Tolérance aux fautes dans les systèmes autonomes

Benjamin Lussier

N° d'ordre : 2469

THÈSE

présentée pour obtenir

LE TITRE DE DOCTEUR
DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

École Doctorale Systèmes

Spécialité Systèmes Informatiques

Par M. Benjamin Lussier

TOLÉRANCE AUX FAUTES DANS LES SYSTÈMES AUTONOMES

Soutenue le 24 avril 2007 devant le jury composé de :

M.	C. PECHEUR	Président
M.	M. BANÂTRE	Rapporteur
M.	B. ESPIAU	Rapporteur
M.	J.P. BLANQUART	Membre
M.	F. INGRAND	Membre
M.	D. POWELL	Directeur de thèse

Avant-propos

Les travaux présentés dans ce mémoire ont été réalisés au *Laboratoire d'Analyse et d'Architecture des Systèmes* du *Centre National de la Recherche Scientifique* (LAAS-CNRS). Je remercie Messieurs Malik Ghallab et Raja Chatila, qui ont assuré la direction du LAAS-CNRS depuis mon entrée, pour m'avoir reçu au sein de ce laboratoire. Je remercie également Monsieur Jean Arlat, responsable du groupe *Tolérance aux fautes et Sûreté de Fonctionnement informatique* (TSF), et Messieurs Raja Chatila et Rachid Alami, responsables successifs du groupe *Robotique et InteractionS* (RIS), pour m'avoir acueilli dans ces groupes de recherche.

Je remercie Monsieur Charles Pecheur, Professeur à l'Université Catholique de Louvain, pour avoir présidé mon jury de thèse, ainsi que :

- Monsieur Michel Banâtre, directeur de recherche à INRIA Rennes (IRISA),
- Monsieur Jean-Paul Blanquart, ingénieur d'études à Astrium,
- Monsieur Bernard Espiau, directeur de recherche à INRIA Rhônes-Alpes,

pour avoir participé à ce jury. Je remercie en particulier Messieurs Michel Banâtre et Bernard Espiau qui ont accepté la charge d'être rapporteurs.

Merci, surtout, à ceux qui m'ont encadré pendant ces longs travaux : Raja Chatila, Félix Ingrand, Marc-Olivier Killijian et David Powell. Merci particulièrement à David, pour sa rigueur scientifique, ses conseils et ses relectures diligentes, et à Félix, pour son expérience en robotique, sa compréhension et sa disponibilité constantes.

Je remercie aussi les autres participants des réunions *Systèmes Autonomes Critiques* (SAC), épisodiques ou réguliers : Alexandre, Étienne, Jérémie et Matthieu. Merci en particulier à Matthieu, pour son aide inestimable dans la compréhension et la manipulation du planificateur IxTeT.

Merci aux membres du groupe RIS, passés et présents, qui m'ont appris à manipuler des robots tout au long de mon parcours : Alexandre, Aurélie, Frédéric, Guillaume, Matthieu (Gallien), Matthieu (Herrb), Sara, Sylvain...

Merci aux membres du groupe TSF, avec qui j'ai passé de nombreuses journées bien accompagné. Pour ceux qui n'ont pas encore fini leur rédaction de thèse, bon courage. Merci en particulier aux anciens et nouveaux camarades du célèbre bureau 10 : Anis, Caroline, *mentor* Éric, *an apple a day* Étienne, *libriciel* Ludo, Manel, Noredine, *youyou* Youssef.

Merci, enfin, à mes parents, mon frère, mes amis, sans lesquels ce mémoire serait probablement très différent.

Table des matières

Introduction **9**

1 Terminologie et état de l'art **11**
1.1 La sûreté de fonctionnement informatique 11
1.1.1 Principes généraux de la sûreté de fonctionnement 11
1.1.2 La tolérance aux fautes 13
1.1.2.1 Principe de la tolérance aux fautes 13
1.1.2.2 Tolérance aux fautes de développement 14
1.1.2.3 Validation des mécanismes de tolérance aux fautes 16
1.2 Les systèmes autonomes 16
1.2.1 Notion d'autonomie 17
1.2.2 Mécanismes décisionnels 18
1.2.2.1 Propriétés et fonctions 18
1.2.2.2 Exemples de mécanismes décisionnels 19
1.2.2.3 Particularité vis-à-vis des fautes 20
1.2.3 Concept de robustesse 21
1.2.4 Architecture des systèmes autonomes 23
1.2.4.1 Style d'architecture subsomptif 24
1.2.4.2 Style d'architecture hiérarchisé 24
1.2.4.3 Approche multi-agents 26
1.3 Analyse de l'existant 27
1.3.1 État de l'art en robustesse 27
1.3.1.1 Observation et choix 27
1.3.1.2 Détection et traitement 28
1.3.2 État de l'art en sûreté de fonctionnement 31
1.3.2.1 Prévention des fautes 31
1.3.2.2 Élimination des fautes 31
1.3.2.3 Tolérance aux fautes 32
1.3.2.4 Prévision des fautes 34
1.4 Conclusion 34

2 Architectures et méthodes pour une autonomie sûre de fonctionnement **37**
2.1 Aspect architectural 37
2.1.1 Les limites du type d'architecture hiérarchisé 38
2.1.1.1 Risque de désaccords 38

2.1.1.2 Lenteur de réaction face à certaines situations adverses . . . 39
2.1.1.3 Criticité du mécanisme décisionnel . . . 40
2.1.2 Une alternative : le type d'architecture multi-agents . . . 40
2.1.2.1 Réponses aux limites d'une architecture hiérarchisée 41
2.1.2.2 Inconvénients spécifiques au type d'architecture multi-agents . . . 42
2.2 Composant indépendant de sécurité . . . 42
2.2.1 Caractéristiques . . . 43
2.2.1.1 Règles de sécurité . . . 43
2.2.1.2 Observation et action . . . 43
2.2.2 Applications dans les systèmes autonomes . . . 44
2.2.2.1 Request and Report Checker . . . 44
2.2.2.2 Guardian Agent . . . 45
2.3 Mécanismes liés au contrôle d'exécution et à la planification . . . 46
2.3.1 Reprise de situations adverses par le contrôle d'exécution . 46
2.3.1.1 Détection d'une situation adverse . . . 46
2.3.1.2 Réaction par reprise . . . 46
2.3.1.3 Modalités . . . 47
2.3.2 Planification et sûreté de fonctionnement . . . 47
2.3.2.1 Difficultés spécifiques liées à la planification . . . 48
2.3.2.2 Méthodes existantes . . . 49
2.4 Conclusion . . . 50

3 Tolérance aux fautes pour la planification 53
3.1 Mécanismes proposés . . . 53
3.1.1 Mécanismes de détection . . . 54
3.1.2 Mécanismes de rétablissement . . . 55
3.1.2.1 Planification successive . . . 55
3.1.2.2 Planification concurrente . . . 57
3.1.3 Le composant FTplan . . . 59
3.1.3.1 Place dans l'architecture . . . 59
3.1.3.2 Principales fonctions du composant . . . 60
3.2 Mise en œuvre . . . 61
3.2.1 L'architecture LAAS . . . 61
3.2.1.1 Présentation de l'architecture . . . 61
3.2.1.2 IxTeT . . . 63
3.2.2 Intégration de FTplan dans l'architecture LAAS . . . 66
3.2.2.1 Place dans l'architecture LAAS . . . 66
3.2.2.2 Fonctionnement de FTplan . . . 67
3.2.2.3 Modifications nécessaires de l'existant . . . 70
3.3 Conclusion . . . 72

4 Évaluation des mécanismes de tolérance aux fautes 75
4.1 Environnement d'évaluation . . . 75
4.1.1 Environnement logiciel . . . 76
4.1.2 Ensemble d'activités . . . 78

4.1.2.1 Missions et environnements 79
4.1.2.2 Modèles de planification 80
4.1.3 Ensemble de fautes 83
4.1.4 Relevés et mesures 86
4.2 Résultats 88
4.2.1 Comportement sans injection de fautes 88
4.2.1.1 Comportement nominal 88
4.2.1.2 Coût de l'utilisation de FTplan 90
4.2.2 Comportement en présence de fautes injectées 93
4.2.2.1 Fautes injectées 93
4.2.2.2 Exemple d'expérimentations 99
4.2.2.3 Résultats généraux 102
4.3 Conclusion 104

5 Conclusions et perspectives **107**
5.1 Démarche suivie 107
5.2 Leçons apprises 109
5.3 Perspectives 110

A Exemples de modèles de planification **113**
A.1 Modèle1 113
A.2 Modèle2 119

Bibliographie **131**

Introduction

Les systèmes autonomes couvrent un large spectre de fonctionnalités, allant de l'animal de compagnie artificiel jusqu'au robot d'exploration martien. Des prototypes expérimentaux ont déjà été utilisés comme guides de musée, et l'autonomie est sérieusement envisagée comme réponse à des défis de société comme le vieillissement de la population, à travers l'utilisation d'assistants personnels robotisés. D'autres recherches portent sur des systèmes autonomes aquatiques, des essaims de robots, des hélicoptères autonomes de surveillance, etc...

Le développement de telles applications a été rendu possible par deux facteurs technologiques fondamentaux améliorant la robustesse de ces systèmes et leur permettant d'accomplir leurs missions dans des environnements de plus en plus diversifiés et ouverts. D'une part, les fonctionnalités de bas niveau (telles que la localisation ou la navigation), critiques pour un système autonome, commencent à montrer des résultats satisfaisants sur des plate-formes expérimentales, ainsi que le prouve l'expérience du DARPA Grand Challenge 2006 [Monterlo et al. 2006]. D'autre part, le développement des mécanismes de décision dérivés de l'intelligence artificielle, et en particulier la planification, permettent de tenir compte de concepts de haut niveau, et ainsi de résoudre des missions de plus en plus complexes.

Dans la pratique cependant, les applications de grande envergure dans le domaine de l'autonomie restent majoritairement centrées sur les fonctionnalités de bas niveau. En effet, les systèmes autonomes disposant de mécanismes de décision complexes sont confrontés à un manque de confiance dans l'évaluation et la prédiction de leur comportement, ce qui réduit significativement leurs applications.

Dans des systèmes informatisés critiques plus traditionnels que les systèmes autonomes, par exemple en aéronautique ou dans des centrales nucléaires, ce manque de confiance est traité à travers l'utilisation de techniques de la sûreté de fonctionnement, qui a justement pour but de donner une confiance justifiée dans un système. La tolérance aux fautes, en particulier, cherche comment mettre en place un système à même de remplir sa fonction malgré la présence de fautes. L'application de ces techniques dans le cadre des systèmes autonomes n'est cependant pas triviale, notamment vis-à-vis des fautes de conception. En effet, la plupart des mécanismes de tels systèmes, et en particulier les mécanismes décisionnels, utilisent des langages et des algorithmes spécifiques (par exemple une approche de programmation déclarative), différents de ceux utilisés dans des systèmes informatisés classiques.

Ainsi, l'application des techniques de sûreté de fonctionnement aux systèmes

autonomes demande tout d'abord de répondre à deux questions primordiales :

- quels liens existe-t-il entre la sûreté de fonctionnement et les mécanismes robustes utilisés par les roboticiens pour maîtriser la complexité d'un environnement ouvert ?
- quelles sont les particularités des systèmes autonomes qui entravent l'application des mécanismes classiques de la sûreté de fonctionnement informatique, et quelles solutions peut-on y apporter ?

Le but de cette thèse, centrée sur la tolérance aux fautes dans les systèmes autonomes, est d'apporter un début de réponse à cette problématique. Entre autres, elle cherche à présenter les aspects complémentaires de la robustesse et de la sûreté de fonctionnement, et à proposer des mécanismes de tolérance aux fautes pour la planification, une des fonctions décisionnelles essentielles aux systèmes autonomes, ainsi qu'une approche d'évaluation de tels mécanismes. Le présent manuscrit est décomposé en quatre chapitres :

Le premier chapitre présente les domaines de la sûreté de fonctionnement et des systèmes autonomes. Il donne les définitions et développe les notions fondamentales relatives à chaque communauté, en particulier celle de tolérance aux fautes pour la sûreté de fonctionnement, ainsi que celles de mécanismes décisionnels et de robustesse pour les systèmes autonomes. Il introduit finalement un état de l'art relatif aux mécanismes de robustesse et de sûreté de fonctionnement informatique employés à l'heure actuelle dans les systèmes autonomes.

Le deuxième chapitre étudie plusieurs pistes d'étude cherchant à améliorer la sûreté de fonctionnement dans les systèmes autonomes. Il s'intéresse en particulier à leur aspect architectural, à l'utilisation prometteuse pour la sûreté de fonctionnement d'un composant indépendant de sécurité, ainsi qu'aux mécanismes liés à deux fonctionnalités primordiales dans les mécanismes décisionnels : le contrôle d'exécution et la planification.

Le troisième chapitre présente différents mécanismes de tolérance aux fautes que nous proposons pour la planification dans les systèmes autonomes. Ces mécanismes, basés sur le principe de diversification, visent en priorité à tolérer des fautes de conception et de programmation dans les connaissances utilisées par des planificateurs : les modèles et les heuristiques de recherche qu'ils utilisent pour décider des actions à exécuter par le système.

Le quatrième chapitre introduit un environnement d'évaluation que nous avons développé dans le but de valider les mécanismes de tolérance aux fautes proposés. Cet environnement s'appuie sur l'injection de fautes et la simulation des composants matériels d'un système autonome réel. Nous présentons également dans ce chapitre les résultats de notre campagne d'évaluation, et jugeons de l'efficacité des mécanismes proposés.

Enfin, une conclusion retrace les points essentiels de ce mémoire, et présente plusieurs pistes de recherche prospective dans le prolongement de nos travaux.

Chapitre 1

Terminologie et état de l'art

Les travaux de cette thèse abordent deux domaines très différents des systèmes informatiques, chacun employant un vocabulaire et une terminologie propres à résoudre les difficultés qu'ils considèrent. La sûreté de fonctionnement informatique est centrée sur la notion de faute, comme cause potentielle de mauvais fonctionnement d'un système informatique, et propose divers moyens, dont la tolérance aux fautes, pour y faire face. La robotique et l'intelligence artificielle, à travers les notions de mécanisme décisionnel, incertitude et robustesse, permettent de développer des systèmes autonomes, capables de remplir des missions complexes avec un minimum d'intervention humaine.

Ce premier chapitre décrit le terminologie et les concepts autour desquels s'articulent nos travaux de thèse. Il présente tout d'abord les notions relatives aux systèmes informatiques sûrs de fonctionnement, et celles relatives aux systèmes autonomes. Il propose ensuite un rapide état de l'art des mécanismes de sûreté de fonctionnement et de robustesse dans les systèmes autonomes.

1.1 La sûreté de fonctionnement informatique

Les outils et logiciels informatiques sont aujourd'hui incontournables et omniprésents dans notre société, de la gestion des communications téléphoniques et du traitement de transactions bancaires, à la commande de systèmes de transports, de production automatisés, ou de centrales énergétiques. L'indisponibilité, l'imprécision ou la défaillance de tels systèmes peuvent avoir des répercussions bien plus graves que le simple inconfort des utilisateurs, allant dans certains cas jusqu'à des conséquences catastrophiques en terme économique ou de vie humaine. La notion de sûreté de fonctionnement informatique a été développée en réponse à cette préoccupation, en fournissant des méthodes permettant de prévenir et traiter la défaillance de systèmes informatiques.

1.1.1 Principes généraux de la sûreté de fonctionnement

La *sûreté de fonctionnement* d'un système est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service que le système leur délivre,

c'est-à-dire dans son comportement tel qu'ils le perçoivent [Laprie et al. 1996] [Avižienis et al. 2005]. Cette propriété englobe trois différentes notions : ses *attributs*, les propriétés complémentaires qui la caractérisent ; ses *entraves*, les circonstances indésirables qui sont causes ou résultats de la non-sûreté de fonctionnement ; ses *moyens*, les méthodes et techniques qui cherchent à rendre un système capable d'accomplir correctement sa fonction, et à donner confiance dans cette aptitude.

Attributs

Les attributs de la sûreté de fonctionnement sont définis par diverses propriétés, dont l'importance relative dépend de l'application et de l'environnement auxquels est destiné le système informatique considéré. Les six attributs les plus habituellement considérés sont : la *disponibilité*, la *fiabilité*, la *sécurité-innocuité*, la *confidentialité*, l'*intégrité*, et la *maintenabilité*.

Dans ce manuscrit, nous nous intéressons en particulier à la fiabilité, qui caractérise la continuité de service, et la sécurité-innocuité, qui caractérise la non-occurrence de conséquences catastrophiques pour l'environnement.

Entraves

Dans les entraves à la sûreté de fonctionnement, on différencie les défaillances, les erreurs et les fautes. Une *défaillance* survient lorsque le service dévie de l'accomplissement de la fonction du système ; une *erreur* est la partie de l'état du système qui est susceptible d'entraîner une défaillance ; une *faute* est la cause adjugée ou supposée d'une erreur.

Une faute activée produit une erreur, qui peut se propager dans un composant ou d'un composant à un autre jusqu'à provoquer une défaillance. La défaillance d'un composant cause une faute permanente ou temporaire dans le système qui le contient, tandis que la défaillance d'un système cause une faute permanente ou temporaire pour les systèmes avec lesquels il interagit. Ce processus de propagation est représenté sur la Figure 1.1.

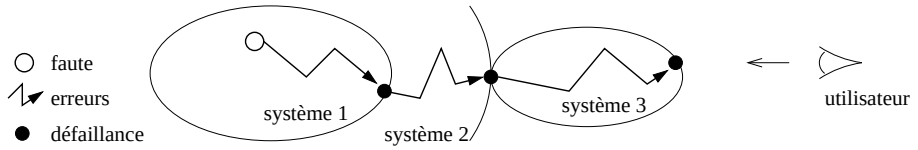

Figure 1.1: Propagation d'erreurs dans un système de systèmes

Suivant leur origine, nature, ou phase de création, les fautes sont classées en trois types principaux : *fautes de conception*, *fautes physiques* et *fautes d'interaction*. Dans la suite de ce manuscrit, nous considérons principalement les erreurs et défaillances liées aux fautes de conception : des fautes internes au système, introduites par le concepteur ou le programmeur pendant la phase de développement du système.

Moyens

Le développement d'un système sûr de fonctionnement passe par l'utilisation

combinée d'un ensemble de méthodes qui sont classées en quatre moyens :

- *prévention des fautes* : comment empêcher l'occurrence ou l'introduction de fautes ; elle est principalement atteinte par des méthodes de spécification et de développement relevant de l'ingénierie des systèmes ;
- *élimination des fautes* : comment réduire le nombre et la sévérité des fautes ; elle peut être réalisée pendant la phase de développement d'un système par vérification, diagnostic et correction, ou pendant sa phase opérationnelle par maintenance ;
- *tolérance aux fautes* : comment fournir un service à même de remplir la fonction du système en dépit des fautes ; elle est mise en œuvre par la détection d'erreurs et le rétablissement du système ;
- *prévision des fautes* : comment estimer la présence, la création et les conséquences des fautes ; elle est effectuée par évaluation du comportement du système par rapport à l'occurrence des fautes et à leur activation.

La prévention et l'élimination des fautes peuvent se regrouper sous le concept d'*évitement des fautes* : on cherche à concevoir un système sujet au moins de fautes possible. La tolérance aux fautes et la prévision des fautes peuvent se regrouper sous le concept d'*acceptation des fautes* : partant du principe qu'il y a toujours des fautes qu'on ne peut pas éviter, on essaie d'évaluer leurs impacts sur le système et de réduire la sévérité des défaillances qu'elles peuvent causer (si possible jusqu'à la suppression des défaillances). Ces deux concepts sont des approches *complémentaires* pour concevoir un système sûr de fonctionnement.

1.1.2 La tolérance aux fautes

Dans ce manuscrit, nous ciblons en particulier la tolérance aux fautes, le moyen de la sûreté de fonctionnement qui cherche à maintenir un service correct délivré par le système malgré la présence de fautes. Après une rapide revue des principes de base de la tolérance aux fautes, nous nous attardons sur les spécificités relatives aux fautes de conception, et le problème de la validation des mécanismes de tolérance aux fautes.

1.1.2.1 Principe de la tolérance aux fautes

La tolérance aux fautes s'effectue le plus souvent en deux temps :

- la *détection d'erreurs* provoque la levée d'un signal ou d'un message d'erreur à l'intérieur du système,
- le *rétablissement du système*, déclenché par le signal ou message, substitue un état jugé exempt d'erreurs et de fautes à l'état erroné détecté.

Détection d'erreurs

Les formes les plus courantes de détection d'erreur sont : duplication et comparaison, contrôle temporel, et contrôle de vraisemblance.

- La méthode de *duplication et comparaison* utilise au moins deux unités redondantes, qui doivent être indépendantes vis-à-vis des fautes que l'on souhaite tolérer : typiquement, redondance des composants matériels pour les fautes physiques, et diversification pour des fautes de conception.

- Le *contrôle temporel* par "chien de garde" est typiquement utilisé pour détecter la défaillance d'un périphérique en vérifiant que son temps de réponse ne dépasse pas une valeur-seuil, ou pour surveiller périodiquement l'activité d'une unité centrale.
- Le *contrôle de vraisemblance* cherche à détecter des erreurs en valeur aberrantes pour le système. Il peut être mis en œuvre soit par du matériel pour détecter par exemple des adresses mémoires erronées, soit par du logiciel pour vérifier la conformité des entrées, des sorties ou des variables internes du système par rapport à des invariants.

Il existe de nombreux autres mécanismes de détection d'erreurs, plus éloignés de nos travaux, et que, faute de place, nous ne développerons pas ici.

Rétablissement du système

Le rétablissement du système peut être assuré par deux méthodes complémentaires : le *traitement d'erreur*, qui vise à corriger l'état erroné du système, et le *traitement de faute*, qui vise à empêcher la faute à l'origine de l'erreur d'être à nouveau activée.

Parmi les techniques de traitement d'erreur, on distingue :

- le *reprise d'erreur*, qui consiste à sauvegarder périodiquement l'état du système pour le ramener dans un état antérieur à la détection d'erreur ; les principaux inconvénients de la reprise sont la taille des sauvegardes, la difficulté d'effectuer des sauvegardes cohérentes, et le sur-coût temporel nécessaire à leur établissement,
- le *rétablissement par poursuite*, qui consiste à créer un nouvel état à partir duquel le système peut continuer à fonctionner de façon acceptable, éventuellement en mode dégradé ; une forme limite de cette technique est la mise du système dans un état sûr, par exemple en l'arrêtant,
- la *compensation d'erreurs*, qui consiste à utiliser des redondances présentes dans le système pour fournir un service correct en dépit des erreurs ; cette compensation peut être consécutive à une détection d'erreur (détection et compensation), ou appliquée systématiquement (masquage d'erreur).

Le traitement de faute est composé de quatre phases successives :

- le *diagnostic* a pour but d'identifier la localisation et le type de la faute responsable de l'état erroné du système,
- l'*isolement* consiste à exclure la participation du composant erroné de la délivrance du service, par moyen physique ou logiciel,
- la *reconfiguration* cherche à compenser l'isolement du composant défaillant, soit en basculant sur des composants redondants, soit en réassignant ses tâches à d'autres composants,
- la *réinitialisation* vérifie et met à jour la nouvelle configuration du système.

Des exemples concrets d'application de ces différents mécanismes peuvent se trouver dans [Essamé et al. 2000].

1.1.2.2 Tolérance aux fautes de développement

La tolérance aux fautes de développement se démarque de la tolérance aux fautes physiques ou d'interaction par le type de redondance à mettre en place pour la détection d'erreur et le rétablissement du système. Cette redondance doit en

effet favoriser l'indépendance de l'activation des fautes de développement et fait donc généralement appel à des conceptions diversifiées. On peut distinguer deux types d'approche à la tolérance aux fautes de conception, selon que l'on cherche simplement à limiter les conséquences de ces fautes, ou à assurer la continuité de service de la fonction affectée.

Détection et mise en état sûr

Pour limiter les conséquences des fautes de conception, on cherche à mettre en place un composant autotestable au moyen d'une détection d'erreur, pour déclencher ensuite la mise en état sûr du composant (une forme extrême de rétablissement par poursuite). La détection d'erreur doit être appropriée pour les fautes de conception, ce qui exclut une duplication à l'identique de ce composant. Cette approche cherche à assurer la sécurité-innocuité du composant, et éviter la propagation d'erreur dans le reste du système. Elle est principalement réalisée par la méthode *fail-fast* basée sur une détection par comparaison entre deux exemplaires fonctionnellement diversifiés d'un composant, et l'utilisation d'un *composant indépendant de sécurité*.

Un *composant indépendant de sécurité*, parfois appelé *safety bag*, est un composant qui intercepte les actions sollicitées par un utilisateur ou un composant du système, et contrôle leur validité suivant des règles de sécurité établies pendant le développement (il s'agit d'une forme de contrôle de vraisemblance). Le composant indépendant de sécurité possède sa propre représentation de l'état du système, qui doit être suffisamment précise et récente pour permettre d'appliquer l'ensemble de règles qu'il supervise. Cette technique est introduite dans [Klein 1991].

Détection et rétablissement

Pour assurer la continuité de service du système (c'est-à-dire sa fiabilité), on fait obligatoirement appel à la diversification fonctionnelle, qui repose sur l'utilisation de plusieurs exemplaires d'un composant, appelés *variantes*, conçus et réalisés séparément à partir de la même spécification. Elle fait également appel à un *décideur*, destiné à fournir un résultat supposé exempt d'erreur à partir de l'exécution des variantes. On peut distinguer trois méthodes fondamentales exploitant la diversification fonctionnelle pour assurer la continuité de service : les blocs de recouvrement, la programmation en N-versions et la programmation N-autotestable.

Dans les *blocs de recouvrement*, le décideur est un test d'acceptation qui est appliqué séquentiellement aux résultats fournis par les variantes. Si les résultats fournis par la première variante ne sont pas satisfaisants, la deuxième variante est exécutée, et ainsi de suite jusqu'à la satisfaction du test d'acceptation ou l'épuisement des variantes disponibles, auquel cas le bloc de recouvrement est globalement défaillant.

Dans la *programmation en N-versions*, le décideur effectue un vote majoritaire sur les résultats de toutes les variantes.

Dans la *programmation N-autotestable*, au moins deux composants autotestables sont exécutés en parallèle, chacun étant constitué soit de l'association d'une variante et d'un test d'acceptation, soit de deux variantes et d'un algorithme de comparaison.

1.1.2.3 Validation des mécanismes de tolérance aux fautes

Quelles que soient les méthodes de détection et de rétablissement appliquées, la tolérance aux fautes est un processus coûteux, autant en coût financier et en temps de développement du système, qu'en terme de performances pendant son exécution. Il est donc indispensable de connaître l'apport des mécanismes utilisés en terme de sûreté de fonctionnement par rapport à ces divers coûts. De plus, la conception et la réalisation des mécanismes de tolérance aux fautes sont des activités de développement aussi faillibles que les autres, et peuvent introduire de nouvelles fautes dans le système. Il est donc important de procéder à l'évaluation et la validation de ces mécanismes.

Pour répondre à ces problèmes, l'évaluation et la validation des mécanismes de tolérance aux fautes peuvent être réalisées suivant deux approches complémentaires : la vérification formelle et l'injection de fautes.

- Les méthodes de *vérification formelle* consistent à utiliser des techniques formelles comme l'analyse statique, la preuve mathématique, ou l'analyse de comportement, pour obtenir des certitudes sur le comportement du système. Ces méthodes permettent également de valider le comportement d'un système en présence de fautes, à condition de disposer de modèles formels décrivant les fautes considérées et leurs conséquences sur le comportement du système.
- L'*injection de fautes* est une méthode de test ou d'évaluation des mécanismes de tolérance aux fautes, qui consiste à introduire délibérément des fautes dans un système ou dans un modèle du système, puis à observer son comportement, afin de révéler les déficiences de ces mécanismes.
  On peut simuler des fautes pour représenter des fautes physiques ou logicielles, par altération du contenu des composants mémoires du système, ou par modification des entrées et des sorties d'un composant. L'*injection de mutations* est une technique spécifique à la simulation de fautes logicielles : à partir d'un programme original supposé correct, on crée un ensemble de programmes appelés mutants qui diffèrent du programme original par une seule modification élémentaire syntaxiquement correcte. Le comportement des mécanismes de tolérance aux fautes logicielles est ensuite observé en appliquant à ces mutants des entrées de test cherchant à activer la faute injectée.

1.2 Les systèmes autonomes

Le concept de *système autonome* est une généralisation de la notion de *robot* développée dans le domaine de la robotique. Un robot est défini par le dictionnaire comme "un appareil automatique capable de manipuler des objets ou d'exécuter des opérations selon un programme fixe ou modifiable". La robotique permet à l'homme d'étendre son emprise sur l'environnement par l'exécution reproductible de tâches laborieuses et répétitives, et la possibilité d'agir à distance, et d'éviter l'exposition d'êtres humains à un environnement dangereux.

Le développement d'une *autonomie* pour de tels systèmes, rendu possible par les mécanismes d'*intelligence artificielle*, augmente encore les possibilités qu'ils

peuvent offrir : dans le domaine de l'exploration spatiale par exemple, où le temps de latence des communications rend difficiles les téléopérations, mais de nombreux domaines sont concernés, du milieu médical jusqu'au divertissement.

1.2.1 Notion d'autonomie

D'après le dictionnaire, l'autonomie est "la capacité de se gérer soi-même". Cependant, cette définition n'est pas suffisante dans le cadre de la robotique, puisqu'elle ne permet pas d'établir une distinction entre un *système automatique*, qui ne fait qu'appliquer des consignes préprogrammées dépendant directement de ses entrées, et un *système autonome*, chargé de taches plus complexes, dont les détails de réalisation ont été laissés à sa propre initiative par nécessité ou par volonté de simplifier son utilisation.

Plusieurs travaux ont tenté de définir et caractériser le degré d'autonomie d'un système. [Clough 2002] donne deux définitions de l'autonomie : "avoir sa volonté propre" et "la capacité de générer ses propres objectifs sans instructions externes". Elles nous semblent cependant à la fois trop vagues et trop ambitieuses pour caractériser le type de système qui nous intéresse. L'article propose également une classification selon quatre axes orthogonaux, représentant différents domaines de compétence d'un système autonome (perception, analyse de la situation, prise de décision, et exécution des actions).

Des travaux ultérieurs du *National Institute of Standards and Technology* [Huang 2004] établissent une définition qui nous parait plus appropriée à l'autonomie d'un système robotique, et que nous adopterons dans la suite de ce manuscrit comme description d'un système autonome :

> l'autonomie est la capacité d'un système de percevoir, analyser, communiquer, planifier, établir des décisions, et agir, afin d'atteindre des objectifs assignés par un opérateur humain ; cette autonomie est mesurée par les aptitudes du système selon divers facteurs, incluant la complexité de la mission, la difficulté de l'environnement, et les interactions désirées ou non désirées avec différentes catégories d'êtres humains (opérateur, co-équipier, passant)[1].

Il faut noter que cette description ne permet pas de faire une dichotomie entre automatique et autonome. En fait, on trouve un large spectre de systèmes, allant du plus simple asservissement à la plus complexe machine cognitive.

Cette définition de l'autonomie fait néanmoins apparaître deux aspects majeurs des systèmes autonomes. Le premier aspect est la capacité de prendre des décisions, mise en œuvre par les *mécanismes décisionnels* développés dans le domaine de l'intelligence artificielle. Le second est la nécessité de prendre en compte les incertitudes et l'évolution de l'environnement dans lequel le système évolue.

[1] "*Autonomy: An unmanned system's own ability of sensing, perceiving, analyzing, communicating, planning, decision-making, and acting, to achieve its goals as assigned by its human operator(s) through desired Human Robot Interaction. Autonomy is characterized into levels by factors including mission complexity, environmental difficulty, and level of HRI to accomplish the missions.*" [Huang 2004]

Cet aspect se retrouve dans le domaine plus spécifique de la robotique, et est communément associé avec la notion de *robustesse*.

1.2.2 Mécanismes décisionnels

Les mécanismes décisionnels sont essentiels à tout système autonome, parce qu'ils réalisent les analyses de données, les choix d'actions, ou les adaptations à l'environnement susceptibles de mener à l'accomplissement des objectifs du système. Un mécanisme décisionnel est composé de *connaissances* spécifiques au domaine d'application, telles que des heuristiques ou un modèle de l'environnement, et un *mécanisme d'inférence* utilisé pour résoudre des problèmes et manipuler ces connaissances. En théorie, le mécanisme d'inférence est indépendant du domaine d'application, et peut être réutilisé dans un autre domaine avec des connaissances appropriées. En pratique, connaissances et mécanisme d'inférence sont parfois difficiles à dissocier : les heuristiques de recherche, par exemple, font partie des connaissances du mécanisme décisionnel et dépendent de l'application, mais sont étroitement intégrées au mécanisme d'inférence utilisé.

1.2.2.1 Propriétés et fonctions

Appliqué à des connaissances idéalement complètes et sans faute, un mécanisme d'inférence peut être caractérisé par trois propriétés principales [Nilsson 1998] :

- la *justesse* indique que toute conclusion déduite par le mécanisme d'inférence est correcte,
- la *complétude* indique que si une conclusion correcte existe, elle sera inéluctablement inférée,
- la *calculabilité* caractérise la complexité du mécanisme d'inférence : elle indique s'il est capable de trouver la solution d'un problème en espace et en temps polynômiaux[2].

Les mécanismes d'inférence utilisés actuellement dans les systèmes autonomes sont généralement justes et complets, mais non-calculables. En pratique, des heuristiques sont ainsi souvent incluses dans ces mécanismes, améliorant les performances du système, mais sacrifiant en contrepartie la complétude du mécanisme d'inférence.

Les mécanismes décisionnels peuvent aussi être caractérisés par la fonction qu'ils occupent dans le système pour enrichir l'autonomie. On peut distinguer cinq fonctions principales liées à l'autonomie :

- La *planification* a pour but de choisir et d'organiser les tâches à entreprendre, en fonction de leur résultat prévu, afin d'atteindre un ou plusieurs objectifs.
- Le *contrôle d'exécution* coordonne et supervise l'exécution de séquences de tâches, en décomposant d'une part chaque tache en une séquence d'actions d'abstraction plus basses susceptible de la remplir, et d'autre part en réagissant à d'éventuelles erreurs dues à l'environnement ou au système.

[2] C'est-à-dire, si le temps et l'espace mémoire nécessaires au mécanisme pour trouver une solution sont du même ordre qu'une fonction polynômiale dépendant de la taille du problème posé.

- La *reconnaissance de situations* a pour but d'observer un historique d'évènements, et d'en tirer des conclusions sur l'état actuel du système et de son environnement, et éventuellement sur les intentions des agents qui en font partie.
- Le *diagnostic* permet d'identifier un état erroné du système, généralement après une détection d'erreurs.
- L'*apprentissage* cherche à améliorer les capacités d'un système en utilisant des informations liées à de précédentes exécutions. L'apprentissage est une fonction quelque peu à part, puisqu'il est principalement utilisé pour développer des modèles, qui sont ensuite appliqués aux fonctions précédentes.

1.2.2.2 Exemples de mécanismes décisionnels

Différentes approches ont été développées dans le domaine de l'intelligence artificielle pour mettre en œuvre les fonctions de l'autonomie décrites précédemment, chaque approche permettant de mettre en œuvre une ou plusieurs de ces fonctions. Nous présentons ci-dessous cinq principales approches qui sont utilisées de nos jours ; elles sont décrites avec plus de détail dans [Russell & Norvig 2002] :

- La *recherche dans un espace d'états* manipule un graphe d'actions et d'états, et consiste à examiner plusieurs séquences possibles d'actions à partir d'un état initial, puis à choisir la séquence la plus appropriée pour atteindre un état réalisant les objectifs demandés.
  Cette approche peut être utilisée pour la planification, le contrôle d'exécution, la reconnaissance de situations, et le diagnostic.
- La *technique de satisfaction de contraintes* vise à déterminer, pour un ensemble de variables, les domaines de valeurs qui satisfont un jeu de contraintes donné. La solution d'un problème de satisfaction de contraintes (CSP[3]) est obtenue en affectant successivement une valeur aux différentes variables, et en vérifiant à chaque fois que cette valeur respecte les différentes contraintes.
  À l'heure actuelle, les techniques de satisfaction de contraintes sont souvent utilisées pour réaliser de la *planification par contraintes*, en ajoutant une dimension temporelle aux CSP.
- Un *processus de décision de Markov* (MDP[4]) est un problème de décision séquentiel, dont la solution se présente sous la forme d'une *politique de conduite* précisant, pour chaque état possible du système, l'action à entreprendre pour maximiser la réalisation des objectifs. Une variante des MDP permet de traiter des cas plus complexes et réalistes, dans lesquels le système ne sait pas forcément dans quel état exact il se trouve.
  Cette approche est généralement utilisé pour les fonctions de planification ou d'apprentissage.
- Les *modèles de Markov cachés* (HMM[5]) sont des modèles temporels proba-

[3] *Contraint Satisfaction Problem*
[4] *Markov Decision Process*
[5] *Hidden Markov Model*

bilistes, et les *réseaux bayésiens dynamiques* (DBN[6]) sont des graphes orientés acycliques. Ils représentent tous deux des modèles équivalents, chacun adapté à des applications différentes, qui décrivent le comportement d'un système ou d'un processus, et permettent de traiter les problèmes d'incertitudes par l'usage des probabilités.
Ces approches sont le plus souvent utilisées pour mettre en œuvre des mécanismes de diagnostic ou d'apprentissage.

- Un *réseau neuronal* est constitué d'un ensemble d'unités, ou "neurones", interconnectées et organisées en une ou plusieurs couches. Il permet de définir une fonction non-linéaire complexe dépendant du schéma et du poids des connections.
  Les réseaux neuronaux sont principalement utilisés pour la fonction d'apprentissage.

D'autres approches existent dans le domaine de l'intelligence artificielle, mais ne sont pas utilisées actuellement dans le développement de systèmes autonomes, car trop restrictives ou pas encore assez abouties. Par exemple, les *systèmes experts* utilisent des bases de données et des règles logiques pour tirer des conclusions sur l'état du système, mais les règles qu'ils utilisent sont souvent trop simples pour permettre un raisonnement complexe, et leur base de connaissance de taille trop importante pour assurer sa cohérence ou sa convergence.

1.2.2.3 Particularité vis-à-vis des fautes

À partir des risques d'erreurs présentés dans [Boden 1989] et [Fox & Das 2000], nous pouvons distinguer des *fautes de développement* d'un mécanisme décisionnel, et des *fautes d'interaction* entre un mécanisme décisionnel et un autre composant ou système (par exemple un opérateur humain ou un autre mécanisme décisionnel). Ces différentes fautes sont reproduites sur la Figure 1.2 (nous ne considérons que les fautes d'origine accidentelle).

Les fautes de développement d'un mécanisme décisionnel peuvent porter sur son mécanisme d'inférence ou sur ses connaissances.

Les fautes dans le *mécanisme d'inférence* peuvent être soit des fautes de conception (par exemple, un choix de mécanisme décisionnel inadapté à la fonction du système, ou un défaut fondamental dans le principe d'inférence), soit des fautes de programmation (telle qu'une erreur de frappe ou d'algorithme).

