université de BORDEAUX

Département de formation doctorale en informatique

École doctorale EDMI Bordeaux

Approche pour le Développement de Logiciels Intégrant des Concepts de Qualité de Service

THÈSE

soutenue le 12 Février 2014

pour l'obtention du grade de

Docteur de l'Université de Bordeaux

(Spécialité Informatique)

par

Stéphanie GATTI épouse GEOFFROY

Jury

Président :	Marc PHALIPPOU	Directeur de l'ENSEIRB-MATMECA, Bordeaux
Rapporteurs :	Jean-Michel BRUEL	Professeur à l'IRIT, Toulouse
	Jean-Charles FABRE	Professeur à l'INP-ENSEEIHT, Toulouse
Examinateurs :	Emilie BALLAND	Chargée de Recherche à Inria Bordeaux Sud-Ouest
	Charles CONSEL	Professeur à l'Université de Bordeaux
	Isabelle DEMEURE	Professeur à Télécom ParisTech
	Jean JOURDAN	Directeur Technique et Produits à Thales Avionics, Toulouse

A mon fils Nathan

REMERCIEMENTS

Ce manuscrit de thèse marque la fin d'une expérience riche d'enseignement. Il concrétise l'aboutissement d'une période de travail intense et est le fruit d'une étroite collaboration entre le laboratoire Inria Bordeaux Sud-Ouest et Thales Systèmes Aéroportés à Pessac. Ce travail n'aurait pu se concrétiser sans l'aide précieuse des personnes que je tiens ici à remercier aussi bien pour leur encadrement que pour leur soutien.

Je tiens tout d'abord à remercier chaleureusement mes deux directeurs de thèse, Charles CONSEL et Jean JOURDAN, de m'avoir accueillie respectivement au sein de l'équipe Phœnix et de la Direction Technique de CCM&F. Tous deux m'ont accompagnée durant ces années à la fois dans le domaine de la Recherche et dans son application au Domaine Industriel. Je les remercie pour leurs conseils avisés et leur rigueur qui m'ont toujours encouragée à avancer.

Je remercie Jean-Charles FABRE et Jean-Michel BRUEL d'avoir accepté d'être rapporteurs de ma thèse. Leurs relectures et questions avisées durant la soutenance m'ont apporté un regard nouveau sur ma thèse et fait entrevoir d'autres perspectives de réflexion.

Je remercie Isabelle DEMEURE d'avoir accepté de faire partie de mon jury de thèse. Je la remercie pour sa minutieuse relecture et ses questions et remarques lors de la soutenance, qui ont permis d'ouvrir le champ des investigations futures.

Mes remerciements vont également à Marc PHALIPPOU pour avoir accepté d'être Président de mon jury de thèse, pour sa relecture de mon manuscrit de thèse et ses remarques avisées.

Merci à ma co-directrice de thèse, Emilie BALLAND, pour ses conseils et son aide précieuse lors de la rédaction de publications.

Je remercie également Franck AIME pour son encadrement et ses précieux conseils ; j'ai beaucoup appris à ses côtés. Un merci également à Stéphane TREUCHOT, pour l'accueil réservé lors de mon arrivée à Thales et ses conseils.

Je tiens également à remercier tous les membres de l'équipe Phœnix, avec qui j'ai beaucoup apprécié travailler ; ils ont contri-

bué à faire régner une ambiance conviviale tout au long de ma thèse, et plus particulièrement Quentin, Julien, PengFei, Henner, Damien, Young-Joo, Nicolas, Milan, Lucile, Charles, Paul et Camille. Merci également au service support pour son aide au quotidien, et en particulier à Chrystel, Catherine, Sylvie et Jean-Christophe pour leur disponibilité et leur gentillesse.

Un très grand merci à toute ma famille, à mes parents toujours présents et attentifs, qui ont toujours cru en moi, à mes frères : Sébastien, Christophe et Nicolas, pour leur soutien inconditionnel. Merci à Papi Jean, le niçois, d'avoir fait le voyage pour voir sa petite-fille soutenir sa thèse.

Je partage cette thèse avec mon époux Rémi, qui m'a toujours encouragée à donner le meilleur pendant ces années de thèse.

Enfin, je dédie cette thèse à mon fils Nathan, ma plus belle motivation.

ABSTRACT

A STEP-WISE APPROACH FOR INTEGRATING QOS THROUGHOUT SOFTWARE DEVELOPMENT PROCESS

In critical domains such as avionics, railways or automotive, to certify a system, it is required to demonstrate that it achieves its function, with respect to specified timing requirements. Indeed, longer-than-predicted function computing can make data erroneous, leading potentially to endanger people lives. Today, most approaches propose to ensure these Quality of Service requirements at platform level, *e.g.,* through deterministic bandwidth, static time slots allocation and predefined scheduling. These constraints ensure applications can't overpass allocated time slots; applications are then fed with requirements decoupled to their functionality. However, it shall be possible to certify timing requirements, dedicated to an application. Hence, guarantees at platform-level are not sufficient anymore. It should be possible to take into account these requirements from the stage of application design. Today, most of existing approaches in this domain, focus on supporting QoS at individual stages of the software development process, preventing requirements traceability.

This thesis proposes a design-driven approach to supporting QoS throughout software development process, integrated in a tool-based methodology, namely DiaSuite. The QoS extension enriches the DiaSpec design language, with the capability to instantiate QoS requirements onto software components. A runtime execution support to monitoring these timing requirements, is then generated, directly from the specification. This thesis uniformly integrates timing concepts with error ones, around DiaSuite methodology, to propose a supervision layer that could lead to application reconfiguration in case of QoS contract violation. Contributions of this thesis are evaluated through respect of coherence and conformance critera, illustrated through a case study in avionics.

KEYWORDS: Software Architecture, Domain-Specific Language, Generative Programming, Quality of Service

RÉSUMÉ

Dans les domaines critiques tels que l'avionique, le ferroviaire ou encore l'automobile, il faut, afin de pouvoir certifier un système, démontrer qu'il réalise la fonction pour laquelle il a été conçu, selon des exigences temporelles spécifiées. En effet, un rendu temporel trop long peut rendre des données erronées, et ainsi mettre en danger la sûreté des personnes. Aujourd'hui, la plupart des approches proposent d'assurer ces exigences de Qualité de service au niveau des couches basses, *e.g.*, au travers d'une bande passante déterministe, d'allocation statique d'intervalles de temps, et d'un ordonnancement prédéfini. Ces contraintes assurent que les applications ne peuvent dépasser le temps d'exécution alloué ; les applications récupèrent de ce fait des exigences qui sont découplées de leur fonctionnalité. En revanche, il faut aussi pouvoir certifier des exigences temporelles spécifiques à une application. De là, les garanties au niveau des couches basses ne sont plus suffisantes. Il faudrait pouvoir prendre en compte ces exigences dès la phase de conception des applications. Aujourd'hui, la plupart des approches existant dans ce domaine se concentrent sur le support de QoS à des phases isolées du processus de développement logiciel, empêchant la traçabilité des exigences.

Cette thèse propose une approche dirigée par la conception pour supporter les exigences de QoS tout au long du processus de développement logiciel, intégrée dans une méthodologie outillée, appelée DiaSuite. L'extension de QoS enrichit le langage de conception DiaSpec avec la capacité d'instancier les exigences de QoS sur les composants logiciels. Un support de surveillance à l'exécution de ces exigences temporelles est ensuite généré, directement à partir de la spécification. Cette thèse intègre uniformément les concepts temporels avec les concepts de gestion d'erreurs, au travers de la méthodologie DiaSuite, afin de proposer une couche de supervision qui puisse effectuer une reconfiguration applicative, dans le cas de violation de contrat de QoS. Les contributions de cette thèse sont évaluées au regard du respect des critères de cohérence et de conformité, illustrés au travers d'une étude de cas dans le domaine avionique.

MOTS CLÉS : Architecture Logicielle, Langage Dédié, Programmation Générative, Qualité de Service

LISTE DES PUBLICATIONS

Les travaux discutés dans cette thèse ont fait l'objet de publications, à la fois académiques et industrielles.

CONFÉRENCES INTERNATIONALES

- « A Step-wise Approach For Integrating QoS throughout Software Development, » dans *FASE'11 : Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering*, 2011, Stéphanie Gatti, Emilie Balland et Charles Consel [50]

- « Incremental Functional Certification For Avionic Functions Reuse & Evolution, » dans *DASC'12 : Proceedings of the 31st AIAA/IEEE Digital Avionics System Conference*, 2012, Stéphanie Gatti, Franck Aimé, Stéphane Treuchot et Jean Jourdan [51]

- « Design-driven Development of Dependable Applications : A Case Study In Avionics, » dans *PECCS'13 : Proceedings of the 3rd International Conference on Pervasive and Embedded Computing and Communication Systems*, 2013, Quentin Enard, Stéphanie Gatti, Emilie Balland, Young-Joo Moon, Charles Consel et Julien Bruneau [43]

- « Incremental Functional Certification, » dans *Proceedings of the Avionics Europe 2013 Conference for the Commercial & Defence Aerospace Industrie*, 2013, Stéphanie Gatti, Franck Aimé, Stéphane Treuchot et Jean Jourdan [52]

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES LISTINGS

1 INTRODUCTION

Lors du développement de systèmes critiques temps-réel, dans les domaines tels que l'avionique, l'automobile ou encore le ferroviaire, il est nécessaire de prendre en compte les exigences fonctionnelles, afin de pouvoir assurer que de tels systèmes fournissent la fonction qu'ils sont censés rendre. En particulier, dans le domaine avionique, le cadre de la certification impose la prise en compte d'exigences haut-niveau, à raffiner et tracer tout au long du cycle de développement [38].

En outre, de tels systèmes doivent également respecter des exigences non-fonctionnelles strictes, de temps ou encore de sûreté de fonctionnement. Ces exigences représentent en effet la *qualité* requise d'un système, dans le sens où elles se réfèrent à des contraintes sur la façon dont le système implémente et délivre sa fonctionnalité (*e.g.,* performance, fiabilité, sécurité) [99]. Par exemple, l'assurance doit être apportée que le rendu de la fonction par les systèmes s'effectue dans un intervalle de temps borné.

Aujourd'hui, ces exigences sont majoritairement prises en charge au niveau des couches basses, *i.e.,* système d'exploitation, matériel, technologies de systèmes distribués, indépendamment de toute application, *e.g.,* via des contraintes d'ordonnancement.

Par exemple, dans le domaine avionique, le standard ARINC 653 définit un système d'exploitation temps-réel fournissant un ordonnancement déterministe des applications [6], assurant ainsi une équité dans l'exécution des applications. Un autre exemple dans le domaine avionique, est le standard ARINC 664 qui définit le réseau AFDX, fournissant une qualité de service déterministe pour la communication de données [7]. Dans ce domaine, et de façon plus globale, dans les domaines des systèmes embarqués critiques, les exigences de Qualité de Service (ou "Quality of Service", QoS) sont ainsi garanties au niveau des couches basses. Ces garanties facilitent, certes, la certification, mais ne sont pas suffisantes pour assurer des exigences temporelles spécifiques à une application.

Le problème résultant est que, lorsque l'on adresse la conception d'un tel système à partir d'exigences de très haut niveau, ces garanties au niveau de la plateforme ne sont pas suffisantes. En effet, la réalisation d'une fonction débute par une phase très haut-niveau de capture des exigences, où des

exigences temporelles sont déposées sur la chaîne fonctionnelle de réalisation de cette fonction. Le cycle de développement logiciel classique se poursuit ensuite avec la modélisation de la fonction lors de la phase de conception, puis vient le développement propre de la fonction, suivi par des phases de test et de déploiement. Ainsi, de même que pour le support et le raffinement des exigences fonctionnelles tout au long du développement d'un système, il faut pouvoir également supporter le raffinement des exigences non-fonctionnelles tout au long du cycle de développement. Le support d'exigences de QoS dès les premières phases de développement permet ensuite d'affiner la granularité d'exigences à déposer sur les artefacts impliqués, en particulier, les composants logiciels.

Actuellement, il existe de nombreux langages de spécification pour déclarer des exigences de QoS au niveau architectural [1]. Initialement, ces langages étaient purement contemplatifs. Leur but était en effet de pouvoir spécifier les exigences non-fonctionnelles afin d'enrichir une spécification fonctionnelle. Plusieurs approches récentes fournissent également du support pour la gestion d'aspects spécifiques, comme la vérification de cohérence [31], la prédiction de propriétés [71] ou encore la surveillance à l'exécution [57]. En revanche, ces approches sont généralement dédiées à une phase particulière du développement. Cet aspect conduit à une perte de traçabilité des exigences, *i.e.*, la capacité à suivre les exigences de QoS tout au long du cycle de développement est perdue. Or, la traçabilité des exigences, fonctionnelles et non-fonctionnelles est requise dans les domaines critiques de l'avionique (certification civile [38]), de l'automobile (standard AUTOSAR [47]) et du ferroviaire. Cette traçabilité est généralement réalisée à la main, et est donc très coûteuse. Il est donc nécessaire de pouvoir proposer une méthodologie qui la supporte, et qui soit moins fastidieuse et sujette à erreur que la réalisation manuelle de matrices de conformité d'exigences [75].

La traçabilité fonctionnelle est généralement assurée par des méthodologies de développement systématique qui guident les acteurs depuis l'analyse des exigences jusqu'au déploiement du système [102]. De façon similaire, la QoS devrait être entièrement intégrée dans le processus de développement, car c'est un aspect transverse [70]. Il faut donc pouvoir proposer une méthodologie de développement qui supporte la prise en compte d'exigences de QoS, tout en respectant les objectifs suivants. D'une part, cette méthodologie doit pouvoir garantir la traçabilité des exigences de QoS, c'est-à-dire la capacité de pouvoir suivre leur propagation, depuis leur naissance tout au

long de leur raffinement lors du cycle de développement logiciel. D'autre part, du fait du couplage inhérent existant entre les aspects fonctionnels et non-fonctionnels d'une application, ainsi qu'entre les aspects non-fonctionnels, il est critique d'assurer leur cohérence globale afin d'éviter des impacts non prévisibles. En effet, les exigences non-fonctionnelles peuvent impacter les caractéristiques fonctionnelles d'un système. Par exemple, des mécanismes de tolérance aux fautes peuvent déteriorer de façon significative la performance d'une application. Généralement, ces problèmes sont détectés lors des phases finales du développement, conduisant à une augmentation des coûts de développement [5].

Pour répondre à cette problématique, la thèse présentée dans ce manuscrit propose une approche de gestion des exigences de qualité de service temporelle dirigée par la conception [50], intégrée dans une méthodologie outillée appelée DiaSuite [28]. Notre thèse repose ainsi sur un processus de développement intégré centré autour du paradigme conceptuel de *Sense, Compute, Control* [99]. Comme le démontre Shaw, l'usage d'un paradigme spécifique fournit un cadre conceptuel, conduisant à un processus d'ingénierie plus discipliné et guidant le processus de vérification [92]. Une application SCC est une application qui interagit avec un environnement physique. De telles applications sont typiquement présentes dans les domaines tels que l'Avionique, la Robotique ou encore la Domotique. Dans cette thèse, nous présentons notre approche [50], son intégration avec d'autres aspects non-fonctionnels tels que la tolérance aux fautes [80], et une étude de cas sur le développement et la validation d'un système avionique concret [22, 43].

1.1 CONTRIBUTIONS

Les contributions de cette thèse sont les suivantes.

Une approche de développement pas à pas de QoS dédiée aux systèmes SCC. Notre approche propose une gestion systématique des exigences de QoS, en guidant pas à pas la prise en compte de ces exigences dans le cycle de développement logiciel. Cette approche intégrée est dédiée aux systèmes ayant une boucle de contrôle avec l'environnement extérieur, reposant sur un patron architectural particulier. Ce dernier permet d'améliorer la conception de tels systèmes en restreignant la syntaxe du langage selon les concepts du domaine. Ces contraintes de langage offrent une capacité plus fine à offrir du support. Notamment, notre approche offre un support dédié de programmation et de

tests pour les aspects non-fonctionnels de QoS.

Bien que l'approche présentée dans ce chapitre soit appliquée à la performance temporelle, elle peut être généralisée à d'autres propriétés non-fonctionnelles, comme le temps CPU ou la consommation mémoire.

La traçabilité des exigences est requise par le processus de certification. Dans notre approche, la traçabilité est assurée par une propagation systématique des contraintes dérivées des déclarations de QoS et appliquées à chaque étape du développement, par génération de programme.

Mise en œuvre de la méthodologie dans une suite d'outils. Notre approche a été intégrée au sein de DiaSuite, une méthodologie de développement outillée dédiée aux systèmes SCC [27]. La méthodologie DiaSuite est basée sur un langage de conception dédié qui a été enrichi dans notre approche avec des propriétés de performance temporelle. Cette extension non-fonctionnelle a été utilisée afin d'offrir du support de vérification et de programmation à chaque étape du cycle de développement.

Validation dans le domaine avionique. Notre approche a été validée dans le cadre d'une étude de cas portant sur le développement d'une application concrète de guidage. Cette application devait respecter des exigences fonctionnelles et non-fonctionnelles fortes. L'étude de cas a démontré que notre approche peut effectivement guider le processus de certification, avec la démonstration de respect d'objectifs de traçabilité des exigences et de cohérence entre les exigences.

1.2 ORGANISATION DU MANUSCRIT

Ce document est organisé comme suit.

Le **Chapitre 2** présente un état de l'art sur les modèles à composants, ainsi que le support de qualité de service temporelle offert dans les modèles actuels, en présentant leurs restrictions.

Le **Chapitre 3** présente notre approche de développement guidée par la conception intégrant les aspects de QoS. Elle montre comment s'inscrit notre contribution dans la méthodologie outillée DiaSuite.

Le **Chapitre 4** détaille l'utilisation couplée de notre support de QoS avec d'autres dimensions non-fonctionnelles, telles que la gestion d'erreurs. Cette combinaison vise à proposer une supervision de l'application et des méthodes de recouvrement en cas de non-respect des exigences de QoS.

Le **Chapitre 5** présente la validation de notre approche dans le domaine avionique, au travers d'une étude de cas portant sur le développement d'une application de guidage avionique concrète, couplée à un simulateur de vol.

Enfin, le **Chapitre 6** conclut en rappelant les principes de l'approche présentée tout au long de ce manuscrit. Le **Chapitre 7** propose une généralisation de notre approche et le **Chapitre 8** présente des perspectives futures pour cette thèse.

Première partie

CONTEXTE

2

SUPPORT DE QUALITÉ DE SERVICE DANS LE DÉVELOPPEMENT LOGICIEL

Les exigences non-fonctionnelles de Qualité de Service ("Quality of Service", QoS) telles que la performance, constituent un ensemble d'exigences que doivent satisfaire les logiciels, en plus de leur fonctionnalité primaire. En effet, comme le présentent Vogel *et al.*, ces exigences déterminent la qualité attendue d'un logiciel, afin d'assurer sa fonctionnalité [101]. Parmi ces exigences, nous pouvons citer la Qualité de Service Temporelle, sur laquelle se concentre notre thèse. Une telle QoS est en effet critique dans les domaines temps-réel tels que l'avionique ou l'automobile et requise pour la certification de ces applications [38, 47].

Ce chapitre est constitué de quatre parties. Après une brève introduction sur les différentes catégories de QoS, nous présentons les approches qui se concentrent sur la gestion de QoS au niveau de la conception. Nous nous concentrons ensuite sur les approches qui offrent du support de QoS durant l'implémentation d'un logiciel complexe. Puis nous discutons des approches dirigées par la conception, s'appuyant sur une description du logiciel afin de faciliter le processus de développement de ce logiciel au travers du respect de ses exigences fonctionnelles et de QoS. Enfin, la dernière partie conclut sur les besoins pour notre approche.

2.1 PRÉAMBULE

La norme ISO/IEC 13236 X.641 présente un découpage de la QoS en sept catégories : Temps, Cohérence, Capacité, Intégrité, Sûreté de fonctionnement, Sécurité, Fiabilité [61]. Cependant, certaines de ces catégories sont généralement regroupées. Ainsi, Sabata *et al.* présentent une classification des QoS selon les trois catégories suivantes [89] : Performance, Fiabilité et Sécurité. Parmi les aspects de Performance, les éléments suivants sont dénotés :

- Temps (temps de réponse, synchronisation, ordonnancement,...)
- Consommation de ressources (CPU, mémoire,...)

Parmi ces exigences, la QoS temporelle est aujourd'hui requise dans de nombreux domaines critiques où la démonstration du

déterminisme associée à l'envoi des données doit être apportée, par exemple dans l'avionique [38].

Dans cette thèse, nous nous intéressons ainsi à l'intégration de concepts de QoS temporelle dans le développement de logiciels. Cette intégration que nous proposons se veut déclarative, c'est-à-dire offrant du support tout au long du cycle de développement, dans le respect de la conformité des exigences de QoS entre chaque phase du cycle de développement, et dans le respect de la cohérence avec les exigences fonctionnelles et non-fonctionnelles que doit supporter le logiciel.

Dans ce cadre, la section suivante présente les approches se concentrant sur la phase de conception du logiciel.

2.2 CONCEPTION

2.2.1 *Généralités*

Une architecture logicielle décrit la structure et le comportement d'un système logiciel ainsi que les entités non-logicielles avec lesquelles le système est interfacé. Ainsi, dans une architecture logicielle, un système est représenté comme un ensemble d'éléments logiciels liés par un ensemble de connecteurs. Les architectures logicielles ont considérablement évolué, depuis leurs considérations en tant que "boîtes et lignes" accompagnées d'explications informelles vers l'émergence d'**architectures en composants**, **architectures orientées services**, ou encore l'apparition de la notion de **programmation orientée objet** [93]. Dans sa description des propriétés apportées par les architectures dans le cadre du développement logiciel, Garlan insiste sur la compréhension qu'elles offrent d'un système, et les moyens d'analyse qu'elles permettent de mettre en œuvre pour la vérification de tels systèmes [49].

Un langage de description d'architectures ("Architecture Description Language", ADL) permet la description et la modélisation d'architectures logicielles.

Medvidovic *et al.* indiquent qu'un ADL doit explicitement modéliser les notions de définition d'une architecture logicielle [79] :

- **Composant** : Unité de calcul ou de stockage de données. Dans la définition de Szyperski *et al.*, le composant est vu comme un élément dit "boîte noire" qui cache son implémentation à l'environnement extérieur, et rend compte de sa

spécification au travers d'interfaces décrivant les méthodes / services / fonctionnalités qu'il fournit [97] ;
- **Connecteur** : Composante architecturale utilisée afin de modéliser, d'une part, les interactions entre composants, et, d'autre part, les règles qui régissent ces interactions ;
- **Configuration architecturale** : Graphes connectés de composants et connecteurs qui décrivent la structure architecturale.

Un exemple de modèles à composants est CORBA Component Model (CCM) [103]. Dans ce modèle, l'approche "composant-container" est introduite. Le container sert à fournir l'environnement d'exécution aux composants. Il crée et gère les composants à l'exécution, c'est-à-dire qu'il est responsable de leur cycle de vie (démarrage, réinitialisation, arrêt,...). Aussi, un container a deux interfaces, afin de représenter l'interface de programmation (API).

