Développement d'applications logicielles sûres de fonctionnement : une approche dirigée par la conception

Quentin Enard

UNIVERSITÉ DE BORDEAUX

Département de formation doctorale en informatique

École doctorale EDMI Bordeaux

N° d'ordre : 4781

Développement d'applications logicielles sûres de fonctionnement : une approche dirigée par la conception

THÈSE

soutenue le 6 mai 2013

pour l'obtention du

Doctorat de l'Université de Bordeaux 1

(spécialité informatique)

par

Quentin Enard

Jury

Président :	Mohamed Mosbah,	Professeur à l'Institut Polytechnique de Bordeaux
Rapporteurs :	Christophe Dony,	Professeur à l'Université de Montpellier II
	Marc-Olivier Killijian,	Chargé de Recherche au LAAS-CNRS, Toulouse
Examinateurs :	Charles Consel,	Professeur à l'Institut Polytechnique de Bordeaux
	Nicolas Loriant	Research Associate à Imperial College, London

ABSTRACT

DEVELOPMENT OF DEPENDABLE APPLICATIONS: A DESIGN-DRIVEN APPROACH

In many domains such as avionics, medecine or home automation, software applications play an increasingly important role that can even be critical for their environment. In order to trust these applications, their development is contrained by dependability requirements. Indeed, it is necessary to demonstrate that these high-level requirements are taken into account throughout the development cycle and concrete solutions are implemented to achieve compliance. Such constraints make the development of dependable applications particularly complex and difficult. Easing this process calls for the research of new development approaches that integrate dependability concepts and guide the developers during each step of the development of trustworthy applications.

This thesis proposes to leverage a design-driven approach to guide the development of dependable applications. This approach is materialized through a tool-suite called DiaSuite and offers dedicated support for each stage of the development. In particular, a design language is used to describe both functional and non-functional applications. This language is based on a dedicated paradigm and integrates dependability concepts such as error handling. From the description of an application, development support is generated to guide the implementation and verification stages. Indeed, the generation of a dedicated programming framework allows to guide the implementation while the generation of a formal model allows to guide the static verification and simulation support eases the testing. This approach is evaluated through case studies conducted in the domains of avionics and pervasive computing.

KEYWORDS: Software Development, Design Language, Code Generation, Dependability

RÉSUMÉ

Dans de nombreux domaines tels que l'avionique, la médecine ou la domotique, les applications logicielles jouent un rôle de plus en plus important, allant jusqu'à être critique pour leur environnement. Afin de pouvoir faire confiance à ces applications, leur développement est contraint par des exigences de sûreté de fonctionnement. En effet il est nécessaire de démontrer que ces exigences de haut-niveau sont prises en compte tout au long du cycle de développement et que des solutions concrètes sont mises en oeuvre pour parvenir à les respecter. De telles contraintes rendent le développement d'applications sûres de fonctionnement particulièrement complexe et difficile. Faciliter ce processus appelle à la recherche de nouvelles approches de développement qui intègrent des concepts de sûreté de fonctionnement et guident les développeurs lors de chacune des étapes nécessaires à la production d'une nouvelle application digne de confiance.

Cette thèse propose ainsi de s'appuyer sur une approche dirigée par la conception pour guider le développement des applications sûres de fonctionnement. Cette approche est concrétisée à travers une suite d'outils nommée DiaSuite et offre du support dédié à chaque étape du développement. En particulier, un langage de conception permet de décrire à la fois les aspects fonctionnels et non-fonctionnels des applications en se basant sur un paradigme dédié et en intégrant des concepts de sûreté de fonctionnement tels que le traitement des erreurs. A partir de la description d'une application, du support est généré pour guider les phases d'implémentation et de vérification. En effet, la génération d'un *framework* de programmation dédié permet de guider l'implémentation tandis que la génération d'un modèle formel permet de guider la vérification statique de l'application et qu'un support de simulation permet de faciliter les tests. Cette approche est évaluée grâce à des cas d'études réalisés dans les domaines de l'avionique et de l'informatique ubiquitaire.

MOTS CLÉS : Développement Logiciel, Langage de Conception, Génération de Code, Sûreté de Fonctionnement

LISTE DES PUBLICATIONS

Les travaux discutés dans cette thèse ont été présentés précédemment.

CONFÉRENCES INTERNATIONALES

- « 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, Julien Bruneau, Young-Joo Moon, Emilie Balland et Charles Consel
- « A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing », dans *OOPSLA'10 : Proceedings of the 25th International Conference on Object Oriented Programming Systems Languages and Applications*, 2010, Julien Marcadal, Quentin Enard, Charles Consel et Nicolas Loriant

WORKSHOPS INTERNATIONAUX

- « An Experimental Study of A Design-driven, Tool-based Development Approach », dans *USER'12 : User Evaluation for Software Engineering Researchers*, 2012, Quentin Enard, Christine Loubery, Charles Consel et Xavier Blanc

POSTERS

- « Towards a Tool-based Development Methodology for Sense/Compute/Control Applications », dans *SPLASH'10 : Proceedings of the 1st International Conference on Systems, Programming, Languages and Applications : Software for Humanity*, 2010, Damien Cassou, Julien Bruneau, Julien Mercadal, Quentin Enard, Emilie Balland, Nicolas, Loriant et Charles Consel

REMERCIEMENTS

Cette thèse n'aurait jamais pu arriver à son terme sans l'aide et le soutien de nombreuses personnes.

Je tiens tout d'abord à remercier mon directeur de thèse, Charles Consel, pour m'avoir soutenu, guidé et conseillé au cours de cette thèse ainsi que pour m'avoir fait découvrir le monde de la recherche. J'ai beaucoup appris à ses côtés.

Je remercie également Nicolas Loriant pour son aide, sa motivation, son humour et ses conseils. Travailler à ses côtés a été un réel plaisir, de mes premiers pas de doctorant jusqu'à la soutenance.

Je tiens également à remercier Marc-Olivier Kilijian et Chistophe Dony de m'avoir fait l'honneur d'accepter d'être rapporteurs de ma thèse. Mes remerciements vont également à Mohamed Mosbah pour avoir accepté de présider mon jury de thèse.

Je remercie chaleureusement tous les membres de l'équipe Phoenix. Emile Balland pour ses nombreux conseils et pour son zèle à rendre notre quotidien plus agréable et instructif. Julien B. pour ses conseils de doctorant aguerri et pour être un collègue des plus agréable, que cela soit derrière un clavier, un piano, une raquette, ou une bière. Damien M. pour nos échanges plaisants (voire lunaires). Stéphanie pour avoir partagé cette aventure au sein de l'équipe, à Thales et en tant que jeune parent. Benjamin pour son humour et ses conseils. Julien M. pour sa gentillesse et notre aventure à Reno, du papier au casino. Damien C. pour ses conseils et ses sushis. Pengfei pour sa bonne humeur à toute épreuve. Zoé pour ses discussions pétillantes. Christine et Young-Joo pour leur collaboration et leur amabilité. Camille, Milan, Charles Jr, Paul et Luc, pour nos bons moments au sein de l'équipe et en dehors. Hélène pour ses discussions enrichissantes qui nous changent de l'informatique. Hong, Ghislain, Amélie, Stephen, Joao, Colline, Nicolas C. pour avoir participé au travaux et à la bonne ambiance de l'équipe. Sylvie et Chrystel pour leur aide et leur gentillesse au quotidien.

Je tiens également à remercier les membres de l'ARC SERUS pour les bons moments passés ensemble et plus particulièrement Jean-Charles Fabre, Laurence Duchien et Miruna pour leurs conseils et leur gentillesse.

Enfin, je remercie ma mère, mon frère et ma soeur pour le soutien qu'ils m'ont apporté tout au long de la thèse. J'ai également une pensée pour mon père qui m'a transmis sa passion pour l'informatique. Mes derniers remerciements vont à mes deux princesses qui font de mon quotidien une aventure merveilleuse et qui me donnent la motivation pour avancer.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES LISTINGS

1

INTRODUCTION

Dans notre environnement quotidien, les applications logicielles occupent une place de plus en plus importante. Dans des domaines tels que les transports, l'énergie, la santé, la communication ou la domotique, cette présence accrue ne peut se faire au détriment de la sûreté de fonctionnement. En effet, dans de tels domaines, la défaillance d'un système logiciel peut avoir de sévères conséquences sanitaires, environnementales ou économiques. En aéronautique, par exemple, la défaillance d'un système de navigation peut avoir des conséquences catastrophiques, engendrant la perte de vies humaines. Dans les communications, le crash d'un réseau téléphonique peut avoir de sévères répercussions économiques, alors qu'en domotique, la défaillance d'un système de détection d'incendie peut avoir de graves conséquences sur la santé des occupants d'un bâtiment et sur l'environnement.

Tandis que de nombreuses solutions ont fait leurs preuves pour garantir la sûreté de fonctionnement des systèmes matériels, le développement de logiciels sûrs de fonctionnement fait face à deux difficultés majeures : la complexité croissante des logiciels et les contraintes imposées par les exigences de sûreté. Par exemple, en avionique, les systèmes logiciels critiques doivent être certifiés afin d'apporter la garantie de leur sûreté. Le processus de certification impose que chaque exigence exprimée lors de la spécification du logiciel puisse être tracée jusqu'à sa réalisation dans le code exécuté. Une fois la traçabilité établie, il est alors plus aisé d'apporter une garantie de la sûreté de fonctionnement du logiciel. Cependant, établir cette traçabilité est une tâche d'autant plus contraignante qu'elle est généralement réalisée manuellement et s'avère particulièrement difficile lorsque le logiciel est complexe.

Ainsi, le développement de logiciels sûrs de fonctionnement représente un vrai défi, alors même que ces logiciels ont de plus en plus de responsabilités. De nouvelles approches et de nouveaux outils sont donc nécessaires pour faciliter le développement logiciel et permettre une meilleure adoption des logiciels sûrs dans notre environnement.

1.1 THÈSE

Pour faire face à la complexité grandissante des logiciels sûrs de fonctionnement, la thèse présentée dans ce document propose de s'appuyer sur une approche de développement dirigée par la conception. En effet, les approches de développement dirigées par la conception permettent de maîtriser la complexité d'un logiciel en le décrivant à un haut niveau d'abstraction. Cette description permet ainsi de raisonner sur le comportement du logiciel dès les premières étapes du développement et de guider le reste du processus.

Cependant, la plupart de ces approches sont génériques et n'offrent donc qu'un support limité pour la sûreté de fonctionnement. Par exemple, les exigences de sûreté sont souvent exprimées sous la forme de propriétés génériques dans la description du logiciel. Ces propriétés génériques peuvent être mal interprétées lors du développement, entraînant l'introduction de fautes pouvant mener à des défaillances imprévues du logiciel. Afin de guider au mieux le développement, notre approche repose sur un paradigme de conception pour les logiciels sûrs de fonctionnement. En effet, l'utilisation d'un paradigme permet un développement plus rigoureux ainsi que l'a démontré Shaw [84]. L'approche proposée permet notamment de guider de façon rigoureuse la mise en place de politiques de traitement des erreurs visant à limiter les défaillances d'un logiciel.

Une des contraintes majeures du développement de logiciels sûrs de fonctionnement est de s'assurer de la conformité entre la spécification du logiciel et le code exécuté. En effet cette conformité est nécessaire pour s'assurer que le logiciel se comporte selon les exigences formulées lors de la spécification et limiter les risques de défaillances non prévues. Les exigences de traçabilité pour la certification de logiciels en avionique sont un exemple de telles contraintes imposées lors du développement. Pour faire face à ces contraintes, une suite d'outils offrant du support de développement accompagne l'approche proposée dans cette thèse. En s'appuyant sur un paradigme de conception, ces outils permettent de fournir du support dédié à chaque étape du développement à partir de la description d'un logiciel. Ce support permet de guider de façon systématique le développement et de s'assurer par construction de la conformité du logiciel avec sa description.

L'approche proposée dans cette thèse est validée par le développement d'applications logicielles variées. Ainsi, nous illustrons et validons cette approche par des développements dans deux domaines : aéronautiques et informatique ubiquitaire. Afin de mesurer plus en détail les bénéfices de notre approche pour le

développement de logiciels sûrs de fonctionnement, la réalisation d'une étude empirique est également abordée.

1.2 ORGANISATION DU DOCUMENT

Ce document est organisé en trois parties. Nous présentons tout d'abord le contexte scientifique dans lequel notre thèse s'inscrit. Nous décrivons et illustrons ensuite notre approche de développement au travers d'un exemple de système logiciel de navigation pour appareils aéronautiques. Enfin nous validons notre approche en présentant sa mise en oeuvre dans une suite d'outils, ses avantages pour le développement d'applications logicielles variées et la réalisation d'une étude empirique.

CONTEXTE La première partie porte sur l'étude du contexte scientifique de cette thèse. Le Chapitre 2 introduit les différentes notions de sûreté de fonctionnement utilisées à travers le document et le Chapitre 3 dresse un état de l'art des approches d'ingénieries logicielles abordant la problématique du développement de logiciels sûrs de fonctionnement.

APPROCHE PROPOSÉE La seconde partie présente l'approche proposée pour le développement de logiciels sûrs de fonctionnement. Une vue globale de l'approche est d'abord présentée dans le Chapitre 4. Puis les Chapitres 5, 6 et 7 décrivent notre proposition pour les phases de conception, d'implémentation et de vérification du développement d'un logiciel sûr de fonctionnement.

VALIDATION La validation de l'approche proposée est abordée dans la troisième partie de ce manuscrit. Le Chapitre 8 présente une évaluation qui repose principalement sur des études de cas réalisées dans les domaines de l'aéronautique et de l'informatique ubiquitaire. Les travaux connexes à ceux présentés dans cette thèse sont dicutés dans le Chapitre 9. Enfin, le Chapitre 10 conclut cette thèse et aborde les directions futures de nos travaux.

Première partie

CONTEXTE

2

SÛRETÉ DE FONCTIONNEMENT : VOCABULAIRE ET CONCEPTS

Ce chapitre a pour objectif d'introduire les concepts de base de la sûreté de fonctionnement. Ces concepts ainsi que la terminologie employée pour les décrire sont empruntés aux travaux de Avizienis *et al.* [10] et de Laprie [65]. Après avoir défini les principales notions de la sûreté de fonctionnement, nous présentons les différentes méthodes utilisées pour parvenir à un système sûr de fonctionnement.

2.1 DÉFINITIONS

La sûreté de fonctionnement d'un système est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre. Elle peut être vue selon des propriétés différentes mais complémentaires qui permettent de définir les attributs suivants :

- La disponibilité : le système répond lorsqu'il est sollicité.
- La fiabilité : le système ne défaille pas en présence de fautes.
- La sécurité-innocuité : le système n'engendre pas de catastrophes ou d'événements critiques.
- La confidentialité : les informations du système sont accessibles uniquement à ceux qui en ont l'autorisation.
- L'intégrité : le système ne peut être altéré par un élément extérieur.
- La maintenabilité : le système peut être réparé.

La **défaillance** d'un système survient lorsque le service délivré dévie de ce à quoi le service est destiné. Un système peut défaillir par valeur lorsque la valeur du service délivré ne correspond pas au résultat attendu, où bien être l'objet de défaillances temporelles lorsque les conditions temporelle de délivrance du service ne correspondent pas à ce qui est prévu.

Une défaillance est la conséquence d'une ou plusieurs **erreurs**. Une erreur est une déviation d'un état correct à une état incorrect d'une partie du système. Lorsque cette déviation atteint l'état externe du système, c'est-à-dire lorsque le service fournit par le système dévie de ce à quoi il est destiné, elle entraîne une défaillance. Les erreurs peuvent être détectées ; dans le cas contraire, elles sont considérées comme latentes.

La cause supposée ou avérée d'une erreur est une **faute**. Une faute peut être de différentes natures à la fois : selon qu'elle soit volontaire ou accidentelle, interne ou externe au système, qu'elle se produise lors du développement du système ou lors de son exécution, *etc.* Une classification des différent types de fautes est proposée par Avizienis *et al.* [10].

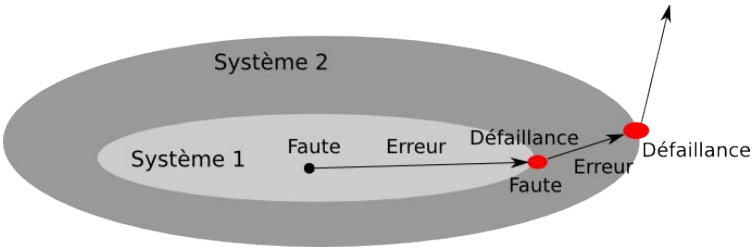

FIGURE 1: Fautes, Erreurs, Défaillances

Les notions de faute, erreur et défaillance sont récursives. La Figure 1 illustre ce phénomène : une défaillance du Système 1 peut être considérée comme une faute pour le Système 2 qui le contient.

2.2 MÉTHODES

Afin d'obtenir un système sûr de fonctionnement, il existe quatre catégories complémentaires de méthodes :

- La prévention des fautes regroupe les méthodes dont le but est d'empêcher l'occurrence ou l'introduction de fautes.
- La tolérance aux fautes vise à éviter la défaillance d'un système malgré la présence de fautes.
- L'élimination des fautes permet de réduire le nombre et la sévérité des fautes.
- La prévision des fautes cherche à estimer le nombre de fautes et leurs conséquences.

La prévention des fautes fait généralement partie du processus de développement d'un système logiciel. Par exemple, le choix d'un langage fortement typé ou de règles strictes de programmation permettent de limiter le nombre de fautes durant le développement. Dans le domaine avionique, la DO-178B [33] peut être assimilée à un standard définissant un certains nombre de recommandations pour le développement de systèmes logiciels sûrs de fonctionnement.

La tolérance aux fautes repose sur le traitement des fautes et des erreurs. Tandis que le traitement des fautes vise à éviter qu'une ou plusieurs fautes soient activées à nouveau, le traitement des erreurs vise à éliminer les erreurs, si possible avant la défaillance du système. Traiter une erreur nécessite la mise en

place de mécanismes de détection et de recouvrement d'erreur afin d'identifier un état erroné du système et de le remplacer par un état exempt d'erreur. Généralement, les mécanismes de recouvrement d'erreur sont basés sur une forme de redondance. Par exemple, l'état erroné peut être compensé par une nouvelle exécution de la partie concernée du système pour tolérer une faute intermittente, où bien cette partie peu être dupliquée et déployée sur un autre élément matériel pour éviter une faute matérielle.

L'élimination des fautes repose essentiellement sur la vérification du système. Cette vérification peut être statique ou dynamique. La vérification statique consiste en la vérification du système sans avoir à l'exécuter. Elle peut être effectuée à différentes étapes du développement : en phase de conception, un modèle du système peut être vérifié, grâce à des techniques de *model-checking*, tandis qu'en phase d'implémentation, le code peut être analysé. La vérification dynamique consiste essentiellement à effectuer des tests.

La prévision des fautes consiste en une évaluation qualitative et quantitative du comportement du système vis à vis des fautes. L'évaluation qualitative vise à identifier et classer les modes de défaillances ainsi que les événements pouvant mener à une défaillance du système. L'évaluation quantitative permet d'indiquer, en terme de probabilité, dans quelle mesure les objectifs de sûreté sont atteints. Des exemples de telles méthodes sont les FMEA (*Failure Mode and Effect Analysis*) et la construction d'arbre de fautes (*fault-trees*). Ces méthodes sont parfois utilisées dans les premières étapes de la phase de conception afin de définir les exigences de sûreté du logiciel [48].

Bien que ces méthodes aient fait leurs preuves pour les systèmes matériels, leur mise en oeuvre pour le développement de systèmes logiciels est tout de même nécessaire [78]. La prévention et la prévision des fautes sont parfois utilisées pour le développement de logiciels critiques. Cependant, elles requièrent une connaissance précise du système logiciel, faisant de leur application pour le développement d'une plus grande variété de logiciels un défi [67]. Les approches visant à faciliter l'intégration de la tolérance aux fautes dans les logiciels font également l'objet de nombreuses recherches [44]. En fait, les approches de développement actuelles se concentrent essentiellement sur l'élimination des fautes par les tests. Cependant, cette méthode implique de corriger le logiciel pour chaque faute découverte, ce qui s'avère également contraignant et particulièrement couteux. Ainsi, faciliter l'application de ces méthodes constitue un défi qu'il est important de relever pour promouvoir le développement de logiciels sûrs de fonctionnement.

3

DÉVELOPPEMENT DE LOGICIELS SÛRS DE FONCTIONNEMENT

L'ingénierie logicielle propose de nombreuses solutions pour faire face à la taille et à la complexité croissante des logiciels. Dans ce chapitre, nous dressons un état de l'art des différentes solutions visant en particulier à faciliter le développement de logiciels sûrs de fonctionnement. Les approches visant à simplifier l'implémentation sont présentées en première partie. Puis les approches dirigées par la conception, qui visent à guider l'intégralité du processus de développement à partir d'une description du logiciel, sont discutées dans la seconde partie.

3.1 IMPLÉMENTATION

L'implémentation d'un logiciel sûr de fonctionnement est une tâche complexe et sujette aux erreurs. Dans des domaines tels que l'aéronautique, les méthodologies de développement imposent des règles de codage drastiques aux développeurs afin de réduire le nombre de fautes potentiellement introduites lors de l'implémentation [33]. Cependant, l'objectif d'un système à zéro défauts peut être difficile, voire impossible, à atteindre, notamment lorsque l'implémentation de certaines parties du logiciel n'est pas maîtrisée. C'est notamment le cas lors de l'utilisation de composants logiciels sur étagère. L'utilisation de tels composants s'avère pourtant de plus en plus fréquente.

Puisque les fautes ne peuvent être toujours évitées, certaines approches proposent d'intégrer des mécanismes de tolérance aux fautes lors de l'implémentation tandis que d'autres facilitent la vérifications de propriétés de sûreté de fonctionnement. Cette section présente ces différentes approches.

3.1.1 *Bibliothèques de programmation*

Certaines bibliothèques de programmation permettent de faciliter l'implémentation de mécanismes de tolérance aux fautes. Par exemple, les bibliothèques EFTOS [29] et SwiFT [56] permettent de faciliter l'application de techniques telles que les *Recovery Blocks* [78, 79] et le *N-Version Programming* [9]. Ces techniques

consistent à introduire de la redondance dans le code d'un programme, permettant ainsi de tolérer une faute présente dans un *bloc* de code ou une *version* du programme en exécutant un *block* de recouvrement ou une autre *version* du programme. Cependant, l'utilisation de telles bibliothèques requiert du développeur qu'il ait un niveau d'expertise en sûreté de fonctionnement suffisant pour sélectionner et intégrer les différents mécanismes qu'elles proposent. De plus, l'entremêlement du code fonctionnel et de celui de l'intégration des mécanismes rend le programme plus complexe et difficile à comprendre, ce qui complique sérieusement la maintenance du code et son évolution [44].

3.1.2 *Exceptions*

Une exception fait référence à une situation où une opération en cours ne peut pas se finir correctement [49]. De nombreux langages de programmation, principalement objets, proposent des constructions dédiées à la gestion des exceptions [45]. Par exemple, le langage de programmation Java propose la construction **try/catch/finally** pour la gestion des exceptions. Un bloc **try** permet de tenter une ou plusieurs opérations dont les exceptions seront traitées dans un bloc **catch**. Les exceptions sont signalées par des objets de type *Exception* et peuvent être levées (ou "lancées") par l'instruction **raise** [1]. Lorsqu'une exception est levée par une opération, les instructions qui lui succèdent dans le bloc **try** ne sont pas exécutées. C'est le bloc **catch** approprié au type de l'exception qui est alors appelé. Finalement, le bloc **finally** est systématiquement appelé, qu'une exception ait été levée ou non. Ce bloc est particulièrement utile pour remettre le système dans un état cohérent, par exemple en fermant proprement un fichier, que sa lecture ou son écriture aient fonctionné ou non.

En terme de sûreté de fonctionnement, la notion d'erreur englobe la notion d'exception : une exception peut être ainsi considérée comme une erreur détectée. Les langages de programmation généralistes offrent donc la possibilité au programmeur d'utiliser la gestion des exceptions pour traiter des erreurs et éviter une défaillance du logiciel, appliquant ainsi les principes de la tolérance aux fautes. Cependant, la gestion des exceptions reste une tâche fastidieuse : à chaque invocation d'une méthode pouvant lever une exception, le programmeur doit traiter de potentielles exceptions ou les laisser se propager. Il en résulte un programme dont la logique applicative est polluée par le code de gestion des

1. Le bloc **catch** et l'instruction **raise** de Java peuvent êtres utilisés avec des objets implémentant l'interface *Throwable*. Les types *Error* et *Exception* implémentent cette interface mais seules les exceptions sont sensées pouvoir être traitées par le programmeur.

exceptions, ce dernier étant souvent redondant et enchevêtré dans le code de l'application, comme l'ont démontré Lippert et Lopez [66]. En raison de ces inconvénients, la gestion des exceptions est souvent négligée par les programmeurs.

Pour faciliter l'implémentation de la gestion des exceptions et éviter l'enchevêtrement du code de gestion des exceptions dans celui de la logique applicative, il est possible de se reposer sur des patrons de conception dédiés aux exceptions. Par exemple, *The Portland Pattern Repository* [38] est un site de dépôt de ces patrons qui met à disposition un ensemble de patrons applicables pour la plupart des langages de programmation généralistes. Le dépôt est organisé en plusieurs catégories : définir des types d'exceptions, lever des exceptions, traiter des exceptions, *etc.* Les descriptions des patrons de conception sont ouvertes à la discussion et peuvent être librement étendues. Des exemples de patrons consultables sur le dépôt sont les patrons *BottomPropagation* et *ExceptionReporter*. Le premier consiste à utiliser des exceptions dénuées d'information, à la façon des *maybe monads* du langage de programmation Haskell [2], afin de traiter uniquement l'échec de l'opération ayant levée une exception. Le second propose de découpler la gestion des exceptions du flot de contrôle et permet à une exception d'être traitée simultanément à plusieurs endroits. Ce patron adapte en fait le modèle *publish-subscribe* [37] pour la signalisation d'erreur et permet de structurer la gestion des exceptions.