Dans les *connaissances* d'un mécanisme décisionnel, les fautes de conception peuvent être soit des situations adverses spécifiées qui n'ont pas été implantées par les développeurs du système (par exemple une procédure ou une action manquante et nécessaire pour résoudre une situation adverse, ou l'absence de certaines situations dans le jeu d'apprentissage d'un réseau neuronal), soit l'imperfection du critère de décision qui peut amener le mécanisme décisionnel à inférer des conclusions erronées (par exemple une faute dans une heuristique, ou des données erronées dans certaines situations). Les fautes de programmation incluent des informations manquantes ou erronées dans les connaissances du système, formulées pendant le développement du système.

[6] *Dynamic Bayesian Network*

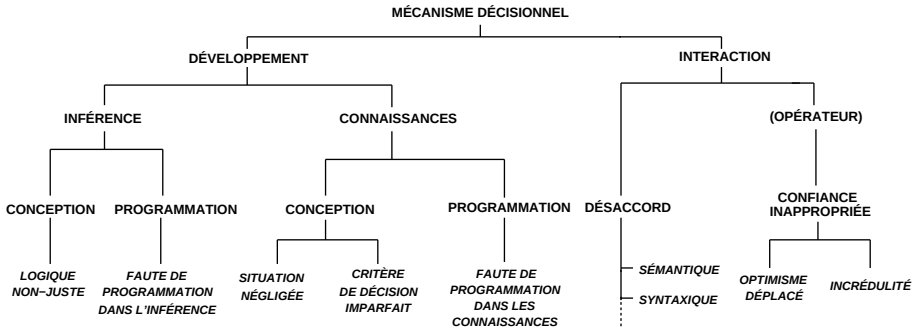

Figure 1.2: Fautes liées aux mécanismes décisionnels

Les principales fautes liées à l'interaction avec un autre composant ou système sont des *désaccords* (ou *mismatches*) sur les informations échangées. Ces fautes ne sont pas spécifiques aux mécanismes décisionnels. Elles peuvent être la cause de pertes ou de mauvaises interprétations de l'information, et sont classifiées dans différentes catégories [Jones et al. 2001]. En particulier, les *désaccords sémantiques* sont des fautes de développement qui apparaissent lorsque différents langages ou concepts sont utilisés par les composants communicants. Ce problème revêt une importance particulière dans les systèmes autonomes, souvent constitués de composants manipulant des niveaux d'abstractions différents.

Deux sources d'erreurs d'interaction ont encore été identifiées, mais correspondent à une problématique différente : l'interaction avec un opérateur humain. Ces risques sont :

- *un optimisme déplacé* dans les capacités du système : l'utilisateur place une trop grande confiance dans les décisions prises par le mécanisme décisionnel ; par exemple les résultats donnés par le mécanisme décisionnel sont très précis, conférant une illusion d'exactitude, mais faux,
- une *incrédulité* dans les capacités du système : l'utilisateur n'a pas confiance dans les décisions du système, par exemple si le processus de raisonnement du système lui paraît incompréhensible.

1.2.3 Concept de robustesse

Le concept de *robustesse* se retrouve dans de nombreux contextes différents. Dans le domaine de la robotique, la robustesse est vue comme une réponse à la variabilité importante du contexte d'exécution auquel peut être confronté un robot. Cette variabilité est due à l'*environnement ouvert* dans lequel le robot évolue, qui peut présenter, entre autres, des obstacles ou des variations de luminosité, à l'occurrence d'*évènements imprévus*, et aux *incertitudes d'observation et d'action*, résultant de l'incapacité du système à observer exactement son propre état et celui de l'environnement, et à prévoir les conséquences précises de ses actions.

Il n'existe cependant pas de définition précise de ce terme dans la communauté, et nous nous sommes efforcés d'établir une distinction claire entre la notion de robustesse dans le domaine robotique, et celle assez proche de tolérance aux fautes dans le domaine de la sûreté de fonctionnement.

D'après le dictionnaire, l'adjectif robuste qualifie quelqu'un ou quelque chose de solide ou de bien-portant. Le mot est utilisé dans ce sens dans de nombreux domaines (biologie, analyse statistique, informatique), mais les définitions plus techniques varient. En effet, dix-sept définitions, parfois contradictoires, ont été recensées par le *Santa Fe Institute* sur un site internet[7] dédié à la robustesse [Santa Fe Institute 2001]. Le dénominateur commun de ces définitions est l'aptitude d'un système à délivrer un service correct *face à un ensemble de situations*. Suivant la définition considérée, cet ensemble peut être (Figure 1.3) :

1. *toutes les situations* : cet ensemble regroupe les *situations nominales*, c'est-à-dire les situations idéales de fonctionnement du système, dans lesquelles il remplit sa fonction sans perturbations, et les *situations adverses* ;
2. les *situations adverses* : cet ensemble regroupe les situations où une perturbation menace la délivrance de la fonction du système ; cette perturbation peut être l'atteinte des limites de fonctionnement du système, l'activation d'une faute, ou l'occurrence d'évènements exceptionnels ou imprévus ;
3. les *situations adverses spécifiées* : cet ensemble regroupe les situations adverses dont la survenance est attendue, ou au moins envisagée pendant la phase de conception du système ;
4. les *situations adverses imprévues* : cet ensemble regroupe les situations adverses non prises en compte dans le domaine de fonctionnement spécifié du système, c'est-à-dire les situations adverses qu'on ne veut pas considérer ou qui n'ont pas été envisagées lors de la conception du système.

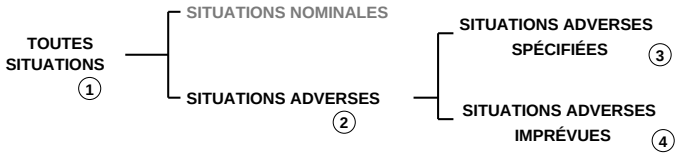

Figure 1.3: Ensembles de situations

Ces différentes définitions nous permettent d'établir un premier lien possible entre robustesse et tolérance aux fautes : les deux visent à augmenter l'aptitude du système à délivrer un comportement correct, mais la tolérance aux fautes cible un sous-ensemble spécifique des situations adverses auxquelles est confronté un système autonome : les fautes. D'après ce point de vue, la tolérance aux fautes apparaît donc comme un sous-ensemble de la robustesse.

Cependant, il nous a semblé plus productif (sur le plan de la définition des méthodes et des mécanismes à mettre en place) d'établir une distinction selon *l'origine des situations adverses* : soit interne, soit externe au système autonome. Nous retenons ainsi les deux définitions suivantes (Figure 1.4) :

- la *robustesse* est la délivrance d'un service correct en dépit de situations adverses dues aux incertitudes vis-à-vis de l'environnement du système (telles

[7] http ://discuss.santafe.edu/robustness

qu'un obstacle inattendu, ou un changement de la luminosité affectant les capteurs),

- la *tolérance aux fautes* est la délivrance d'un service correct en dépit de fautes affectant les différentes ressources du système (telles qu'un pneu crevé, la défaillance d'un capteur, ou une faute logicielle).

Il est à noter que cette définition de la robustesse se rapproche de celle utilisée en sûreté de fonctionnement informatique [Avižienis et al. 2005], qui est "la sûreté de fonctionnement par rapport aux fautes externes".

Figure 1.4: Robustesse et tolérance aux fautes

Une remarque peut encore être faite sur le service délivré par un système autonome. Un *service correct* est un comportement perçu par l'utilisateur comme conforme à la fonction du système [Avižienis et al. 2005]. Si celle-ci est définie *avant exécution*, et se retrouve dans les spécifications du système, les notions de service correct et de défaillance sont claires et non ambiguës. Un système sûr de fonctionnement devrait donc fournir un service correct dans toutes les situations prises en compte dans les spécifications, c'est-à-dire les situations nominales et les situations adverses spécifiées.

Cependant, un système autonome est souvent destiné à opérer dans un environnement ouvert, où les conditions d'utilisation ne peuvent pas toutes être déterminées ou décrites au préalable. Face à ces situations imprévues, il n'est pas possible de définir le comportement souhaité a priori (car dans le cas contraire la situation rencontrée serait justement prévue). Cependant, l'utilisateur peut juger *après exécution* que le service était *acceptable*.

1.2.4 Architecture des systèmes autonomes

L'approche SPA (pour *Sense-Plan-Act*) est une des premières approches dans la conception de systèmes autonomes. Aujourd'hui, cette approche est quasiment abandonnée, au moins dans sa forme de base. Elle propose une organisation du système en trois composants en boucle fermée : un *composant sensoriel* qui transforme les données des capteurs en une représentation du monde, un *composant de planification* qui développe à partir de cette représentation une séquence d'actions à effectuer, et un *composant exécutif* qui commande les actionneurs du système pour réaliser les actions prévues. Les limitations de cette approche sont importantes vis-à-vis du temps de réaction et de la supervision du système, à cause de la circulation unilatérale du flot de données et de l'utilisation quasi-exclusive de la planification comme mécanisme décisionnel.

Nous présentons dans cette section deux styles d'architectures qui lui ont succédé, et qui sont encore majoritairement employés à l'heure actuelle : le style d'ar-

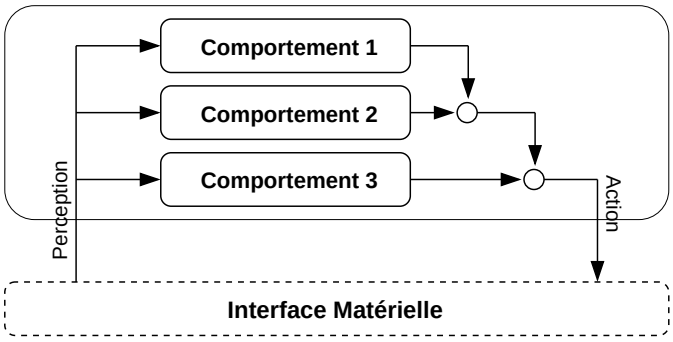

Figure 1.5: Style d'architecture subsomptif

chitecture *subsomptif* et le style d'architecture *hiérarchisé* (décrites toutes deux dans [Gat 1997]). Nous introduisons également le style d'architecture *multi-agents*, encore très prospectif.

1.2.4.1 Style d'architecture subsomptif

Le style d'architecture subsomptif (*Subsumption architecture*) a été introduit par Rodney Brooks dans [Brooks 1986] ; son principe est présentée dans la Figure 1.5. Il n'est pas basé sur une représentation symbolique de l'environnement ou du système, mais utilise plutôt différents programmes d'exécution appelés comportements (*behaviors*), spécifiques à une fonctionnalité donnée. À chaque cycle d'exécution, un comportement génère des commandes en fonction de ses entrées, c'est-à-dire des valeurs des capteurs du système. Les commandes exécutées par le système sont obtenues par combinaison des commandes des différents comportements : ils peuvent par exemple être ordonnés suivant la priorité de la fonction qu'ils adressent, ou s'inhiber les uns les autres.

L'architecture subsomptive est néanmoins limitée dans la complexité de ses fonctionnalités et la pertinence de ses réactions à l'environnement, à cause de la modularité insuffisante des comportements, qui ne permet pas de manipuler plusieurs niveaux d'abstraction dans le système. Cette architecture est donc généralement employée pour des systèmes de criticité faible, et dont les tâches ne sont pas très complexes. On peut trouver des exemples dans [Gat 1997], ou avec *Aibo*, le chien robot de Sony.

1.2.4.2 Style d'architecture hiérarchisé

Le style d'architecture hiérarchisé est à l'heure actuelle le plus utilisé pour la conception de systèmes autonomes complexes. Il organise le système en niveaux manipulant des représentations différentes du système, et possédant des contraintes temporelles spécifiques. Ces niveaux sont généralement au nombre de trois[8] : un niveau décisionnel, un niveau exécutif, et un niveau fonctionnel (Figure 1.6).

[8]Pour cette raison, une architecture hiérarchisé est souvent appelée architecture "trois niveaux".

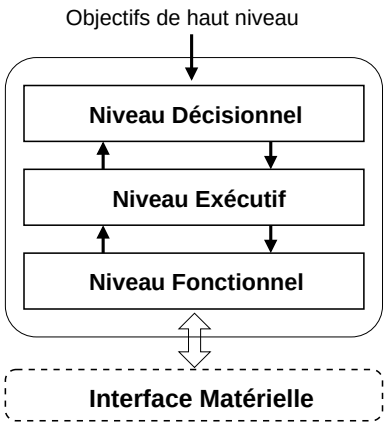

Figure 1.6: Style d'architecture hiérarchisé

- Le *niveau décisionnel* effectue les prises de décision du système à partir d'une représentation symbolique du monde. Il peut interagir avec le reste des composants de deux façons : en développant des plans à partir d'objectifs fixés par un opérateur, qui sont ensuite appliqués par le niveau exécutif, ou en traitant un problème ou une erreur soulevés par les niveaux inférieurs. Le temps d'exécution du niveau décisionnel ne peut pas être garanti, car ses mécanismes décisionnels sont souvent incalculables. Actuellement, les systèmes autonomes chargés de missions quelque peu complexes utilisent systématiquement des mécanismes de planification pour le niveau décisionnel, afin de traiter le haut niveau d'abstraction lié à cette complexité.
- Le *niveau exécutif* a pour but de sélectionner les fonctions élémentaires que le niveau fonctionnel doit exécuter pour réaliser la tâche de haut niveau planifiée par le niveau décisionnel ; il doit également réagir aux erreurs soulevées par le niveau fonctionnel et aux résultats imprévus des fonctions exécutées, et les faire remonter au niveau supérieur s'il s'avère incapable de les traiter lui-même. L'exécution du niveau exécutif doit être contrainte dans le temps pour assurer une supervision régulière des activités du système.
- Le *niveau fonctionnel* offre une interface entre la couche matérielle du système et les niveaux supérieurs, combinant capteurs et actionneurs en fonctions élémentaires dont l'exécution est commandée par les niveaux supérieurs. Il ne contient pas de représentation globale du système, mais doit garantir des contraintes d'exécution permettant un rafraîchissement des données des capteurs et des commandes des actionneurs suffisant pour assurer les fonctionnalités du système.

En pratique, certaines architectures adaptent cette décomposition : par exemple elles peuvent réunir décisionnel et exécutif dans le même niveau, ou ne pas considérer le niveau fonctionnel comme autre chose que l'interface matérielle.

On peut citer parmi les systèmes hiérarchisés l'architecture RAX [Muscettola et al. 1998] développée par le *NASA Ames Research Center* dans le cadre du projet *Deep Space One*, l'architecture CIRCA [Musliner et al. 1993] [Musliner et al. 1995] développée par le *Honeywell Research Center*, l'architecture

CLARAty [Volpe et al. 2000] développée principalement par le JPL (*Jet Propulsion Laboratory*), et l'architecture LAAS [Alami et al. 1998] [Ingrand et al. 2001] développée par le LAAS-CNRS.

ControlShell [Schneider et al. 1998] développée par l'entreprise *Real-Time Innovations*, SIGNAL [Marchand et al. 1998] développée par l'IRISA de Rennes, et ORCCAD [Borrelly et al. 1998] développée par l'INRIA Sophia Antipolis et l'INRIA Rhônes-Alpes, proposent également des méthodes et des environnements pour développer le niveau fonctionnel de systèmes robotiques.

1.2.4.3 Approche multi-agents

L'*approche multi-agents* considère un ensemble de systèmes autonomes (ou *agents*), homogènes ou hétérogènes, qui évoluent dans le même environnement et interagissent pour réaliser des tâches ou atteindre leurs objectifs. Ces objectifs peuvent être *égoïstes*, c'est-à-dire propre à chacun des agents, ou *communs* à plusieurs agents, et leur réalisation doit s'effectuer en évitant des conflits entre les différents systèmes (Figure 1.7).

L'approche multi-agents s'intéresse au développement de *protocoles de communication et d'interaction* pour les différents agents, éventuellement à travers leur hiérarchisation dynamique ou statique, et aux moyens de représenter pour chaque agent les *intentions* des autres agents et l'incertitude liée aux actions qu'ils vont décider d'effectuer. Une branche de cette approche considère un essaim (*swarm*) d'agents aux capacités de raisonnement limités, et cherche à faire émerger à travers leurs interactions un comportement "intelligent".

L'architecture IDEA [Muscettola et al. 2002] et [Finzi et al. 2003], développée par le *NASA Ames Research Center*, est un exemple d'architecture multi-agents, qui a pour objectif d'introduire des mécanismes décisionnels, plus particulièrement des mécanismes de planification partageant le même modèle global, à différents niveaux d'un système autonome.

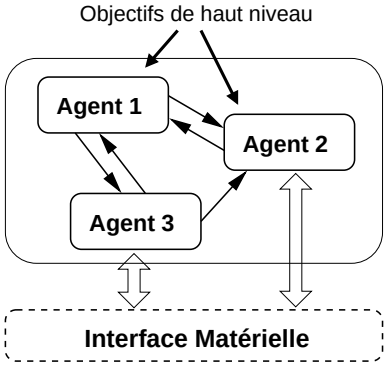

Figure 1.7: Style d'architecture multi-agents

1.3 Analyse de l'existant

L'autonomie des systèmes robotisés est aujourd'hui mise en œuvre avec de plus en plus de succès dans le milieu de la recherche, et on commence à en voir des concrétisations dans des lieux publics, dans l'espace ou dans des foyers et des hôpitaux. Pour assurer leur bon fonctionnement dans ces environnements critiques, les systèmes autonomes intègrent de plus en plus des techniques spécifiques visant à éviter ou tolérer les aléas et les fautes. Nous présentons dans cette partie un état de l'art des différentes techniques utilisées actuellement pour le développement des systèmes autonomes, d'abord du point de vue de la robustesse, puis suivant les quatre différents moyens de la sûreté de fonctionnement.

1.3.1 État de l'art en robustesse

La robustesse des systèmes autonomes peut être assurée de deux façons différentes :

- soit par un traitement implicite des situations adverses dues aux incertitudes de l'environnement, par une *observation* de l'état du système et de l'environnement dans leur ensemble, suivie d'un *choix* sur les actions à entreprendre,
- soit par un traitement explicite de situations particulières, par une *détection* à partir des perceptions du système suivie de *traitements* appropriés.

1.3.1.1 Observation et choix

Le traitement implicite par observation et choix se rapproche du principe du masquage d'erreurs de la tolérance aux fautes, dans le sens où on compte sur la combinatoire d'actions possibles par le système pour tolérer des situations adverses imprévues qui empêchent un sous-ensemble de ces actions. Il est mis en œuvre principalement par des mécanismes de planification et d'apprentissage.

La fonction de *planification* permet à un système autonome de choisir une suite d'actions à entreprendre pour réaliser un ou plusieurs objectifs, en fonction de l'état actuel du système et de son environnement. Elle apporte ainsi un traitement implicite des situations dues aux incertitudes de l'environnement, d'une part en couvrant un ensemble de situations qui auraient été extrêmement fastidieux (voire impossible) à programmer de façon impérative, et d'autre part en permettant de tolérer des situations qui n'ont pas été envisagées lors de sa conception mais qui sont compatibles avec ses connaissances.

La *planification de moindre engagement* permet également de contraindre au minimum les valeurs des variables temporelles et non-temporelles ; ces variables sont fixées pendant l'exécution du plan, mais le plus tard possible. Le plan obtenu possède ainsi une plus grande flexibilité dans les échéances temporelles et la consommation de ressources, permettant de tolérer des situations adverses qui n'étaient pas connues lors de la génération du plan, ou qui surviennent pendant celle-ci. Le planificateur IxTeT de l'architecture LAAS [Lemai-Chenevier 2004] et le planificateur des architectures RAX

[Muscettola et al. 1998] et IDEA [Muscettola et al. 2002] font de la planification de moindre engagement.

Exécuté hors-ligne, l'*apprentissage* fournit, au même titre que la planification, un traitement implicite des situations adverses dues aux incertitudes de l'environnement, en permettant de couvrir un plus grand nombre de situations qu'une programmation impérative. En particulier, il permet de générer des modèles, appris à partir des situations décrites dans les spécifications du système, qui peuvent rester valides face à des situations imprévues proches des situations spécifiées.

1.3.1.2 Détection et traitement

Le traitement explicite d'une situation par détection et traitement se rapproche du principe de la reprise d'erreurs, dans le sens où elle consiste à traiter une situation adverse que l'on a préalablement détectée. Cette technique de robustesse est ainsi mise en œuvre en deux temps : détection puis traitement.

Détection

Dans un système autonome, la détection d'une situation adverse due aux incertitudes de l'environnement est le plus souvent réalisée par des mécanismes de contrôle d'exécution, mais elle peut également être mise en œuvre par des mécanismes de reconnaissance de situations et de diagnostic.

Un des objectifs du *contrôle d'exécution* est de superviser l'exécution des plans que doit réaliser le système autonome. Un plan est généralement développé en partant du principe que chaque action s'exécutera correctement ; l'échec d'une action du plan représente ainsi l'apparition d'une situation adverse qui n'a pas été prise en compte par le planificateur. Cet échec peut être une erreur reportée par le niveau fonctionnel, le dépassement de l'éventuelle échéance temporelle de l'action, ou un manque de ressource pour réaliser la fonction du système. Si un signalement d'erreur a effectivement été activé, des mécanismes de tolérance aux fautes peuvent être utilisés pour traiter le composant fonctionnel erroné. Dans le cas contraire, la situation adverse qui a causé cet échec est traitée par des mécanismes de robustesse, qu'elle soit due à une faute non-spécifiée ou aux incertitudes de l'environnement. Le contrôle d'exécution est employé dans les architectures LAAS, RAX, CIRCA et IDEA comme principale méthode pour détecter une situation adverse.

Des actions devenues erronées par l'occurrence d'évènements inattendus ou la complexité des interactions entre composants fonctionnels peuvent aussi être détectées avant qu'elles ne soient exécutées : ainsi, le système ne perd pas de temps d'exécution, et ne court pas les risques de défaillance liés à cette erreur. Cette détection est réalisée à travers un contrôle par test d'assertions ou de propriétés, établies par le développeur à partir des spécifications du système. Un tel mécanisme est réalisé dans l'architecture LAAS par le composant R2C [Py & Ingrand 2004], dont le principe s'approche d'un composant indépendant de sécurité utilisé en tolérance aux fautes. Après la détection d'une requête d'action erronée, le R2C interdit l'exécution de cette requête, ou arrête les activités qui l'invalidaient, ce qui entraîne une réparation de plan ou une replanification.

À partir d'une chronique des évènements observés par le système, la *reconnaissance de situations* cherche à identifier la situation du système et de son environne-

ment parmi plusieurs modèles génériques de situations, spécifiés ou appris lors de la conception. Après cette reconnaissance, un traitement peut être effectué. Des travaux et des applications sur la reconnaissance de situation peuvent être trouvés dans [Dousson 1994] et [Despouys 2000] ; néanmoins, ce type de détection reste peu utilisé dans les systèmes autonomes. On peut noter que la reconnaissance de situations peut aussi s'appliquer à la détection d'erreurs.

La fonction première du *diagnostic* est d'identifier et localiser les fautes dans le système (voir plus loin, paragraphe 1.3.2.3). Cependant, les outils de diagnostic peuvent aussi servir à déterminer plus généralement la situation dans laquelle se trouve le système. Par exemple dans [Morisset et al. 2004], un MDP est appris au système pour diagnostiquer le type de situation où il se trouve, en faisant correspondre des variables d'état observables à un type de situation.

Traitement

Le traitement d'une situation adverse préalablement détectée peut être mis en œuvre par des mécanismes de planification, de contrôle d'exécution, et d'apprentissage.

Les mécanismes de *planification* qui permettent le traitement d'une situation adverse détectée sont la replanification, la réparation de plan, et la prise en compte de situations adverses spécifiées.

- Le procédé de *replanification* a lieu après détection d'un échec du plan par le contrôle d'exécution. Il peut également avoir lieu à intervalle temporel régulier (appelé *horizon* de la planification), auquel cas il se rapproche du traitement implicite des situations adverses. La replanification consiste à développer un nouveau plan à partir de la situation actuelle et des objectifs restant à remplir par le système, après, si besoin est, s'être mis dans un état sûr. Le planificateur IxTeT utilisé dans l'architecture LAAS [Lemai-Chenevier 2004] et le planificateur de l'architecture CIRCA [Goldman et al. 1997] procèdent par replanification.
- Une *réparation de plan* peut être tentée avant une replanification, après la détection d'un échec du plan. Elle consiste à développer un nouveau plan qui conserve une partie du plan précédent, en effectuant des retours-arrière du planificateur pour éliminer les actions qui sont devenues impossibles à exécuter. Si une réparation de plan est jugée impossible (par exemple si elle prend trop de temps), une replanification est réalisée. Une réparation de plan a les mêmes objectifs qu'une replanification, mais cherche à atteindre plus rapidement une solution en partant du principe qu'une partie du plan précédemment développé peut encore être utilisée ; elle permet également au système d'exécuter cette partie du plan pendant que la réparation s'effectue. Le planificateur temporel IxTeT utilisé dans l'architecture LAAS utilise la réparation de plan [Lemai & Ingrand 2004].
- La *prise en compte de situations adverses spécifiées* consiste à identifier lors de la conception un ensemble de situations adverses qui peuvent porter atteinte aux fonctions du système, que le planificateur cherchera à éviter en priorité par rapport aux objectifs qu'on lui demande de remplir. Elle est utilisée dans le niveau décisionnel de l'architecture CIRCA, qui cherche lors du développement d'un plan à préempter par des actions du système toutes les transitions vers les états représentant ces situations, empêchant théoriquement

leur activation [Goldman et al. 1997]. Bien qu'appliqué à des situations dues aux incertitudes de l'environnement (couverture radar, présence ennemie), ce traitement pourrait éventuellement être appliqué vis-à-vis de certaines classes de fautes de plate-forme.

Vis-à-vis du traitement d'une situation adverse détectée, le *contrôle d'exécution* peut apporter deux mécanismes de traitement explicite, tous deux basés sur la possiblité de réaliser une action de plusieurs façons différentes : le mécanisme de reprise d'action, et la mise en œuvre de modalités.

- Le mécanisme de *reprise d'action* part du principe qu'une action du planificateur peut être décomposée par le contrôle d'exécution en plusieurs ensembles de tâches fonctionnellement diversifiés et ordonnés. Ainsi, lorsque l'échec d'une action du plan est détecté, le contrôle d'exécution peut sélectionner d'autres décompositions, jusqu'à ce que l'action soit effectuée ou que toutes les décompositions aient échoué, auquel cas l'action est reportée comme échouée au planificateur.
- La mise en œuvre de *modalités* étend le mécanisme de reprise d'action : il part du principe que chaque ensemble de tâches (appelé modalité) est plus adapté à un type de situations donné. Le contrôle d'exécution sélectionne la modalité la plus performante en fonction de la situation du système, et change de modalité quand l'échec de l'action a été détecté, ou si une autre modalité est plus adaptée à la nouvelle situation du système. Dans [Morisset et al. 2004], le principe de modalités est appliqué à la localisation et la navigation d'un robot : de nombreuses techniques sont possibles (par exemple odométrie, stéréo-odométrie et GPS pour la localisation, planification de trajectoire ou navigation réactive pour la navigation), le choix étant réalisé par un mécanisme d'apprentissage après avoir identifié le type de situation où évolue le système (d'après plusieurs critères, comme la confiance dans la position supposée du robot, ou le type d'environnement). Ranganathan et Koenig [Ranganathan & Koenig 2003] présentent un basculement entre plusieurs modes de navigation de nature similaire aux modalités : un système autonome utilise un mode de navigation réactive pour atteindre un but, mais bascule sur un mode de navigation par planification, qui demande plus de temps et de ressources, lorsqu'il est bloqué dans un minimum local (c'est-à-dire, lorsqu'il n'est pas capable de progresser vers l'objectif pendant un intervalle de temps donné).

Les méthodes de reprise d'action et de mise en œuvre de modalités nous semblent également pertinentes pour effectuer du recouvrement d'erreur par blocs de recouvrement.

Exécuté pendant l'opération du système, le mécanisme d'*apprentissage* apporte également un traitement explicite d'une situation adverse imprévue, à travers l'existence de plusieurs choix et la possibilité d'en essayer un autre en cas d'échec. La distinction entre bon ou mauvais comportement d'une exécution, utilisée pour guider l'apprentissage, est généralement réalisée par une fonction de récompense, qui évalue les performances de l'exécution par rapport à des critères spécifiées (par exemple, pour un processus de navigation, la vitesse de déplacement, l'occurrence de chocs avec des obstacles, la progression par rapport à la position de l'objectif...).

1.3.2 État de l'art en sûreté de fonctionnement

Nous avons regroupé les différents mécanismes de sûreté de fonctionnement selon les quatre moyens de ce domaine, présentés dans la Section 1.1.1 : prévention des fautes, élimination des fautes, tolérance aux fautes, et prévision des fautes.

1.3.2.1 Prévention des fautes

Les principales techniques de prévention des fautes identifiées dans les systèmes autonomes sont la modularité des composants logiciels et l'utilisation d'outils de conception, qui visent toutes deux à prévenir les fautes de développement.

La *modularité des composants logiciels* est motivée par trois intentions : décomposer un système complexe en plusieurs composants logiciels plus simples et dissociés, capitaliser des bibliothèques d'algorithmes ou de structures génériques, et faciliter la mise en œuvre de communications et d'ordonnancements inter ou intra composants. Du point de vue de la sûreté de fonctionnement, cette décomposition rend d'une part plus aisé le développement de chaque composant, tout en demandant un effort final d'intégration, et permet d'autre part, dans une certaine mesure, de considérer les composants individuellement (par exemple pour leur développement ou leur test).

Cette modularité peut se retrouver dans l'architecture générale d'un système autonome (par exemple le style d'architecture hiérarchisé des architectures LAAS, CIRCA, RAX et CLARAty, ou le style d'architecture multi-agents de IDEA), ainsi qu'à l'intérieur de chaque élément de cette architecture (modules du niveau fonctionnel de l'architecture LAAS et CLARAty, niveau décisionnel de l'architecture LAAS, CIRCA et RAX).

L'utilisation d'*outils de conception* et de génération automatique de code permet d'une part d'accélérer le processus de développement du système, et d'autre part d'éviter que des fautes ne se produisent lors de l'écriture manuelle de la tâche automatisée. Ils contiennent généralement des bibliothèques de composants génériques, qui peuvent être instanciés pour s'adapter à un algorithme ou un support spécifique.

Actuellement les outils de conception sont utilisés en particulier pour le niveau fonctionnel des systèmes autonomes, avec par exemple l'outil GenoM de l'architecture LAAS [Alami et al. 1998], et les environnements ControlShell [Schneider et al. 1998], ORCCAD [Borrelly et al. 1998] et SIGNAL [Marchand et al. 1998].

1.3.2.2 Élimination des fautes

Les principaux mécanismes d'élimination des fautes identifiés dans le cadre des systèmes autonomes sont le test (en simulation et en opération), et la vérification formelle.

En pratique, des *tests* en simulation et en opération sont incontournables dans le développement des systèmes autonomes ; ces tests peuvent cibler un composant particulier, comme un mécanisme décisionnel ou un composant du niveau fonctionnel (test unitaire), ou l'ensemble du système. Cependant, dans le cas de plates-formes de recherche, ils sont généralement réalisés plus dans une optique

de déverminage que d'une véritable validation : le développeur vérifie simplement que le système ou le composant fonctionne dans quelques situations.

Des tests en simulation intensifs ont néanmoins été effectués pour le projet DS1 sur l'architecture RAX [Bernard et al. 2000] [Feather & Smith 1999]. Six bancs de test ont été réalisés tout au long du développement, ciblant différentes parties du système ou le système dans son ensemble ; en tout, près de 600 tests ont été effectués. [Bernard et al. 2000] souligne l'importance de ces tests intensifs, et remarque des difficultés posées au test par les systèmes autonomes, notamment le problème de l'oracle[9].

La *vérification formelle* peut être utilisée pour vérifier certaines propriétés des composants logiciels, par exemple des propriétés de sécurité que l'on veut assurer. Cependant elle demande généralement des conditions rigoureuses sur la conception du système, telles que le synchronisme des communications, ou l'utilisation d'un certain type de langage, et doit être prévue au plus tôt dans la conception.

Des techniques de vérification formelle sont mises en œuvre dans les environnements ORCCAD [Borrelly et al. 1998] et SIGNAL [Marchand et al. 1998].

1.3.2.3 Tolérance aux fautes

La tolérance aux fautes est rarement explicitement mentionnée dans la littérature sur les systèmes autonomes. Bien que certaines techniques soient assez répandues (comme la mise dans un état sûr en cas de collision avec un obstacle, ou le contrôle temporel par chien de garde), nous pensons que sa mise en œuvre est loin d'être systématisée. Ceci s'explique en partie par le fait que de nombreux systèmes autonomes sont encore des plates-formes de recherche, et que les travaux sont principalement menés sur leurs fonctionnalités. Nous présentons ici successivement les mécanismes de détection d'erreur et de rétablissement du système que nous avons identifiés dans les systèmes autonomes.

Détection d'erreur

Les méthodes de détection d'erreur identifiées dans les systèmes autonomes sont les techniques de contrôle temporel par chien de garde, de contrôle de vraisemblance, et par diagnostic.

Le *contrôle temporel par chien de garde* est mis en œuvre dans le système autonome RoboX, présenté dans [Tomatis et al. 2003]. Il vérifie que certaines fonctions critiques pour la sécurité-innocuité du système ne se sont pas arrêtées inopinément, ou respectent des contraintes temporelles ; dans RoboX, ces fonctions sont le superviseur de vitesse, l'évitement d'obstacle, les capteurs des pare-chocs, et les capteurs lasers. De par son principe, il ne permet de détecter que des erreurs temporelles, et non des erreurs en valeur. Bien que nous ne disposons pas d'autres exemples détaillés, nous supposons que l'utilisation de chiens de garde est courante dans les systèmes autonomes : c'est en tout cas le cas dans la robotique médicale, par exemple avec les robots non-autonomes Hippocrate et SCALPP [Duchemin et al. 2004].

[9] Comment décider de l'exactitude des résultats fournis par le programme en réponse aux entrées de test ? Nous évoquons cette problématique plus en détail dans le chapitre suivant.

Dans des systèmes autonomes, le *contrôle de vraisemblance* est souvent utilisé en effectuant un test sur un intervalle de valeurs. De tels tests sont mis en place dans le système autonome RoboX pour contrôler la vitesse maximale du système, et les robots médicaux non-autonomes Hippocrate et SCALPP ; sans disposer d'autres exemples nous supposons que leur utilisation est courante dans les systèmes autonomes. Les tests de vraisemblance permettent de détecter des erreurs en valeur. Dans le principe, les contrôles par tests d'assertions ou de propriétés de composant R2C de l'architecture LAAS [Py & Ingrand 2004] peuvent permettre de détecter des erreurs dues à des fautes de conception, en plus d'assurer la robustesse du système face à la complexité de son environnement.

Le *diagnostic* est dans la pratique majoritairement utilisé pour détecter des fautes sur les composants physiques du système (pneu crevé, moteur en sous régime), plutôt que des fautes logicielles. La comparaison du comportement du système avec un modèle mathématique est par exemple utilisée dans le composant MIR de l'architecture RAX [Muscettola et al. 1998]. Elle consiste à comparer les valeurs des capteurs du système avec des valeurs théoriques déterminées par un modèle du système et de son environnement. Un écart significatif entre la valeur réelle et la valeur théorique indique la présence d'une erreur dans le système. Cette détection est suivie d'un diagnostic de fautes, puis d'une reconfiguration.

Rétablissement du système

Les méthodes de rétablissement d'erreur identifiées dans les systèmes autonomes sont le confinement d'erreur, la mise en état sûr, et la reconfiguration, matérielle ou logicielle, du système.

La *mise dans un état sûr* d'un système autonome, une forme de recouvrement par poursuite, peut se faire pendant une replanification après l'échec d'un plan (architectures LAAS [Lemai-Chenevier 2004] et RAX [Muscettola et al. 1998]), ou après défaillance d'un composant critique. C'est le cas par exemple pour le système autonome RoboX qui se met dans un état sûr en cas de défaillance d'un de ses sous-systèmes critiques, ou du robot autonome Care-O-Bot [Graf et al. 2004] qui s'arrête en cas de collision avec un objet ou individu.

Une *reconfiguration matérielle* peut être réalisée dans l'architecture RAX en utilisant la diversification fonctionnelle des composants de la couche matérielle : après détection d'une défaillance matérielle, le composant MIR cherche une nouvelle configuration du système permettant de remplir les fonctions du composant défaillant [Muscettola et al. 1998].

Un système autonome peut effectuer une *reconfiguration logicielle* soit en effectuant un basculement entre plusieurs modes de fonctionnement, soit par une mise à jour sous forme de rustine (ou *patch*). Le basculement entre plusieurs modes de fonctionnement peut utiliser les mêmes mécanismes que les modalités décrites dans la Section 1.3.1.2. Le changement de modalité peut en effet s'effectuer soit lorsque les conditions de l'environnement ne sont pas appropriées pour la modalité courante (robustesse), soit si la modalité échoue à cause d'une défaillance du système (tolérance aux fautes) ; cette défaillance peut par exemple être causée par une faute dans le logiciel mettant en œuvre le mode considéré, ou une défaillance d'un capteur ou actionneur nécessaire pour l'exécution du mode. Ce basculement peut également être déclenché lors de traitements d'exception, par exemple dans le système autonome RoboX [Tomatis et al. 2003]. Une mise à jour sous forme de

rustine, requérant un travail soutenu des opérateurs au sol, a été utilisée pour traiter les défaillances des robots d'exploration martiens *Pathfinder*[10] et *Spirit*[11].

⋆ ⋆ ⋆

Citons enfin le *confinement d'erreur* par partitionnement, mis en œuvre dans l'architecture RoboX [Tomatis et al. 2003]. Il consiste en l'utilisation de processeurs dédiés selon deux niveaux de criticité : un processeur est utilisé pour les tâches les plus critiques (localisation et déplacement), tandis qu'un autre processeur est utilisé pour les tâches d'interaction, qui sont moins fondamentales à la sécurité-innocuité du système et utilisent des composants logiciels sur étagère dans lesquels les développeurs n'accordent pas une très grande confiance. Cette technique peut être considéré comme de la tolérance aux fautes, puisque les erreurs éventuelles dues à des fautes de conception dans la partie non-critique du système n'affectent pas la partie critique du système.

1.3.2.4 Prévision des fautes

Nous avons identifié peu de travaux en prévision des fautes dans les systèmes autonomes. [Carlson & Murphy 2003] et [Tomatis et al. 2003] sont deux études de fiabilité, qui ne donnent pas des résultats très encourageants. La première étude a été menée pendant deux ans sur treize robots de recherche de sept modèles différents, certains circulant à l'extérieur, et donne un MTBF (*Mean Time Between Failure*, c'est à dire le temps moyen entre deux défaillances) de 8 heures. La seconde étude, a été menée pendant cinq mois sur une dizaine de robots autonomes d'un même modèle, assistant les visiteurs à l'intérieur d'un musée, et donne un MTBF de 4,6 heures. Ces deux études montrent par ailleurs que les fautes de logiciel occupent une place significative dans les causes de défaillance : dans la première étude 30% des défaillances sont dues à des fautes logicielles, tandis que, dans la seconde étude, plus de 90% des défaillances critiques du système autonome (c'est-à-dire des défaillances dont le rétablissement nécessite une intervention humaine) sont dues aux plates-formes logicielles.

1.4 Conclusion

Nous avons présenté dans ce chapitre les notions relatives aux deux domaines liés à nos travaux : la sûreté de fonctionnement informatique, et les systèmes autonomes. Nous avons également présenté un état de l'art sur les mécanismes de robustesse et de sûreté de fonctionnement appliqués aux systèmes autonomes. Les mécanismes de robustesse, essentiels à l'autonomie, sont couramment utilisés, et

[10] En 1997, le robot *Pathfinder* a été confronté a une faute de gestion de priorité, qui l'amenait à se réinitialiser lorsqu'il devait effectuer une certaine action [Reeves 1997].