L'interface *interne* représente le cadre de travail utilisé par le développeur du composant. Il contient, d'une part, la représentation du contrat entre le composant et son container, et, d'autre part, un ensemble de services standardisés. Ces services offrent la capacité pour un composant à être hébergé par un autre container offrant a minima le même ensemble de services que le container hôte initial. Par exemple, un composant requérant des services techniques (e.g., trace) et hébergé par un container peut migrer à la condition que cet autre container offre également les services techniques requis par le composant, alors même que l'implémentation de ces services techniques diffère.

L'interface *externe* est l'interface disponible du point de vue d'un composant client. Elle peut être définie comme sa spécification, fournissant une abstraction de l'implémentation des fonctionnalités qu'il rend. L'interface constitue une vue contractuelle entre le développeur du composant et l'utilisateur des fonctionnalités rendues par ce composant.
De même, la spécification du comportement du composant peut être contractualisée :
- Invariant : exigences globales que le composant doit maintenir tout au long de son exécution ;
- Pré-conditions : exigences requises sur les données d'entrée du composant ;
- Post-conditions : exigences que satisfait le composant sur ses données en sortie.

Un certain nombre d'ADLs ont été développés par différentes équipes de recherche. Parmi eux, Rapide est un langage orienté objet qui s'adresse à la spécification d'architectures de systèmes distribués et basé sur les événements [77]. De tels systèmes consistent en un ensemble de composants qui s'exécutent

indépendamment les uns des autres, mais sous une exigence de temps. Les protocoles d'interaction entre composants peuvent être spécifiés à l'aide de réseaux de Petri. Rapide offre donc une simulation et analyse comportementale d'architectures de systèmes à une phase anticipée du processus de développement du système (dite d'*early-validation*). L'*early-validation* a pour but de valider au plus tôt du développement logiciel, des propriétés déposées sur le logiciel, *e.g.*, la consistance des exigences temporelles. De façon similaire, Darwin est un langage qui se concentre sur la description d'architectures distribuées [78]. S'il sépare le code fonctionnel du code de communication, il n'offre en revanche pas de possibilité de configurer la technologie de communication.

Wright est un ADL basé sur la description formelle du comportement abstrait de composants et connecteurs architecturaux [3]. Cet ADL fournit une capacité de description des architectures logicielles à l'aide de graphes hiérarchiques de composants et connecteurs. Wright ne possède pas de support outillé, tel qu'un environnement d'exécution ; il s'agit d'un pur langage formel pour la spécification d'architectures.

2.2.2 *Prise en compte de la QoS*

La gestion de QoS peut être effectuée via un langage dédié de spécification au niveau architectural. Il en existe de nombreux qui sont contemplatifs, comme l'exprime Aagedal dans sa thèse [1]. S'ils n'apportent pas de support à proprement parler, ces langages sont principalement utilisés afin de documenter la partie non-fonctionnelle des applications.

La déclaration de concepts de QoS à haut niveau d'abstraction peut également être utilisée dans le cadre de l'*early-validation* d'une application enrichie de concepts de QoS. xADL, déclaré comme "ADL hautement extensible basé sur XML" [30], s'inscrit dans cette optique. Il se caractérise par son support outillé fournissant une capacité de description fonctionnelle et temporelle de l'application, au travers de deux mécanismes. La prescription architecturale fournit un modèle de conception du temps qui est utilisé pour l'instanciation de l'architecture, tandis que la description architecturale fournit une description de l'état d'exécution du système. Ces mécanismes offrent une capacité de prédiction de performance sur le système.

Defour *et al.* se concentrent également sur la validation d'une application à la conception. Ils proposent une méthode permettant de générer des exigences numériques à partir d'exigences temporelles spécifiées à l'aide du langage QoSCL, qui est une extension non-fonctionnelle de UML 2.0 [31]. Ces exigences sont utilisées à la fois pour vérifier la compatibilité entre les exigences et l'architecture logicielle proposée, mais également pour prédire la QoS que l'on obtient à partir de la spécification d'un nombre d'instances pour chaque composant. Dans un cadre similaire, AlTurki *et al.* s'intéressent à la vérification de différentes propriétés temps-réel. Leur approche repose sur un modèle de réécriture temps-réel à l'aide du moteur Maude, qui prend en entrée un langage de spécification des exigences temporelles [4]. Le modèle applicatif spécifié est ensuite utilisé pour la vérification de ces exigences. De façon similaire, Bertolino *et al.* proposent une approche de prédiction de performance de l'application, basée sur l'assemblage de composants existants [16].

Egalement dans le domaine de la conception et de l'*early-validation* des exigences de QoS temporelle, Krogmann *et al.* ont mis en place un outil de prédiction de performance couplé à leur modèle à composants PCM ("Palladio Component Model") [72]. Ce modèle permet aux architectes de choisir entre différentes conceptions architecturales, avec la connaissance du rendu probabilistique de performance temporelle associé à chacun d'entre eux. Les exigences de performance prises en compte dans ce modèle sont de différents types et se concentrent sur les caractéristiques inhérentes aux couches basses du système. Elles sont en effet exprimées en termes d'accès aux ressources matérielles. Un exemple d'exigences que l'on peut spécifier dans cet outil est le nombre d'accès au CPU.

Dans le cadre des systèmes temps-réel embarqués également, Fredriksson *et al.* proposent un *framework* de conception pour faire levier sur les exigences non-fonctionnelles (telles que le temps ou la consommation mémoire) afin de construire des composants de niveau système dédiés au contrôle de ces exigences [46]. Cette approche leur permet de prédire le comportement attendu du système composé, à la fois au niveau fonctionnel (rendu de la fonction) et non-fonctionnel (qualité de la fonction). Doose *et al.* formalisent les systèmes temps-réel comme un ensemble de fonctionnalités reliées au travers de liens de communication temporisés [40]. Ils peuvent ainsi effectuer des vérifications sur la cohérence temporelle du système complet en utilisant des techniques de *model-checking*. Carcenac *et al.* s'intéressent également à la validation de systèmes

temps-réel selon une spécification incrémentale des exigences non-fonctionnelles [25]. Leur approche repose sur une abstraction du réseau de communication entre composants au travers de canaux point-à-point et sur la démonstration d'un ensemble d'obligations sur cette abstraction.

Ces approches se concentrent presque exclusivement sur la phase de validation au plus tôt des systèmes à partir de leur spécification fonctionnelle et non-fonctionnelle. Elles offrent en effet du support de QoS, mais se limitent à la phase de conception dans le processus de développement logiciel. En revanche, il reste à pouvoir traiter le support des exigences non-fonctionnelles de QoS depuis les phases amont de spécification applicative, jusqu'au code, afin d'assurer leur traçabilité tout au long du cycle de développement logiciel.

D'autres approches proposent un support d'intégration de concepts de QoS directement au niveau de l'implémentation.

2.3 IMPLÉMENTATION

L'implémentation d'un logiciel doté de capacités de QoS est une tâche difficile qui requiert de bien séparer les attendus fonctionnels, représentant ce que le logiciel doit faire, des attendus non-fonctionnels, représentant les exigences déposées sur la façon dont le logiciel réalise sa fonction. Parmi les approches se concentrant sur un support de QoS à l'implémentation, nous pouvons tout d'abord présenter la Programmation par Aspects, qui a l'avantage de supporter une séparation des préoccupations entre exigences fonctionnelles et non-fonctionnelles.

2.3.1 *Programmation par Aspects*

La programmation par aspects ("Aspect-Oriented Programming", AOP) est une technique de programmation qui vise à séparer la partie fonctionnelle de la partie non-fonctionnelle d'une application afin de permettre une amélioration de la maintenance et du support de réutilisabilité. Pour cela, elle consiste à déclarer des *aspects*, correspondant aux différentes facettes d'une application, qui, lorsqu'ils sont assemblés, forment l'application finale [35]. Cette approche de programmation en déclarant les briques de l'application, avant de les assembler, est particulièrement appropriée aux systèmes devant prendre en compte des exigences telles que le temps réel [17].

Parmi les langages de programmation par aspects, AspectJ est une extension orientée aspect de Java, basée sur l'identification de points de jonction ("Joint Points") correspondant à des endroits dans le code de description de propriétés non-fonctionnelles, sous la forme d'aspects [67]. Lors de la compilation, ces aspects sont ensuite tissés dans le code, c'est-à-dire insérés dans les points de jonction spécifiés. Par exemple, Duclos *et al.* proposent de spécifier des exigences de QoS en tant qu'aspects, afin d'offrir ensuite du support de surveillance de ces exigences à l'exécution [41].

La phase de test est une phase importante à considérer dans le développement de systèmes critiques. Elle permet de s'assurer, d'une part, de la réalisation des exigences fonctionnelles et non-fonctionnelles au sein d'un composant, et d'autre part, de leur robustesse. Afin de supporter la validation de composants critiques, Bruel *et al.* proposent une approche d'intégration de tests au sein de composants, en utilisant les aspects afin de réduire la duplication de code pour la gestion de différents environnements d'exécution [18]. Leur approche considère la notion de *Built-in-Test component* caractérisée comme la spécification d'un composant, enrichie d'interfaces de tests pour la détection d'erreurs internes et externes au composant. A partir de contrats spécifiés entre les composants, les *BIT-components* permettent de valider le respect de ce contrat en les surveillant à l'exécution. De cette façon, l'approche couvre également des aspects de surveillance de Qualité de Service à l'exécution.

2.3.2 *Intergiciels*

Les intergiciels permettent de supporter le développement d'applications distribuées, en faisant la relation entre les couches basses (système d'exploitation, *hardware*, protocoles de communication) et les couches hautes (applications). Ce niveau intermédiaire permet aux intergiciels d'abstraire l'hétérogénéité potentielle des couches basses, vis-à-vis des couches hautes. Les intergiciels peuvent être en charge de la réalisation de différentes tâches, telles que la mesure de performance ou encore la mise en œuvre de nouveaux services. Parmi les intergiciels les plus connus, la spécification CORBA 1.0 est la référence de l'OMG (*Object Management Group*).

CORBA (*Common Object Request Broker Architecture*) est un intergiciel offrant une représentation de services distants pour les applications distribuées [87]. Plusieurs approches visent à enrichir cet intergiciel avec des aspects temps-réel.

Par exemple, l'approche de Zhang *et al.* consiste à enrichir CORBA à l'aide d'aspects afin de permettre la prise en compte des différentes exigences non-fonctionnelles de l'application, au travers de mécanismes de composition architecturale pouvant évoluer [105]. Demeure *et al.* présentent une approche de spécification d'exigences de QoS temporelles, couplée à l'intergiciel POLKA étendant CORBA, permettant un ordonnancement automatique des applications, dans le but de respecter ces exigences [32, 33]. Les exigences de QoS supportées sont des délais entre événements, ou encore la périodicité d'un événement, représentés sous forme d'équations. A partir de cette spécification, POLKA traduit les équations de délais de QoS en des *deadlines* de tâches et des temps de *release*, afin d'ordonnancer les invocations de méthodes. Lorsque les ressources nécessaires à l'application sont disponibles (CPU, mémoire, débit), alors l'ordonnancement garantit la tenue des exigences de QoS.

Dans le même esprit, Harrison *et al.* proposent une approche de conception temps-réel d'une application avionique traitant des exigences de performance requises dans ce domaine [58]. Leur approche se base sur une extension des services CORBA avec l'apport d'une capacité de définition et de support périodique du temps. Harrison *et al.* montrent ensuite comment utiliser les techniques "orientées objet" afin de mettre en place des exigences d'ordonnancement pour la tenue des exigences de QoS temporelles.

Haverkamp *et al.* proposent un intergiciel dédié au domaine avionique, destiné à isoler le logiciel applicatif d'une part, et le réseau de système distribué sous-jacent d'autre part, afin d'optimiser les capacités de réutilisation du logiciel [59]. Leur intergiciel est basé sur CORBA, enrichi de concepts de partitionnement spatio-temporel, qui permet de garantir que tout applicatif hébergé, possède un espace mémoire et un *slot* de temps qui lui sont alloués et garantis strictement : toute occurrence d'erreurs pouvant survenir sur un applicatif sera détectée et confinée au sein de l'applicatif, donc sans effet sur les autres applicatifs hébergés. Ce mécanisme de partitionnement permet de supporter la cohabitation de logiciels applicatifs de différents niveaux de criticité sur une même plateforme. Dans le domaine critique de l'avionique, cette capacité est un pré-requis à la certification incrémentale des applicatifs [39]. En outre, la Qualité de Service supportée est typiquement la minimisation du délai de délivrance des données, avec le pré-requis d'un système d'exploitation temps-réel. Au niveau des couches basses également, RT-CORBA intègre des exigences de QoS pour la gestion des ressources CPU, mémoire et réseau [90].

Dans le contexte de la tolérance aux fautes, Fabre *et al.* proposent une approche réflexive pour l'implémentation de systèmes tolérants aux fautes en CORBA, basée sur l'utilisation d'un framework pour le développement d'applications distribuées FRIENDS (*Flexible and Reusable Implementation Environment for your Next Dependable System*) [44, 68]. Leur approche repose sur la description d'un système construit à partir de l'intégration de composants et enrichi avec des mécanismes de tolérance aux fautes, qui spécifient les exigences attendues. Des exemples de tolérance aux fautes couvrent la consistance des données, QoS attendue et performance du système. A partir de cette base, leur approche permet de déterminer l'impact d'hypothèses déposées sur les couches basses, sur les fonctionnalités haut-niveau fournies par les entités du système, afin de proposer une adaptation du système en cohérence.

2.3.3 *Langages dédiés*

Par opposition aux langages généralistes (*General Purpose Languages*, GPL) tels que Java, les langages dédiés (*Domain-Specific Languages*, DSL) ciblent un domaine particulier. La sémantique des DSLs est ainsi issue des exigences du domaine.

Si un DSL est moins expressif qu'un GPL, la motivation principale de l'apparition et de l'utilisation des DSLs est leur gain en termes d'abstraction du domaine [95], ainsi que leur facilité d'utilisation par rapport aux GPLs dans un domaine d'application donné [81]. Mernik *et al.* présentent en effet l'équivalence qui peut être établie entre un DSL et un GPL disposant d'une bibliothèque particulière (*Application Programming Interface*, API) [81].

De nombreux DSLs existent, couvrant de larges domaines. Par exemple, dans le domaine des systèmes réactifs, le langage formel LUSTRE (*LUcid Synchrone Temps-REel*) est un langage déclaratif pour la programmation de systèmes synchrones [55, 56].

LUSTRE s'adresse à la spécification de systèmes réactifs qui interagissent en continu avec l'environnement, tels que les systèmes de contrôle ou de surveillance, ou encore le *hardware*. La sémantique du langage se base sur les notions de flots de données, au travers de la caractérisation de variables comme des séquences temporisées de valeurs de certains types, possédant leur horloge. LUSTRE supporte la spécification d'expressions fonctionnelles couplées à la spécification temporelle d'applications. Ce langage offre en effet quatre opérateurs dits "temporels", afin de caractériser les exigences mises en place sur

les flots de données. Un programme LUSTRE est représenté sous la forme d'un système d'équations avec des variables possédant leurs propres horloges. De ce fait, LUSTRE permet le développement de programmes temporels déterministes. En revanche, le faible niveau d'abstraction offert par ce langage rend la tâche d'implémentation d'autant plus compliquée que la complexité des systèmes à modéliser croît.

Ces approches de support de QoS sont dédiées aux systèmes temps-réel et offrent du support à l'exécution, tel que de la surveillance. Seulement, elles n'offrent pas de capacité de description haut-niveau des systèmes et des exigences qui leur sont associées.

La section suivante présente ainsi des approches dirigées par la conception pour répondre à cette problématique.

2.4 SUPPORT DE QOS TOUT AU LONG DU CYCLE DE DÉVELOPPEMENT

Afin de permettre la traçabilité des exigences, les approches dirigées par la conception adressent la gestion des exigences non-fonctionnelles de QoS au travers d'un support offert à chaque étape du cycle de développement logiciel.

Dans cette section, nous nous concentrons tout d'abord sur les approches qui reposent sur des ADLs, avant de présenter les approches dirigées par les modèles, qui sont enrichies avec des exigences de QoS.

2.4.1 *Approches reposant sur des ADLs*

Dans le cadre des ADLs, ArchJava s'adresse également à la phase d'implémentation. C'est une extension de Java dédiée à l'intégration de spécifications d'architectures logicielles dans le code d'implémentation Java [2]. Ce langage fournit des garanties telles que l'intégrité de communication : les composants ne peuvent communiquer qu'avec les composants auxquels ils sont reliés dans la spécification architecturale. Cette propriété est nécessaire à respecter dans les domaine où l'on doit apporter la preuve que le logiciel implémenté est conforme à sa spécification.

Les Architectures Orientées Services (*Service-Oriented Architecture*, SOA) permettent aux concepteurs d'applications de se concentrer sur les fonctionnalités à développer, *i.e.*, les services, tout en offrant un couplage faible entre les composants

producteurs et consommateurs de services [84]. Cette approche permet de composer les différents services afin d'offrir une vue plus haut-niveau du service global rendu par l'ensemble des composants mis en jeu. Dans ce domaine, la QoS constitue une composante de facto dans la description des services, afin de spécifier des exigences de performance, de sécurité ou encore d'intégrité. La composition de services supporte également la composition des QoS associées. Les travaux de Parra *et al.* reposant sur la plateforme SCA **FraSCAti**, proposent de garantir les contraintes de QoS spécifées sur l'application, à l'aide de mécanismes de reconfiguration applicative à l'exécution [85, 98].

L'approche de Yu *et al.* s'intéresse à la sélection statique et dynamique de services, à partir de la fonctionnalité offerte et des exigences de performance déposées [104]. Dans le domaine des services web, le langage WSLA permet de spécifier des exigences de performance temporelle au travers de l'utilisation de mécanismes de *Service Level Agreement* (SLA) [65] et de générer du support de surveillance à l'exécution [66]. Cardellini *et al.* se concentrent sur l'aspect de spécification de QoS, afin de proposer une adaptation à l'exécution de ces architectures pour garantir les QoS spécifiées [26]. Leur approche propose de spécifier différents mécanismes d'adaptation, afin d'offrir une flexibilité dans le contexte d'environnements d'exploitation différents. La méthodologie proposée se base sur un modèle comportemental du système qui est rafraîchi lors de l'exécution grâce à un support de surveillance.

Dans le domaine des systèmes temps-réel embarqués, de nombreuses approches se concentrent sur les couches basses afin d'adresser la prise en compte des aspects non-fonctionnels au niveau de l'implémentation. Par exemple, l'approche de Kopetz *et al.* sur les architectures basées sur des déclenchements temporels (*Time-Triggered Architectures*, TTA) fournit une infrastructure de calcul pour la conception et l'implémentation de systèmes embarqués [69]. Cette approche se base sur la découpe d'une application en un ensemble de noeuds, régis par une horloge globale. La méthodologie proposée se décompose en deux phases : conception de l'architecture et implémentation des composants. La première phase permet la spécification d'interactions entre les systèmes distribués, à partir des interfaces qu'ils fournissent et des plages temporelles associées. La communication entre composants est réalisée à l'aide d'événements temporisés. La seconde phase permet l'implémentation de ces composants à partir de la description des interfaces réalisée dans la première phase. Cette approche permet d'offrir du support de QoS temporelle à la conception et à l'implémentation, en préservant l'isolation temporelle entre les composants. Elle vise également à

supporter la composabilité d'applications TTA et la réutilisation de composants pré-validés au sein de TTA. En revanche, cette approche se concentre sur la caractérisation "couches basses" d'une application, ce qui rend complexe le développement de systèmes à partir de leurs exigences fonctionnelles haut-niveau.

Dans le domaine des périphériques embarqués, l'approche de Genßler *et al.* permet la génération de support d'ordonnancement basé sur les déclarations de QoS temporelle [54]. Leur approche se base sur le modèle à composants PECOS (*PErvasive COmponent Systems*) centré sur les données. Dans le modèle PECOS, les interfaces des composants consistent uniquement en des ports de données, et les interactions entre composants s'effectuent via des connexions entre ces ports. PECOS fournit au concepteur un environnement pour la spécification, la composition, la gestion de configuration et le déploiement de systèmes embarqués, réalisés à partir de composants logiciels. Les exigences de QoS peuvent être spécifiées sur les composants, par exemple sous la forme de temps maximal alloué à l'exécution de la tâche. La vérification de l'ordonnancement est ensuite possible sur la composition de composants.

Egalement, l'approche de Robert *et al.* permet la génération de support de surveillance à partir des exigences non-fonctionnelles, représentées sous la forme de transitions exceptionnelles dans des automates temporisés [88].

Dans le domaine des ADL, AADL (*Architecture Analysis & Design Language*) est un standard dédié aux systèmes temps-réel embarqués [45]. AADL est issu de MetaH, un ADL dédié aux systèmes avioniques [100]. MetaH se base sur un processus de développement dit *Bottom-Up*, à partir des caractéristiques temps-réel fines des systèmes, supportées par une description de leurs flots de contrôle et de données, enrichi d'exigences non-fonctionnelles, telles que les exigences temporelles. MetaH permet la génération de code à partir de la spécification et offre également un outil permettant d'effectuer des analyses sur l'architecture, telles que des analyses d'ordonnancement. Alors que la plupart des ADLs fournissent peu ou pas de support à l'implémentation, l'environnement Ocarina permet la génération d'un support de programmation dédié à une description AADL [60].

Au sein de AADL, il est possible de spécifier des systèmes logiciels en utilisant les constructeurs dédiés, tels que les composants (*component*) ou les ports de communication entre composants (*port*). Leur déploiement sur une plateforme d'exécution peut

également être spécifié, à l'aide de tâches (*thread*), processus (*process*), mémoire (*memory*). La couche non-fonctionnelle vient enrichir la spécification fonctionnelle. En effet, les exigences non-fonctionnelles du système peuvent être spécifiées avec l'ajout de propriétés sur les constructeurs du langage, telles que la période d'un *thread* par exemple. Si AADL adresse les systèmes temps-réel, étant orienté *Bottom-Up*, il offre peu de support pour guider véritablement la conception architecturale du système à partir du cahier des charges recensant les exigences haut niveau.

AADL est un standard. De ce fait, de nombreux axes de recherche ont pour but de l'enrichir avec du support d'analyse et de développement. Par exemple, Dissaux *et al.* travaillent sur une approche offrant une analyse de performance d'architectures temps-réel [37]. Ils proposent un ensemble de patrons de conception (*design patterns*) pour modéliser les concepts inhérents aux systèmes temps-réel, tels que la synchronisation de tâches. Pour chaque patron, ils listent ensuite un ensemble de critères de performance, tels que la borne temporelle d'une tâche pour l'accès à une donnée, qui peuvent être vérifiés avec un outil d'analyse de performance [94]. L'approche de Gaudel *et al.* leur permet de vérifier automatiquement l'ordonnancement d'une application [53]. De récents travaux proposent une évolution du standard AADL par couplage avec un autre standard SystemC offrant des capacités de description dynamique aux modèles statiques de AADL [73, 74]. Leur approche propose de combiner AADL et SystemC afin de représenter et simuler la consommation de ressources matérielles à différents degrés de granularité.

