Berlin Heidelberg.

[JSKT11] Shahrokh Jalilian, Fatemeh Salar, Alireza Khani, and H. Kazimov Tofig. Using aop for developing fault tolerant software. In *5th International Conference on Application of Information and Communication Technologies*, ICAICT'11, pages 1–3, Azerbaijan, Baku, October 2011. IEEE.

[JVB07] Anjali Joshi, Steve Vestal, and Pam Binns. Automatic generation of static fault trees from aadl models. In *Workshop on Architecting Dependable Systems of The 37th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks*, pages 1–6, Edinburgh, UK, June 2007. IEEE.

[KEM+13] Eun-Young Kang, Eduard Paul Enoiu, Raluca Marinescu, Cristina Cerschi Seceleanu, Pierre-Yves Schobbens, and Paul Pettersson. A methodology for formal analysis and verification of east-adl models. *Reliability Engineering & System Safety*, 120 :127–138, 2013.

[KGHZ12] Fatma Krichen, Amal Ghorbel, Brahim Hamid, and Bechir Zalila. An mde-based approach for reconfigurable DRE systems. In *21st IEEE International Workshop on Enabling Technologies : Infrastructure for Collaborative Enterprises*, WETICE'12, pages 78–83, Toulouse, France, June 2012. IEEE Computer Society.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP'01, pages 327–353, Budapest, Hungary, 2001. Springer.

[kK95] kane Kim. The distributed recovery block scheme. In *Software Fault Tolerance*, pages 189–209, New York, USA, 1995. John Wiley & Sons Ltd.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming*, ECOOP'97, pages 220–242, Jyväskylä, Finland, June 1997. SpringerVerlag.

[KOS06] Philippe Kruchten, Henk Obbink, and Judith Stafford. The past, present, and future for software architecture. *IEEE Software*, 23(2) :22–30, 2006.

[KS07] Yusuf Bora Kartal and Ece G. Schmidt. An evaluation of aspect oriented programming for embedded real-time systems. In *22nd international symposium on Computer and information sciences*, ISCIS'07, pages 1–6, Ankara, Turkey, November 2007. IEEE.

[LA95] Chen Liming and Algirdas Avizienis. N-version programming : A fault-tolerance approach to reliability of software operation. In *Highlights from Twenty-Five Years., 25 International Symposium on Fault Tolerant Computing FTCS*, pages 113–119, Pasadena, California, June 1995. IEEE Comput. Soc. Press.

[LBS04] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice : On the combination of aop with generative programming in aspect c++. In *Third International Conference on Generative Programming and Component Engineering*, GPCE'04, pages 55–74, Vancouver, Canada, October 2004. Springer.

[LHFD13] Brian Larson, John Hatcliff, Kim Fowler, and Julien Delange. Illustrating the aadl error modeling annex (v.2) using a simple safety-critical medical device. In *ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT'13, pages 65–84, Pittsburgh, Pennsylvania, USA, 2013. ACM.

[LKZJ13] Sihem Loukil, Slim Kallel, Bechir Zalila, and Mohamed Jmaiel. Ao4aadl : Aspect oriented extension for aadl. *Central Europ. J. Computer Science*, 3(2) :43–68, June 2013.

[LL73] Chang L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1) :46–61, 1973.

[Lou10] Sihem Loukil. Extension d'un langage de description d'architecture pour la programmation orientée aspect. Master's thesis, École Nationale d'Ingénieurs de Sfax, Tunisia, July 2010.

[LRPK10] Gilles Lasnier, Thomas Robert, Laurent Pautet, and Fabrice Kordon. Behavioral Modular Description of Fault Tolerant Distributed Systems with AADL Behavioral Annex. In *10th international conference on New Technologies of Distributed Systems*, NOTERE'10, pages 17–24, Tozeur, Tunisia, 2010. IEEE.

[Luc96] David C. Luckham. Rapide : A language and toolset for simulation of distributed systems by partial orderings of events. Technical report, Stanford, CA, USA, 1996.

[LZPH09] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérôme Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In *14th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe'09, pages 237–250, Brest, France, June 2009. Springer.

[MBP+15] Zhibao Mian, Leonardo Bottaci, Yiannis Papadopoulos, Septavera Sharvia, and Nidhal Mahmud. Model transformation for multi-objective architecture optimisation of dependable systems. In *Dependability Problems of Complex Information Systems*, pages 91–110, Cham, 2015. Springer International Publishing.

[MFP06] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[MLM+13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Anthony Tang. What industry needs from architectural languages : A survey. *IEEE Transactions on Software Engineering*, 39(6) :869–891, June 2013.

[MS04] Javier Miranda and Edmond Schonberg. *GNAT : The GNU Ada Compiler*, 2004.