[11] Le 21 janvier 2004, le robot *Spirit* cessa de répondre aux commandes, se réinitialisant incessamment. Cette défaillance était due à une mauvaise gestion de la mémoire vive, qui se remplissait petit à petit à chaque saisie de données, jusqu'à ce que le robot n'en ait plus assez pour fonctionner correctement. Un récit informel de l'observation et du traitement de cette défaillance par des opérateurs au sol peut être trouvé sur http ://www.planetary.org/blog/article/00000702/.

sujets de nombreux travaux de recherche à l'heure actuelle. En ce qui concerne la sûreté de fonctionnement, qui nous intéresse particulièrement dans ce mémoire, nous établissons l'analyse suivante :

- D'une manière générale, il nous semble qu'il manque dans le développement des systèmes autonomes une approche holistique à la sûreté de fonctionnement à travers l'utilisation combinée de ses quatre moyens. Actuellement, parmi ces moyens, la prévention des fautes est largement privilégiée.
- La mise en œuvre de la tolérance aux fautes est focalisée sur les capteurs et les actionneurs. Les études menées en prévision des fautes montrent cependant que les fautes affectant les processeurs et les composants logiciels sont également à considérer.
- On trouve peu de techniques visant explicitement la tolérance aux fautes de développement. Les techniques de robustesse apportent une contribution certaine à la sûreté de fonctionnement, mais sûrement insuffisante pour des applications critiques.
- Le domaine des systèmes autonomes n'a pas encore de véritable culture de mesures, ni en terme de performances temporelles, ni en terme de sûreté de fonctionnement ; nous avons par exemple identifié peu de travaux relatifs à la prévision des fautes.

Dans le chapitre suivant, nous utilisons les notions et informations précédemment introduites pour étudier différentes méthodes ou architectures susceptibles d'améliorer la sûreté de fonctionnement dans les systèmes autonomes. Nous détaillons également le cadre de la contribution que nous proposons dans ce domaine, et qui sera ensuite précisée dans les chapitres suivants.

Ce qu'il faut retenir :

1. 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.
2. La *tolérance aux fautes* est le moyen de la sûreté de fonctionnement qui cherche à maintenir un service correct délivré par le système, malgré la présence de fautes. Sa mise en œuvre passe par la détection d'erreur et le rétablissement du système, qui nécessitent tous deux de la redondance dans le système.
3. L'*autonomie* est la capacité d'un système à percevoir, analyser, communiquer, planifier, établir des décisions, et agir, afin d'atteindre des objectifs assignés par un opérateur humain, malgré les incertitudes liées à l'environnement.
4. Un *mécanisme décisionnel* est constitué d'un mécanisme d'inférence utilisé pour résoudre différents types de problèmes, et de connaissances spécifiques à l'application du système.
5. La *robustesse* se distingue de la tolérance aux fautes par les situations adverses que chacune cherche à tolérer : la robustesse caractérise la tolérance vis-à-vis des incertitudes de l'environnement, tandis que la tolérance aux fautes s'attache aux fautes qui affectent les ressources matérielles et logicielles du système.

6. On distingue trois principaux types d'architecture pour les systèmes autonomes : subsomptif, hiérarchisé, et multi-agents. Le type d'architecture *hiérarchisé* et actuellement le type d'architecture le plus utilisé dans les systèmes autonomes complexes.
7. Dans les systèmes autonomes, la prévention des fautes est largement privilégiée par rapport aux autres moyens de la sûreté de fonctionnement, même si son utilisation est encore loin d'être systématique. De plus, les fautes de développement ne sont que peu considérées dans les mécanismes de tolérance aux fautes employés, et bien que la robustesse des systèmes autonomes peut dans une certaine mesure compenser ce problème, le besoin en tolérance aux fautes est réel pour des applications critiques.

Chapitre 2

Architectures et méthodes pour une autonomie sûre de fonctionnement

Après l'étude et l'analyse de l'état de l'art, nous avons identifié plusieurs pistes d'étude pour améliorer l'autonomie et la sûreté de fonctionnement des systèmes autonomes. Nous présentons dans ce chapitre le résultat de ces recherches, avant de préciser en conclusion l'angle que nous avons choisi de privilégier dans la suite de nos travaux.

Nous développons tout d'abord l'aspect architectural, en présentant plusieurs observations sur les types d'architecture hiérarchisé et multi-agents. Nous nous attardons ensuite sur le concept de composant indépendant de sécurité, qui nous semble particulièrement prometteur pour faire respecter des consignes de sécurité-innocuité. Nous revenons ensuite sur deux fonctionnalités particulièrement importantes dans les systèmes autonomes : le contrôle d'exécution et la planification, en détaillant certaines de leurs caractéristiques du point de vue de la sûreté de fonctionnement.

2.1 Aspect architectural

Nous avons présenté dans le chapitre précédent les trois principaux types d'architecture employés dans la conception de systèmes autonomes. Le type d'architecture hiérarchisé est le plus couramment utilisé à l'heure actuelle : on le retrouve sur un satellite [Muscettola et al. 1998], des robots d'exploration spatiale [Nesnas et al. 2003], des guides de musée [Tomatis et al. 2003], et de nombreuses plates-formes de recherche. Il permet d'appréhender efficacement la complexité des problèmes posés par l'autonomie, en décomposant les modèles et les activités du système suivant différents niveaux d'abstraction. Il offre aussi simultanément des fonctions délibératives et réactives, les premières offertes par les niveaux supérieurs, alors que les secondes sont implantées dans les actions de navigation ou de contrôle du matériel du niveau fonctionnel.

Cependant, ce type d'architecture possède aussi certaines limites intrinsèques, que la communauté robotique a essayé d'aborder. Nous présentons dans une première section les trois désavantages majeurs du type d'architecture hiérarchisé ainsi que divers mécanismes proposés pour y remédier, avant d'introduire une solution architecturale alternative : les systèmes multi-agents.

2.1.1 Les limites du type d'architecture hiérarchisé

Trois reproches majeurs peuvent être imputés au type d'architecture hiérarchisé : un risque constant de désaccords entre les différents niveaux de l'architecture, une lenteur de réaction face à certaines situations adverses, et la criticité du mécanisme décisionnel principal.

2.1.1.1 Risque de désaccords

La structure en divers niveaux d'abstraction, qui permet une décomposition successive des tâches à effectuer, et un traitement des différents niveaux par des logiciels spécialisés, est une source importante de *désaccords*[1].

Les désaccords sémantiques sont particulièrement courants, puisque les différents niveaux d'abstraction n'utilisent pas le même modèle de représentation du monde, ni la même granularité dans cette représentation. Du point de vue temporel, par exemple, le niveau décisionnel peut avoir une granularité et une précision de l'ordre de la seconde, tandis que les niveaux exécutif et fonctionnel peuvent descendre en-dessous de la milliseconde ; des conditions temporelles (telles que la durée d'une fenêtre de communication) peuvent ainsi être jugées correctes par le niveau décisionnel, alors que la plus grande précision des niveaux inférieurs les considère incorrectes. Dans d'autres cas, l'échec d'une tâche peut ne pas être traité ou diagnostiqué correctement, parce que les informations nécessaires pour ces actions sont réparties dans les différents niveaux, aucun ne disposant de leur intégralité. Par exemple, le niveau décisionnel d'un robot peut ne pas enregistrer les positions des obstacles fournies par la navigation et la localisation, parce que ces positions sont généralement superflues pour ses plans de haut niveau ; le système est alors incapable de détecter si une destination est accessible ou non. Des désaccords syntaxiques ou temporels risquent également de survenir entre les composants logiciels des différents niveaux : un planificateur peut avoir enregistré dans sa base de données qu'un moteur du système était en fonctionnement, tandis que l'exécutif le considère arrêté.

L'architecture CLARAty [Volpe et al. 2000] propose de rassembler les niveaux décisionnel et exécutif, ce qui permettrait en théorie de limiter les désaccords sémantiques, syntaxiques et temporels entre ces deux niveaux : ils partageraient en effet les mêmes modèles, formalismes et structures de données. Cette solution impose cependant des restrictions significatives sur le mécanisme décisionnel employé, qui doit être capable d'accomplir à la fois des fonctions de planification et de contrôle d'exécution, et dans la pratique, ces deux fonctions sont encore implantées par deux composants séparés.

[1] voir Section 1.2.2.3

Un autre moyen de limiter l'impact des désaccords sémantiques, proposé dans [Py 2005], se base sur l'utilisation d'un superviseur général du système proche d'un *composant indépendant de sécurité*, qui a pour but de rejeter des requêtes incompatibles avec l'état du système en regard d'un ensemble de règles préétablies. Ce superviseur dispose d'une vision globale du système, et peut ainsi, dans une certaine mesure, avoir une connaissance plus précise que le niveau décisionnel des actions entreprises et de la situation du système, et éviter l'exécution de mauvaises décisions du planificateur ayant pour cause des désaccords avec les niveaux inférieurs. L'efficacité de ce superviseur reste encore limitée par l'absence d'une réelle représentation de l'état du système, et le peu d'observations dont il dispose (uniquement les entrées et sorties des différents composants).

2.1.1.2 Lenteur de réaction face à certaines situations adverses

La lenteur des capacités de réaction est un des principaux reproches adressés au concept architectural classique Sense/Plan/Act, qui doit effectuer un cycle complet avant de pouvoir traiter un évènement particulier. Le type d'architecture hiérarchisé répond dans une certaine mesure à ce problème, en déléguant des capacités de réaction aux niveaux fonctionnel et exécutif, notamment à travers les fonctions de navigation et de localisation. Cependant, certaines situations adverses ne peuvent pas être traitées par la robustesse de ces mécanismes, et doivent être remontées de la couche matérielle jusqu'au niveau délibératif en traversant successivement toute l'architecture. Le traitement de la situation peut également être coûteux en terme de performance, par exemple lorsque la situation adverse provoque une replanification du système.

La lenteur de réaction d'un système hiérarchisé est traitée d'une part par la mise en place de mécanismes de robustesse et de reprise dans les niveaux inférieurs du système, et d'autre part par des techniques cherchant à produire le plus rapidement possible une sortie du mécanisme décisionnel.

Le premier type de solutions consiste d'une part à développer au niveau fonctionnel des composants robustes, capables de traiter diverses situations externes sans nécessiter d'appel aux niveaux supérieurs, et d'autre part à utiliser la diversification fonctionnelle du système pour essayer de faire exécuter une tâche échouée d'une façon différente par le niveau exécutif, par exemple à travers un changement de modalités [2].

Le second type de solutions comprend la planification *anytime*, et la réparation de plan. La planification *anytime* propose à tout moment un plan exécutable susceptible de faire avancer la résolution des objectifs du système, et d'autant plus fiable que le planificateur a eu de temps pour le produire. Ce type de planification permet de s'adapter à l'urgence d'une situation adverse, et de limiter le temps passé à planifier sans exécuter d'actions. On retrouve par exemple la planification *anytime* dans le planificateur CASPER [Knight et al. 2001], utilisé par la NASA dans des robots d'exploration. La réparation de plan cherche également à réduire le temps de planification, en réutilisant une partie du plan en échec, et en permettant au système d'exécuter cette partie du plan pendant que le mécanisme décisionnel travaille sur la partie échouée.

[2] Voir Section 1.3.1.2, et plus loin Section 2.3.1

2.1.1.3 Criticité du mécanisme décisionnel

La place primordiale du niveau décisionnel dans le type d'architecture hiérarchisé, fait du mécanisme décisionnel implanté *un composant critique du système*. Cependant, peu de travaux de sûreté de fonctionnement ciblent efficacement de tels mécanismes, ce qui limite leur utilisation, et par conséquent celle des systèmes autonomes. Dans le cas de la planification, qui est majoritairement utilisée dans le type d'architecture hiérarchisé, l'application directe des mécanismes de tolérance aux fautes des systèmes informatisés se heurte à plusieurs difficultés, détaillées plus loin dans la section 2.3.2.1.

À notre connaissance, seul le test intensif est utilisé pour répondre à ce problème. Il a été ainsi employé lors du développement du projet *Deep Space One* [Bernard et al. 2000].

2.1.2 Une alternative : le type d'architecture multi-agents

Le type d'architecture multi-agents, présenté dans le chapitre précédent, a été développé comme alternative au type d'architecture hiérarchisé, et propose des solutions à certains des problèmes évoqués ci-dessus. Le concept multi-agents a été mis en avant avec l'architecture IDEA [Muscettola et al. 2002], mais peu de systèmes l'ont actuellement mis en pratique, d'une part à cause de sa nouveauté, d'autre part parce que son implantation reste complexe.

Un agent de l'architecture IDEA, présenté dans la Figure 2.1, comporte cinq éléments :

- L'*enveloppe de communication* sert d'interface avec d'autres agents du système, et éventuellement avec la couche matérielle. Il n'y a pas de restriction sur la topologie des communications entre composants : par exemple deux composants peuvent se commander mutuellement.
- Le *modèle* contient la représentation du système et de son environnement du point de vue de l'agent. Chaque agent ne possède qu'une partie d'un modèle global unique, généralement trop complexe pour être traité par un seul planificateur.
- Le *planificateur* produit un plan à partir de l'état du système maintenu dans la base de données du plan, des objectifs fournis par les requêtes de l'utilisateur ou d'autres composants IDEA, et des contraintes et heuristiques fournies par le modèle. Suivant le composant, plusieurs types de planification peuvent être envisagées : une planification à court terme, une planification à long terme, ou une combinaison des deux. Dans ce dernier cas, le planificateur à long terme définit des plans flexibles (notamment temporellement) que le planificateur à court terme complète au fur et à mesure du déroulement du plan.
- L'*exécutif* met à jour la base de données du plan, active le planificateur à la réception d'une requête ou d'un bilan, et exécute la suite de requêtes demandées par le planificateur.
- La *base de données du plan* représente l'état actuel de la partie du système liée à l'activité de l'agent, ainsi que son évolution passée, et le comportement futur prévu par le planificateur. Elle est mise à jour par l'exécutif lors de

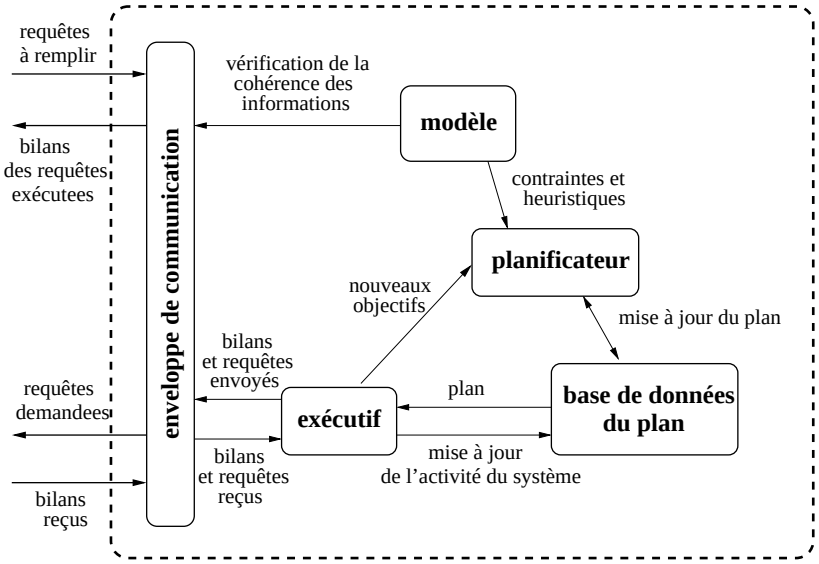

Figure 2.1: Agent IDEA

la réception de bilan, et le planificateur lorsqu'il développe ou complète un plan.

2.1.2.1 Réponses aux limites d'une architecture hiérarchisée

Dans l'architecture IDEA, les différents composants d'un agent partagent la même vision du système, contenue dans la base de données du plan, ce qui limite fortement les désaccords temporels, sémantiques ou syntaxiques à l'intérieur d'un même agent. De plus, les modèles des différents agents sont obtenus à partir d'un modèle global unique du système, ce qui réduit les chances de désaccord sémantique ou syntaxique entre agents. Des désaccords temporels entre agents sont par contre possibles, puisqu'il faut du temps aux bilans pour être remontés et traités par les différents composants de l'architecture qui sont concernés par une éventuelle évolution de l'environnement ou du système.

La réactivité des différents composants du système est plus importante que dans un type d'architecture trois niveaux. En effet, chaque agent, et les fonctions du système que chacun englobe, disposent de mécanismes décisionnels, ce qui assure une plus grande robustesse au plus tôt dans le système, en permettant de répondre plus rapidement à une situation adverse imprévue. Le temps de planification, fonction de l'horizon du planificateur, est de plus modulable dans une certaine mesure : les agents possédant un horizon court se retrouvent généralement près de la couche matérielle, pour permettre une plus grande réactivité.

En ce qui concerne la criticité du mécanisme décisionnel, l'architecture de type multi-agents n'offre pas de solution satisfaisante. En effet, si le mécanisme décisionnel n'est pas centralisé comme dans le type d'architecture hiérarchisé, il reste fondamental à chaque agent, et tout aussi vulnérable. La structure en multiples composants autonomes offre théoriquement des possibilités de confinement d'er-

reur et de reprise, mais une faute de conception dans les connaissances ou le mécanisme d'inférence d'un agent se retrouvera également dans les autres agents. Des fautes physiques peuvent être tolérées de cette manière, à la condition coûteuse que chaque agent s'exécute sur un processeur dédié. La mise en œuvre des mécanismes de détection d'erreur et de rétablissement se heurtent également aux différents problèmes de tolérance aux fautes particuliers aux mécanismes décisionnels[3].

2.1.2.2 Inconvénients spécifiques au type d'architecture multi-agents

En plus des problèmes liés à la criticité du mécanisme décisionnel, non-résolus dans ce type d'architecture, un système multi-agents nous semble soulever des difficultés particulières de conception et de validation.

Tout d'abord, le développement se focalise sur les fonctionnalités de chaque agent, définissant indirectement plutôt que directement les fonctionnalités du système. Les différentes étapes de conception du système (décomposition en agents, détermination du rôle de chaque agent, développement du modèle global) nous semblent également plus complexes et délicates que dans la conception d'un type d'architecture hiérarchisé. De plus, si le comportement d'un agent ou, à plus haut niveau, d'un groupe d'agents peut être en partie validé séparément par des envois de requêtes ou de bilan, la validité de la décomposition du système ne peut être testée qu'une fois le système complètement intégré, et une erreur dans cette décomposition sera très coûteuse pour le développement du système.

2.2 Composant indépendant de sécurité

Comme unité indépendante d'un système, chargé de faire respecter des consignes de sécurité-innocuité, le principe de *composant indépendant de sécurité* a connu plusieurs applications efficaces dans les systèmes informatisés, par exemple avec le système d'aiguillage ferroviaire automatisé ELEKTRA [Klein 1991], ou le système de surveillance SPIN pour centrale nucléaire [Guesnier et al. 1997]. Une définition est proposée dans [Guiochet & Powell 2006], et caractérise un *(sous-)système indépendant de sécurité*[4] comme étant :

- indépendant vis-à-vis du sous-système fonctionnel principal, c'est-à-dire qui possède son propre fil d'exécution et une mémoire protégée,
- dont la spécification est différente de celle du sous-système fonctionnel, notamment en ce qui concerne les règles de sécurité,
- qui surveille en permanence le système global (donc le sous-système fonctionnel), afin d'empêcher toute transition vers un état dangereux,
- qui, en cas de danger, amène le système global dans un état sûr, tel que défini par les concepteurs du système.

Nous nous intéressons dans cette section à l'application de ce type de composant aux systèmes autonomes, en présentant tout d'abord leurs caractéristiques, puis deux exemples d'implantation.

[3] voir plus loin, section 2.3.2.1.

[4] independent safety (sub)system

2.2.1 Caractéristiques

Un composant indépendant de sécurité est principalement caractérisé par ses règles de sécurité, et ses moyens d'observation et d'action. Tous deux dépendent à la fois de l'application envisagée, et du système dans lequel le composant est implanté.

2.2.1.1 Règles de sécurité

Un composant indépendant de sécurité peut permettre de tolérer différents types de situations adverses pour améliorer la sécurité-innocuité du système. Dans les systèmes autonomes, ces situations comportent :

- des désaccords entre différents composants du système, en s'assurant que des actions contradictoires ne sont pas entreprises par les composants du système (tolérance aux fautes),
- des fautes dans les mécanismes d'inférence ou les connaissances des mécanismes décisionnels, dues à la complexité et aux difficultés de validation des mécanismes décisionnels, (tolérance aux fautes),
- des situations adverses imprévues face auxquelles le système ne réagit pas correctement, dues à un environnement ouvert qui rend impossible un test exhaustif comprenant tous les contextes d'exécution (robustesse).

L'expression des règles de sécurité est fondamentale au composant indépendant de sécurité, et conditionne quelles situations adverses celui-ci permet de tolérer. Cependant, peu de travaux se sont concentrés sur la définition de ces règles, privilégiant plutôt la conception et la mise en place du composant de sécurité. À notre avis, le développement des règles de sécurité passe par différentes phases-clés, dont :

- la mise en place d'un modèle du système, représentant son état et son évolution à partir des informations accessibles au composant de sécurité,
- l'identification des situations adverses dangereuses ; cette identification est basée sur les spécifications du système, des outils et résultats de prévision des fautes, et le modèle précédemment établi,
- l'établissement des règles de sécurité à partir de l'étape précédente, et leur traduction dans un langage formel utilisable par le composant de sécurité.

Une étape de validation des règles formelles nous semble aussi indispensable, en utilisant le test et des méthodes formelles.

2.2.1.2 Observation et action

Afin de mettre à jour son modèle du système, et vérifier le respect des règles de sécurité, un composant indépendant de sécurité a besoin d'accéder à des informations sur l'état du système qu'il supervise.

Dans les systèmes informatisés, une approche *boîte noire* est souvent privilégiée, c'est-à-dire que les développeurs considèrent que le composant de sécurité ne doit pas accéder à l'état interne du système qu'il surveille, cet état étant potentiellement erroné, et ne doit baser ses observations que sur ses propres capteurs.

Une approche *boîte blanche* est aussi envisageable pour les systèmes autonomes. Elle part du principe que les capteurs et les mécanismes de communication

du système disposent de leurs propres mécanismes de tolérance aux fautes, et que le composant de sécurité a pour but principal de surveiller les composants logiciels décisionnels et exécutifs. Pour répondre à ses besoins d'observation, le composant de sécurité intercepte dans ce cas des paramètres internes du système, qui peuvent être de deux types différents :

- D'une part, des données de capteurs de la plate-forme matérielle, brutes ou déjà traitées par d'autres composants du système, permettent de connaître l'état du système dans son environnement (vitesse, présence d'obstacles, etc...).
- D'autre part, des données propres à l'exécution de la tâche du système, en particulier les commandes de lancement de requêtes, doivent être interceptées afin que le composant de sécurité détermine les intentions du mécanisme décisionnel, et stoppe une requête potentiellement dangereuse avant son exécution.

Une fois qu'une intention du mécanisme décisionnel ou une évolution dans l'état du système risque d'enfreindre une règle de sécurité, le composant de sécurité peut effectuer différents types de rétablissement, dont principalement :

- l'exécution d'une séquence d'actions dans le but de revenir dans un comportement sûr de fonctionnement, ultimement à travers une mise en état sûr du système,
- le rejet préventif d'une tâche demandée par le mécanisme décisionnel et qui risquerait d'enfreindre les règles de sécurité.

2.2.2 Applications dans les systèmes autonomes

Des composants indépendants de sécurité ont déjà été conçus pour des systèmes autonomes. Nous présentons ci-dessous l'exemple du R2C dans l'architecture LAAS [Py 2005], et le principe de *Guardian Agent* présenté dans [Fox & Das 2000].

2.2.2.1 Request and Report Checker

Le *Request and Report Checker*, ou R2C, est un composant développé pour l'architecture de type hiérarchisé LAAS, dont la structure est présentée sur la Figure 2.2.

Le R2C possède une base de données sur l'état du système, qu'il met à jour à partir de bilans et de requêtes interceptés entre le mécanisme décisionnel et les différents modules fonctionnels du robot. Cette base de données est utilisée pour vérifier si une nouvelle requête est valide, et déterminer l'action à suivre en cas d'infraction aux règles de sécurité : généralement l'abandon de la requête incriminée, ou l'arrêt d'une tâche contradictoire en cours d'exécution.

Intégré à l'architecture LAAS, le R2C vérifie en particulier le comportement correct des interactions entre modules du niveau fonctionnel, qui s'exécutent de façon asynchrone et concurrente. En ce sens, il cherche à tolérer des désaccords entre l'activité de différents modules, et implicitement des fautes dans le mécanisme décisionnel ou dans la programmation des modules qui pourraient en être responsables. D'autres règles de sécurité lui ont aussi été appliquées, comme le

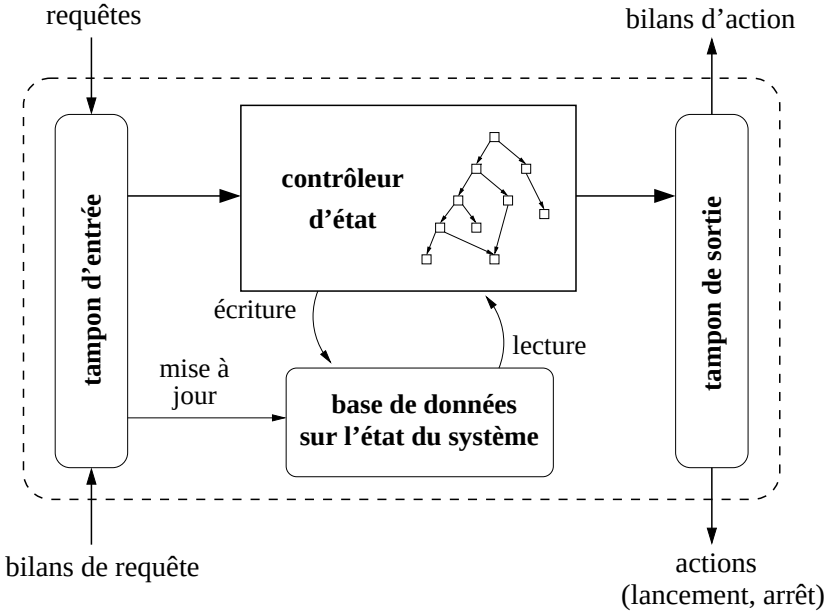

Figure 2.2: Composant R2C

contrôle de la vitesse maximale du robot, permettant de tolérer dans une certaine mesure des situations adverses imprévues (par exemple une collision avec un obstacle, un sol glissant, etc...) ou d'autres fautes de conception.

2.2.2.2 Guardian Agent

Le principe de *Guardian Agent* [Fox & Das 2000] essaie d'appliquer et d'étendre le concept de composant indépendant de sécurité aux mécanismes décisionnels, en particulier aux systèmes experts. Pour cela, les auteurs proposent un langage formel comportant des axiomes relatifs à la sécurité pour permettre une analyse et un contrôle pertinent des intentions et des actions du système, et éventuellement l'utilisation de mécanismes décisionnels pour obtenir une meilleure compréhension et réaction face aux situations adverses rencontrées.

Le langage proposé par les auteurs, nommé L_{Safe}, inclut par exemple des notions de permission, d'obligation ou d'autorisation sur l'exécution d'une action. Un exemple de proposition est : *L'action A doit être autorisée avant d'être permise, et l'action B est obligatoire avant que l'action A ne soit permise* ; elle signifie qu'avant que l'action A ne puisse être exécutée par le système, elle doit d'une part être autorisée par une autorité supérieure (un opérateur, ou un autre système), et d'autre part être précédée par l'exécution de l'action B. Les auteurs avancent aussi différents éléments de réflexion sur le processus d'identification et de construction des règles de sécurité.

Cependant, le *Guardian Agent* n'a pas été mis en pratique sur un système existant, et les difficultés soulevées par sa mise en œuvre n'ont pas encore été étudiées.

2.3 Mécanismes liés au contrôle d'exécution et à la planification

Deux fonctions des mécanismes décisionnels sont omniprésentes dans les systèmes autonomes complexes : le contrôle d'exécution et la planification. Dans cette section, nous étudions plus en détail les particularités de ces deux fonctions, en regard de l'autonomie et de la sûreté de fonctionnement.

2.3.1 Reprise de situations adverses par le contrôle d'exécution

Lors du premier chapitre, nous avons défini le contrôle d'exécution comme la supervision de l'exécution de plans de haut niveau, par la décomposition de chaque tâche du plan en séquence d'actions plus simples, et le contrôle de la bonne exécution de ces actions. Nous avons ensuite vu que cette fonction participait à la robustesse du système, et dans une moindre mesure à sa tolérance aux fautes, à travers la détection et la reprise de situations adverses. Nous reprenons plus en détail dans cette section les diverses possibilités de détection et de reprise, avant de nous intéresser au concept de modalités.

2.3.1.1 Détection d'une situation adverse

Typiquement, une situation adverse est détectée lorsque une tâche lancée par le contrôle d'exécution échoue. Cet échec peut être soit une erreur détectée par les composants fonctionnels, ensuite remontée au contrôleur sous la forme d'un message d'erreur, soit des informations de retour inattendues par le contrôleur, indiquant un état final insatisfaisant du système.

Dans ce dernier cas, des valeurs théoriques attendues de l'état du système sont comparées avec les informations remontées par les composants fonctionnels, une différence indiquant une situation adverse. Ces valeurs théoriques peuvent être codées en dur dans des branchements conditionnels des procédures du contrôleur, ou obtenues dynamiquement à partir d'un modèle du système mis à jour par le contrôleur, et indiquant le comportement nominal du niveau fonctionnel du système en fonction des tâches qui lui sont demandées. La comparaison peut également être effectuée sur des critères de valeur (par exemple, une position incorrecte du système à la fin d'un déplacement), ou des critères temporels (par exemple, l'action ne se termine pas avant la date limite originalement prévue). Cette tâche de détection peut aussi être mise en place par des mécanismes plus complexes de diagnostic.

2.3.1.2 Réaction par reprise

Une fois que l'échec d'une action a été détecté, plusieurs techniques de reprise peuvent être exécutées successivement par le contrôleur d'exécution, jusqu'à réussite de l'action ou épuisement des techniques de reprise mises en place :

- Un traitement particulier peut avoir été prévu pour la situation adverse détectée. Par définition, cette possibilité ne peut s'appliquer qu'à des situations adverses spécifiées, correctement identifiées. Elle peut ainsi être utili-

sée pour traiter des défaillances matérielles par reconfiguration du système, ou exécuter un traitement de certaines erreurs remontées par les composants fonctionnels.
- L'action échouée peut être relancée, selon le point de vue optimiste que le contexte d'exécution est plus favorable maintenant qu'à sa première exécution : par exemple un obstacle dynamique peut s'être déplacé, ou une tâche concurrente peut avoir fini son exécution. Cependant, il n'y a aucune garantie que la situation adverse détectée soit temporaire, ni qu'elle ait disparu.
- Une action alternative remplissant les mêmes buts peut être sélectionnée et exécutée par le contrôleur. Pour ne pas se heurter au problème précédent, il est préférable qu'elle soit le plus diversifié possible, et si possible qu'elle utilise des composants fonctionnels différents. Plusieurs alternatives peuvent exister pour la même action, et être exécutées successivement en cas d'échecs répétés, tant que le temps alloué pour l'action n'est pas dépassé.

En cas d'insuccès de ces différentes alternatives, ou de dépassement de la fenêtre temporelle allouée à l'exécution de la tâche de haut niveau, le contrôleur d'exécution remonte généralement l'échec au planificateur, qui essaie de trouver une nouvelle solution à partir de l'état courant du système.

2.3.1.3 Modalités

Les modalités forment un cas particulier d'actions alternatives. Elles cherchent principalement à tolérer la variabilité de l'environnement, en offrant différentes méthodes diversifiées pour réaliser un même but. Chacune de ces méthodes est plus performante que les autres dans un certain contexte d'exécution, et la robustesse du système est renforcée par leur utilisation complémentaire.

Par exemple, dans [Morisset 2002], cinq modalités de navigation sont développées. Un superviseur de navigation est également proposé, capable de sélectionner la modalité la plus adaptée à la situation actuelle du robot. Cette sélection est assurée par un MDP (processus de décision de Markov) à partir de différents observables, comme la dernière modalité utilisée, l'encombrement de la position du robot, ou la complexité géométrique du lieu.

Bien que le principe de modalité ait été développé pour améliorer la robustesse d'un système, nous pensons qu'elle peut également être adaptée à la tolérance aux fautes des composants fonctionnels : mettant en jeu différentes méthodes diversifiées, un basculement d'une modalité à une autre peut être utilisée pour tolérer des fautes physiques ou de conception dans les composants de la première modalité.

2.3.2 Planification et sûreté de fonctionnement

Les mécanismes de planification sont actuellement des composants centraux dans la plupart des systèmes autonomes complexes, au cœur du type d'architecture hiérarchisé. Ce sont eux qui offrent les facultés de décision les plus complexes et abstraites dans ces systèmes autonomes, et ils s'avèrent critiques à leur fonctionnement. Cependant, très peu de techniques de sûreté de fonctionnement leur sont adaptées, et le manque d'une réelle confiance dans leur comportement correct limite grandement leur utilisation.

Nous présentons ici les difficultés spécifiques liées à l'utilisation des méthodes traditionnelles de sûreté de fonctionnement dans la planification, ainsi que plusieurs travaux que nous avons identifiés dans ce domaine.

2.3.2.1 Difficultés spécifiques liées à la planification

L'application directe des techniques classiques de sûreté de fonctionnement aux mécanismes décisionnels, dont la planification, pose des problèmes spécifiques, liés aux caractéristiques de ces mécanismes. C'est en particulier le cas pour des techniques classiques d'élimination des fautes et de tolérance aux fautes.

En ce qui concerne l'*élimination des fautes*, des difficultés se posent en ce qui concerne le test et la validation des mécanismes de planification, d'une part à cause de la dimension ouverte de l'environnement d'exécution face auquel le système doit être validé, et d'autre part à travers le problème de l'*oracle*.

Dans les systèmes informatisés traditionnels, les contextes d'exécution sont déjà très nombreux, et il est rare de pouvoir en tester l'intégralité. La sélection et la représentativité des contextes d'exécution est un problème encore plus aigu pour les systèmes autonomes, qui sont par nature confrontés à un environnement ouvert qui n'est ni maîtrisable ni complètement connu, et dans lequel leurs actions ont des conséquences qui ne sont pas totalement prédictibles. Des tests intensifs sur les mécanismes de planification sont donc d'autant plus nécessaires qu'il n'est possible de couvrir qu'une petite partie des contextes d'exécution possibles. De plus, le mécanisme de planification ne peut pas être entièrement validé hors de l'architecture complète du système, puisqu'une partie conséquente de son rôle est de compenser les limitations des composants de niveau d'abstraction moins élevé par ses facultés d'abstraction et de décision. Des tests intégrés sont donc à réaliser le plus tôt possible, car ils peuvent faire apparaître des fautes de conception fondamentales dans les connaissances du mécanisme de planification, et obliger une nouvelle itération du cycle de conception.

Le problème de l'oracle est un problème classique posé par le test dans les systèmes informatisés : comment décider de l'exactitude des résultats fournis par le programme en réponse aux entrées de test ? Un dépouillement automatique et l'utilisation de simulations forment généralement une partie de la solution, mais dans le cas de la planification, le critère de jugement sur la correction du plan est particulièrement délicat. Tout d'abord, des plans totalement différents (par le séquencement et le choix des actions) peuvent être également corrects : ainsi la comparaison de la sortie du planificateur avec une sortie correcte n'est pas adaptée. De plus, le service offert par le système peut être dégradé à cause d'une situation adverse qui rend impossible une exécution totalement correcte : par exemple, le système ne remplit qu'une partie des déplacements demandés, parce que trop d'obstacles sont présents pour qu'il ait physiquement le temps de tous les remplir. Un tel comportement peut être jugé acceptable, même s'il n'est pas totalement correct vis-à-vis des objectifs établis a priori. Le jugement n'est alors plus dichotomique, mais dépend aussi de l'environnement d'exécution.

En ce qui concerne la *tolérance aux fautes*, certains de ces problèmes se retrouvent pour la détection d'erreurs et la reprise du système.

La détection d'erreurs se heurte au même problème que le test pour décider de

la qualité d'un plan : il est difficile de savoir avant exécution si ce plan est correct ou erroné, puisqu'on ne peut pas le comparer avec un plan produit par un planificateur redondant. D'une part, les planificateurs ne sont pas toujours déterministes, et deux planificateurs identiques peuvent produire des plans corrects mais différents. D'autre part, la diversification des connaissances, qui est nécessaire pour se prémunir des erreurs de conception, accentue par son principe même les divergences entre les plans redondants produits.

Ces problèmes de détection limitent les mécanismes de reprise qui peuvent être utilisés : les mécanismes de masquage d'erreur, par exemple, ne semblent pas adaptés puisqu'on ne dispose pas de critères de décision permettant de privilégier certaines sorties plutôt que d'autres. Outre le problème de détection d'erreur, le basculement sur un composant diversifié est aussi compliqué par la complexité de l'état interne des mécanismes de planification et les représentations différentes des connaissances, qui rendent très difficile le passage d'un mécanisme à l'autre pendant une planification ou l'exécution d'un plan. Nous essayons de traiter ce problème avec les mécanismes proposés dans le chapitre suivant.

Finalement, il faut ajouter que les techniques de *prévention des fautes* sont peu développées en ce qui concerne la conception des connaissances utilisées par les mécanismes décisionnels. Tout d'abord, les modèles utilisés par les planificateurs font appel à de la programmation déclarative plutôt que fonctionnelle, et ont donc une conception plus délicate et moins vérifiable formellement. Ils nécessitent également une profonde connaissance de l'application envisagée, et mobilisent ainsi souvent plusieurs concepteurs de domaines différents. Finalement, certaines connaissances comme les paramètres des heuristiques, ou certaines valeurs-seuils temporelles ou numériques des modèles sont souvent déterminées empiriquement, et modifiées petit à petit pendant la conception jusqu'à obtenir un comportement souhaitable dans les contextes d'exécution testés.

2.3.2.2 Méthodes existantes

En planification, la sûreté de fonctionnement est principalement mise en œuvre à travers deux méthodes d'élimination de fautes : test et analyse statique automatique ou manuelle.

Nous avons discuté précédemment du problème de l'oracle appliqué aux systèmes autonomes. Pour résoudre ce problème hors-ligne, un ensemble de contraintes temporelles et en valeur, qui est censé caractériser un plan correct de façon nécessaire et suffisante, peut être appliqué à la sortie du planificateur : un plan qui respecte cet ensemble de contraintes est alors jugé correct. Des tests intensifs ont été menés sur le mécanisme de planification de l'architecture RAX, en préparation au projet *Deep Space One* de la NASA [Bernard et al. 2000]. Les contraintes étaient alors emmagasinées sous la forme d'une base de données, générée par des études manuelles poussées sur le comportement du système et le modèle du planificateur [Feather & Smith 1999]. L'outil de validation de plan VAL [Howey et al. 2004] permet de valider des plans développés dans le langage PDDL[5]. Il cherche à détecter des invraisemblances dans le plan et l'ensemble de contraintes à partir d'une modélisation physique de l'évolution des variables de

[5] *Planning Domain Definition Language*, une initiative cherchant à normaliser les langages

l'environnement. Une extension présentée dans [Fox et al. 2005] étudie la robustesse d'un plan en insérant des erreurs temporelles sur ses actions, et en vérifiant si le plan reste correct malgré celles-ci. Aver [Bedrax-Weiss et al. 2005] est un outil de validation utilisé pour le développement de modèles du planificateur EUROPA, vérifiant un plan par rapport à différents jeux de contraintes saisis par l'utilisateur.