Ces approches mises en place au sein d'AADL se concentrent principalement sur les aspects d'implémentation et de déploiement. Dans le domaine des systèmes temps-réel, ces approches ont l'avantage de permettre une granularité fine pour la spécification de exigences de QoS temporelles, et d'offrir une garantie sur la tenue de ces exigences à l'exécution. En revanche, il n'y a peu de support fourni pour guider le développement de tels systèmes, depuis les exigences de haut-niveau.

2.4.2 *Approches MDE enrichies avec des exigences de QoS*

Une tendance actuelle forte pour le développement de systèmes sont les approches d'Ingénierie Dirigée par les Modèles (*Model-Driven Engineering*, MDE). De telles approches consistent en la description d'un système au travers de modèles haut niveau, suivie de la génération d'un modèle cible à partir de ces modèles. Le modèle cible est généralement textuel : code ou

squelette de code.

Dans le domaine des MDEs, certaines approches se concentrent sur les systèmes temps-réel, en proposant de gérer les exigences non-fonctionnelles. Burmester *et al.* proposent une approche de développement dédiée aux systèmes mécatroniques [1] [23]. Cette approche est basée sur une extension d'UML dédiée aux systèmes temps-réel. Afin d'autoriser la vérification formelle du système mécatronique complet, les auteurs proposent de développer une bibliothèque de patrons de coordination qui définissent des rôles de composants spécifiques, ainsi que leurs interactions et des exigences temps-réel. Les composants de l'application sont ainsi construits à partir de la bibliothèque de patrons avec la spécification de leurs rôles enrichie de détails sur leurs comportements. L'approche proposée présente également un support outillé pour la spécification, la vérification et la synthèse de code source en tant que *plug-in* pour la suite outillée Fujaba [24]. L'utilisation de patrons de coordination peut être assimilée à un paradigme qui guide la conception de systèmes mécatroniques, au travers des exigences fonctionnelles et temporelles qui les qualifient.

Un autre exemple de méthodologie de développement dédiée aux systèmes temps-réel est SCADE (*Safety Critical Application Development Environment*) [36]. SCADE est basé sur le langage formel LUSTRE présenté en Section 2.3.3.

SCADE repose ainsi sur des machines à état hiérarchiques pour la spécification d'applications sûres. Une application est spécifiée en utilisant ces machines à état, permettant ainsi la vérification de cohérence comportementale de l'application dès la phase de conception. En outre, le paradigme synchrone assure par construction le déterminisme d'une application, générée à partir des modèles. Cette approche offre une abstraction des spécificités de temps physique, en permettant la vérification des propriétés temps-réel dès la spécification.

Dans le domaine avionique, parmi les approches dirigées par la conception qui offrent la possibilité de générer le code directement à partir de la spécification, il faut distinguer celles dont le générateur est qualifié, comme c'est le cas pour SCADE, offrant ainsi la démonstration que le code est conforme aux exigences, et celles dont le code généré en sortie doit être qualifié à chaque regénération. Dans ce dernier cas, il est alors important de fournir

1. Un système mécatronique couvre à la fois les aspects de la mécanique, de l'électronique et de l'informatique temps-réel

des preuves attestant de la conformité du code par rapport aux exigences. Par exemple, l'approche de Denney *et al.* est basée sur une annotation automatique du code généré avec des formules de vérification de sûreté de fonctionnement sous la forme de pré/post conditions et d'invariants [34].

2.5 CONCLUSION

De nombreuses approches proposent de supporter la gestion de QoS à certaines phases du développement logiciel. Or, comme nous avons pu le constater dans cet état de l'art, spécifier les exigences non-fonctionnelles de QoS uniquement au niveau architectural n'est pas suffisant. Il faut pouvoir prendre en compte les exigences de QoS à chaque étape du développement afin d'assurer la traçabilité de ces exigences, c'est-à-dire le suivi des exigences depuis leur spécification haut niveau, tout au long de leur raffinement dans le cycle de développement logiciel. En outre, les approches dirigées par la conception aident à abstraire les exigences physiques et guident le développement.

Par ailleurs, une approche généraliste n'offre pas suffisamment de support pour aider et guider le développement d'une application à partir de ses exigences fonctionnelles. En effet, les approches MDE généralistes offrent certes la capacité de spécifier différentes vues d'un même système, mais la problématique est de pouvoir vérifier la cohérence entre ces vues ainsi que la conformité des artefacts entre les différentes phases de développement. Une approche dédiée permet de fournir davantage de support pour les phases de conception, implémentation et de test. En particulier, l'usage d'un paradigme offre une capacité de conception applicative restreinte au domaine, guidant davantage le concepteur d'une application.

Bruel *et al.* détournent un ensemble de modèles existants permettant la gestion d'exigences non-fonctionnelles. Au travers de leur présentation, ils démontrent toute l'importance d'assurer une cohérence entre les modèles utilisés par différentes équipes, pour représenter les différentes facettes d'une application [19].

Le niveau d'abstraction du langage de spécification influe également sur les capacités de description fonctionnelle des applications. Un plus haut niveau d'abstraction permettra d'évaluer au plus tôt les QoS attendues au niveau d'une application et de les raffiner tout au long du cycle de développement logiciel. En outre, la conduction de ce raffinement jusqu'au déploiement

offre des capacités de vérification plus fines.

Enfin, comme l'ont observé Koziolek *et al.*, il est crucial d'identifier clairement les acteurs et les interactions entre le développement fonctionnel et la couche non-fonctionnelle [70].

L'approche à mettre en œuvre pour l'intégration de concepts de QoS dans le développement de logiciels doit donc permettre la spécification d'exigences de QoS, adaptée à la granularité des artefacts considérés. Il faut ainsi pouvoir offrir une capacité de description haut-niveau des exigences de QoS, et également pouvoir définir certains détails de déploiement afin de pouvoir exprimer finement les exigences de QoS. Une telle approche doit couvrir le cycle de développement complet, et garantir la traçabilité des exigences spécifiées, intégrée de façon unifiée avec les exigences fonctionnelles.

Pour atteindre ce but, nous proposons une approche unifiée qui intègre la QoS dans le processus de développement complet. Notre approche se base sur une méthodologie de développement outillée, DiaSuite, reposant sur le paradigme *Sense* / *Compute* / *Control*.

Deuxième partie

APPROCHE PROPOSÉE

3

APPROCHE

Pour répondre à la problématique présentée dans le Chapitre 2, nous proposons une approche pas-à-pas de développement de logiciels intégrant des concepts de QoS. Ce travail a été présenté à la conférence FASE (*International Conference on Fundamental Approaches to Software Engineering*) [50].

Notre approche est intégrée à la méthodologie outillée DiaSuite, et se concentre sur la capacité de prise en compte des exigences non-fonctionnelles de QoS tout au long du processus de développement logiciel, assurant la traçabilité de ces exigences.

Dans ce chapite, nous présentons donc tout d'abord l'approche DiaSuite et l'exemple que nous allons dérouler tout au long de ce chapitre. Puis nous décrivons notre approche intégrant la QoS et illustrée sur l'exemple présenté précédemment.

3.1 APPROCHE DIASUITE

DiaSuite est une méthodologie outillée de développement d'applications SCC, dirigée par la conception [27, 28].

3.1.1 *Paradigme SCC*

Le paradigme SCC provient du patron architectural *Sense / Compute / Control* mis en évidence par Taylor *et al.* [99]. Ce patron convient parfaitement à la description d'applications qui interagissent avec un environnement externe. De telles applications sont typiques des domaines tels que l'avionique, la robotique ou encore la domotique. La Figure 1 présente le paradigme SCC.

Dans la méthodologie de développement outillée DiaSuite, DiaSpec est le langage de conception de l'architecture, basé sur le paradigme SCC.

Comme l'illustre la Figure 2, le patron architectural SCC traduit les deux facettes de DiaSpec : la description de la

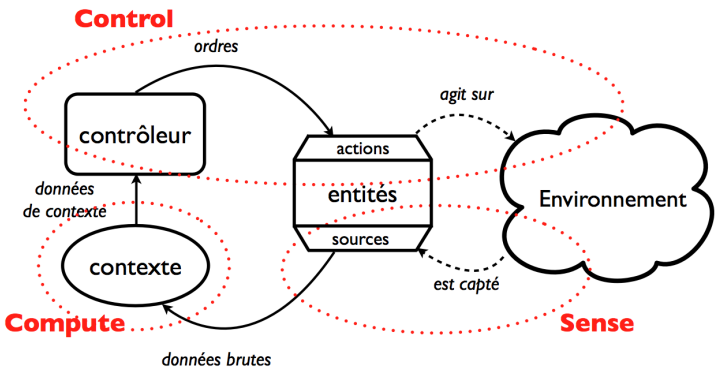

FIGURE 1: Le paradigme SCC

taxonomie et la description de l'**architecture** de l'application.

La taxonomie est composée d'**entités** et l'application se décompose selon des opérateurs de **contexte** et de **contrôle**.

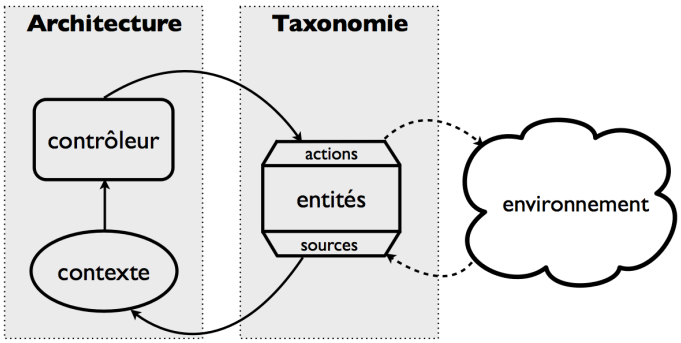

FIGURE 2: Paradigme SCC et Facettes DiaSpec

Ainsi, le langage DiaSpec repose sur les trois types suivants de composants :

- Les **entités** correspondent aux périphériques, qui peuvent être matériels ou logiciels, et interagissent avec un environnement externe au travers de leurs capacités à capter les informations et agir sur l'environnement externe ;
- Les **composants contextes** raffinent, c'est-à-dire filtrent, agrègent et interprètent, les données brutes captées par les entités ;
- Les **composants contrôleurs** utilisent cette information raffinée pour contrôler l'environnement en déclenchant des actions sur les entités.

A partir de la conception architecturale d'une application réalisée avec DiaSpec, la suite outillée DiaSuite fournit du support à chaque phase du cycle de développement logiciel.

3.1.2 *Vue globale DiaSuite*

DiaSuite fournit du support à chaque phase du cycle de développement logiciel, depuis la conception jusqu'au déploiement, comme l'illustre la Figure 3.

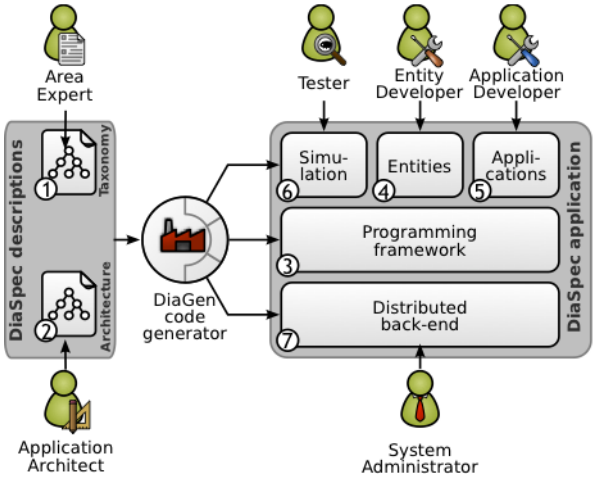

FIGURE 3: DiaSuite : Vue globale

En particulier, la suite d'outils DiaSuite, se base sur une description DiaSpec pour fournir du support adapté à chaque phase : implémentation, test et déploiement. Différents acteurs interviennent le long du cycle de développement, comme l'illustre la Figure 3.

Dans notre thèse, nous avons enrichi la suite d'outils DiaSuite avec du support de QoS temporelle, tout au long du cycle de développement logiciel. Dans ce cadre, nous proposons une intégration de la QoS avec la partie applicative fonctionnelle, de façon unifiée et cohérente, dans le but d'assurer la traçabilité des exigences.

3.1.3 *Exemple considéré*

Dans ce chapitre, afin d'illustrer notre approche, nous considérons un exemple simplifié d'une application avionique de suivi d'un plan de vol.

Une telle application utilise deux capteurs pour le calcul de la trajectoire courante de l'avion : la centrale inertielle (*Inertial Reference Unit*), fournissant la localisation, et l'unité fournissant les données de vol telles que l'angle d'attaque (*Air Data Unit*).

La synchronisation de ces deux sources d'information permet de calculer la trajectoire actuelle de l'avion.

Cette trajectoire est ensuite comparée au plan de vol entré par le pilote et utilisée pour contrôler et corriger les ailerons, si nécessaire.

Les sections suivantes illustrent le développement fonctionnel de cette application au travers du suivi de la méthodologie DiaSuite enrichie avec la dimension de QoS. Parallèlement, elles décrivent comment les exigences de QoS temporelles sont systématiquement traitées tout au long du cycle de développement, pour chaque phase du cycle de développement.

3.2 CAPTURE DES EXIGENCES ET SPÉCIFICATION FONCTIONNELLE

Dans les méthodologies de développement logiciel, la phase de capture des exigences est dédiée à l'identification des besoins utilisateurs. Ensuite, la phase de spécification fonctionnelle identifie les principales fonctionnalités que doit implémenter l'application afin de satisfaire les exigences des utilisateurs. Dans le domaine avionique, chacune de ces fonctionnalités est généralement associée à la notion de *chaîne fonctionnelle*, représentant une chaîne de calcul.

Au regard de la QoS, une contrainte de pire temps d'exécution (*Worst Case Execution Time*, WCET) peut être spécifiée sur une chaîne fonctionnelle, afin de montrer la conformité au regard d'une allocation temporelle sur l'application [38].

L'application de suivi du plan de vol possède ici une seule chaîne fonctionnelle. Dans notre exemple, nous considérons qu'il faut un temps d'exécution, inférieur à 3 secondes. Cette contrainte peut être assimilée à une contrainte WCET.

Par suite, un WCET peut également être raffiné et déposé sur une sous-chaîne fonctionnelle. Dans notre exemple, nous pouvons identifier un segment de chaîne fonctionnelle correspondant au calcul de la trajectoire actuelle. Ce segment de chaîne fonctionnelle peut être réutilisé dans d'autres applications, *e.g.*, l'affichage de la trajectoire courante sur la console de visualisation. Nous considérons qu'un tel segment de chaîne ne doit pas dépasser 2 secondes pour son exécution.

3.3 CONCEPTION ARCHITECTURALE

Lors de la phase de conception architecturale, DiaSuite offre un langage dédié à la description des applications SCC, sur lequel

nous reposons pour intégrer des concepts de QoS au niveau des composants.

3.3.1 *DiaSuite (Dimension fonctionnelle)*

Au niveau de la phase de conception architecturale, le langage DiaSpec permet de concevoir une application en utilisant des déclarations spécifiques au paradigme SCC [27, 28].

3.3.1.1 *Concepts*

Tout d'abord, l'expert du domaine applicatif décrit la **taxonomie**. La taxonomie représente en effet une restriction à un domaine applicatif (par exemple, le domaine avionique). Elle peut ainsi être partagée entre plusieurs applications de même domaine. Dans le langage DiaSpec, elle se présente sous la forme de classes d'entités, représentant les capteurs ou actionneurs de l'environnement.

Le concepteur décrit ensuite le **flot de données** de l'application. Il décompose une application selon une architecture faisant intervenir les composants fonctionnels (composants contexte et contrôleur) ainsi que les échanges de données entre les composants.

Le concepteur spécifie ensuite le **flot de contrôle** de l'application, au travers des interactions que les composants peuvent effectuer entre eux [28]. Ces interactions sont décrites sous la forme de contrats représentant :

- Les conditions d'activation des composants
  - Le composant est activé au moyen d'une requête de communication *pull* provenant d'un autre composant (mode d'interaction synchrone "un-à-un" avec une valeur de retour) ;
  - Le composant est activé à la réception d'une donnée provenant d'un autre composant, transmise au moyen d'une communication *push* (mode d'interaction *publish/subscribe*) ;
- Les données nécessaires
  - Récupération de données d'autres composants : entités ou contextes
- La caractérisation des émissions en sortie
  - "pas d'émission de données"
  - "parfois, des émissions de données peuvent avoir lieu"
  - "des émissions de données ont toujours lieu"

3.3.1.2 *Application*

En suivant la méthodologie de développement DiaSuite, la première étape consiste en l'identification des entités mises en jeu dans l'application de suivi du plan de vol. Dans cet exemple, nous avons identifié quatre entités :

- **InertialReferenceUnit** : centrale inertielle, fournissant des informations telles que la localisation courante de l'avion ;
- **AirDataUnit** : unité fournissant les données de vol telles que l'angle d'attaque ;
- **FlightPlanDatabase** : base contenant le plan de vol entré initialement par le plan ;
- **AutomaticPilot** : interface permettant le contrôle des ailerons.

Un extrait de la taxonomie de notre application est présenté dans le listing 1. Les entités sont déclarées à l'aide du mot-clef `device`. Ces entités peuvent être des sources d'information, comme le suggère le mot-clef `source` suivi du nom de l'information et du mot-clef `as` suivi du type de l'information.

```
1  device InertialReferenceUnit {
2     source localization as Coordinates;
3  }
4
5  device AirDataUnit {
6     source airData as Float[];
7  }
8
9  device FlightPlanDataBase {
10    source plannedTrajectory as WaypointSet;
11 }
12
13 action ControlAilerons {
14    control(command as Float);
15 }
16
17 device AutomaticPilot {
18    action ControlAilerons;
19 }
```

Listing 1: Extrait de la taxonomie des entités

Dans l'exemple présenté en Listing 1, l'entité `InertialReferenceUnit` fournit ainsi une information `localization` de type `Coordinates`.

Une entité représente l'interface d'une application avec l'environnement externe. Dans le langage DiaSpec, toute méthode permettant d'agir sur l'environnement est déclarée au moyen du mot-clef `action`. Une `action` possède un nom, et représente une méthode avec des paramètres. Par exemple, la taxonomie décrite en Listing 1 présente la déclaration d'une

action ControlAilerons correspondant à une méthode control comportant un paramètre command de type Float. Cette action est supportée par l'entité AutomaticPilot.

L'interface générique ControlAilerons offre une abstraction haut-niveau de la réalisation concrète (mécanique ou électronique) des commandes à appliquer sur les ailerons. Une telle abstraction permet de faciliter l'intégration de composants sur l'étagère (*Commercial Off-The-Shelf*, COTS). En effet, toute implémentation compatible de la déclaration de l'entité peut ensuite être utilisée par l'application.

La seconde étape de la méthodologie consiste en la conception architecturale au moyen du style architectural SCC de DiaSpec. La description du système est illustrée en Figure 4, faisant apparaître les quatre couches de composants de DiaSpec.

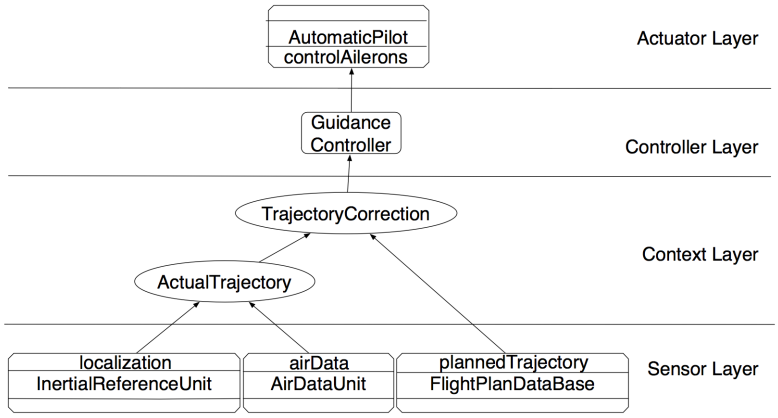

FIGURE 4: Vue en flot de données de l'application de suivi du vol

Dans l'exemple présenté en Figure 4, les composants entités InertialReferenceUnit et AirDataUnit fournissent respectivement la localisation courante de l'avion et les données de navigation. Ces données brutes sont envoyées au composant contexte ActualTrajectory qui calcule la trajectoire courante de l'avion. Cette information est ensuite envoyée au composant contexte TrajectoryCorrection. Lorsqu'il reçoit une nouvelle trajectoire, le composant TrajectoryCorrection récupère la trajectoire planifiée à partir du composant FlightPlanDatabase. En comparant ces sources d'information, le composant TrajectoryCorrection calcule les corrections de trajectoire à appliquer, qu'il envoie ensuite au composant GuidanceController, en charge du contrôle des ailerons au travers du composant AutomaticPilot.

Les composants contextes ActualTrajectory et TrajectoryCorrection ainsi que le composant contrôleur GuidanceController sont décrits avec des contrats d'interaction qui explicitent leur mode de communication, comme le présentent les Listings 2 et 3.

```
context ActualTrajectory as Float {
   interaction {
      when provided (localization from InertialReferenceUnit,
                              airData from AirDataUnit);
     always publish;
   }
}

context TrajectoryCorrection as Float {
   interaction {
      when provided ActualTrajectory;
     always publish;
}
```

Listing 2: Déclaration des composants contexte ActualTrajectory et TrajectoryCorrection

Dans le listing 2, le composant contexte ActualTrajectory est déclaré au moyen du mot-clef context et renvoie des valeurs de type Float. Il traite deux sources d'information : localization et airData, comme l'illustre le contrat d'interaction.

Le contrat d'interaction est déclaré au moyen du mot-clef interaction. Comme nous l'avons décrit en introduction de ce chapitre, les contrats d'interaction ont trois composantes :

- Les conditions d'activation des composants : elles sont décrites au moyen des mots-clefs *when provided*. Dans l'exemple présenté, le composant contexte ActualTrajectory attend la réception de deux événements : localization et airData, respectivement publiés par les entités InertialReferenceUnit et AirDataUnit ;
- Les données nécessaires. Dans l'exemple présenté, lors de la réception d'un événement par le composant contexte ActualTrajectory, le composant contexte TrajectoryCorrection récupère l'information plannedTrajectory du composant entité FlightPlanDataBase, comme le présente la déclaration à l'aide du mot-clef get ;
- La caractérisation des émissions en sortie. Ici, les mots-clefs *always publish* signifient que lors de la réception des deux événements, le composant contexte ActualTrajectory publie systématiquement une donnée, de type Float comme

indiqué en entête de la déclaration du contexte.

```
1
2 controller GuidanceController {
3    interaction {
4       when provided TrajectoryCorrection;
5       always publish;
6    }
7 }
```

Listing 3: Déclaration du composant contrôleur GuidanceController

Dans la spécification DiaSpec présentée en Listing 3, le composant contrôleur GuidanceController est déclaré avec le mot-clef controller.