3.1.3 *Méta-objets*

Un méta-objet est la réification d'un objet, c'est-à-dire la représentation concrète des notions abstraites d'objet. Un méta-objet peut, par exemple, manipuler la classe, les méthodes ou l'interface de l'objet qu'il réifie [70]. Les approches basées sur les protocoles méta-objet ont pour principal objectif de séparer les aspects fonctionnels d'une application, c'est-à-dire sa logique applicative, de ceux non-fonctionnels tels que la tolérance aux fautes. En effet, en implémentant les aspects non-fonctionnels via des méta-objets, i. e., au niveau méta, ceux-ci deviennent complètement transparents pour le développeur implémentant les aspects fonctionnels via des objets, i. e., au niveau de base.

Dans le contexte de la tolérance aux fautes, Fabre *et al.* proposent un *framework* pour le développement d'applications distribuées nommé FRIENDS (*Flexible and Reusable Implementation Environment for your Next Dependable System*) [39]. FRIENDS considère les applications distribuées comme un ensemble d'objets

2. http ://www.haskell.org/all_about_monads/html/maybemonad.html

communicants par l'intermédiaire de *proxys*, ces derniers étant une représentation locale d'un objet distant. En utilisant des méta-objets réifiant notamment les appels de méthodes sur les *proxys*, FRIENDS permet d'intercepter les communications entre les objets du niveau de base pour appliquer de façon transparente, au niveau méta, des mécanismes de tolérance aux fautes fournis par le *framework*. FRIENDS utilise le protocole méta-objet Open C++, un préprocesseur du langage de programmation objet C++ mais est également à l'origine de travaux permettant d'utiliser une approche réflexive pour implémenter des système tolérants aux fautes en CORBA [63, 64].

3.1.4 *Programmation orientée aspect*

La programmation orientée aspect (ou AOP pour *Aspect-Oriented Programming*) est une technique visant à modulariser les préoccupations transversales à un programme telles que la gestion des exceptions [62]. Pour ce faire, le programmeur décrit les aspects non-fonctionnels d'un programme en définissant des points de coupure où interviennent ces aspects. Les aspects sont ensuite "tissés" dans le code de l'application : un compilateur insert le code des aspects aux points de coupure. Il est à noter que des approches, telles que *AspectS*, proposent d'utiliser les protocoles à méta-objets offerts par certains langages de programmation, en l'occurrence Smalltalk, afin de tisser dynamiquement les aspects [55].

Lippert et Lopez ont étudié la réingénierie du code de gestion d'exception en utilisant le langage orienté aspect AspectJ [66]. Leur étude démontre que l'AOP réduit l'enchevêtrement lié à la gestion d'exceptions et les portions de code correspondantes. Cependant, l'AOP est généralement spécifique à un programme, rendant difficile la réutilisation de tissages d'aspects pour plusieurs applications. Cacho *et al.* proposent une approche de programmation orientée aspect, nommée EJFlow, qui étend AspectJ et permet aux développeurs de modulariser le code dédié à la gestion des exceptions au niveau du système [20]. Pour ce faire, ils présentent des mécanismes qui associent des gestionnaires d'exceptions avec le flot de propagation des exceptions de Java.

3.1.5 *Intergiciels*

Les intergiciels ont pour objectif de faciliter le développement de systèmes distribués. Ils permettent notamment de masquer la complexité du réseau et offrent divers mécanismes génériques (e. g., annuaire de services, notification d'événements) pour sup-

porter les interactions entre les différents objets communicants déployés. CORBA [51], Java RMI [50] ou les services web [27] sont des exemples d'intergiciels répandus. Ils fournissent un moyen (e. g., un IDL pour CORBA ou une interface Java pour Java RMI) de définir non seulement une représentation abstraite des services offerts par les objets communicants, mais également une représentation intermédiaire des données échangées. Ces définitions sont ensuite utilisées pour générer du support générique de communication (e.g., des stubs) et de découverte de services. Elles permettent ainsi d'abstraire une partie de l'hétérogénéité des systèmes distribués en assurant l'interopérabilité des différents objets communicants.

Dans sa thèse, Salatgé propose une plateforme nommée IWSD (*Infrastructure for Web Services Dependability*) pour la sûreté de fonctionnement des services web [81]. Un langage dédié à la description de services web, nommé DeWeL (*Dependable Web service Language*), permet de configurer de façon sûre des propriétés à vérifier sur les opérations de services web et d'actionner des mécanismes de recouvrement ou de signalement d'erreur. Tandis que les propriétés à vérifier sont exprimées sous la forme de pré-conditions et de post-conditions, les mécanismes de recouvrement font référence à des techniques usuelles de la tolérance aux fautes. Par exemple, DeWeL permet d'utiliser les techniques de réplication active ou passive en définissant des point d'accès alternatif à un service web donné. La défaillance du service web et les erreurs de communications sont alors compensées par la plateforme en utilisant les point d'accès alternatifs simultanément, dans le cas de la réplication active, ou l'un après l'autre, dans celui de la réplication passive. A partir d'une description DeWeL, un compilateur génère un connecteur dédié à la mise en oeuvre des vérifications et des mécanismes déclarés dans la plateforme IWSD [81].

Edstrom et Tilevich proposent une approche similaire pour appliquer la tolérance aux fautes aux applications REST [3] [34]. Leur approche vise également à faciliter la réutilisation et l'extension de stratégies de tolérance aux fautes pour des services REST. Ces stratégies sont exprimées via un langage nommé FTDL (*Fault Tolerance Description Language*). Un compilateur permet ensuite de générer un *framework* de programmation facilitant l'implémentation d'une application utilisant ces services. FTDL étant basé sur XML, le langage se veut facilement extensible et facilite la portabilité de l'approche.

3. REST (*Representational State Transfer* [42]) est un standard alternatif à SOAP (*Simple Object Access Protocol* [95]) qui est le standard usuellement utilisé pour le développement de services web

Dans le domaine de l'informatique ubiquitaire, plusieurs approches offrent du support pour améliorer la fiabilité des logiciels [25, 77, 80]. Par exemple, le projet one.world [80] propose des mécanismes de tolérance aux fautes tels que des points de contrôle (*checkpoints*) qui permettent aux développeurs de sauvegarder l'état d'un composant et de le restaurer ultérieurement afin de reprendre l'exécution dans un état cohérent suite à une défaillance (e.g., une coupure de courant). *One.world* permet également d'améliorer la robustesse des systèmes d'informatique ubiquitaire en fournissant de la persistance des données et des transactions.

3.1.6 *Langages dédiés*

Un langage dédié (ou *DSL* pour *Domain-Specific Language*) est un langage de programmation conçu dans le but d'adresser un ensemble de problèmes lié à un domaine particulier. En fournissant des notations appropriées et des abstractions spécifiques à un domaine, il permet d'exprimer des solutions de programmation sur mesure et son utilisation s'avère souvent plus avantageuse que celle d'un langage généraliste, notamment en terme de productivité, de vérification et de facilité de programmation [14]. Il existe des langages dédiés pour une grande variété de domaines, allant des bases de données (e. g., SQL), en passant par la mise en forme de documents (e. g., HTML, LaTeX) jusqu'aux pilotes de périphériques (e. g., Devil [73]).

Dans le domaine des systèmes distribués, des paradigmes de communication variés sont utilisés, qu'ils soient basés sur les données (e. g., *tuple spaces* [47]) ou sur le contrôle (e. g., le modèle des acteurs [3]). Ces paradigmes permettent de découpler les communications entre les objets communicants de leur dimension spatiale et temporelle. Cette stratégie améliore la résilience d'un système, permettant notamment d'isoler les erreurs en prévenant nombre d'entre elles d'être propagées. Cependant, ces approches se concentrent uniquement sur les erreurs de communication.

Dedecker *et al.* proposent un langage nommé *AmbientTalk*, dédié au développement d'applications dans un réseau mobile [30]. En particulier, ils introduisent un mécanisme de gestion des exceptions dans un environnement distribué pour faire face aux spécificités du matériel mobile [75]. Ce mécanisme consiste en un ensemble de constructions du langages qui permettent de gérer les exceptions à différents niveaux de granularité dans le code d'une application : message, bloque et collaborations.

Demsky et Dash proposent un langage de tâches, appelé *Bristlecone*, pour le développement de systèmes robustes [31]. Leur

approche privilégie l'exécution continue d'une application en tolérant notamment certaines dégradations d'un comportement spécifié. L'environnement d'exécution de *Bristlecone* utilise la spécification des tâches pour déterminer celle à exécuter. Dans *Bristlecone*, les tâches ont une sémantique transactionnelle. Lorsqu'une tâche échoue suite à une erreur provenant d'une faute matérielle ou logicielle, la transaction d'une tâche est annulée. Afin d'éviter de déclencher à nouveau la faute, l'environnement d'exécution mémorise la combinaison de tâches ayant entraîné la défaillance et exécute d'autres tâches.

Dans un autre domaine, Halbwachs *et al.* proposent un langage dédié aux systèmes réactifs temps-réel nommé *Lustre* [53]. *Lustre* adopte le paradigme de la programmation synchrone. Les langages synchrones fournissent des constructions permettant aux développeurs de considérer leurs programmes comme réagissant instantanément aux événements extérieurs. Chaque événement interne au programme se déroule donc a un moment précis par rapport à l'historique des événement extérieurs. De ce fait, en fournissant uniquement des constructions déterministes, un langage synchrone permet de développer des programmes à la fois déterministes d'un point de vue fonctionnel et temporel. Cet aspect déterministe est particulièrement intéressant dans le cadre du développement de logiciels sûrs de fonctionnement car il permet de faciliter la vérification de propriétés de sûreté de fonctionnement. *Lustre* est présenté à la fois comme un langage d'implémentation et de conception, permettant de décrire le comportement de systèmes logiciels et matériels sous la forme d'automates et de les vérifier. L'utilisation des langages synchrones requiert cependant de détailler précisément le comportement d'un logiciel afin de pouvoir vérifier ses propriétés de sûreté de fonctionnement. La description précise et complète d'un système s'avère pourtant particulièrement complexe et fastidieuse pour des logiciels dont la taille ne cesse de croître et qui reposent de plus en plus sur des composants sur étagère et dont le comportement n'est pas toujours précisément connu.

Ainsi, des approches ont été proposées dans des domaines variés afin de faciliter le développement de logiciels sûrs de fonctionnement. Tandis que les bibliothèques de programmation et les intergiciels fournissent des mécanismes de tolérance aux fautes aux développeurs, les approches basées sur les protocoles à méta-objets ou la programmation orientée aspect permettent de faciliter la séparation des aspects fonctionnels et non-fonctionnels dans le code de programmes et réduisent ainsi la complexité inhérente à l'enchevêtrement du code de la logique applicative et de la tolérance aux fautes. Enfin, les langages dédiés et, dans une moindre mesure, les langages génériques fournissent des constructions appropriées pour faciliter l'implémentation de logi-

ciels sûrs de fonctionnement. Cependant, bien qu'elle soit facilitée par l'utilisation de telles approches, l'implémentation de logiciels sûrs reste une tâche complexe. En effet, le fait de traiter une erreur ou de la laisser se propager, ou bien d'utiliser un mécanisme de tolérance aux fautes particulier plutôt qu'un autre, sont des décisions importantes et sujettes aux erreurs des développeurs. Ces décisions requièrent une vision globale du logiciel qu'il est difficile d'appréhender uniquement à partir du code.

3.2 CONCEPTION

Les approches dirigées par la conception ont pour objectif de guider le développement d'un logiciel à partir de sa description à un haut niveau d'abstraction. Cette notion regroupe les approches basées sur les architectures logicielles et l'ingénierie des modèles. Qu'elles décrivent l'*architecture* d'un logiciel ou un *modèle*, les descriptions à haut niveau d'abstraction permettent de faciliter la compréhension de systèmes complexes en masquant certains détails technologiques. Il est alors plus aisé de raisonner sur le comportement et les propriétés du logiciel. Ces descriptions sont également un vecteur de communication entre les développeurs et permettent donc de guider le processus de développement. Ainsi, dans le cadre de la sûreté de fonctionnement, ce type d'approche permet de limiter certaines fautes de développement liées à une mauvaise compréhension du système. Cette section dresse un état de l'art de ces différentes approches, que cela soit dans le domaine des architectures logicielles ou de l'ingénierie des modèles.

3.2.1 *Architectures logicielles*

L'architecture d'une application ou d'un système logiciel définit ce dernier en termes de *composants* et d'*interactions* entre ces composants [12, 92]. Les composants effectuent chacun une partie du calcul global d'un logiciel et interagissent pour former son comportement. Les composants et les interactions sont donc les briques de base à partir desquelles une architecture est construite. Cette séparation entre les calculs et les interactions permet d'augmenter l'indépendance des composants, facilitant ainsi leur implémentation et promouvant leur réutilisation. En offrant une vision globale et haut niveau de la structure et du comportement d'un système, l'architecture logicielle permet de guider le développement d'un logiciel. En effet, elle facilite la prise de décisions visant à satisfaire les exigences du système. De plus, l'architecture d'un logiciel peut être analysée afin de vérifier

un large spectre de propriétés sur la structure et le comportement d'un logiciel [2, 58, 69]. La conception d'une architecture peut être facilitée par l'utilisation d'un style architectural [1] et l'utilisation d'un langage de description d'architecture (*Architecture Description Languages* ou ADLs). Cependant, s'assurer de la conformité entre architecture et implémentation reste un défi majeur de ce domaine d'ingénierie logicielle [26].

3.2.1.1 *Style architectural*

Un style architectural regroupe un ensemble de règles, de motifs (*patterns*) et d'idiomes d'un domaine particulier afin d'en faciliter la réutilisation et de permettre d'effectuer des vérifications spécifiques au domaine. Les styles *pipes and filter*, client-serveur ou d'architecture en couches sont des exemples de styles répandus [85]. Pour faciliter la conception d'une architecture d'un logiciel tolérant aux fautes, Guerra *et al.* proposent un motif architectural (*architectural pattern*) de composant tolérant aux fautes idéal [28] pour le style architectural C2 [91]. Un motif architectural est une description rigoureuse de certains éléments d'une architecture et peut être vu comme une version restreinte d'un style architectural. Le style C2 est initialement dédié au domaine des interfaces utilisateurs mais peut être étendu à d'autres domaines pour lesquels une architecture en couche est appropriée. Le motif architectural proposé par Guerra *et al.* est illustré par la Figure 2.

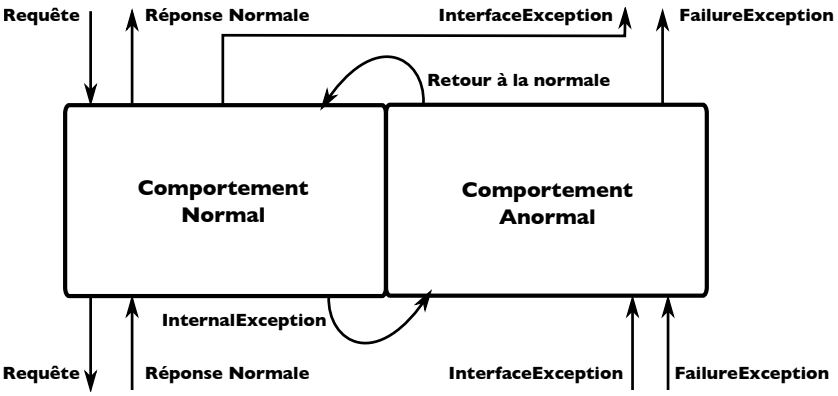

FIGURE 2: Un composant tolérant aux fautes idéal

Lorsqu'une erreur est détectée lors de l'exécution du comportement normal d'un composant, ou bien lorsque ce composant reçoit une exception d'un autre composant, le composant adopte alors un comportement anormal. Ce dernier a pour but de traiter l'erreur signalée par une exception *InternalException* dans le cas d'une erreur détectée pendant le comportement normal, une exception *FailureException* signalant la défaillance d'un autre composant, ou une exception *InterfaceException* signalant une réponse

anormale à une requête. Si le traitement réussit, le composant peut alors retourner à son comportement normal. Si il échoue, le composant propage alors une exception *FailureException*. Le principal avantage d'une telle représentation est d'expliciter la distinction entre les aspects fonctionnels (le comportement normal) et les aspects de tolérance aux fautes (le comportement anormal).

Ainsi, en offrant un support de conception additionnel aux éléments architecturaux classiques, un style architectural permet de guider la conception d'une architecture et d'effectuer des analyses spécifiques à un domaine donné. Afin de renforcer ce support de conception dans la pratique, des langages, textuels ou graphiques, sont généralement utilisés pour la description d'architecture.

3.2.1.2 *Langages de description d'architecture*

Les ADLs fournissent des constructions dédiées à la description formelle de composants et d'interactions. Ces dernières sont généralement représentées sous la forme de *connecteurs* [92]. Une combinaison de composants et de connecteurs forment alors une *configuration* décrivant le logiciel. Il existe de nombreux ADLs, couvrant des domaines d'application variés et se focalisant sur certains aspects spécifiques de l'architecture logicielle. Par exemple, *Darwin* [71] se concentre sur la conception de systèmes distribués. *Rapide* [69] s'intéresse plus particulièrement à la spécification et l'analyse par la simulation de l'aspect dynamique des architectures logicielles, tandis que *Wright* [7] permet la spécification et l'analyse du comportement des éléments architecturaux. Bien qu'UML (*Unified Modeling Language*) puisse parfois être considéré comme un ADL, notamment dans sa version 2.0, il ne dispose pas de certains concepts importants tels que les connecteurs. Les approches basées sur UML seront plus particulièrement abordées dans la Section 3.2.2.

Peu d'ADLs se concentrent sur la spécification de concepts de sûreté de fonctionnement tels que le flot de propagation des erreurs ou les mécanismes de tolérance aux fautes. Filho *et al.* proposent un *framework* conceptuel, nommé *Aereal*, pour la description et l'analyse du flot des exceptions entre des composants architecturaux [43]. Dans le domaine des systèmes embarqués, AADL (*Architecture Analysis & Design Language*) est un standard fournissant des constructions dédiées à la spécification de systèmes logiciel (e. g., composants, ports) et leur déploiement sur des plateformes d'exécution (e. g., *thread*, processus, mémoire) [41]. En utilisant AADL, les concepteurs d'un système spécifient les aspects non-fonctionnels en ajoutant des propriétés

aux constructions du langage (e. g., la période d'un *thread*). La spécification du flot des erreurs est réalisée en utilisant une extension standardisée d'AADL, *the Error Model Annex* [94]. Cependant, ces approches se basent sur des concepts généralistes et offrent peu de support pour guider les concepteurs et s'assurer de la conformité entre l'architecture et le code d'une application.

AADL étant un standard, de nombreux travaux de recherche sont consacrés à des outils d'analyse et de développement basés sur ce langage. En particulier, l'environnement *Ocarina* offre du support pour la vérification de modèles AADL et la génération de code [57]. Le support de vérification consiste à générer des réseaux de Petri à partir de spécifications AADL. Tandis que de nombreux ADLs ne fournissent peu ou pas de support pour l'implémentation, *Ocarina* permet la génération d'un support de programmation dédié à une description AADL. Ce support de programmation consiste en du code "glu" pour faciliter l'utilisation d'un intergiciel temps-réel nommé *PolyORB-HI*, une version à l'intégrité renforcée de *PolyORB* [98]. Par exemple, un *thread* AADL résulte dans la génération d'une tâche en *ADA*, le langage de programmation utilisé. Pour s'assurer de la traçabilité entre le code généré et la spécification, tout comportement dynamique (e. g., allocation dynamique de la mémoire) y est prohibé. De plus, des vérifications sont faites pour s'assurer que le code utilisateur respecte ces même contraintes. Bien que ce type d'approche offre des outils efficaces pour faciliter à la fois la vérification d'une architecture et du code qui en résulte, le support fourni se concentre sur le déploiement de composants logiciels sur une plateforme d'exécution et ne permet pas de guider la conception et l'implémentation de la logique applicative d'un système logiciel et des politiques de tolérance aux fautes pouvant être appliquées.

3.2.1.3 *Conformité*

Tandis que les architectures logicielles sont particulièrement utiles pour structurer une application et effectuer des vérifications indépendamment du code, s'assurer de la conformité entre le code et l'architecture reste un problème majeur [26]. En effet, cette conformité est nécessaire pour assurer la traçabilité des décisions architecturales dans le code de l'application et garantir la préservation des propriétés vérifiées à partir de l'architecture. Parmi les approches qui se concentrent sur cette problématique, *ArchJava* [4], *ComponentJ* [82] et *ACOEL* [88] proposent d'ajouter des constructions architecturales au langage de programmation Java afin de coupler fortement l'architecture et le code d'une application. *Archface* [93] utilise la programmation orientée aspect pour synchroniser la description architecturale et l'implémentation mais requiert de la part des développeurs d'anticiper la

structure de l'application. Tandis que ces approches ont en commun le fait de mélanger l'architecture et le code, Zheng et Taylor proposent une approche basée sur la génération de code à partir de l'architecture et nommée *1.x-way mapping* [99]. Cette approche a la particularité de séparer le code dédié à l'implémentation de la logique applicative du code chargé d'assurer la conformité avec l'architecture. Cette approche permet notamment de faciliter les évolutions aussi bien au niveau de l'architecture que du code.

Que cela soit en unifiant le code et l'architecture ou par la génération de code dédié, ces approches permettent d'obtenir un code *correct par construction* et assurent ainsi la conformité entre code et architecture [99]. Par exemple, le système de type d'*ArchJava* permet d'assurer l'*intégrité des communications* [69] entre l'architecture et l'implémentation [5]. Cette propriété signifie que les composants peuvent uniquement communiquer, au niveau de l'implémentation, avec les composants auxquels ils sont directement connectés dans l'architecture. Cette propriété est fondamentale et permet la vérification de propriétés plus variées telles que celles portant sur la façon dont communiquent les composant entre eux. Ainsi, les approches proposées semblent particulièrement adaptées pour le développement de logiciels sûrs de fonctionnement. Cependant, ces approches sont généralistes et aucune d'entre elles n'intègre des concepts de sûreté de fonctionnement ou n'offre de support pour guider l'implémentation de la logique applicative.

3.2.2 *Ingénierie des modèles*

L'ingénierie dirigée par les modèle est une approche de génie logiciel visant à générer tout ou partie d'une application à partir de modèles. Un modèle est une représentation simplifiée d'un système construite dans un but précis [15]. La conception d'une application consiste à construire un ou plusieurs modèles représentant l'application à un haut niveau d'abstraction puis construire des modèles de plus en plus proches du code. Ces modèles sont alors utilisés pour vérifier une grande variété de propriétés puis pour générer le code de l'application. L'*Object Management Group* (OMG) propose une vision de l'ingénierie dirigée par les modèles, nommé *Model-Driven Architecture* (MDA [52]), qui repose sur des standards tels que UML, OCL, QVT et MOF. Le principal objectif de ce type d'approche est de masquer la complexité du développement d'applications en proposant des technologies qui combinent des langages de conception dédiés, des outils de transformation de modèles et des générateurs de code [89].

Des approches d'ingénierie des modèles variées ont été proposées pour prendre en compte les aspects non-fonctionnels des logiciels. Par exemple, *SecureUML* [68] se concentre sur le contrôle d'accès dans les systèmes distribués tandis que *Context-UML* [86] permet la prise en compte d'information contextuelles pour les applications à base de services web. Parmi les approches pour le développement d'applications sûres de fonctionnement, Burmester *et al.* proposent une approche de développement dédiée aux systèmes mécatroniques [4] [18]. Cette approche est basée sur une extension d'UML dédiée aux systèmes temps-réel. Pour permettre la vérification formelle de l'intégralité d'un système mécatronique, les auteurs proposent de développer une bibliothèque de patrons de coordination qui définissent les rôles des composants, leurs interactions et les contraintes temps-réel. Les composants de l'application sont ensuite construits à l'aide de cette bibliothèque en spécifiant leurs rôles et certains détails supplémentaires sur leurs comportements. L'approche comprend du support outillé pour la spécification, la vérification et la synthèse de code source sous la forme d'un *plug-in* de la suite d'outils *Fujaba* [19]. Bien que cette approche se concentre sur les aspects temps-réels d'une application et ne fournit donc que peu de support pour la tolérance aux fautes, l'utilisation de patrons de coordination permet de faciliter la conception et la vérification de systèmes mécatroniques.

Dion *et al.* proposent une autre approche de développement pour les applications sûres nommée SCADE (*Safety Critical Application Development Environment* [32]). Cette approche est basée sur l'utilisation de langages dédiés synchrones tels que *Lustre* (voire la Section 3.1.6) et d'automates hiérarchiques pour la spécification d'applications sûres. L'utilisation d'un paradigme de développement dédié, celui des langages synchrones, ainsi que des méthodes formelles, permettent d'effectuer de nombreuses vérifications dès la phase conception d'un système. En particulier, le paradigme synchrone assure par construction le déterminisme d'une spécification et facilite ainsi les vérifications de sûreté. De plus, le paradigme synchrone permet de s'abstraire du temps physique et de vérifier ainsi des propriétés temps-réel au niveau du code. La suite d'outils SCADE préserve le déterminisme depuis les spécifications jusqu'à l'implémentation grâce à la génération de code. En effet, ces outils sont certifiés afin d'assurer la conformité du code généré et des spécifications. Ils sont donc particulièrement appropriés pour faciliter le développement de logiciels critiques à partir de spécifications détaillées. Cependant, le niveau de détail des spécifications requis par une telle approche peut parfois nuire à la simplicité recherchée en phase de

4. Un système mécatronique est un système complexe combinant à la fois de la mécanique, de l'électronique et de l'informatique temps-réel.

conception et n'est pas toujours accessible, notamment lors de l'utilisation de composants sur étagères.