Lié au test, mais sans rapport avec l'oracle, [Howe 1995] propose un mécanisme d'analyse de reprise de défaillance (FRA, pour *Failure Recovery analysis*), dans le but de faciliter la modification de modèle ou de mécanismes de réparation du plan lors de tests de développement.

L'analyse statique automatique est utilisée pour exécuter différents contrôles sur les modèles de planification, tandis que l'analyse manuelle consiste en une étude détaillée de ces modèles par une équipe différente de celle qui les a développés. Planware [Becker & Smith 2005] est un environnement de développement qui propose ces deux types d'analyse : d'une part des outils de vérification syntaxique et structurelle, et d'autre part un processus de documentation sur les assertions et décisions prises par le développeur, consultable par des spécialistes du domaine d'application.

Nous avons également identifié quelques études en prévision des fautes. [Chen et al. 1995] propose des études sur le calcul de la fiabilité d'un planificateur, et compare leurs résultats théoriques avec des résultats expérimentaux. Ces études font ressortir un compromis nécessaire entre défaillances temporelles (liées à la calculabilité du planificateur) et défaillances en valeurs (liées à la justesse du planificateur). Dans [Chen 1997], l'auteur propose également l'exécution de plusieurs planificateurs sur des processeurs différents. Un superviseur collecte les plans générés et sélectionne ensuite le meilleur plan à exécuter dans les temps impartis. Le critère de sélection choisi est basé sur le type d'heuristique utilisé : une première heuristique, rapide mais produisant un plan comportant potentiellement des défauts, est choisi si la seconde heuristique, produisant un plan valide mais moins calculable que la première, ne fournit pas de sorties dans le temps imparti.

À part cette dernière proposition, nous n'avons identifié aucun mécanisme de tolérance aux fautes pour les planificateurs. Nous sommes cependant convaincus de leur nécessité, en regard des difficultés de développement des mécanismes d'inférence et des connaissances utilisés par les planificateurs, et des limites des techniques d'élimination des fautes.

2.4 Conclusion

Un des défis majeurs à la conception de systèmes autonomes est l'apparition des situations adverses imprévues. Bien qu'assurer un comportement acceptable du système dans de telles situations soit le rôle de la robustesse, la sûreté de fonctionnement doit cependant chercher à minimiser l'apparition de comportements inacceptables (fiabilité et disponibilité), voire dangereux (sécurité-innocuité).

En ce qui concerne ce dernier point, il nous semble que l'utilisation de composants indépendants de sécurité est fortement recommandée dans les systèmes au-

décrivant domaines et problèmes dans les planificateurs.

tonomes, car ce composant peut superviser l'activité générale du système et donc mieux appliquer des règles globales de sécurité. Des travaux restent cependant à faire en ce qui concerne la définition des règles de sécurité, et les moyens d'action sur le reste du système les plus efficaces pour les faire respecter.

Nous avons choisi de concentrer nos efforts sur la fiabilité des mécanismes décisionnels, et en particulier les mécanismes de planification, car ils s'avèrent fondamentaux et critiques dans les systèmes autonomes actuels, sans que des techniques de tolérance aux fautes n'offrent encore de solution vis-à-vis de cet aspect. En particulier, les connaissances de ces mécanismes (modèles et heuristiques) représentent un aspect clé des systèmes autonomes, comme facteur majeur dans leur comportement, mais aussi comme source de fautes.

Deux contextes principaux peuvent être distingués dans cet aspect : l'acceptabilité des décisions et leur temps de réponse.

L'*acceptabilité des décisions* peut être mise en défaut par des compromis ou des fautes de conception des mécanismes décisionnels. Du point de vue tolérance aux fautes, une diversification des modèles et des heuristiques utilisées dans les mécanismes d'inférence peut participer à une éventuelle solution. En ce qui concerne l'évitement des fautes, il nous semble que des travaux portant sur l'expression et la validité des connaissances sont à réaliser dans le domaine des systèmes autonomes.

Pour améliorer le *temps de réponse*, on peut par exemple chercher des méthodes capables d'identifier des heuristiques efficaces pour un type de problème donné, ou exécuter en parallèle plusieurs processus décisionnels diversifiés (par exemple un même planificateur, mais muni de différentes heuristiques) et choisir la première décision obtenue.

Dans le chapitre suivant, nous proposons des mécanismes de tolérance aux fautes cherchant à améliorer la fiabilité des mécanismes décisionnels dans les systèmes autonomes, à la fois à travers l'acceptabilité de leur décisions et leur temps de réponse.

Ce qu'il faut retenir :

1. Le type d'architecture hiérarchisé a prouvé son efficacité à travers plusieurs applications réussies, mais possède trois désavantages majeurs : les désaccords possibles entre niveaux d'abstraction, sa lenteur de réaction face à des situations adverses complexes, et la criticité du mécanisme de planification.
2. Le type d'architecture multi-agents répond à certains de ces désavantages. Cependant, ce type d'architecture semble long et difficile à développer, et des méthodes de conception particulières restent à développer pour faciliter son utilisation.
3. Les composants indépendants de sécurité apparaissent comme une solution avantageuse pour assurer la sécurité-innocuité d'un système autonome. Ils sont conçus et validés indépendamment du système autonome, disposent d'une vision globale permettant de mesurer les conséquences dangereuses de certaines actions, et offrent un filet de secours aux problèmes de validation des mécanismes décisionnels.

4. Les mécanismes de planification sont actuellement incontournables dans les systèmes autonomes complexes. Cependant, leur comportement est difficile à prévoir, et les techniques classiques de sûreté de fonctionnement leur sont mal adaptés.
5. Des travaux existent pour améliorer validation et conception des planificateurs, même si de notre point de vue ils ne sont pas encore assez répandus. La conception de mécanismes de planification tolérants aux fautes de développement nous semble en particulier une voie innovante et prometteuse.

Chapitre 3

Tolérance aux fautes pour la planification

En tenant compte de l'étude présentée dans le chapitre précédent, nous présentons ici des mécanismes de tolérance aux fautes pour la planification, cherchant en particulier à tolérer les fautes de développemment dans les connaissances d'un planificateur (c'est-à-dire dans ses modèles et ses heuristiques). En effet, les mécanismes d'inférence des planificateurs sont proches des programmes impératifs habituellement traités en sûreté de fonctionnement informatique, et peuvent ainsi être validés, par exemple, à l'aide de méthodes formelles et de tests intensifs. Ces techniques sont par contre plus difficilement applicables aux connaissances des planificateurs, généralement exprimées en programmation déclarative.

Dans ce chapitre, nous décrivons tout d'abord le principe général des mécanismes de tolérance aux fautes que nous proposons, en précisant les techniques de détection et de rétablissement proposées. Nous présentons ensuite un exemple concret d'application : leur mise en œuvre sur l'architecture hiérarchisée LAAS.

3.1 Mécanismes proposés

Les mécanismes que nous proposons se basent principalement sur la diversification des connaissances du planificateur. Cette diversification permet, en théorie, de tolérer des fautes de développement dans son modèle et ses heuristiques, qui sont tous deux essentiels dans la définition du comportement du système. Leur principe général consiste à exécuter plusieurs planificateurs, disposant de modèles et/ou d'heuristiques diversifiés.

Nous proposons ici quatre mécanismes de détection et deux mécanismes de rétablissement, ainsi que le principe général d'un composant indépendant chargé de les mettre en place. Ces mécanismes sont particulièrement adaptés à un système autonome de type hiérarchisé, qui dispose d'un planificateur central au niveau décisionnel.

3.1.1 Mécanismes de détection

Dans le chapitre précédent, nous avons expliqué pourquoi les mécanismes de planification posent des problèmes particuliers dans le cadre de la tolérance aux fautes, en particulier pour la détection d'erreurs (difficulté de porter un jugement sur la correction d'un plan et complexité de l'état interne)[1]. Nous proposons quatre mécanismes de détection d'erreurs de planification permettant dans une certaine mesure de contourner ces difficultés : un chien de garde temporel, un analyseur de plan, la détection d'un échec d'action, et une vérification en ligne de l'accomplissement des objectifs.

Un *chien de garde temporel* peut être lancé au début d'une planification, pour détecter un temps de planification trop long, ou une défaillance totale du planificateur, telle qu'un blocage ou une erreur d'allocation. Ces erreurs peuvent être causées par des fautes dans les connaissances du planificateur (modèle ou heuristiques), mais aussi par des fautes dans son mécanisme d'inférence ou encore la non-calculabilité du problème posé. Cependant, nos mécanismes de rétablissement ne sont pas focalisés sur ces deux dernières sources d'erreurs, et n'y offrent qu'un traitement rudimentaire.

Un *analyseur de plan* peut être appliqué au plan fourni par le mécanisme décisionnel. En effet, bien que les spécificités des planificateurs ne permettent pas d'utiliser des mécanismes classiques comme la duplication et comparaison, il est cependant possible de vérifier sur le plan produit un ensemble de propriétés et de contraintes, selon le même principe que les mécanismes de validation de plan présentés dans le chapitre précédent, mais exécuté en ligne. L'ensemble de contraintes doit être obtenu à partir des spécifications du système et d'expertise sur le domaine, de façon diversifiée par rapport au modèle de planification. Ce mécanisme de détection permet de détecter les erreurs dues à des fautes dans le modèle ou les heuristiques, ainsi que dans le mécanisme décisionnel.

La *détection d'un échec d'action* est un mécanisme que nous avons présenté dans le premier chapitre, utilisé en robustesse pour détecter une situation adverse. Cet échec d'action indique que le plan ne s'est pas déroulé comme prévu, et peut donc être dû à une situation externe au système, mais aussi à des erreurs dans le plan en raison de fautes dans les connaissances du mécanisme décisionnel, dans son mécanisme d'inférence, ou dans le reste du système. Comme le système ne dispose pas des informations objectives et des capacités de jugement nécessaires pour diagnostiquer la cause de l'échec, nous choisissons par précaution de considérer chaque échec d'action comme une erreur du planificateur. Ce choix peut néanmoins avoir une incidence négative sur la fiabilité du système, quand l'échec est effectivement dû à une situation adverse externe, et donc met en cause sans raison le planificateur.

Une *vérification des objectifs en ligne* permet de vérifier si, à la fin de l'exécution du plan, tous les objectifs sont bien remplis. Cette vérification doit être mise à jour au cours de l'exécution du plan, en s'assurant à la fin de chaque action si on a atteint ou manqué des objectifs, mais ne peut être validée qu'une fois que toutes les actions du plan ont été accomplies, ou lors d'une replanification en envoyant au planificateur la liste des objectifs restant à atteindre. Le maintien d'une repré-

[1] Voir 2.3.2.1.

sentation interne de l'état du système et des objectifs réalisés est nécessaire pour effectuer cette vérification.

Le chien de garde temporel et l'analyseur de plan permettent de détecter des erreurs *avant* l'exécution du plan, tandis que la détection d'un échec d'action et la vérification d'objectifs en ligne détectent des erreurs *pendant* cette exécution. Une détection avant exécution est avantageuse, puisqu'elle permet, d'une part, de ne pas perdre du temps à exécuter le plan erroné, et, d'autre part, de ne pas risquer une défaillance causée par l'erreur présente dans le plan.

3.1.2 Mécanismes de rétablissement

Nous proposons deux mécanismes de rétablissement, basés sur l'utilisation de plusieurs planificateurs disposant de connaissances diversifiées, selon une approche similaire aux blocs de recouvrement présentés par Randell [Randell 1975]. Le premier mécanisme utilise une politique de *planification successive*, changeant de planificateur à chaque détection d'erreurs. Le second mécanisme se rapproche des travaux de [Kim & Welch 1989] sur les blocs de recouvrement distribués, en ce que les différentes alternatives sont exécutées *de façon concurrente*. Cependant, contrairement aux blocs de recouvrement distribués, nos mécanismes de détection d'erreurs ne sont pas exécutés en parallèle par chaque alternative, mais centralisés dans un composant indépendant, présenté dans la Section 3.1.3. Quel que soit le mécanisme choisi, une étape de mise en état sûr du système consistant à arrêter l'exécution des activités en cours est nécessaire avant le rétablissement. Cette étape est déjà mise en place dans les systèmes autonomes effectuant une replanification, dans le but de laisser le système évoluer aussi peu que possible pendant l'activité du planificateur, de façon à ne pas mettre en défaut le plan produit par des actes entrepris pendant la planification. Cette mise en état sûr n'est cependant pas utilisée dans le cas d'une réparation de plan ou d'une planification anytime (voir Section 2.1.1.2), qui exécutent toutes deux une partie du plan pendant le processus de planification.

3.1.2.1 Planification successive

Ce premier mécanisme consiste à exécuter les planificateurs de façon *successive*. Le principe de fonctionnement, présenté en détail dans la Figure 3.1 sous forme d'algorithme informel, vise à changer de planificateur à chaque fois qu'une erreur est détectée. On cherche ainsi à privilégier la fiabilité du système face aux fautes de développement, en basculant sur un mécanisme décisionnel diversifié de celui qui a échoué. Lorsque tous les planificateurs ont été utilisés, on se permet d'utiliser à nouveau les premiers planificateurs. Cette décision est motivée principalement par deux raisons :

- Tout d'abord, la détection d'un échec d'action peut être déclenchée aussi bien par une situation adverse ou une faute externe au planificateur qu'en raison du planificateur lui-même. Au lieu d'écarter à tort un planificateur sans fautes, on préfère donc le ré-utiliser.
- D'autre part, certains modèles, même en comportant des fautes, peuvent s'avérer performants dans certaines situations ; en particulier, lorsque la si-

```
1.   begin mission
2.       défaillants ← ∅
         % défaillants : ensemble des planificateurs
         % ayant défailli successivement
3.       while(objectifs ≠ ∅)
4.           candidats ← planificateurs ;
             % planificateurs : ensemble des planificateurs disponibles
5.           while(candidats ≠ ∅ & objectifs ≠ ∅)
6.               choisir k tel que (k ∈ candidats) & (k ∉ défaillants) ;
7.               candidats ← candidats \ k ;
8.               amorcer_chien_de_garde(temps_max_planif) ;
9.               envoyer(demande_plan) à k ;
10.              wait
                 % on attend une des deux conditions □
11.              □ reçu(plan_trouvé) de k
12.                  désamorcer_chien_de_garde() ;
13.                  if analyse(plan)=OK then
14.                      défaillants ← ∅ ;
15.                      k.executer(plan) ;
                         % l'exécution est considérée bloquante ;
                         % en cas d'échec du plan, on a objectifs ≠ ∅
                         % et on reprend la boucle de la ligne 4
                         % (ou de la ligne 3 si candidats = ∅)
16.                  else
17.                      envoyer(plan_invalide(k)) à l'opérateur ;
18.                      défaillants ← défaillants ∪ k ;
19.                  end if
20.              □ échéance_chien_de_garde
21.                      défaillants ← défaillants ∪ k ;
22.              end wait
23.              if défaillants = planificateurs then
24.                  raise exception "échec mission : aucun plan
                                      valide trouvé dans les temps" ;
                     % tous les planificateurs ont échoué
                     % successivement : les redondances sont
                     % épuisées et la mission a échoué
25.              end if
26.          end while
27.      end while
28.  end mission
```

Figure 3.1: Algorithme de planification successive

tuation adverse ayant activé la faute a disparu.

Cependant, si tous les planificateurs défaillent successivement en considérant le même problème (soit par des défaillances temporelles détectées par le chien de garde, soit par des défaillances en valeur détectées par l'analyseur de plan, soit par une combinaison des deux), alors les redondances disponibles ont été épuisées, et la poursuite de la mission est impossible. Dans notre algorithme, cette situation est détectée à l'aide de l'ensemble `défaillants` : l'ensemble les planificateurs ayant défailli successivement. Cet ensemble est initialisé à l'ensemble vide au début de la mission (ligne 2). Un planificateur qui défaille en valeur ou temporellement lui est ajouté (ligne 18 et 21) : si l'ensemble comprend tous les planificateurs candidats, une exception est levée pour signaler la défaillance du système (ligne 24). Finalement, l'ensemble `défaillants` est réinitialisé dés qu'un plan détecté sans erreurs est trouvé par un planificateur (ligne 14).

On retrouve dans cet algorithme les quatre mécanismes de détection présentés précédemment : chien de garde temporel (ligne 20), analyseur de plan (ligne 13), détection d'échec d'action (ligne 15), vérification d'objectifs en ligne (ligne 3 et 5).

Un point important, qui n'est pas complètement explicité dans l'algorithme, concerne le choix d'un planificateur parmi plusieurs candidats (ligne 6). Ce choix peut être imposé par le développeur par un ordre strict fixé avant l'exécution, par exemple après une étude de prévision des fautes. Il est également possible d'imaginer des critères plus évolués, prenant en compte, par exemple, un diagnostic réalisé à partir des exécutions précédentes, ou des performances attendues de chaque modèle dans différentes situations.

3.1.2.2 Planification concurrente

Dans ce mécanisme, les planificateurs sont exécutés en parallèle pendant la planification. Ce principe général est repris en détail dans la Figure 3.2. Après planification, un des plans produits est sélectionné, puis validé par l'analyseur de plan ; si aucune erreur n'a été détectée, la planification des autres planificateurs est interrompue, et le plan du planificateur sélectionné est exécuté. En cas d'erreur, un autre plan est sélectionné et passé à l'analyseur de plan, jusqu'à obtention d'un plan correct ou dépassement du chien de garde temporel. Dans ce dernier cas, le système est placé dans un état sûr, et un message d'erreur est envoyé à l'utilisateur.

On retrouve toujours les quatre mêmes mécanismes de détection présentés précédemment : chien de garde temporel (ligne 20), analyseur de plan (ligne 10), détection d'échec d'action (ligne 13), vérification d'objectifs en ligne (ligne 2).

Ici, le problème de choix du planificateur est masqué par la décision implicite de sélectionner le premier plan valide trouvé. Cependant, il s'avère ainsi possible qu'un sous-ensemble des planificateurs candidats ne soient jamais choisis par le système, limitant ainsi la redondance et les capacités de tolérance aux fautes du mécanisme de rétablissement. Par exemple, un planificateur peut comporter des fautes dans ses connaissances et produire de façon répétitive un plan erroné non-détecté par l'analyseur de plan, mais avoir un meilleur temps de calcul que les autres, et ainsi être sélectionné à chaque cycle. Éventuellement, des mécanismes de diagnostic peuvent être mis en place pour résoudre ce problème dans une certaine mesure, comme dans le cas de la planification successive, en prenant en compte

```
1.   begin mission
2.       while(objectifs ≠ ∅)
3.           candidats ← planificateurs ;
             % planificateurs : ensemble des planificateurs disponibles
4.           amorcer_chien_de_garde(temps_max_planif) ;
5.           envoyer(demande_plan) aux candidats ;
6.           while(candidats ≠ ∅)
7.               wait
                 % on attend une des deux conditions □
8.               □ reçu(plan_trouvé) de k ∈ candidats
9.                   candidats ← candidats \ k ;
10.                  pause_chien_de_garde() ;
11.                  if analyse(plan)=OK then
12.                      désamorcer_chien_de_garde() ;
13.                      envoyer(abandon_planif) aux candidats ;
14.                      candidats ← ∅ ;
15.                      k.executer(plan) ;
                         % l'exécution est considérée bloquante ;
                         % en cas d'échec du plan, on a objectifs ≠ ∅
                         % et on reprend la boucle de la ligne 2
16.                  else
17.                      reprendre_chien_de_garde() ;
18.                      envoyer(plan_invalide(k)) à l'opérateur ;
19.                      if candidats = ∅
20.                          raise exception "échec mission : aucun
                                 plan valide trouvé" ;
                             % les redondances sont épuisées et
                             % la mission a échoué
21.                      end if
22.                  end if
23.              □ échéance_chien_de_garde
24.                  raise exception "échec mission : aucun
                             plan valide trouvé dans les temps" ;
                     % les redondances sont épuisées et
                     % la mission a échoué
25.          end while
26.      end while
27.  end mission
```

Figure 3.2: Algorithme de planification concurrente

les planifications et les exécutions précédentes du système. Cependant, contrairement à la planification successive, la succession de tous les planificateurs n'est pas garantie explicitement par l'algorithme, et il est toujours possible que des planificateurs avec de mauvaises performances d'exécution soient constamment ignorés par le système. Des méthodes de passivation des planificateurs effectivement sélectionnés, ou de leur pénalisation par un retard artificiel ajouté au temps de planification pour chaque échec d'exécution, peuvent être envisagées pour répondre à cette nouvelle difficulté.

3.1.3 Le composant FTplan

La synchronisation et les interactions des différents planificateurs sont gérées par un composant appelé FTplan, pour *Fault Tolerant planning*, qui assure un fonctionnement transparent au reste du système, et met en place les mécanismes de détection et de rétablissement présentés précédemment.

3.1.3.1 Place dans l'architecture

Dans la conception du composant FTplan, deux aspects fondamentaux sont à considérer : son *indépendance* vis-à-vis des planificateurs qu'il doit gérer, et les *moyens d'observation et d'action* nécessaires à la mise en place des mécanismes proposés.

Ayant pour but de tolérer les fautes dans les planificateurs, FTplan doit en être le plus indépendant possible afin de limiter les risques de propagation d'erreurs venant du composant surveillé. Il faut entre autres s'assurer que les erreurs ne peuvent pas se propager à l'interface d'un planificateur. En pratique, cela veut dire que FTplan ne doit pas faire confiance aux informations que possèdent les mécanismes décisionnels qu'il contrôle, d'où la nécessité de posséder une représentation propre de l'état du système, nécessaire à sa fonction, mais ne reposant pas sur des données venant des planificateurs.

Pour mettre à jour cette représentation du système, et pour exécuter les mécanismes de rétablissement proposés, FTplan a besoin d'informations et de moyens d'action. Comme son objectif est uniquement de tolérer des fautes dans les planificateurs, nous choisissons de considérer que les composants inférieurs du système sont sûrs, et disposent de leurs propres mécanismes de tolérance aux fautes. Le plus judicieux nous semble de récupérer principalement les informations possédées par le niveau exécutif : elles ont déjà été traitées par la couche fonctionnelle et s'approchent du niveau d'abstraction des modèles du planificateur, tout en gardant une précision suffisante pour vérifier la bonne exécution d'un plan. En effet, les planificateurs utilisent souvent des données plus abstraites et moins précises pour simplifier leurs recherches (par exemple, la position du robot dans son environnement peut être caractérisée de façon symbolique ou approximative, par opposition à des coordonnées numériques précises utilisées pour le contrôle d'exécution).

En ce qui concerne ses moyens d'action, FTplan doit pouvoir gérer plusieurs planificateurs, et donc être capable de communiquer avec eux. Il doit également

être capable de les arrêter et de les relancer, par exemple dans le cas où un planificateur met trop de temps pour planifier.

La Figure 3.3 présente la place de notre composant FTplan au sein d'une architecture hiérarchisée.

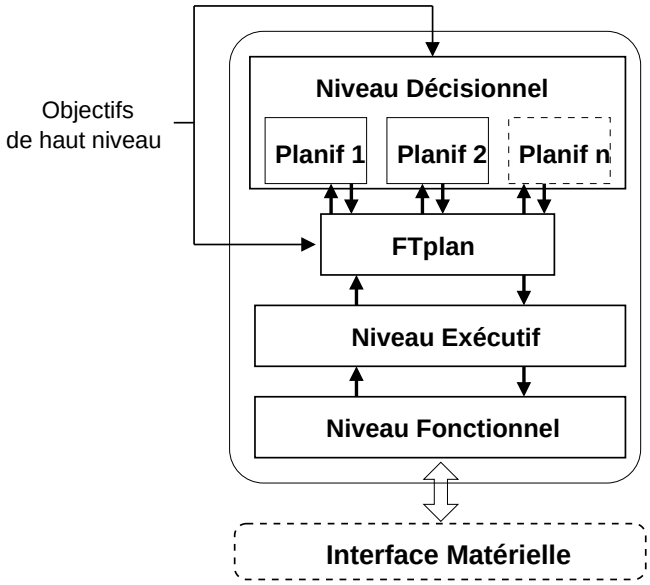

Figure 3.3: Place de FTplan dans une architecture hiérarchisée

3.1.3.2 Principales fonctions du composant

Le composant FTplan a quatre fonctions principales dans le système autonome :

- gestion de plusieurs planificateurs avec modèles ou heuristiques diversifiés (lancement, arrêt, synchronisation),
- mise en place de trois mécanismes de détection d'erreurs sur quatre (chien de garde temporel, analyse de plan, et vérification des objectifs en ligne) ; en effet, le dernier mécanisme (détection d'un échec d'action) est généralement déjà réalisé dans les systèmes autonomes par le contrôleur d'exécution du niveau exécutif,
- mise en place d'un des deux mécanismes de rétablissement proposé (planification successive ou concurrente),
- maintien d'une représentation de l'état du système et de l'accomplissement des objectifs, à travers l'analyse des informations fournies par les niveaux bas du système, en particulier le niveau exécutif ; cet état est fourni à chaque planificateur comme état initial d'une replanification, mais sert également à vérifier la réalisation des objectifs.

3.2 Mise en œuvre

Cette section présente un exemple d'application des mécanismes présentés ci-dessus à l'architecture LAAS, que nous avons réalisé pendant nos travaux. Il donne une idée générale des moyens possibles pour mettre en œuvre les mécanismes de tolérance aux fautes proposés, et démontre leur faisabilité.

Nous introduisons tout d'abord l'architecture sur laquelle nous avons travaillé, avant de décrire le développement et l'implantation du composant FTplan dans cette architecture.

3.2.1 L'architecture LAAS

Parmi les nombreuses architectures hiérarchisées existant dans la communauté (CLARAty, CIRCA, etc.), l'architecture LAAS présente pour nous deux avantages pratiques significatifs. En premier lieu, cette architecture est développée sur le site, ce qui permet une coordination plus aisée avec l'équipe de développement. En effet, les systèmes autonomes sont des systèmes complexes comportant de nombreux composants, qu'il est difficile à une seule personne de maîtriser. L'échange d'informations et le support technique sont ainsi deux aspects fondamentaux pour leur utilisation. De plus, une interface a été développée entre le logiciel de simulation de robot Gazebo et les modules fonctionnels de cette architecture, facilitant le développement d'un environnement d'évaluation[2].

Nous donnons ici une présentation générale de cette architecture, avant de nous attarder un peu plus sur le composant de planification : IxTeT.

3.2.1.1 Présentation de l'architecture

L'architecture LAAS est présentée de manière générale dans [Alami et al. 1998] et [Ingrand et al. 2001] ; des évolutions plus récentes sont détaillées dans [Py & Ingrand 2004] et [Lemai & Ingrand 2004]. Elle est utilisée sur plusieurs robots appelés HILARE, Diligent, Dala et LAMA, en s'exécutant au-dessus du système d'exploitation Linux. En pratique, elle est constituée de quatre principaux composants, représentés sur la Figure 3.4 :

- Le sous-système de modules développés avec l'outil GenoM assure l'exécution de tâches de bas niveau, tenant donc le rôle de niveau fonctionnel. Un module peut être chargé de contrôler un composant matériel du système (par exemple le moteur ou les caméras), ou de réaliser une fonction particulière à partir d'informations ou de fonctions offertes par les autres modules (par exemple la localisation ou la navigation).
- R2C, présenté dans le chapitre précédent, est un composant indépendant vérifiant pendant l'exécution que le système respecte bien des contraintes spécifiées pendant le développement.
- OpenPRS est un contrôleur d'exécution procédural. Il applique la fonction autonome de contrôle d'exécution et tient donc le rôle de niveau exécutif, en décomposant les actions de haut niveau demandées par IxTeT en séquences d'actions plus simples, et en gérant leur exécution.

[2] Nous reviendrons plus en détail sur cet aspect dans le chapitre suivant.

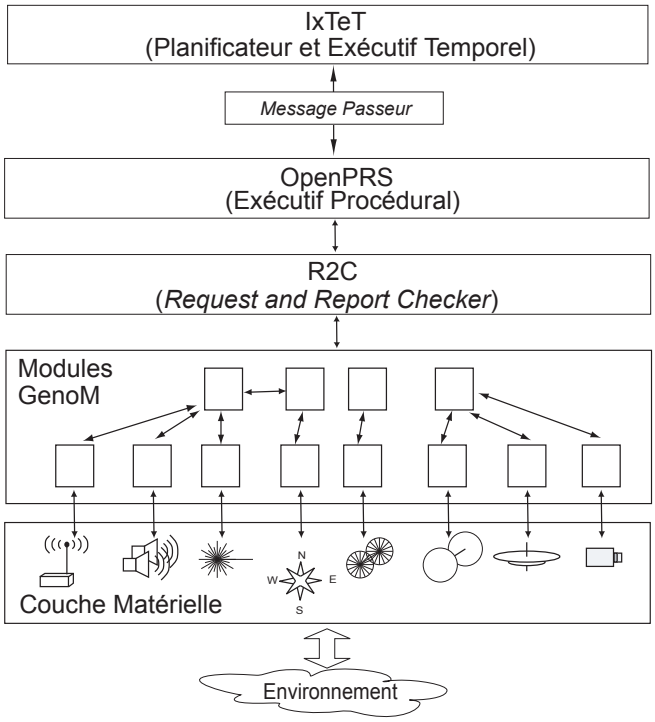

Figure 3.4: Architecture LAAS

- IxTeT est un composant regroupant la fonction autonome de planification et celle de contrôle d'exécution temporel, tenant une partie du rôle du niveau exécutif en plus de celui du niveau décisionnel. Il est constitué d'un planificateur temporel par contraintes, enrichi par des fonctionnalités de replanification, réparation de plan, et contrôle d'exécution temporelle.

Un composant supplémentaire de gestion de messages, appelé *MessagePasseur*, assure la communication entre OpenPRS et IxTeT, en fournissant un service d'envoi et de réception de chaînes de caractères pour ces deux processus. Nous présentons rapidement ci-dessous le sous-système de modules GenoM et le composant OpenPRS. La section suivante décrit un peu plus longuement le fonctionnement de IxTeT.

Modules GenoM

Le niveau fonctionnel est composé de plusieurs entités logicielles appelées *modules*, qui offrent des services liés à des ressources physiques (capteurs, effecteurs) ou logiques (données). En particulier, un module contient les procédures de commande, de détection d'erreur et de gestion de ses ressources, masquant la complexité et le fonctionnement de la couche matérielle au niveaux supérieurs.

Un module met ses services à la disposition du niveau supérieur ou d'autres modules selon une relation client/serveur. Les services sont paramétrés et activés de façon asynchrone par des requêtes envoyées au module serveur ; la fin du service est marquée par une réponse envoyée au client, contenant un rapport d'exé-

cution (exécution correcte ou message d'erreur) et éventuellement des données. Pour permettre un échange de données entre les modules malgré l'asynchronisme de leurs activités, chaque module met à jour un espace de mémoire partagée appelé *poster*, lisible par les autres modules.

Les modules sont générés automatiquement par l'outil GenoM : celui-ci prend en entrée la description d'un module (contenant le nom du module et la déclaration de ses données, requêtes et *posters*) et une bibliothèque d'algorithmes, à partir desquelles il construit la représentation interne d'un module puis son code [Fleury et al. 1997].

OpenPRS

OpenPRS effectue un contrôle d'exécution sur les actions planifiées par IxTeT. En pratique, il raffine les actions de haut niveau transmises par le planificateur en séquences de requêtes fonctionnelles exécutables par les modules GenoM, et retourne au planificateur un bilan d'exécution à la fin de chaque action.

Pour ce faire, OpenPRS dispose de plusieurs bibliothèques de procédures permettant de commander les modules GenoM. Une de ces bibliothèques concerne l'initialisation du système : elle séquence les requêtes successives à envoyer à chaque module pour mettre le système dans un état initial prêt à accomplir sa mission, et n'a pas de lien direct avec le planificateur. Une autre bibliothèque décrit pour chaque action de haut niveau du planificateur une procédure particulière, contenant la séquence de requêtes ou de procédures de plus bas niveau nécessaires à l'exécution de l'action, ainsi que le bilan final à renvoyer en fonction de leurs résultats.

Lors de l'exécution du système, OpenPRS reçoit d'IxTeT les actions à exécuter, et lance la séquence de procédures et de requêtes GenoM associées. Comme plusieurs actions IxTeT peuvent être réalisées en même temps, il a également pour but d'ordonner les procédures pour permettre leur exécution parallèle.

3.2.1.2 IxTeT

Le composant IxTeT met en œuvre la planification et le contrôle temporel d'exécution dans l'architecture LAAS[3]. Ce composant est codé en C++, et comprend plusieurs milliers de lignes de code. Nous présentons dans cette section le formalisme et le fonctionnement général du planificateur, puis la boucle de contrôle d'exécution temporelle qui étend le planificateur.

Formalisme

IxTeT [Ghallab & Mounir-Alaoui 1989, Lemai-Chenevier 2004] est un planificateur temporel par contraintes. Il raisonne sur son environnement en considérant un ensemble de fonctions temporelles constantes par morceaux, utilisées d'une

[3]Ne fournissant à l'origine que des fonctions de planification, ce composant a fait l'objet de nombreuses contributions, par exemple une recherche dans l'espace des plans partiels avec une stratégie de moindre engagement [Laruelle 1994] et l'extension des gestionnaires de contraintes pour traiter des variables atemporelles numériques et des contraintes mixtes entre variables temporelles et atemporelles [Trinquart & Ghallab 2001]. Notamment, il a été étendu pour fournir des fonctionnalités de contrôle d'exécution temporel, ainsi que de réparation de plan et de replanification [Lemai-Chenevier 2004].

```
1.  task PRENDRE_IMAGE( ?x, ?y)(t_début, t_fin) {
2.      ?x in [-oo,+oo] ; ?y in [-oo,+oo] ;
3.      hold(POS_X() : ?x,(t_début, t_fin)) ;
4.      hold(POS_Y() : ?y,(t_début, t_fin)) ;
5.      hold(POS_CAMERA() :bas,(t_début, t_fin)) ;
6.      event(IMAGE( ?x, ?y) :(a_faire,faite),t_fin) ;
7.      use(CAMERA() :1,(t_début, t_fin)) ;
8.      (t_début - t_fin) in [0,60] ;
9.  }nonPreemptive
```

Figure 3.5: Action PRENDRE_IMAGE dans le formalisme IxTeT

part pour représenter l'état du système et de son environnement (par exemple la position du système ou la satisfaction d'un objectif), et d'autre part les ressources du système. La description de ces fonctions et leur évolution pendant l'activité du système sont décrites dans des fichiers, qui constituent le modèle du planificateur pour une application donnée.

Les fonctions représentant l'état du système et de l'environnement, appelées *attributs*, peuvent être qualifiées par le prédicat d'assertion *hold*, qui représente la persistance de la valeur d'un attribut pour un intervalle temporel donné (par exemple : *hold(robot, position, (début, fin))*), ou par le prédicat d'évènement *event*, qui représente une modification instantanée dans la valeur d'un attribut (par exemple : *event(robot, (position1, position2), instant)*). Les ressources du système sont qualifiées par les prédicats *produce*, *consume* et *use*, qui représentent respectivement une production, consommation ou utilisation temporaire de la ressource.

Pour construire un plan, le planificateur manipule des actions de haut niveau. Chaque action regroupe un ensemble de contraintes, qui décrivent l'évolution prévue du système et de l'environnement lors d'une exécution nominale de l'action. Ces contraintes peuvent porter sur les attributs ou les ressources, ou sur des variables numériques ou temporelles, par exemple pour spécifier la durée d'une action, ou la quantité d'énergie consommée. Un plan valide est constitué d'un ensemble d'actions dont l'exécution n'est pas conflictuelle, et qui satisfait les objectifs de la mission.

Un exemple d'action décrivant une prise d'image est donné dans la Figure 3.5. La ligne 1 déclare l'action et ses paramètres numériques et temporels. Les lignes 3 à 5 donnent une liste d'assertions définissant les contraintes à respecter sur les attributs du système tout au long de l'action ; par exemple, les lignes 3 et 4 stipulent que la position du robot ne doit pas changer pendant une prise d'image, c'est-à-dire qu'il doit rester immobile. La ligne 6 définit un évènement qui se produit pendant l'action : en l'occurrence, une fois qu'une photo est prise, l'attribut qui lui correspond passe de l'état `a_faire` à `faite`. La ligne 7 donne un exemple d'utilisation de ressources : la ressource CAMERA est utilisée pendant toute l'action. La ligne 8 donne un exemple de contraintes sur les valeurs temporelles, en spécifiant l'intervalle de temps possible que prend l'action.

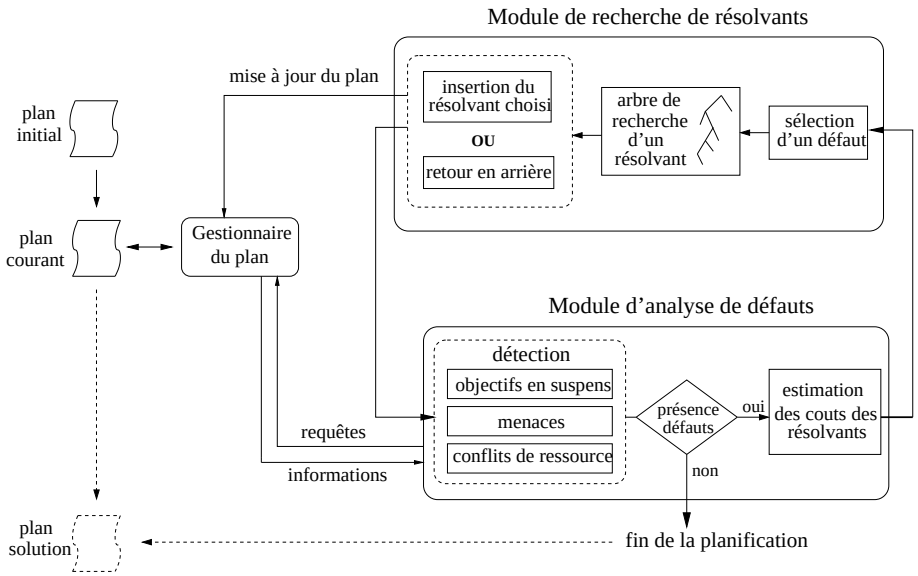

Figure 3.6: Fonctionnement du planificateur IxTeT

Fonctionnement

Le fonctionnement général du mécanisme de planification, basé sur une recherche dans un espace de plans, est présenté sur la Figure 3.6.

L'entrée du planificateur est un plan partiel, qui comprend les valeurs initiales des attributs, les objectifs à atteindre, la disponibilité initiale des ressources, et des contraintes temporelles ou numériques associées à ces différents éléments. Une fois le plan initial entré, le planificateur applique sa boucle d'exécution jusqu'à ce qu'une solution soit trouvée, ou que l'arbre de recherche complet ait été parcouru (ce qui peut, pour certains modèles, durer indéfiniment). Le planificateur IxTeT est complet dans le sens où s'il existe une solution, il la trouvera, mais il ne donne aucune garantie sur le temps nécessaire ou sur l'optimalité de la solution.