3.3.2 *Support de QoS*

Comme nous l'avons présenté, lors de la phase de conception architecturale, les chaînes fonctionnelles sont décomposées en des composants connectés. L'architecte peut alors raffiner le WCET global sur la chaîne fonctionnelle en des contraintes de temps sur les composants, selon leur type de communication.

3.3.2.1 *Concepts*

Les interactions de type *pull* sont typiquement adressées par une exigence de temps de réponse, comme cela est fait dans le domaine des services Web [14].

Les interactions de type *push* font apparaître plusieurs besoins. Dans le cas spécifique où une valeur est émise de façon périodique, notre approche offre la capacité de préciser la fréquence.
Le besoin de synchroniser deux ou plusieurs événements en entrée d'un composant apparaît également. Nous adressons ce besoin en introduisant une exigence de fraîcheur sur les valeurs d'événement en entrée. Cette exigence est dans l'esprit de l'intervalle de synchronisation (*synchronization skew*) présent dans le domaine multimédia [64]. En complément de cette contrainte de fraîcheur, nous définissons une contrainte de temps de synchronisation borné, qui autorise une désynchronisation durant un intervalle de temps fini.

Dans notre approche, nous proposons un outil d'*early-validation* permettant de vérifier la cohérence des contrats de QoS dès la

spécification. Ce support permet également d'évaluer l'impact de technologies de déploiement sur la performance globale de l'application, comme nous le détaillons dans la section 3.6. Dans les phases suivantes du développement, nous offrons également du support de QoS. A l'implémentation, du support de surveillance des exigences de QoS à l'exécution est généré, avec des stratégies de remontée d'exceptions. En phase de test, le support d'injections d'erreurs de QoS permet de valider le comportement système spécifié.

Nous présentons maintenant l'application de nos concepts de QoS sur l'exemple considéré.

3.3.2.2 *Application*

La Figure 5 illustre les contrats de QoS associés sur les composants mis en œuvre dans notre application.

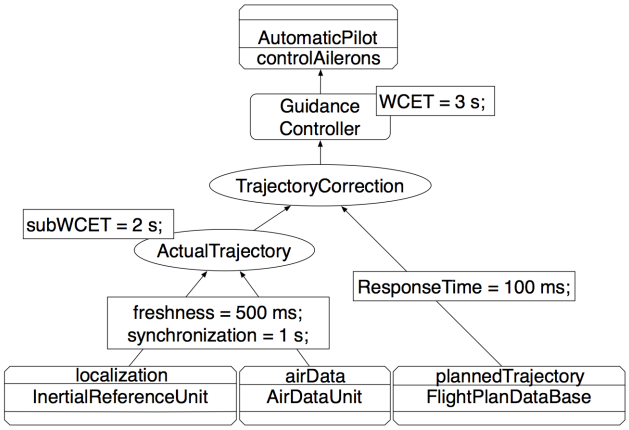

FIGURE 5: Architecture de l'exemple, enrichie de contrats de QoS

Sur la Figure 5, on peut noter que le WCET déposé sur la chaîne fonctionnelle est associé au composant contrôleur `GuidanceController`. En effet, dans le paradigme de conception DiaSpec, le contrôleur est responsable d'une chaîne fonctionnelle.

Par suite des exigences non-fonctionnelles adressées durant durant la phase de Capture des exigences (section 3.2), le WCET de la sous-chaîne fonctionnelle de calcul de la trajectoire est déposé sur le composant contexte `ActualTrajectory`. Il correspond au temps mis par le composant pour rendre l'information de trajectoire, à partir de la réception des informations requises depuis les capteurs.

Le WCET de la chaîne fonctionnelle de suivi d'un plan de vol est raffiné en :

1. Une contrainte de fraîcheur de 500 ms entre localization et airData, ainsi qu'un temps de synchronisation borné de 1 s, autorisant une désynchronisation temporaire entre l'arrivée de ces deux événements, restant compatible du WCET spécifié ;
2. Un temps de réponse de 100 ms pour la récupération de la donnée plannedTrajectory de FlightPlanDataBase.

Comme nous l'avons présenté, les contrats de QoS sont introduits comme une extension de DiaSpec. Le code extrait de la spécification DiaSpec des contextes ActualTrajectory et TrajectoryCorrection enrichie de contrats de QoS est présenté en Listing 4.

```
context ActualTrajectory as Float {
   interaction {
      when provided (localization from InertialReferenceUnit,
                                 airData from AirDataUnit);
      with (freshness = 500ms, synchronization = 1s);
      always publish;
   }
}

context TrajectoryCorrection as Float {
   interaction {
      when provided ActualTrajectory;
      get plannedTrajectory from FlightPlanDataBase in 100ms;
      always publish;
}
```

Listing 4: Intégration de QoS dans la déclaration des composants contexte ActualTrajectory et TrajectoryCorrection

Comme le montre le Listing 4, le contrat de QoS est intégré de façon unifiée à la spécification fonctionnelle du composant contexte. Il est en effet déposé sur les contrats d'interaction, afin d'être conforme au mode de communication entre les composants. Dans l'exemple présenté, la communication est de type *push* entre les entités émettrices de données, InertialReferenceUnit et AirDataUnit, et le composant contexte récepteur de données ActualTrajectory. utilisé entre les deux composants. Le mot-clef *with* permet ainsi d'intégrer une contrainte de QoS sur le lien de communication.

L'exigence haut-niveau de WCET est prise en charge par le composant contrôleur GuidanceController. De ce fait, elle se retrouve dans sa déclaration DiaSpec, comme l'illustre le Listing 5.

Le WCET de 3 s déposé sur la chaîne fonctionnelle dont est responsable ce composant, est spécifié à l'aide du mot-clef *wcet*.

```
controller GuidanceController {
   interaction {
      when provided TrajectoryCorrection;
      always publish;
   }
   wcet = 3s;
}
```

Listing 5: Intégration de QoS dans la déclaration du composant contrôleur GuidanceController

On peut noter que cette contrainte de WCET se situe hors du contrat d'interaction du composant, car elle s'applique non pas à une communication entre deux niveaux de composants, mais sur toute la chaîne fonctionnelle, contrairement aux autres contraintes de QoS qui sont exprimées sur les relations de communication entre composants.

Dans la phase de conception architecturale, nous montrons comment la génération de contraintes numériques à partir de ces exigences est utilisée afin de montrer qu'aucune incohérence n'a été introduite, participant à la démonstration de la traçabilité, requise dans les domaines critiques.

3.3.2.3 *Traçabilité*

La génération de contraintes numériques à partir de la conception architecturale de l'application permet d'assurer que le raffinement du WCET en les contraintes sur les composants ne vient pas invalider les exigences de la précédente phase. Cette génération est similaire au travail de Defour *et al.* [31] et repose sur le patron architectural DiaSuite. Il consiste en la traduction automatique des contrats de QoS en des équations numériques spécifiant les relations temporelles entre composants.

Par exemple, détaillons cette génération de contraintes sur notre application simplifiée de suivi d'un plan de vol.

WCET sur la chaîne fonctionnelle. Le WCET sur la chaîne fonctionnelle de notre application a été spécifié précédemment à 1 s, ce qui conduit à la contrainte numérique suivante.

$$\begin{aligned} &T\_wcet\_GuidanceController \leqslant 3; \\ &T\_wcet\_GuidanceController = \\ &\quad T\_provide\_GuidanceController+ \end{aligned}$$

$$T\_com(GuidanceController, AutomaticPilot);$$

La première équation représente le WCET attaché au composant GuidanceController. La seconde équation raffine la chaîne fonctionnelle en une séquence de deux fonctions :

- Calcul des ordres (segment de chaîne GuidanceController) ;
- Communication des ordres au composant AutomaticPilot.

Ainsi, le temps global correspond à la somme du temps T_provide_GuidanceController et du temps de communication entre GuidanceController et AutomaticPilot (dénoté par la fonction T_com). Le temps T_provide_GuidanceController correspond au segment de chaîne entre le moment où les entités InertialReferenceUnit ou AirDataUnit envoient une valeur et le moment où GuidanceController émet des ordres à destination de AutomaticPilot.

De façon similaire, le temps T_provide_GuidanceController peut également être raffiné en le temps mis par GuidanceController afin de calculer les ordres à émettre, le temps de communication entre TrajectoryCorrection et GuidanceController, ainsi que le temps global associé au segment de chaîne de TrajectoryCorrection.

$$\begin{aligned} &T\_provide\_GuidanceController = \\ &\quad T\_provide\_TrajectoryCorrection+ \\ &\quad T\_com(TrajectoryCorrection, \\ &\qquad GuidanceController)+ \\ &\quad T\_compute\_GuidanceController; \end{aligned}$$

Temps de réponse. Pour les contraintes de temps de réponse, l'accès à une donnée au moyen d'une communication *pull* de façon synchrone entraîne un temps de récupération de la donnée qui peut être décomposé en le temps de calcul de la donnée par le composant et l'aller-retour de communication entre les deux composants, en supposant que la taille de la requête et de la réponse soient compatibles d'une unité maximum de transmission (*Maximum Transmission Unit*, MTU). Le temps précédemment défini et associé à TrajectoryCorrection peut ainsi être raffiné au regard de sa relation avec ActualTrajectory et FlightPlanDataBase.

$$\begin{aligned} &T\_provide\_TrajectoryCorrection = \\ &\quad T\_provide\_ActualTrajectory+ \\ &\quad T\_com(ActualTrajectory, TrajectoryCorrection)+ \end{aligned}$$

$$2 * \mathrm{T\_com}(\mathrm{FlightPlanDataBase}, \mathrm{TrajectoryCorrection}) +$$

$$\mathrm{T\_provide\_FlightPlanDataBase} +$$
$$\mathrm{T\_compute\_TrajectoryCorrection};$$

Temps de fraîcheur et temps de synchronisation borné. Le contrat de QoS associé au composant contexte ActualTrajectory spécifie des contraintes de fraîcheur et de temps de synchronisation borné entre InertialReferenceUnit et AirDataUnit. Dans le pire cas, la synchronisation prend au maximum de temps la somme du temps de synchronisation borné et du temps maximum pour recevoir un événement depuis InertialReferenceUnit ou AirDataUnit. En outre, le sous-WCET spécifié, correspondant au WCET de la sous-chaîne fonctionnelle de calcul de la trajectoire, correspond au temps requis pour que le composant ActualTrajectory fournisse son information, soit 2 s.

Les contraintes suivantes sont ainsi générées.

$$\mathrm{T\_provide\_ActualTrajectory} \leqslant 2;$$
$$\mathrm{T\_synchronization} \leqslant 1;$$
$$\mathrm{T\_provide\_ActualTrajectory} =$$
$$\max($$
$$\mathrm{T\_provide\_airData} +$$
$$\mathrm{T\_com}(\mathrm{AirDataUnit}, \mathrm{ActualTrajectory}),$$
$$\mathrm{T\_provide\_localization} +$$
$$\mathrm{T\_com}(\mathrm{InertialReferenceUnit}, \mathrm{ActualTrajectory})) +$$
$$\mathrm{T\_synchronization} +$$
$$\mathrm{T\_compute\_ActualTrajectory};$$

Chaque contrainte numérique générée à partir de la spécification directement, est définie à l'aide de Prolog IV [29], un langage de programmation logique par contraintes, avec un solveur arithmétique d'intervalles réels afin de vérifier la cohérence des exigences non-fonctionnelles à chaque étape de raffinement.

Egalement, ces contraintes permettent de tester différentes technologies de déploiement, comme le présente la section 3.6.

3.4 IMPLÉMENTATION

Lors de la phase d'implémentation, un *framework* de programmation est généré à l'aide du compilateur DiaSuite. Nous enrichissons ce dernier afin de générer un mécanisme de surveillance des exigences de QoS à l'exécution.

3.4.1 *DiaSuite*

La phase de conception guide le développeur tout au long du cycle de développement et contraint le processus d'**implémentation**.

En effet, à partir d'une description DiaSpec, le compilateur génère un *framework* de programmation en Java dédié à l'application. Ce framework guide et supporte le programmeur durant la phase d'implémentation en fournissant des opérations haut-niveau pour la découverte d'entités et l'interaction entre composants. De plus, l'utilisation du système de type Java permet de contraindre le développement selon la spécification des contrats d'interaction. Le principe d'inversion de contrôle est appliqué uniformément au *framework* de programmation généré. Ainsi, il est garanti que seules les communications spécifiées entre composants existeront au niveau de l'implémentation, empêchant la création de liens supplémentaires (non-accès aux méthodes Java requises pour ce faire) qui pourraient engendrer des malversations sur l'application.

Un exemple de code généré pour le contexte `ActualTrajectory` est présenté en Listing 6.

```
1  public abstract class AbstractActualTrajectory {
2
3      [...]
4
5      public abstract ActualTrajectoryValuePublishable
6          onLocalizationFromInertialReferenceUnitAnd
7              AirDataFromAirDataUnit(
8                  LocalizationFromInertialReferenceUnit
                       localization,
9                  AirDataFromAirDataUnit airData
10         );
11
12 }
```

Listing 6: Génération du framework DiaSuite en Java pour le composant contexte `ActualTrajectory`

Comme l'illustre le Listing 6, dans le cas d'attente de valeurs synchronisées, une seule méthode est générée, donnant les deux informations attendues en paramètres. Le code correspondant à la synchronisation, que l'on génère, est en effet caché dans le *framework*.

L'utilisateur peut ensuite réaliser l'implémentation de la logique applicative en étendant la classe abstraite générée `AbstractActualTrajectory`, comme présenté en Listing 7. Un environnement de développement intégré (*Integrated Development*

Environment, IDE) comme Eclipse, permet, lors de cette étape, de fournir directement un squelette de code à compléter par le développeur.

```
public class ActualTrajectory extends AbstractActualTrajectory {

    [...]

    @Override
    public abstract ActualTrajectoryValuePublishable
         onLocalizationFromInertialReferenceUnitAnd
             AirDataFromAirDataUnit(
                   LocalizationFromInertialReferenceUnit
                       localization,
                   AirDataFromAirDataUnit airData
              )
         {
           // A remplir par le developpeur
         }

}
```

Listing 7: Implémentation de la classe Java `ActualTrajectory`

En outre, le support généré pour les aspects fonctionnels et non-fonctionnels est distinct, offrant ainsi une séparation claire des préoccupations.

3.4.2 *Support de QoS*

En phase d'implémentation, notre approche enrichit le compilateur de DiaSuite en générant du **support de surveillance à l'exécution**, à partir des déclarations de QoS.

3.4.2.1 *Extension du compilateur*

Le compilateur DiaSpec a été étendu afin de générer des composants dédiés à la surveillance de QoS à l'exécution, directement à partir des extensions du langage DiaSpec. Ce support est complètement transparent pour le développeur.

Afin d'illustrer la génération du support de surveillance à l'exécution, considérons le composant contexte `TrajectoryCorrection`. La déclaration de la contrainte de temps de réponse spécifiée est réalisée de façon interne au container qui intercepte les appels au composant, et n'est pas accessible par le composant. Cette méthode qui positionne le *timeout* à 100 ms fait appel à la méthode `setTimeout()`, interne au cœur DiaSuite

qui offre une abstraction du support de communication entre les composants.

Le *framework* de programmation généré, c'est-à-dire la classe abstraite AbstractTrajectoryCorrection, ainsi que le code à implémenter par le développeur lorsqu'il étend cette classe, ne fait pas apparaître cette surveillance d'exigence de temps de réponse.

```
1  public class TrajectoryCorrection extends
       AbstractTrajectoryCorrection {
2
3      [...]
4
5      @Override
6      public TrajectoryCorrectionValuePublishable
7           onActualTrajectory(
8              FlightPlanDataBase flightPlanDataBase)
9           {
10           WaypointSet plannedTrajectory =
                flightPlanDataBase.getPlannedTrajectory();
11
12           [...]
13
14          }
15
16 }
```

Listing 8: Implémentation de la classe Java TrajectoryCorrection

Le code implémenté par le développeur d'applications au sein de la classe TrajectoryCorrection est agnostique de la surveillance des exigences de QoS spécifiées durant la phase de conception architecturale. En effet, l'appel à la méthode getPlannedTrajectory() pour récupérer la trajectoire planifiée entrée par le pilote, ne fait pas apparaître l'exigence de 100 ms.

3.4.2.2 *Surveillance des contraintes de QoS*

La surveillance est une capacité critique dans le domaine avionique, traitant de la *safety* d'un système. Elle est en effet définie comme "fonctionnalité dans un système conçue pour détecter le comportement anormal de ce système" dans les standards de certification [38].

En particulier, la surveillance d'exigences de QoS temporelle permet de détecter tout défaut de respect des exigences de QoS au sein de l'application et d'enregistrer cette information. La surveillance permet, lors de la phase de maintenance, une analyse plus fine des piles d'appels ayant causé la violation du

contrat de QoS. Egalement, la surveillance permet de détecter les cas probables de panne de capteurs (enchaînement de violations de contrat de QoS) et d'en informer le pilote afin qu'il puisse adopter la réaction appropriée.

Au niveau de l'implémentation, les exigences de QoS déposées sur les composants deviennent des vérifications à l'exécution qui reposent sur les méthodes de communication du *framework* de programmation généré. Ces mécanismes de surveillance sont encapsulés dans des containers de composants qui assurent que tout défaut de respect des exigences spécifiées de temps de réponse et de fraîcheur des données sera détecté et signalé. Cette approche permet la séparation des préoccupations entre les exigences fonctionnelles et non-fonctionnelles car un container est uniquement en charge de l'interception des appels pour la surveillance des exigences, et de la transmission des appels au composant fonctionnel.

Si un contrat de QoS est violé, le container lève des exceptions spécifiques : `TimeoutException`, `SynchronizationException`.

Le traitement de telles exceptions est laissé au développeur. Dans l'étude de cas que nous avons effectuée et que nous détaillons davantage dans le Chapitre 5 [43], ce traitement se traduit sous la forme d'enregistrement des violations de QoS qui se sont produites et de reconfiguration applicative. Cette dernière est rendue possible en utilisant le support déclaratif fourni au niveau architectural pour le traitement exceptionnel [80], qui permet de séparer le fonctionnement `nominal` de l'application du fonctionnement en cas d'`erreur`, comme le présente le Chapitre 4.

La génération de ces mécanismes de surveillance est complètement transparente pour le développeur. Ces mécanismes sont en effet embarqués dans le *framework* de programmation afin de s'insérer entre les composants communicants pour contrôler le temps qui s'écoule lors de la transmission des données. Ainsi, le développeur ne peut pas introduire d'erreur dans le code de surveillance ni désactiver, par erreur, un container correspondant à un contrat de QoS temporelle. Cette approche offre donc la garantie que tout contrat de QoS spécifié sera vérifié à l'exécution.

Notre approche enrichit le compilateur de DiaSuite en générant des composants dédiés à la surveillance à l'exécution des contrats de QoS. Le support de surveillance de QoS étant généré directement à partir des déclarations de QoS, la traçabilité des exigences de QoS est automatiquement assurée entre les

phases de conception architecturale et d'implémentation, tout comme la traçabilité des exigences fonctionnelles, du fait du mécanisme de génération automatique de code [28].

L'algorithme interne du container s'appuie, pour le temps de réponse, sur le calcul du temps qui s'écoule entre la requête et la réponse, et l'évaluation de sa conformité à la spécification.

La partie la plus élaborée concerne les exigences de fraîcheur et de synchronisation déposées sur le contexte `ActualTrajectory`. Ces exigences sont décrites au moyen de l'automate temporisé présenté en Figure 6.

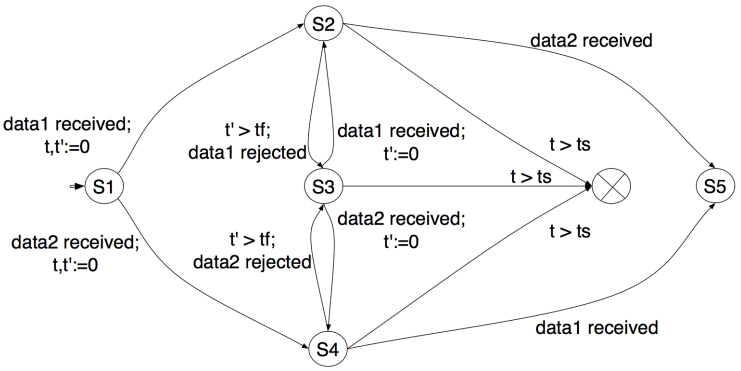

FIGURE 6: Automate de temporisation

Afin de détailler l'automate de fraîcheur et synchronisation illustré sur la Figure 6, supposons que nous voulions synchroniser les valeurs `data1` et `data2`, où `data1` représente la donnée `localization` et `data2` représente la donnée `airData` .

L'état initial est S1.

Lorsque la première donnée est reçue (états S2 et S4), le container de surveillance active les timers t et t' afin de mesurer respectivement le temps de synchronisation borné et le temps de fraîcheur ($t, t' := 0$).

Tant que le temps de synchronisation borné n'est pas atteint ($t \leqslant ts$, *i.e.*, $t \leqslant 1$), le container attend la réception de données fraîches.

Si l'autre donnée est reçue avant que le temps de fraîcheur ne se soit écoulé, le container de surveillance transmet ces données synchronisées au composant fonctionnel contenu (état final S5).

En revanche, si le temps de fraîcheur est dépassé ($t' > tf$, *i.e.*, $t' > 0.5$), la donnée est rejetée. Cet état n'est pas considéré comme un état d'erreur car le temps de synchronisation borné autorise justement la désynchronisation des données pendant

une période de temps finie.

Si le temps de synchronisation borné s'est écoulé ($t > ts$), la synchronisation est abandonnée et une erreur de type `SynchronizationException` est levée (état d'erreur).

L'implémentation de l'automate de temporisation présenté en Figure 6 est cachée dans le *framework*.

3.5 TEST

3.5.1 *DiaSuite*

La phase de test repose également sur le paradigme SCC. Le compilateur DiaSpec génère, outre un *framework* de programmation, du support pour le test et la simulation du système. Le simulateur paramétré DiaSim offre en effet un moyen de tester les classes d'entité et le comportement applicatif, au travers de stimuli qui sont envoyés [20, 21].

DiaSuite offre la capacité d'exécuter le même code applicatif, aussi bien dans l'environnement concret que dans l'environnement simulé grâce à DiaSim. DiaSim permet en effet d'émuler l'exécution d'une application sans requérir le moindre changement dans le code applicatif.

Cette capacité est supportée du fait que la taxonomie fait abstraction de la réalisation des entités, et de l'hétérogénéité entre entités matérielles ou logicielles. Les entités peuvent ainsi être simulées afin de valider le comportement de l'application.

En outre, DiaSim offre un environnement de simulation hybride, c'est-à-dire qu'il permet de tester l'application en la déployant avec des entités physiques rélles et des entités simulées. Il permet ainsi de réaliser des tests unitaires d'entités, selon la disponibilité du matériel.