3.3 BILAN

De nombreuses approches visent à faciliter le développement de logiciels sûrs de fonctionnement. En particulier, des approches basées sur la programmation orientée aspect, les protocoles à méta-objet ou les langages de programmation proposent de faire face à la complexité des logiciels tolérants aux fautes en permettant la séparation des aspects fonctionnels et non-fonctionnels en phase d'implémentation. Ces approches permettent ainsi de simplifier en partie une des phases principales du développement de logiciels sûrs de fonctionnement. Cependant, la complexité des systèmes logiciels limite les bénéfices de ces approches en l'absence d'une vision globale du système. Il est donc nécessaire de prendre en compte la sûreté de fonctionnement à travers toutes les phases du développement logiciel, de la conception jusqu'au déploiement.

Les approches de développement dirigées par la conception telles que les architectures logicielles ou l'ingénierie des modèles proposent des solutions permettant de guider le développement de logiciels sûrs de fonctionnement à partir de spécifications faites lors de la phase de conception. Cependant la plupart des approches existantes sont génériques et n'offrent qu'un support de développement limité, en particulier pour les aspects de sûreté de fonctionnement tels que la tolérance aux fautes. En effet, malgré l'existence de styles architecturaux ou de langages de conception dédiés à des domaines variés, peu d'approches permettent de guider systématiquement le développement de logiciels sûrs de fonctionnement. Une autre problématique rencontrée par les approches dirigées par la conception est celle de la conformité entre les spécifications ou l'architecture et l'implémentation. Cette conformité, particulièrement importante pour limiter les erreurs pouvant être introduites lors de l'implémentation et s'assurer de la traçabilité des exigences de haut-niveau, requiert généralement un niveau de détail élevé des spécifications. Cette contrainte rend les approches préservant la conformité difficiles à appliquer en dehors des domaines particulièrement critiques.

Ainsi, faciliter le développement de logiciels sûrs de fonctionnement requiert de prendre en compte les concepts de sûreté de fonctionnement à chaque étape du cycle de développement. Cette prise en compte doit notamment se traduire par du support de développement dédié fourni à chaque étape du développement et l'assurance de la conformité entre chacune de ces étapes et les spécifications de haut niveau décrivant le logiciel.

Deuxième partie

APPROCHE PROPOSÉE

4

VUE D'ENSEMBLE

Ce chapitre présente une vue d'ensemble de l'approche proposée dans cette thèse. Avant d'en dresser une présentation générale, nous en listons les principales contributions. Nous présentons ensuite une application de gestion de vol. Cette application sert de fil conducteur pour illustrer l'application de notre approche à travers le document.

4.1 CONTRIBUTIONS

Les principales contributions présentées dans ce document sont les suivantes :

UN CADRE DE CONCEPTION Afin de permettre de guider au mieux le développement d'applications sûres, notre approche s'appuie sur un paradigme de développement dédié, le paradigme *Sense/Compute/Control* (SCC) [92]. En particulier, nous utilisons ce paradigme pour faire la distinction entre la logique applicative d'une application et la supervision d'aspects non-fonctionnels. Ce cadre de conception permet de raisonner sur des aspects non-fonctionnels tels que le traitement des erreurs dès la phase de conception. Il permet également de guider les développeurs en fournissant du support de développement dédié.

UN SUPPORT DE DÉVELOPPEMENT DÉDIÉ L'approche proposée s'appuie sur une méthodologie de développement dirigée par la conception, nommée DiaSuite, et sa suite d'outils [24]. Pour faciliter la conception de logiciels sûrs, nous avons étendu le langage de conception DiaSpec afin de permettre la spécification et la vérification d'aspects non-fonctionnels. En particulier, cette extension offre un support de conception dédié au traitement des erreurs [72]. À partir d'une spécification, le compilateur de DiaSpec génère un *framework* de programmation qui guide l'implémentation. Nous avons étendu ce compilateur afin de fournir du support de programmation additionnel dédié au traitement des erreurs. Enfin, du support de vérification est fourni grâce à la génération d'un modèle formel de la spécification permettant l'utilisation de techniques de *model-checking* [36].

PRÉSERVATION DE LA CONFORMITÉ En phase de conception, nous fournissons du support de vérification pour s'assurer la conformité d'une spécification vis-à-vis de la formalisation des exigences de haut niveau. Cette conformité est automatiquement préservée lors de l'implémentation grâce a l'approche générative de DiaSuite. De plus, puisqu'il n'est pas toujours possible de vérifier statiquement certaines propriétés lors de la conception, nous fournissons du support de test additionnel pour valider la conformité de l'implémentation.

VALIDATION DANS DES DOMAINES VARIÉS Nous avons validé notre approche dans des domaines variés. En particulier, nous avons développé des systèmes de gestion de vol pour l'aéronautique ainsi que des applications d'informatique ubiquitaires déployées dans une école d'ingénieurs. De plus nous avons étudié la mise en place d'une évaluation empirique de notre approche auprès d'utilisateurs.

4.2 PRÉSENTATION DE L'APPROCHE

L'approche proposée dans cette thèse s'appuie sur le paradigme SCC et une méthodologie de développement dédiée, DiaSuite.

4.2.1 *Le paradigme SCC*

Le paradigme SCC est issu du style architectural *Sense/Compute/-Control* promu par Taylor *et al.* [92]. Ce style est particulièrement approprié pour les applications qui interagissent avec un environnement externe. De telles applications sont caractéristiques de domaines tels que la domotique (et l'immotique), la robotique, l'automobile et l'avionique.

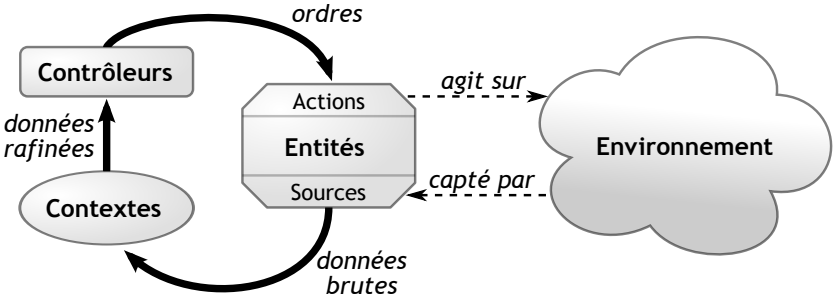

FIGURE 3: Le paradigme SCC

Le style architectural SCC comprend trois types de composants représentés dans la Figure 3 :

- Les *entités* correspondent aux périphériques matériels ou logiciels. Ils interagissent avec l'environnement externe en captant des données et en agissant sur ce dernier.
- Les *contextes* sont des composants qui raffinent (c'est-à-dire qui filtrent, agrègent et interprètent) les données de base fournies par les entités.
- Les *contrôleurs* sont des composants qui utilisent l'information raffinée par les contextes afin de contrôler l'environnement en déclenchant des actions à partir des entités.

Ainsi, le style architectural SCC permet de décrire *des boucles d'interaction* avec un environnement externe. Lorsque la notion d'environnement externe désigne une maison, un immeuble, une voiture ou un avion, ces boucles représentent les fonctions de l'application décrite. Par ailleurs, dans des domaines tels que l'aéronautique, on appelle *chaîne fonctionnelle* une succession de traitements pouvant être effectuée par l'application lors d'une itération de la boucle. Plus généralement, ces boucles décrivent les **aspects fonctionnels** d'une application.

Dans cette thèse, nous proposons d'utiliser ce même style architectural pour représenter des traitements dédiés à la sûreté de fonctionnement de l'application. En effet, le style architectural SCC permet de décrire des *boucles de supervision* de l'application. Ces boucles permettent de traiter certaines erreurs et d'implémenter les **aspects non-fonctionnels** de fiabilité et de performance impliqués par la sûreté de fonctionnement en faisant, par exemple, de la surveillance (*monitoring*) et de la reconfiguration. Les aspects non fonctionnels d'un système réfèrent en effet aux contraintes sur la manière dont le système implémente et délivre son service [92].

4.2.2 *DiaSuite*

DiaSuite[1] est une méthodologie outillée de développement dédiée aux applications SCC [24]. La Figure 4 illustre la façon dont DiaSuite s'appuie sur le paradigme pour fournir du support à chaque étape du processus de développement. En phase de conception, le langage DiaSpec [22, 23] fournit des constructions spécifiques au paradigme SCC (étape ① de la Figure 4). Ces constructions couvrent les aspects fonctionnels d'une application, tels que les flots de contrôle et de données, mais également ceux non-fonctionnels comme la *QoS* (*Quality of Service*) [46] et le traitement des erreurs [72]. De plus, le support de conception inclut du support pour la vérification formelle des spécifications DiaSpec [23, 36].

1. DiaSuite est développée au sein de l'équipe-projet Inria Phoenix.

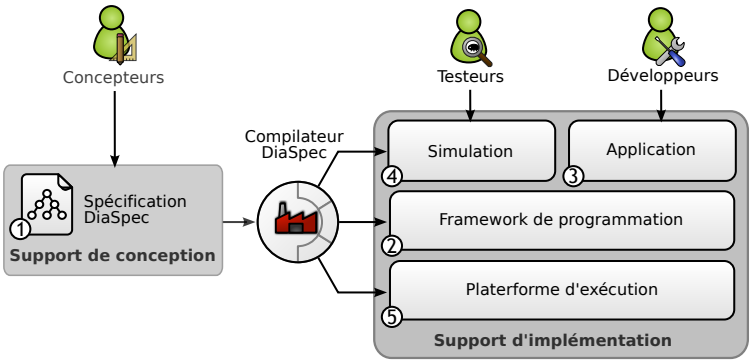

FIGURE 4: La méthodologie outillée DiaSuite

À partir d'une description DiaSpec, un compilateur génère un *framework* de programmation pour guider le programmeur (étapes ② et ③ de la Figure 4). Le compilateur génère également du support de test ciblant un simulateur spécifique à un domaine particulier (étape ④). Dans le domaine de l'aéronautique, le simulateur de vol *FlightGear* [76] est ciblé, tandis que dans celui de l'informatique ubiquitaire, le support généré cible le simulateur DiaSim [16, 17]

Enfin, DiaSuite offre du support pour déployer des applications en utilisant différentes technologies des systèmes distribués comme les Services Web, RMI ou SIP (étape ⑤ de la Figure 4).

4.3 GESTIONNAIRE DE VOL

Afin d'illustrer la méthodologie DiaSuite pour le développement d'applications SCC sûres de fonctionnement, nous l'appliquons au développement d'un gestionnaire de vol pour l'avionique. En raison de sa criticité, notamment vis-à-vis de la sécurité des passagers, cette application doit respecter de strictes exigences de haut-niveau.

Le gestionnaire de vol est en charge de la navigation de l'appareil et fonctionne sous la supervision du pilote [74]. Par exemple, le pilote peut directement spécifier les paramètres de vol comme l'altitude. Il peut également définir un plan de vol qui sera automatiquement suivi par l'appareil. Chaque paramètre de vol est géré par un mode de navigation spécifique (e. g., gestion de l'altitude, gestion du cap). Lorsqu'un mode est sélectionné par le pilote, l'application contrôle les ailerons et les élévateurs afin d'atteindre la valeur cible du paramètre correspondant. Par exemple, si le pilote spécifie un cap à suivre, l'application compare ce cap cible au cap courant qui est capté par des appareils tels que les centrales inertielles (ou IRU pour *Inertial Reference Unit*) puis manœuvre les ailerons en conséquence. Chaque mode

de navigation est généralement associé à une chaîne fonctionnelle représentant la succession des traitements, depuis les capteurs jusqu'aux actionneurs [97].

Dans le domaine aéronautique, des analyses de sûreté sont conduites afin d'identifier les situations dangereuses. Il en résulte la définition d'exigences de sûreté [8]. Voici des exemples de ces exigences de haut-niveau pour le gestionnaire de vol telles qu'elles sont définies par des experts du domaine :

ex1. Le temps d'exécution de la chaîne fonctionnelle associée au mode de gestion du cap ne doit pas excéder 650 ms.

ex2. La fraîcheur des données de navigation utilisée doit être inférieure à 200 ms.

ex3. Le dysfonctionnement ou la défaillance d'un capteur doit être systématiquement signalé au pilote dans les 300 ms.

ex4. Un mode de navigation doit être désactivé lors de la défaillance d'un capteur impliqué dans son fonctionnement.

Pour certifier l'application, afin de pouvoir la déployer, il est nécessaire de traduire ces exigences vers une conception cohérente et d'assurer leurs traçabilité à travers le processus de développement. Dès lors, la prise en compte de telles contraintes suggère fortement l'utilisation d'une méthodologie de développement dirigée par la conception telle que DiaSuite.

Les chapitres suivant illustrent l'application de la méthodologie DiaSuite pour développer une telle application de gestion de vol. L'accent sera mis sur le support fourni pour prendre en compte les aspects de sûreté de fonctionnement à travers le processus de développement. En particulier, nous montrerons comment, à partir d'une spécification DiaSpec, du support d'implémentation et de vérification est généré pour guider le développement et assurer la traçabilité des exigences de haut niveau.

5

CONCEPTION

Ce chapitre présente notre approche pour la conception d'applications SCC. En décrivant les différentes étapes de la conception d'un gestionnaire de vol à partir des exigences formulées dans la Section 4.3, ce chapitre permet d'illustrer le support offert aux concepteurs d'une application. Ce support permet notamment de guider la description fonctionnelle de l'application et de raffiner cette description pour concevoir des stratégies de traitement des erreurs et, plus généralement, des stratégies de supervision des aspects non-fonctionnels.

5.1 DESCRIPTION FONCTIONNELLE

En s'appuyant sur le paradigme SCC, le langage de conception DiaSpec offre du support spécifique pour la description d'une application SCC. Tandis qu'une **taxonomie** permet de caractériser les différentes entités qui interagissent avec l'environnement, les déclarations fournies par DiaSpec permettent de décrire le **flot des données** qui circulent au sein de l'application puis le **flot de contrôle** de cette dernière.

5.1.1 *Taxonomie*

Afin d'appliquer le paradigme SCC, le langage de conception DiaSpec offre des déclarations spécifiques aux entités, contextes et contrôleurs. Une entité est définie comme un ensemble de capacités qui permettent de capter des informations de l'environnement ou d'agir sur ce dernier. Le Listing 1 présente une taxonomie des entités utilisées par notre application de gestion de vol. Comme l'illustre le mot-clé **source** (lignes 2, 3 et 4), l'entité `IRU` capte entre autre la position, le cap (`heading`) et le roulis [1] (`roll`) de l'avion depuis l'environnement. L'entité `NavMMI` est une abstraction des interactions avec le pilote et fournit directement les informations qu'il saisit, comme par exemple un cap cible (`targetHeading`). Ainsi que l'indique le mot-clé **action** (ligne 20), l'entité `Aileron` fournit, quant à elle, l'interface `Control` qui permet d'agir sur l'environnement en modifiant le comportement de

1. Le roulis d'un avion est son mouvement d'oscillation autour de son axe longitudinal.

l'avion et notamment sont roulis. Le haut niveau d'abstraction de ces déclarations facilite l'intégration de composants sur étagère : tant qu'une implémentation respecte ces déclarations, elle peut être utilisée par l'application.

```
1  device IRU {
2    source position as Coordinates;
3    source heading as Float;
4    source roll as Float;
5    ...
6  }
7
8  device NavMMI {
9    source targetHeading as Float;
10   ...
11   action DisableMode;
12   action Display;
13 }
14
15 action Control{
16   incline(targetRoll as Float);
17 }
18
19 device Aileron {
20   action Control;
21 }
```

Listing 1: Extrait de la taxonomie des entités

5.1.2 *Flot de données*

À partir de cette taxonomie, des contextes et contrôleurs sont définis pour décrire le flot de données de l'application. Par exemple, pour concevoir le mode de gestion du cap, nous définissons un contexte `IntHeading` qui calcule un cap intermédiaire à atteindre en fonction du cap actuel de l'avion fourni par l'entité `IRU` et du cap cible défini par le pilote via l'entité `NavMMI`. À partir de ce cap intermédiaire et du roulis actuel de l'avion fourni par l'entité `IRU`, le contexte `TargetRoll` calcule un roulis cible. Ce dernier est alors utilisé par le contrôleur `AileronController` pour actionner les ailerons et se diriger vers le cap souhaité par le pilote.

La spécification de composants SCC est illustrée par le Listing 2. Cet extrait de spécification DiaSpec décrit le flot de données du mode de gestion du cap à travers les déclarations des contextes `IntHeading` et `TargetRoll` ainsi que du contrôleur `AileronController`. Le contexte `IntHeading` déclare produire des données de type `Float` à partir de données d'entré `heading` et `targetHeading`, fournies par les entité `IRU` et `NavMMI` respecti-

```
1  context IntHeading as Float {
2    source heading from IRU;
3    source targetHeading from NavMMI;
4  }
5
6  context TargetRoll as Float {
7    source roll from IRU;
8    context Intheading;
9  }
10
11 controller AileronController {
12   context TargetRoll;
13   action Control on Aileron;
14 }
```

Listing 2: Extrait de la spécification du mode de gestion du cap

vement, et déclarées en utilisant le mot-clé **source** (lignes 2 et 3). Le contexte `TargetRoll` produit le même type de données mais utilise quant à lui les données provenant du contexte `IntHeading`, ainsi que l'illustre le mot-clé **context** (ligne 8), et des données `roll` provenant de l'entité `IRU`. Enfin, le contrôleur `AileronController` utilise les données provenant du contexte `TargetRoll` et peut déclencher des actions de type `Control` sur l'entité `Aileron` ainsi que l'indique le mot-clé **action** (ligne 13).

5.1.3 *Flot de contrôle*

DiaSpec permet également de raffiner la spécification du flot de données pour spécifier le flot de contrôle à l'aide de contrats d'interaction [23]. Le Listing 3 illustre un tel contrat définissant le flot de contrôle du contexte `IntHeading`. Ce contrat est introduit par la clause **interaction** (ligne 4). Il déclare que le contexte `IntHeading` peut accéder au cap cible indiqué par le pilote (la donnée `targetHeading` fournie par l'entité `NavMMI`) lorsqu'il reçoit un cap (`heading`) depuis l'entité `IRU`. Tandis que les mots-clés **when provided** (ligne 5) indiquent la condition d'activation du contrat, en l'occurrence la réception du cap actuel de l'avion, la clause **get** (ligne 6) liste les autorisations d'accès du contexte lorsque le contrat est activé. Ces dernières se résument au cap cible dans le cas de ce contrat. Enfin, il est également spécifié, grâce aux mots-clés **always publish** (ligne 7), que le contexte publie systématiquement un résultat lorsque la condition d'activation est satisfaite. Par ailleurs, il est possible de spécifier une condition d'activation **when required** qui correspond à la réception d'une requête de la part d'un autre contexte, ainsi que la publication conditionnelle d'un résultat (**maybe publish**) ou l'interdiction de publication (**no publish**).

```
context IntHeading as Float {
  source heading from IRU;
  source targetHeading from NavMMI;
  interaction {
    when provided heading from IRU;
    get targetHeading from NavMMI;
    always publish;
  }
}
```

Listing 3: Spécification d'un contrat d'interaction

La Figure 5 illustre la spécification fonctionnelle du mode de gestion du cap à l'aide d'un graphe dirigé du flot des données. Tandis que les noeuds du graphe sont des composants SCC, les arêtes indiquent un échange de données entre ces derniers. Les différents types d'échange entre composants, spécifiés dans les contrats d'interaction, sont représentés par des flèches à l'orientation différente, donnant un aperçu du flot de contrôle de l'application. Ainsi, lorsqu'un nouveau cap est publié par l'entité `IRU`, le contexte `IntHeading` accède au cap cible fourni par le pilote puis publie le résultat de son traitement, un cap intermédiaire à atteindre. Le contexte `TargetRoll` reçoit ce cap intermédiaire et accède au roulis pour ensuite calculer le roulis cible à utiliser par le contrôleur `AileronController` pour diriger les ailerons de l'avion.

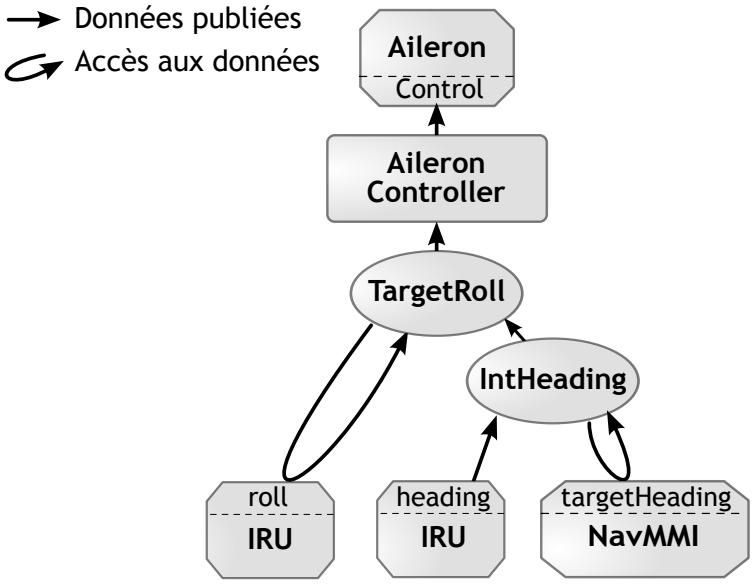

FIGURE 5: Aperçu de la spécification du mode de gestion du cap

5.2 TRAITEMENT DES ERREURS

Tandis que le traitement des erreurs est généralement abordé lors de la phase d'implémentation, ce qui a pour conséquence de rendre le code difficilement maintenable [66], nous proposons ici de spécifier comment traiter les erreurs dès la phase de conception. Nous avons étendu le langage de conception DiaSpec pour fournir du support dédié au traitement des erreurs. Ce support permet de caractériser une erreur et de spécifier comment la traiter, à la fois au niveau de l'application et au niveau du système [72]. Le traitement d'une erreur au niveau de l'application consiste uniquement à la compenser pour poursuivre l'exécution du programme, par exemple en passant dans un mode d'exécution dégradé. Le traitement d'une erreur au niveau du système se concentre, quant à lui, sur la préservation de la cohérence globale du système et permet, par exemple, de réparer ou d'éliminer un élément défectueux à l'origine de l'erreur. Avant de présenter le support offert pour concevoir le traitement des erreurs au niveau de l'application puis au niveau du système, nous présentons le support dédié à la caractérisation des erreurs.

5.2.1 *Caractérisation des erreurs*

Les applications SCC doivent essentiellement faire face aux erreurs provenant des capteurs et actionneurs utilisés pour interagir avec l'environnement. En effet, ces entités sont particulièrement exposées à l'environnement extérieur et donc aux fautes physiques et d'interaction. De plus, elles sont fréquemment considérées comme des composants sur étagère exposés aux fautes de développement. Quelle que soit leurs natures, ces fautes peuvent causer le dysfonctionnement d'une entité, engendrant une erreur au sein de l' application SCC.

Afin de pouvoir détecter puis traiter une erreur, il est nécessaire de l'identifier. Nous avons donc étendu DiaSpec afin de permettre de raffiner la taxonomie des entités (Section 5.1.1) avec des exceptions qui, levées par les entités, signalent une erreur lorsque celle-ci est détectée. Ainsi, les développeurs d'une entité savent quelles exceptions peuvent être levées et sont guidés dans leurs choix d'implémentation de mécanismes de détection d'erreur. De plus, les développeurs d'une application utilisant cette entité savent désormais quelles exceptions doivent être gérées. Par exemple, la taxonomie du gestionnaire de vol (Listing 1) a été raffinée afin de spécifier quelles sont les exceptions pouvant être levées par les entités comme la centrale inertielle (`IRU`). Le Listing 4 présente la spécification de cette dernière.

```
1 device IRU {
2   source position as Coordinates;
3   source heading as Float;
4   source roll as Float;
5   ...
6   raises LowAccuracyException;
7   raises FailureException;
8 }
```

Listing 4: Spécification des exceptions

L'entité `IRU` peut lever une exception `LowAccuracyException` (ligne 6) pour signaler une erreur concernant la précision des données fournies et une exception `FailureException` (ligne 7) pour signaler la défaillance du capteur. Ces déclarations vont alors guider l'implémentation des entités. En effet, la première déclaration incite les développeurs de l'entité à mettre en place un mécanisme de détection d'erreur par valeur, à l'aide d'assertions par exemple. La seconde déclaration, plus générale, peut inciter à les développeurs de l'entité à mettre en place un mécanisme de surveillance comme le *heartbeat* ou le *watchdog*.

Certaines exceptions peuvent être levées par la plateforme d'exécution. Par exemple, les technologies de communication réseau disposent de mécanismes de *timeout* qui lèvent une exception lorsque le dépassement d'un délai de communication est détecté. Pour permettre de gérer uniformément ces exceptions, nous avons intégré une hiérarchie d'exceptions *built-in* au langage DiaSpec. Un extrait de cette hiérarchie est présenté par la Figure 6.

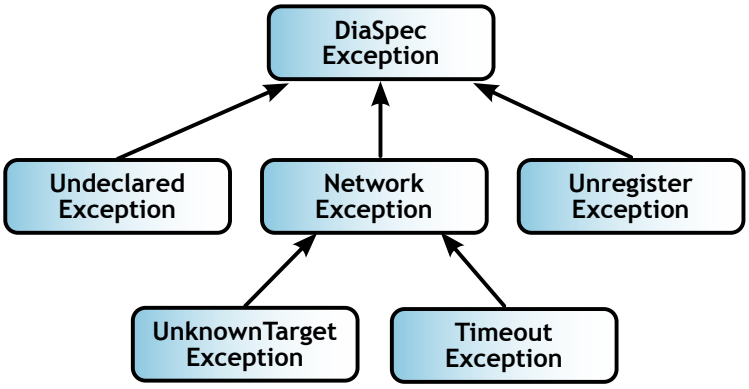

FIGURE 6: Aperçu de la hiérarchie des exceptions *built-in* de DiaSpec

Les exceptions `TimeoutException` et `UnknownTargetException` sont des exceptions du réseau de communication, elles héritent donc de l'exception `NetworkException`. Elles signalent respectivement le dépassement d'un délai et une tentative de joindre un objet communiquant inconnu du réseau. Avec `NetworkException`,

`UndeclaredException` et `UnregisterException` sont deux types d'exception héritant de `DiaSpecException`, le type d'exception au plus haut niveau de la hiérarchie. Les exceptions de type `UndeclaredException` servent à signaler tout événement inconnu pouvant entraîner la défaillance du système, c'est-à-dire toute erreur qui ne correspond à aucune exception déclarée au niveau de la spécification d'une entité et à aucune exception *built-in*. Quant aux exceptions de type `UnregisterException`, elles permettent de signaler la disparition non prévue d'une entité suite au non-renouvellement de son enregistrement auprès de la plateforme d'exécution.