Le module d'analyse détecte l'ensemble des *défauts* du plan partiel courant, qui peuvent être des *objectifs en suspens* (des assertions ou des évènements qui n'ont pas encore été établis), des *menaces* (des assertions ou des évènements conflictuels), ou des *conflits de ressource*. S'il n'y a plus de défaut, le plan est une solution du problème, et la planification a réussi. Dans le cas contraire, un défaut est choisi grâce à une heuristique, et transmis au module de recherche qui va chercher à le résoudre soit en insérant une nouvelle action, soit en ajoutant une contrainte d'ordonnancement entre les actions. Si aucun résolvant n'est trouvé, le planificateur effectue un retour en arrière jusqu'à un nœud comportant des défauts qui n'ont pas encore été essayés. Si l'arbre de recherche a été complètement parcouru sans qu'une solution ait été trouvée, la planification a échoué.

Contrôle temporel d'exécution

Dans [Lemai-Chenevier 2004], le planificateur IxTeT est étendu en offrant la supervision temporelle de l'exécution des actions, et des fonctionnalités de replanification et de réparation de plan.

En particulier, il vérifie régulièrement si l'exécution du plan se déroule convenablement ; lors d'un échec d'une partie du plan, il essaie de "réparer" le plan en ne replanifiant que les objectifs ou les contraintes mis en échec. Si cela s'avère trop difficile, une replanification totale est alors nécessaire.

Il est principalement mis en œuvre par une boucle d'exécution en quatre temps : perception, réparation de plan, replanification et exécution. À chacun de ces cycles, il va chercher à mettre à jour le plan qu'il exécute en fonction des bilans d'action remontés par OpenPRS. Si un problème est détecté dans l'exécution (échec d'une action à travers un bilan échoué ou le dépassement d'une échéance temporelle), il va d'abord essayer de réparer le plan actuel, avant d'arrêter son exécution et de replanifier en dernier recours. Si le plan n'est pas mis en difficulté, ou qu'une solution est trouvée, la phase d'exécution envoie les actions à entreprendre à OpenPRS, s'il y en a. À moins qu'une réparation de plan ou une replanification ne soit nécessaire, un tel cycle d'exécution prend environ deux secondes sur une plate-forme i386 à 3.2GHz sous linux.

3.2.2 Intégration de FTplan dans l'architecture LAAS

Nous présentons ici l'application des principes généraux de nos mécanismes de tolérance aux fautes, présentés en début de chapitre, à l'architecture LAAS. Nous utilisons comme planificateurs redondants deux planificateurs IxTeT disposant de modèles et d'heuristiques diversifiés. Cet exemple pratique nous permet de démontrer la faisabilité de notre approche, et représente un premier pas vers sa validation.

Nous précisons tout d'abord les relations de notre composant avec l'architecture LAAS, avant de détailler son fonctionnement ainsi que les modifications nécessaires des composants IxTeT et OpenPRS.

3.2.2.1 Place dans l'architecture LAAS

D'après le principe général que nous avons présenté en début de chapitre, la place de FTplan dans un système autonome se trouve entre le contrôleur d'exécution et le planificateur. Cependant dans l'architecture LAAS, le contrôle d'exécution est partagé entre les composants OpenPRS et IxTeT, ce dernier contenant également le planificateur. Afin d'éviter la propagation d'erreurs des composants IxTeT vers FTplan et de limiter le travail de développement nécessaire, nous avons choisi de placer FTplan entre OpenPRS et les IxTeT redondants, plutôt qu'à la frontière, difficile à définir, entre les fonctions de planification et de contrôle d'exécution temporel de IxTeT.

Pour favoriser l'indépendance entre FTplan et les processus IxTeT, nous utilisons la séparation entre processus offerte par le noyau Linux, en exécutant FTplan comme un processus distinct. L'indépendance entre processus est assurée parce que d'une part un processus P1 ne peut pas accéder à l'espace mémoire d'un autre processus P2 (une erreur est soulevée dans P1 s'il essaie d'accéder à un espace mémoire qui ne lui est pas attribué), et que d'autre part le contexte d'exécution d'un processus P est automatiquement enregistré par le système d'exploitation

lorsqu'un processus différent est sélectionné pour être exécuté par le processeur, et restauré lorsque P est à nouveau exécuté[4] [Bovet & Cesati 2002].

Le moyen d'observation le plus simple pour FTplan nous a paru d'utiliser le mécanisme de communication déjà employé par OpenPRS et IxTeT : le *Message-Passeur*. Au lieu de s'échanger directement les messages, OpenPRS et IxTeT les envoient d'abord à FTplan, qui les transmettra à leur destination originelle après traitement. Ce même mécanisme permet la communication, et donc la synchronisation, avec les processus IxTeT. Enfin, FTplan utilise des fonctions du système d'exploitation pour lancer et arrêter l'exécution des planificateurs.

Le composant FTplan est codé dans le langage objet C++. Le choix de ce langage de programmation est principalement motivé par la volonté de réutiliser l'interface avec le *MessagePasseur* présente dans IxTeT.

L'implantation de FTplan dans les mécanismes décisionnels de l'architecture LAAS est illustrée dans la Figure 3.7.

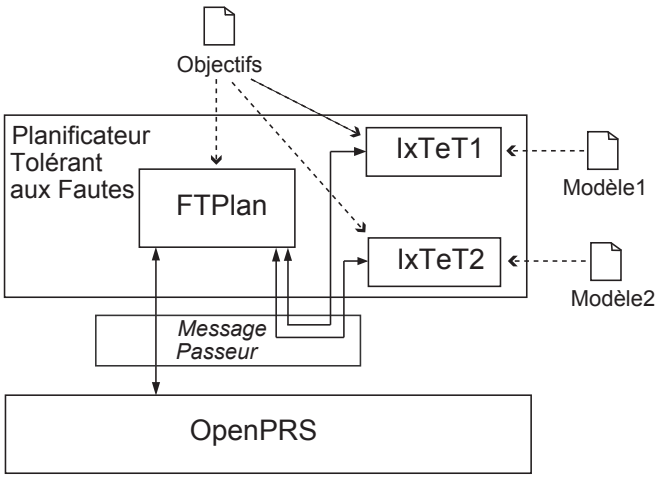

Figure 3.7: FTplan et les mécanismes décisionnels de l'architecture LAAS

3.2.2.2 Fonctionnement de FTplan

Cette section décrit le fonctionnement du composant FTplan, en précisant son algorithme général, ainsi que les choix effectués dans la mise en œuvre des mécanismes de tolérance aux fautes présentés en début de chapitre.

Afin d'implanter FTplan dans des délais raisonnables, quatre compromis de développement ont été réalisés :

- Le mécanisme de détection d'erreurs par analyse de plan n'a pas été développé. En effet, cet analyseur de plan nécessite un travail significatif de re-

[4]Dans le système Linux, le contexte d'exécution d'un processus comprend entre autres la description des droits du processus, l'adresse de l'espace mémoire qui lui est attribué, et un identifieur des fichiers qu'il manipule.

cherche sur l'expression et le moyen de vérifier des contraintes sur un plan IxTeT, que nous n'avons pas eu le temps d'aborder.

- La planification successive a été privilégiée par rapport à la planification concurrente parce qu'elle nous semble plus intéressante du point de vue de la sûreté de fonctionnement. En effet, elle privilégie moins la tolérance aux défaillances temporelles que la planification concurente, cette dernière activant des planificateurs en parrallèle pour gagner du temps en cas de délai de planification dépassé, et elle s'avère plus performante puisqu'un seul planificateur est actif dans le système. Elle est également plus facile à mettre en œuvre du point de vue de la synchronisation des processus IxTeT.
- Seule la diversification des modèles des planificateurs redondants a été réalisée. La diversification des heuristiques, voire l'utilisation de planificateurs différents (applicable à nos mécanismes de rétablissement à la seule condition que ces planificateurs fournissent en sortie un plan à partir d'un état initial et d'objectifs donnés), n'ont pas été implantées.
- IxTeT ne signale pas la fin de l'exécution de son plan après le bilan de la dernière action entreprise, mais poursuit son contrôle d'exécution temporel jusqu'à quelques secondes avant la borne maximale du temps imparti. Dans ce cas, la vérification d'objectifs en cours d'exécution ne permet pas de remplir les objectifs non-achevés, puisque le système n'a alors plus le temps de les accomplir. Pour résoudre ce problème, il faudrait améliorer notre version modifiée d'IxTeT de façon à ce qu'il envoie un message à FTplan dès que toutes les actions de son plan ont été accomplies.

Algorithme général

L'algorithme général de FTplan consiste en une procédure d'initialisation, suivie d'une boucle d'exécution qui s'exécute tant que restent des objectifs qui peuvent être remplis par le système.

La procédure d'initialisation connecte FTplan au *MessagePasseur*, initialise la représentation du système et des objectifs à partir de fichiers de configuration, et lance les différents IxTeT. La première planification du système est alors lancée, et FTplan met donc en place un chien de garde temporel avant d'entrer dans sa boucle principale d'exécution.

Cette boucle d'exécution, mettant en œuvre l'algorithme de planification successive présentée dans le Tableau 3.1 de la Section 3.1.2.1, est constituée de deux étapes majeures : l'attente d'un message du *MessagePasseur*, puis le traitement du message reçu. Le seul moyen de quitter la boucle, et donc de mettre un terme à l'exécution de la mission, est soit d'avoir un échec de tous les planificateurs dans l'élaboration d'un plan, auquel cas les capacités de tolérance du système ont été épuisées, soit d'avoir rempli tous les objectifs jugés possibles par FTplan, c'est à dire tous les objectifs dont l'échéance temporelle n'est pas encore dépassée. Une fois qu'une de ces conditions est remplie, FTplan tue les processus IxTeT et termine son exécution.

Ftplan utilise un temporisateur pour vérifier toutes les dix millisecondes la présence de message transmis par le *MessagePasseur* dans sa file d'attente des messages. Si un message est présent, il passe à l'étape de traitement, sinon il s'endort à nouveau pendant dix millisecondes, fonction système nanosleep(), afin d'économiser l'occupation du processeur. La durée du temps de sommeil est arbitraire,

bien que choisie pour être insignifiante par rapport au temps d'exécution d'un cycle d'IxTeT (de l'ordre de deux secondes), mais suffisamment importante pour ne pas surcharger le processeur par des réveils intempestifs. Si une replanification est en cours, l'attente de message peut être interrompue par l'échéance temporelle d'un chien de garde, et activer le rétablissement correspondant.

L'activité de traitement de message dépend du type de message reçu. Si le message est une requête d'action d'IxTeT ou un bilan d'action d'OpenPRS, FTplan met à jour sa représentation du système avant de transmettre le message. Si le message indique un échec de l'exécution du plan, FTplan prépare une replanification successive ou concurrente, suivant le mécanisme de rétablissement choisi. Si le message indique une fin de l'exécution du plan d'IxTeT, FTplan vérifie que tous les objectifs possibles ont bien été remplis. Dans le cas contraire, il relance une planification à partir du nouvel état et des objectifs restants.

État du système et des objectifs

Nous avons vu que FTplan devait posséder une représentation propre du système afin d'assurer son indépendance par rapport à IxTeT. Cette indépendance est nécessaire pour donner à chaque replanifaction un état initial qui n'est pas basé sur les données d'un processus IxTeT, potentiellement erroné. Elle permet également de valider en cours d'exécution la réalisation des objectifs de la mission.

Cette représentation doit donc disposer de suffisamment d'informations d'une part pour exprimer l'état actuel du système dans tous les différents modèles de planification utilisés, et d'autre part pour vérifier qu'un objectif a correctement été accompli. Elle est mise à jour à travers les bilans envoyés par OpenPRS, qui décrivent les dernières modifications apportées à l'état du système, et à travers les requêtes du processus IxTeT actif, qui décrivent les actions lancées par le système.

Mécanismes de détection

Deux mécanismes de détection sont directement mis en place dans la version actuelle de FTplan : un chien de garde temporel sur la planification, et une vérification de l'accomplissement des objectifs en cours d'exécution. La détection d'un échec d'exécution est réalisée soit par les modules GenoM et le contrôleur d'exécution OpenPRS (échec en valeur), soit par le contrôleur d'exécution temporel d'IxTeT (échec temporel) ; l'échec est ensuite signalé par IxTeT à FTplan au moyen d'un message via le *MessagePasseur*. L'analyseur de plan n'est pas implanté dans le prototype.

Le chien de garde temporel est mis en place par les mécanismes d'alarme du système d'exploitation Linux. En pratique, une alarme est mise en place au début d'une planification : le processus demande au système d'exploitation de lui envoyer un signal d'alarme au bout d'un certain délai temporel. Si ce délai est dépassé, FTplan arrête le IxTeT en cours d'exécution, et demande une planification à un autre processus IxTeT jusqu'à ce qu'un plan soit trouvé ou que tous les planificateurs aient échoué. Dès qu'un planificateur envoie un message de fin de planification à FTplan, l'alarme est désamorcée.

Une vérification des objectifs est systématiquement mise en place par FTplan après chaque action réussie par OpenPRS capable d'accomplir un objectif, ou de mettre en défaut un objectif précédemment atteint. Par exemple, un objectif de prise de photo en un point particulier est vérifié chaque fois qu'une action de prise

d'image demandée par IxTeT est réussie par OpenPRS. Cette vérification est obtenue d'une part en analysant l'état du système à la fin de l'action (par exemple pour vérifier que l'image a été prise au bon endroit) et d'autre part en vérifiant au lancement de chaque action si des actions incompatibles n'ont pas été exécutées simultanément (par exemple pour vérifier si le robot ne s'est pas déplacé pendant la prise d'image).

Mécanisme de rétablissement par planification successive

En pratique, une variable interne au composant FTplan indique en permanence quel planificateur est actuellement exécuté. Lors d'une replanification, cette variable change selon un ordre préétabli de façon à parcourir successivement tous les planificateurs. Lorsque FTplan reçoit une demande de replanification, il change la variable de numéro de modèle, et envoie la liste des assertions et des évènements correspondant à l'état actuel du système et aux objectifs restant à remplir. Le processus IxTeT ciblé envoie un message en fin de planification si un plan a été trouvé. Si la planification prend trop de temps ou si aucune solution n'a été trouvée, FTplan tue le processus IxTeT en échec, le relance, puis bascule sur un autre modèle. Si tous les processus IxTeT échouent à la suite, FTplan émet un message d'erreur et arrête l'exécution de la mission.

Le principe de ces mécanismes est illustré sur la Figure 3.8, qui présente un court scénario d'exécution.

3.2.2.3 Modifications nécessaires de l'existant

Nous décrivons ici les modifications qui ont été apportés aux composants de l'architecture LAAS pour permettre leurs interactions avec FTplan.

Modification d'IxTeT

Des modifications importantes ont été apportées au code d'IxTeT pour permettre l'intégration de FTplan dans l'architecture LAAS. Ces modifications incluent des changements mineurs liés à la synchronisation ou aux communications nécessaires pour nos mécanismes de rétablissement, mais aussi deux fonctionnalités plus importantes, facilitant notre implantation.

Les principaux changements mineurs sont les suivants :

- les requêtes émises par un processus IxTeT sont systématiquement envoyées à FTplan plutôt qu'à OpenPRS,
- après un échec de la réparation de plan et avant une replanification, un message est envoyé à FTplan pour lui permettre d'exécuter ses mécanismes de rétablissement ; après une planification réussie, IxTeT envoie un message à FTplan avant de lancer l'exécution du plan,
- une procédure d'initialisation supplémentaire a été mise en place. En effet, la version originale d'IxTeT exécute une planification dès son lancement, à partir d'un modèle et d'un état initial décrits dans des fichiers. Cette seconde procédure d'initialisation ne lit que les fichiers du modèle, et attend que FTplan lui envoie une demande de replanification et l'état courant du système pour planifier. Elle est par exemple utilisée pour relancer un processus IxTeT pendant l'exécution de la mission, après que ce processus dépasse l'échéance temporelle du chien de garde de la planification.

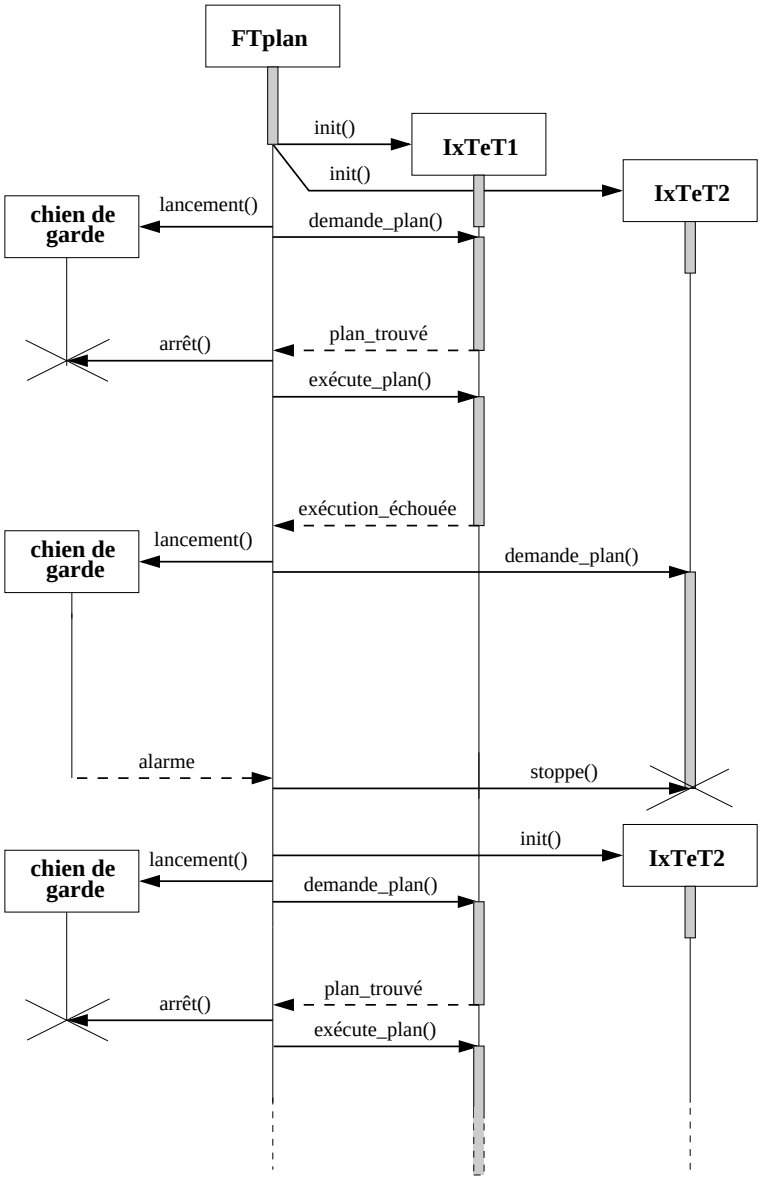

Figure 3.8: Planification successive avec FTplan

Les deux fonctionnalités que nous avons ajoutées sont l'élaboration d'un plan initial à partir d'un état envoyé par FTplan, et une procédure de replanification adaptée à cet état initial :

- Afin d'élaborer un nouveau plan initial conforme à la situation du système en cours d'exécution, IxTeT reçoit de FTplan une liste d'assertions et d'évènements (*hold* et *event*), décrivant tout d'abord l'état courant du système puis les objectifs restant à accomplir. IxTeT crée ensuite une structure de plan partiel, contenant le modèle d'actions et les heuristiques de contrôle qui sont donnés durant son initialisation, avant d'y insérer les assertions et évènements donnés par FTplan, ainsi que des contraintes temporelles de précédence. Il reçoit également de FTplan plusieurs informations nécessaires pour la replanification et la reprise de l'exécution, comme le temps courant de la mission.
- Une fonction de replanification simplifiée a été développée à partir de la fonction de replanification originelle d'IxTeT. Principalement, elle omet la récupération de l'état initial de replanification à partir du plan précédent, qui peut être erroné et n'est plus à jour, lui substituant le plan initial construit à partir des informations de FTplan.

Modification des procédures OpenPRS

Aucune modification du code d'OpenPRS n'a été nécessaire, mais trois modifications mineures ont été apportées dans les procédures décomposant les actions de haut niveau des modèles d'IxTeT :

- Les procédures relatives aux actions de tous les modèles utilisés sont chargés par OpenPRS au début de l'exécution, pour ne pas avoir à perdre de temps à chaque basculement de modèle. Cela implique que toutes les actions des différents modèles IxTeT aient des noms différents les unes des autres pour éviter toute ambiguïté (par exemple, si deux modèles comportent chacun une action appelée Communication, elles doivent être différenciées en Communication1 et Communication2).
- Les bilans des procédures relatives aux actions des modèles doivent systématiquement être envoyés à FTplan plutôt qu'à un processus IxTeT.
- Des informations supplémentaires sont remontées dans les bilans destinés à FTplan, puisque celui-ci possède une représentation du système généralement plus détaillée que celles des modèles IxTeT. Dans la pratique, un champ particulier est réservé dans les bilans à FTplan, contenant les informations supplémentaires nécessaires. Après l'avoir analysé, FTplan retire ce champ avant de transmettre le message à un processus IxTeT.

3.3 Conclusion

Nous avons proposé dans ce chapitre des mécanismes de tolérance aux fautes cherchant à répondre aux problèmes et aux exigences des systèmes autonomes. Ces mécanismes reposent principalement sur la diversification des modèles et des heuristiques des planificateurs, technique connue et validée dans la sûreté de fonctionnement informatique, mais dont l'efficacité appliquée aux systèmes autonomes n'est pas certaine, et demande à être vérifiée.

Ces mécanismes sont mis en place par un composant que nous avons appelé FTplan, qui réalise un traitement non trivial d'informations échangées entre planificateurs et contrôleurs d'exécution, tout en mettant en place des mécanismes d'alarme ou de synchronisation. L'ajout de ce composant a donc un impact potentiellement négatif sur les performances du système, qu'il faut étudier. De plus, la modification du composant de planification, nécessaire à l'implantation de FTplan, et l'utilisation conjointe de plusieurs modèles de planification altèrent inévitablement le comportement du système.

Il est donc important de mesurer l'efficacité des mécanismes de tolérance aux fautes proposés, ainsi que leur impact sur les performances du système.

Deux améliorations distinctes peuvent également être apportées au composant FTplan pour le rendre plus indépendant de l'application, et réduire les risques de propagation d'erreurs venant du planificateur.

Tout d'abord, la représentation de l'état du système ainsi que l'analyse des bilans OpenPRS et des requêtes IxTeT, toutes dépendantes de l'application, sont encore directement référencées dans le code source de FTplan. Un travail de développement supplémentaire est ainsi nécessaire pour déplacer ces informations dans des fichiers de configuration lus par le composant FTplan pendant son initialisation.

De plus, la détection d'un échec temporel dans l'exécution du plan est réalisée par le contrôle d'exécution temporel du composant IxTeT, pouvant causer des risques de propagation d'erreur au composant FTplan qui agit à partir de cette détection. Ce risque nous paraît faible en regard des fautes que nous considérons (fautes de conception dans le modèle ou les heuristiques du planificateur), puisqu'elles ont un impact principalement sur le choix et l'ordonnancement des actions de haut niveau du système, mais il reste un défaut de conception à corriger, notamment pour permettre l'application de notre prototype à des mécanismes de planification diversifiés.

Finalement, il faut noter que les mécanismes proposés sont adaptés à un système autonome de type hiérarchisé, pouvant être appliqués à d'autres architectures que l'architecture LAAS utilisée dans notre implémentationpour notre prototype. Par contre, leur application à un système de type multi-agents peut être compliquée par deux difficultés. D'une part, le modèle de chaque agent a souvent des données en commun avec ceux d'autres agents, ce qui peut compliquer le changement de modèles diversifiés. D'autre part, la multiplication des mécanismes de planification dans l'architecture nécessite une multiplication des mécanismes proposés, ce qui peut avoir des coûts de développement et de performance significatifs.

Ce qu'il faut retenir

1. Dans le but de tolérer les fautes de conception dans les modèles et les heuristiques de planificateurs, nous proposons quatre mécanismes de détection et deux mécanismes de rétablissement, gérés par un composant spécifique du système que nous avons appelé FTplan.
2. Les mécanismes de détection sont : un chien de garde temporel sur la durée

de la planification, un analyseur de plan à la sortie du planificateur, une détection d'échec d'action déjà utilisée en robotique pour la robustesse, et une vérification en ligne des objectifs réalisés par le système.

3. Les mécanismes de rétablissement utilisent plusieurs planificateurs diversifiés, et peuvent être mis en place selon deux politiques : exécution successive ou concurrente des planificateurs.
4. Un composant FTplan a été développé et intégré à l'architecture hiérarchisée LAAS. Des modifications dans certains composants de l'architecture ont été nécessaires, en particulier dans le mécanisme décisionnel IxTeT. Afin d'assurer un développement dans un délai raisonnable, nous avons limité la mise en œuvre du composant au rétablissement par planification successive, et à la détection d'erreur par chien de garde temporel, par détection d'échec d'action, et par vérification en ligne des objectifs.

Chapitre 4

Évaluation des mécanismes de tolérance aux fautes

Nous avons proposé dans le chapitre précédent plusieurs mécanismes susceptibles d'améliorer la tolérance aux fautes de conception dans les connaissances d'un planificateur. Nous avons présenté un composant mettant en œuvre ces mécanismes, et décrit son implantation dans un système autonome réel. Dans ce chapitre, nous proposons une étude cherchant à évaluer l'impact de ce composant sur le comportement du système autonome cible : le coût de son utilisation en terme de performance, et l'efficacité de ses mécanismes de tolérance aux fautes confrontés à des fautes réelles.

Nous présentons tout d'abord l'environnement et les principes que nous avons utilisés pour réaliser les expériences nécessaires à cette évaluation, avant de présenter les résultats de notre campagne d'expérimentation.

4.1 Environnement d'évaluation

Notre environnement d'évaluation repose sur l'*injection de fautes* et la *simulation du robot physique*. L'injection de fautes est nécessaire pour confronter le système avec des fautes simulant celles pouvant survenir lors de l'expression des connaissances du planificateur. Nous n'avons pas trouvé d'études approfondies sur les fautes dans les connaissances des mécanismes décisionnels, mais des travaux existants sur les langages impératifs [Daran 1996] et les systèmes d'exploitation [Jarboui 2003] ont montré que des mutations[1] peuvent simuler efficacement des fautes logicielles réelles. Le logiciel du système autonome cible est exécuté sur un robot simulé plutôt qu'un robot réel principalement pour deux raisons :

- Un grand nombre d'expériences est nécessaire pour réaliser une évaluation significative : les expériences réalisées sur des systèmes réels prennent généralement plus de temps à exécuter, demandent plus de matériel, et sont plus difficiles, voire impossibles, à automatiser.

[1] Une mutation est une modification syntaxique élémentaire dans un code existant.

- Les actions d'un robot réel peuvent avoir des conséquences dangereuses durant une expérience : comme nous injectons des fautes dans le système autonome cible, nous ne pouvons pas prévoir son comportement, et le système peut causer des dégâts sur lui-même ou son entourage. Notons que la simulation que nous utilisons permet de conserver les composants logiciels d'un système autonome réel jusqu'au plus bas niveau fonctionnel.

Les principaux attributs d'une campagne d'injection de fautes peuvent être caractérisés par les ensembles FARM [Arlat et al. 1990] : comme domaine d'entrée, l'*ensemble de fautes* F et l'*ensemble d'activités* A demandées au robot pendant les expériences, et, comme domaine de sortie, les *relevés* directs des expériences R, et les *mesures* M qui en sont dérivées. Dans cette section, nous présentons tout d'abord le support logiciel de notre environnement d'évaluation, avant de détailler successivement les différents ensembles FARM de caractérisation des expériences.

4.1.1 Environnement logiciel

L'environnement logiciel utilisé lors de nos expériences est représenté dans la Figure 4.1. Il incorpore deux composants permettant la simulation des composants matériels et de l'environnement physique d'un robot (Gazebo et Pocosim) ainsi que les composants de l'architecture LAAS présentés dans la Section 3.2.1.1.

Le simulateur de robot Gazebo[2], disponible en logiciel libre sous la *GNU General Public License*, est utilisé pour simuler les composants physiques du robot, son environnement, et leurs interactions. En particulier, il propose la simulation de la cinématique d'objets rigides (accélération, vitesse, collision...), et de capteurs et actionneurs typiques de robot (caméras, capteurs-lasers, odométrie, roues motorisées...). Ce simulateur prend en entrée un fichier de description de l'environnement initial, comprenant d'une part la masse et la position des obstacles (et éventuellement leur vitesse pour des obstacles dynamiques), et d'autre part la position initiale et une description du robot, comprenant la liste de ses capteurs.

La bibliothèque Pocosim [Joyeux et al. 2005] fournit un pont logiciel entre les modules GenoM qui contrôlent l'architecture matérielle du robot réel et le robot simulé dans Gazebo. Elle transforme les requêtes matérielles des modules en actions ou mouvements à exécuter par le robot simulé, et transmet aux modules les données de capteurs simulées par Gazebo.

Le système autonome cible reproduit un système autonome existant, développé au LAAS et implanté sur un robot de type ATRV (*All Terrain Robotic Vehicle*) commercialisé par iRobot. Il utilise des procédures OpenPRS et des modules GenoM directement repris du système autonome existant. FTplan est intégré à l'architecture et gère deux instances d'IxTeT, utilisant deux modèles diversifiés ainsi que présenté dans la Section 3.2.2. Le robot dispose d'un chassis à roues motorisées non-holonome, de deux caméras disposées sur un banc orientable, et d'un capteur-laser de proximité.

Le composant R2C n'est pas implanté dans le système autonome simulé, car il est incompatible dans sa version actuelle avec le pont logiciel Pocosim. En effet, Pocosim enveloppe les modules GenoM pendant leur exécution, masquant au

[2] "The player/stage project", http ://playerstage.sourceforge.net

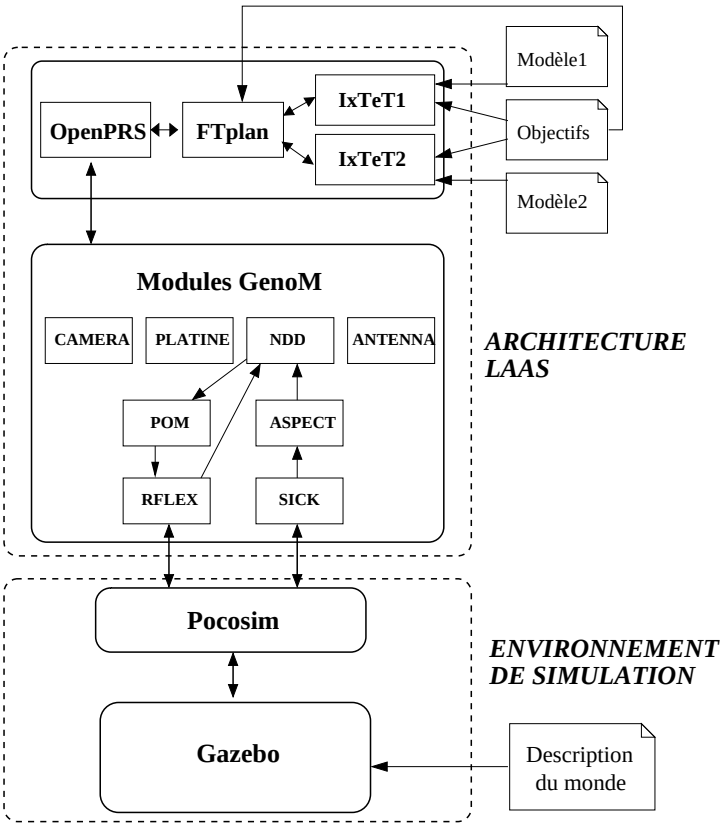

Figure 4.1: Environnement logiciel des expériences

reste du système un espace de mémoires partagées utilisées par les modules pour s'échanger des données. Dans le système réel, R2C récupère des informations sur la couche fonctionnelle directement dans ces mémoires partagées, alors qu'il n'y a pas accès dans le système simulé. La suppression du composant R2C diminue la robustesse globale de l'architecture LAAS, avec ou sans le composant FTplan, et probablement, dans une moindre mesure, sa tolérance aux fautes. Cependant, l'architecture LAAS, même sans le composant R2C, nous semble une architecture suffisamment représentative des systèmes autonomes pour évaluer l'efficacité du composant FTplan proposé dans le chapitre précédent.

La partie fonctionnelle du système autonome comporte huit modules GenoM, qui peuvent être distingués selon trois groupes :

- Les modules SICK et RFLEX utilisent la bibliothèque Pocosim pour contrôler des composants matériels simulés par Gazebo. SICK contrôle le capteur-laser du robot, tandis que RFLEX supervise le mouvement des roues et l'odométrie.
- NDD, ASPECT et POM sont des modules logiciels qui utilisent SICK et RFLEX pour réaliser les fonctions de navigation et d'évitement d'obstacles. POM fusionne habituellement les données de position des différents capteurs afin de fournir au système une position unique gérant les incerti-

tudes et les imprécisions des équipements matériels ; dans notre cas, seule l'odométrie est utilisée, et POM sert principalement de pont entre RFLEX et NDD. ASPECT crée une carte de l'environnement immédiat du robot en tenant compte des données du capteur-laser. NDD utilise à son tour les données de POM et ASPECT pour générer des données de navigation, à partir d'un algorithme basé sur les diagrammes de proximité [Minguez & Montano 2004].

- Les modules PLATINE et ANTENNA sont destinés au contrôle des composants matériels qui ne sont pas simulés par Gazebo (un moteur d'orientation des caméras, et un matériel de communication avec un orbiteur). Les caméras du système, dont le contrôle est prévu par le module CAMERA, peuvent théoriquement être simulées par Gazebo, mais le pont logiciel correspondant n'est pas développé. Le code source de ces trois modules a donc été modifié afin de fournir une simulation simple de leur comportement, leurs requêtes étant supposées réussir systématiquement.

L'environnement simulé est moins complexe qu'un environnement réel : malgré le simulateur physique de Gazebo, les interactions entre le robot et l'environnement sont simplifiées, et les situations adverses moins nombreuses et diversifiées. Cependant, nous estimons que la simulation des composants et des interactions physiques de notre système reste compatible avec l'évaluation des mécanismes de tolérance aux fautes, parce que nous souhaitons évaluer les performances de la couche décisionnelle du système plutôt que la couche matérielle. En effet, une exécution plus réaliste, bien que préférable en terme de représentativité ou de diversification des situations adverses, ne nous semble pas indispensable tant qu'il est possible d'étudier le comportement et la réaction des mécanismes décisionnels intégrés dans le système autonome complet, et de mettre en jeu de situations adverses externes au système susceptibles de stresser leurs différentes capacités.

Une restriction importante liée à notre expérimentation est l'impossibilité d'accélérer le temps de simulation. En effet, bien qu'il soit en théorie possible d'accélérer le temps dans le simulateur physique Gazebo, les ressources du planificateur IxTeT restent limitées par la puissance du processeur, et le processus de planification ne peut donc pas être accéléré. Ceci nous oblige à exécuter chaque simulation en temps réel, et restreint donc le nombre de simulations que nous pouvons faire en un temps donné.

4.1.2 Ensemble d'activités

Nous appelons *activité* une mission demandée au robot dans un environnement donné. Nous avons choisi de simuler un robot d'exploration spatiale, effectuant des missions définies par trois types d'objectifs : prendre des photos scientifiques à certaines positions spécifiées, effectuer des communications avec un orbiteur pendant des fenêtres de visibilité temporelles spécifiées, et revenir au point de départ avant une date butoir en fin de la mission.

Il faut noter que certaines des fonctionnalités du robot simulé sont plus appropriées à des utilisations de robotique d'intérieur et n'auraient pas été implantées sur un robot d'exploration spatiale, notamment la détection d'obstacles par capteur laser et la navigation basée sur des diagrammes de proximité. Nous avons

cependant choisi cette activité car d'une part elle donne un bon exemple d'application concrète requérant la prise de décision sur des aspects temporels et physiques, et d'autre part un premier modèle était déjà développé et testé pour ce contexte.

Afin d'activer autant que possible les différentes fonctionnalités du système autonome cible, et plus généralement pour répondre aux problèmes présentés dans la Section 2.3.2.1, il est nécessaire de tester les réponses du système dans différentes missions et face à différents environnements. Idéalement, les nombres de missions et d'environnements doivent être élevés afin de fournir un ensemble aussi représentatif que possible de l'infinie diversité des contextes d'exécution envisageables. Cependant, nous devons appliquer le même ensemble d'activités à chaque faute que nous injectons, et, dans le but de garder un nombre raisonnable d'expérimentations à exécuter, nous avons développé quatre missions et quatre environnements, soit seize activités en tout.

Pour prendre au compte au moins partiellement le problème d'indéterminisme dans l'exécution du système, nous exécutons plusieurs fois chaque activité afin d'obtenir une moyenne des résultats sur plusieurs expériences. Un nombre important d'exécutions équivalentes aurait été nécessaire pour obtenir un résultat statistiquement représentatif, mais nous avons choisi de réaliser seulement trois exécutions équivalentes, encore une fois pour limiter le nombre total d'expériences à réaliser. Pour chaque mutation, le nombre total d'expériences exécutées est donc de 48 (4 missions x 4 environnements x 3 exécutions).

Nous détaillons dans cette section les différentes missions et environnements qui font partie de notre environnement d'évaluation, avant d'introduire les deux modèles de planification associés.

4.1.2.1 Missions et environnements

Une mission consiste à photographier un certain nombre de cibles à différents endroits, exécuter une communication durant chacune des fenêtres de visibilité spécifiées, et revenir à la position de départ avant la fin de la mission. La durée maximale de chaque mission est fixée à 800 secondes, durée suffisante pour réaliser les objectifs de chaque activité et assez courte pour permettre de nombreuses expériences dans des délais raisonnables, et les positions envisagées pour les cibles à photographier se concentrent sur une grille de 12 mètres sur 12 (la position initiale du robot étant le point central de la grille). Une mission particulière est donc définie, d'une part, par le nombre et la position des photos à prendre et, d'autre part, par le nombre et l'occurrence temporelle des fenêtres de visibilité pendant lesquelles communiquer. La figure 4.2 présente les 4 missions que nous avons utilisées pour notre évaluation.

La mission 1 demande de prendre 3 photos rapprochées, et 2 communications. La mission 2 demande les mêmes communications et autant de photos, mais plus éloignées les unes des autres. La mission 3 demande 4 photos rapprochées et trois communications, dont deux à la suite l'une de l'autre. Finalement, la mission 4 demande 5 photos éloignées, et quatre communications.

Le monde dans lequel évolue le robot est une surface plane, potentiellement occupée par des obstacles statiques. Cet environnement simple s'est avéré suffisant pour stresser le système, voire provoquer des échecs dans l'exécution des

missions, comme nous le verrons dans la Section 4.2. La description d'un environnement consiste à définir la forme et la position des obstacles présents. Ces obstacles, comme la position des photos à effectuer, sont concentrés dans un carré de 12 mètres sur 12 autour de la position initiale du robot et sont définis comme faisant un mètre de haut afin de pouvoir être détectés par le capteur-laser du robot[3]. Les 4 environnements utilisés pour l'activité de notre évaluation sont présentés dans la figure 4.3.

Le premier environnement, vide, ne pose pas de situations adverses au système. Le second environnement comprend quelques obstacles cylindriques, facilement évitables par le système. Le troisième environnement comprend plus d'obstacles, certains très proches de cibles photos des missions 3 et 4, et pouvant donc poser des difficultés supplémentaires au système. Finalement, le quatrième environnement comporte des obstacles rectangulaires, qui peuvent dans certaines circonstances mettre en défaut l'algorithme de navigation, en entravant définitivement le cheminement du robot. Une fois bloqué par un de ces obstacles, le robot n'a plus dde moyens de reprendre son déplacement : ni réparation de plan, ni replanification, ni même changement de modèle ne peuvent lui permettre de repartir. Dans ce dernier environnement, le succès de la mission dépend donc plus de circonstances favorables que de l'exactitude du modèle. Néanmoins, cet environnement nous semble intéressant puisqu'il permet de voir le comportement du système confronté à des situations adverses extrêmement défavorables face auxquelles il ne peut pas répondre, une éventualité qui est toujours possible dans l'utilisation de systèmes autonomes.