3.5.2 *Support de QoS*

Durant la phase de test, le support de surveillance généré à partir d'une spécification de QoS est utilisé afin de tester le traitement exceptionnel spécifié suite aux défaillances de QoS. La traçabilité des exigences de QoS jusqu'à la phase de test est donc supportée.

Dans notre exemple, la violation de contrats de QoS conduit à l'enregistrement dans un fichier dédié, des erreurs qui ont été

levées. Ainsi, l'étude de ce fichier de log permet d'analyser la détection des violations de contraintes temporelles.

En phase de test, le support de surveillance des contraintes de QoS qui est généré sert également à la vérification de traitements de violations de QoS. En particulier, l'implémentation que nous avons réalisée dans le cadre de l'étude de cas qui sera présentée Chapitre 5, repose sur une injection d'erreurs de QoS afin de contrôler l'adéquation du comportement de gestion de violation de QoS avec la spécification de ce traitement.

3.6 DÉPLOIEMENT

3.6.1 *DiaSuite*

Le déploiement d'une application au sein de DiaSuite s'effectue avec le choix de la cible de communication, parmi celles supportées : Web Services, RMI, CORBA, SIP. Le code applicatif est complètement agnostique de cette cible de communication.

Le déploiement d'une application DiaSuite repose sur le déploiement du serveur DiaSuite, qui offre aux composants la capacité de s'enregistrer (*i.e.,* être instancié) et de découvrir les services offerts par les composants distribués qui les entourent. Le serveur DiaSuite est également en charge de la gestion des événements, c'est-à-dire de la transmission des événements entre composants abonnés, à partir de la spécification de l'application. Concrètement, le serveur DiaSuite est généré dans le *framework* et suit la même procédure de déploiement que tout autre composant.

Ainsi, lors du déploiement de l'application, chaque composant DiaSpec (entité, contexte ou contrôleur) est instancié et initialisé. La phase d'instanciation permet de préparer la communication entre les composants grâce à la définition d'un *ServiceConfiguration* indiquant le choix de la technologie de communication retenu (*e.g.,* RMI). La phase d'initialisation permet d'autoriser le composant à la souscription à des événements envoyés par un composant, en cohérence avec la spécification, ou encore à la publication d'événements.

3.6.2 *QoS*

Grâce à la génération de contraintes numériques à partir des exigences de QoS spécifiées (section 3.3.2.3), notre approche offre la possibilité de tester différentes technologies de déploiement, dans l'esprit de l'approche de Becker *et al.* [12], et d'analyser le rendu en termes de performance sur l'application, au travers de la tenue ou non des objectifs de QoS. Cet outil se base d'une part, sur les contraintes numériques de QoS, et d'autre part, sur les performances de technologies de déploiement afin de vérifier la cohérence de ces contraintes grâce à l'utilisation d'un solveur de contraintes externe. L'outil détermine si la configuration de déploiement testée est compatible avec le WCET spécifié.

Parmi les contraintes générées, plusieurs variables numériques dépendent du déploiement.

Les variables de type T_com_ < componentName > dépendent du média de communication utilisé. En effet, si l'application est déployée sur une plateforme qui n'est pas distribuée, le temps de communication entre composants peut être considéré comme négligeable au regard des temps de calcul des composants applicatifs. En revanche, si l'application s'exécute sur une plateforme distribuée, le temps de communication entre composants applicatifs dépend des technologies utilisées. Ainsi, un réseau donné influera sur les contraintes de communication détourées dans notre outil, selon sa bande-passante par exemple. En avionique, un réseau couramment utilisé est l'AFDX. Dans notre application, les contraintes de communication entre composants applicatifs peuvent ainsi être raffinées selon la bande passante AFDX et le BAG (*Bandwidth Allocation Gap*) associé. D'autres médias de communication peuvent être utilisés, par exemple pour la communication entre les composants applicatifs et les capteurs réels (*e.g.,* ARINC429, RS422), conduisant à des temps de communication différents.

Les variables de type T_compute_ < componentName > dépendent de la complexité des algorithmes de calcul au sein de chaque composant ainsi que de la puissance offerte par la plateforme d'exécution, au travers de la fréquence CPU et du temps d'accès mémoire par exemple.

Enfin, les variables de type T_provide_ < dataSensed > dépendent des technologies utilisées pour les composants entités (*e.g.,* probes mécaniques ou LASER pour l'unité AirData).

Par exemple, supposons que nous souhaitons enrichir le système avec une nouvelle fonctionnalité d'affichage de la trajectoire

sur la console de visualisation. Nous voulons ainsi réutiliser le segment de chaîne fonctionnelle calculant la trajectoire courante, mais avec le respect d'exigences de QoS plus contraignantes, conduisant à la contrainte $T\_provide\_ActualTrajectory \leqslant 0.8$. A partir de ces exigences, notre outil d'aide au choix d'une architecture de déploiement infère une intervalle de valeur acceptable pour $AirDataUnit : 0 \leqslant T\_provide\_airData \leqslant 0.8$. Cette contrainte peut être transmise aux systémiers en charge de la sélection des technologies utilisées pour la plateforme d'exécution. Dans cette situation, le choix s'orientera vers les probes LASER dont la performance est compatible avec cet intervalle d'exigences.

3.7 CONCLUSION

Au cours de ce chapitre, nous avons présenté notre approche d'intégration de la QoS tout au long du cycle de développement. En particulier, nous avons présenté la couche de QoS que nous avons réalisée pour enrichir la méthodologie DiaSuite, et le support fourni à chaque étape de développement. Notamment, notre support pour la déclaration d'exigences de QoS est adapté au mode de communication entre composants (*pull* et *push*). Un mécanisme de surveillance des QoS à l'exécution est, par suite, généré automatiquement à partir de la spécification DiaSpec enrichie avec les contrats de QoS.

Il reste maintenant à répondre à la question suivante :

Que faire en cas de violation des exigences de QoS ?

En effet, il faut tenir compte également de l'impact de cette violation au niveau système.

Le prochain chapitre décrit le traitement résultant de cette détection de violation de QoS, afin de remettre l'application dans un état stable, c'est-à-dire, l'intégration unifiée avec le mécanisme de gestion d'erreurs offert par DiaSuite.

4

INTÉGRATION UNIFIÉE DE LA QOS ET LA GESTION D'ERREURS

Afin d'offrir un traitement à la violation d'exigences de QoS détectée lors de l'exécution du système, nous avons intégré nos travaux avec ceux réalisés dans le cadre de la gestion d'erreurs au sein de DiaSuite [80]. L'intérêt offert est que nous reposons sur le même paradigme pour cette intégration, afin d'offrir une couche de **supervision**.

Ce chapitre présente tout d'abord les concepts de cette intégration unifiée des exigences de QoS et de la gestion d'erreurs, avant de s'intéresser à la présentation du mécanisme d'exceptions de DiaSuite, puis l'intégration avec notre approche.

4.1 CONCEPTS D'UNIFICATION DES EXIGENCES

Cette section présente les concepts qui ont trait à l'intégration des aspects fonctionnels et non-fonctionnels de façon unifiée au sein de la méthodologie DiaSuite.

Tout d'abord, les aspects fonctionnels et non-fonctionnels sont vus comme deux couches *Sense / Compute / Control*.

4.1.1 *Couches SCC*

Comme l'illustre la Figure 7, la méthodologie DiaSuite enrichie des aspects fonctionnels et non-fonctionnels consiste en la découpe de la conception de l'application en une couche représentant la **logique applicative** et une couche de supervision des aspects **non-fonctionnels**.

Dans la Figure 7, nous pouvons noter que la méthodologie DiaSuite repose entiérement sur le paradigme de conception SCC, et ce, à la fois pour les aspects fonctionnels et non-fonctionnels de l'application. Lorsqu'une situation de défaillance est détectée au niveau fonctionnel, telle qu'une panne de capteur par exemple, un événement exceptionnel est levé, afin de laisser le contrôle de l'application à la couche de supervision, qui peut effectuer le traitement adéquat. Cette conception en couches permet ainsi de factoriser le traitement du superviseur de l'application. Un autre

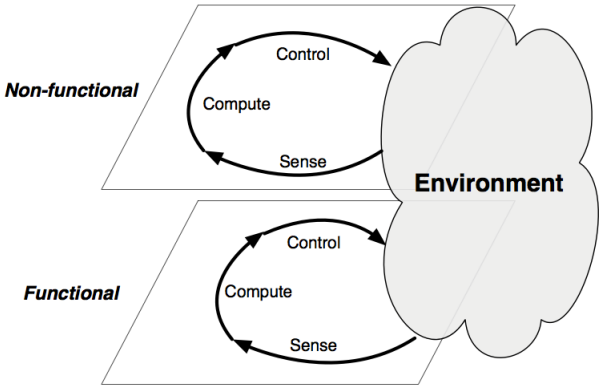

FIGURE 7: Vue en couches du paradigme SCC

avantage offert est la capacité de pouvoir traiter les erreurs au niveau système, sans avoir à introduire de nouveaux concepts dans le langage.

L'unification des aspects fonctionnels et non-fonctionnels au sein de la méthodologie DiaSuite vise à répondre aux besoins de cohérence et de conformité fixés en introduction de ce manuscrit.

4.1.2 *Cohérence et Conformité*

Pour l'intégration des concepts de QoS et de gestion d'erreurs, nous faisons levier sur la méthodologie outillée DiaSuite, qui garantit un support unifié de spécification de ces aspects (**Cohérence**), ainsi qu'une conformité entre chaque étape de développement, du fait du support de géneration (**Conformité**). Ces caractéristiques sont en effet requises dans les domaines critiques tels que l'Avionique ou l'Automobile.

Cohérence.
Les aspects fonctionnels et non-fonctionnels d'une application sont intrinsèquement couplés. De ce fait, il est critique d'assurer leur cohérence afin d'éviter l'apparition de pannes non prévues [76]. Par exemple, l'ajout de mécanismes de gestion d'erreurs dans un système peut dégrader de façon significative la performance de ce système. Généralement, ce genre de problème est détecté de façon tardive dans le cycle de développement logiciel, impactant le coût de re-développement de l'application [5].

Conformité.
Dans le développement d'une application, démontrer la

conformité entre l'application finale, c'est-à-dire le code, et ses exigences haut-niveau, est typiquement réalisé en traçant la propagation de ces exigences au travers des différentes phases du développement. En pratique, ce processus est sujet à erreur car il est souvent réalisé manuellement [75].

Afin de répondre à ces critères, de façon similaire à un paradigme de **programmation**, le paradigme de **conception** DiaSuite fournit des concepts spécifiques au domaine SCC ainsi que des abstractions. Nous proposons donc d'utiliser ce paradigme afin de décrire de façon uniforme les aspects fonctionnels et non-fonctionnels d'une application, permettant ainsi d'assurer la cohérence entre ces différents aspects et leur traçabilité tout au long du cycle de développement.

Les couches unifiées d'intégration des aspects fonctionnels et non-fonctionnels d'une application permettent de superviser l'application.

4.1.3 *Supervision*

L'unification des aspects fonctionnels et non-fonctionnels d'une application au sein de DiaSuite, grâce à l'emploi commun du paradigme SCC pour ces différentes couches, permet de supporter la supervision applicative au niveau système.

De façon conjointe avec la logique applicative, les traitements effectués pour la supervision de l'application sont spécifiés en DiaSpec en utilisant des chaînes fonctionnelles SCC séparées. La conception de toute application peut ainsi être décomposée en plusieurs chaînes fonctionnelles et de supervision : une chaîne fonctionnelle pour chaque mode applicatif, selon l'analyse des besoins opérationnels de l'application, et une chaîne de supervision pour chaque traitement de supervision. Ce traitement peut participer, durant l'exécution, au recouvrement d'erreurs liées aux événements exceptionnels réceptionnés par la couche de supervision, tel que la reconfiguration de l'application. Il peut également participer à l'enregistrement d'erreurs survenant au cours de l'exécution, à des fins d'analyse et de traitements correctifs éventuels lors de la phase de maintenance.

Les contraintes de QoS sont utilisées spécifiquement par la couche de supervision qui contrôle tout événement survenant au sein de la logique applicative. En effet, dans la couche de supervision, un composant est dédié à la récupération de

toute exception provenant d'une violation de contrat de QoS à l'exécution. Ce composant est ensuite chargé de déterminer si cette violation peut être synonyme de défaillance du composant associé.

Maintenant que nous avons présenté les concepts globaux de l'unification des aspects fonctionnels et non-fonctionnels, conduisant à une capacité de supervision des aspects applicatifs au niveau système, la section suivante présente le mécanisme d'exceptions DiaSuite, avec lequel s'interface notre approche pour offrir une capacité de traitement de violation de contrats de QoS.

4.2 MÉCANISME D'EXCEPTIONS DIASUITE

La sûreté de fonctionnement d'un système est sa capacité à éviter les pannes dont la fréquence et la sévérité sortent du cadre acceptable [9]. Ce concept générique inclut des attributs tels que la disponibilité, l'intégrité et la fiabilité. Les systèmes sûrs de fonctionnement requis dans de nombreux domaines, tels que l'avionique, doivent démontrer la conformité du code développé avec la spécification, identifiée sous forme d'exigences haut-niveau.

Les mécanismes de gestion d'erreurs au sein de DiaSuite reposent à l'heure actuelle sur une identification et une caractérisation des fautes possibles pouvant survenir au sein d'applications SCC [42, 80]. En particulier, les défaillances possibles ciblent les *entités*, exposées aux fautes physiques, de par leur interface avec l'environnement.

4.2.1 *Déclaration des exceptions*

La taxonomie des entités DiaSpec a donc été enrichie afin de pouvoir décrire des exceptions qui, lorsqu'elles sont levées, traduisent une faute de l'entité. Cette approche permet aux composants contexte qui communiquent avec ces entités, de pouvoir connaitre leur changement d'état éventuel.
Un exemple de spécifications d'exceptions est donné en Listing 9, à partir de l'application présentée tout au long du Chapitre 3.

Dans cette approche, l'entité `InertialReferenceUnit` peut lever une exception de type `LowAccuracyException` afin de signaler une erreur quant à la précision des données de pression fournies. De façon similaire, la déclaration de l'exception `FailureException` permet à l'entité de signaler sa panne.

```
device InertialReferenceUnit {
   source localization as Coordinates;
   ...
   raises LowAccuracyException;
   raises FailureException;
  }
}
```

Listing 9: Spécification des exceptions

4.2.2 *Traitement des exceptions*

En complément de la déclaration des exceptions effectuée au niveau des entités, des contraintes sur le traitement des exceptions peuvent être ajoutées au contrat d'interaction du composant qui risque de se voir lever une erreur par un composant qu'il a appelé ou bien par un composant auquel il est abonné. Des choix peuvent donc être effectués par le concepteur d'applications afin de décider du traitement à effectuer.

4.2.2.1 *Spécifications*

Reprenons notre exemple en imaginant un composant contexte `RequireLocalization`, qui, lorsqu'il reçoit une information `tick` d'une entité `Timer`, récupère l'information `localization` de l'entité `InertialReferenceUnit`, au moyen d'une communication *pull*.

```
context RequireLocalization as Float {
 interaction {
      when provided tick from Timer;
      get localization from InertialReferenceUnit [mandatory
          catch];
      always publish;
   }
}
```

Listing 10: Spécification du traitement des exceptions

Dans l'exemple présenté en 10, nous avons illustré un exemple de spécification de traitement des erreurs à l'aide des mots-clefs `mandatory catch`.

Quatre sorte de traitements peuvent être spécifiés au niveau des contrats d'interaction :

1. Traitement **obligatoire**, à l'aide des mots-clefs `mandatory catch` : l'erreur doit être compensée systématiquement, permettant ainsi de poursuivre l'exécution de l'application ;

2. Traitement **facultatif**, à l'aide des mots-clefs `optional catch` : le choix de traiter ou non l'erreur est laissé au développeur ;
3. Traitement **évité**, à l'aide des mots-clefs `skipped catch` : l'erreur est systématiquement propagée au composant appelant ;
4. **Absence** de traitement, à l'aide des mots-clés `no catch`.

L'exemple présenté en Figure 10 implique que le composant contexte `RequireLocalization` est responsable de traiter systématiquement toute exception levée par l'entité `InertialReferenceUnit`, qu'il s'agisse d'une exception de type `LowAccuracyException` ou `FailureException`.

4.2.2.2 *Implémentation*

Selon la spécification de traitement, un support dédié est généré pour le développeur d'applications. En particulier, dans le cas de (1) et (2), la méthode permettant d'accéder à l'information `localization` implique de fournir une continuation en paramètre. Dans cette continuation sera spécifié le traitement de l'erreur, *e.g.*, renvoi d'une valeur par défaut. Dans le cas de (3), la méthode est générée sans continuation.

4.3 INTÉGRATION DE LA QOS ET LA GESTION D'ERREURS

Dans la suite logique de l'intégration de concepts de QoS dans DiaSuite telle que nous l'avons présenté dans le Chapitre 3, nous avons couplé la gestion de QoS avec le mécanisme de gestion d'erreurs offert par DiaSuite [80].

4.3.1 *Exceptions de QoS*

Nous avons tout d'abord enrichi les déclarations d'exceptions dites *built-in* au sein du cœur DiaSuite [42] afin de supporter leur propagation dans tout le système, et notamment la possbilité de leur capture par la couche de supervision.

Ainsi, les exigences non respectées de temps de synchronisation borné résultent en la levée d'une exception de type `SynchronizationException`. De même, les exigences non-respectées de temps de réponse entraînent une exception de type `TimeoutException`. Notons qu'il n'est pas nécessaire de lever une exception lors d'un dépassement de temps de fraîcheur, le temps maximal requis étant borné par le temps de

synchronisation borné.

Ainsi, le développeur d'applications n'a pas besoin de déclarer ces types d'exceptions qui sont automatiquement levées par le container de surveillance dédié, généré à partir de la spécification DiaSpec enrichie des contraintes de QoS. Les exceptions de QoS qui sont levées rejoignent donc le flot d'exceptions commun de DiaSuite. Elles peuvent donc être capturées par un composant contexte afin d'effectuer un traitement au niveau **système** pour pallier ou corriger ce problème.

4.3.2 *Supervision de QoS*

La réception de toute erreur de type `TimeoutException` au sein de la couche de supervision signifie la violation d'un contrat de QoS de temps de réponse. Elle s'accompagne d'un ensemble d'informations sur le composant défaillant. Ainsi, des politiques de gestion d'erreurs de QoS peuvent ensuite être définies, comme la reconfiguration applicative, ou encore l'enregistrement des erreurs à des fins de maintenance du système.

Par exemple, la levée consécutive de plusieurs erreurs de QoS impliquant un même composant constitue une suspicion sur le fonctionnement correct du composant impliqué. Aussi, il peut être décidé qu'en cas de trois erreurs consécutives, le composant mis en défaut doit être désactivé. Le traitement sera ainsi réalisé par la couche de supervision.

De même, lorsque deux erreurs consécutives sont relevées, dues à un non-respect de QoS, mais qu'au tour d'horloge suivant, la contrainte de QoS est respectée, on peut admettre que le composant mis en jeu a peut être subi trop de sollicitations, d'où ses difficultés à répondre dans les temps impartis, et le laisser poursuivre son exécution, tout en le surveillant.

Considérons un composant contexte `Supervisor` décidant de surveiller les exceptions levées par le composant contexte `RequireLocalization` spécifié en Listing 10 et par le composant contexte `ActualTrajectory` spécifié en Listing 2.

La déclaration de ce composant est présentée en Listing 11.

Comme nous pouvons le constater, le composant contexte `Supervisor` est abonné aux exceptions qu'il souhaite suivre. Le choix est fait de systématiquement traiter au niveau système, toute erreur résultant d'une panne physique, comme l'indiquent les mots-clefs `always publish`. En revanche, la stratégie adoptée pour un non-respect d'exigences de QoS est d'en recueillir plusieurs avant d'en déduire la panne d'un équipement, aussi,

```
context Supervisor as StructInfo {
  exception LowAccuracyException from InertialReferenceUnit;
  exception FailureException from InertialReferenceUnit;
  exception TimeoutException from InertialReferenceUnit;
  exception SynchronizationException from ActualTrajectory;

 interaction {
      when caught LowAccuracyException from InertialReferenceUnit,
               FailureException from InertialReferenceUnit;
      always publish;
  }

  interaction {
      when caught TimeoutException from InertialReferenceUnit;
      maybe publish;
   }

  interaction {
      when caught SynchronizationException from ActualTrajectory;
      maybe publish;
   }
}
```

Listing 11: Spécification du contexte *Supervisor* dédié à la gestion des exceptions

elles ne sont pas toujours publiées, comme l'indiquent les mots-clefs *maybe publish*.

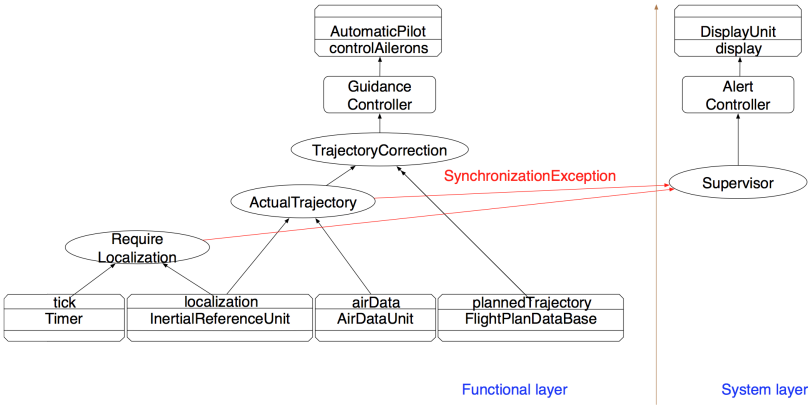

FIGURE 8: Application et gestion d'erreurs

Comme l'illustre la Figure 8, en cas de levée d'erreurs au niveau de la couche applicative fonctionnelle, via le mécanisme des exceptions, ces erreurs sont récupérées au niveau de la couche système, de supervision, par le composant contexte *Supervisor* qui les raffine afin de les envoyer ensuite au composant contrôleur *AlertController* en charge de l'affichage d'alertes sur la console.

4.4 CONCLUSION

Nous avons présenté tout au long de ce chapitre l'intégration des aspects fonctionnels et non-fonctionnels dans la méthodologie DiaSuite. Cette intégration est réalisée de façon unifiée. Notamment, la partie fonctionnelle de l'application est décrite à l'aide du langage DiaSpec. Les aspects non-fonctionnels de QoS sont ensuite intégrés en cohérence avec les contrats d'interaction spécifiés pour chaque composant. De façon similaire, le mécanisme d'exceptions s'intègre dans la spécification d'une application.

Cette unification des aspects permet de faire levier sur le mécanisme de gestion d'erreurs afin de proposer un traitement en cas de violation des contrats de QoS qui ont été spécifiés. En particulier, la couche de supervision de l'application est utilisée afin de surveiller les exceptions de QoS levées par l'application et d'effectuer le traitement approprié. Par exemple, nous avons montré qu'une stratégie pouvait être de considérer la levée consécutive d'erreurs de QoS par un composant comme symptomatique de son dysfonctionnement. Le traitement de ce dysfonctionnement est ensuite réalisé au sein de la couche de supervision au niveau système, en suivant le paradigme SCC. Par exemple, la couche de supervision procède à l'arrêt du composant impliqué et à une reconfiguration applicative.