Ainsi, les développeurs bénéficient d'une large palette d'exceptions pour signaler les erreurs devant être traitées par une application : en plus de pouvoir définir des exceptions spécifiques aux entités, ils peuvent se reposer sur une hiérarchie des exceptions couramment utilisées dans les applications SCC.

5.2.2 *Traitement au niveau de l'application*

Concevoir le traitement des erreurs au niveau de l'application consiste à spécifier quel est l'impact d'une erreur sur le flot de contrôle de l'application. En particulier, il s'agit de définir comment les composants SCC de l'application réagissent lorsqu'une exception, signalant une erreur dans le traitement d'une de leurs requêtes, leur est propagée. Ainsi, les concepteurs peuvent raffiner les contrats d'interaction des composants, présentés dans la Section 5.1.3, pour spécifier différentes contraintes sur le traitement d'une exception propagée suite à une requête d'accès aux données :

1. Un traitement *obligatoire* permet de systématiquement poursuivre l'exécution de l'application en compensant l'erreur.
2. Un traitement *facultatif* permet de laisser la décision de traiter l'erreur au développeur, à la façon des exceptions non vérifiées (*unchecked*) en Java.
3. Un traitement *évité* permet de systématiquement propager l'exception au composant appelant. En l'absence de ce dernier, l'erreur signalée n'est pas compensée et l'exécution de la chaîne fonctionnelle est stoppée.
4. L'*absence* de traitement permet d'indiquer qu'un traitement est inutile car aucune exception ne peut être propagée au composant. Cette contrainte peut également être utilisée pour s'assurer que toutes les exceptions qui pourraient être propagées jusqu'à ce composant sont déjà traitées par d'autres composants.

Un traitement *obligatoire* est spécifié à l'aide des mots clés **mandatory catch**, un traitement *facultatif* à l'aide des mots clés **optional catch**, un traitement *évité* à l'aide des mots clés **skipped catch** et, enfin, l'*absence* de traitement est spécifiée à l'aide des mots clés **no catch**.

Ces déclarations permettent d'exiger qu'une erreur soit traitée différemment en fonction du caractère critique de la chaîne fonctionnelle impactée et des capacités de compensation. Par exemple, nous avons étendu la description du contexte `IntHeading` du gestionnaire de vol, présentée dans le Listing 3, pour indiquer que les exceptions propagées lors de l'accès au cap cible fourni par le pilote doivent être systématiquement traitées. Un exemple de traitement pouvant compenser une erreur survenue lors de l'accès au cap cible est l'utilisation de la dernière valeur avant que l'erreur n'ai eu lieu en tant que valeur par défaut. Ainsi, un cap intermédiaire est toujours fourni au reste de l'application suite à la réception d'une nouvelle mesure du cap réel de l'avion. L'application peut alors continuer à fonctionner malgré une défaillance temporaire de l'interface avec le pilote. Le Listing 5 présente la nouvelle déclaration du contexte `IntHeading` : la clause **get** du contrat d'interaction autorise l'accès au cap cible (`targetHeading`) mais indique également un traitement *obligatoire* des exceptions à l'aide des mot clés **mandatory catch** (ligne 6).

```
1 context IntHeading as Float {
2   source heading from IRU;
3   source targetHeading from NavMMI;
4   interaction {
5     when provided heading from IRU;
6     get targetHeading from NavMMI [mandatory catch];
7     always publish;
8   }
9 }
```

Listing 5: Aperçu de la spécification du traitement des erreurs au niveau de l'application

Bien que la compensation du cap cible permet de tolérer la défaillance de l'interface avec le pilote de façon temporaire, certaines erreurs ne peuvent être compensées. Par exemple, le contexte `TargetRoll` utilise le roulis fourni par la centrale inertielle pour calculer un roulis cible qui sert à diriger les ailerons de l'avion (Figure 5). En l'absence du roulis fourni par la centrale, le gestionnaire de vol ne peut plus contrôler correctement le cap de l'avion. Il peut alors s'avérer imprudent de tenter de traiter une exception propagée lors de l'accès aux données de la source `roll` de l'entité `IRU` qui mesure le roulis de l'avion. Dans ce cas, le contrat d'interaction spécifiant l'accès à ces données doit être enrichi de la déclaration **skipped catch** afin de s'assurer qu'aucun

traitement d'exception n'est tenté. Ainsi, lorsqu'une exception est propagée jusqu'au contexte, celui-ci ne peut la traiter et ne publie donc aucun résultat. Le contrôleur des ailerons ne reçoit aucune valeur et les ailerons ne sont pas actionnés inutilement. En revanche, si l'on spécifie un nouveau contexte pouvant accéder directement au roulis cible calculé par le contexte `TargetRoll`, ce nouveau composant devra indiquer si le traitement des exceptions provenant du contexte `TargetRoll` est *obligatoire*, *facultatif* ou également *évité*.

5.2.3 *Traitement au niveau du système*

Concevoir le traitement des erreurs au niveau du système consiste à décrire des contextes et des contrôleurs dédiés à la gestion des événements signalant les erreurs, les *évenements exceptionnels*. Ces événements sont de même type que les exceptions *built-in* ou définies dans la taxonomie et peuvent fournir des informations sur les causes de l'erreur. Ainsi, lorsqu'une erreur est détectée, un événement exceptionnel est publié par une entité ou la plateforme d'exécution puis traité par un contexte. Ce dernier raffine les informations pouvant être tirées de l'exception afin de fournir une information de plus haut niveau à d'autres contextes ou directement à un contrôleur. Finalement, un contrôleur pourra prendre en compte les informations de haut niveau qui lui sont fournies pour mettre en oeuvre une stratégie de réparation du système en effectuant différentes actions comme par exemple le remplacement d'une entité défectueuse. Ces stratégies sont décomposées en contextes et contrôleurs et sont alors explicites dans la spécification DiaSpec du système. En permettant de les spécifier, il est alors plus aisé de raisonner sur ces stratégie de traitement des erreurs dès la phase de conception.

```
1  context SensorFailure as SensorID {
2    exception LowAccuracyException from IRU;
3    exception FailureException from IRU;
4    exception DiaSpecException from IRU;
5    interaction {
6      when caught LowAccuracyException from IRU,
7                  DiaSpecException from IRU;
8      maybe publish;
9    }
10   interaction {
11     when caught FailureException from IRU;
12     always publish;
13   }
14 }
```

Listing 6: Spécification d'un contexte dédié à la gestion d'exceptions

Le Listing 6 présente un exemple de contexte dédié à la gestion des événements exceptionnels qui permet de déterminer si un capteur est défectueux ou non. En effet, le contexte `SensorFailure` est dédié à la gestion des événements `LowAccuracyException`, `FailureException` et `DiaSpecException` ainsi que l'indiquent les lignes 2, 3 et 4. Tandis que les deux premiers événements correspondent à des exceptions levées par l'entité `IRU` (voir le Listing 4, Section 5.1.1), le dernier est un événement qui correspond à l'ensemble des exceptions *built-in* pouvant impliquer cette entité (voir la Figure 6, Section 5.2.1). Deux contrats d'interaction (lignes 5 à 9 et lignes 10 à 13) viennent raffiner la définition du contexte. Le premier spécifie que le contexte peut publier un résultat lors de la réception d'un événement `LowAccuracyException` ou `DiaSpecException` mais qu'il peut également ne pas propager l'information. En effet, ces événements signalent des erreurs qui sont généralement transitoires, et qui peuvent être ignorées dans ce cas, mais qui peuvent également révéler une défaillance plus sérieuse qui doit alors être traitée. Par exemple, les erreurs de mesure ou de réseau peuvent être tolérées jusqu'à un certain point avant qu'une centrale inertielle ne soit considérée défectueuse, pour des raisons de précision ou d'accessibilité. Le second contrat, quant à lui, indique que la réception de l'événement `FailureException` résulte systématiquement par la publication d'un résultat. En effet, cet événement indiquant la défaillance d'une centrale inertielle, le contexte se contente de transmettre l'identifiant de la centrale.

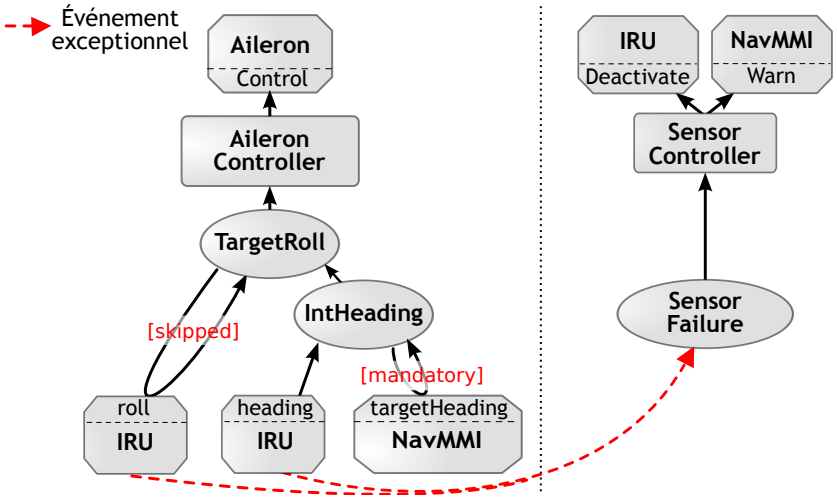

FIGURE 7: Aperçu de la spécification du traitement des erreurs du mode de gestion du cap

Comme l'indique la Figure 7, la spécification de composants SCC dédiés à la gestion des événements exceptionnels permet de concevoir des stratégies de traitement des erreurs au niveau du système qui sont indépendantes de la logique applicative de

l'application. En effet, dans cette figure, les composants SCC définissant la logique applicative de l'application sont représentés à gauche tandis que ceux définissant la stratégie de traitement des erreurs sont à droite. Ces derniers sont le contexte `SensoreFailure` présenté dans le paragraphe précédent et un contrôleur nommé `SensorController` qui a pour tâche de désactiver complètement toute entité `IRU` considérée défectueuse par le contexte et de le signaler au pilote. Ces deux composants permettent ainsi d'éviter de déclencher à nouveau une erreur qui a été détectée et de respecter en partie l'exigence **ex3.** présentée dans la Section 4.3.

5.3 SUPERVISION

Le découplage entre la logique applicative et le traitement des erreurs peut être généralisé pour prendre en compte d'autres aspects non-fonctionnels. Par exemple, une spécification DiaSpec peut être raffinée pour spécifier des contraintes de sécurité [59–61] ou de *QoS* [36, 46]. La violation de l'une de ces contraintes résulte systématiquement par la publication d'un événement exceptionnel pouvant être traité par des composants SCC. Nous appelons alors l'ensemble des composants SCC dédiés à la gestion des événements signalant la violation d'une contrainte non fonctionnelle la *couche de supervision* de l'application. En effet, ces contextes et contrôleurs mettent généralement en oeuvre des techniques de *monitoring* et de reconfiguration afin de garantir le respect des exigences non fonctionnelles telle que les exigences **ex3.** et **ex4.** présentées dans la Section 4.3.

```
1 context IntHeading as Float {
2   source heading from IRU;
3   source targetHeading from NavMMI;
4   interaction {
5     when provided heading from IRU;
6     get targetHeading from NavMMI in 100 ms [mandatory catch];
7     always publish;
8   }
9 }
```

Listing 7: Spécification de contraintes de *QoS*

Le Listing 7 présente un exemple simple de spécification DiaSpec raffinée non seulement avec une contrainte de traitement d'erreur au niveau de l'application, mais également avec une contrainte de *QoS*. En effet le contexte `IntHeading`, dont les premières spécifications sont présentées dans le Listing 3 et le Listing 5, doit également respecter des contraintes temporelles afin que l'application puisse correspondre à des exigences telles que

l'exigence **ex1.**, présentée dans la Section 4.3 et qui définit un temps maximal d'exécution pour la chaîne fonctionnelle à laquelle le contexte appartient. Une de ces contraintes est illustrée à la ligne 6 : elle définit un temps de réponse maximum autorisé pour l'accès à la source `targetHeading` de l'interface avec le pilote, 100ms en l'occurrence. En cas de non-respect de cette contrainte, un événement exceptionnel de type `TimeoutException` est publié.

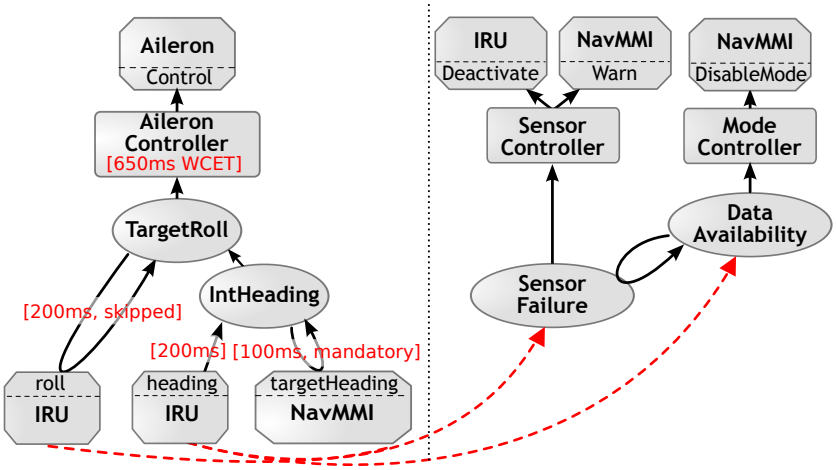

FIGURE 8: Aperçu de la spécification de la *QoS* du mode de gestion du cap

La couche de supervision du mode de gestion du cap doit entre autre gérer des événements exceptionnels `TimeoutException` et `DataFreshnessException` ainsi que l'illustre la Figure 8. Tandis que le premier événement signale un dépassement du délai maximum autorisé pour l'accès aux données de roulis (`roll`) ou au cap cible (`targetHeading`), le second signale une tentative de publication d'une donnée de cap (`heading`) trop âgée. Ces événements sont publiés suite à la violation d'une contrainte temporelle, illustrées par les annotations des flèches représentant les interactions entre composants dans la Figure 8. Ces contraintes sont définies pour s'assurer du respects de l'exigence du WCET (*Worst Case Execution Time* pour temps d'exécution dans le pire cas) définie par l'exigence **ex1.** présentée dans la Section 4.3 et illustrée au niveau du contrôleur `AileronController`. Elles peuvent également être utilisées pour surveiller le bon fonctionnement des capteurs et la disponibilité des données dans le but de respecter les exigences **ex3.** et **ex4.**.

Ainsi les événements exceptionnels signalant la violation des contraintes temporelles viennent alimenter le contexte `SensorFailure` utilisé par le contrôleur `SensorControler` afin de détecter une défaillance d'une entité `IRU`, de la désactiver le cas échéant et de le signaler au pilote ainsi que nous l'avons présenté dans la Section 5.2.3. La Figure 8 introduit également un nouveau contexte, `DataAvailability`, chargé de déterminer si la

qualité de service de l'environnement d'exécution est suffisante pour que le mode de gestion du cap puisse utiliser les données fournies par les capteurs. Dans le cas contraire, ce mode doit être désactivé, ainsi que l'indique l'exigence **ex4.**, ce qui est effectué par le contrôleur `ModeController`.

CONCLUSION En s'appuyant sur le paradigme SCC, DiaSpec permet de décrire à la fois les aspects fonctionnels et non-fonctionnels d'une application. Pour décrire les aspects fonctionnels, les concepteurs d'une application peuvent s'appuyer sur une taxonomie des entités qui permet de s'abstraire de l'hétérogénéité des capteurs et actionneurs d'un domaine, facilitant ainsi l'utilisation de composants sur étagère. A partir de cette taxonomie, la spécification des contextes et contrôleurs, ainsi que de leurs interactions, permet de décrire le flot des données et le flot de contrôle d'une application.

Pour décrire les aspects non-fonctionnels, DiaSpec offre des constructions dédiées qui permettent aux concepteurs de raffiner une spécification avec des contraintes indiquant, par exemple, si les erreurs propagées par les entités doivent être systématiquement compensées au niveau de l'application. Le langage permet également de spécifier des composants SCC dédiés au traitement d'événements exceptionnels. Ces événements signalent des erreurs, compensées ou non au niveau de l'application, mais également la violation de certaines contraintes non-fonctionnelles comme le dépassement d'un délai pour l'acquisition d'une donnée. En spécifiant des contextes et contrôleurs dédiés au traitement de ces événements, les concepteurs définissent une couche de supervision qui permet de surveiller et d'agir sur les aspects non-fonctionnels d'une application indépendamment de la logique applicative.

Finalement, en permettant de spécifier à la fois les aspects fonctionnels et non-fonctionnels d'une application, le support de conception offert par DiaSpec permet de raisonner sa sûreté mais également de guider de façon rigoureuse son développement.

6

IMPLÉMENTATION

Le compilateur de DiaSpec génère un *framework* de programmation Java dédié à la spécification d'une application [22]. Ce *framework* est généré pour guider les développeurs : il offre du support pour les aider à implémenter l'application tout en les limitant à ce qui est autorisé par la spécification, veillant ainsi à préserver la conformité du code vis à vis de la spécification [23].

Dans ce chapitre, nous illustrons le support d'implémentation offert par DiaSuite à travers la présentation du *framework* de programmation généré pour guider l'implémentation du mode de gestion du cap présenté dans la Section 4.3. Nous détaillons ensuite le support offert pour guider l'implémentation des stratégies de traitement des erreurs spécifiées durant la phase conception.

6.1 PROGRAMMATION GUIDÉE PAR LA CONCEPTION

La technique employée pour guider l'implémentation d'une application à partir de sa spécification DiaSpec est de générer une classe abstraite pour chaque composant SCC spécifié. Cette classe abstraite contient des méthodes qui facilitent l'implémentation des composants, notamment en simplifiant les interactions avec d'autres composants. Elle contient également des déclarations de méthodes abstraites qui doivent être implémentées par les développeurs afin de fournir la logique applicative du composant (e. g., calculer une valeur, décider des actions à effectuer, *etc.*).

Pour implémenter un composant SCC, les développeurs doivent créer une sous-classe de la classe abstraite correspondante et implémenter chacune des méthodes abstraites. Cette séparation entre le code généré (i. e., la classe abstraite) et le code utilisateur (i. e. la sous-classe de la classe abstraite) contraste avec la stratégie de génération de code incomplet (ou "à trous") et facilite l'évolution de la spécification comme de l'implémentation [22, 99]. De plus, chaque méthode abstraite correspond à un contrat d'interaction spécifié lors de la conception, ce qui permet d'appliquer le principe de l'*inversion de contrôle* pour garantir la conformité entre le code et la spécification. En effet, les méthodes abstraites à implémenter sont uniquement appelées par le *framework* généré. Cette technique permet de garantir *l'intégrité de communication* des composants : un composant ne peut interagir avec un autre

composant au niveau de l'implémentation que si cela est autorisé par la spécification [23].

```
1  public abstract class AbstractIntHeading {
2
3    public abstract Float onHeadingFromIRU(Float heading,
4                 BindingForHFI binding);
5    ...
6  }
```

Listing 8: Extrait de la classe abstraite générée `AbstractIntHeading`

Par exemple, le Listing 8 présente un extrait de la classe abstraite générée à partir de la spécification DiaSpec du contexte `IntHeading` illustrée par les Listings 3, 5 et 7. La méthode abstraite `onHeadingFromIRU` est générée à partir du contrat d'interaction indiquant que le contexte est activé par la réception de données de cap provenant d'entités `IRU`. Cette méthode est systématiquement appelée par le *framework* de programmation lorsque des données de cap provenant d'une centrale inertielle sont reçues. Afin de guider les développeurs lors de l'implémentation de la méthode, le premier paramètre de la méthode (`heading`) fournit la valeur de la donnée reçue. Le second paramètre, `binding`, permet d'accéder aux autres composants SCC avec lesquels une interaction est autorisée. En l'occurrence, le paramètre `binding` ne permet d'accéder qu'au cap cible fourni par le pilote ainsi que l'indique le contrat d'interaction à partir duquel la méthode est générée. Enfin, le type de retour de la méthode correspond au type du contexte déclaré dans la spécification DiaSpec.

```
1  public class IntHeading extends AbstractIntHeading {
2
3    @Override
4    public Float onHeadingFromIRU(Float heading,
5                 BindingForHFI binding) {
6      NavMMIProxy mmi = binding.navMMI();
7      Float targetHeading = mmi.getTargetHeading();
8      return controllerPID.compute(heading, targetHeading);
9    }
10   ...
11 }
```

Listing 9: Extrait de l'implémentation du contexte `IntHeading`

Ainsi, pour implémenter un composant SCC, les développeurs étendent une classe abstraite générée qui les guide et les contraint dans leur implémentation de la logique applicative du composant. Le Listing 9 présente un exemple d'implémentation du contexte `IntHeading`. La classe `IntHeading` est créée par un développeur. Elle étend la classe abstraite générée `AbstractIntHeading` présen-

tée dans le Listing 8 et implémente donc le composant SCC correspondant, le contexte `IntHeading`. Pour implémenter la logique applicative du composant, les développeurs ont implémenté la méthode abstraite `onHeadingFromIRU`. L'implémentation de cette méthode consiste donc à retourner un résultat fonction du cap reçu (le paramètre `heading`) et du cap cible fourni par le pilote (récupéré à partir du paramètre `binding` et par l'intermédiaire d'une classe *proxy* générée, `NavMMIProxy`). La fonction utilisée pour calculer le résultat, un cap intermédiaire à atteindre, est la fonction `compute` (ligne 8). Elle est implémentée par le développeur et calcule un cap intermédiaire, de type `Float`, en fonction du cap actuel et d'un cap cible, celui fourni par le pilote dans ce cas.

6.2 SUPPORT POUR LE TRAITEMENT DES ERREURS

Pour guider l'implémentation des politiques de traitement des erreurs spécifiées lors de la conception, le compilateur de DiaSpec a été étendu afin de générer un *framework* de programmation qui fournit du support dédié au traitement des erreurs à partir des extensions du langage présentées dans la Section 5.2. Nous présentons tout d'abord ce support puis illustrons comment il permet de guider et contraindre les développeurs au travers d'exemples d'implémentation de composants SCC.

6.2.1 *Le support de programmation généré*

Afin de pouvoir traiter les erreurs à la fois au niveau de l'application et au niveau du système, les erreurs détectées sont signalées par des exceptions de types différents. Puis ces exceptions sont propagées entre différents composants SCC de l'application afin de traiter les erreurs qu'elles signalent. Ainsi, le *framework* de programmation généré offre du support pour la signalisation des erreurs et la propagation des exceptions.

6.2.1.1 *Signalisation des erreurs*

Le *framework* de programmation généré à partir d'une spécification DiaSpec intercepte systématiquement toutes les exceptions Java propagées à travers l'application afin d'offrir un traitement uniforme des exceptions. Ce traitement consiste à signaler les erreurs au niveau du système en publiant un événement exceptionnel puis à lever une exception Java particulière afin de déclencher le traitement au niveau de l'application.

Les exceptions interceptées sont soit définies par l'utilisateur, soit *built-in*. Les exceptions définies par l'utilisateur correspondent à celles déclarées dans la spécification DiaSpec de l'application. Elles sont levées par les implémentations des composants SCC, en utilisant l'instruction **throw** de Java. Une exception qui n'est pas déclarée dans la spécification est considérée comme étant *buit-in* [1]. Par exemple, lorsqu'un composant SCC ne répond pas à une requête d'un composant SCC client, une exception `TimeoutException` est levée après un certain temps par le *proxy* du client. De plus, le compilateur étendu de DiaSpec génère certains mécanismes de détection d'erreurs (e. g., *heartbeat*). Ces mécanismes permettent de détecter, signaler puis traiter les erreurs de manière proactive (i. e., avant qu'elles n'affectent l'application), améliorant ainsi la fiabilité du système.

Lorsqu'une exception est interceptée, un événement exceptionnel est publié pour signaler l'erreur au niveau du système. Cet événement est ensuite raffiné par un composant SCC dédié, un contexte ayant déclaré gérer l'exception interceptée. Ce contexte peut alors agréger, combiner, transformer ou interpréter les événements exceptionnels qu'il reçoit pour propager les informations relatives à l'erreur entre les différents composants SCC chargés de traiter les erreurs au niveau du système.

Suite à la publication de l'événement exceptionnel, une exception de type `ApplicationLevelException` est systématiquement levée par le *framework* de programmation généré. Cette exception sert à signaler qu'une erreur s'est produite au niveau de l'application. Pour éviter que le développeur d'un composant SCC ne traite l'erreur alors qu'il ne le doit pas, par exemple dans le cas où une contrainte **skipped catch** est spécifiée au niveau de la déclaration DiaSpec du composant, l'exception `ApplicationLevelException` étend la classe Java `RuntimeException`. Cette dernière définie des exceptions dont le traitement en Java n'est pas obligatoire, c'est-à-dire qui ne requièrent pas de bloc *try/catch*. De plus, les informations sur l'erreur ayant déjà été signalées au niveau du système, l'exception `ApplicationLevelException` est dénuée de toute information et limite donc les possibilités de traitements lourds pouvant polluer le code au niveau de l'application.