4.1.2.2 Modèles de planification

Les deux modèles présentés dans cette section sont détaillés dans l'Annexe A.

Le premier modèle que nous utilisons, appelé par la suite Modèle1, a été développé pour tester l'intégration de l'architecture LAAS, et en particulier différents mécanismes du planificateur IxTeT, dans le cadre de missions d'exploration spatiale présenté précédemment. Il a ainsi été exécuté plusieurs fois sur un robot réel en intérieur et en extérieur avant notre évaluation, ce qui nous a donné une certaine confiance dans son comportement, et nous avons choisi de l'utiliser comme première alternative dans notre mécanisme de rétablissement par planification successive.

Un second modèle, appelé Modèle2, a été développé au cours de nos travaux, comme redondance pour la tolérance aux fautes. Il a été conçu de façon à maximiser sa diversification par rapport au premier, en forçant certains choix de développement. En particulier, la localisation (c'est-à-dire la caractérisation de la position du robot) est définie de façon symbolique plutôt que selon les coordonnées cartésiennes utilisées dans Modèle1, et, à l'opposé, la position de la platine sup-

[3]Dans la pratique, des obstacles lointains continus sont également nécessaires au fonctionnement du module GenoM SICK contrôlant le capteur-laser. En effet, une absence complète de faisceaux lasers réfléchis par des obstacles et reçus par le capteur, possible en pratique dans un monde simulé sans aucun obstacle, est considérée par le module GenoM comme un symptôme de défaillance du capteur. Dans nos environnements, ces obstacles sont placés à 20 mètres du robot, distance suffisante pour éviter leurs interactions.

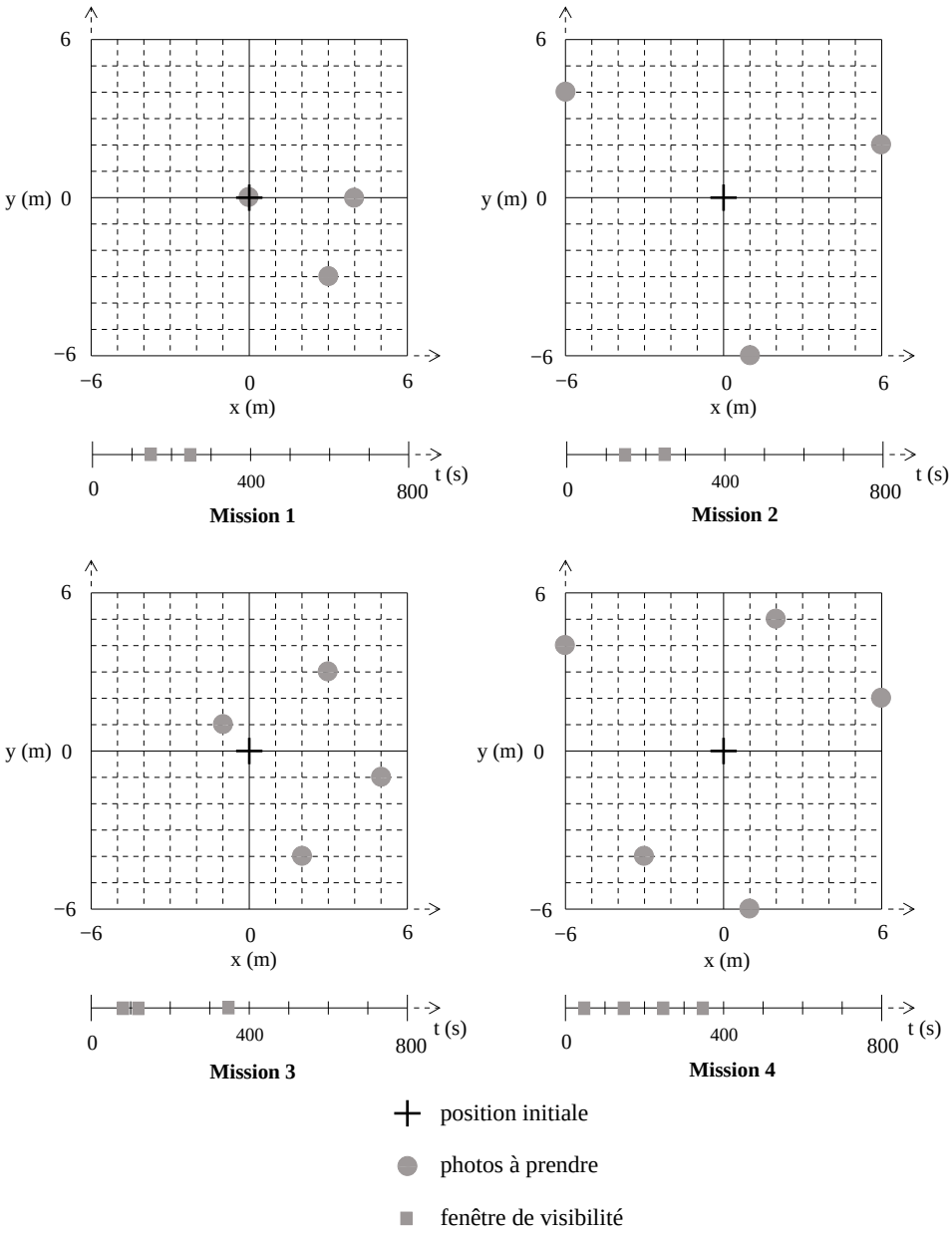

Figure 4.2: Missions de l'activité

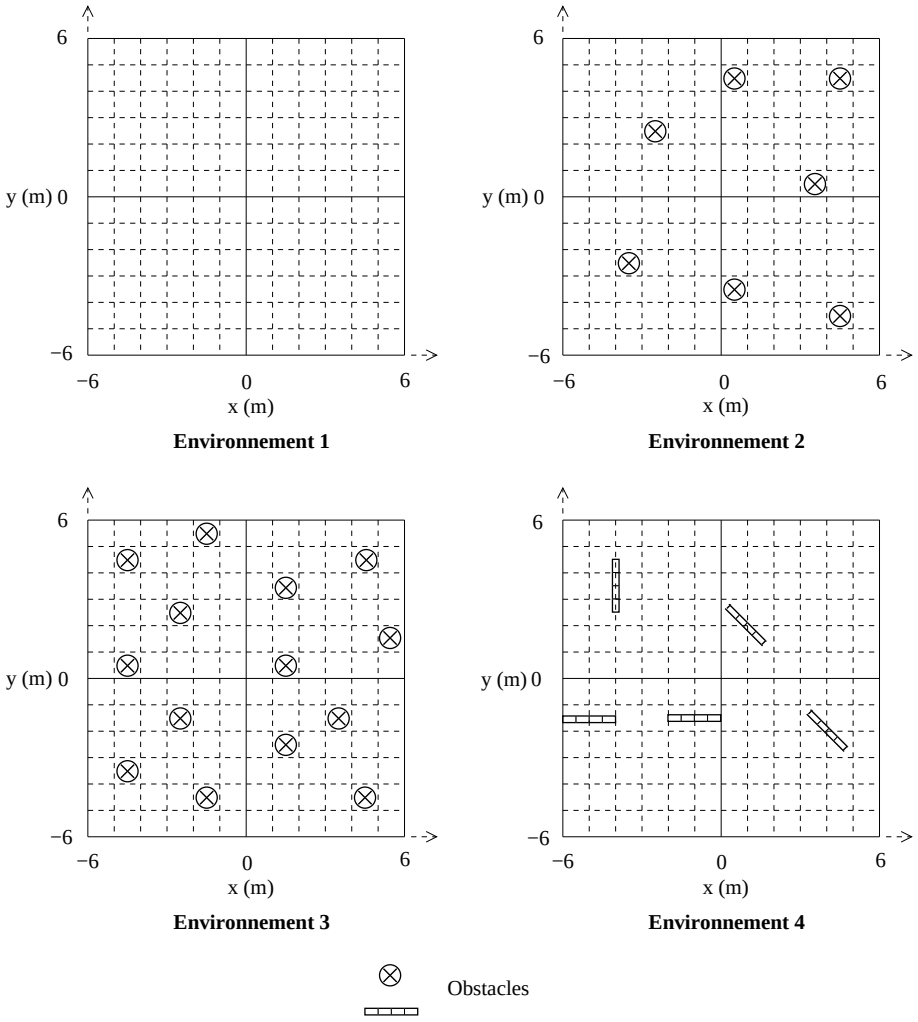

Figure 4.3: Environments de l'activité

portant la caméra est définie de façon numérique réelle plutôt que symbolique. De nombreux autres aspects ont été diversifiés, en particulier des contraintes et des paramètres redondants ont été supprimés, et des attributs complémentaires ont été fusionnés.

Afin de limiter le travail de développement lié à Modèle2, les procédures OpenPRS décrivant la décomposition de chaque action en tâches de niveau fonctionnel ont été directement réutilisées. Les deux modèles partagent donc les mêmes actions de haut niveau. Une action de déplacement supplémentaire a cependant été ajoutée dans Modèle2 pour optimiser la planification avec la localisation symbolique.

Modèle1 comprend 6 actions, définies autour de 9 attributs du système. Modèle2 comprend 7 actions (les 6 actions de Modèle1 plus une action supplémentaire de déplacement) et 10 attributs. Les six actions communes aux deux modèles sont :

- une action d'initialisation de mouvement, permettant au système de réinitialiser les modules et les composants matériels liés à la navigation après une erreur de navigation,
- une action d'initialisation de la platine des caméras,
- une action de déplacement de la platine des caméras, permettant d'alterner entre une position nécessaire pour la prise d'image, et une position nécessaire au déplacement,
- une action de prise de photo,
- une action de communication,
- une action de déplacement, permettant au système de se rendre d'un point A à un point B ; dans Modèle1 ces points peuvent être quelconques, tandis que dans Modèle2 ils spécifient les points singuliers de la mission (position initiale et emplacement des cibles à photographier). La seconde action de déplacement de Modèle2 est utilisée lorsque le système se retrouve positionné à un point quelconque, par opposition aux points singuliers de la mission (par exemple après un déplacement échoué), et permet de revenir de ce point à un point singulier.

Le Tableau 1 détaille le contenu syntaxique de chaque action des deux modèles, mettant en évidence une différence de contenu importante. En particulier, il dénombre le nombre de variables, de contraintes numériques, d'évènements et d'assertions pour chacune des actions.

La variation des nombre de contraintes pour les actions de déplacement de Modèle2 est due à la nécessité de spécifier les distances entre les différentes positions symboliques. Ce nombre de contraintes dépend donc du nombre de cibles à photographier, et donc de la mission considérée. Les chiffres donnés correspondent pour la borne minimale à une mission à 3 points singuliers, et pour la borne maximale à une mission à 6 points singuliers.

4.1.3 Ensemble de fautes

Afin d'évaluer la performance et l'efficacité des mécanismes de tolérance aux fautes proposés, nous injectons des fautes par mutation directement dans le modèle de planification de notre système autonome. Dans nos travaux, nous nous

	Modèle1				Modèle2			
	V	C	E	A	V	C	E	A
déplacement	19	32	6	5	8	19-45	4	5
	-	-	-	-	8	67-116	5	5
dép. platine	4	4	2	2	6	5	4	3
prise photo	5	4	2	4	4	3	1	4
communication	3	2	2	3	4	3	1	3
init. dép.	2	1	2	2	2	1	1	1
init. platine	2	1	2	2	3	2	2	1

(V : variables ; C : contraintes ; E : évènements ; A : assertions)

Tableau 1: Comparaison syntaxique de Modèle1 et Modèle2

sommes concentrés sur l'injection de fautes dans Modèle1.

A partir de la syntaxe du formalisme IxTeT, nous avons identifié cinq types de mutation possible :

- *Substitution de valeurs numériques* : chaque valeur numérique est successivement remplacée par un élément d'un ensemble de valeurs numériques, comprenant (a) toutes les valeurs numériques comprises dans le modèle, (b) un ensemble de valeurs particulières des nombres réels (telles que 0, 1 et -1), (c) un ensemble de valeurs sélectionnées aléatoirement.
- *Substitution de variables* : comme la portée d'une variable est limitée à l'action IxTeT dans laquelle elle est déclarée, les variables numériques (respectivement temporelles) sont remplacées par toutes les variables numériques (respectivement temporelles) contenues dans la même action.
- *Substitution de valeurs d'attribut non-numériques* : les valeurs des attributs sont remplacées par les autres valeurs possibles du domaine de l'attribut considéré.
- *Substitution des opérateurs du langage* : en plus des opérateurs classiques sur les valeurs numériques et temporelles, le formalisme IxTeT utilise des opérateurs spécifiques sur les actions (par exemple `nonPreemptive` indique qu'une action ne peut pas être interrompue), ou sur des contraintes de durée (par exemple `contingent` indique que la durée d'une tache est incertaine et ne doit pas être réduite dans la conception d'un plan).
- *Suppression d'une contrainte* : une contrainte choisie aléatoirement dans le modèle est supprimée.

Les quatre premiers types de mutations ont été générés à l'aide de l'outil SESAME [Crouzet et al. 2006]. Cet outil établit une base de données des mutations en réalisant toutes les substitutions spécifiées dans des fichiers, appelés tables de substitutions, et en compilant le fichier muté. Les mutants dont la compilation aboutit à une erreur, ou dont le fichier compilé est identique au fichier non-muté (par exemple si la mutation a portée sur des commentaires) ne sont pas considérés. La Figure 4.4 présente un exemple de table de substitution.

Le principe de l'outil SESAME est simple : il parcourt le fichier cible en recherchant des chaînes de caractères, qu'il substitue par d'autres chaînes de caractères. Chaque ligne de la table de mutation, constituée de deux ensembles, dé-

% substitution de valeurs numériques
{«-oo», «+oo», «0.0», «60», «1»} → {«-1», «-4», «26.3»}

% substitution de variables physiques ou temporelles
{« ?obj», « ?x», « ?y»}
{« ?t_start», « ?t_end»}

% substitution de valeurs d'attribut
{«downward», «straight», «other»}
{«none», «done», «doing»}

% substitution d'opérateurs du langage
{«nonPreemptive», «latePreemptive»} → {« »}
{«contingent»} → {« »}

Figure 4.4: Exemple de table de mutation

finit plusieurs substitutions. Le premier ensemble, à gauche de l'opérateur "→", liste les chaînes de caractères qui vont être recherchées et remplacées dans le modèle par l'outil SESAME : chaque élément de cet ensemble sera substitué par tous les autres éléments de ce même ensemble. Ainsi, le premier ensemble de la première substitution décrit déjà 20 substitutions. Le second ensemble, à droite de l'opérateur "→" et facultatif, décrit des chaînes de caractères supplémentaires qui ne seront pas recherchées dans le modèle, mais seront substituées à toutes les chaînes de caractères du premier ensemble. Ce second ensemble est en particulier nécessaire quand on veut éliminer un opérateur optionnel du langage (comme `nonPreemptive` ou `contingent`) : on le substitue alors par une chaîne de caractère vide.

La simplicité de l'outil SESAME, et le fait qu'il procède par simple recherche de chaîne de caractère plutôt que par analyse grammaticale, peut générer des mutations incorrectes : par exemple, si une variable à substituer est nommée "`x`", SESAME remplacera tous les chaînes de caractère correspondantes qu'il trouvera dans le fichier cible, y compris celles contenues dans des commentaires, ou d'autres noms de variables. Ces mutations incorrectes sont toutefois écartées en majorité avant d'être inscrites dans la base de données, car elles provoquent quasi-systématiquement des erreurs de compilation.

En tout, plus de 1000 mutations compilables ont été générées à partir de notre modèle de fautes. Afin d'avoir une meilleure représentativité des fautes injectées, nous avons choisi de ne considérer que les mutants :

- qui sont capables de trouver un plan dans au moins une des quatre missions proposées (nous considérons que des modèles qui échouent systématiquement à trouver un plan seraient facilement détectés durant le développe-

ment, pendant la phase de validation),
- dont les résultats ne sont pas équivalents à ceux du modèle originel non muté (si les résultats sont les mêmes, nous considérons que la faute injectée n'a pas eu d'impact dans les activités exécutées).

De manière plus générale, nous définissons comme équivalents dans notre activité deux modèles qui ont le même ensemble d'exécutions possibles. Dans la pratique, la détermination de cette équivalence est un problème particulièrement difficile, sur lequel nous revenons dans la Section 4.2.2.1.

4.1.4 Relevés et mesures

De nombreux fichiers de journaux sont générés par une expérience : les données de position simulées par Gazebo, les journaux des différents modules GenoM, des composants OpenPRS, IxTeT, et éventuellement FTplan. Pour une expérience, ces fichiers peuvent prendre une place de 4 à 16 méga-octets, une série complète de 48 expériences prenant environ 320 Mo[4].

Il est difficile d'établir, à partir de ces différents relevés, une mesure unique significative représentant l'efficacité d'un système. En effet, contrairement à des expériences d'injection de fautes sur des systèmes informatisés plus classiques, le résultat d'une expérience ne peut pas être facilement dichotomisé en réussite ou en échec. Comme nous l'avons mentionné dans la Section 2.3.2.1, un système autonome est confronté à des situations et des environnements au moins partiellement inconnus, et certains de ses objectifs peuvent être difficiles, voire impossibles, à accomplir dans certains contextes.

Pour caractériser le comportement d'un système face à une activité donnée, nous avons défini différents relevés objectifs et compréhensibles basés sur les données brutes d'expérience :

- u_p, u_c et u_h caractérisent les *objectifs réussis* pendant une mission : u_p représente le nombre de photographies réussies, u_c le nombre de communications réussies, et u_h indique si le robot est bien revenu à sa position initiale à la fin de la mission (u_h = 1 si le retour est réussi, u_h = 0 sinon). De la même façon, v_p, v_c et v_h indiquent le nombre d'objectifs (respectivement de photographies, de communication, et un retour) demandés au système dans la mission. A partir de ces mesures, nous définissons encore le *quotient d'objectifs réussis* pour chaque type d'objectifs $\sigma_i = \frac{u_i}{v_i}$, et le quotient d'objectifs échoués $\phi_i = 1 - \sigma_i = \frac{v_i - u_i}{u_i}$ (avec $i \in \{p, c, h\}$). Dans la suite de ce mémoire, nous exprimons ces deux quotients sous forme de pourcentage.
- κ caractérise la *défaillance du planificateur* : $\kappa = 1$ si le processus de planification a causé une erreur non-récupérée par le système d'exploitation (par exemple une faute de segmentation), ou s'il a défailli temporellement en étant incapable de fournir un plan à temps pour le système.
- μ caractérise l'*échec de la mission*, une mission étant considérée échouée si un ou plusieurs objectifs n'ont pas été atteints. Formellement, on a donc : $\mu = 1$

[4]Nous parlons ici de données non-compressées. Ces données étant des fichiers de texte, le gain de compression est de l'ordre de 10, amenant une série complète d'expériences à une quarantaine de méga-octets.

si $\forall i \in \{p, c, h\}$, $\sigma_i = 1$, et $\mu = 0$ sinon. Par opposition, ν caractérise le *succès de la mission*, avec $\nu = 1 - \mu$.

- R caractérise le *nombre de replanifications* effectuées par le système pendant une expérience. Dans le cas d'un système équipé de FTplan, ce nombre correspond également au nombre de basculements de modèle effectués.

Deux relevés de performance sont encore considérés :

- le *temps* T d'activité physique du système (c'est-à dire le temps durant lequel le système réalise des actions, l'activité du planificateur n'étant pas prise en compte),
- la *distance* D parcourue par le système pendant une expérience.

Ces deux derniers relevés sont moins significatifs que les précédents, puisqu'ils ne peuvent servir à comparer directement que deux expériences sur la même activité, de mêmes quotients d'objectifs réussis σ_p, σ_c, et σ_h. Ils permettent néanmoins de faire ressortir des différences de comportement significatives entre deux exécutions, parfois impossibles à observer avec les relevés précédents, et de mieux comprendre certains échecs de mission. En effet, un temps d'activité physique faible laisse présager que le système a rencontré un problème inattendu tôt dans son exécution, qu'il n'a pas réussi à surmonter.

Pour identifier exactement chaque expérience considérée, trois paramètres sont associés aux relevés :

1. La mission demandée au système pendant l'expérience. Dans notre environnement d'évaluation, cette mission peut être M1, M2, M3 ou M4, identifiant respectivement les missions un, deux, trois et quatre de la Section 4.1.2.1.
2. L'environnement (ou monde) dans lequel évolue le système pendant l'expérience. Dans notre environnement d'évaluation, ces mondes (*worlds*) peuvent être W1, W2, W3 ou W4, identifiant respectivement les environnements un, deux, trois et quatre de la Section 4.1.2.1.
3. Un identifiant indiquant quelle faute a été injectée dans le modèle du planificateur (ou dans le modèle utilisé comme première alternative pour un système comportant FTplan). Le symbole $\emptyset$ est utilisé pour indiquer qu'aucune faute n'a été injectée.

Par exemple, le quotient de photographies réussies dans une expérience sans faute injectée utilisant comme activité la mission un et l'environnement quatre est représentée par σ_p(M1,W4,$\emptyset$). Il est important de noter que, à cause de l'indéterminisme des expériences, les relevés présentés sont en fait des réalisations de variables aléatoires sur l'ensemble d'exécutions possibles $\mathscr{E}$.

Nous étendons maintenant les notations précédentes à la définition de mesures correspondant aux moyennes arithmétiques des différents relevés. Soient :

1. $\mathscr{M}$ un ensemble des missions,
2. $\mathscr{W}$ un ensemble des mondes,
3. $\mathscr{A} = \mathscr{M} \times \mathscr{W}$ un ensemble d'activités,
4. $\mathscr{F}$ un ensemble de fautes,
5. $\mathscr{Y} = \mathscr{A} \times \mathscr{F}$ un ensemble d'expériences.

On définit alors $\overline{\sigma_p}(\mathscr{A}, \mathscr{F})_{(n)}$ comme la moyenne arithmétique des quotients de photographies réussies sur n exécutions de l'ensemble d'expériences $\mathscr{Y}$.

Dans la suite de ce manuscrit, l'ensemble d'activité $\mathscr{A}$ sera représenté :

- soit par la notation $\{\mathscr{M}, \mathscr{W}\}$, représentant le produit cartésien $\mathscr{M} \times \mathscr{W}$,
- soit par un ensemble d'activités spécifiques MiWj (avec (i,j) $\in (1,2,3,4)^2$),
- soit par une combinaison des deux.

Par exemple, $\{(M1, M2), W2\}$ représente toutes les missions M1 et M2 exécutées dans le monde W2, et $\{((M1, M3), (W1, W2, W3)) \backslash (M1W1, M3W2)\}$ toutes les missions M1 et M3 exécutées dans les mondes W1, W2 et W3, à l'exception des activités M1W1 et M3W2.

Ainsi, $\overline{\sigma_p}(\{(M1, M2), W1\}, \emptyset)_{(3)}$ représente la moyenne arithmétique des quotients de photographies réussies lors de 3 exécutions dans chacune des 2 activités M1W1 et M2W2, soit 6 expériences en tout, sans faute injectée.

Dans la Section 4.1.2, nous avons choisi de réaliser trois exécutions d'une même activité, et nous posons donc trois comme valeur par défaut du nombre d'exécutions considéré dans nos mesures. Ainsi, $\overline{\sigma_p}(M3, W2, \emptyset) \equiv \overline{\sigma_p}(M3, W2, \emptyset)_{(3)}$. Pour simplifier encore notre notation, nous utilisons les opérateurs $M*$, $W*$, $A*$ et $F*$ pour désigner les différents ensembles maximaux considérés dans nos travaux d'évaluation, respectivement $(M1, M2, M3, M4)$ pour les missions, $(W1, W2, W3, W4)$ pour les environnements, $\{M*, W*\}$ pour les activités et l'ensemble des fautes injectées pour les fautes. Ainsi, $\overline{\sigma_p}(M*, W2, \emptyset)$ représente la moyenne arithmétique des quotients de photographies réussies pour trois exécutions de toutes les missions, réalisées dans le monde 2 et sans fautes injectées dans le modèle.

4.2 Résultats

Nous présentons dans cette section les résultats que nous avons obtenus pendant notre campagne d'expérimentation. Nous donnons tout d'abord des résultats d'expériences réalisées sans injection de fautes, avant de présenter les mesures réalisées pendant notre campagne d'injection de fautes.

Toutes les expériences ont été réalisées sur des plates-formes i386 Pentium 4 à 3,2 GHz, disposant de 1 Go de mémoire vive, et utilisant le système d'exploitation Linux. Des scripts d'automatisation sont utilisés pour lancer successivement différentes expériences, sauvegarder les journaux de chaque exécution, et en tirer les mesures relatives à un ensemble d'expériences.

4.2.1 Comportement sans injection de fautes

Les premières expériences que nous présentons concernent des modèles sans fautes injectées. Ces résultats nous sont utiles pour étudier le coût de l'utilisation du composant FTplan dans le système cible, et comme échantillon de comparaison pour les expériences en présence de fautes.

4.2.1.1 Comportement nominal

Dans cette section, nous considérons trois systèmes différents, utilisant des planificateurs distincts basés sur les deux modèles diversifiés présentés dans la Section 4.1.2.2 :

- le système *Robot1* comprend un planificateur non-tolérant aux fautes équipé de Modèle1,
- le système *Robot2* comprend un planificateur non-tolérant aux fautes équipé de Modèle2,
- le système *Robot1/2* comprend un planificateur tolérant aux fautes avec rétablissement par planification successive, exécutant Modèle1 comme première alternative et Modèle2 comme seconde alternative.

Les Figures 4.5 et 4.6 présentent le comportement nominal des trois systèmes étudiés face aux activités présentées dans la Section 4.1.2. Les mesures présentées sont les moyennes présentées dans la Section 4.1.4, réalisées sur trois exécutions équivalentes de chaque activité.

La Figure 4.5 présente les mesures caractérisant le comportement du système durant les expériences :

- les moyennes du nombre de replanifications par exécution $\overline{R}(\{M_i, W_j\}, \emptyset)$,
- les moyennes d'échec de missions $\overline{\mu}(\{M_i, W_j\}, \emptyset)$,
- les moyennes de défaillance du planificateur $\overline{\kappa}(\{M_i, W_j\}, \emptyset)$,
- les moyennes des quotients d'objectifs échoués : $\overline{\phi_h}(\{M_i, W_j\}, \emptyset)$ pour les retours à la position initiale, $\overline{\phi_c}(\{M_i, W_j\}, \emptyset)$ pour les communications, et $\overline{\phi_p}(\{M_i, W_j\}, \emptyset)$ pour les photographies (avec $(i, j) \in (1, 2, 3, 4)^2$).

La Figure 4.6 présente les mesures caractérisant les performances du système durant les expériences :

- les moyennes de la distance parcourue $\overline{D}(\{M_i, W_j\}, \emptyset)$,
- les moyennes de temps d'activité physique du système $\overline{T}(\{M_i, W_j\}, \emptyset)$ (avec $(i, j) \in (1, 2, 3, 4)^2$).

Ces expériences sans faute injectée servent trois objectifs principaux : (a) elles donnent un échantillon de comparaison pour les expériences d'injection de fautes présentées dans la Section 4.2.2, (b) elles permettent de tester les deux modèles développés, et de comparer leur efficacité, (c) elles permettent d'étudier le coût de l'utilisation de FTplan sur le comportement et les performances du système.

On observe immédiatement qu'un système disposant d'un modèle sans faute injectée a très peu d'échecs pour toutes les missions dans les mondes W1, W2 et W3.

En considérant que la moyenne de missions réussies $\overline{\nu}$ est égale à 100% - $\overline{\mu}$, et que les moyennes de quotients d'objectifs réussis $\overline{\phi_i}$ sont égaux à 100% - $\overline{\sigma_i}$, ces résultats peuvent se résumer de la façon suivante :

- $\overline{\nu}(\{M*, (W1, W2, W3)\}, \emptyset)$ = 100% pour Robot1 et Robot1/2,
- $\overline{\nu}(\{M*, (W1, W2, W3)\}, \emptyset)$ = 86% pour Robot2.

Dans le cas de Robot2, on obtient des moyennes de quotients d'objectifs réussis de :

- $\overline{\sigma_h}(\{M*, (W1, W2, W3)\}, \emptyset)$ = 96%,
- $\overline{\sigma_c}(\{M*, (W1, W2, W3)\}, \emptyset)$ = 98%,
- $\overline{\sigma_p}(\{M*, (W1, W2, W3)\}, \emptyset)$ = 86%.

Cependant, dans le cas du monde W4, les résultats sont moins bons et plus aléatoires, en adéquation avec ce qu'on pouvait attendre d'un environnement contenant des obstacles critiques pour le système (voir Section 4.1.2). En effet, les moyennes de mission réussie sont de

- $\overline{\nu}(\{M*, W4\}, \emptyset)$ = 33% pour Robot1,

- $\overline{\nu}(\{M*, W4\}, \emptyset)$ = 42% pour Robot2
- $\overline{\nu}(\{M*, W4\}, \emptyset)$ = 25% pour Robot1/2.

On peut remarquer que les résultats obtenus dans cet environnement dépendent davantage de conditions circonstancielles que de l'adéquation du modèle. Dans cet environnement, il est en effet possible que le robot se trouve confronté à des situations adverses face auxquelles aucun modèle ne serait adéquat.

Les résultats de ces expériences dans les mondes W1, W2 et W3 nous donnent donc une certaine confiance dans les deux modèles que nous utilisons. Deux remarques peuvent être encore faites sur nos deux modèles.

En premier lieu, les moins bonnes performances de Robot2 peuvent être expliquées par une période de test plus courte pour Modèle2 que pour Modèle1 (donc la présence possible de fautes involontaires) et la diversification forcée du modèle. En effet, une analyse détaillée des traces d'exécution nous permet de conclure que l'échec des retours de l'activité M4W1 est causé par la sous-estimation d'une distance dans le modèle, tandis que les échecs des missions M3W1 et M3W3 sont causés par des problèmes de performance découlant directement de l'utilisation d'une position symbolique plutôt que numérique. En particulier, l'implantation de la position symbolique nécessite un algorithme peu adapté au formalisme IxTeT, qui amène une surcharge de contraintes causant un impact négatif sur le temps de planification.

En deuxième lieu, ces expériences montrent l'importance du modèle utilisé sur le comportement du système : en particulier dans les cas où le système est confronté a des situations difficiles, voire impossibles (monde W4), les Figures 4.5 et 4.6 montrent que le comportement du système varie significativement selon le modèle employé.

Pour les mondes W1, W2 et W3, Robot1/2 a une moyenne de mission réussie $\overline{\nu}(\{M*, (W1, W2, W3)\}, \emptyset)$ de 100 %, et des performances en temps et en distance similaires à Robot1. Cependant, on peut voir d'après le nombre moyen de replanifications $\overline{R}(-, \emptyset)$ que très peu de basculements de modèle ont été effectués (en fait, seulement deux basculements en 36 expériences, tous deux dans l'activité M4W3, et causés par l'échec d'une action de déplacement). Or, un basculement de modèle prend un temps significatif (de une à deux secondes suivant le modèle considéré), et peut donc avoir un impact important sur le comportement du système. Néanmoins, ces expériences montrent que les autres activités de FTplan, en particulier l'interception et le traitement des messages, ont un impact négatif négligeable dans nos expériences sur ce comportement.

4.2.1.2 Coût de l'utilisation de FTplan

Afin de mettre plus en évidence l'impact des *basculements de modèle*, nous avons effectué une série d'expériences supplémentaires, en supprimant la fonctionnalité d'optimisation par réparation de plan du planificateur IxTeT. Ainsi, toute action qui échoue, ou qui prend plus de temps que prévu, cause immédiatement une replanification complète, sans possibilité de réparation du plan.

Deux systèmes ont été confrontés :

- le système *Robot1** comprend un planificateur non-tolérant aux fautes et sans réparation de plan, équipé de Modèle1,

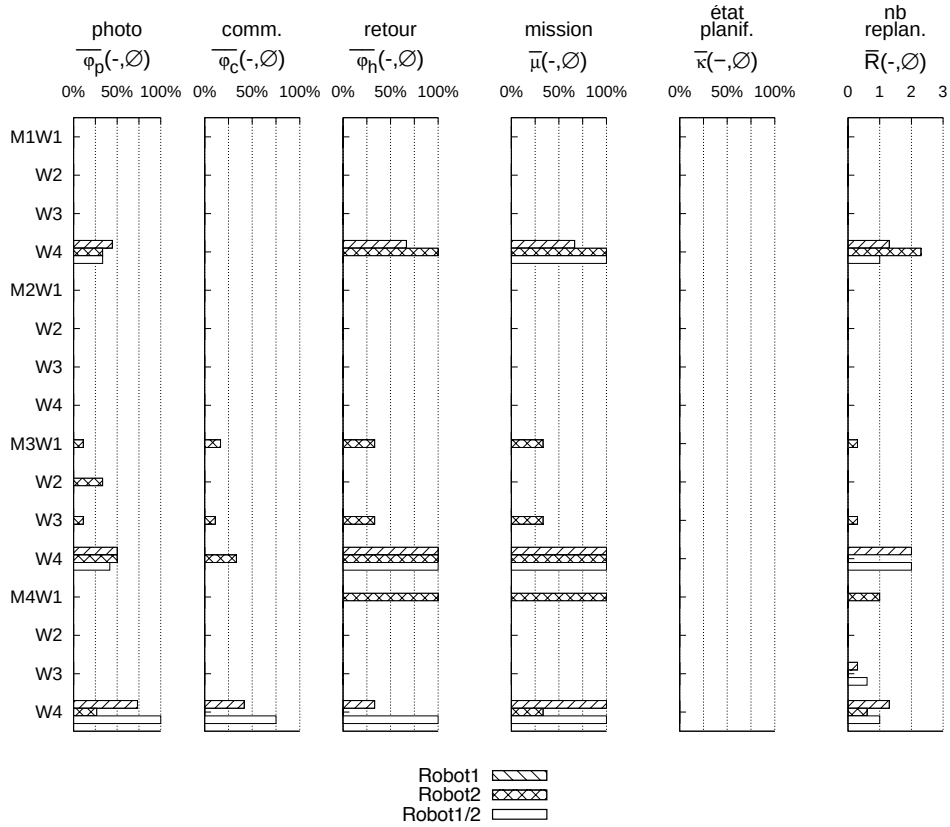

Figure 4.5: Comportement de Robot1, Robot2 et Robot1/2 avec des modèles sans faute injectée

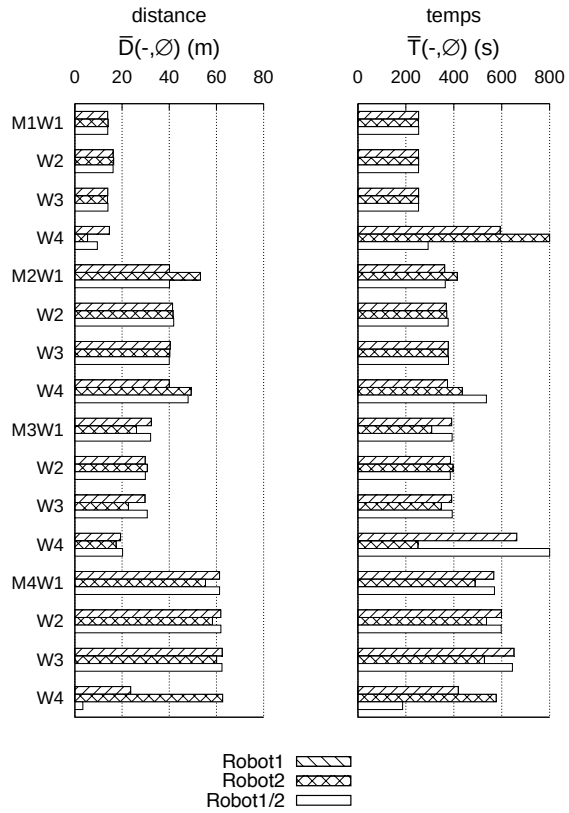

Figure 4.6: Performances de Robot1, Robot2 et Robot1/2 avec des modèles sans faute injectée

- le système *Robot1/1** comprend un planificateur tolérant aux fautes avec rétablissement par planification successive et sans réparation de plan, exécutant Modèle1 comme première et seconde alternative.

Le fait que Robot1/1* utilise exclusivement Modèle1 permet une comparaison directe des résultats avec ceux du système étalon Robot1* : les différences entre les deux systèmes ne peuvent ainsi être attribués qu'à l'impact de FTplan.

Les résultats de ces expériences sont présentés dans les Figures 4.7 et 4.8.

Ils montrent qu'il y a beaucoup plus de replanifications (et donc de basculements de modèle) que dans les expériences précédentes. En réalisant une moyenne sur les différentes activités, ces résultats conduisent à un $\overline{R}(\{M*, W*\}, \emptyset)$ de 8,3 pour Robot1* contre 0,3 pour Robot1, et de 8,9 pour Robot1/1* contre 0,4 pour Robot1/2. L'activité M1W2 apparaît comme une singularité pour Robot1/1*, car après quelques minutes d'exécution le planificateur IxTeT échoue à trouver un plan lors d'une replanification. La même expérience exécutée avec deux modèles diversifiés (Robot1/2*) s'accomplit en réussissant tous les objectifs, ce qui nous pousse à croire que cette singularité n'est pas causée par le composant FTplan (Robot1/2*, qui utilise également FTplan, manifeste un comportement correct), ni par Modèle1 (puisque Robot1* manifeste un comportement correct), mais plutôt par une faute non-identifiée présente dans la version modifiée d'IxTeT.

Mise à part cette singularité, Robot1/1* rate plus d'objectifs que le système étalon Robot1* seulement dans le monde W4 et dans l'activité complexe M4W3. Sans considérer le monde W4 et la singularité M1W2, la moyenne des temps d'activité physique du système $\overline{R}(\{(M*, (W1, W2, W3))\backslash M1W2\}, \emptyset)$ est de 381 secondes pour Robot1*, et de 431 secondes pour Robot1/1* (soit un surcoût de 13% en défaveur de Robot1/2*). En incluant le monde W4, la moyenne $\overline{R}(\{(M*, W*)\backslash M1W2\}, \emptyset)$ est de 456 secondes pour Robot1*, et 535 secondes Robot1/1* (soit un sur-coût de 17% en défaveur de Robot1/1*).

Nous considérons ces résultats satisfaisants, particulièrement en regard du nombre important de replanifications effectuées, bien supérieur à celui du système complet qui utilise la réparation de plan.

4.2.2 Comportement en présence de fautes injectées

Dans le but d'évaluer l'efficacité des mécanismes de tolérance aux fautes proposés, nous avons réalisé des expériences avec injection de fautes sur les systèmes Robot1 et Robot1/2 (présentés dans la Section 4.2.1.1). Nous avons injecté 38 fautes dans notre modèle principal (Modèle1), totalisant plus de 3500 expériences, soit environ 1200 heures de simulation. Nous présentons ici les résultats de ces expériences : nous précisons tout d'abord quelles fautes ont été injectées et considérées dans notre compagne d'expérimentation, avant de faire plusieurs remarques à partir de résultats particuliers. Nous présentons finalement l'analyse de résultats globaux portant sur l'ensemble des expériences.