Nous pouvons maintenant présenter l'étude de cas que nous avons réalisée, afin de valider notre approche, et qui met en œuvre les dimensions fonctionnelles et non-fonctionnelles d'une application, incluant QoS et gestion d'erreurs.

Troisième partie

VALIDATION

5

ETUDE DE CAS

Afin de valider notre approche de développement de logiciels intégrant des concepts de QoS, nous l'avons instanciée sur une étude de cas, au travers d'une application avionique concrète couplée à un simulateur de vol. Ce travail a été présenté à la conférence PECCS (*International Conference on Pervasive and Embedded Computing and Communication Systems*) [43].

Les critères retenus pour l'évaluation de l'etude de cas sont d'une part, la gestion de la cohérence entre les aspects fonctionnels et non-fonctionnels, et, d'autre part, le respect de la conformité des exigences entre chaque étape du processus de développement logiciel.

5.1 APPLICATION ÉTUDIÉE

Afin d'être représentatif des domaines critiques dans lequel notre approche s'applique, nous avons choisi de considérer comme étude de cas le développement d'une application de guidage avionique. Une telle application doit respecter des exigences haut-niveau fortes, à la fois fonctionnelles et non-fonctionnelles (tolérance aux fautes, performance).

Dans le domaine avionique, l'application de guidage est en charge de la navigation de l'avion, sous la supervision du pilote [82]. Le pilote peut définir un plan de vol qui sera automatiquement suivi. Il peut également directement spécifier des paramètres à atteindre durant le vol, tels que l'altitude ou le cap.

Chaque paramètre est géré par un mode de navigation spécifique : le mode de suivi de l'altitude prend en charge le paramètre "altitude", et de même pour les autres paramètres. Nous pouvons ainsi citer le mode de suivi du cap, ou encore le mode de suivi de la vitesse. Lorsque le pilote sélectionne un de ces modes de navigation, l'application de guidage est en charge du calcul des actions à effectuer sur les ailerons et les élévateurs afin d'atteindre la position cible.

Par exemple, si le pilote spécifie un cap à suivre, l'application récupère l'information de cap courant à partir de périphériques tels que la centrale inertielle (*Inertial Reference Unit*, IRU)),

compare cette information au cap courant et agit sur les ailerons en conséquence. Ce mode de suivi du cap, comme les différents modes de navigation supportés par l'application, est généralement associée à une chaîne fonctionnelle, représentant une chaîne de calculs, depuis les capteurs jusqu'aux actionneurs.

Dans le domaine avionique, la première étape de découpe d'une fonction avionique en systèmes consiste en le déroulement d'analyses de *safety* afin d'identifier les situations dangereuses, à partir de conditions de panne [8]. A partir de ces analyses, sont définies des exigences non-fonctionnelles sur l'application. Dans notre application, nous considérons les exemples d'exigences fonctionnelles et non-fonctionnelles suivantes :
Req1. Le temps d'exécution de la chaîne fonctionnelle de suivi du cap ne doit pas excéder 650ms.
Req2. La fraîcheur des données de navigation utilisées par l'application doit être inférieure à 200ms.
Req3. Le dysfonctionnement ou la panne d'un capteur doit être systématiquement signalé au pilote, dans un intervalle de temps de 300ms après la détection du dysfonctionnement ou de la panne.
Req4. Tout mode de navigation doit être désactivé (notion de *fail-safe*) si un des capteurs mis en jeu dans la chaîne de calculs tombe en panne.

Ces exigences sont typiquement détourées dans les phases de *Capture des exigences et de Spécification Fonctionnelle*. Dans la phase suivante, la conception architecturale de l'application doit ensuite être conforme à ces exigences haut-niveau. Egalement, la traçabilité de ces exigences jusqu'au produit final doit être garantie dans le suivi du processus de certification. Ce besoin suggère l'intérêt de suivre une méthodologie de développement dirigée par la conception, comme DiaSuite.

5.2 CONCEPTION

Cette section présente la conception de l'application de guidage, avec la prise en compte des exigences fonctionnelles et non-fonctionnelles, ainsi que le support de validation.

5.2.1 *Approche*

Comme nous l'avons présenté dans le Chapitre 4, la détection d'une situation fonctionnelle provoque l'envoi d'un événement,

afin de permettre à la couche superviseur de contrôler l'application.

Par exemple, l'exigence **Req4** décrit la nécessité de désactiver tout mode de navigation reposant sur un capteur faisant défaut. Dans cette situation, le mécanisme de gestion d'erreurs présent au sein de DiaSuite [80] est utilisé de la façon suivante : si un capteur de navigation tombe en panne, un événement est levé, permettant à la couche superviseur de prendre le contrôle de l'application afin de désactiver les modes de navigation qui dépendent de ce capteur.

La conception de l'application de guidage avionique peut ainsi être décomposée en plusieurs chaînes fonctionnelles et de supervision : une chaîne fonctionnelle pour chaque mode de navigation et une chaîne de supervision pour chaque traitement de supervision, tel que la reconfiguration de l'application, l'enregistrement d'erreurs ou l'affichage d'erreurs sur la console de visualisation à destination du pilote. Dans le reste de cette section, nous nous concentrons sur la chaîne fonctionnelle dédiée au mode de navigation du suivi du cap ainsi que sur la chaîne de supervision dédiée à la désactivation des modes de navigation en cas d'erreur de capteurs mis en jeu.

5.2.2 *Couche fonctionnelle*

Comme nous l'avons décrit en Section 3.3.1, le langage de conception DiaSpec fournit des constructeurs pour la déclaration spécifique des composants entités, contextes et contrôleurs [28].

La taxonomie des entités utilisées par le mode de suivi du cap de notre application est présentée en Listing 12.

Dans cette taxonomie, l'entité IRU capte la position, le cap et le roulis de l'avion à partir de l'environnement. L'entité NavMMI permet d'abstraire l'interaction du pilote afin de fournir le cap cible qu'il a spécifié. L'entité Aileron propose une interface Control afin de pouvoir agir sur l'environnement.

La conception architecturale de l'application est ensuite réalisée au travers de la définition des composants contextes et contrôleurs, comme le présente le Listing 13.

Cette conception est conforme à la logique applicative du mode de suivi du cap. En effet, le composant contexte IntHeading calcule un cap intermédiaire à partir 1) du cap courant heading, que lui renseigne l'entité IRU, et 2) du cap cible targetHeading

```
device IRU {
   source heading as Float (frequency 200ms);
   source position as Coordinates;
   source roll as Float;
   ...
   action Deactivate;
   raises FailureException;
}

device NavMMI {
   source targetHeading as Float;
   ...
   action DisableMode;
   action Display;
}

action Control {
   incline(targetRoll as Float);
}

device Aileron {
   action Control;
}
```

Listing 12: Extrait de la taxonomie des entités

```
context IntHeading as Float {
  interaction {
      when provided heading from IRU;
      get targetHeading from NavMMI in 100 ms
           (mandatory catch);
      always publish;
   }
}
```

Listing 13: Déclaration du composant contexte IntHeading

rentré par le pilote, que lui renseigne l'entité NavMMI. A partir de ce cap intermédiaire et de la valeur courante du roulis (roll) obtenue à partir de l'entité IRU et représentant la rotation de l'avion sur un axe longitudinal, le composant contexte TargetRoll calcule ensuite un roulis cible. Ce dernier est utilisé par le composant contrôleur AileronController afin de contrôler les ailerons en conséquence.

5.2.3 *Couche non-fonctionnelle*

La conception architecturale de l'application est ensuite enrichie avec la prise en compte des exigences non-fonctionnelles. Par exemple, dans la taxonomie, l'entité IRU est déclarée comme pouvant lever une erreur de type FailureException.

Nous enrichissons également la conception architecturale fonctionnelle avec des contraintes de QoS, comme requis par le cahier des charges. En effet, l'exigence **Req1** définit la contrainte haut-niveau de temps maximum d'exécution pour la chaîne fonctionnelle de suivi du cap, de 650ms. En appliquant notre approche d'intégration de QoS, nous attachons cette contrainte de WCET au composant contrôleur responsable de la chaîne fonctionnelle de suivi du cap : AileronController, comme illustré sur la Figure 9.

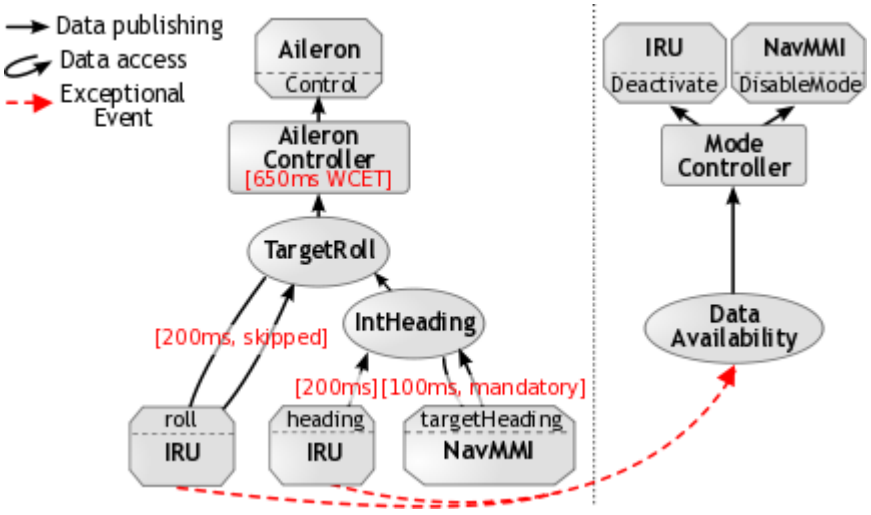

FIGURE 9: Extrait de la conception de l'application de guidage

Par suite, cette contrainte de WCET se décline dans le cahier des charges en des contraintes de fraîcheur de données de navigation et de temps maximum d'exécution pour la chaîne fonctionnelle de détection et traitement de dysfonctionnement ou panne d'un capteur. L'exigence **Req2** est prise en compte dans

la spécification de l'application au travers de la contrainte de fréquence de 200ms sur l'entité IRU pour l'envoi du cap (*heading*). La contrainte de temps de réponse de 100ms pour l'acquisition de la donnée `targetHeading` de NavMMI par le composant contexte `IntHeading` est ensuite déposée en cohérence avec la contrainte de WCET sur la chaîne fonctionnelle, comme l'illustre la Figure 9. En outre, le Listing 13 présentait l'exigence *mandatory catch* associée à l'exigence de temps de réponse de 100 ms. Cette catégorie d'exception signifie que toute erreur doit être systématiquement traitée par le composant, afin de ne pas bloquer l'exécution du système.

En couplant la QoS et le mécanisme d'exception au sein de DiaSuite, la détection du non-respect à l'exécution de la contrainte de temps de réponse de 100ms pour l'acquisition de la donnée `targetHeading` s'accompagne de la levée d'une exception de type `TimeoutException`.

Nous avons présenté dans le Chapitre 3, notre outil pour déterminer la cohérence des contraintes temporelles de QoS qui sont spécifiées. De récents travaux ont intégré à la méthodologie DiaSuite un support formel à l'aide d'UPPAAL [13] pour la vérification de cohérence des contraintes temporelles de QoS [43]. Le fonctionnement du support est le suivant. A partir d'une spécification DiaSpec, est généré un ensemble de modèles formels UPPAAL. Ces modèles sont représentés sous la forme d'automates temporisés ; chaque automate décrit le comportement d'un composant DiaSpec [1]. La détection d'incohérence entre les contraintes temporelles est ensuite automatiquement effectuée par le *model-checker* UPPAAL. Par exemple, si l'entité NavMMI mettait plus de 200ms pour envoyer une donnée en réponse de la sollicitation du composant contexte `IntHeading`, un état d'inter-blocage serait détecté.

5.2.4 *Couche de supervision*

La Figure 9 présente la vue graphique du mode de suivi du cap de l'application de guidage avionique, représentant le flot de contrôle.

Toute erreur pouvant survenir au niveau de l'entité IRU est envoyée au composant contexte `DataAvailability`. Ce dernier reçoit donc, au niveau de la couche de supervision, des événe-

1. Pour plus d'informations sur cette translation, le lecteur peut se référer à l'adresse suivante : http ://diasuite.inria.fr/validation

ments pouvant être soit une panne (représentée par l'exception FailureException), soit une violation de QoS (représentée par l'exception TimeoutException). Dans ce cadre, le composant DataAvaiability peut déterminer une présomption de panne du fait de l'occurrence de plusieurs violations de QoS,. Ce dernier détermine la disponibilité des données requises et l'envoie au composant contexte ModeController qui peut agir en désactivant l'entité IRU et en désactivant la capacité de sélectionner ce mode sur la console de navigation, représentée par l'entité NavMMI. Comme nous l'avons décrit dans le Chapitre 4, du fait de la contrainte *mandatory* qui est spécifiée, le composant contexte IntHeading doit systématiquement compenser l'erreur pouvant survenir au sein de l'entité NavMMI, par exemple en fournissant une valeur par défaut. Cette contrainte permet de poursuivre temporairement l'exécution de l'application.

Comme nous l'avons détaillé dans le Chapitre 4, les traitements effectués pour la supervision de l'application sont spécifiés en DiaSpec en utilisant des chaînes fonctionnelles SCC séparées. Dans le domaine avionique, cette couche de supervision est similaire à la surveillance de l'application et au déclenchement de reconfiguration si nécessaire, comme le détaillent les exigences **Req3** et **Req4** du cahier des charges de notre application de guidage. En effet, ces exigences ont pour but de :

- Informer le pilote en cas de panne d'un périphérique ou de données non-disponibles ;
- Désactiver les modes qui font intervenir des données fournies par des périphériques en état avéré de dysfonctionnement ou de panne ;
- Enregistrer toute information sur les échanges de données au sein de l'application, afin de pouvoir ensuite les utiliser lors de la phase de maintenance.

Par exemple, la partie de droite sur la Figure 9 représente la chaîne de supervision dédiée à la désactivation des modes de navigation dépendants de données non-disponibles.

Dans le contrat de QoS associé au composant contexte TargetRoll, la contrainte de temps de réponse maximal de 200ms a été déposée pour la récupération de la donnée roll provenant de l'entité IRU. Dans le cas où cette contrainte n'est pas respectée à l'exécution, une erreur de type TimeoutException est envoyée. Le composant contexte DataAvailability est en charge de la récupération de toute erreur provenant de la logique applicative. La politique de gestion d'erreurs de QoS ici est le choix de considérer, au bout de trois violations consécutives de QoS par le même composant, la panne du

composant, laquelle nécessite une désactivation dudit composant.

La Figure 9 illustre donc qu'au travers du composant contexte DataAvailability, la disponibilité de l'entité IRU est vérifiée. Cette information est ensuite utilisée par le composant contrôleur ModeController afin d'activer ou de désactiver des modes de navigation ainsi que les périphériques fautifs.

5.3 IMPLÉMENTATION

Dans la méthodologie DiaSuite, un support de programmation dédié est généré, pour le support de développement, aussi bien fonctionnel [27, 28] que non-fonctionnel [50, 80]. Cette génération de *frameworks* de programmation Java dédiés assure la conformité entre les phases de conception et d'implémentation.

5.3.1 *Couche fonctionnelle*

Un extrait de la classe abstraite générée directement à partir de la spécification du composant contexte IntHeading est présenté en Listing 14.

```
1 public abstract class AbstractIntHeading {
2    public abstract Float onHeadingFromIRU
3      (Float heading, Binding binding) ;
4    ...
5 }
```

Listing 14: Extrait de la classe générée pour le composant contexte IntHeading

Lorsque le développeur étend la classe abstraite AbstractIntHeading, il doit implémenter la méthode abstraite onHeadingFromIRU afin de recevoir toute donnée publiée par ce périphérique, comme l'illustre le code présenté en Listing 15.

Comme le présente le Listing 15, le contexte IntHeading récupère un binding qui lui permet ensuite d'accéder au contenu de NavMMI.

5.3.2 *Couche non-fonctionnelle*

De même que pour les déclarations fonctionnelles, les déclarations non-fonctionnelles peuvent être tracées depuis les spécifications DiaSpec jusqu'à l'implémentation. Dans le cas du

```
1  public class IntHeading extends AbstractIntHeading {
2     public Float onHeadingFromIRU
3      (Float heading, Binding binding) {
4        NavMMI mmi = binding.navMMI();
5        Float targetHeading = mmi.getTargetHeading(
6           new TargetHeadingContinuation(){
7              public Float onError(){
8                 return DEFAULT_VALUE;
9              }
10          }
11       );
12       return controllerPID.compute(heading, targetHeading);
13    }
14 }
```

Listing 15: Extrait de la classe concrète IntHeading impl 'ementée par le développeur

support de QoS que nous apportons, la gestion de containers dédiés à la surveillance des contraintes à l'exécution est réalisée de façon complètement transparente pour le développeur. Par exemple, cette couche de surveillance mesure le temps passé par le composant contexte IntHeading afin de récupérer la donnée targetHeading. Si ce temps est supérieur à la contrainte de 100ms spécifiée dans le contrat associé au composant IntHeading, une erreur de QoS est automatiquement levée par le *framework*.

Dans le cas de la gestion d'erreur, un support spécifique est généré également [80]. Par exemple, dans la spécification du composant contexte IntHeading, le contrat d'interaction définit une déclaration mandatory catch. Celle-ci est utilisée afin d'obliger le développeur à gérer toute erreur pouvant survenir lors de la récupération de la donnée targetHeading de l'entité NavMMI.

En effet, lors de la récupération de la donnée targetHeading, il est possible qu'une panne entraîne une erreur de type *timeout*, due à l'exigence de 100 ms de QoS. Or, comme le présente le Listing 15, l'exigence *mandatory catch* est déposée, signifiant que le composant contexte IntHeading doit systématiquement traiter toute erreur qu'il peut récupérer, due à un souci de disponibilité de la donnée, ou encore à la défaillance du périphérique. Lors de l'extension de la classe abstraite AbstractIntHeading, c'est-à-dire dans la classe Java IntHeading, une continuation est générée et doit être remplie par le développeur. Cette continuation permet de spécifier le traitement en cas de réception d'une erreur : l'envoi d'une valeur par défaut (returnDEFAULT_VALUE).

5.3.3 *Couche de supervision*

Lorsque l'on considère la couche de supervision, le comportement à adopter en cas de violation de contrats de QoS est à implémenter de façon séparée de la logique applicative fonctionnelle. Cette approche permet la séparation des préoccupations et peut être généralisée afin de tenir compte d'autres aspects non-fonctionnels. Par exemple, des contraintes de sécurité peuvent être spécifiées pour enrichir la spécification DiaSpec [63]. La violation de ces contraintes entraîne la publication d'un événement exceptionnel pouvant être traité par d'autres composants DiaSpec, qui peuvent ainsi intervenir sur la partie fonctionnelle afin de compenser ces erreurs.

5.4 TEST

L'implémentation de chaque chaîne fonctionnelle SCC peut être testée de façon indépendante. Par exemple, l'aspect fonctionnel de l'application peut être testé à l'aide d'un environnement externe simulé [20, 21]. Cette simulation est particulièrement recommandée dans les domaines où la représentation concrète de l'environnement pour le test n'est que peu disponible, comme c'est le cas dans les domaines avioniques, automobiles ou encore ferroviaires, où les tests d'intégration se font généralement sur des bancs, après tests applicatifs.

Afin de tester notre application de guidage, tant sur ses aspects fonctionnels que non-fonctionnels, nous avons dû faire face à la contrainte que l'environnement extérieur de l'application est en fait l'avion, qui, au travers de ses capteurs et actionneurs, s'interface avec l'application, comme l'illustre la Figure 10. Le moyen de test requis devait donc être suffisamment représentatif du système physique de l'avion, tout en étant compatible des contraintes inhérentes à un tel système.

Afin de pallier ce problème, nous avons choisi de tester notre application en la couplant au simulateur de vol réaliste FlightGear [86], comme l'illustre la Figure 11.

Les personnes en charge du test de l'application peuvent ainsi utiliser une bibliothèque Java qui s'interface avec FlightGear afin d'implémenter des versions simulées des composants entités sans modifier la logique applicative codée au préalable, comme l'illustre l'extrait suivant de l'implémentation d'une entité `IRU` simulée.

L'entité `SimulatedIRU` est implémentée au moyen de l'héritage de la classe `AbstractIRU` fournie par le *framework*

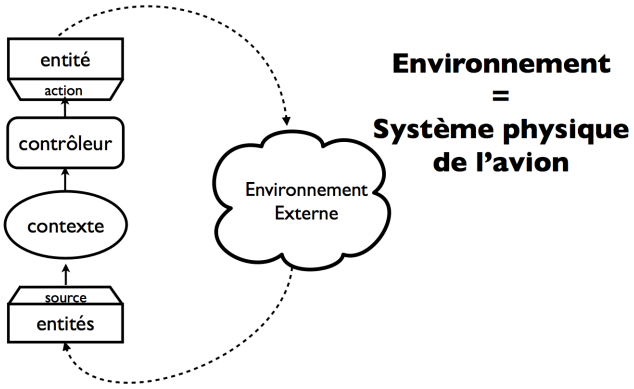

FIGURE 10: Test de l'application : Environnement avion

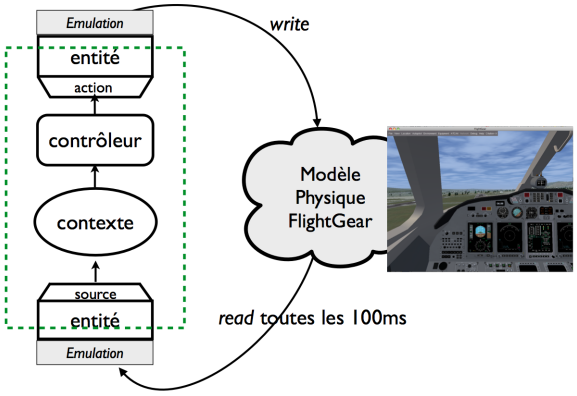

FIGURE 11: Test de l'application : Environnement FlightGear

```
1 public class SimulatedIRU extends AbstractIRU
2  implements SimulatorListener {
3    public SimulatedIRU(FGModel model){
4       model.addListener(this) ;
5    }
6    public void simulationUpdated(FGModel model){
7       publishPosition(model.getInertialPosition());
8    }
9 }
```

Listing 16: Extrait de l'implémentation d'une entité IRU simulée

de programmation. Afin d'interagir avec l'environnement simulé, l'entité implémente l'interface SimulatorListener. Cette interface définit une méthode simulationUpdated, appelée périodiquement par la bibliothèque de simulation. Le paramètre model permet de lire ou d'écrire l'état courant du simulateur FlightGear. La position de l'avion est ainsi publiée dans cet extrait de l'implémentation d'une entité IRU simulée, en appelant la méthode publishPosition de la classe AbstractIRU.