Le Listing 10 illustre un exemple de code généré pour que les erreurs soient systématiquement signalées au niveau de l'application et au niveau du système. La classe générée, `NavMMIProxy` correspond au *proxy* retourné par la méthode `binding.navMMI` et utilisé pour accéder au cap cible fourni par le pilote dans d'implémentation du contexte `IntHeading` (voir le Listing 9). La méthode privée `proxyGetTargetHeading` est systématiquement

1. La Figure 6 de la Section 5.2.1 présente la hiérarchie des différentes exceptions *built-in*

```
1  public class NavMMIProxy {
2
3    private Float proxyGetTargetHeading() {
4      try {
5        return bus.getComponent(...).getTargetHeading();
6      } catch (RangeException e) {
7        publishRangeException(e);
8        throw new ApplicationLevelException();
9      } catch (DiaSpecException e) {
10       publishDiaSpecException(e);
11       throw new ApplicationLevelException();
12     } catch (ApplicationLevelException e) {
13       throw e;
14     } catch (Exception e) {
15       publishUndeclaredException(new UndeclaredException(e));
16       throw new ApplicationLevelException();
17     }
18   }
19   ...
20 }
```

Listing 10: Signalisation systématique des erreurs

appelée par le *framework* lorsque le développeur fait appel à la méthode `getTargetHeading` sur le *proxy*. Cette méthode est générée pour intercepter et signaler toutes les exceptions provenant du composant SCC distant ou de la plateforme d'exécution. Les exceptions définies par l'utilisateur sont interceptées et signalées les premières. Par exemple, l'entité `NavMMI` peut lever des exceptions de type `RangeException` pour signaler une valeur de cap ne respectant pas un intervalle particulier (0 à 360 degrés). Ces exceptions sont interceptées (ligne 6) puis publiées sous la forme d'un événement exceptionnel (ligne 7). Les exceptions *built-in* sont également interceptées (ligne 9) puis publiées (ligne 10). Seule l'exception `ApplicationLevelException` est interceptée mais n'est pas publiée : cette exception est propagée à travers le *framework* de programmation généré afin de déclencher le traitement des erreurs au niveau de l'application. Ainsi, l'exception `ApplicationLevelException` est systématiquement levée lorsqu'une exception est interceptée (lignes 8, 11 et 16). Enfin, toute autre exception est interceptée (ligne 14) puis transformée en une exception *built-in* particulière de type `UndeclaredException`. Cette dernière est signalée au niveau du système (ligne 15) avant qu'une exception `ApplicationLevelException` ne soit propagée au niveau de l'application (ligne 16).

6.2.1.2 *Propagation des exceptions*

Lorsqu'une exception est interceptée, une exception `ApplicationLevelException` est propagée au niveau de

l'application tandis que les informations sur l'erreur sont propagées au niveau du système suite à la publication d'un événement exceptionnel. La propagation de l'exception `ApplicationLevelException` est assurée par le *framework* de programmation généré afin d'assurer le respect des contraintes de traitement des erreurs spécifiées en phase de conception. Complètement transparentes pour les développeurs de l'application, les exceptions `ApplicationLevelException` sont systématiquement interceptées par les *proxys* des composants SCC afin de pouvoir déclencher un traitement d'erreur dans le cas d'une contrainte **mandatory catch** ou **optional catch**. Dans le cas d'une contrainte **skipped catch**, l'exception est propagée de façon transparente pour le développeur afin de ne pas l'inciter à effectuer un traitement d'erreur. En effet, l'exception `ApplicationLevelException` étend l'exception Java `RuntimeException` et ne réclame donc pas de bloc *try/catch* de la part du développeur.

```
1  public class NavMMIProxy {
2    ...
3    public Float getTargetHeading(FloatContinutation fc) {
4      try {
5        return this.proxyGetTargetHeading();
6      } catch (ApplicationLevelException e) {
7        return fc.onError();
8      }
9    }
10   ...
11 }
```

Listing 11: Application systématique du traitement des erreurs au niveau applicatif

Le Listing 11 illustre la façon dont les exceptions `ApplicationLevelException` sont utilisées pour appliquer le traitement des erreurs requis dans le cas d'une contrainte **mandatory catch** ou **optional catch**. Il présente un extrait de la classe `NavMMIProxy`, le *proxy* de l'entité `NavMMI` généré dans le *framework* de programmation pour guider l'implémentation du contexte `IntHeading`. En particulier, c'est la méthode `getTargetHeading` qui est présentée. Cette méthode est directement appelée par le développeur du contexte `IntHeading` pour accéder à la source `targetHeading` de l'entité (voir l'exemple d'implémentation du contexte présenté par le Listing 9). La méthode est générée de telle sorte qu'elle requiert de fournir une continuation en paramètre. C'est cette continuation qui contient le traitement d'erreur fourni par le développeur. Son type dépend du type de valeur retourné par la méthode. Ainsi, lorsque le développeur appelle la méthode pour accéder au cap cible fourni par le pilote, il doit également fournir un paramètre de type `FloatContinuation` qui

contient une méthode `onError` retournant une valeur de type `Float`. La méthode `getTargetHeading` va alors appeler la méthode `proxyGetTargetHeading` présentée par le Listing 10 pour obtenir le cap cible et intercepter les exceptions propagées en cas d'erreur. Cependant, lorsqu'une erreur est détectée, une exception de type `ApplicationLevelException` est propagée par la méthode `proxyGetTargetHeading`. Dans ce cas, la méthode `getTargetHeading` va intercepter l'exception et appliquer le traitement des erreurs en appelant la méthode `onError` fournie par le développeur via la continuation. Lorsque c'est la contrainte **skipped catch** qui est spécifiée en phase de conception au lieu des contraintes **mandatory catch** ou **optional catch**, la méthode est générée sans la continuation en paramètre et n'applique aucun traitement. L'exception `ApplicationLevelException` est donc automatiquement propagée.

```
1 public interface Continuation<T> {
2   public T onError ();
3 }
4
5 public interface FloatContinuation extends Continuation<Float> {}
```

Listing 12: Les interfaces des continuations

Le Listing 12 présente l'interface générée pour la continuation utilisée par la méthode `getTargetHeading`. En effet, l'interface `FloatContinuation` est générée à partir de la déclaration DiaSpec du contexte `IntHeading`, lorsque qu'il est spécifié que les erreurs provenant de l'entité `NavMMI` doivent (ou peuvent) être traitées suite à une tentative d'accès à la source `targetHeading`. Elle étend l'interface générique `Continuation` qui déclare une méthode `onError` dont le type de retour reste à définir. Ainsi, pour chaque contrainte **mandatory catch** ou **optional catch** utilisée au niveau de la spécification DiaSpec de l'application, une interface spécifique est générée pour contraindre le type de retour de la méthode `onError` en fonction de l'accès à la source faisant l'objet de la contrainte.

Au niveau du système, les événements exceptionnels sont traités par des contextes dédiés. Ces événement contiennent les exceptions Java interceptées par le *framework* et permettent donc de propager des informations sur l'erreur à l'origine de l'exception entre différents composants SCC. Le Listing 13 présente une exemple de classe abstraite générée pour guider l'implémentation des contextes qui traitent des événements exceptionnels. En l'occurrence, c'est la classe abstraite générée à partir de la déclaration du contexte `SensorFailure` (voir le Listing 6) qui est présentée. Le contexte déclarant traiter les exceptions `LowAccuracyException`, `DiaSpecException` et `FailureException`, des méthodes abstraites

```
1  public class AbstractSensorFailure {
2
3    public abstract SensorIDPublishable
         onNewLowAccuracyException(IRUProxyForLFI proxy,
         LowAccuracyException e);
4
5    public abstract SensorIDPublishable
         onNewDiaSpecException(IRUProxyForDFI proxy, DiaSpecException
         e);
6
7    public abstract SensorID onNewFailureException(IRUProxyForFFI
         proxy, FailureException e);
8    ...
9
10 }
```

Listing 13: Extrait de la classe abstraite générée `AbstractSensorFailure`

sont générées pour traiter chacune d'entre elles. Par exemple, la méthode abstraite `onNewLowAccuracyException` est générée pour traiter l'exception `LowAccuracyException`. Ces méthodes sont invoquées par le *framework* de programmation lorsque le contexte `SensorFailure` reçoit un événement exceptionnel correspondant à l'une des exceptions. Pour aider les développeurs à extraire les informations au sujet de l'erreur à l'origine de l'exception, ces méthodes fournissent en paramètre un *proxy* représentant l'entité à l'origine de l'exception (`IRU` dans le cas présent) et l'exception Java interceptée à l'origine de la publication de l'événement exceptionnel. Le type de retour des deux première méthodes présentées dans le listing, `SensorIDPublishable`, est un type généré qui encapsule une valeur de type `SensorID` et un booléen indiquant si la valeur en question doit être publiée. La génération de ce type particulier est la conséquence directe de la clause **maybe publish** du contrat d'interaction correspondant dans la déclaration du contexte. Au contraire, le second contrat d'interaction du contexte spécifie qu'un `SensorID` doit être systématiquement publié à la réception d'un événement exceptionnel indiquant une exception `FailureExcepiton`. Il en résulte que le type de retour généré pour la méthode abstraite `onNewFailureException` est de type `SensorID`. Ainsi, la valeur retournée par la méthode sera systématiquement publiée par le *framework* de programmation sans que le développeur n'ait à l'indiquer.

6.2.2 *Implémentation du traitement des erreurs*

Le support généré pour le traitement des erreurs fait parti du *framework* de programmation généré et repose également sur le

principe de l'inversion de contrôle. Ainsi, les développeurs de l'application sont guidés lors de l'implémentation des composants SCC et l'établissement de la traçabilité entre conception et implémentation est automatisé, aussi bien au niveau de l'application que du système.

6.2.2.1 *Au niveau de l'application*

Au niveau de l'application, les développeurs doivent compenser les erreurs lorsqu'un contrat d'interaction comporte une déclaration **mandatory catch**. Le *framework* de programmation assure le respect de cette contrainte en faisant systématiquement appel à une continuation lorsqu'une exception est interceptée (voir les Listings 11 et 12). Le Listing 14 présente un exemple d'implémentation d'un contexte, le contexte `IntHeading`, pour lequel un traitement systématique des erreurs est exigé. En effet, il a été indiqué en phase de conception que le contexte doit systématiquement traiter les erreurs lors d'un accès au cap cible fourni par le pilote (voir le Listing 5, Section 5.2.2). Ainsi un développeur de ce contexte doit systématiquement traiter les erreurs pouvant être propagées lors de l'appel à la méthode `getTargetHeading` sur le *proxy* `NavMMIProxy`. Grâce au *framework* de programmation généré, le développeur est contraint de respecter cette contrainte : il doit implémenter la continuation de type `FloatContinuation` exigée par la méthode `getTargetHeading` et qui sera systématiquement utilisée pour compenser les erreurs. Le développeur est alors contraint d'implémenter cette interface et de fournir une méthode `onError` qui retourne une valeur de type `Float` pour compenser l'échec de l'accès au cap cible. Dans l'exemple présenté, c'est une valeur par défaut qui est utilisée comme compensation.

```
1  public class IntHeading extends AbstractIntHeading {
2
3    @Override
4    public Float onHeadingFromIRU(Float heading,
5                  BindingForHFI binding) {
6      NavMMIProxy mmi = binding.navMMI();
7      Float targetHeading = mmi.getTargetHeading(
8        new TargetHeadingContinuation() {
9          public Float onError() {
10           return DEFAULT_VALUE;
11         }});
12     return controllerPID.compute(heading, targetHeading);
13   }
14   ...
15 }
```

Listing 14: Obligation d'implémentation pour le traitement des erreurs

En devant systématiquement retourner une valeur de même type que celle retournée par la méthode compensée et en n'ayant pas accès aux informations sur les causes de l'erreur, le développeur est contraint dans sa façon d'implémenter le traitement des erreurs. Cependant, il reste suffisamment libre pour implémenter différentes techniques de traitement des erreurs. Par exemple, le Listing 15 présente une compensation par redondance temporelle. Dans cet exemple, le développeur appel à nouveau la méthode `getTargetHeading` en cas d'erreur et n'utilise une valeur par défaut qu'en ultime recours.

```
Float targetHeading = mmi.getTargetHeading(
  new TargetHeadingContinuation() {
    public Float onError() {
      return mmi.getTargetHeading(
        new TargetHeadingContinuation() {
          public Float onError() {
            return DEFAULT_VALUE;
          }
        }
      );
    }
  }
);
```

Listing 15: Exemple d'une obligation d'implémentation de traitement des erreurs : utilisation de la redondance temporelle

Lorsqu'une déclaration **skipped catch** est utilisée, les exceptions pouvant être propagées sont masquées aux développeurs. Dans ce cas, les méthodes d'accès aux sources des entités ou aux informations fournies par les autres contextes ne requiert pas de continuation. Un développeur ne peut donc pas traiter les erreurs comme il l'aurait fait dans le cas d'un **mandatory catch** et le code n'est donc pas pollué par du code de traitement des erreurs inutile. Par exemple, le Listing 9 utilisé pour illustrer le support de base fourni par DiaSpec, c'est-à-dire sans traitement des erreurs, est en tout point identique à une implémentation du contexte `IntHeading` pour lequel la déclaration **skipped catch** aurait été préférée. En effet, la seule différence notable se trouverait au niveau de l'exécution : le support généré pour le traitement des erreurs intercepterait les exceptions propagées lors de l'appel de la méthode `getTargetHeading` afin qu'elles n'atteignent pas le haut de la pile d'appel Java, évitant ainsi la défaillance de l'application.

Dans le cas d'une déclaration **optional catch**, un développeur peut choisir entre traiter les erreurs ou les ignorer afin qu'elles puissent éventuellement être traitées par un autre contexte. Ainsi, lorsque la déclaration **optional catch** est utilisée dans un contrat

d'interaction, les deux versions de la méthode générée pour l'accès aux données des autres composants SCC (e. g., la méthode `getTargetHeading`) sont générées. Le développeur peut alors décider de traiter les erreur en appelant la méthode avec une continuation en paramètre ou de les ignorer en appelant celle qui n'exige pas de continuation en paramètre.

6.2.2.2 *Au niveau du système*

Au niveau du système, les développeurs doivent implémenter les composants SCC dédiés au traitement des erreurs qui ont été définis en phase de conception. L'implémentation de ces composants est similaire à celle des composant qui constituent la logique applicative de l'application. Les développeurs sont donc à la fois guidés pour l'implémentation du traitement des erreurs et contraints afin d'assurer la conformité entre le code et la spécification DiaSpec de l'application. Ils peuvent ainsi se concentrer uniquement sur l'implémentation des différentes étapes des stratégies de traitement des erreurs définies lors de la phase de conception.

```
1  public class SensorFailure extends AbstractSensorFailure {
2
3    @Override
4    public SensorIDPublishable
         onNewLowAccuracyException(IRUProxyForLFI proxy,
         LowAccuracyException e) {
5      recordException(proxy.id(), e,
           System.getCurrentTimeInMillis());
6      boolean doPublish = getExceptionFrequency(proxy.id(), e) > 0.1;
7      return new SensorIDPublishable(proxy.id(), doPublish);
8    }
9
10   @Override
11   public SensorIDPublishable onNewDiaSpecException(IRUProxyForDFI
         proxy, DiaSpecException e) {
12     recordException(proxy.id(), e,
           System.getCurrentTimeInMillis());
13     boolean doPublish = getExceptionFrequency(proxy.id(), e) >
           0.25;
14     return new SensorIDPublishable(proxy.id(), doPublish);
15   }
16
17   @Override
18   public SensorID onNewFailureException(IRUProxyForFFI proxy,
         FailureException e) {
19     return proxy.id();
20   }
21   ...
22 }
```

Listing 16: Extrait de l'implémentation du contexte `SensorFailure`

En pratique, seule la façon d'implémenter les contextes qui traitent les événements exceptionnels diffère de celle dont sont implémentés les contextes qui raffinent les sources de données déclarées par les entités. En effet, la méthodes abstraite générée à partir d'un contrat d'interaction indiquant qu'un contexte traite des événements exceptionnels fournit à la fois l'exception à l'origine de l'événement et des informations sur l'entité responsable de l'exception. Ces informations sont particulièrement utiles pour surveiller individuellement l'état de fonctionnement des différentes entités présentes au sein du système. Par exemple, le Listing 16 présente l'implémentation du composant `SensorFailure` qui traite les événements exceptionnels `LowAccuracyException`, `DiaSpecException` et `FailureException` afin de déterminer si une instance d'une entité `IRU` est défaillante. Le cas échéant, son identifiant est publié. Ainsi, dans le cas d'un événement signalant une erreur pouvant être temporaire (e. g., `LowAccuracyException`, `DiaSpecException`), le contexte mémorise l'exception et l'identifiant de l'entité incriminée puis calcule la fréquence à laquelle l'exception a été rencontrée pour cette entité. Si cette fréquence est trop élevée, l'entité est considérée comme défaillante et son identifiant est publié. Dans le cas d'un événement signalant directement la défaillance d'une entité (un événement `FailureException`), le contexte se contente d'extraire l'identifiant et de le publier.

CONCLUSION A partir de la spécification DiaSpec d'une application, un *framework* de programmation dédié est généré pour guider les développeurs et préserver la conformité entre conception et implémentation. Ce support de développement repose essentiellement sur la génération de classes abstraites que les développeurs doivent implémenter pour chaque composant SCC de la spécification. L'extension du langage DiaSpec pour le traitement des erreurs permet également de générer un support de programmation dédié. En effet, tandis que la déclaration **mandatory catch** est à l'origine de la génération de continuations dans les classes abstraites du *framework* pour contraindre les développeurs à compenser les erreurs au niveau de l'application, la déclaration **skipped catch** est à l'origine de la génération de code interceptant les exceptions Java pour les masquer et éviter tout traitement inutile. Qu'elles soient compensées ou non, les erreurs sont signalées aux composant SCC de la couche de supervision par des événements exceptionnels. Grâce à la génération de ce support, la conformité entre conception et implémentation est préservée, aussi bien pour les aspects fonctionnels que pour les aspects non-fonctionnels. Cependant, il reste nécessaire de vérifier la cohérence de la spécification DiaSpec et de tester le code des développeurs afin de garantir que l'application est bien sûre et conforme aux exigences de haut-niveau.

7

VÉRIFICATION

Afin d'aider les développeurs à éliminer un maximum de fautes pendant le développement d'une application SCC, DiaSuite fournit du support pour la vérification statique et dynamique de l'application. Le support pour la vérification statique permet d'effectuer des analyses et de faciliter la vérification de propriétés fonctionnelles et non-fonctionnelles de l'application pendant sa phase de conception. Le support pour la vérification dynamique, c'est-à-dire le support de test, permet de faciliter la vérification du comportement de l'application lors de son exécution, notamment grâce à la simulation.

7.1 VÉRIFICATION STATIQUE

De nombreuses vérifications statiques peuvent être effectuées à partir de la spécification DiaSpec d'une application. Le compilateur du langage effectue systématiquement des analyses d'une spécification avant de générer le *framework* de programmation qui guide son implémentation [21, 23]. De plus, il est désormais possible de générer des modèles décrits par des automates temporisés afin d'utiliser des techniques de *model checking* pour la vérification de propriétés fonctionnelles et non-fonctionnelles [36].

7.1.1 *Analyses d'une spécification DiaSpec*

Le compilateur de DiaSpec effectue un ensemble d'analyses sur une spécification avant de générer le *framework* de programmation. Les analyses syntaxiques sont automatiquement conduites à partir de la grammaire du langage qui est décrite en *ANTLR* [1]. Par exemple, la grammaire interdit l'utilisation du mot-clé `context` dans la déclaration d'une entité. La grammaire de la version de base de DiaSpec est présentée par Cassou dans sa thèse [21].

Différentes vérifications sémantiques sont également effectuées sur une spécification DiaSpec. Par exemple, l'unicité des noms et des dépendances est vérifiée afin d'assurer que chaque composant, source ou action ainsi que chaque dépendance ne soit déclaré qu'une seule et unique fois. De plus, l'existence des références est vérifiée pour éviter qu'un composant ne déclare

1. `http://antlr.org/`

une référence vers un composant, une source ou une action inexistant. La vérification de l'absence de cycle de dépendance est également effectuée pour empêcher les situations d'inter-blocages (*dead-locks*).

En spécifiant le flot des données et de contrôle entre les composants SCC, les contrats d'interactions permettent d'effectuer des analyses plus fines [23]. Par exemple, le compilateur de DiaSpec peut signaler des contrats inutiles lorsqu'aucun flot de données n'en permet l'activation. De plus, afin d'éviter tout indéterminisme, les contrats d'un même composant ne peuvent interférer entre eux en ayant une même condition d'activation.

Les extensions du langages permettant la déclarations d'exceptions et leur traitement par des contextes font également l'objet de vérifications. Au niveau du système, les exceptions et les contextes qui les traitent fonctionnent comme les sources des entités et les contextes qui les raffinent, les vérifications effectuées sont donc les mêmes (unicité des noms et références, absence de dépendance cyclique, contrats d'interactions, *etc.*).

Nous avons étendu le compilateur de DiaSpec afin de permettre d'effectuer des analyses plus fines sur les contraintes de traitement des erreurs au niveau de l'application. Ces analyses consistent essentiellement à vérifier la couverture des flots de propagation des erreurs par les déclarations **mandatory catch**. En effet, une alerte de compilation est signalée pour tout flot de propagation des erreurs ne faisant pas l'objet d'une déclaration **mandatory catch** afin de prévenir les concepteurs que certaines erreurs sont ignorées. Par exemple, lors de la compilation des spécifications illustrées par les Figures 7 et 8 du Chapitre 5, une alerte signale que les erreurs provenant des tentatives d'accès au roulis fourni par l'entité `IRU` ne sont jamais traitées au niveau de l'application. Les concepteurs peuvent alors choisir d'ignorer volontairement les erreurs en ne tenant pas compte de l'alerte où bien de remplacer la déclaration **skipped catch** du contexte `TargetRoll` par une déclaration **mandatory catch**.

Une autre analyse est effectuée pour vérifier l'absence de traitement des erreurs inutiles. En effet, puisque l'on considère que seules les entités peuvent être la source des erreurs propagées entre les composants SCC, l'utilisation d'une déclaration **mandatory catch** stoppe la propagation des erreurs provenant de cette entité et rend donc inutile l'utilisation d'une autre déclaration. Par exemple, une déclaration **mandatory catch** ou **skipped catch** est inutile pour tout contexte accédant à un autre contexte qui traite déjà systématiquement toutes les erreurs pouvant lui être propagées.

Finalement, les analyses sur la propagation des erreurs au niveau de l'application se rapprochent des analyses effectuées

sur la propagation des exceptions en Java. En effet l'effet de la déclaration **mandatory catch** sur le flot de propagation des erreurs est similaire à celui de la clause `catch`. Cependant, DiaSpec diffère de Java par l'absence de typage pour le signalement des erreurs au niveau de l'application et l'impossibilité de propager à nouveau des erreurs.

7.1.2 *Model checking*

Puisque DiaSpec permet d'expliciter les flots de données et de contrôle, il est possible de vérifier certaines propriétés lors de la conception d'une application. En effet, un modèle formel de l'application peut être généré à partir de sa spécification DiaSpec, permettant ainsi de vérifier la cohérence et la conformité de l'application par rapport aux exigences de haut niveau dès les premières phases du développement [36]. Pour permettre la vérification de propriétés issues d'exigences fonctionnelles et non-fonctionnelles, un compilateur génère des modèles exprimés à l'aide d'automates temporisés qui prennent en compte les contraintes de traitement des erreurs et de *QoS* d'une spécification DiaSpec. La spécification est transformée en un réseau d'automates temporisés où chaque automate décrit le comportement d'un composant SCC écrit en DiaSpec. Ce réseau d'automates permet de vérifier des propriétés de sûreté en utilisant des techniques de *model checking*. En l'occurrence, nous utilisons UPPAAL, une suite d'outils qui fournit un environnement de vérification dédié à la modélisation, la validation et la vérification de réseaux d'automates temporisés [13].

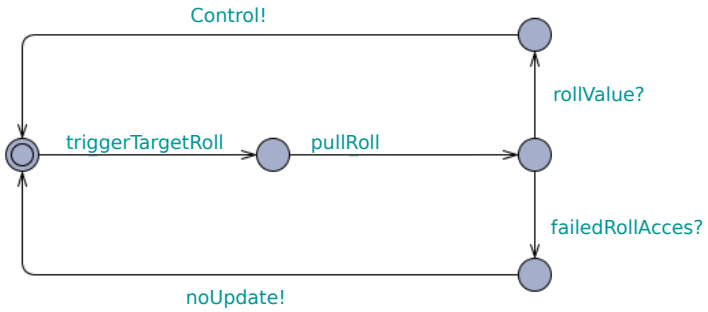

FIGURE 9: Automate temporisé du contexte `TargetRoll`

La Figure 9 présente un exemple d'automate temporisé généré à partir d'une spécification DiaSpec. Cet automate est généré à partir de la déclaration DiaSpec du contexte `TargetRoll` (illustrée dans la Figure 8, Section 5.3). Le contexte est activé à la réception d'un événement provenant du contexte `IntHeading`, ce qui correspond à la transition `triggerTargetRoll` de l'auto-

mate. Dès son activation, le contexte accède aux données de roulis fournies par l'entité `NavMMI`. Cet accès est représenté par la transition `pullRoll`. Lorsqu'une valeur est récupérée (transition `rollValue`), le contexte peut alors publier un résultat au contrôleur `AileronController` (transition `Control!`). En cas d'échec de l'accès aux données de roulis (transition `failedRollAccess?`), la déclaration **skipped catch** indique que les erreurs ne doivent pas être traitées, le contexte revient alors dans son état initial (transition `noUpdate!`).

En plus des automates décrivant le comportement des composants SCC, des automates d'interaction sont générés pour former un réseau d'automates qui peut être utilisé pour vérifier à la fois des propriétés sur les flots de contrôle, de données et de propagation d'erreur ainsi que sur les aspects temporels de l'application. La vérifications de ces propriété permet ainsi de vérifier la cohérence de la spécification DiaSpec de l'application ainsi que sa conformité avec les exigences de haut niveau.