4.2.2.1 Fautes injectées

Les 38 fautes injectées ont été sélectionnées aléatoirement dans l'ensemble de fautes présenté dans la Section 4.1.3, en suivant quatre consignes réalisant une

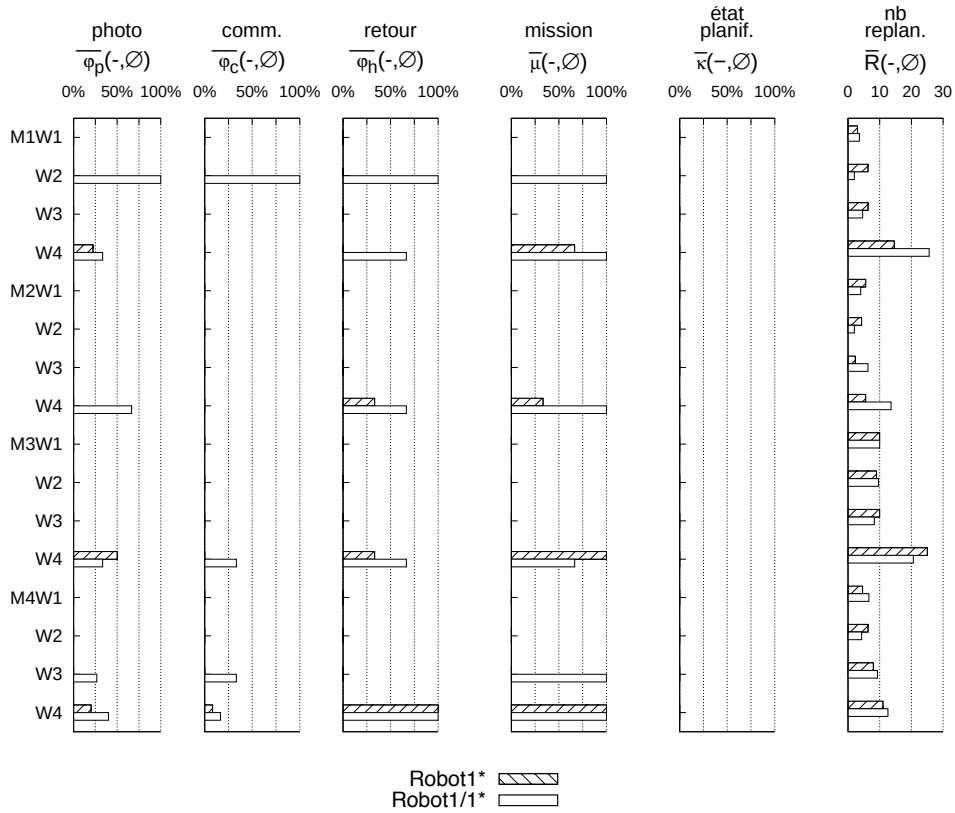

Figure 4.7: Comportement de Robot1* et Robot1/1*, sans réparation de plan et avec Modèle1 sans faute injectée

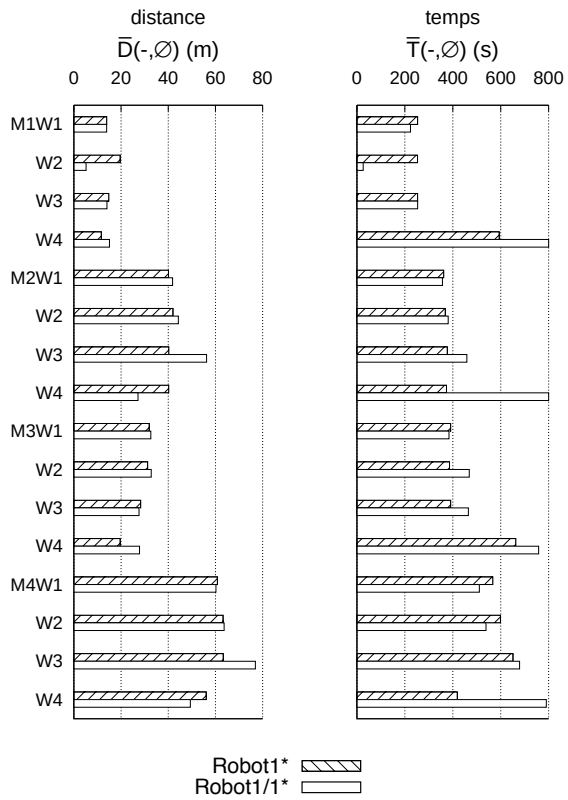

Figure 4.8: Performances de Robot1* et Robot1/1*, sans réparation de plan et avec Modèle1 sans faute injectée

optimisation partielle des mutants choisis, afin d'avoir un échantillon statistique plus significatif :

1. Le tirage aléatoire est réalisé de sorte que les fautes sélectionnées *couvrent tous les types de mutation* présentés en Section 4.1.3,
2. Les mutants *systématiquement incapables de trouver un plan* pour les quatre missions étudiées (par exemple si deux contraintes trivialement contradictoires sont présentes) ne sont pas retenus.
3. Les mutants *équivalents au modèle sans faute* (par exemple dont la mutation porte sur une ligne de commentaires) sont également écartés.
4. Nous ne retenons pas non plus les mutations *qui n'ont pas d'influence sur le comportement du système* dans les activités considérées (par exemple une définition des positions possible du robot en abscisse de [-100,+oo] à la place de [-oo,+oo], alors que notre système n'évolue que dans l'intervalle [-20,+20]).

En pratique, les conditions (2), (3) et (4) sont appliquées par une analyse manuelle rapide sur le modèle muté. Cette analyse est loin de garantir l'élimination de tous les mutations équivalentes, sans influence sur le comportement du système dans nos activités, ou échouant systématiquement.

En effet, sur les 38 mutants sélectionnées, 10 mutants se sont avérés incapables de trouver un plan pour les quatre missions de notre activité, et ont donc été retirés de nos résultats. Il est cependant positif de noter que, dans ces cas, le système équipé de FTplan bascule sur son deuxième modèle et obtient des résultats proches de Modèle2 : un taux de réussite d'objectifs presque parfait dans les mondes 1, 2 et 3.

De plus, 5 des 28 mutations restantes présentent des mesures très proches de celles du modèle non-muté originel, et pourraient être des mutations équivalentes. Nous n'avons cependant pas trouvé de critères satisfaisants permettant de déterminer l'équivalence entre modèles à partir des mesures que nous avons établies.

Pour illustrer la difficulté de ce problème, les Figures 4.9 et 4.10 présentent les résultats de 3 jeux des 48 expériences, effectués sur Robot1 en utilisant le même modèle : Modèle1 sans faute injectée.

Les résultats des différents jeux apparaissent comme très proches, mais il est cependant difficile d'établir une équivalence objective. Pour résoudre ce problème, nous pensons qu'un nombre plus important d'exécutions de chaque activité serait nécessaire, afin d'obtenir une caractérisation statistique plus précise. Nous nous sommes limités à trois exécutions afin de diminuer le nombre d'expériences à réaliser pour chaque mutation.

Dans l'incapacité de déterminer une réelle équivalence entre les cinq mutants suspects et le modèle original non-muté, nous avons choisi de conserver ces cinq mutants dans nos résultats généraux, bien que leur présence ait une influence négative sur l'efficacité observée de nos mécanismes de tolérance aux fautes.

Les 28 mutations que nous avons conservées représentent notre ensemble de fautes injectées $F*$, et se répartissent comme suit :

- 9 substitutions de valeurs numériques,
- 6 substitutions de variables,
- 3 substitutions de valeurs d'attribut non-numériques,
- 4 substitutions d'opérateurs du langage,

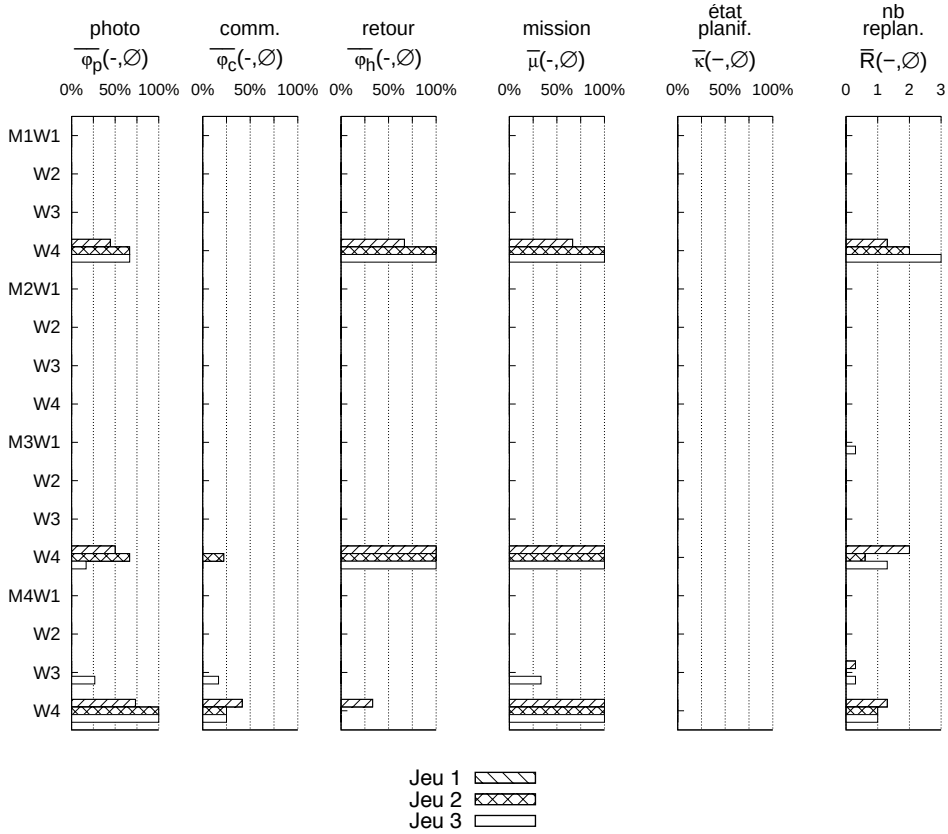

Figure 4.9: Comportement de Robot1 pour 3 jeux de Modèle1 sans faute injectée

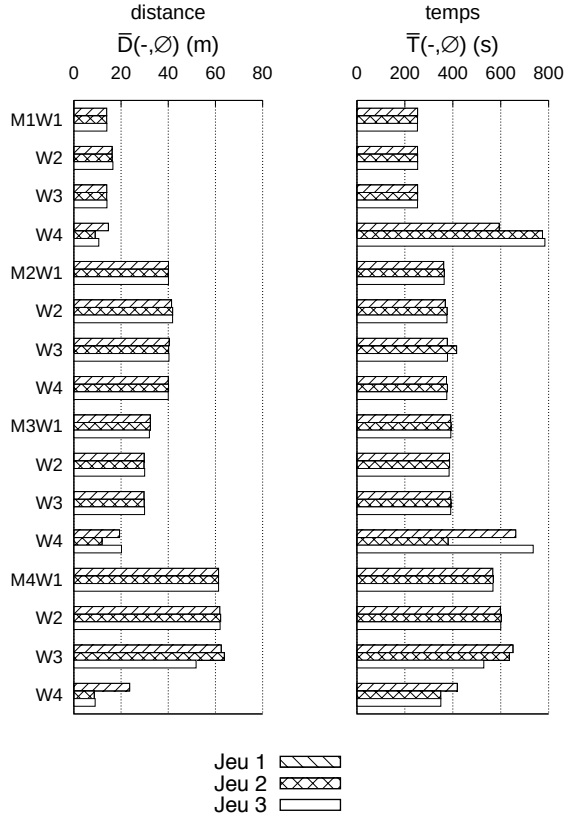

Figure 4.10: Performances de Robot1 pour 3 jeux du Modèle1 sans faute injectée

- 6 suppressions de contraintes.

4.2.2.2 Exemple d'expérimentations

Dans cette section, nous nous attardons sur trois expériences qui donnent un exemple concret des résultats de nos expérimentations, et permettent plusieurs remarques intéressantes sur le fonctionnement de FTplan et l'évaluation des systèmes autonomes de manière plus générale.

Mutation 1-39

La première mutation que nous considérons, identifiée dans nos expériences par l'indice 1-39, consiste en une restriction trop importante du domaine d'une variable numérique. En particulier, la position du robot permise par le modèle pendant une prise d'image est contrainte par le domaine [-4,+oo] au lieu de [-oo, +oo]. Les résultats de cette mutation sont présentées dans la Figure 4.11.

Pour Robot1, la moyenne de missions échouées $\overline{\mu}(\{(M2, M4), W*\}, 1-39)$ de 100% correspond en fait à une impossibilité pour le mécanisme de raisonnement de conjuguer la contrainte mutée de position du robot pendant une prise d'image, et la présence de cibles à photographier d'abscisse inférieure à -4 (voir les définitions des missions M2 et M4 sur la Figure 4.2, page 81). Incidemment, cette faute injectée a révélé la présence d'une faute réelle dans le mécanisme d'inférence, qui amène le planificateur à s'arrêter brusquement plutôt qu'envoyer un message d'erreur face à cette contradiction.

Cet exemple montre l'importance du choix de l'activité lors de la validation du planificateur : il est important de générer des missions de test aussi diversifiées et nombreuses que possible car de nombreuses fautes ne peuvent pas être révélées par une mission unique. Comme il est impossible de tester entièrement le contexte d'exécution (l'espace des activités possibles est en pratique infini), la présence de mécanismes de tolérance dans un système autonome paraît fondamentale afin de couvrir les fautes résiduelles qui n'ont pas pu être révélées lors de l'évaluation du planificateur. Au moins trois mutations similaires (c'est-à-dire pour lesquelles le modèle muté cause une défaillance systématique du plan pour *certaines* missions) ont été identifiées.

Mutation 1-589

Contrairement à l'exemple précédent, la mutation d'indice 1-589 fournit un plan pour chacun des environnements proposés. Qui plus est, les plans sont corrects lorsque le robot est en situation nominale, car la mutation porte sur l'action de réinitialisation des déplacements du système qui n'est pas nécessaire au planificateur dans le problème initial posé. Cependant, lorsque le déplacement du système échoue en raison d'une situation adverse, le planificateur se retrouve incapable d'établir un plan permettant de réinitialiser les modules de déplacement du système, ce qui s'avère être une défaillance critique pour la mission. Les résultats des expériences pour cette mutation sont présentés dans la Figure 4.12

Robot1/2 tolère très bien la faute injectée, la mission M3 mise à part. Dans ce dernier cas, la moyenne des quotients de photographies échouées $\overline{\phi_p}(\{M3, W*\}, 1-589)$ est de 100%, égale à celle de Robot1. L'analyse des traces d'exécution nous permet de constater que les échecs de Robot1/2 sont causés par

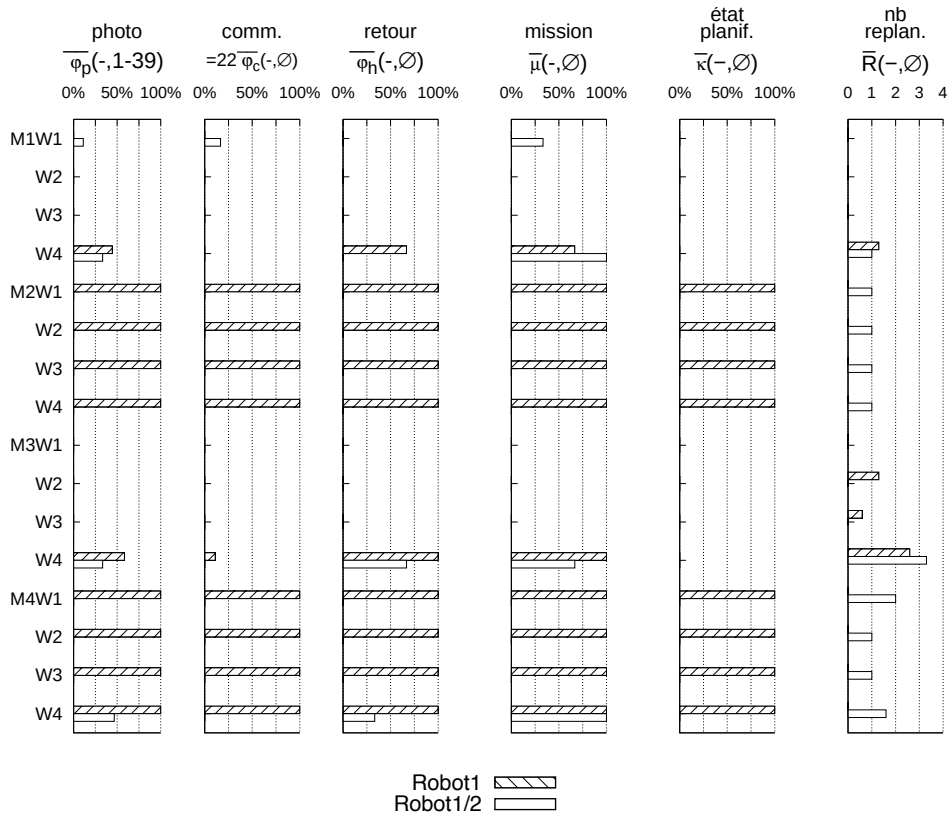

Figure 4.11: Comportement de Robot1 et Robot1/2 pour la mutation 1-39

l'insuccès de la première communication, dont l'échéance est dépassée en raison du basculement de modèle et de la replanification causés par l'activation de la faute injectée. En effet, les communications sont prioritaires par rapport aux photographies dans notre activité, et le planificateur abandonne donc les objectifs de photographies pour privilégier celui de la première communication, devenue irréalisable.

Deux remarques générales peuvent être faites sur cette mutation. Tout d'abord, l'importance de tester le système par rapport à des contextes d'exécution variés est encore une fois démontrée par la présence de missions et d'environnements où tous les objectifs sont atteints, et d'autres où les résultats sont au contraire désastreux. Cette mutation montre également qu'il est nécessaire de réaliser des tests en utilisant un système intégré, plutôt que de valider uniquement le mécanisme décisionnel ciblé. En effet, les plans initiaux produits par le planificateur sont sans erreurs, puisque la mutation porte sur une action inutile pour une planification partant de l'état initial du système. Le modèle contient pourtant une faute grave, activée pendant certaines exécutions, qui conduit à une défaillance critique du système.

Mutation 1-583

Cette troisième mutation est un exemple de cas où Robot1 accomplit en

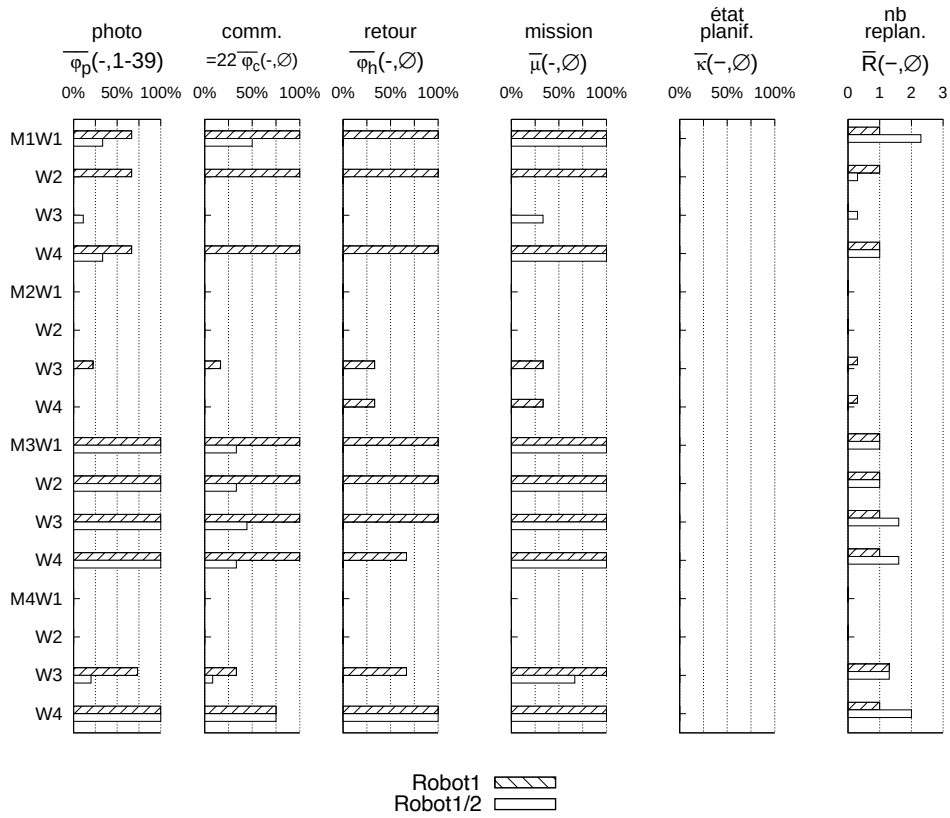

Figure 4.12: Comportement de Robot1 et Robot1/2 pour la mutation 1-589

moyenne plus d'objectifs que Robot1/2 : pour cette mutation, il vaut mieux replanifier avec Modèle1 muté plutôt que de basculer sur Modèle2. La mutation injectée cause une mauvaise optimisation de la durée des déplacements, prévus plus courts dans le plan. Ce problème occasionne des erreurs d'exécution suivies de replanifications, causant ainsi par propagation l'échec de certains objectifs. Les résultats de cette mutation sont présentés dans la Figure 4.13. En particulier, la moyenne de quotients de photographies réussies $\overline{\sigma_p}(\{M*,(W1,W2,W3)\},1-583)$ est de 84% pour Robot1 et de 59% pour Robot1/2, la moyenne de quotients de communication réussies $\overline{\sigma_c}(\{M*,(W1,W2,W3)\},1-583)$ est de 96% pour Robot1 et de 85% pour Robot1/2, et la moyenne de missions réussies $\overline{\nu}(\{M*,(W1,W2,W3)\},1-583)$ est de 81% pour Robot1 et de 50% pour Robot1/2.

Les échecs plus nombreux de Robot1/2 ne sont pas directement liés à la faute injectée. Dans la mission M3, les échecs de prise de photos sont dus à une moins grande efficacité de Modèle2, qui a une vision pessimiste du temps requis pour réaliser certains déplacements, et doit abandonner deux objectifs photos pour avoir un plan réalisable dans les temps. Pour la mission M4, des problèmes de manque d'optimisation de Modèle2 provoquent des planifications plus longues, ce qui entraîne le dépassement d'une échéance de communications pendant la planification. De la même façon que pour la mutation précédente, cet échec entraîne ensuite l'échec de tous les objectifs de photographies du système.

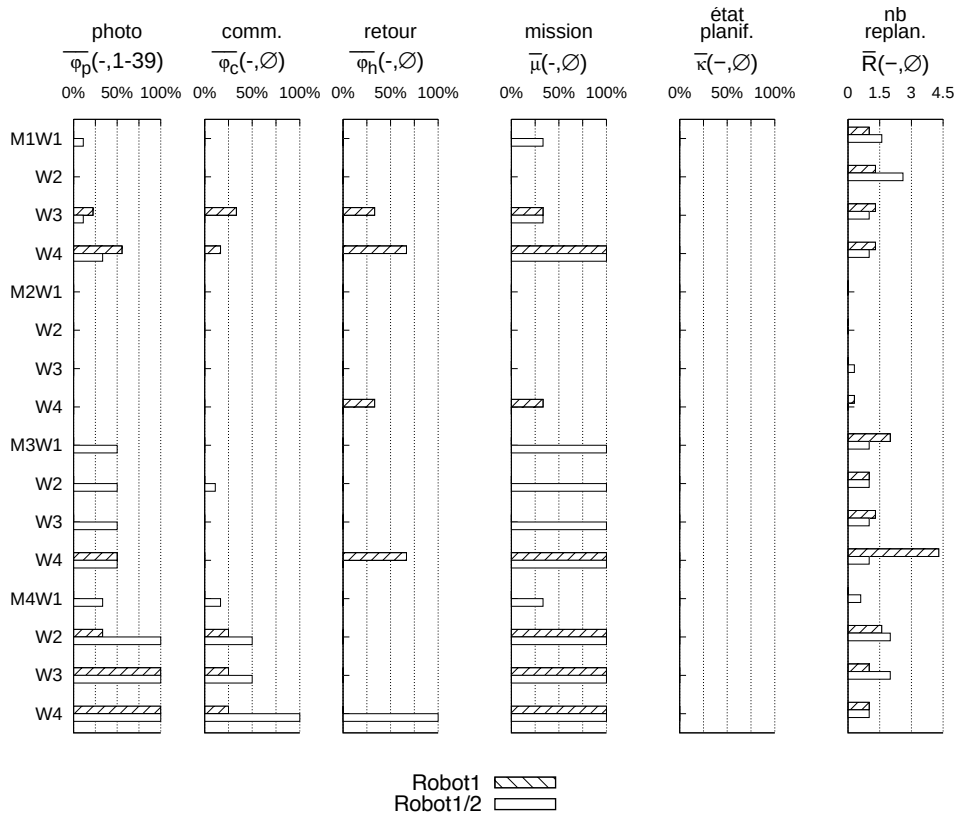

Figure 4.13: Comportement de Robot1 et Robot1/2 pour la mutation 1-583

Dans les 28 mutations que nous avons étudiées, cette mutation est la seule pour laquelle Robot1/2 présente de moins bons résultats que le système étalon Robot1. Cependant, 8 mutants (dont 5 mutants potentiellement équivalents à Modèle1 non-muté) ont été identifiés, pour lesquels le nombres d'objectifs atteints par les deux systèmes sont similaires : présentant un écart de moins de 5%.

4.2.2.3 Résultats généraux

Les résultats généraux de nos expérimentations sont présentés dans la Figure 4.14. Compte tenu de l'adversité excessive du monde W4 (voir Section 4.2.1.1), nous avons décidé de présenter également des résultats pour lesquels les expériences dans le monde W4 ont été exclues.

Ces résultats montrent sur un nombre conséquent d'expériences que la diversification de modèle améliore de façon significative la tolérance aux fautes d'un système autonome en présence de fautes. En particulier, monde W4 mis à part, l'utilisation de FTplan apporte une diminution absolue de 17% à la moyenne des quotients de photographies réussies $\overline{\sigma_p}(\{W*,(M1,M3,M3)\},F*)$ (soit une amélioration relative de 62%), de 22% à la moyenne des quotients de communications réussies $\overline{\sigma_c}(\{W*,(M1,M3,M3)\},F*)$ (soit une amélioration de 70%), de 21% % à la moyenne des quotients de retours réussis $\overline{\sigma_h}(\{W*,(M1,M3,M3)\},F*)$

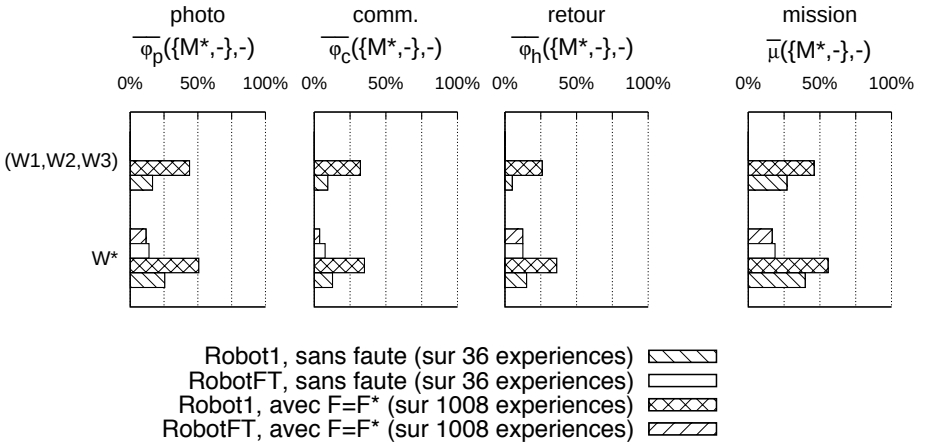

Figure 4.14: Résultats généraux de la campagne d'expérience avec et sans le monde W4

(soit une amélioration de 80%), et de 19% à la moyenne des missions réussies $\overline{\nu}(\{W*, M*\}, F*)$ (soit une amélioration de 41%). En considérant le monde W4, on retrouve des résultats similaires, avec une diminution absolue de 25% de la moyenne des quotients de photographies réussies $\overline{\sigma_p}(\{W*, M*\}, F*)$ (soit une amélioration relative de 50%), de 12% à la moyenne des quotients de communications réussies $\overline{\sigma_c}(\{W*, M*\}, F*)$ (soit une amélioration de 64%), de 19% à la moyenne des quotients de retours réussis $\overline{\sigma_h}(\{W*, M*\}, F*)$ (soit une amélioration de 58%), et de 16% à la moyenne des missions réussies $\overline{\nu}(\{W*, M*\}, F*)$ (soit une amélioration de 29%).

En considérant le monde W4 et sans injection de fautes, on peut observer que Robot1 réussit légèrement plus d'objectifs et de missions que Robot1/2. Néanmoins, le nombre limité d'expériences effectuées (seulement une quarantaine d'expériences sans injection de fautes, contre plus de mille expériences avec injection de fautes) n'est pas suffisant pour déterminer si cette différence est causée par un meilleur comportement général de Robot1, dû par exemple à la baisse de performances minime induite par l'utilisation de FTplan dans Robot1/2, ou par des situations circonstancielles légèrement plus favorables à Robot1 dans le monde W4.

On peut également noter qu'un système utilisant FTplan en présence de fautes injectées reste *moins* efficace en terme de quotients d'objectifs échoués qu'un système utilisant un modèle unique sans fautes injectées. Cet écart de performance peut s'expliquer principalement par deux raisons :

- Dans la version actuelle de FTplan, un plan erroné n'est détecté qu'une fois que son exécution a causé un échec, ce dernier ayant pu rendre un des objectifs inatteignables malgré le rétablissement du système (par exemple, une échéance de communication a été dépassée).
- Ainsi que mentionné dans la Section 3.2.2.2, l'exécutif temporel utilisé attend la fin de la mission pour avertir FTplan que son activité est terminée, bien qu'il puisse avoir déjà fini sa dernière action depuis plusieurs mi-

nutes. Une telle situation diminue l'efficacité de la vérification des objectifs en cours d'exécution, puisque le système n'a alors plus le temps de remplir les objectifs qui n'ont pas été exécutés par le plan.

D'après les traces d'exécution, la mutation 1-592 illustre bien ce cas. En effet, cette mutation donne lieu à la spécification dans Modèle1 d'une mauvaise position de la platine lors de la prise de photographie. Cette faute ne provoque pas d'erreurs détectables par Robot1, mais cause l'échec systématique des objectifs de photographies puisque les caméras du robot ne pointent pas sur la cible à photographier ; on a ainsi $\overline{\sigma_p}(W*, (M1, M3, M3), 1-592)$ égale à 0% pour Robot1. Pour Robot1/2, la vérification des objectifs en cours d'exécution permet de tolérer cette faute lorsqu'une situation adverse provoque un basculement de modèle, en donnant au planificateur alternatif la liste des objectifs qui n'ont pas encore été réalisés. Cependant, comme ce basculement dépend de la présence d'une situation externe au système, et n'est pas activé systématiquement à la fin de l'activité du système, $\overline{\sigma_p}(\{W*, (M1, M3, M3)\}, 1-592)$ n'est encore égale qu'à 50%.

4.3 Conclusion

Nous avons présenté dans ce chapitre une évaluation de mécanismes de tolérance aux fautes basée sur l'injection de fautes et la simulation du robot physique. La réalisation d'une campagne d'évaluation est un processus long et laborieux, particulièrement dans le cas des systèmes autonomes, ou la variabilité des contextes d'exécution et l'indéterminisme du comportement du système sont exarcerbés. Notre campagne d'évaluation a d'ailleurs mis en avant l'importance d'utiliser des contextes d'exécution aussi variés et diversifiés que possible, ainsi que la nécessité d'évaluer le système autonome dans son ensemble.

Les expériences que nous avons réalisées montrent de façon objective que, dans l'ensemble d'activités que nous avons considéré, les mécanismes de tolérance aux fautes proposés améliorent significativement le nombre d'objectifs atteints et de missions réussies par le système autonome en présence de fautes. Cependant, malgré le nombre important d'expériences réalisées, nous n'avons couvert qu'une faible portion de l'ensemble de fautes que nous avons généré (moins de 4%). De plus, nos contextes d'exécution ne représentent qu'une part infime des contextes possibles. De nombreuses autres expérimentations sont donc nécessaires, impliquant des ressources plus importantes que celles dont nous disposons. Nos résultats ont aussi montré que le composant FTplan actuellement développé ne peut pas tolérer toutes les fautes, car ses mécanismes de détection ne peuvent détecter un plan erroné que *pendant* son exécution. L'intégration de la détection par analyse de plan, capable de détecter un plan erroné *avant* son exécution, est donc fondamentale pour encore améliorer l'efficacité du composant proposé.

Ce qu'il faut retenir

1. Notre évaluation est basée sur l'injection de fautes et la simulation du robot physique, qui permet d'effectuer des expérimentations plus facilement et sans risques.

2. Malgré plus de 3500 expériences réalisées, de nombreuses autres expériences seraient nécessaires pour avoir une caractérisation statistiquement représentative du comportement des systèmes autonomes cibles.
3. Dans l'ensemble d'activités considérées, les expériences réalisées montrent que, en l'absence de fautes injectées, le composant FTplan n'a pas un impact négatif important sur les performances du système.
4. Dans l'ensemble d'activités considérées, le composant FTplan apporte une amélioration significative (plus de 33%) au nombre d'objectifs accomplis par le système en présence de fautes.
5. Pour approcher des résultats équivalents à un système sans faute, un analyseur de plan, capable de détecter un plan erroné *avant* son exécution, est indispensable au composant FTplan.

Chapitre 5

Conclusions et perspectives

A l'avenir, les systèmes autonomes vont probablement se multiplier et se démocratiser, comme compagnons de jeu, véhicules de surveillance, d'exploration ou de transport, assistants personnels pour les personnes âgées ou handicapées... Leur utilisation comme gadgets technologiques ou, à l'opposé, comme réponses à des problèmes complexes et critiques de notre société dépendra des capacités décisionnelles qu'ils comporteront, et donc directement de la confiance que ces capacités suscitent. Pour nous, l'utilisation de systèmes autonomes complexes passe ainsi inévitablement par le développement de techniques de sûreté de fonctionnement appliquées aux mécanismes décisionnels.

Nous concluons ici ce mémoire dédié à la tolérance aux fautes dans les systèmes autonomes en rappelant tout d'abord nos résultats principaux, puis en présentant plusieurs pistes possibles dans le prolongement de nos travaux.

5.1 Démarche suivie

Nous avons cherché à étudier de quelles façons les techniques de la sûreté de fonctionnement informatique peuvent être appliquées aux systèmes autonomes. Notre recherche s'est trouvée motivée par l'impossibilité d'utiliser les mécanismes décisionnels de l'intelligence artificielle dans des systèmes critiques, tant que ces mécanismes ne sont pas capables de donner une confiance justifiée dans la sécurité-innocuité et la fiabilité de leur comportement. Cette limitation restreint en pratique l'application des systèmes décisionnels à des applications fonctionnelles plutôt que décisionnelles, comme la reconnaissance ou le suivi de visage.

Un premier état de l'art de la sûreté de fonctionnement dans les systèmes autonomes nous a convaincu de l'importance de cette problématique. Dans la pratique, le développement des systèmes autonomes s'appuie principalement sur la prévention des fautes, à travers des conceptions modulaires et l'utilisation d'environnements de programmation, et l'élimination des fautes, à travers le test et la validation. L'évaluation quantitative des systèmes autonomes et, en particulier, la prévision de leur comportement vis-à-vis des fautes sont encore quasiment absentes des pratiques de développement. En ce qui concerne la tolérance aux fautes, certains mécanismes de sûreté de fonctionnement classiques (tests d'assertions,

chien de garde temporel, mise en état sûr) sont appliqués avec succès au niveau fonctionnel des systèmes autonomes existants pour garantir un certain degré de sécurité-innocuité, mais aucun accent n'est mis sur les mécanismes décisionnels du système, pourtant centraux à son comportement. Les mécanismes de robustesse développés dans le domaine de la robotique peuvent, dans une certaine mesure, contribuer à la tolérance aux fautes d'un système autonome, mais sont loin d'être suffisants.

A partir de cette analyse, nous avons établi plusieurs remarques générales relatives aux types d'architectures employées dans les systèmes autonomes. Nous avons, entre autres, souligné l'intérêt d'un composant indépendant de sécurité, chargé de faire respecter des contraintes de sécurité-innocuité. Nous nous sommes ensuite particulièrement intéressés à la fonction de planification, actuellement incontournable dans les systèmes autonomes pour la résolution de missions complexes. Malgré son importance, très peu de travaux de tolérance aux fautes lui sont actuellement accordés. Nous avons ainsi décidé de nous focaliser sur la tolérance aux fautes dans les mécanismes de planification, et en particulier dans les connaissances (modèles et heuristiques) qu'ils emploient, un domaine encore totalement prospectif à notre connaissance.

Nous avons proposé quatre mécanismes de détection d'erreurs (détection par chien de garde temporel sur l'élaboration du plan, par analyse de plan, par vérification en ligne des objectifs, et détection d'un échec d'exécution), et deux mécanismes de rétablissement du système (par planification successive et par planification concurrente). Les mécanismes de rétablissement présentés utilisent plusieurs instances d'un même planificateur, et sont basés sur la diversification de leurs heuristiques, de leurs modèles, et éventuellement de leurs mécanismes d'inférence. Nous avons introduit un composant unique implanté à l'interface des planificateurs, dont le but est de mettre en place ces différents mécanismes. Nous avons développé un prototype utilisant trois mécanismes de détection et un mécanisme de rétablissement[1], qui a été implanté avec succès dans l'architecture de système autonome LAAS.

Finalement, nous avons mis en place un environnement d'évaluation du prototype développé, utilisant l'ensemble des composants logiciels d'un robot réel, et simulant l'environnement physique et les composants matériels du robot. Nous avons transposé la technique d'injection de fautes par mutation aux modèles déclaratifs utilisés dans la planification, afin d'injecter des fautes dans les modèles des mécanismes décisionnels utilisés par le robot. En comparant l'efficacité et les performances de deux systèmes, avec et sans nos mécanismes de tolérance aux fautes, nous avons montré que les mécanismes proposés améliorent significativement les accomplissements du système cible en présence de fautes, sans impact négatif important sur ses performances.

[1] Les mécanismes mis en place sont la détection par chien de garde temporel, la détection par vérification en ligne des objectifs, la détection d'un échec d'exécution, et le rétablissement par planification successive.

5.2 Leçons apprises

Les systèmes autonomes se distinguent des systèmes informatisés plus traditionnels par la présence de *mécanismes décisionnels* dérivés de l'intelligence artificielle. Ces mécanismes sont difficiles à valider et évaluer, d'une part, à cause de la vaste dimension de leurs contextes d'exécution et, d'autre part, parce qu'il est complexe, voire impossible, de dichotomiser leur comportement en correct ou incorrect de façon objective. Pour ces raisons, l'application de techniques d'élimination des fautes (comme le test) ou de tolérance aux fautes (comme le doublement par comparaison ou le masquage de fautes) est fortement complexifiée. La nécessité d'intégrer de telles techniques pour le développement des systèmes autonomes est cependant démontrée dans les études de prévision des fautes et d'élimination des fautes que nous avons identifiées [Carlson & Murphy 2003, Tomatis et al. 2003, Bernard et al. 2000].