Une fois que les entités simulées sont implémentées, la partie fonctionnelle de l'application de guidage est testée au moyen de la simulation des différents modes de navigation implémentés. Par exemple, une interface permet de rentrer une valeur de cap cible. Le test se poursuit en vérifiant que l'avion au sein de FlightGear prend en compte cette commande de nouveau cap à atteindre, et modifie son orientation en conséquence. La Figure 12 présente une capture d'écran de cet environnement de test [2].

2. La spécification DiaSpec ainsi qu'une vidéo de démonstration de cette application, sont disponibles à l'adresse suivante : http ://diasuite.inria.fr/avionics/51.

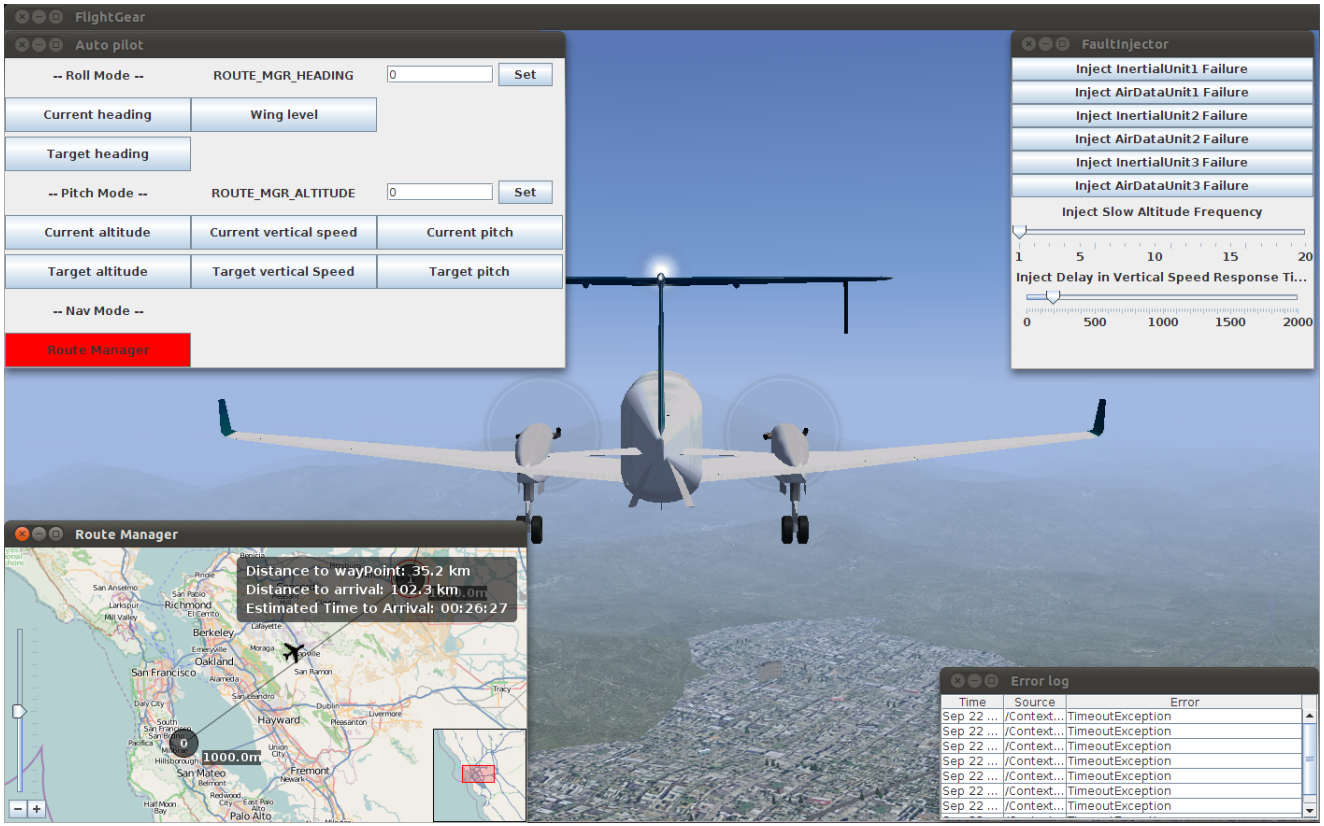

FIGURE 12: Capture d'écran d'un vol simulé

Sur la fenêtre principale, le simulateur FlightGear permet de contrôler et de visualiser l'avion simulé. Dans le coin en haut à gauche, le testeur peut sélectionner un des modes de navigation implémentés, au travers de l'interface dédiée. Sur la Figure 12, le mode "Route Manager" de suivi automatique d'un plan de vol est sélectionné. Ledit plan de vol a été au préalable rentré en définissant des points de passage sur la carte géographique, comme illustré dans le coin en bas à gauche.

L'environnement simulé permet également de tester la chaîne de supervision de l'application. Pour ce faire, nous avons défini une interface d'injection de fautes, représentée par la fenêtre "Fault Injector" dans le coin en bas à droite sur la Figure 12. Cette interface permet d'injecter non seulement des simulations de panne de périphériques, mais également des retards dans la communication de données. Cette dernière partie permet de simuler le cas de violation de contrat de QoS et la réponse apportée par la couche de supervision. En l'occurrence, toute violation de QoS conduit à l'enregistrement (*log*) de ces événements dans un fichier journalisé dédié, situé en bas à droite sur la Figure 12, et éventuellement à la reconfiguration de l'application en cas de violations successives de QoS, au travers de la désactivation de mode de navigation.

5.5 EVALUATION

Dans cette section, nous évaluons l'étude de cas que nous avons réalisée, selon les critères retenus de cohérence des aspects fonctionnels et non-fonctionnels, et de conformité des exigences entre les phases de développement.

5.5.1 *Cohérence*

Afin d'assurer la cohérence entre les aspects fonctionnels et non-fonctionnels durant la phase de conception, la méthodologie DiaSuite repose sur un langage de conception unique, et sur un style architectural commun [28]. Contrairement aux approches basées sur la capacité à représenter de multiples vues d'une même application, de façon indépendante, comme c'est le cas pour la collection de diagrammes UML (voir Partie "Contexte"), le langage DiaSpec intègre des déclarations fonctionnelles et non-fonctionnelles, ce qui contribue à éviter la plupart des incohérences pouvant survenir lors de la spécification d'une application. Par exemple, la cohérence entre les déclarations de levée d'erreurs peut être vérifiée de façon statique, étant

donné que ces déclarations consistent en un raffinement des contrats d'interaction qui décrivent le flot de contrôle de l'application [42]. Si le concepteur déclare qu'une entité va pouvoir lever une exception, alors des vérifications effectuées à la compilation assurent qu'une déclaration de levée d'erreurs est bien spécifiée pour chacun des composants déclarés comme abonnés à l'entité, c'est-à-dire qui a été déclaré comme requérant une donnée captée et transmise par cette entité. Au regard des déclarations de QoS, leur cohérence est directement vérifiée à la conception à partir de la génération d'équations [50] et plus récemment du modèle formel d'automates temporisés UPPAAL généré à partir des spécifications DiaSpec [43], permettant de détecter une inconsistance entre des contraintes de temps : une telle problématique dans le modèle résulterait en effet en un inter-blocage. Ainsi, contrairement aux approches existantes qui sont également basées sur un aspect génératif pour garantir la traçabilité des exigences, notre approche offre un plus haut niveau d'abstraction pour l'*early-validation* des contraintes temporelles.

Durant la phase d'implémentation, la génération du *framework* de programmation dédié à la spécification assure la cohérence entre les déclarations de levée d'erreurs qui aura été préalablement vérifiée. En effet, le support généré pour la levée d'erreurs, comme dans le composant contexte `DataAvailability`, empêche le développeur d'implémenter du code ad-hoc pour la propagation d'erreurs. En ce qui concerne les déclarations de QoS, le support généré consiste en des composants containers qui sont dédiés à la surveillance des contraintes de temps spécifiées dans le contrat durant la phase de conception. Ces containers de surveillance sont intégrés au *framework* de programmation et sont complètement transparents pour le développeur. Ils forment des gardes qui, s'ils n'assurent pas la cohérence de l'application lors de son exécution, permettent en revanche de superviser les contraintes de temps, et de lever des erreurs en cas de non-respect de ces contraintes. Cette levée d'erreurs consiste en l'envoi d'une information au superviseur, de la non-tenue du contrat spécifié, identifiant le composant responsable. De ce fait, combinées avec le support de simulation et d'injection de fautes, les gardes de QoS permettent de guider la vérification de la cohérence à l'exécution [50]. Par exemple, une défaillance de capteurs pourrait être déduite à l'exécution, suite à la levée consécutive de multiples erreurs de QoS.

5.5.2 *Conformité*

Afin d'assurer la conformité du développement de l'application avec les exigences haut-niveau de l'application, DiaSuite fournit du support de validation tout au long du cycle de développement. Nous illustrons dans cette partie comment ce support de validation permet de guider la vérification de la conformité, au travers de l'exigence **Req3**. Cette exigence a été fournie en entrée de la méthodologie, comme un point de spécification haut-niveau de l'application de guidage. Elle indique que tout dysfonctionnement ou panne d'un capteur doit être systématiquement signalé au pilote, dans un délai maximal de 300 ms, représentant une contrainte de WCET sur la chaîne fonctionnelle de supervision.

Au niveau de la phase de conception, cette exigence conduit à la spécification d'une chaîne fonctionnelle SCC dédiée à la supervision de l'application. Cette chaîne est en charge de l'affichage de toute défaillance de capteur, à destination du pilote, via la console de visualisation. Le support de vérification que nous avons mis en place, permet de vérifier de façon statique que tout événement exceptionnel envoyé par l'entité IRU, du fait de son dysfonctionnement ou du fait de retards répétés en termes de temps de réponse, résulte systématiquement en le déclenchement de l'action Display sur l'entité d'affichage NavMMI.

En phase d'implémentation, la génération du *framework* de programmation dédié à la spécification assure la conformité fonctionnelle de l'application avec les spécifications de flots de données et de contrôle [28]. En particulier, l'intégrité de communication est préservée, permettant de garantir que seuls les liens de communication spécifiés entre composants conduiront à une capacité de communication entre lesdits composants. Le développeur est en effet contraint de développer la méthode découlant de la méthode abstraite générée, qui représente la transmission de données vis à vis du composant avec lequel il est relié. De même, le support non-fonctionnel fourni, au niveau des containers de surveillance de QoS à l'exécution, préserve les exigences spécifiées sur l'architecture [50].

En outre, le support de test généré directement à partir de la spécification fournit une interface d'injections d'erreurs, à la fois de sûreté de fonctionnement (simulation de panne d'une entité) et temporel (simulation de retard dans les communications). Ce support permet de valider l'exigence **Req3** même si l'entité IRU

n'est pas encore implémentée, en utilisant des versions simulées.

Enfin, nous pouvons dire que cette étude de cas a également été reproduite dans le contexte du développement d'une application de guidage pour un drone : le système Parrot A.R. Drone [3]. Cette application a permis de mettre en exergue les capacités d'abstraction offertes par DiaSuite et la facilité d'implémentation d'un tel système, à partir d'un système similaire existant. Les différences majeures se situaient au niveau des équipements requis pour la localisation de l'A.R. Drone.

5.6 CONCLUSION

Dans ce chapitre, nous avons présenté une étude de cas avionique reflétant la mise en application de notre approche sur un cas concret et offrant une base pour l'évaluation, au regard de deux critères.

En premier lieu, nous avons montré comment l'intégration de concepts de QoS et du mécanisme de gestion d'erreurs, de façon unifiée au sein de la méthodologie DiaSuite, permettent de préserver la cohérence des aspects fonctionnels et non-fonctionnels d'une application, contrairement aux approches basées sur un système en "multi-vues" qui ne traiterait pas de la cohérence entre ces vues. Egalement, cette approche permet de découpler le traitement applicatif fonctionnel du traitement non-fonctionnel, en reposant sur la couche de supervision de niveau système, pour le traitement correctif approprié : *logs* d'erreurs, affichages au pilote ou encore reconfiguration applicative (choix d'autres instances de *devices* à déployer de façon semi-dynamique).

En second lieu, nous avons montré comment notre approche systématique de traitement de la QoS tout au long du développement logiciel préserve la traçabilité des exigences, en reposant sur le mécanisme de génération automatique de code.

3. La spécification DiaSpec ainsi qu'une vidéo de démonstration de cette application, sont disponibles à l'adresse suivante : http ://diasuite.inria.fr/avionics/ardrone.

6

CONCLUSION

Afin de proposer une approche pour le développement de logiciels intégrant des concepts de QoS, nous avons étudié d'une part, les pré-requis, et d'autre part, les approches existantes dans ce domaine. Tout d'abord, il ressort qu'une approche guidée par la conception permet de supporter la traçabilité des exigences, tout en guidant et contraignant le développeur durant la phase d'implémentation, offrant une préservation des exigences entre chaque phase du processus de développement. En revanche, dans ce cadre, nous avons noté que les approches existantes dites MDE proposent de traiter les exigences de QoS comme une vue séparée de la vue fonctionnelle applicative, rendant difficile la tâche de vérification avec les autres vues, aussi bien fonctionnelles que non-fonctionnelles. D'autres approches proposent de spécifier finement et de façon unifiée, les aspects fonctionnels et non-fonctionnels d'une application. Seulement, cette granularité fine conduit à considérer les exigences de bas-niveau (*Low-Level Requirements*) dès la phase de Conception, arrivant trop tôt dans le développement logiciel. En effet, l'étape de raffinement depuis la capture du besoin avec la spécification de la chaîne fonctionnelle, est manquante. De ce fait, un compromis est une approche permettant de spécifier les aspects fonctionnels et non-fonctionnels d'une application de façon unifiée, avec une granularité adaptée à la phase de spécification, couvrant ainsi la vue haut-niveau d'un système, garantissant cohérence et conformité des exigences, permettant d'assurer ensuite la traçabilité de ces exigences haut-niveau.

Ainsi, pour notre approche de gestion de QoS, nous avons fait levier sur la méthodologie de développement outillée DiaSuite afin d'offrir du support de QoS tout au long du cycle de développement logiciel, adapté à l'artefact associé à chacune des phases [50]. Notre approche étend la méthodologie DiaSuite [28] en offrant du support pour la spécification, la validation et la surveillance des exigences temporelles à l'exécution.

Dans notre thèse, afin de pouvoir prendre en compte les contraintes de QoS dès les premières phases du cycle de développement, nous avons enrichi DiaSuite avec du support de QoS. Nous avons tout d'abord proposé une syntaxe dédiée à la spécification de contraintes temporelles, adaptée à la phase du processus de développement logiciel, et en cohérence avec

le modèle fonctionnel du paradigme SCC. L'intégration des contrats de QoS se fait directement de façon unifiée au niveau des contrats d'interaction décrivant les communications entre composants.

Nous avons également enrichi le modèle d'exceptions DiaSuite afin de pouvoir tracer toute violation de contrat de QoS et permettre de spécifier des stratégies de compensation de ces violations au niveau système, par exemple au travers d'une reconfiguration applicative (désactivation de certaines capacités de l'application).

En outre, nous avons également étendu le compilateur DiaSpec, afin d'offrir, pour chaque contrat de QoS déposé sur un composant, la génération d'un container dédié à la surveillance des contraintes de QoS, de façon complètement transparente pour le développeur. Cette approche permet de garantir que la tenue des contraintes de QoS spécifiées sera vérifiée à l'exécution, empêchant le développeur d'introduire des erreurs dans les contraintes temporelles. D'autre part, la transparence des mécanismes de surveillance des contraintes à l'exécution permet aux développeurs de se concentrer uniquement sur l'implémentation de la logique applicative et d'autre part, de garantir que le code de surveillance sera bien exécuté, avec les contraintes spécifiées.

Nous avons montré comment cette approche dédiée permet de guider les acteurs dans le raffinement systématique des exigences non-fonctionnelles et d'assurer la traçabilité des exigences de QoS en générant des contraintes numériques.

Enfin, nous avons validé notre approche sur une étude de cas dans le domaine avionique où de telles exigences de QoS sont critiques.

L'approche que nous avons proposée dans cette thèse adresse la QoS temporelle, requise dans les domaines critiques, tels que l'avionique, le ferroviaire, ou encore l'automobile. Il est en effet important d'assurer que les applications développées dans ces domaines respectent les exigences temporelles qui leur sont allouées. Ceci permet de préserver l'**intégrité temporelle** des informations calculées, contribuant à assurer la sécurité des passagers et de l'équipage.

Lors de la présentation de notre approche, nous avons évoqué la capacité pour notre approche, de s'étendre à d'autres QoS. En particulier, il serait intéressant d'adresser les contraintes de type consommation mémoire ou CPU, raffinée selon la phase du cycle de développement. Si le fait d'adresser de telles catégories d'exigences requiert certains pré-requis, notre approche d'intégration de QoS dans le développement de systèmes offre

l'avantage d'être extensible et transparente pour le développeur, lors de la phase d'implémentation. Le container pourra ainsi être enrichi afin d'étendre son spectre de QoS à surveiller.

Par ailleurs, il est également intéressant de noter que la QoS temporelle peut apporter un service supplémentaire dans le cadre d'autres domaines applicatifs. Le chapitre suivant détaille ces points.

7

GÉNÉRALISATION

L'approche de QoS que nous proposons a été mise en œuvre dans le cadre de la méthodologie DiaSuite, et appliquée à dans divers domaines, dont l'étude de cas avionique que nous avons présentée dans ce manuscrit. Cependant, cette approche peut tout à fait être généralisée à d'autres domaines, qui n'ont pas nécessairement de contraintes temporelles fortes, mais qui présentent un besoin de qualité de service. D'autres exemples où notre approche s'applique concerne également les domaines classant la pertinence d'une information selon ses caractéristiques temporelles.

La méthodologie DiaSuite adresse les domaines couverts par le paradigme "Sense / Compute / Control", tels que l'avionique, la robotique ou encore la domotique. Afin de pouvoir concevoir des applications fiables dans ce dernier domaine, il est important de pouvoir se doter d'une base temporelle afin d'évaluer les réponses des systèmes et de prévoir l'action corrective associée. Par exemple, considérons l'exemple d'une application de détection d'un incendie dans un immeuble. Cette application s'architecture autour de deux périphériques, respectivement pour la détection de fumée dans la pièce et pour le signalement d'une montée en température. A partir de la combinaison de ces informations, l'application déclenche une alarme et active les extincteurs afin d'étouffer l'incendie. Dans cette application, la QoS temporelle permet de justifier de la validité unitaire des informations collectées (fraîcheur de la donnée de température) et également de la cohérence temporelle des informations collectées (intervalle de temps borné pour la considération des données de fumée et de température).

Notre approche pourrait également être applicable dans d'autres domaines. De récents travaux sur la méthodologie DiaSuite adressent l'aide à la personne. Dans ce cadre, les contraintes temporelles apportent une fiabilité et un confort important pour le public visé. Considérons par exemple une application s'exécutant sur un *smartphone* ou une tablette, qui, en cas de détection de chute d'une personne, affiche un message à acquitter, et appelle un membre de la famille en cas de non-acquittement, pour la prévenir d'un risque éventuel d'inconscience de la personne. L'ajout d'une contrainte temporelle pourrait viser à définir le

temps laissé à la personne pour acquitter le message. Une autre contrainte pourrait être déposée afin de borner la tentative de communication avec le membre de la famille, pour permettre de joindre un autre membre, ou encore décider de prévenir les secours dans le cas où le temps écoulé depuis la chute dépasse un délai fixé. Cet aspect est non-négligeable dans l'aide aux personnes âgées. De ce fait, la méthodologie que nous proposons peut être généralisable à d'autres domaines, où elle apportera cohérence et conformité.

8

PERSPECTIVES

La présentation de notre approche d'intégration de concepts de QoS tout au long du cycle de développement logiciel, a permis de démontrer l'importance de la prise en compte des exigences de QoS au plus tôt dans le développement, en accentuant l'apport offert par l'architecture. Les travaux que nous avons conduits et validés au travers d'une étude de cas peuvent faire l'objet de plusieurs extensions. Les perspectives futures à cette thèse sont variées.

Prise en compte de contraintes de QoS étendues.
Dans le cadre de la mise en œuvre de notre approche, nous nous sommes concentrés sur l'axe temporel. Dans la classification de la QoS, le domaine temporel fait partie de la catégorie plus large de la performance qui couvre d'autres types de contraintes, telles que la consommation mémoire ou encore la puissance en termes de CPU. En particulier, la spécification de ces contraintes, ainsi que leur tenue à l'exécution (par exemple, au moyen de mécanismes tels que l'ordonnancement), ainsi que leur surveillance (par exemple, à des fins de maintenance) est requise dans les domaines embarqués critiques. Il devient intéressant d'étendre notre approche à d'autres contraintes de QoS, et d'envisager un déploiement plus fin vers des ADLs couvrant les couches basses des systèmes, afin de combiner adaptabilité de la granularité de QoS et apport des garanties de la plateforme d'exécution.
Dans cette optique, une extension immédiate est l'intégration de notre approche dans une approche globale adressant la résilience des systèmes. En effet, nos travaux ont été intégrés de façon unifiée avec le concept d'exceptions de DiaSuite. En ayant une vue plus haut-niveau, cette combinaison permettrait de spécifier des politiques de tolérance aux fautes basées sur des pannes induites par des fautes de QoS, en se rapprochant des travaux effectués sur des patrons de tolérance aux fautes adaptables [96]. Des travaux sont en cours pour annoter DiaSpec avec de telles politiques et peuvent s'intégrer aisément avec les exceptions de QoS afin de permettre la gestion de ces politiques et de faciliter les évolutions à chaud, en reposant sur les capacités de reconfiguration d'une plateforme d'exécution telle que FraSCAti [91].

Robotique.
Dans le domaine de la robotique, des approches existantes se concentrent sur l'apprentissage des robots, que ce soit au niveau de la parole [62] ou des gestes [83], dans un but de sociabilisation du robot. D'autres approches s'intéressent aux robots dans l'espace [48]. Une application possible de notre approche émerge très vite, dès lors que l'on considère par exemple le temps de trajet d'un voyage dans l'Espace. Afin de pouvoir étudier d'autres planètes, nous reposons sur des robots qui photographient et étudient le sol, afin de transmettre ces informations à la Terre. Les travaux qu'ont menés Bensalem *et al.* dans ce cadre insistent sur l'importance des caractéristiques temporelles pour le robot [15]. Afin de pouvoir proposer une autonomie des robots, ils préconisent une planification des tâches, en conformité avec les temps de communication et d'émission / réception des informations entre la Terre et Mars, tout en prévoyant une marge temporelle due aux incertitudes sur la durée des tâches à effectuer par le robot. Cette QoS temporelle est à définir, intégrer et supporter tout au long du développement du robot. Leur approche de développement, incrémentale se base sur une architecture en trois couches : niveau fonctionnel, décisionnel et de contrôle à l'exécution [11]. Le couplage de notre proposition pour l'intégration de QoS dans le développement de logiciels, avec cette approche, permettrait de proposer une spécification du robot tenant compte des différentes facettes du logiciel, fonctionnelles et non-fonctionnelles, tout en séparant les préoccupations. Dans ce cadre, il serait intéressant de travailler sur un *mapping* entre les niveaux de composants DiaSpec et les niveaux architecturaux *LAAS*. Un premier *mapping* peut être effectué, entre les composants entités qui permettent à l'application de s'interfacer avec l'environnement externe, et le niveau décisionnel *LAAS*.