7.1.2.1 *Vérification de la cohérence*

Les incohérences entre des contraintes temporelles peuvent être automatiquement détectées par le *model checker* de UPPAAL. Les propriété temporelles dépendent des hypothèses faites sur les modes de communications entre composant (e. g. communication synchrones/asynchrones, mise en tampon des données). Ces hypothèses sont exprimées sous la forme de paramètres des modèles UPPAAL générés. Par exemple, dans le modèle généré à partir de la description DiaSpec de la gestion du cap, les composants n'utilisent pas de tampon et consomment donc les valeurs qu'ils reçoivent immédiatement. Dans ce cas, un *deadlock* est détecté si l'entité `NavMMI` prend plus de 200 ms pour répondre à une demande formulée par le contexte `IntHeading` sur la valeur du cap cible fourni par le pilote. En effet, le contexte `IntHeading` reçoit toutes les 200 ms un cap provenant de l'entité `IRU` et le compare à celui qu'il demande à l'entité `NavMMI`. Il ne peut donc être disponible pour recevoir et traiter le cap provenant de l'entité `IRU` si il attend plus de 200 ms une réponse de l'entité `NavMMI`. Suite à cette vérification, la spécification DiaSpec de l'application a été enrichie afin de spécifier un délai maximal de réponse de 100 ms pour l'accès au cap cible par le contexte `IntHeading` (voir la Figure 8, Section 5.3).

Un exemple plus complexe de vérification similaire porte sur l'interaction entre le contexte `TargetRoll` et l'entité `IRU`. Un *deadlock* est détecté quand l'accès aux données de roulis prend plus de 300 ms : le contexte `TargetRoll` n'est plus disponible pour traiter les informations provenant du contexte `IntHeading` dans

ce cas. Le contre-exemple le plus court fournit par UPPAAL pour illustrer ce problème inclut trois requêtes de données et peut donc être facilement manqué par les concepteurs en l'absence d'outils.

7.1.2.2 *Vérification de la conformité*

Pour vérifier la conformité de la spécification DiaSpec avec les exigences de haut niveau, ces dernières sont exprimées sous la forme de propriétés basées sur la logique temporelle. Le *model checker* de UPPAAL utilise un sous-ensemble de TCTL (*Timed Computation Tree Logic*) [54]. Par exemple la propriété TCTL ci-dessous sert à exprimer en partie l'exigence de haut niveau **ex4.**, "Un mode de navigation doit être désactivé lors de la défaillance d'un capteur impliqué dans son fonctionnement" (voir Section 4.3).

$$\texttt{IRU.Failure} \rightsquigarrow \texttt{NavMMI.DisableMode}$$

Cette propriété indique en effet que lorsque l'automate généré à partir de l'entité `IRU` se trouve dans l'état `Failure` représentant la défaillance de l'entité, celui généré à partir de l'entité `NavMMI` doit finir par atteindre l'état `DisableMode` représentant la désactivation d'un mode de navigation.

Même si la conformité et la cohérence ne peuvent être entièrement garanties lors de la conception, le support de vérification fourni permet de guider la phase de conception d'une application à partir des exigences de haut-niveau. En effet, lorsqu'une propriété n'est pas satisfiable, les contre-exemples générés par UPPAAL permettent d'aider le concepteur d'une application à améliorer la spécification DiaSpec de cette dernière. De plus, l'approche générative de DiaSuite permet de garantir la conformité entre la conception et l'implémentation, préservant ainsi la validité des propriétés vérifiées à travers les différentes étapes du développement de l'application.

7.2 VÉRIFICATION DYNAMIQUE

Bien que certaines fautes peuvent être évitées grâce aux vérifications statiques effectuées à partir de la spécification DiaSpec d'une application, d'autres fautes peuvent êtres introduites pendant le développement. Eliminer ces fautes requiert alors de vérifier le comportement de l'application lors de son exécution. Cette vérification dynamique est effectuée pendant la phase de test du développement de l'application. Par exemple, une faute de programmation peut être introduite lors de l'implémentation du traitement effectué par un composant SCC. Pour éliminer cette

faute, il faut détecter les erreurs qui en sont la conséquence grâce à des tests puis en déduire les modifications à effectuer dans le code de l'application.

L'approche générative de DiaSuite permet de faciliter les tests d'une application. En effet, le *framework* de programmation généré à partir d'une spécification DiaSpec permet de s'abstraire de l'hétérogénéité des entités. Ce niveau d'abstraction permet notamment de tester l'application en utilisant un environnement simulé. A partir de la la taxonomie des entités, et grâce au *framework* de programmation généré, les testeurs peuvent implémenter des pseudo-entités qui interagissent avec un environnement simulé. Ces entités sont alors utilisées pour tester l'application sans qu'aucune modification ne soit nécessaire pour le reste de l'application.