[MSH11] John W. McCormick, Frank Singhoff, and Jérôme Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*. Cambridge University Press, New York, USA, 2011.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, January 2000.

[MZHJ15] Hana Mkaouar, Bechir Zalila, Jérôme Hugues, and Mohamed Jmaiel. From AADL model to LNT specification. In *20th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe'15, pages 146–161, Madrid, Spain, June 2015. Springer.

[NDP+05] Priya Narasimhan, TA Dumitraş, Aaron M Paulos, Soila M Pertet, Carlos F Reverte, Joseph G Slember, and Deepti Srivastava. Mead : support for real-time fault-tolerant corba. *Concurrency and Computation : Practice and Experience*, 17(12) :1527–1545, October 2005.

[NF09] Dionisio De Niz and Peter. H. Feiler. Verification of replication architectures in aadl. In *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS'09, pages 365–370, Potsdam, Germany, June 2009. IEEE Computer Society.

[OMG05] OMG. UML Profile for Schedulability, Performance, and Time Specification. `http://www.omg.org/technology/documents/formal/schedulability.htm`, January 2005.

[OMG06] OMG. Corba component model specification version 4.0. OMG Technical Document formal/06-04-01, `http://www.omg.org/spec/CCM/4.0`, April 2006.

[OMG08] OMG. A UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded systems. `www.omgmarte.org/Documents/Specifications/08-06-09.pdf`, June 2008.

[OMG09] OMG. OMG Unified Modeling Language (OMG UML), Superstructure. `http://www.omg.org/spec/UML/2.2/Superstructure`, February 2009.

[PC05] Knut H. Pedersen and Constantinos Constantinides. Aspectada : aspect oriented programming for ada95. *ACM SIGAda Ada Letters*, 25(4) :79–92, December 2005.

[PDFS02] Renaud Pawlak, Laurence Duchien, Gerard Florin, and Lionel Seinturier. JAC : un framework pour la programmation orientée aspect en java. *Revue des Sciences et Technologies de l'Information - Série L'Objet*, 8(4) :145–168, 2002.

[PKJ11] Anshu Gupta PK Kapur, Hoang Pham and PC Jha. Software reliability assessment with or applications. pages 451–512, London, May 2011. Springer.

[PP12] Hu Ping and Li Peng. An aspect-based model for non-invasive fault tolerant software. In *International Conference on Industrial Control and Electronics Engineering*, ICICEE '12, pages 136–139. IEEE Computer Society, August 2012.

[Pua02] Isabelle Puaut. Real-time performance of dynamic memory allocation algorithms. In *14th Euromicro Conference on Real-Time Systems*, ECRTS'02, pages 41–49, Vienna, Austria, June 2002. IEEE Computer Society.

[PVW04] Luís Miguel Pinho, Francisco Vasques, and Andy Wellings. Replication Management in Reliable Real-Time Systems. *Real-Time Systems*, 26(3) :261–296, April 2004.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4) :40–52, October 1992.

[Qui03] Thomas Quinot. *Conception et réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables*. PhD thesis, Université Pierre et Marie Curie, Paris, France, March 2003.

[RKK08] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche. The adapt tool : From aadl architectural models to stochastic petri nets through model transformation. In *7th European Dependable Computing Conference*, EDCC'08, pages 85–90, Kaunas, Lithuania, May 2008. IEEE.

[RPM05] Alexandersson Ruben, Öhman Peter, and Ivarson Martin. Aspect oriented software implemented node level fault tolerance. In *Proceedings of the 9th IASTED International Conference on Software Engineering and Applications*, SEA'05, pages 56–74, Sweden, November 2005. IEEE.

[Rug08] Ana-Elena Rugina. *Dependability modeling and evaluation : from AADL to stochastic Petri nets*. PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, France, April 2008.

[Rus94] John Rushby. Critical system properties : Survey and taxonomy. *Reliability Engineering & System Safety*, 43(2) :189–219, 1994.

[SAE04] SAE. *Architecture Analysis & Design Language(AS5506)*, November 2004. available at `http://www.sae.org`.

[SAE06] SAE. *Architecture Analysis & Design Language Annex : Behavioral Annex*, 2006.

[SAE12] SAE. *Architecture Analysis & Design Language(AS5506B)*, September 2012. available at `http://www.sae.org`.

[SAE15] SAE. *Architecture Analysis & Design Language Annex E : Error Model Annex (AS5506/1A)*, September 2015.

[SLNM04] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar : a flexible real time scheduling framework. In *ACM SIGAda International Conference on Ada : The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies*, SIGAda'04, pages 1–8, Atlanta, GA, USA, November 2004. ACM.