Facteurs Humains.
Les logiciels actuels tendent à s'orienter vers un monde ouvert, où l'ergonomie, la facilité d'utilisation font partie des facteurs humains à prendre en compte dans le développement de tels logiciels. Par exemple, dans le domaine critique de l'avion, tout logiciel interagissant avec l'utilisateur, *e.g.*, le système de gestion des alertes, doit tenir compte d'exigences sur l'affichage des alertes à l'équipage, afin d'assurer la bonne prise en compte des informations (pas de multiples interprétations possibles, actions correctives associées claires). Egalement, dans l'orientation des travaux DiaSuite vers l'aide à la personne [10], il est indispensable d'adapter la présentation des informations fournies par le logiciel, au public visé. Actuellement, parmi les approches existantes, il existe peu de méthodologie pour véritablement guider le déve-

loppement de logiciels intégrant de telles contraintes, en plus de leurs réponses fonctionnelles. Un cadre de travail intéressant pour la suite de nos travaux intervient sur la mise en place de contraintes de QoS temporelles, à partir de besoins IHM (Interface Homme-Machine) identifiés pour l'aide à la personne [10].

BIBLIOGRAPHIE

[1] Jan Øyvind Aagedal. *Quality of service support in development of distributed systems*. PhD thesis, University of Oslo, 2001. (Cited on pages 2 and 12.)

[2] Jonathan Aldrich, Craig Chambers, et David Notkin. Archjava : connecting software architecture to implementation. Dans *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 187–197. IEEE, 2002. (Cited on page 18.)

[3] Robert Allen et David Garlan. The wright architectural specification language. *Rapport technique CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science*, 1996. (Cited on page 12.)

[4] Musab Alturki, Dinakar Dhurjati, Dachuan Yu, Ajay Chander, et Hiroshi Inamura. Formal specification and analysis of timing properties in software systems. Dans *Fundamental Approaches to Software Engineering*, pages 262–277. Springer, 2009. (Cited on page 13.)

[5] Peter Amey. Correctness by construction : Better can also be cheaper. *CrossTalk : the Journal of Defense Software Engineering*, 2 :24–28, 2002. (Cited on pages 3 and 52.)

[6] arinc653. ARINC 653, system partitioning and scheduling (Aeronautical Radio, Inc.), 2003. (Cited on page 1.)

[7] arinc664. ARINC 664, AFDX : Avionics Full DupleX switched ethernet (Aeronautical Radio, Inc.), 2005. (Cited on page 1.)

[8] ARP4761. ARP4761, Aerospace Recommended Practice - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment (SAE ARP4761), 1996. (Cited on page 64.)

[9] Algirdas Avizienis, J-C Laprie, Brian Randell, et Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1) :11–33, 2004. (Cited on page 54.)

[10] Emilie Balland, Charles Consel, Bernard N'Kaoua, et Hélène Sauzéon. A case for human-driven software development. Dans *Proceedings of the 2013 International Conference*

on Software Engineering, pages 1229–1232. IEEE Press, 2013. (Cited on pages 88 and 89.)

[11] Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Félix Ingrand, et Joseph Sifakis. Incremental component-based construction and verification of a robotic system. Dans *ECAI*, pages 631–635, 2008. (Cited on page 88.)

[12] Steffen Becker, Heiko Koziolek, et Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1) :3–22, 2009. (Cited on page 48.)

[13] Gerd Behrmann, Alexandre David, et Kim G Larsen. A tutorial on uppaal. Dans *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004. (Cited on page 68.)

[14] R Ben Halima, Khalil Drira, et Mohamed Jmaiel. A qos-oriented reconfigurable middleware for self-healing web services. Dans *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 104–111. IEEE, 2008. (Cited on page 35.)

[15] Saddek Bensalem, Matthieu Gallien, Félix Ingrand, Imen Kahloul, et Thanh-Hung Nguyen. Toward a more dependable software architecture for autonomous robots. *IEEE Robotics and Automation Magazine*, 16(1) :1–11, 2009. (Cited on page 88.)

[16] Antonia Bertolino et Raffaela Mirandola. Cb-spe tool : Putting component-based performance engineering into practice. Dans *Component-Based Software Engineering*, pages 233–248. Springer, 2004. (Cited on page 13.)

[17] Noury MN Bouraqadi-Saâdani et Thomas Ledoux. Le point sur la programmation par aspects. *Technique et Sciences Informatiques*, 20(4) :505–528, 2001. (Cited on page 14.)

[18] Jean-Michel Bruel, João Araújo, Ana Moreira, et Albert Royer. Using aspects to develop built-in tests for components. Dans *AOSD Modeling with UML Workshop, 6th International Conference on the Unified Modeling Language (UML), San Francisco, USA*, 2003. (Cited on page 15.)

[19] Jean-Michel Bruel, Geri Georg, Heinrich Hussmann, Ileana Ober, Christoph Pohl, Jon Whittle, et Steffen Zschaler. Models for non-functional aspects of component-based software (nfc'04). Dans *UML Modeling Languages and Applications*, pages 62–66. Springer, 2005. (Cited on page 23.)

[20] Julien Bruneau. *Développement et Test d'Applications d'Informatique Ubiquitaire : Une Méthodologie Outillée*. PhD thesis, Université de Bordeaux, 2012. (Cited on pages 46 and 72.)

[21] Julien Bruneau, Wilfried Jouve, et Charles Consel. Diasim : A parameterized simulator for pervasive computing applications. Dans *Mobile and Ubiquitous Systems : Networking & Services, MobiQuitous, 2009. MobiQuitous' 09. 6th Annual International*, pages 1–10. IEEE, 2009. (Cited on pages 46 and 72.)

[22] Julien Bruneau, Quentin Enard, Stéphanie Gatti, Emilie Balland, Charles Consel, et al. Design-driven development of safety-critical applications : A case study in avionics. 2011. (Cited on page 3.)

[23] Sven Burmester, Matthias Tichy, et Holger Giese. Modeling reconfigurable mechatronic systems with mechatronic uml. *Proceedings of Model Driven Architecture : Foundations and Applications (MDAFA 2004), Linköping, Sweden*, pages 155–169, 2004. (Cited on page 22.)

[24] Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, et Matthias Tichy. The fujaba real-time tool suite : model-driven development of safety-critical, real-time systems. Dans *Proceedings of the 27th international conference on Software engineering*, pages 670–671. ACM, 2005. (Cited on page 22.)

[25] Francois Carcenac et Frederic Boniol. A formal framework for verifying distributed embedded systems based on abstraction methods. *International Journal on Software Tools for Technology Transfer*, 8(6) :471–484, 2006. (Cited on page 14.)

[26] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, et Raffaela Mirandola. Qos-driven runtime adaptation of service oriented architectures. Dans *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 131–140. ACM, 2009. (Cited on page 19.)

[27] Damien Cassou, Benjamin Bertran, Nicolas Loriant, et Charles Consel. A generative programming approach to developing pervasive computing systems. Dans *ACM Sigplan Notices*, pages 137–146. ACM, 2009. (Cited on pages 4, 27, 31, and 70.)

[28] Damien Cassou, Emilie Balland, Charles Consel, et Julia Lawall. Leveraging software architectures to guide and

verify the development of sense/compute/control applications. Dans *Proceedings of the 33rd International Conference on Software Engineering*, pages 431–440. ACM, 2011. (Cited on pages 3, 27, 31, 45, 65, 70, 76, 78, and 81.)

[29] Alain Colmerauer. Specifications of Prolog IV, 1996. (Cited on page 40.)

[30] Eric M Dashofy, André Van der Hoek, et Richard N Taylor. A highly-extensible, xml-based architecture description language. Dans *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103–112. IEEE, 2001. (Cited on page 12.)

[31] Olivier Defour, Jean-Marc Jézéquel, et Noël Plouzeau. Extra-functional contract support in components. Dans *Component-Based Software Engineering*, pages 217–232. Springer, 2004. (Cited on pages 2, 13, and 38.)

[32] I Demeure, L Leboucher, N Rivierre, et F Singhoff. Support of temporal qos constraints for distributed object oriented multimedia applications, 2002. (Cited on page 16.)

[33] Isabelle Demeure et Elie Najm. *Les intergiciels - développements récents dans CORBA, Java RMI et les agents mobiles*. Hermès Science Publications - Lavoisier, 2002. (Cited on page 16.)

[34] Ewen Denney et Bernd Fischer. Certifiable program generation. Dans *Generative Programming and Component Engineering*, pages 17–28. Springer, 2005. (Cited on page 23.)

[35] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, et Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976. (Cited on page 14.)

[36] Bernard Dion. Correct-by-construction methods for the development of safety-critical applications. *SAE transactions*, 113(7) :242–249, 2004. (Cited on page 22.)

[37] Pierre Dissaux et Frank Singhoff. Stood and cheddar : Aadl as a pivot language for analysing performances of real time architectures. Dans *Proceedings of the European Real Time System conference. Toulouse, France*, 2008. (Cited on page 21.)

[38] DO-178. DO-178B / ED-12B, Software Considerations In Airborne Systems and Equipment Certification (RTCA, Inc. - EUROCAE), 1992. (Cited on pages 1, 2, 9, 10, 30, and 43.)

[39] DO-297. DO-297 / ED-124, Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations (RTCA, Inc. - EUROCAE), 2005. (Cited on page 16.)

[40] David Doose et Zoubir Mammeri. Polyhedra-based approach for incremental validation of real-time systems. Dans *Embedded and Ubiquitous Computing–EUC 2005*, pages 184–193. Springer, 2005. (Cited on page 13.)

[41] Frédéric Duclos, Jacky Estublier, et Philippe Morat. Describing and using non functional aspects in component based applications. Dans *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 65–75. ACM, 2002. (Cited on page 15.)

[42] Quentin Enard. *Développement d'applications logicielles sûres de fonctionnement : une approche dirigée par la conception*. PhD thesis, Université Sciences et Technologies-Bordeaux I, 2013. (Cited on pages 54, 56, and 77.)

[43] Quentin Enard, Stéphanie Gatti, Julien Bruneau, Young-Joo Moon, Emilie Balland, et Charles Consel. Design-driven development of dependable applications : A case study in avionics. Dans *PECCS-3rd International Conference on Pervasive and Embedded Computing and Communication Systems*, 2013. (Cited on pages vii, 3, 44, 63, 68, and 77.)

[44] Jean-Charles Fabre et Tanguy Pérennou. Friends : A flexible architecture for implementing fault tolerant and secure distributed applications. Dans *Dependable Computing—EDCC-2*, pages 1–20. Springer, 1996. (Cited on page 17.)

[45] Peter H Feiler, David P Gluch, John J Hudak, et Bruce A Lewis. Embedded system architecture analysis using sae aadl. Technical report, DTIC Document, 2004. (Cited on page 20.)

[46] Johan Fredriksson, Massimo Tivoli, et Ivica Crnkovic. A component-based development framework for supporting functional and non-functional analysis in control system design. Dans *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 368–371. ACM, 2005. (Cited on page 13.)

[47] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, et Klaus Lange. AUTOSAR–A Worldwide Standard is on the Road. Dans *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, 2009. (Cited on pages 2 and 9.)

[48] Matthieu Gallien et Félix Ingrand. Planification et exécution de plan pour la robotique autonome. (Cited on page 88.)

[49] David Garlan. Software architecture : a roadmap. Dans *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101. ACM, 2000. (Cited on page 10.)

[50] Stéphanie Gatti, Emilie Balland, et Charles Consel. A stepwise approach for integrating qos throughout software development. Dans *Fundamental Approaches to Software Engineering*, pages 217–231. Springer, 2011. (Cited on pages vii, 3, 27, 70, 77, 78, and 81.)

[51] Stéphanie Gatti, Franck Aimé, Stéphane Treuchot, et Jean Jourdan. Incremental functional certification for avionic functions reuse & evolution. Dans *DASC'12-Proceedings of the 31st AIAA/IEEE Digital Avionics System Conference*, 2012. (Cited on page vii.)

[52] Stéphanie Gatti, Franck Aimé, Stéphane Treuchot, et Jean Jourdan. Incremental functional certification. Dans *Proceedings of the Avionics Europe 2013 Conference for the Commercial & Defence Aerospace Industrie*, 2013. (Cited on page vii.)

[53] Vincent Gaudel, Frank Singhoff, Alain Plantec, Stéphane Rubini, Pierre Dissaux, et Jérôme Legrand. An ada design pattern recognition tool for aadl performance analysis. *ACM SIGAda Ada Letters*, 31(3) :61–68, 2011. (Cited on page 21.)

[54] Thomas Genßler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller, et Chris Stich. Components for embedded software : the pecos approach. Dans *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 19–26. ACM, 2002. (Cited on page 20.)

[55] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, et Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991. (Cited on page 17.)

[56] Nicolas Halbwachs. A synchronous language at work : the story of lustre. Dans *Formal Methods and Models for Co-Design, 2005. MEMOCODE'05. Proceedings. Third ACM and IEEE International Conference on*, pages 3–11. IEEE, 2005. (Cited on page 17.)

[57] Riadh Ben Halima, Khalil Drira, et Mohamed Jmaiel. A QoS-oriented reconfigurable middleware for self-healing web services. Dans *Proceedings of the 6th IEEE International Conference on Web Services*, pages 104–111. 2008. (Cited on page 2.)

[58] Timothy H Harrison, David L Levine, et Douglas C Schmidt. The design and performance of a real-time corba event service. *ACM SIGPLAN Notices*, 32(10) :184–200, 1997. (Cited on page 16.)

[59] DA Haverkamp et RJ Richards. Towards safety critical middleware for avionics applications. Dans *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on*, pages 716–722. IEEE, 2002. (Cited on page 16.)

[60] Jerome Hugues, Bechir Zalila, Laurent Pautet, et Fabrice Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4) :42, 2008. (Cited on page 20.)

[61] iso13236. International Standard – Information Technology - Quality of Service : Framework. ISO/IEC 13236 :1998 ITU-T X.641, 1998. (Cited on page 9.)

[62] Naoto Iwahashi. Robots that learn language : Developmental approach to human-machine conversations. Dans *Symbol Grounding and Beyond*, pages 143–167. Springer, 2006. (Cited on page 88.)

[63] Henner Jakob, Charles Consel, et Nicolas Loriant. Architecturing conflict handling of pervasive computing resources. Dans *Distributed Applications and Interoperable Systems*, pages 92–105. Springer, 2011. (Cited on page 72.)

[64] Sanjay Jha et Aruna Seneviratne. Synchronisation skew : a qos measurement study. Dans *Local Computer Networks, 1999. LCN'99. Conference on*, pages 77–78. IEEE, 1999. (Cited on page 35.)

[65] Li-jie Jin, Vijay Machiraju, et Akhil Sahai. Analysis on service level agreement of web services. *HP June*, 2002. (Cited on page 19.)

[66] Alexander Keller et Heiko Ludwig. The wsla framework : Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1) :57–81, 2003. (Cited on page 19.)

[67] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, et William G Griswold. An overview of aspectj. Dans *ECOOP 2001—Object-Oriented Programming*, pages 327–354. Springer, 2001. (Cited on page 15.)

[68] M-O Killijian et J-C Fabre. Adaptive fault tolerant systems : reflective design and validation. Dans *Parallel and Distri-*

buted Processing Symposium, 2003. Proceedings. International, pages 4–pp. IEEE, 2003. (Cited on page 17.)

[69] Hermann Kopetz et Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1) :112–126, 2003. (Cited on page 19.)

[70] Heiko Koziolek et Jens Happe. A QoS driven development process model for component-based software systems. Dans *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, pages 336–343. 2006. (Cited on pages 2 and 24.)

[71] Klaus Krogmann, Christian M. Schweda, Sabine Buckl, Michael Kuperberg, Anne Martens, et Florian Matthes. Improved Feedback for Architectural Performance Prediction Using Software Cartography Visualizations. Dans *Proceedings of the 5th International Conference on the Quality of Software Architectures*, pages 52–69. 2009. (Cited on page 2.)

[72] Klaus Krogmann, Christian M Schweda, Sabine Buckl, Michael Kuperberg, Anne Martens, et Florian Matthes. Improved feedback for architectural performance prediction using software cartography visualizations. Dans *Architectures for Adaptive Software Systems*, pages 52–69. Springer, 2009. (Cited on page 13.)

[73] Michaël Lafaye, David Faura, Marc Gatti, et Laurent Pautet. A new modeling approach for ima platform early validation. Dans *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 17–20. ACM, 2010. (Cited on page 21.)

[74] Michaël Lafaye, Marc Gatti, D Faura, et L Pautet. Model driven early exploration of ima execution platform. Dans *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pages 7A2–1. IEEE, 2011. (Cited on page 21.)

[75] Gilles Lasnier, Bechir Zalila, Laurent Pautet, et Jérome Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. Dans *Reliable Software Technologies–Ada-Europe 2009*, pages 237–250. Springer, 2009. (Cited on pages 2 and 53.)

[76] Bev Littlewood et Lorenzo Strigini. Software reliability and dependability : a roadmap. Dans *Proceedings of the Conference on the Future of Software Engineering*, pages 175–188. ACM, 2000. (Cited on page 52.)

[77] David C Luckham, John J Kenney, Larry M Augustin, James Vera, Doug Bryan, et Walter Mann. Specification and ana-

lysis of system architecture using rapide. *Software Engineering, IEEE Transactions on*, 21(4) :336–354, 1995. (Cited on page 11.)

[78] Jeff Magee, Naranker Dulay, Susan Eisenbach, et Jeff Kramer. Specifying distributed software architectures. Dans *Software Engineering—ESEC'95*, pages 137–153. Springer, 1995. (Cited on page 12.)

[79] Nenad Medvidovic et Richard N Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1) :70–93, 2000. (Cited on page 10.)

[80] Julien Mercadal, Quentin Enard, Charles Consel, et Nicolas Loriant. A domain-specific approach to architecturing error handling in pervasive computing. Dans *ACM Sigplan Notices*, pages 47–61. ACM, 2010. (Cited on pages 3, 44, 51, 54, 56, 65, 70, and 71.)

[81] Marjan Mernik, Jan Heering, et Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4) :316–344, 2005. (Cited on page 17.)

[82] Steven P Miller. Specifying the mode logic of a flight guidance system in core and scr. Dans *Proceedings of the second workshop on Formal methods in software practice*, pages 44–53. ACM, 1998. (Cited on page 63.)

[83] Pierre-Yves Oudeyer. Intelligent adaptive curiosity : a source of self-development. 2004. (Cited on page 88.)

[84] Mike P Papazoglou. Service-oriented computing : Concepts, characteristics and directions. Dans *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003. (Cited on page 19.)

[85] Carlos Parra, Daniel Romero, Sébastien Mosser, Romain Rouvoy, Laurence Duchien, et Lionel Seinturier. Using constraint-based optimization and variability to support continuous self-adaptation. Dans *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 486–491. ACM, 2012. (Cited on page 19.)

[86] Alexander R. Perry. The FlightGear flight simulator. 2004. (Cited on page 72.)

[87] Alan LaMont Pope. *The CORBA reference guide : understanding the common object request broker architecture*. Addison-

Wesley Longman Publishing Co., Inc., 1998. (Cited on page 15.)

[88] Thomas Robert, J-C Fabre, et Matthieu Roy. On-line monitoring of real time applications for early error detection. Dans *Dependable Computing, 2008. PRDC'08. 14th IEEE Pacific Rim International Symposium on*, pages 24–31. IEEE, 2008. (Cited on page 20.)

[89] Bikash Sabata, Saurav Chatterjee, Michael Davis, Jaroslaw J Sydir, et Thomas F Lawrence. Taxonomy for qos specifications. Dans *Object-Oriented Real-Time Dependable Systems, 1997. Proceedings., Third International Workshop on*, pages 100–107. IEEE, 1997. (Cited on page 9.)

[90] DG Schmidt et Fred Kuhns. An overview of the real-time corba specification. *Computer*, 33(6) :56–63, 2000. (Cited on page 16.)

[91] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, et Jean-Bernard Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software : Practice and Experience*, 42(5) :559–583, 2012. (Cited on page 87.)

[92] Mary Shaw. Beyond objects : A software design paradigm based on process control. *ACM SIGSOFT Software Engineering Notes*, 20(1) :27–38, 1995. (Cited on page 3.)

[93] Mary Shaw et Paul Clements. The golden age of software architecture. *Software, IEEE*, 23(2) :31–39, 2006. (Cited on page 10.)

[94] Frank Singhoff, Jérôme Legrand, Laurent Nana, et Lionel Marcé. Cheddar : a flexible real time scheduling framework. Dans *ACM SIGAda Ada Letters*, pages 1–8. ACM, 2004. (Cited on page 21.)

[95] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1) : 91–99, 2001. (Cited on page 17.)

[96] Miruna Stoicescu, Jean-Charles Fabre, et Matthieu Roy. Architecting resilient computing systems : overall approach and open issues. Dans *Software Engineering for Resilient Systems*, pages 48–62. Springer, 2011. (Cited on page 87.)

[97] Clemens Szyperski. *Component software : beyond object-oriented programming*. Pearson Education, 2002. (Cited on page 11.)

[98] Gabriel Tamura. *QoS-CARE : Un Système Fiable pour la Préservation de Contrats de Qualité de Service à travers de la Reconfiguration Dynamique*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2012. (Cited on page 19.)

[99] Richard N Taylor, Nenad Medvidovic, et Eric M Dashofy. *Software architecture : foundations, theory, and practice*. Wiley Publishing, 2009. (Cited on pages 1, 3, and 27.)

[100] Steve Vestal. MetaH Programmer's Manual, 1996. (Cited on page 20.)

[101] Andreas Vogel, Brigitte Kerhervé, Gregor von Bochmann, et Jan Gecsei. Distributed multimedia and qos : A survey. *Multimedia, IEEE*, 2(2) :10–19, 1995. (Cited on page 9.)

[102] Markus Volter, Thomas Stahl, Jorn Bettin, Arno Haase, et Simon Helsen. Model-driven software development : Technology, engineering, management. *John Wiley and Sons Ltd*, 2006. (Cited on page 2.)

[103] Nanbor Wang, Douglas C Schmidt, et Carlos O'Ryan. Overview of the corba component model. Dans *Component-Based Software Engineering*, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., 2001. (Cited on page 11.)

[104] Tao Yu et Kwei-Jay Lin. Service selection algorithms for composing complex services with multiple qos constraints. Dans *Service-Oriented Computing-ICSOC 2005*, pages 130–143. Springer, 2005. (Cited on page 19.)

[105] Charles Zhang, Dapeng Gao, et Hans-Arno Jacobsen. Towards just-in-time middleware architectures. Dans *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 63–74. ACM, 2005. (Cited on page 16.)