En avionique, il est nécessaire de vérifier le comportement de l'application dans des conditions environnementales particulières. Puisque certains de ces scénarios sont difficiles à reproduire (e. g., des conditions de vol extrêmes), un support de test a été réalisé pour faciliter l'utilisation d'un simulateur de vol nommé *FlightGear* [76].

```
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 17: Extrait de l'entité simulée `IRU`

En utilisant une bibliothèque de programmation Java qui fait l'interface avec *FlightGear*, les testeurs peuvent aisément implémenter les versions simulées des entités utilisées par une application comme le gestionnaire de vol. Le Listing 17 présente un extrait de l'implémentation d'une entité `IRU` simulée. L'entité `SimulatedIRU` est implémentée en héritant de la classe abstraite `AbstractIRU` fournie par le *framework* de programmation généré. Pour interagir avec l'environnement simulé, l'entité implémente l'interface `SimulatorListener`. Cette interface définit une méthode `simulationUpdated` qui est appelée périodiquement par la bibliothèque de programmation afin de transmettre les données provenant de l'environnement simulé. Le paramètre `model` permet d'écrire et de lire l'état courant du simulateur *FlightGear*. Dans le Listing 17, la position de l'avion est publiée en appelant la méthode `publishPosition` de la classe abstraite `AbstractIRU`.

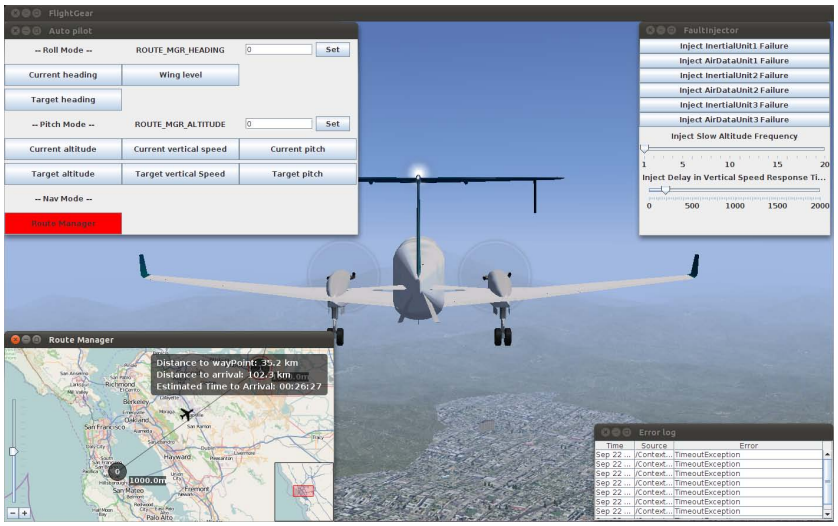

FIGURE 10: Capture d'écran d'un vol simulé

Une fois les entités simulées implémentées, l'application de gestion de vol est testée en contrôlant un avion simulé au sein de *FlightGear*. Un exemple de scénario de test consiste à fournir un cap cible via l'interface de pilotage (l'entité `NavMMI`) et de vérifier que l'application contrôle les ailerons de l'avion simulé conformément à ce qui est attendu. La Figure 10 présente une capture d'écran de l'environnement de test construit autour du simulateur de vol. Dans la fenêtre principale, le simulateur de vol *FlightGear* permet de contrôler et de visualiser l'avion simulé et divers paramètres de vol comme la météo. Dans le coin en haut à gauche, l'interface de pilotage permet au testeur de sélectionner un mode de navigation et d'indiquer, par exemple, un cap cible. Dans le cas présent, le mode *Route Manager* est sélectionné pour indiquer que l'altitude et le cap à suivre sont fournis par un plan de vol. Ce plan de vol est défini par l'intermédiaire de la carte interactive affichée dans le coin en bas à gauche de la capture d'écran.

L'environnement simulé permet également de tester la couche de supervision d'une application SCC. En effet, *FlightGear* permet de simuler les pannes de différents systèmes embarqués dans l'avion. De plus, la bibliothèque de programmation permet d'injecter des fautes depuis les entités simulées. Cette fonctionnalité est illustrée dans la Figure 10 : la fenêtre `FaultInjector`, en haut à droite de la capture d'écran, permet de provoquer le crash ou d'altérer la performance des différents capteurs utilisés par l'application. La couche de supervision du gestionnaire de vol qui est en charge d'afficher les erreurs détectées au pilote doit alors indiquer les erreurs qui résultent des fautes injectées grâce à la fenêtre située en bas à droite de la capture d'écran. Ce type de support permet de tester à la fois les aspects fonctionnels et non-

fonctionnels d'une application et facilite ainsi la vérification de la conformité de l'application avec des exigences de haut-niveau telles que celles présentées dans la Section 4.3.

Une fois l'implémentation de l'application testée, il reste à réaliser des tests d'intégration sur bancs d'essais afin de vérifier que l'application se comporte correctement lorsqu'elle est déployée d'une certaine façon sur un matériel donné. Cependant cette phase de test peut s'avérer particulièrement pénible : lorsqu'une erreur est détectée, il faut corriger l'application et la déployer à nouveau sur les bancs de tests. Le support de simulation fourni a pour avantage de permettre de combiner des entités réelles et simulées dans un environnement hybride afin de déployer progressivement l'application sur les bancs d'essais. En effet, les versions réelles et simulées des entités héritant de la même classe abstraite générée, l'utilisation d'une version plutôt qu'une autre n'a aucun impact sur le reste de l'application. Ainsi, en testant l'application grâce à un environnement simulé puis en la déployant progressivement sur les bancs de tests, il est possible de limiter certains déploiement inutiles.

CONCLUSION Notre approche offre du support pour vérifier la conception d'une application à partir de sa spécification DiaSpec mais également pour effectuer des tests dans un environnement simulé. Grâce à la génération d'un réseau d'automates temporisés à partir de la spécification, différentes propriétés fonctionnelles et non-fonctionnelles peuvent êtres vérifiées. La génération du *framework* de programmation dédié permet alors de préserver la validité de ces propriétés. Cependant, certaines propriétés ne peuvent être vérifiées statiquement. Le support de simulation permet alors de faciliter la phase de test de l'application en permettant d'observer le comportement de l'application et sa fiabilité dans un environnement réaliste. Ainsi, notre approche offre du support pour le développement et la vérification des applications sûres de fonctionnement tout au long du processus de développement.

Troisième partie

VALIDATION

8

EVALUATION

Bien qu'appartenant à des domaines variés, de nombreuses applications requièrent de mettre en oeuvre des solutions pour assurer leur sûreté de fonctionnement. Cependant, la complexité de ces applications rend la mise en place de ces solutions de plus en plus difficile. Notre approche vise à simplifier le développement de telles applications en prenant en compte les aspects de sûreté de fonctionnement dès la conception d'une application et en reposant sur une approche outillée afin de guider l'ensemble du processus de développement. Pour valider cette approche, nous avons développé des applications appartenant aux domaines de l'aéronautique et de l'informatique ubiquitaire.

8.1 AÉRONAUTIQUE

Dans l'aéronautique, le développement d'applications logicielles est fortement contraint par des normes telles que la DO-178B [33]. Ces normes ont pour objectif d'encadrer rigoureusement le développement d'une application, en fonction de son niveau de criticité, afin de certifier le logiciel. En particulier, elles mettent l'accent sur la démonstration de la conformité de l'application avec les exigences de haut niveau en préconisant l'établissement de la traçabilité de ces exigences à travers les différentes étapes du développement. De plus, cette démonstration requiert de vérifier la cohérence de l'application, notamment en prenant en compte à la fois les aspects fonctionnels et non-fonctionnels tels que la sûreté de fonctionnement et la performance.

Nous avons validé notre approche au regard de ces éléments en réalisant une étude de cas qui consiste à développer un gestionnaire de vol pour avion. Ce dernier est présenté dans la Section 4.3 et utilisé comme exemple fil rouge tout au long de ce document. Afin de s'assurer de sa validité, nous l'avons testé grâce au simulateur de vol réaliste *FlightGear* [76]. Au cours de son développement, nous nous sommes notamment attachés à vérifier la cohérence de ses aspects fonctionnel et non-fonctionnels ainsi que la conformité avec les exigences de haut niveau. De plus, nous avons reproduit ce cas d'étude en développant une version du gestionnaire de vol pour un drone commercial, l'*A.R.Drone* de *Parrot* [1]. Pour ce nouveau développement, nous nous sommes

1. http://ardrone.parrot.com

concentrés sur les bénéfices de notre approche en terme de génie logiciel au travers d'une évaluation sur la réutilisation de code existant.

8.1.1 *Cohérence*

Pour assurer la cohérence d'une application dès la conception, DiaSuite repose sur un unique langage de conception. Au lieu d'utiliser des vues indépendantes, comme les différents diagrammes UML par exemple, DiaSpec intègre des déclarations dédiées à la fois aux aspects fonctionnels et non-fonctionnels, ce qui contribue à prévenir la plupart des incohérences. Par exemple, la cohérence des déclarations de traitement des erreurs est vérifiée statiquement ainsi que nous l'avons présenté dans la Section 7.1.1. Ainsi, il peut être vérifié qu'il existe une déclaration de traitement d'erreur pour chaque composant SCC qui interagit avec une entité déclarant lever des exceptions. De plus, les automates temporisés générés à partir d'une spécification DiaSpec permettent d'effectuer de nombreuses vérifications quant à la cohérence des aspects fonctionnel et non-fonctionnels. Par exemple, la Section 7.1.2 décrit des vérifications sur la cohérence de contraintes temporelles.

Lors de l'implémentation, la cohérence des déclarations de traitement des erreurs est automatiquement préservée grâce au *framework* de programmation généré. En effet, le support généré à partir d'une spécification DiaSpec empêche les développeurs d'implémenter du code de traitement des erreurs de façon *ad-hoc* ainsi que l'illustre les différentes contraintes de programmation présentées dans la Section 6.2.2.1 et l'implémentation du contexte `SensorFailure` présentée dans la Section 6.2.2.2. De plus, les contraintes de *QoS* telles que les délais maximum d'attente (*timeout*) peuvent être vérifiées pendant l'exécution de l'application grâce à la génération de gardes au sein du *framework* de programmation [46]. Cependant, ces gardes n'assurent pas directement la cohérence de l'application lors de son exécution mais fournissent du support pour la supervision de sa *QoS*. En effet, lorsqu'une contrainte de *QoS* est violée, un événement exceptionnel est levé, facilitant l'identification des composants impliqués. Utilisé en combinaison du support de simulation et d'injection de fautes présenté dans la Section 7.2, ce support permet de faciliter la détection d'incohérences avant d'avoir à déployer intégralement l'application sur des bancs de tests.

8.1.2 *Conformité*

DiaSuite fournit du support tout au long du processus de développement dans le but d'assurer la conformité de l'application avec les exigences de haut-niveau. Nous illustrons maintenant de quelle façon ce support permet de guider la vérification de la conformité en prenant l'exemple de l'exigence **ex3.**. Cette exigence, présentée dans la Section 4.3, indique que le dysfonctionnement ou la défaillance d'un capteur doit systématiquement être signalé au pilote dans un délais de 300 ms.

Lors de la conception, cette exigence mène à la spécification d'une chaîne de supervision SCC dédiée au signalement de la défaillance des capteurs au pilote. Le support de vérification présenté dans la Section 7.1.2 permet alors de vérifier statiquement que la levée d'un événement exceptionnel par l'entité `IRU` entraîne systématiquement le déclenchement de l'action `Display` sur l'entité `NavMMI`.

Lors de l'implémentation, la génération du *framework* de programmation assure la conformité de l'application avec la spécification DiaSpec des flots de données et de contrôle [23]. Cependant, la conformité de l'application avec les aspects temporels de l'exigence **ex3.** ne peut pas être vérifiée uniquement à partir de la spécification DiaSpec et des contraintes temporelles qui y sont exprimées. En effet, la performance de l'application dépend également des spécificités de la plateforme d'exécution. Lorsque ces dernières sont connues, il est possible de les intégrer dans le modèle généré à partir de la spécification DiaSpec et d'effectuer des vérifications supplémentaires. Dans tous les cas, l'application doit tout de même être testée afin vérifier que tous les paramètres ont bien été pris en compte. Ainsi, pour faciliter la vérification de la conformité de l'application avec les contraintes temporelles exprimées au niveau de la spécification DiaSpec, le *framework* de programmation fournit des gardes qui détectent la violation de ces contraintes. De plus, le support de test présenté dans la Section 7.2 inclu la possibilité d'injecter des fautes, permettant ainsi de valider l'exigence **ex3.** même si l'entité `IRU` n'est pas encore implémentée.

8.1.3 *Un gestionnaire de vol pour drones*

Le paradigme de développement présenté à travers ce document favorise la réutilisation de code existant. Par exemple, la définition d'une taxonomie d'entités permet de faire abstraction de la diversité des entités concrètes disponibles pour interagir avec un environnement donné. Une même taxonomie peut donc

être réutilisée pour le développement d'applications différentes mais interagissant avec un environnement similaire. C'est dans le but d'évaluer cette propension à la réutilisation que nous avons adapté le gestionnaire de vol pour avion au pilotage de drones.

Nous avons donc considéré un drone composé d'un accéléromètre, de deux gyromètres et d'une caméra. Cette configuration est standard pour les drones commerciaux tels que l'*A.R. Drone* de *Parrot*. Généralement, ces drones sont contrôlables par un utilisateur via une application pour *smartphone*. Le but de l'application que nous avons développé est de rendre le drone autonome en lui faisant suivre un plan de vol de façon similaire à ce qui se fait pour les avions.

La spécification DiaSpec du gestionnaire de vol pour drones reprend en partie celles de celui pour avions. Par exemple, au niveau de la taxonomie, les entités liées à la gestion du plan de vol et celles utilisées pour enregistrer et afficher des informations ont été réutilisées (e. g., l'entité `NavMMI`). La plupart des données de navigation fournies par les capteurs du drones ont pu être converties en des informations de haut niveau utilisables par des contextes issus de l'application avionique. En effet, nous avons implémenté un système de localisation reposant sur l'utilisation de tags disposés au sol et filmés par la caméra du drone afin d'obtenir sa position absolue, similaire à celle fournie par l'entité `GPS` d'un avion. Un contexte `Position` permet de coupler la position absolue à celle fournie par l'entité `IRU` dans les deux applications. La position raffinée est ensuite comparée à celle du prochain point de passage sur le plan de vol par le contexte `IntHeading`, également réutilisé depuis l'application avionique afin d'obtenir un cap intermédiaire à atteindre. Pour contrôler le cap, les actionneurs du drone utilisent des données de vitesse angulaire au lieu du roulis. Le contexte `TargetRoll`, le contrôleur `AileronController` et l'entité `Aileron` ont donc été remplacées respectivement par un contexte `TargetVA`, un contrôleur `DroneController` et une entité `DroneActuator`.

Pour les deux applications, la même politique de traitement des erreurs a été mise en place. Au niveau de l'application, cela se traduit par l'utilisation des mêmes déclarations (e. g., **skipped catch** et **mandatory catch**) pour les composants SCC similaires. Au niveau de la supervision de l'application, la plupart des composants SCC impliqués dans la gestion des défaillances ont été réutilisés (e. g., `SensorFailure`, `DataAvailability`, `SensorController`). Tandis que nous avons considéré des exigences de haut-niveau plus souples en terme de signalisation et d'enregistrement des erreurs, nous avons pris en compte de nouvelles exigences liées à la gestion de l'alimentation. En effet, pour éviter de détériorer le drone, il est important qu'il puisse se poser au sol

de façon sûre lorsque son niveau de batterie est faible. Une nouvelle chaîne de composants SCC a donc été définie et implémentée afin de surveiller le niveau de la batterie et la position du drone, et pour le faire atterrir si nécessaire. Cette chaîne fait notamment intervenir le contexte `Position` et le contrôleur `ModeController`. Bien que réutilisé, ce dernier a tout de même du être adapté afin de définir et implémenter un nouveau contrat d'interaction responsable de l'activation du mode "Atterrissage". Cependant, cet ajout n'a pas nécessité de modifier le code existant. Au niveau de la *QoS*, nous avons réutilisé les mêmes contraintes temporelles bien qu'il soit possible de les optimiser afin d'encadrer plus précisément les tests de performance.

Finalement, une majorité des composants SCC utilisés dans l'application dédiée aux drones sont issus de l'application avionique. Au total, parmi les 7 entités déployées pour capter des informations et agir sur le drone et les 30 composants SCC qui constituent l'application du gestionnaire de vol, 5 entités et 18 composants SCC ont été réutilisés depuis l'application avionique. Ainsi, cette évaluation nous permet d'observer que notre approche favorise la réutilisation en permettant d'utiliser des composants existant au sein de nouvelles chaînes fonctionnelles (e. g., le contrôle de la vitesse angulaire du drone) et de supervision (e. g., la gestion de la batterie).

8.2 INFORMATIQUE UBIQUITAIRE

Les systèmes d'informatique ubiquitaire coordonnent une grande variété d'entités, que ce soient des périphériques ou des composants logiciels, communiquent via une grande variété de protocoles, reposent sur des imbrications de technologies pour systèmes distribués et invoquent de nombreuses et complexes APIs (*Application Programming Interfaces* ou interfaces de programmation). Il en résulte que ces systèmes doivent faire face à une quantité impressionnante d'erreurs issues de différents niveaux, depuis l'infrastructure physique du système (e. g., une pane de courant), en passant par le matériel (e. g., le dysfonctionnement d'un périphérique), le système d'exploitation (e. g., l'épuisement des ressources), le réseau (e. g., un *timeout* du protocole), l'intergiciel (e. g., l'échec d'une invocation distante), jusqu'aux APIs (e. g., des paramètres inadéquats). Détecter et traiter ces erreurs est critique pour la sûreté d'un système d'informatique ubiquitaire mais il en résulte généralement des applications dont le code est pollué, entremêlé de traitements fonctionnels et non-fonctionnels.

En appliquant notre approche pour le développement de tels systèmes, notre objectif est d'élever le niveau d'abstraction à partir duquel le traitement des erreurs est pris en compte pour per-

mettre de traiter systématiquement et rigoureusement les diverses erreurs sans polluer le code des applications. Pour en évaluer l'efficacité, nous avons utilisé cette approche pour développer trois applications faisant partie d'un projet visant à automatiser le bâtiment de 13 500 m^2 d'une école d'ingénieurs [17, 72]. Ce dernier combine six applications et implique 400 instances d'entités appartenant à 21 classes définies dans la taxonomie ainsi que 20 composants SCC pour un total de plus de 3000 lignes de codes (LOC) en Java. Après avoir présenté les 3 applications que nous avons enrichies en utilisant notre approche de traitement des erreurs, nous expliquons comment les erreurs seraient traitées au sein de ces applications sans notre approche, en utilisant directement les exceptions Java.

8.2.1 *Le système anti-incendie*

Le système anti-incendie est responsable de la détection du feu à l'aide de capteurs de fumée et de température ainsi que du contrôle de l'alarme, des arroseurs et des portes coupe-feu.

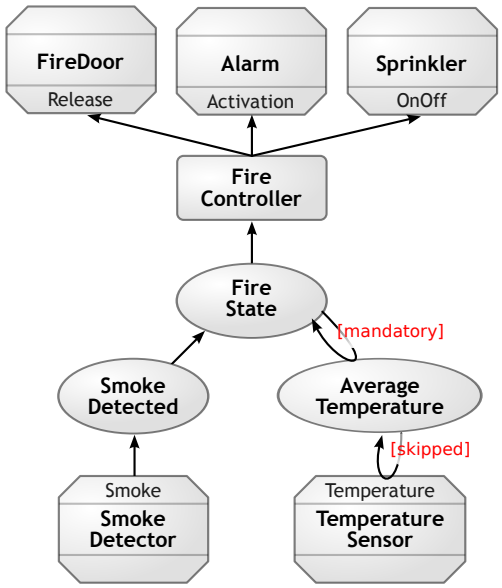

FIGURE 11: Aperçu de la spécification fonctionnelle du système anti-incendie

La Figure 11 illustre la spécification DiaSpec de cette application. Les entités `SmokeDetector` publient régulièrement des informations sur la présence de fumée au contexte `SmokeDetected`. Ce dernier agrège les informations des différents capteurs d'une même zone (e. g. une pièce d'un bâtiment), puis publie systématiquement l'information au contexte `FireState`. Celui-ci interroge alors le contexte `AverageTemperature`, qui calcule la température moyenne d'une zone donnée à partir des informations fournies

par les capteurs de température qui y sont déployés (les entités `TemperatureSensor`). En fonction des informations sur la température et la fumée, le contexte `FireState` détermine si un incendie vient d'être détecté et publie l'information. Le contrôleur `FireController` contrôle alors les portes coupe-feu (les entités `FireDoor`), les alarmes (les entités `Alarm`) et les arroseurs (les entités `Sprinkler`) en fonction de l'information communiquée par le contexte `FireState`.

TRAITEMENT DES ERREURS En cas de défaillance d'un capteur de fumée, ce dernier ne publie aucune information au niveau de l'application et le code de l'application n'est donc pas directement impacté par une erreur. En effet, le contrat d'interaction du contexte `SmokeDetected` spécifie qu'il n'est activé qu'en cas de réception d'une information d'une entité `SmokeDetector`. Le traitement des erreurs au niveau de l'application se concentre donc sur la défaillance des capteurs de température. Afin de ne pas fausser la moyenne calculée en cas de défaillance d'un capteur, il a été décidé de ne pas traiter les erreurs propagées lorsque les capteurs sont interrogés. Ainsi, le contexte `AverageTemperature` déclare ne pas traiter les erreurs provenant des entités `TemperatureSensor` grâce à la déclaration **skipped catch**. Cependant, afin d'assurer la continuité d'exécution du contexte `FireState`, qui n'utilise la température qu'en complément des informations de fumée, ce dernier doit déclarer traiter l'erreur à l'aide de la déclaration **mandatory catch**. Même si le traitement consiste en réalité à ignorer l'erreur, la déclaration oblige le développeur à le faire de façon explicite dans le code et permet d'appliquer systématiquement ce traitement particulier.

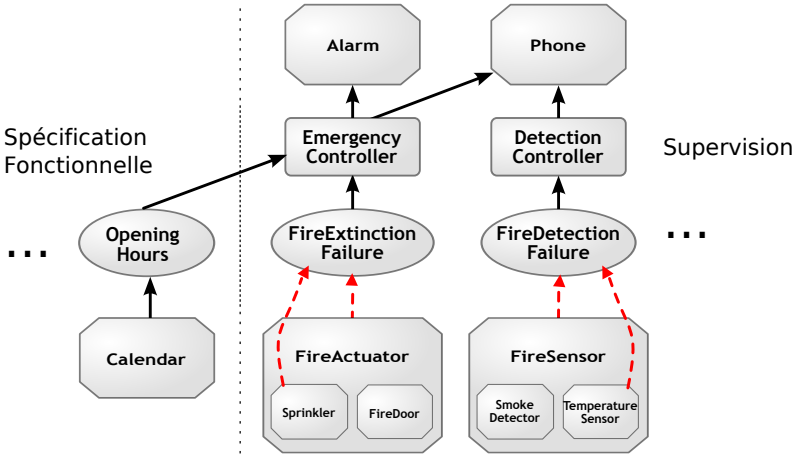

FIGURE 12: Aperçu de la spécification de la couche de supervision du système anti-incendie

La supervision du système anti-incendie repose sur la détection de deux situations critiques : l'absence de moyens de détection

valides dans une zone et l'absence de moyens de contrôle d'un incendie dans une zone. Les contextes `FireDetectionFailure` et `FireExtinctionFailure` sont en charge de détecter la première et la deuxième situation respectivement. Afin d'y parvenir, `FireDetectionFailure` (*resp.* `FireExtinctionFailure`) traite toutes les exceptions *built-in* provenant des entités de type `FireSensor` (*resp.* `FireActuator`)[2] ainsi que la plupart des exceptions spécifiques à un type d'entité particulier comme l'exception `MeasureException` (*resp.* `WaterPressureException`) levée par l'entité `TemperatureSensor` (*resp.* `Sprinkler`).

8.2.2 *Le système anti-intrusion*

Cette application est en charge de la sécurité de l'école d'ingénieurs. Elle permet de définir des zones à sécuriser et y détecte les intrusions à l'aide de capteurs de mouvement déployés dans toute l'école. Lorsqu'une intrusion est détectée, l'application déclenche des alarmes et alerte le gardien du bâtiment en affichant un message d'alerte et une image prise par des caméras sur son écran de supervision.

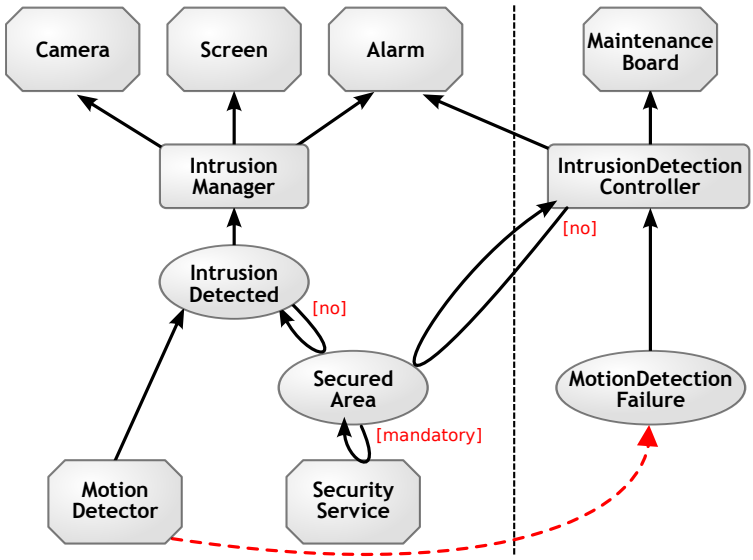

FIGURE 13: Aperçu de la spécification du système anti-intrusion

La Figure 13 illustre un extrait de la spécification de cette application. Lorsque les entités de type `MotionDetector` détectent un mouvement dans une zone de l'école d'ingénieurs, le contexte `IntrusionDetected` demande au contexte `SecuredArea` de déterminer si la zone où le mouvement a été détecté est considérée

2. La taxonomie DiaSpec permet d'utiliser l'héritage afin de raffiner la description des entités. Ainsi les entités `SmokeDetector` et `TemperatureSensor` héritent de `FireSensor` et les entités `FireDoor`, `Alarm` et `Sprinkler` héritent de `FireActuator`.

comme sécurisée par l'entité `SecurityService`. Si c'est le cas, l'information est transmise au contrôleur `IntrusionManager`. Ce dernier agit alors sur les entités `Alarm`, `Camera` et `Screen` pour signaler l'intrusion au gardien.

TRAITEMENT DES ERREURS Au niveau de l'application, seule la défaillance de l'entité `SecurityService` peut être prise en compte. En effet, de façon similaire à la défaillance d'un capteur de fumée pour l'application anti-incendie, la défaillance d'un capteur de mouvement ne requiert aucun traitement d'erreur au niveau de l'application. Les erreurs propagées par l'entité `SecurityService` peuvent, quant à elles, impacter les contextes `SecuredArea` et `IntrusionSetected` ainsi que le contrôleur `IntrusionDetectionController`. Afin d'appliquer une même stratégie de traitement des erreurs, défensive en l'occurrence, c'est le contexte `SecuredArea` qui est en charge de compenser les erreurs propagées en utilisant une valeur par défaut indiquant que la zone est sécurisée.

Pour tolérer la défaillance des capteurs de mouvement sans nuire à la capacité de détection d'intrusion, plusieurs d'entre eux doivent être déployés afin que la défaillance d'un seul n'ai pas de conséquences sur l'ensemble du système. Pour gérer cet ensemble de capteurs, une stratégie de supervision a été mise en place. Cette dernière consiste à surveiller les capteurs en permettant au contexte `MotionDetectionFailure` de traiter les événements exceptionnels provenant des entités `MotionDetector`. Lorsque le contexte détermine qu'un certain seuil de capteurs défectueux est atteint, il fournit l'information au contrôleur `IntrusionDetectionController` qui affiche un rapport sur le tableau de maintenance et déclenche une alarme si le seuil est atteint dans une zone sécurisée.

8.2.3 *Le diffuseur d'informations*

Cette application gère les informations affichées sur les écrans déployés dans l'école d'ingénieurs. En fonction des profils des étudiants à proximité des écrans, l'application sélectionne les informations à afficher comme des nouvelles provenant de flux RSS ou les emplois du temps par exemple. Pour détecter la présence d'étudiants et leur identité, des lecteurs de badges sont placés près des écrans. Les profils des étudiants sont obtenus, quant à eux, en interrogeant une base de données.

La Figure 14 illustre la spécification de la partie de l'application responsable de l'affichage des emplois du temps. L'entité `BadgeReader` informe le contexte `ProfileProximity` de la présence

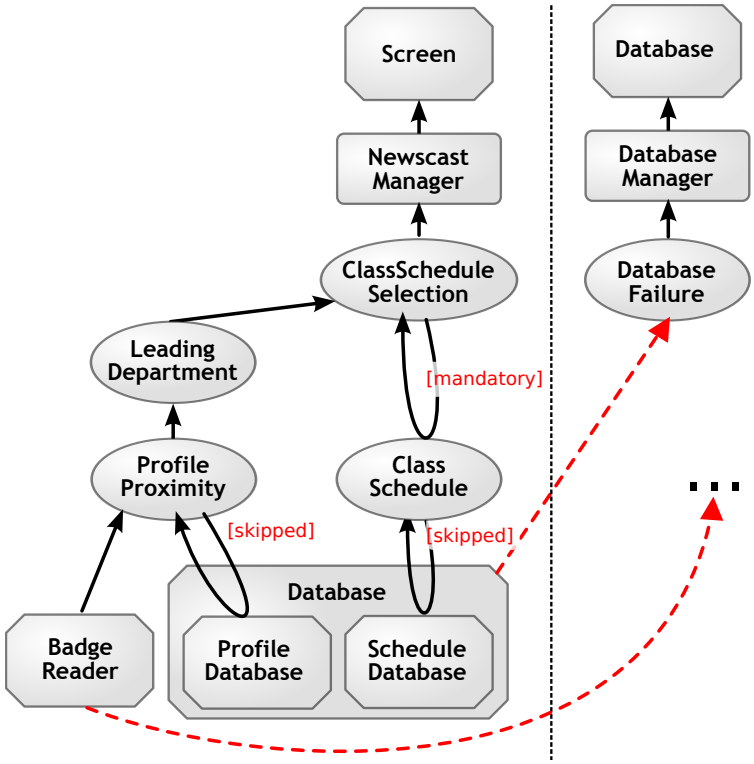

FIGURE 14: Aperçu de la spécification du diffuseur d'informations

d'un étudiant et de son identité. Le contexte utilise alors l'entité `ProfileDatabase` pour fournir le profil des étudiants proches au contexte `LeadingDepartment`. Ce dernier combine les différents profils des étudiants proches pour délivrer un profil majoritaire au contexte `ClassScheduleSelection` qui va alors pouvoir interroger le contexte `ClassSchedule` pour obtenir l'emploi du temps correspondant au profil obtenu. Cet emploi du temps, obtenu en interrogeant l'entité `ScheduleDatabase`, est transmis par le contexte `ClassScheduleSelection` au contrôleur `NewscastManager` dont le rôle est de l'afficher sur les écrans, c'est-à-dire les entités `Screen`.

TRAITEMENT DES ERREURS Les lecteurs de badges peuvent cesser de fonctionner en raison de fautes matérielles ou de défaillances du réseau. Cependant, il n'est pas nécessaire de surveiller la présence d'étudiants à proximité des écrans de façon précise : manquer un badge a peu de chance de modifier le profil majoritaire dans une zone. De la même façon, il n'est pas nécessaire de gérer les situations où un étudiant est détecté alors que son profil n'est pas accessible. Ainsi, grâce à l'utilisation de la déclaration **skipped catch**, le contexte `ProfileProximity` ne traite pas les erreurs propagées lors de l'interrogation de l'entité `ProfileDatabase`. La propagation de ces erreurs n'a alors aucun impact au niveau de l'application puisque le contexte

n'est pas utilisé directement par un autre composant SCC. En effet, les erreurs sont propagées avant que le contexte ne puisse publier une information, interrompant le flot de contrôle illustré dans la Figure 14. Les erreurs pouvant être propagées par l'autre base de données, l'entité `SchedulaDatabase`, peuvent être compensées par une valeur par défaut correspondant au dernier emploi du temps affiché. Ce traitement doit être effectué au niveau du contexte `ClassScheduleSelecetion`. Ainsi la déclaration du contexte `ClassSchedule` a été enrichie de l'annotation **skipped catch** et celle du contexte `ClassScheduleSelecetion` avec l'annotation **mandatory catch** pour guider les développeurs de ces contextes.

Bien que les erreurs soient compensées au niveau de l'application, il est important de gérer les entités défectueuses afin de réparer le système et permettre à l'application de fonctionner correctement. Par exemple, la stratégie de traitement des erreurs employée au niveau de l'application permet de tolérer l'utilisation d'une base de données corrompue. Cependant, la base de donnée doit tout de même être réparée afin d'actualiser les emplois du temps affichés par les écrans. Les bases de données proposant généralement des outils intégrés pour la réparation (e. g., *roll-back*), il est alors possible de définir une stratégie de réparation indépendante de la logique fonctionnel du diffuseur de nouvelles via la couche de supervision. Ainsi, le contexte `DatabaseFailure` traite les événements exceptionnels signalés par les entités de type `Database` et indique au contrôleur `DatabaseManager` s'il est nécessaire d'effectuer une réparation sur une base de donnée.

8.2.4 *Discussion à propos des exceptions Java*

Concentrons-nous maintenant sur les difficultés qu'un développeur pourrait rencontrer s'il devait traiter les erreurs des applications présentées uniquement à l'aide d'exceptions Java. Les composants SCC d'une boucle de supervision peuvent traiter des erreurs variées et provenant d'entités diverses au niveau du système. En Java, pour traiter de la même façon ces erreurs, le développeur devrait dupliquer le code du traitement pour chaque accès à l'une des entités, lorsqu'une exception est propagée pour signaler une erreur. Alternativement, il pourrait laisser les exceptions se propager vers le haut de la pile d'appel afin de factoriser le code. Cependant, la propagation des exceptions empêcherait alors de compenser les erreur au niveau de l'application et de reprendre une exécution normale. Par exemple, implémenter une stratégie de réparation de bases de données au niveau du contexte `ClassScheduleSelection` du diffuseur de nouvelles nécessiterait de propager également les exceptions provenant de

l'entité `ProfileDatabase` jusqu'à ce contexte alors qu'il est possible de les ignorer ou de les compenser au niveau du contexte `ProfileProximity` pour poursuivre l'exécution de l'application.

Puisque qu'assurer la continuité de l'exécution d'une application est important pour de nombreuses applications, et notamment dans le domaine de l'informatique ubiquitaire, les développeurs devraient systématiquement déclencher des stratégies "système" de traitement des erreurs au plus près des sources des exceptions puis relancer les exceptions afin que l'application puisse compenser (ou ignorer) les erreurs à son niveau. Notre modèle de traitement des erreurs automatise cette tâche, qui est susceptible d'introduire de nouvelles erreurs de développement, grâce à la génération de code glu qui publie systématiquement les exceptions au niveau du système puis qui les propage au niveau de l'application. Ainsi, le développeur bénéficie de support pour compenser ou ignorer les erreurs au niveau de l'application afin d'assurer la continuité de l'exécution, mais également pour traiter les erreurs au niveau du système.

Dans l'application du système anti-intrusion, les données fournies par les capteurs de mouvement sont uniquement publiées via des événements. En conséquence, le développeur du contexte `IntrusionDetected` ne dispose d'aucun bloc *try/catch* pour traiter des erreurs telles que la défaillance de ces capteurs. En l'absence de ces uniques constructions du langage dédié au traitement des erreurs, le développeur n'est plus incité à prendre en compte les aspects de sûreté et ne risque plus de négliger le traitement de certaines erreurs pourtant critiques pour l'application. De plus, traiter ces erreurs nécessite d'écrire du code ad-hoc pour détecter et signaler de potentielles défaillances des capteurs. En fournissant des mécanismes de détection des erreurs (e. g., *heartbeat*) qui lèvent de façon proactive des exceptions *built-in* (e. g., l'exception `UnregisterException`), notre approche assure un signalement systématique des défaillance des entités à la couche de supervision.

Une autre difficulté inhérente aux exceptions Java vient du fait qu'elles n'offrent qu'un support minimal au développeur pour savoir où et quand elles doivent être traitées. Une faute de développement répandue est de masquer une exception en l'attrapant trop tôt ou par inadvertance, avec un mauvais type (e. g., une classe parente), et de ne pas la relancer. En effet, cela peut empêcher un traitement d'exception plus haut dans la pile d'appel de compenser l'erreur qui en est à l'origine. Par exemple, dans l'application du diffuseur de nouvelles, Java n'empêche pas les développeurs de traiter les exceptions propagées par l'entité `ScheduleDatabase` au niveau du contexte `ClassSchedule` (voir la Figure 14). Si ces exceptions sont effectivement attrapées au

niveau de ce composant et ne sont pas propagées, il est alors impossible de les traiter au niveau du contexte `ClassScheduleSelection` qui peut pourtant compenser l'échec de l'obtention d'un emploi du temps de façon plus appropriée, c'est-à-dire en utilisant l'emploi du temps correspondant à un profil précédemment majoritaire. Notre approche, au contraire, laisse les concepteurs de l'application décider des endroits où les erreurs doivent être compensées et où les exceptions doivent être propagées.

Ainsi, notre modèle de traitement des erreurs permet de distinguer clairement la logique applicative du traitement des erreurs ainsi que de guider les développeurs de façon plus précise et plus flexible que ce qu'il est permis de faire uniquement avec les exceptions Java. En effet, bien que le code généré à partir des déclarations DiaSpec de traitement des erreurs utilise en partie les exceptions Java (voir la Section 6.2), il permet également aux concepteurs de maîtriser la propagation des exceptions et spécifier l'endroit où les erreurs doivent être compensées. De plus, en séparant la compensation au niveau de l'application du traitement "système" des erreurs, notre approche permet de factoriser le code de traitement des erreurs dans une couche de supervision plutôt que de le diffuser à travers toute l'application.

9

TRAVAUX CONNEXES

Dans ce chapitre, nous positionnons notre approche par rapport aux approches existantes visant à faciliter le développement de logiciels sûrs de fonctionnement. A notre connaissance il n'existe aucune approche de développement guidée par la conception permettant de guider de façon systématique le développement d'un système logiciel tout en offrant du support dédié à des aspects non-fonctionnels tels que le traitement des erreurs. Nous montrons donc les similitudes avec ces approches mais également les principales différences qui rendent notre approche innovante.

9.1 IMPLÉMENTATION

Le support offert à partir de la spécification DiaSpec d'une application peut être comparé à différentes approches visant à simplifier la programmation d'applications sûres de fonctionnement. Comme nous l'avons vu dans l'état de l'art dressé dans le Chapitre 3, ces approches vont de l'utilisation des exceptions aux intergiciels et langages dédiés en passant par les méta-objets et la programmation orientée aspect.

9.1.1 *Exceptions*

Nous avons vu dans la Section 8.2.4 que notre modèle de traitement des erreurs permet d'éviter les principaux inconvénients des exceptions Java comme la difficulté de factoriser le code dédié au traitement des erreurs ou la difficulté de savoir où et quand traiter les exceptions. En réalité, les principes de séparation entre le traitement de l'erreur au niveau de l'application et au niveau du système et de propagation d'exceptions "anonymes" sont similaires aux principes mis en avant par deux patrons de conception disponibles sur le site de dépôt *The Portland Pattern Repisitory* [38]. En effet le patron *ExceptionReporter* propose de découpler la gestion des exceptions du flot de contrôle et permet à une exception d'être traitée simultanément à plusieurs endroits. Le code généré pour publier des événements exceptionnels au niveau de la couche de supervision est une application de ce patron dans notre approche. Le patron *BottomPropagation*