[SN04] Diana Szentiványi and Simin Nadjm-Tehrani. Aspects for improvement of performance in fault-tolerant software. In *10th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC'04, pages 283–291, Papeete, Tahiti, March 2004. IEEE Computer Society.

[SPC+14] Sharvia Septavera, Yiannis Papadopoulos, DeJiu Chen, Martin Walker, Wenjing Yuan, and Henrik Lönn. Enhancing the east-adl error model with hip-hops semantics. *Athens Journal of Technology & Engineering*, 1(2) :119–136, June 2014.

[SRG95] Lui Sha, Ragunathan Rajkumar, and Michael Gagliardl. A software architecture for dependable and evolvable industrial computing systems. Technical report, July 1995.

[STH01] Fernando Sánchez, Miguel Toro, and José Luis Herrero. Fault tolerance as an aspect using jreplica. In *The Eighth IEEE Workshop on Future Trends of Distributed Computing Systems*, FTDCS'01, pages 201–207, Bologna, Italy, November 2001. IEEE Computer Society.

[SY99] Junichi Suzuki and Yoshikazu Yamamoto. Extending uml with aspects : Aspect support in the design phase. In *ECOOP'99 Workshops on Obejct-Oriented Technology*, ECOOP'99 Workshop Reader, pages 299–300, Lisbon, Portugal, June 1999. Springer.

[TCB04] Shiu Lun Tsang, Siobhán Clarke, and Elisa Baniassad. An evaluation of aspect-oriented programming for java-based real-time systems development. In *7th International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC'04, pages 291–300, Vienna, Austria, May 2004. IEEE computer Society.

[VZH06] Thomas Vergnaud, Bechir Zalila, and Jérôme Hugues. Ocarina : a Compiler for the AADL. Technical report, École Nationale Supérieure des Télécommunications, Paris, France, June 2006.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. International Computer Science Series. Addison-Wesley, Redwood City, CA, USA, 1995.

[WRTP+13] Martin Walker, Mark-Oliver Reiser, Sara Tucci-Piergiovanni, Yiannis Papadopoulos, Henrik Lönn, Chokri Mraidha, David Parker, DeJiu Chen, and David Servat. Automatic optimisation of system architectures using east-adl. *Journal of Systems and Software*, 86(10) :2467–2487, October 2013.

[Zal08] Bechir Zalila. *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*. PhD thesis, École Nationale Supérieure des Télécommunications, Paris, France, November 2008.

[ZPH08] Bechir Zalila, Laurent Pautet, and Jérôme Hugues. Towards automatic middleware generation. In *11th International Symposium on Object-Oriented Real-Time*

Distributed Computing, ISORC'08, pages 221–228, Orlando, Florida, USA, May 2008. IEEE computer Society.

[ZWL16] Quan Zhang, Shihai Wang, and Bin Liu. Approach for integrated modular avionics reconfiguration modelling and reliability analysis based on aadl. *Institution of Engineering and Technology Software*, 10(1) :18–25, February 2016.

Liste des publications

— **Publication dans une revue**

1. **IJACCS'2016**

   Wafa Gabsi, Bechir Zalila and Jérôme Hugues. A development process for the design, implementation and code generation of fault tolerant reconfigurable real-time systems. In *International Journal of Autonomous and Adaptive Communications Systems*, vol. 9, Nos. 3/4, pp.269–287.

— **Chapitre de livre**

1. **SCI'2015**

   Wafa Gabsi and Bechir Zalila. Towards a Model Level Replication Technique for Fault Tolerant Systems Using AADL. *Springer's Series : Studies in Computational Intelligence*, Springer, 2015.

— **Publications dans des conférences internationales**

1. Wafa Gabsi, Bechir Zalila and Mohamed Jmaiel. Development of a parser for the AADL Error Model Annex. In 16th IEEE/ACIS International Conference on Computer and Information Science, ICIS, May, 2017, Wuhan, China, IEEE Computer Society.
2. Wafa Gabsi, Bechir Zalila and Mohamed Jmaiel. AspectAda : an Aspect Oriented Extension of Ada for Real-Time Systems. In 15th IEEE/ACIS International Conference on Computer and Information Science, ICIS, June, 2016, Okayama, Japan, IEEE Computer Society.
3. Wafa Gabsi, Bechir Zalila and Mohamed Jmaiel. Extension of the Ocarina Tool Suite to Support Reliable Replication-Based Fault-Tolerance. In 21st International Conference on *Reliable Software Technologies, Ada-Europe*, June, 2016, Pisa, Italy, Springer.
4. Wafa Gabsi, Bechir Zalila and Mohamed Jmaiel. EMA2AOP : From the AADL Error Model Annex to Aspect Language Towards Fault Tolerant Systems. In 14th

IEEE/ACIS International Conference on *Software Engineering Research, Management and Applications*, June, 2016, Towson University, Maryland, USA, IEEE Computer Society.

5. Wafa Gabsi and Bechir Zalila. Towards a Model Level Replication Technique for Fault Tolerant Systems Using AADL. In 16th IEEE/ACIS International Conference on *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, June, 2015, Takamatsu, Japan, IEEE Computer Society.
6. Wafa Gabsi, Rahma Bouaziz, and Bechir Zalila. Towards an Aspect Oriented Language Compliant with Real Time Constraints. In 22nd IEEE International Workshops on *Enabling Technologies : Infrastructures for Collaborative Enterprises, WETICE*, Third track on *Adaptive and Reconfigurable Service-oriented and Component-based Applications and Architectures - AROSA* 2013, Hammamet, Tunisia, 2013. IEEE Computer Society.
7. Wafa Gabsi and Bechir Zalila. Fault Tolerance for Distributed Real Time Dynamically Reconfigurable Systems from Modeling to Implementation. In 22nd IEEE International Workshops on *Enabling Technologies : Infrastructures for Collaborative Enterprises, WETICE* 2013, Third track on *Adaptive and Reconfigurable Service-oriented and Component-based Applications and Architectures - AROSA* 2013, Hammamet, Tunisia, 2013. IEEE Computer Society.

République Tunisienne
Ministère de l'Enseignement Supérieur,
de la Recherche Scientifique
et de la Technologie

Université de Sfax
École Nationale d'Ingénieurs de Sfax

Ecole Doctorale
Sciences et Technologies

Thèse de *DOCTORAT*
Nom du Doctorat

N° d'ordre: 101/2016

Tolérance aux pannes pour les systèmes temps-réel distribués: de la modélisation à l'implantation

Wafa GABSI MASMOUDI

الخلاصة: نقترح في هذه الأطروحة عملية التنمية للأنظمة الآنية الموزعة و المتسامحة في الخطأ مع الفصل بين مختلف الاهتمامات ابتداء من مستوى التصميم. لهذا، تستند العملية على ثلاث مراحل: الأولى هي تصميم النظام الأساسي الذي يحدد الهندسة المعمارية التفصيلية. المرحلة الثانية هي إثراء هذا النموذج بمفاهيم الخطأ والتسامح، وخاصة أنواع الأخطاء التي يمكن أن تحدث، والنسخ المتماثل، ووسائل الكشف والإصلاح. المرحلة الأخيرة هي التوليد التلقائي ليس فقط للتعليمة البرمجية الوظيفية ولكن أيضاً المتسامحة في الخطأ و ذلك استنادا إلى القواعد المنشأة. وأخيرا، للتمتع بايجابيات البرمجة الجانبية المنحى ، قمنا بتوسيع وتكييف جانبا من جوانب اللغة لدعم أنظمة الوقت الحقيقي الحرجة.

Résumé : Nous proposons dans le cadre de cette thèse un processus de développement pour les systèmes temps-réel distribués tolérants aux pannes tout en séparant les différentes préoccupations dés le niveau modélisation. Pour cela, ce processus est basé sur trois phases : La première consiste à modéliser le système de base tout en décrivant son architecture détaillée. La deuxième phase consiste à enrichir ce modèle par les concepts de tolérance aux pannes, en particulier les types de pannes pourront survenir, la réplication, les moyens de détection et de recouvrement. La dernière phase, consiste à générer automatiquement, non seulement le code fonctionnel, mais aussi celui tolérant aux pannes à la base des règles de génération définies. Enfin, pour profiter des apports de la programmation orientée aspect, nous avons étendu et adapté un langage d'aspect pour supporter les systèmes temps-réel critiques.

Abstract: We propose in this thesis a development process for real-time fault tolerant distributed systems while separating the various concerns since the modeling level. For this, the process is based on three steps: The first is to model the core system while outlining its detailed architecture. The second step is to enrich this model with the concepts of fault tolerance, especially the types of failures that can occur, replication, means of detection and repair. The final step is to automatically generate not only the functional code but also fault tolerant code thanks to defined generation rules. Finally, to benefit from advantages of aspect-oriented programming, we have expanded and adapted an aspect language to support critical real-time systems.

المفاتيح: خطأ التسامح، الأنظمة الآنية، عملية التنمية، النسخ المتماثل، توليد التعليمات البرمجية، البرمجة الجانبية المنحى

Mots clés: Tolérance aux pannes, systèmes temps-réel, processus de développement, réplication, génération de code, programmation orientée aspect.

Key-words: Fault tolerance, real-time system, development process, replication, code generation, aspect oriented programming.