consiste à utiliser des exceptions dénuées d'information. L'exception `ApplicationLevelException`, utilisée au sein du *framework* de programmation généré à partir d'une spécification DiaSpec pour propager de façon transparente les exceptions et déclencher le traitement d'erreur au niveau de l'application lorsque cela est nécessaire, s'inspire du même principe. Ce dernier est également appliqué par les *maybe monads* du langage de programmation Haskell.

Non seulement notre approche s'inspire et combine ces patrons de conceptions dédiés aux exceptions, mais elle automatise également leur implémentation au sein du *framework* de programmation généré afin d'offrir un support de haut niveau aux développeurs.

9.1.2 *Méta-objets et programmation orientée aspect*

Tandis que certaines approches comme le projet FRIENDS proposent d'implémenter des politiques de tolérance aux fautes via des méta-objets [39, 63], d'autres approches proposent d'utiliser la programmation orientée aspect pour implémenter le code de traitement des erreurs séparément de celui de l'application [20, 66]. Ces approches ont en commun d'appliquer le principe de séparation des aspects fonctionnels et non-fonctionnels au traitement des erreurs. En permettant de définir des composants SCC dédiés au traitement des erreurs dans une couche de supervision, notre approche applique également ce principe. En fait, il aurait été également possible d'utiliser des méta-objets ou la programmation par aspects dans DiaSuite, au sein du *framework* de programmation qui guide le développement.

Bien que, contrairement à FRIENDS, nous n'offrons pas de support dédié à la tolérance aux fautes, cette dernière peut être implémentée au travers des composants SCC qui constituent la couche de supervision. Cependant, des travaux sont en cours pour étendre le langage de conception DiaSpec afin de permettre la spécification de contraintes de tolérance aux fautes. A partir de ces spécifications, des contraintes de programmation sont générées pour contraindre les développeurs à sélectionner et déployer des mécanismes de tolérance aux fautes adéquats. Ces mécanismes sont fournis par le biais de bibliothèques de programmation et implémentent des patrons de conception dédiés [90].

9.1.3 *Intergiciels et langages dédiés*

De nombreux intergiciels et langages dédiés fournissent du support pour améliorer la fiabilité d'applications dans le domaine

des systèmes distribués [31, 34, 81] et plus particulièrement dans celui de l'informatique ubiquitaire [25, 75, 77, 80]. Bien que ces approches ne permettent pas d'appréhender la sûreté de fonctionnement dès la phases de conception, elles offrent une grande variété de support au niveau de l'implémentation, notamment en ce qui concerne la tolérance aux fautes. Les travaux en cours pour l'intégration de mécanismes de tolérance aux faute dans DiaSuite s'inspirent également de ce qui est proposé par ces approches, et particulièrement de celles basées sur les Services Web [34, 81]. En effet, ces approches proposent de décrire à l'aide de langages dédiés les mécanismes à mettre en oeuvre pour améliorer la sûreté de services web particuliers. En utilisant de tels langages pour décrire plus précisément, en phase de conception, les contraintes de tolérance aux fautes, il est possible de générer du code permettant d'appliquer de façon transparente des mécanismes de tolérance aux fautes au niveau de l'implémentation. Par exemple, la déclaration **mandatory catch** pourrait être raffinée afin de spécifier l'utilisation d'un mécanisme de réplication particulier qui serait systématiquement appliqué grâce à la génération d'une continuation dédiée plutôt que de demander au développeur de l'implémenter.

Dans le domaine de l'avionique, les langages synchrones tels que Lustre permettent de décrire précisément un système et de vérifier certaines propriétés directement à partir de cette description [53]. En particulier, les descriptions étant déterministes à la fois au niveau fonctionnel et au niveau temporel, il est possible de vérifier le comportement d'un programme et de déterminer, par exemple, l'état dans lequel il se trouve à un instant donné. Bien que le support généré pour la vérification statique d'une spécification DiaSpec utilise le formalisme des automates temporisés d'UPPAAL, il permet également de vérifier formellement un certains nombre de propriétés fonctionnelles et temporelles malgré l'absence de contraintes liées au déterminisme. Au contraire, notre approche permet de s'abstraire de ces contraintes afin de permettre d'effectuer des vérifications dès les premières étapes du développement et de prendre en compte les composants sur étagère. Les langages synchrones s'avèrent tout de même complémentaires à cette approche en permettant, par exemple, de vérifier précisément le comportement et l'état interne des composants SCC.

Ainsi, il existe une grande variété d'approches offrant du support pour l'implémentation et la vérification d'applications logicielles sûres de fonctionnement. Cependant, aucune de ces approches ne permet de guider chaque étape du développement. Notre approche est donc complémentaire : elle s'inspire de ces approches pour fournir du support d'implémentation et de véri-

fication mais elle permet également de guider les développeurs de façon systématique en s'appuyant sur un cadre de conception.

9.2 CONCEPTION

Parmi les approches dirigées par la conception, certaines permettent de prendre en compte des aspects non-fonctionnels tels que la sécurité [68] ou le temps [18]. Cependant, peu d'approches prennent en comptes des aspects de sûreté de fonctionnement comme la tolérance aux fautes ou le traitement des erreurs. Bien qu'*Aereal* [43] ou l'*Error Model Annex* d'AADL [94] permettent de spécifier et d'analyser le flot de propagation des erreurs, ces approches restent basées sur des langages de conception génériques et n'offrent qu'un cadre limité pour guider le développement d'applications sûres. Au contraire, notre approche s'appuie sur un paradigme de conception, le paradigme SCC, ce qui permet d'offrir du support de développement dédié tout au long du processus de développement. A notre connaissance, seul le patron de conception IFTC pour le style architectural C2 offre un cadre de conception dédié au traitement des erreurs [28], mais il souffre d'un manque d'outils permettant de guider le développement et d'assurer, par exemple, la conformité entre la conception et l'implémentation.

Pour assurer la conformité entre la conception d'une application et son code, des approches outillées font intervenir différentes techniques comme l'ajout de constructions architecturales à un langage de programmation [4, 82, 88], la programmation par aspect [93] ou la génération de code [99]. Cette dernière approche est d'ailleurs proche de la façon dont est généré le code du *framework* de programmation à partir d'une spécification DiaSpec mais elle diffère notamment dans la façon dont est utilisé le langage cible pour guider le développement. En effet, elle utilise la délégation comme lien entre le code généré et le code du développeur, là où notre approche utilise l'héritage de classes abstraites pour le guider de façon encore plus précise et systématique. Ces approches reposent également sur des notions de conception généraliste (e. g., composants, ports) et n'offrent donc qu'un support limité pour guider l'implémentation de la logique applicative. De plus elles ne prennent pas en compte les aspects non fonctionnels tels que le traitement des erreurs.

Dans le domaine de l'avionique, *Ocarina* [98] et SCADE [32] sont des approches dédiées au développement de systèmes temps-réels qui permettent de générer une partie du code de l'application. Elles se concentrent en effet sur l'intégration de composants logiciels sur des éléments matériels et sur le comportement de l'application à l'exécution. Cependant elles n'offrent que peu de

support pour guider le développement de la logique applicative ou des solutions visant à améliorer la sûreté d'une application. Au contraire, notre approche s'abstrait des problématiques des plateformes d'exécution pour se concentrer sur les flots de données, de contrôle et de propagation des erreurs au sein d'une application ainsi que la supervision des aspects non-fonctionnels. Ces approches sont donc complémentaires à la notre et peuvent être employées lors du déploiement et de l'intégration d'une application SCC afin de maîtriser sont comportement lors de l'exécution.

En offrant un cadre de conception basé sur le paradigme SCC, l'approche présentée dans cette thèse permet non-seulement de décrire les aspects fonctionnels et non-fonctionnels d'une application sûre de fonctionnement mais également de guider son développement de façon systématique. Bien qu'il existe des approches similaires au support fourni à chaque étape du développement ou que ce dernier puisse être complété par une approche plus adaptée à un domaine particulier, notre approche se démarque en offrant du support dédié pour guider l'implémentation et la vérification tout en assurant la conformité avec la conception.

10

CONCLUSION

Dans cette thèse, nous avons présenté une nouvelle approche de développement pour les applications sûres de fonctionnement. Après avoir dressé un état de l'art des différentes approches existantes pour faciliter le développement d'applications sûres, nous avons constaté que la majorité de ces approches n'offrent qu'un support limité en se concentrant sur une seule étape du développement. Au contraire, les approches de développement dirigées par la conception s'avèrent particulièrement adaptées au développement d'applications sûres. En effet, ces approches permettent de guider à la fois l'implémentation et la vérification des applications en s'appuyant sur la conception. Elles permettent également de prendre en compte les contraintes de traçabilité en proposant différentes solutions à la problématique de la conformité entre les descriptions haut-niveau et le code. Cependant ces approches restent trop généralistes ou, au contraire, trop proches du code et souffrent d'un manque d'outils dédiés pour le développement d'applications sûres.

Nous nous sommes donc appuyés sur une approche existante de développement dirigée par la conception, nommée DiaSuite, et nous l'avons étendue pour offrir du support dédié au développement d'applications sûres de fonctionnement. En effet, DiaSuite est une approche outillée qui guide les développeurs en générant un *framework* de programmation dédié à la spécification d'une application réalisée à l'aide d'un langage de conception nommé DiaSpec et basé sur le paradigme SCC. Grâce à l'utilisation de ce paradigme, DiaSuite permet d'offrir un support de haut niveau aux développeurs tout en préservant la conformité entre la conception et l'implémentation.

Afin de pouvoir prendre en compte les exigences non-fonctionnelles des applications sûres de fonctionnement dès la phase de conception, nous avons étendu DiaSpec pour permettre aux concepteurs de spécifier des stratégies de traitement des erreurs au niveau de l'application mais également au niveau du système. Ce modèle innovant de traitement des erreurs permet de spécifier à quel endroit les développeurs doivent compenser une erreur pour assurer la continuité d'exécution de l'application. Il permet également de spécifier des composants SCC dédiés au traitement des exceptions qui signalent ces erreurs. En utilisant ces composants, les concepteurs de l'application peuvent alors concevoir des stratégies de supervision indépendamment de la

spécification fonctionnelle. Grâce aux extensions du langage, les concepteurs bénéficient d'un cadre conceptuel approprié pour la conception d'applications sûres. Nous avons illustré ce support de conception en présentant les différentes étapes de la conception d'un gestionnaire de vol. Par la suite, nous avons montré qu'en permettant de spécifier à la fois les aspects fonctionnels et non-fonctionnels d'une application, notre extension de DiaSpec permet de raisonner sur sa sûreté mais également de guider de façon rigoureuse son développement.

En effet, le *framework* de programmation généré partir de la spécification DiaSpec d'une application permet de guider l'implémentation d'une application. Le compilateur du langage a donc été également étendu pour permettre la génération de support dédié aux extensions portant sur sur les aspects non-fonctionnels. En particulier, nous avons illustré comment les développeurs sont contraints de respecter les déclarations de traitement des erreurs par le *framework* de programmation généré à partir de la spécification du gestionnaire de vol afin de préserver la conformité entre conception et implémentation. En plus de contraindre les développeurs, le *framework* les guide en offrant du support de programmation de haut-niveau pour l'implémentation du traitement des erreurs. Ce support permet en effet de faciliter l'application de notre modèle et ainsi de séparer les aspects fonctionnels et non-fonctionnels, limitant l'intrusion et la diffusion de code de traitement des erreurs dans la logique applicative. Ces bénéfices sont notamment illustrés au travers de l'évaluation de notre approche dans le domaine de l'informatique ubiquitaire.

Enfin, pour permettre de vérifier qu'une application est bien conforme à ses exigences de haut-niveau, tant fonctionnelles que non fonctionnelles, nous avons étendu DiaSuite afin de fournir du support pour la vérification statique d'une spécification DiaSpec et pour tester son implémentation. Tandis que le support pour la vérification statique consiste en la génération d'un réseau d'automates temporisés permettant de vérifier formellement des propriétés fonctionnelles et non fonctionnelles à l'aide du *model checker* de UPPAAL, le support de test repose sur la simulation et l'injection de fautes. Ces supports permettent ainsi de faciliter la vérification de la cohérence de l'application et de la conformité avec les exigences de haut-niveau ainsi que l'illustre les vérifications effectuées sur le gestionnaire de vol.

Ainsi, à travers le développement et la vérification d'applications sûres de fonctionnement dans les domaines de l'avionique et de l'informatique ubiquitaire, nous avons illustré les bénéfices d'une approche de développement dirigée par la conception et outillée, prenant en compte à la fois les aspects fonctionnels et non-fonctionnels. En effet, en offrant du support de dévelop-

pement dédié pour chacune des étapes, notre version étendue de DiaSuite guide de façon rigoureuse et systématique le développent des applications sûres de fonctionnement.

TRAVAUX EN COURS ET FUTURS

La présentation de notre approche de développement et son application aux domaines de l'avionique et de l'informatique ubiquitaire illustrent les bénéfices des approche dirigées par la conception pour le développement d'application sûres de fonctionnement. En facilitant le développement de ces applications, ces travaux ouvrent de nouvelles perspectives de développement dans de nouveaux domaines d'application.

TOLÉRANCE AUX FAUTES L'approche proposée dans cette thèse permet notamment d'appréhender le traitement des erreurs dès la phase de conception. Cependant, de nombreuses politiques de traitement des erreurs correspondent en réalité à des stratégies bien connues et définies de la tolérance aux fautes [10]. Une perspective immédiate de poursuite de nos travaux est d'intégrer du support pour la tolérance aux fautes dans DiaSuite. Dans le cadre de nos travaux dans le domaine de l'informatique ubiquitaire et sur l'intégration de la tolérance aux fautes dans le processus de développement, nous nous intéressons aux systèmes résilients. Ces systèmes ont la particularité de tolérer les fautes malgré des changements dans leur contexte d'exécution. Afin de faciliter le développement de tels système, l'approche que nous étudions est de garantir la conformité de l'application avec des exigences de sûreté malgré les changements en utilisants des patrons de tolérances aux fautes adaptables et en reposant sur une plateforme d'exécution à composants permettant d'effectuer des reconfigurations à chaud. Dans ce but, DiaSpec est étendu pour permettre la spécification de contraintes de tolérance aux fautes qui permettent de générer des contraintes de programmation pour forcer les développeurs à sélectionner et déployer des mécanismes de tolérance aux fautes adéquat. Ces mécanismes sont fournis par le biais de bibliothèques de programmation pour la plateforme à composants FraSCAti [83] et implémentent des patrons de tolérance aux fautes adaptables [90]. Nous envisageons également de généraliser cette approche en reposant sur d'autres technologies, telles que les Services Web, à l'image des approches proposées pour améliorer la sûreté des applications en générant des Services Web tolérants aux fautes à partir de descriptions effectuées dans un langage dédié [34, 81].

PRISE EN COMPTE DES ASPECTS HUMAINS Que cela soit dans le domaine de l'informatique ubiquitaire ou dans celui de l'avionique, une part importante de la sûreté d'un système réside dans le bon déroulement des interactions entre les acteurs humains (utilisateurs, opérateurs de maintenance, *etc.*) et le système [10]. Par exemple, en avionique, le domaine de la sécurité-inocuité considère le rôle des différents acteurs d'un système, comme le pilote d'un avion, afin d'éviter de potentielles catastrophes ou événements critiques. Cependant, les techniques existantes dans le domaine des interactions homme-machine sont généralement isolées du processus outillé de développement et ne servent qu'à guider la rédaction des exigences de haut-niveau du système et principalement de l'interface utilisateur (IU). Ce manque d'intégration est d'autant plus critique que les systèmes logiciels interviennent de plus en plus dans notre environnement quotidien et que les interactions homme-machine deviennent de plus en plus complexes, notamment en raison de leurs conséquences pour la santé des utilisateurs, par exemple lorsque l'informatique ubiquitaire se tourne vers l'aide à la personne. Pour pallier à ce manque d'intégration, des travaux sont en cours pour intégrer un modèle des utilisateurs et la conception des IUs dans la méthodologie de développement d'une application avec DiaSuite [11], ouvrant de nouvelles perspectives pour le développement d'applications dans le domaine de l'aide à la personne mais également dans celui de la sécurité-innocuité.

EVALUATION EXPÉRIMENTALE ORIENTÉE UTILISATEURS Les approches de développement dirigées par la conception ont été intensivement étudiées dans les dernières décennies comme l'illustre la littérature dans le domaine [12, 87, 92]. Concevoir un logiciel avant de l'implémenter est souvent considéré comme la source d'importants bénéfices, notamment en termes de qualité du logiciel [12]. Bien que l'approche présentée dans cette thèse contribue à illustrer certains de ces bénéfices, comme la traçabilité d'exigences de haut niveau, et même s'il existe quelques cas d'études analysant des architectures logicielles [6, 96], il n'existe pas, à notre connaissance, d'étude expérimentale démontrant que ce type d'approche de développement améliore la qualité d'un logiciel [40]. C'est pourquoi nous avons commencé à réaliser une évaluation expérimentale de DiaSuite visant à en mesurer les bénéfices en terme de productivité des développeurs et d'évolutivité des applications développées [35]. Nous envisageons également de nous baser sur le protocole expérimental de cette évaluation pour conduire d'autres études visant à étudier l'impact de notre modèle d'intégration des aspects non-fonctionnels sur la productivité des développeurs et leur propension à commettre des erreurs.

BIBLIOGRAPHIE

[1] Gregory Abowd, Robert Allen, et David Garlan. Using style to understand descriptions of software architecture. *ACM SIGSOFT Software Engineering Notes*, 18(5) :9–20, 1993.

[2] Gregory Abowd, Robert Allen, et David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering Methodology*, 4(4) : 319–364, 1995.

[3] Gul Agha. *Actors : a Model of Concurrent Computation in Distributed Systems*. MIT, 1986.

[4] Jonathan Aldrich, Craig Chambers, et David Notkin. ArchJava : connecting software architecture to implementation. Dans *ICSE'02 : Proceedings of the 24rd International Conference on Software Engineering*, pages 187–197. 2002.

[5] Jonathan Aldrich, Craig Chambers, et David Notkin. Architectural reasoning in archjava. Dans *ECOOP'02 : Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 334–367. 2002.

[6] Muhammad. Ali Babar et Barbara Kitchenham. Assessment of a framework for comparing software architecture analysis methods. Dans *EASE'07 : Proceedings of the 11th International Conference on Evaluation and Assessment in Software Engineering*. Citeseer, 2007.

[7] Robert Allen et David Garlan. A formal basis for architectural connection. *ACM Tansactions on Software Engineering and Methodology*, 6(3) :213–249, 1997.

[8] ARP-4761. ARP-4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment (SAE), 1996. url : `http://standards.sae.org/arp4761`.

[9] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11 (12) :1491 – 1501, 1985.

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

[11] Emilie Balland, Charles Consel, Bernard N'Kaoua, et Hélène Sauzéon. A case for human-driven software development. Dans *ICSE'13 : Proceedings of the 35th International Conference on Software Engineering*, 2013. to appear.

[12] Len Bass, Paul Clements, et Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman, 2 edition, 2003.

[13] Gerd Behrmann, Alexandre David, et Kim G. Larsen. A tutorial on UPPAAL. *Formal methods for the design of real-time systems*, pages 33–35, 2004.

[14] Jon Bentley. Programming pearls : little languages. *ACM Communications*, 29(8) :711–721, 1986.

[15] Jean Bézivin et Olivier Gerbé. Towards a precise definition of the omg/mda framework. Dans *ASE'01 : Proceedings of the 16th International Conference on Automated Software Engineering*, pages 273–. 2001.

[16] Julien Bruneau et Charles Consel. DiaSim : a simulator for pervasive computing applications. *Software : Practice and Experience*, 2012.

[17] Julien Bruneau, Wilfried Jouve, et Charles Consel. DiaSim : A Parameterized Simulator for Pervasive Computing Applications. Dans *Mobiquitous'09 : Proceedings of the 6th International Conference on Mobile and Ubiquitous Systems : Computing, Networking and Services*. 2009.

[18] Sven Burmester, Matthias Tichy, et Holger Giese. Modeling reconfigurable mechatronic systems with mechatronic UML. Dans *MDAFA'04 : Proceedings of Model-Driven Architecture : Foundations and Applications*, pages 155–169. Citeseer, 2004.

[19] 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 *ICSE'05 : Proceedings of the 27th International Conference on Software Engineering*, pages 670–671. 2005.

[20] Nelio Cacho, Fernando Castor Filho, Alessandro F. Garcia, et Eduardo Figueiredo. EJFlow : Taming exceptional control flows in aspect-oriented programming. Dans *AOSD'08 : Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, pages 72–83. 2008.

[21] Damien Cassou. *Développement logiciel orienté paradigme de conception : la programmation dirigée par la spécification*. PhD thesis, Université de Bordeaux, France, 2011.

[22] Damien Cassou, Benjamin Bertran, Nicolas Loriant, et Charles Consel. A generative programming approach to developing pervasive computing systems. Dans *GPCE'09 : Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, pages 137–146. 2009.

[23] Damien Cassou, Emilie Balland, Charles Consel, et Julia Lawall. Leveraging software architectures to guide and verify the development of Sense/Compute/Control applications. Dans *ICSE'11 : Proceedings of the 33rd International Conference on Software Engineering*, pages 431–440. 2011.

[24] Damien Cassou, Julien Bruneau, Charles Consel, et Emilie Balland. Towards a Tool-based Development Methodology for Pervasive Computing Applications. *IEEE Transactions on Software Engineering*, 38(6) :1445–1463, 2012.

[25] Shiva Chetan, Anand Ranganathan, et Roy H. Campbell. Towards Fault Tolerant Pervasive Computing. *IEEE Technology and Society Magazine*, 24(1) :38–44, 2005.

[26] Paul Clements et Mary Shaw. "the golden age of software architecture" revisited. *IEEE Software*, 26(4) :70–72, 2009.

[27] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, et Sanjiva Weerawarana. Unraveling the Web Services web : An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2) :86–93, 2002.

[28] Paulo Asterio de Castro Guerra, Cecília Mary Fischer Rubira, et R. de Lemos. A fault-tolerant software architecture for component-based systems. *Architecting dependable systems*, pages 129–149, 2003.

[29] Vincenzo De Florio, Geert Deconinck, et Rudy Lauwereins. The EFTOS voting farm : a software tool for fault masking in message passing parallel environments. Dans *Proceedings of the 24th Euromicro Conference*, pages 379–386. IEEE, 1998.

[30] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, et Wolfgang De Meuter. Ambient-oriented programming in AmbientTalk. Dans *ECOOP'06 : Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 230–254, Berlin, Heidelberg, 2006.

[31] Brian Demsky et Alokika Dash. Bristlecone : A language for robust software systems. Dans *ECOOP'08 : Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 490–515. 2008.

[32] Bernard Dion. Correct-by-construction methods for the development of safety-critical Applications. *SAE Transactions*, 113(7) :242–249, 2004.

[33] DO-178B. DO-178B, Software considerations in airborne systems and equipment certification (RTCA, Inc.), 1992. url : `http://www.rtca.org/`.

[34] John Edstrom et Eli Tilevich. Reusable and extensible fault tolerance for RESTful applications. Dans *TrustCom'12 : Proceedings of the 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 737–744. IEEE, 2012.

[35] Quentin Enard, Christine Louberry, Charles Consel, et Xavier Blanc. An experimental study of a design-driven, tool-based development approach. Dans *USER'12 : User Evaluation for Software Engineering Researchers*, 2012.

[36] 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'13 : Proceeding of the International Conference on Pervasive and Embedded Computing and Communication Systems*, 2013. A paraître.

[37] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, et Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computer Survey*, 35(2) :114–131, 2003.

[38] ExceptionPatterns. Portland pattern repository, 1995. url : `http://c2.com/cgi/wiki?ExceptionPatterns`.

[39] Jean-Charles Fabre et Tanguy Pérennou. FRIENDS - a flexible architecture for implementing fault tolerant and secure distributed applications. Dans *EDCC'96 : Proceedings of the 2nd European Dependable Computing Conference on Dependable Computing*, pages 3–20. 1996.

[40] Davide Falessi, Muhammad Babar, Giovanni Cantone, et Philippe Kruchten. Applying empirical software engineering to software architecture : Challenges and lessons learned. *Empirical Software Engineering*, 15 :250–276, 2010.

[41] Peter H. Feiler. The Architecture Analysis & Design Language (AADL) : An introduction. Technical report, DTIC Document, 2006.

[42] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[43] Fernando Castor Filho, Patrick Henrique da S. Brito, et Cecília Mary Fischer Rubira. Specification of exception flow in software architectures. *Journal of Systems and Software*, 79 (10) :1397–1418, 2006.

[44] Vincenzo De Florio et Chris Blondia. A survey of linguistic structures for application-level fault tolerance. *ACM Computer Survey*, 40(2) :6 :1–6 :37, 2008.

[45] Alessandro F. Garcia, Cecília Mary Fischer Rubira, Alexander Romanovsky, et Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2) :197 – 222, 2001.

[46] Stéphanie Gatti, Emilie Balland, et Charles Consel. A Stepwise Approach for Integrating QoS throughout Software Development. Dans *FASE'11 : Proceedings of the 14th European Conference on Fundamental Approaches to Software Engineering*, pages 217–231, 2011.

[47] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1) : 80–112, 1985.

[48] Peter L. Goddard. Software FMEA techniques. Dans *Proceedings of the Reliability and Maintainability Symposium*, pages 118 –123, 2000.

[49] John B. Goodenough. Exception handling : issues and a proposed notation. *ACM Communications*, 18(12) :683–696, 1975.

[50] William Grosso. *Java RMI*. Java Series. O'Reilly Media, 2001.

[51] Object Management Group. The common object request broker : Architecture and specification, 1995.

[52] Object Management Group. MDA, Model Driven Architecture, 2000. url : `http://www.omg.org/mda/`.

[53] 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.

[54] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, et Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2) :193–244, 1994.

[55] Robert Hirschfeld. AspectS - aspect-oriented programming with Squeak. Dans Mehmet Aksit, Mira Mezini, et Rainer Unland, editors, *Objects, Components, Architectures, Services,*

and Applications for a Networked World, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer Berlin / Heidelberg, 2003.

[56] Yennun Huang, Chandra M.R. Kintala, Lawrence Bernstein, et Yi-Min Wang. Components for software fault tolerance and rejuvenation. *AT&T technical journal*, 75(2) :29–37, 1996.

[57] 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*, 7 :42 :1–42 :25, 2008.

[58] Valérie Issarny et Apostolos Zarras. Software architecture and dependability. *Formal Methods for Software Architectures*, pages 259–285, 2003.

[59] Henner Jakob. *Towards securing pervasive computing systems by design : a language approach*. These, Université Sciences et Technologies - Bordeaux I, 2011.

[60] Henner Jakob, Nicolas Loriant, et Charles Consel. An aspect-oriented approach to securing distributed systems. Dans *ICPS'09 : Proceedings of the 2009 international conference on Pervasive services*, pages 21–30. 2009.

[61] Henner Jakob, Charles Consel, et Nicolas Loriant. Architecturing Conflict Handling of Pervasive Computing Resources. Dans *DAIS'11 : Proceedings of the 11th International Conference on Distributed Applications and Interoperable Systems*, 2011.

[62] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, et John Irwin. Aspect-oriented programming. Dans *ECOOP'97 : Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[63] Marc-Olivier Killijian et Jean-Charles Fabre. Implementing a reflective fault-tolerant CORBA system. Dans *SRDS'00 : Proceedings of the 19th Symposium on Reliable Distributed Systems*, pages 154 –163. 2000.

[64] Marc-Olivier Killijian, Jean-Charles Fabre, Juan-Carlos Ruiz-Garcia, et Shigeru Chiba. A metaobject protocol for fault-tolerant CORBA applications. Dans *SRDS'98 : Proceedings of the 17th Symposium on Reliable Distributed Systems*, pages 127 –134. 1998.

[65] Jean-Claude Laprie. Dependable computing : Concepts, limits, challenges. Dans *FTCS'25, Proceedings of the 25th International Symposium on Fault-Tolerant Computing - Special Issue*, pages 42–54, 1995.

[66] Martin Lippert et Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. Dans *ICSE'00 : Proceedings of the 22nd International Conference on Software Engineering*, pages 418–427. 2000.

[67] Bev Littlewood et Lorenzo Strigini. Software reliability and dependability : a roadmap. Dans *ICSE'00 : Proceedings of the Conference on The Future of Software Engineering*, pages 175–188. 2000.

[68] Torsten Lodderstedt, David Basin, et Jürgen Doser. SecureUML : A UML-based modeling language for model-driven security. *«UML» : The Unified Modeling Language*, pages 426–441, 2002.

[69] David C. Luckham et James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9) :717–734, 1995.

[70] Pattie Maes. Concepts and experiments in computational reflection. Dans *OOPSLA '87 : Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155, New York, NY, USA, 1987.

[71] Jeff Magee et Jeff Kramer. Dynamic structure in software architectures. *SIGSOFT'96 : Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 21 (6) :3–14, 1996.

[72] Julien Mercadal, Quentin Enard, Charles Consel, et Nicolas Loriant. A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing. Dans *OOPSLA'10 : Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, pages 47–61, 2010.

[73] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, et Gilles Muller. Devil : an IDL for hardware programming. Dans *OSDI'00 : Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 2–2. 2000.

[74] Steven P. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. Dans *FMSP'98 : Proceedings of the second workshop on Formal methods in software practice*, pages 44–53. 1998.

[75] Stijn Mostinckx, Jessie Dedecker, Elisa Gonzalez Boix, Tom Van Cutsem, et Wolfgang De Meuter. Ambient-oriented exception handling. Dans *Advanced Topics in Exception Handling Techniques*, pages 141–160, 2006.

[76] Alexander R. Perry. The FlightGear Flight Simulator. Dans *Proceedings of the USENIX Annual Technical Conference*, 2004.

[77] Shankar R. Ponnekanti, Brad Johanson, Emre Kiciman, et Armando Fox. Portability, extensibility and robustness in iROS. Dans *PerCom'03 : Proceedings of the First International Conference on Pervasive Computing and Communications*, page 11. 2003.

[78] Brian Randell. System structure for software fault tolerance. *ACM SIGPLAN Notices*, 10(6) :437–449, 1975.

[79] Brian Randell et Jie Xu. The evolution of the recovery block concept. Dans *Software Fault Tolerance*, pages 1–22. 1994.

[80] Grimm. Robert. One.world : Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3) : 22–30, 2004.

[81] Nicolas Salatgé. *Conception et mise en oeuvre d'une plate-forme pour la sûreté de fonctionnement des Services Web*. PhD thesis, Institut National Polytechnique de Toulouse, France, 2006.

[82] João Costa Seco et Luís Caires. A basic model of typed components. Dans *ECOOP'00 : Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128. 2000.

[83] 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.

[84] Mary Shaw. Beyond objects : A software design paradigm based on process control. *ACM SIGSOFT Software Engineering Notes*, 20 :27–38, 1995.

[85] Mary Shaw et David Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[86] Quan Z. Sheng et Boualem Benatallah. ContextUML : A UML-based modeling language for model-driven development of context-aware Web Services. Dans *ICMB'05 : Proceedings of the International Conference on Mobile Business*, pages 206–212. IEEE, 2005.

[87] Ian Sommerville. *Software Engineering (6)*. Pearson Studium, 2001.

[88] Vugranam C. Sreedhar. Mixin'up components. Dans *ICSE'02 : Proceedings of the 24th International Conference on Software Engineering*, pages 198–207. 2002.

[89] Thomas Stahl, Markus Voelter, et Krzysztof Czarnecki. *Model-Driven Software Development : Technology, Engineering, Management*. John Wiley & Sons, 2006.

[90] Miruna Stoicescu, Jean-Charles Fabre, et Matthieu Roy. From design for adaptation to component-based resilient computing. Dans *PRDC'12 : Proceedings of the 18th Pacific Rim International Symposium on Dependable Computing*, pages 1–10. 2012.

[91] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., et Jason E. Robbins. A component- and message-based architectural style for GUI software. Dans *ICSE'95 : Proceedings of the 17th International Conference on Software Engineering*, pages 295–304. 1995.

[92] Richard N. Taylor, Nenad Medvidovic, et Eric M. Dashofy. *Software Architecture : Foundations, Theory, and Practice*. Wiley, 2009.

[93] Naoyasu Ubayashi, Jun Nomura, et Tetsuo Tamai. Archface : a contract place where architectural design and code meet together. Dans *ICSE'10 : Proceedings of the 32nd International Conference on Software Engineering*, pages 75–84. 2010.

[94] Steve Vestal. An overview of the Architecture Analysis & Design Language (AADL) error model annex. Dans *AADL Workshop*, 2005.

[95] W3C. Simple Object Access Protocol (SOAP) specification, 2007. url : `http://www.w3.org/TR/soap/`.

[96] Michel Wermelinger, Yijun Yu, Angela Lozano, et Andrea Capiluppi. Assessing architectural evolution : a case study. *Empirical Software Engineering*, 16 :623–666, 2011.

[97] James Windsor et Kjeld Hjortnaes. Time and Space Partitioning in Spacecraft Avionics. Dans *SMC-IT'09 : Proceedings of the 3rd International Conference on Space Mission Challenges for Information Technology*, pages 13–20. 2009.

[98] Bechir Zalila, Irfan Hamid, Jerome Hugues, et Laurent Pautet. Generating distributed high integrity applications from their architectural description. Dans *Ada-Europe'07 : Proceedings of the 12th International Conference on Reliable Software Technologies*, pages 155–167. 2007.

[99] Yongjie Zheng et Richard N. Taylor. Enhancing architecture-implementation conformance with change management and support for behavioral mapping. Dans *ICSE'12 : Proceedings of the 34th International Conference on Software Engineering*, pages 628–638. 2012.