Les mécanismes proposés dans ce mémoire, bien qu'améliorant objectivement le comportement de notre système cible en présence de fautes, ne répondent qu'à un aspect de la tolérance aux fautes dans les systèmes autonomes. Tout d'abord, ils se concentrent sur les fautes de développement dans les connaissances du planificateur, et ne couvrent pas toutes les fautes liées aux mécanismes décisionnels présentées dans ce manuscrit. En particulier, ils se concentrent sur l'aspect fiabilité du système (c'est-à-dire l'accomplissement de sa mission), plutôt que de s'intéresser à l'aspect sécurité. Ils doivent ainsi être utilisés de façon complémentaires à d'autres mécanismes, couvrant d'autres domaines de fautes. Il est aussi important de noter que le travail d'évaluation réalisé ne porte que sur un exemple d'application d'exploration spatiale, et n'a été exécuté qu'avec le seul planificateur IxTeT. Des tests sur d'autres problèmes de planification et utilisant d'autres planificateurs sont nécessaires pour s'assurer du bon fonctionnement des mécanismes proposés dans des cas aussi nombreux et variés que possible. Néanmoins, les travaux menés dans ce mémoire montrent objectivement que des mécanismes de tolérance aux fautes appliqués aux planificateurs améliorent sensiblement le comportement d'un système autonome en présence de fautes. Nous considérons donc souhaitable l'utilisation de tels mécanismes dans les systèmes autonomes critiques.

L'environnement d'évaluation des mécanismes de tolérance aux fautes s'est avéré apte à stresser l'autonomie du système, et à mesurer l'impact des fautes injectées sur le comportement du robot. Basée sur la simulation des composants matériels du robot et de son environnement, cette approche de simulation ne doit pas se substituer aux tests sur des plates-formes réelles, mais doit permettre, à travers une approche complémentaire, d'effectuer des tests intensifs et une évaluation intégrée de mécanismes décisionnels. Malgré des difficultés liées à la quantification du résultat des expériences, cet environnement d'évaluation s'est montré efficace pour la comparaison de systèmes différents. Hors du cadre de cette thèse, de nombreuses autres études peuvent s'appuyer sur son principe, notamment des comparaisons de robustesse ou de tolérance aux fautes entre différents modèles ou différentes heuristiques. Éventuellement, il pourrait également être utilisé pour réaliser des travaux de prévision de fautes, par exemple en étudiant la conséquence de fautes injectées de différentes façons sur le système, de manière contrôlée et sans danger. Notons enfin que le travail nécessaire pour mettre en

œuvre un environnement d'évaluation ne doit pas être sous-estimé, mais il est cependant nécessaire pour l'évaluation de la sûreté de fonctionnement du système. En particulier, le système doit être testé dans son intégrité, et face à des activités nombreuses et diversifiées.

5.3 Perspectives

Dans la continuation des travaux présentés, quatre aspects majeurs peuvent être poursuivis :

1. Ainsi que nous l'avons mentionné dans le manuscrit, des expériences supplémentaires doivent encore être réalisées afin d'obtenir des résultats plus représentatifs. Tout d'abord, davantage d'exécutions équivalentes de chaque activité doivent être effectuées pour permettre d'obtenir des résultats plus représentatifs statistiquement, et peut-être ainsi détecter et éliminer de nos résultats les mutants équivalents à un modèle original non-muté. De plus, un nombre plus important de fautes et de contextes d'exécution peuvent être considérés, afin d'accroître encore la confiance dans nos résultats. Pour rendre possible la réalisation d'un grand nombre d'expériences dans des délais raisonnables, notre environnement d'évaluation peut être porté sur une grille supportant de nombreuses plates-formes, par exemple la grille française GRID5000[2]. Enfin, des travaux supplémentaires pourraient être réalisés sur notre environnement d'évaluation, dans le but d'en faire un outil réutilisable pour l'évaluation de différents aspects liés aux planificateurs intégrés dans des systèmes autonomes : entre autres la robustesse et la tolérance aux fautes de leurs mécanismes d'inférence et de leurs connaissances.
2. Afin d'améliorer l'efficacité des mécanismes évalués dans ce mémoire, il est important de développer et d'intégrer un analyseur de plan comme moyen de détection supplémentaire associé au composant de tolérance aux fautes FTplan. En effet, cet analyseur de plan devrait permettre de détecter des erreurs dans un plan avant son exécution, en comparant le plan produit par le planificateur avec un jeu de contraintes dépendantes de l'application, et ainsi d'augmenter de façon significative la tolérance aux fautes du système cible. Pour couvrir les fautes de développement dans les connaissances du planificateur, les contraintes utilisées par l'analyseur de plan doivent être développées de façon diversifiée au modèle de planification, mais suivant les mêmes spécifications du système. Éventuellement, un ensemble de contraintes demandant un moindre coût de développement, mais ne permettant de tolérer que les fautes dans le mécanisme d'inférence du planificateur, peut être obtenu en réutilisant directement les contraintes du modèle de planification.
3. Dans nos travaux, nous n'avons évalué que la politique de rétablissement du système par planification successive. Il serait intéressant d'implanter la planification concurrente dans le composant FTplan, et de comparer son

[2]https ://www.grid5000.fr

efficacité et ses performances à celles de la planification successive. Nous pensons que l'utilisation de la planification concurrente est particulièrement justifiée pour des systèmes multi-processeurs, permettant l'exécution simultanée de plusieurs planificateurs avec un impact négatif mineur sur la durée de planification. Dans de tels cas, la planification concurrente doit montrer des performances temporelles supérieures à la planification successive en présence de fautes, puisque le temps perdu à effectuer des planifications qui échouent est masqué par l'exécution simultanée des planificateurs.

4. En complément des mécanismes de tolérance aux fautes proposés dans ce manuscrit, d'autres mécanismes peuvent être proposés et validés, afin d'améliorer la couverture de fautes. En particulier, le concept d'un composant indépendant de sécurité, chargé de faire respecter des consignes de sécurité-innocuité, semble prometteur. Bien que mis en pratique dans un cadre plus de robustesse que de tolérance aux fautes, un tel composant a été implanté dans l'architecture de système autonome LAAS [Py 2005]. Trois problèmes fondamentaux doivent cependant être résolus : comment définir un ensemble de règles de sécurité-innocuité à faire respecter ? quels moyens d'observation mettre en place pour réaliser le contrôle en ligne de ces règles ? et comment sélectionner les actions de mise en sécurité à entreprendre lorsqu'une infraction à ces règles risque de survenir ?

Annexe A

Exemples de modèles de planification

Nous présentons ici les deux modèles utilisés dans nos travaux d'évaluation, introduits dans la Section 4.1.2.2. Chaque modèle requiert deux éléments : d'une part, la description des actions qui peuvent être effectuées par le système et, d'autre part, la description du plan partiel initial, qui contient l'état initial du système, les objectifs à remplir, et les évènements non-contrôlables par le système (comme l'entrée ou la sortie d'une fenêtre de visibilité). Pour des raisons pratiques, ces deux éléments sont décomposés en plusieurs fichiers. Notamment, la description des attributs et des constantes du système, nécessaire à chacun des éléments, est regroupée dans un fichier commun.

Ainsi, la description des actions du système comprend : le fichier `Modele#.task` qui décrit ces actions, et le fichier `Modele#-relations` qui décrit les attributs et constantes du système. Dans le cas du Modèle, le fichier `Modele2-distances` est aussi nécessaire, dépendant de la mission demandée au système et décrivant les distances entre les points singuliers de cette mission.

La description du plan partiel initial comprend : le fichier `Modele#-init.task`, qui décrit l'état initial du système, et les éléments non-contrôlables, le fichier `Modele#-relations` qui décrit les attributs et constantes du système, et le fichier `Modele#-goals` qui décrit les objectifs demandés au système et dépendant de la mission.

Dans les deux modèles présentés, les fichiers dépendants de la mission donnés ici sont relatifs à la première mission présentée dans la Section 4.1.2 : trois photographies à prendre et deux communications à effectuer.

A.1 Modèle1

Nous présentons ici les différents fichiers composant le Modèle1.

Modele1-relations

```
constant BOOL = {TRUE, FALSE};
constant OBJECTS = {OBJ1, OBJ2, OBJ3, OBJ4, OBJ5,
                    OBJ6, OBJ7, OBJ8, OBJ9, OBJ10};
```

```
constant STATUS = {NONE,DONE};
constant PANTILT_POSITIONS = {STRAIGHT, AT_MY_FEET,
                              OTHER_POSITION};
constant VISIBILITY_WINDOWS = {W1,W2,W3,W4};
constant IN_OUT = {IN,OUT};


attribute VISIBILITY_WINDOW(?w){
        ?w in VISIBILITY_WINDOWS;
?value in IN_OUT;
}

attribute COMMUNICATION(?w){
        ?w in VISIBILITY_WINDOWS;
?value in STATUS | {COMMUNICATION_IDLE};
}

attribute PTU_DRIVER_INITIALIZED(){
?value in BOOL | {PTU_DRIVER_IDLE};
}

attribute MVT_GENERATION_INITIALIZED(){
?value in BOOL | {MVT_GENERATION_IDLE};
}

attribute AT_ROBOT_X() {
?value in [-oo,+oo];
}

attribute AT_ROBOT_Y() {
?value in [-oo,+oo];
}

attribute ROBOT_STATUS(){
?value in {MOVING,STILL};
}

attribute PICTURE(?o, ?x, ?y) {
      ?o in OBJECTS;
?x in [-oo,+oo];
?y in [-oo,+oo];
?value in STATUS | {PICTURE_IDLE};
}

attribute PTU_POSITION() {
?value in PANTILT_POSITIONS | {PTU_POSITION_IDLE};
}


// La distance entre les deux points est dr = ((xf-xi)^2+(yf-yi)^2)^(1/2)
// On approche cette distance par d = X+Y, avec
// X = |xf-xi| et Y=|yf-yi|
// Cette approximation se rapproche de la distance reelle
// (la plus courte, en ligne droite) en la multipliant par [0.7,1]

#define distance(_xi, _yi, _xf, _yf, _d)\
  variable ?Y;\
  variable ?Yc;\
  variable ?Ypos;\
```

```
  variable ?X;\
  variable ?Xc;\
  variable ?Xpos;\
  _xi in [-oo,+oo];\
  _xf in [-oo,+oo];\
  _yi in [-oo,+oo];\
  _yf in [-oo,+oo];\
  _d in [-oo,+oo];\
  ?X in [-oo,+oo];\
  ?Xc in [-oo,+oo];\
  ?Xpos in [-oo,+oo];\
  ?Y in [-oo,+oo];\
  ?Yc in [-oo,+oo];\
  ?Ypos in [-oo,+oo];\
  ?X =. _xf -. _xi;\
  ?Xc =. _xi -. _xf;\
  ?Y =. _yf -. _yi;\
  ?Yc =. _yi -. _yf;\
  ?Xpos =. max(?X,?Xc);\
  ?Ypos =. max(?Y,?Yc);\
  _d =. ?Xpos +. ?Ypos

#define distance_uncertainty(_u)\
  _u in [0.7,1]

#define speed(_sp)\
  _sp in [0.17,0.35]
```

Modele1.task

```
#include "test-relations"

task MOVE(?x1,?y1,?x2,?y2)(t_start, t_end) {
    ?x1 in [0.0-10.0, 10.0];
    ?x2 in [0.0-10.0, 10.0];
    ?y1 in [0.0-10.0, 10.0];
    ?y2 in [0.0-10.0, 10.0];

    hold(PTU_POSITION():STRAIGHT,(t_start, t_end));
    hold(MVT_GENERATION_INITIALIZED():TRUE,(t_start, t_end));

    variable ?di;
    distance(?x1,?y1,?x2,?y2,?di);

    event(AT_ROBOT_X():(?x1,1000),t_start);
    hold(AT_ROBOT_X():1000,(t_start,t_end));
    event(AT_ROBOT_X():(1000,?x2),t_end);

    event(AT_ROBOT_Y():(?y1,1000),t_start);
    hold(AT_ROBOT_Y():1000,(t_start,t_end));
    event(AT_ROBOT_Y():(1000,?y2),t_end);

    event(ROBOT_STATUS():(STILL,MOVING),t_start);
    hold(ROBOT_STATUS():MOVING,(t_start,t_end));
    event(ROBOT_STATUS():(MOVING,STILL),t_end);

    variable ?du;
    distance_uncertainty(?du);

    variable ?dist;
```

```
    ?dist =. ?di *. ?du;

    variable ?mindist;
    ?mindist =. 0.5;
    ?mindist =. min(?di, ?mindist);

    variable ?s;
    speed(?s);

    variable ?duration;
    ?duration in [1.5, +oo];
    ?dist =. ?s *. ?duration;

    variable ?minduration;
    ?minduration =. 1.4;
    ?minduration =. min(?minduration, ?duration);

    contingent ?duration =. t_end - t_start;
}latePreemptive


task MOVE_PAN_TILT_UNIT(?initpos,?finpos)(t_start, t_end){
    ?initpos in PANTILT_POSITIONS;
    ?finpos in PANTILT_POSITIONS;
    ?initpos != ?finpos;

    event(PTU_POSITION():(?initpos,PTU_POSITION_IDLE),t_start);
    hold(PTU_POSITION():PTU_POSITION_IDLE,(t_start,t_end));
    event(PTU_POSITION():(PTU_POSITION_IDLE,?finpos),t_end);

    hold(PTU_DRIVER_INITIALIZED():TRUE,(t_start,t_end));

    contingent (t_end - t_start) in [2, 10];
}nonPreemptive


task TAKE_PICTURE(?obj,?x,?y)(t_start, t_end){
    ?obj in OBJECTS;
    ?x in [-oo,+oo];
    ?y in [-oo,+oo];

    hold(AT_ROBOT_X():?x,(t_start,t_end));
    hold(AT_ROBOT_Y():?y,(t_start,t_end));
    hold(PTU_POSITION():AT_MY_FEET,(t_start,t_end));

    event(PICTURE(?obj,?x,?y):(NONE,PICTURE_IDLE),t_start);
    hold(PICTURE(?obj,?x,?y):PICTURE_IDLE,(t_start,t_end));
    event(PICTURE(?obj,?x,?y):(PICTURE_IDLE,DONE),t_end);

    contingent (t_end - t_start) in [2, 6];
}nonPreemptive


task COMMUNICATE(?w)(t_start, t_end){
    ?w in VISIBILITY_WINDOWS;

    hold(VISIBILITY_WINDOW(?w):IN,(t_start,t_end));

    event(COMMUNICATION(?w):(NONE,COMMUNICATION_IDLE),t_start);
```

```
    hold(COMMUNICATION(?w):COMMUNICATION_IDLE,(t_start,t_end));
    event(COMMUNICATION(?w):(COMMUNICATION_IDLE,DONE),t_end);

    hold(ROBOT_STATUS():STILL,(t_start,t_end));

    contingent (t_end - t_start) in [18, 25];
}latePreemptive


task INIT_PTU_DRIVER()(t_start, t_end){
    hold(ROBOT_STATUS():STILL,(t_start,t_end));

    event(PTU_DRIVER_INITIALIZED():(FALSE,PTU_DRIVER_IDLE),t_start);
    hold(PTU_DRIVER_INITIALIZED():PTU_DRIVER_IDLE,(t_start,t_end));
    event(PTU_DRIVER_INITIALIZED():(PTU_DRIVER_IDLE,TRUE),t_end);

    contingent (t_end - t_start) in [2, 40];
}nonPreemptive


task INIT_MVT_GENERATION()(t_start, t_end){
    hold(ROBOT_STATUS() : STILL, (t_start, t_end));

    event(MVT_GENERATION_INITIALIZED():(FALSE,MVT_GENERATION_IDLE),t_start);
    hold(MVT_GENERATION_INITIALIZED():MVT_GENERATION_IDLE,(t_start,t_end));
    event(MVT_GENERATION_INITIALIZED():(MVT_GENERATION_IDLE,TRUE),t_end);

    contingent (t_end - t_start) in [15, 35];

}nonPreemptive
```

Modele1-init.task

```
#include "test-relations"

task Init()(t_start,t_end){

   timepoint t_svisi1, t_evisi1;
   timepoint t_svisi2, t_evisi2;

   timepoint t_goal1;
   timepoint t_goal3s,t_goal3e;
   timepoint t_goal4s,t_goal4e;
   timepoint t_goal5s,t_goal5e;
   timepoint t_goal6s,t_goal6e;
   timepoint t_goal7s, t_goal7e;


    //initial situation

    explained event(AT_ROBOT_X() : (1.0, 0.0), t_start);
    explained event(AT_ROBOT_Y() : (1.0, 0.0), t_start);
    explained event(ROBOT_STATUS() : (MOVING, STILL), t_start);

    explained event(PTU_POSITION() : (PTU_POSITION_IDLE, STRAIGHT), t_start);

    explained event(PTU_DRIVER_INITIALIZED() : (FALSE, TRUE), t_start);
    explained event(MVT_GENERATION_INITIALIZED() : (FALSE, TRUE), t_start);

    explained event(COMMUNICATION(W1) : (COMMUNICATION_IDLE, NONE), t_start);
```

```
    explained event(COMMUNICATION(W2) : (COMMUNICATION_IDLE, NONE), t_start);

    contingent event(VISIBILITY_WINDOW(W1) : (IN, OUT), t_start);
    contingent event(VISIBILITY_WINDOW(W2) : (IN, OUT), t_start);

    variable ?x1,?y1;
    ?x1 in [-oo,+oo];
    ?y1 in [-oo,+oo];
    explained event(PICTURE(OBJ1,?x1,?y1):(PICTURE_IDLE,NONE),t_start);

    variable ?x2,?y2;
    ?x2 in [-oo,+oo];
    ?y2 in [-oo,+oo];
    explained event(PICTURE(OBJ2,?x2,?y2):(PICTURE_IDLE,NONE),t_start);

    variable ?x3,?y3;
    ?x3 in [-oo,+oo];
    ?y3 in [-oo,+oo];
    explained event(PICTURE(OBJ3,?x3,?y3):(PICTURE_IDLE,NONE),t_start);

#include "test-goals"

    (t_end-t_start) in [330, 800];

}earlyPreemptive
```

Modele1-goals

```
//- Visibility windows
    contingent event(VISIBILITY_WINDOW(W1) : (OUT, IN), t_svisi1);
    contingent event(VISIBILITY_WINDOW(W1) : (IN, OUT), t_evisi1);

    contingent event(VISIBILITY_WINDOW(W2) : (OUT, IN), t_svisi2);
    contingent event(VISIBILITY_WINDOW(W2) : (IN, OUT), t_evisi2);

   (t_evisi1 - t_start) in [154, 156];
   t_svisi1 < t_evisi1;
   (t_svisi1 - t_start) in [125, 127];

   (t_evisi2 - t_start) in [267, 269];
   t_svisi2 < t_evisi2;
   (t_svisi2 - t_start) in [238, 240];



 //- Goals
    //- Etre Rentre Au Lander A La Fin De L'Horizon
    hold(AT_ROBOT_X() : 0.0, (t_goal1, t_end)) goal(4, 0);
    hold(AT_ROBOT_Y() : 0.0, (t_goal1, t_end)) goal(4, 0);

    //- avoir communique  pendant les deux fenetres de visibilite
    hold(COMMUNICATION(W1) : DONE, (t_goal3s, t_goal3e)) goal(3, 0);
    hold(COMMUNICATION(W2) : DONE, (t_goal4s, t_goal4e)) goal(3, 0);

    //- prendre des images
    hold(PICTURE(OBJ1, 0.0, 0.0) : DONE, (t_goal5s, t_goal5e)) goal(2, 0);
    hold(PICTURE(OBJ2, 4.0, 0.0) : DONE, (t_goal6s, t_goal6e)) goal(2, 0);
    hold(PICTURE(OBJ3, 3.0, 0 - 3.0) : DONE, (t_goal7s, t_goal7e)) goal(2, 0);

   (t_end - t_goal1) in [2, 4];
```

```
   (t_goal3e - t_evisi1) in [2, 4];
   (t_goal4e - t_evisi2) in [2, 4];
```

A.2 Modèle2

Nous présentons ici les différents fichiers composant le Modèle2.

Modele2-relations

```
constant STATE = {IDLE, INITED, WORKING};
constant BOOLEAN = {TRUE,FALSE};
constant WINDOWS = {W1,W2,W3,W4};
constant OBJECTS = {O1,O2,O3,O4,O5,O6};
constant LOCALISATION = {G1, G2, G3, G4, G5, G6, G7, G8, G9, G10,
                         G11,G12, G13, G14, G15, G16, U1, U2, U3,
                         U4, U5, U6, U7, U8};
constant ENSDIST = {D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11,
                    D12, D13, D14, D15, D16, D17, D18, D19, D20, D21,
                    D22, D23, D24, D25, D26, D27, D28, D29, D30, D31,
                    D32, D33, D34, D35, D36, D37, D38, D39, D40, D41,
                    D42, D43, D44, D45, D46, D47, D48, D49, D50, D51,
                    D52, D53, D54, D55, D56, D57, D58, D59, D60, D61,
                    D62, D63, D64, D65, D66, D67, D68};
constant DIST = {D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12,
                 D13, D14, D15, D16, D17, D18, D19, D20, D21, D22,
                 D23, D24, D25, D26, D27, D28, D29, D30};
constant MODE = {NOMINAL, RECOV};


attribute FRAME_STATE () {
?value in STATE;
}

attribute FRAME_POS () {
?value in LOCALISATION | OBJECTS | {MOVING};
}

attribute FRAME_MODE () {
?value in MODE;
}

attribute ANTENNA_STATE () {
?value in STATE;
}

attribute WINDOW(){
?value in BOOLEAN;
}

attribute COMMUNICATION(?w){
        ?w in WINDOWS;
?value in BOOLEAN;
}


attribute SHOT(?o) {
      ?o in OBJECTS;
?value in BOOLEAN;
}
```

```
attribute PTU_STATE () {
?value in STATE;
}

attribute PTU_PAN () {
?value in [0. -. 90. , 100.];
}

attribute PTU_TILT () {
?value in [0. -. 45. , 50.];
}

#include "test-distance"

#define distance_uncertainty(_u)\
  _u in [1,1.2]

#define speed(_sp)\
  _sp in [0.17,0.35]
```

Modele2.task

```
#include "test-relations"

task MOVE_NOMINAL(?p1, ?p2)(t_start, t_end) {
    ?p1 in OBJECTS;
    ?p2 in OBJECTS;

    hold(PTU_PAN():0.0, (t_start,t_end));
    hold(PTU_TILT():0. -. 30. , (t_start,t_end));

    event(FRAME_STATE():(INITED, WORKING), t_start);
    hold(FRAME_STATE():WORKING, (t_start, t_end));
    event(FRAME_STATE():(WORKING, INITED), t_end);

    hold(FRAME_MODE():NOMINAL, (t_start, t_end));

    variable ?di;
    ?di in [0,+oo];
    distance(?p1,?p2,?di);

    event(FRAME_POS():(?p1,MOVING), t_start);
    hold(FRAME_POS():MOVING, (t_start, t_end));
    event(FRAME_POS():(MOVING,?p2), t_end);

    variable ?s;
    speed(?s);

    variable ?duration;
    ?duration in [0, +oo];
    ?di =. ?s *. ?duration;

    contingent ?duration =. t_end - t_start;
}latePreemptive


task MOVE_RECOV(?p1, ?p2)(t_start, t_end) {
    ?p1 in LOCALISATION;
```

```
    ?p2 in OBJECTS;

    hold(FRAME_MODE():RECOV, (t_start, t_end));
    event(FRAME_MODE():(RECOV,NOMINAL), t_end);

    hold(PTU_PAN():0.0, (t_start,t_end));
    hold(PTU_TILT():0. -. 30. , (t_start,t_end));

    event(FRAME_STATE():(INITED, WORKING), t_start);
    hold(FRAME_STATE():WORKING, (t_start, t_end));
    event(FRAME_STATE():(WORKING, INITED), t_end);

    variable ?di;
    ?di in [0,+oo];
    distance_recov(?p1,?p2,?di);

    event(FRAME_POS():(?p1,MOVING), t_start);
    hold(FRAME_POS():MOVING, (t_start, t_end));
    event(FRAME_POS():(MOVING,?p2), t_end);

    variable ?s;
    speed(?s);

    variable ?duration;
    ?duration in [0, +oo];
    ?di =. ?s *. ?duration;

    contingent ?duration =. t_end - t_start;
}latePreemptive


task MOVE_PTU(?pan1,?tilt1,?pan2,?tilt2)(t_start, t_end){
    ?tilt1 in [0. -. 45., 45.];
    ?tilt2 in [0. -. 45., 45.];
    ?pan1 in [0. -. 90., 90.];
    ?pan2 in [0. -. 90., 90.];

    hold(PTU_STATE():INITED, (t_start, t_end));

    event(PTU_TILT():(?tilt1, 50.0), t_start);
    hold(PTU_TILT():50.0,(t_start, t_end));
    event(PTU_TILT():(50.0, ?tilt2), t_end);

    event(PTU_PAN():(?pan1, 100.0), t_start);
    hold(PTU_PAN():100.0,(t_start, t_end));
    event(PTU_PAN():(100.0, ?pan2), t_end);

    contingent (t_end - t_start) in [2, 10];
}nonPreemptive


task TAKE_SHOT(?obj)(t_start, t_end){
    ?obj in OBJECTS;
    variable ?state;
    ?state in {INITED, IDLE};

    hold(FRAME_POS():?obj, (t_start, t_end));

    hold(FRAME_STATE():?state, (t_start, t_end));
```

```
    hold(PTU_PAN():0.0, (t_start, t_end));
    hold(PTU_TILT(): 0. -. 45., (t_start, t_end));

    event(SHOT(?obj):(FALSE, TRUE), t_end);

    contingent (t_end - t_start) in [2, 6];
}nonPreemptive


task COMMUNICATE_ANTENNA(?w)(t_start, t_end){
    ?w in WINDOWS;
    variable ?state;
    ?state in {INITED, IDLE};

    hold(WINDOW():TRUE, (t_start, t_end));

    event(COMMUNICATION(?w):(FALSE, TRUE), t_end);

    hold(ANTENNA_STATE():INITED, (t_start, t_end));


    hold(FRAME_STATE():?state, (t_start, t_end));

    contingent (t_end - t_start) in [18, 25];
}latePreemptive


//- taches pour les reprises d'echec
task INIT_PTU()(t_start, t_end){
    variable ?state;
    ?state in {INITED, IDLE};

    hold(FRAME_STATE():?state, (t_start, t_end));

    hold(PTU_STATE():IDLE, (t_start, t_end));
    event(PTU_STATE():(IDLE, INITED), t_end);

    contingent (t_end - t_start) in [2, 40];
}nonPreemptive


task INIT_FRAME()(t_start, t_end){
    hold(FRAME_STATE():IDLE, (t_start, t_end));
    event(FRAME_STATE():(IDLE, INITED), t_end);

    contingent (t_end - t_start) in [15, 35];
}nonPreemptive
```

Modele2-distances

```
#define distance(_p1, _p2, _d)\
  variable ?D;\
  ?D in DIST;\
  _p2 in {O1} => ?D in {D1};\
  _p2 in {O2} => ?D in {D2,D3};\
  _p2 in {O3} => ?D in {D4,D5};\
  _p1 in {O1} => ?D in {D2,D4};\
  _p1 in {O2} => ?D in {D1,D5};\
  _p1 in {O3} => ?D in {D1,D3};\
```

```
  ?D in {D1} => _d =. 6.24;\
  ?D in {D2} => _d =. 6;\
  ?D in {D3} => _d =. 5.16;\
  ?D in {D4} => _d =. 6.24;\
  ?D in {D5} => _d =. 5.16

#define distance_recov(_p1, _p2, _d)\
  variable ?D;\
  ?D in ENSDIST;\
  _p2 in {O1} => ?D in {D1,D2,D3,D4,D5};\
  _p2 in {O2} => ?D in {D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16};\
  _p2 in {O3} => ?D in {D17,D18,D19,D20,D21,D22,D23,D24,D25};\
  _p1 in {G1} => ?D in {D1,D6,D17};\
  _p1 in {G2} => ?D in {D2,D7,D18};\
  _p1 in {G3} => ?D in {D2,D7,D19};\
  _p1 in {G4} => ?D in {D1,D6,D19};\
  _p1 in {G5} => ?D in {D2,D6,D17};\
  _p1 in {G6} => ?D in {D3,D8,D18};\
  _p1 in {G7} => ?D in {D3,D8,D19};\
  _p1 in {G8} => ?D in {D2,D6,D19};\
  _p1 in {G9} => ?D in {D2,D9,D20};\
  _p1 in {G10} => ?D in {D3,D10,D21};\
  _p1 in {G11} => ?D in {D3,D10,D18};\
  _p1 in {G12} => ?D in {D2,D9,D18};\
  _p1 in {G13} => ?D in {D1,D11,D22};\
  _p1 in {G14} => ?D in {D2,D12,D20};\
  _p1 in {G15} => ?D in {D2,D12,D17};\
  _p1 in {G16} => ?D in {D1,D11,D17};\
  _p1 in {U1} => ?D in {D4,D13,D23};\
  _p1 in {U2} => ?D in {D5,D14,D24};\
  _p1 in {U3} => ?D in {D4,D13,D24};\
  _p1 in {U4} => ?D in {D5,D13,D24};\
  _p1 in {U5} => ?D in {D4,D15,D23};\
  _p1 in {U6} => ?D in {D5,D16,D23};\
  _p1 in {U7} => ?D in {D4,D15,D25};\
  _p1 in {U8} => ?D in {D5,D13,D23};\
  ?D in {D1} => _d =. 8.48;\
  ?D in {D2} => _d =. 6.7;\
  ?D in {D3} => _d =. 4.24;\
  ?D in {D4} => _d =. 11.31;\
  ?D in {D5} => _d =. 8.54;\
  ?D in {D6} => _d =. 7.21;\
  ?D in {D7} => _d =. 3.6;\
  ?D in {D8} => _d =. 5;\
  ?D in {D9} => _d =. 9.21;\
  ?D in {D10} => _d =. 7.61;\
  ?D in {D11} => _d =. 11.66;\
  ?D in {D12} => _d =. 10.44;\
  ?D in {D13} => _d =. 10.63;\
  ?D in {D14} => _d =. 5;\
  ?D in {D15} => _d =. 14.42;\
  ?D in {D16} => _d =. 12.36;\
  ?D in {D17} => _d =. 9.48;\
  ?D in {D18} => _d =. 6.7;\
  ?D in {D19} => _d =. 4.24;\
  ?D in {D20} => _d =. 10.81;\
  ?D in {D21} => _d =. 8.48;\
  ?D in {D22} => _d =. 12.72;\
  ?D in {D23} => _d =. 12.52;\
```

```
  ?D in {D24} => _d =. 7.81;\
  ?D in {D25} => _d =. 15.55
```

Modele2-init.task

```
#include "test-relations"

task Init()(t_start,t_end){

   timepoint t_svisi1, t_evisi1;
   timepoint t_svisi2, t_evisi2;

   timepoint t_goal1;
   timepoint t_goal3s,t_goal3e;
   timepoint t_goal4s,t_goal4e;
   timepoint t_goal5s,t_goal5e;
   timepoint t_goal6s,t_goal6e;
   timepoint t_goal7s, t_goal7e;


    //initial situation

    explained event(FRAME_POS() : (MOVING, O1), t_start);
    explained event(FRAME_MODE() : (RECOV, NOMINAL), t_start);
    explained event(FRAME_STATE() : (IDLE, INITED), t_start);

    explained event(PTU_PAN() : (90, 0), t_start);
    explained event(PTU_TILT() : (45, 0), t_start);
    explained event(PTU_STATE() : (IDLE, INITED), t_start);

    explained event(ANTENNA_STATE() : (IDLE, INITED), t_start);

    explained event(COMMUNICATION(W1) : (TRUE, FALSE), t_start);
    explained event(COMMUNICATION(W2) : (TRUE, FALSE), t_start);
    explained event(COMMUNICATION(W3) : (TRUE, FALSE), t_start);
    explained event(COMMUNICATION(W4) : (TRUE, FALSE), t_start);

    contingent event(WINDOW() : (TRUE, FALSE), t_start);

    explained event(SHOT(O1) : (TRUE, FALSE), t_start);
    explained event(SHOT(O2) : (TRUE, FALSE), t_start);
    explained event(SHOT(O3) : (TRUE, FALSE), t_start);

#include "test-goals"

    (t_end-t_start) in [330, 900];

}earlyPreemptive
```

Modele2-goals

```
//- Visibility windows
    contingent event(WINDOW() : (FALSE, TRUE), t_svisi1);
    contingent event(WINDOW() : (TRUE, FALSE), t_evisi1);

    contingent event(WINDOW() : (FALSE, TRUE), t_svisi2);
    contingent event(WINDOW() : (TRUE, FALSE), t_evisi2);


   (t_evisi1 - t_start) in [154, 156];
   t_svisi1 < t_evisi1;
   (t_svisi1 - t_start) in [125, 127];
```

```
   (t_evisi2 - t_start) in [267, 269];
   t_svisi2 < t_evisi2;
   (t_svisi2 - t_start) in [238, 240];


//- Goals
    //- Etre Rentre Au Lander A La Fin De L'Horizon
    hold(FRAME_POS() : O1, (t_goal1, t_end)) goal(4, 0);

    //- avoir communique pendant les deux fenetres de visibilite
    hold(COMMUNICATION(W1) : TRUE, (t_goal3s, t_goal3e)) goal(3, 0);
    hold(COMMUNICATION(W2) : FALSE, (t_start, t_evisi1));
    hold(COMMUNICATION(W2) : TRUE, (t_goal4s, t_goal4e)) goal(3, 0);

    //- prendre des images
    hold(SHOT(O1) : TRUE, (t_goal5s, t_goal5e)) goal(2, 0);
    hold(SHOT(O2) : TRUE, (t_goal6s, t_goal6e)) goal(2, 0);
    hold(SHOT(O3) : TRUE, (t_goal7s, t_goal7e)) goal(2, 0);


    (t_end - t_goal1) in [2, 4];
    (t_goal3e - t_evisi1) in [2, 4];
    (t_goal4e - t_evisi2) in [2, 4];
```

Bibliographie

[Alami et al. 1998] R. Alami, R. Chatila, S. Fleury, M. Ghallab, & F. Ingrand, An Architecture for Autonomy, *The International Journal of Robotics Research*, 17(4) :315–337, 1998.

[Arlat et al. 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, & D. Powell, Fault Injection for Dependability Validation - A Methodology and Some Applications, *IEEE Transactions on Software Engineering*, 16(2) :166–182, 1990.

[Avižienis et al. 2005] A. Avižienis, J. C. Laprie, B. Randell, & C. Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, *IEEE Trans. on Dependable and Secure Computing*, 1(1) :11–33, 2005.

[Becker & Smith 2005] M. Becker & D. R. Smith, Model Validation in Planware, dans *ICAPS 2005 Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems*, Monterey, California, 2005.

[Bedrax-Weiss et al. 2005] T. Bedrax-Weiss, J. Frank, M. Iatauro, & C. McGann, Inspection and Verification of Domain Models with PlanWorks and Aver, dans *ICAPS 2005 Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems*, Monterey, California, 2005.

[Bernard et al. 2000] D. E. Bernard, E. B. Gamble, N. F. Rouquette, B. Smith, Y. W. Tung, N. Muscettola, G. A. Dorias, B. Kanefsky, J. Kurien, W. Millar, P. Nayal, K. Rajan, & W. Taylor, Remote Agent Experiment DS1 Technology Validation Report, Ames Research Center and JPL, 2000.

[Boden 1989] M. C. Boden, Benefits and Risks of Knowledge-Based Systems, 1989, Oxford University Press (ISBN 0-19854-743-9).

[Borrelly et al. 1998] J. J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, & N. Turro, The ORCCAD Architecture, *The International Journal of Robotics Research*, 17(4) :338–359, 1998.

[Bovet & Cesati 2002] D. P. Bovet & M. Cesati, *Understanding the Linux Kernel, Second Edition*. O'Reilly, 2002, (ISBN 0-596-00213-0).

[Brooks 1986] R. A. Brooks, A Robust Layered Control System for a Mobile Robot, *IEEE Magazine on Robotics & Automation*, 2(1) :14–23, 1986.

[Carlson & Murphy 2003] J. Carlson & R. R. Murphy, Reliability Analysis of Mobile Robots, dans *Proceedings of the 2003 IEEE International Conference on Robotics & Automation*, pages 274–281, Taipei, Taiwan, 2003.

[Chen 1997] I. R. Chen, Effects of Parallel Planning on System Reliability of Real-Time Expert Systems, *IEEE Transactions on Reliability*, 46(1) :81–87, 1997.

[Chen et al. 1995] I. R. Chen, F. B. Bastani, & T. W. Tsao, On the Reliability of AI Planning Software in Real-Time Applications, *IEEE Transactions on Knowledge and Data Engineering*, 7(1) :14–25, 1995.

[Clough 2002] B. Clough, Metrics, Schmetrics ! How the Heck do you Determine a UAV's Autonomy Anyway ?, dans *Proceedings of the Performance Metrics for Intelligent Systems (PerMIS2002) Workshop*, Gaithersburg, MD, 2002.

[Crouzet et al. 2006] Y. Crouzet, H. Waeselynck, B. Lussier, & D. Powell, The SESAME experience : from assembly languages to declarative models, dans *Proceedings of the 2nd Workshop on Mutation Analysis (Mutation'2006)*, Raleigh, NC, 2006.

[Daran 1996] M. Daran, *Modélisation des comportements érronés du logiciel et application à la validation des tests par injection de fautes*, Thèse de doctorat, Institut National Polytechnique de Toulouse, 1996, Rapport LAAS No. 96497.

[Despouys 2000] O. Despouys, *Une architecture intégrée pour la planification et le contrôle d'exécution en environnement dynamique*, Thèse de doctorat, Institut National Polytechnique de Toulouse, 2000, Rapport LAAS No. 00541.

[Dousson 1994] C. Dousson, *Suivi d'évolutions et reconnaissance de chroniques*, Thèse de doctorat, Université Paul Sabatier, 1994, Rapport LAAS No. 94394.

[Duchemin et al. 2004] G. Duchemin, P. Poignet, E. Dombre, & F. Pierrot, Medically safe and sound, *IEEE Magazine on Robotics & Automation*, 11(2) :46–55, 2004.

[Essamé et al. 2000] D. Essamé, J. Arlat, & D. Powell, Tolérance aux Fautes dans les Systèmes Critiques, Rapport 00151, LAAS-CNRS, 2000.

[Feather & Smith 1999] M. S. Feather & B. Smith, Automatic Generation of Test Oracles - From Pilot Studies to Application, dans *Proceedings of the 14th IEEE Automated Software Engineering Conference (ASE-99)*, Cocoa Beach, Florida, 1999.

[Finzi et al. 2003] A. Finzi, F. Ingrand, & N. Muscettola, Robot Action Planning and Execution Control, Rapport 03566, LAAS-CNRS, 2003.

[Fleury et al. 1997] S. Fleury, M. Herrb, & R. Chatila, Genom : A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture, dans *Proceedings of the 1997 International Conference on Intelligent Robots and Systems (IROS 97)*, Grenoble, France, 1997.

[Fox & Das 2000] J. Fox & S. Das, *Safe and Sound, Artificial Intelligence in Hazardous Applications*. The American Association for Artificial Intelligence Press, 2000, (ISBN 0-262-06211-9).

[Fox et al. 2005] M. Fox, R. Howey, & D. Long, Exploration of the Robustness of Plans, dans *ICAPS 2005 Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems*, Monterey, California, 2005.

[Gat 1997] E. Gat, On Three-Layer Architectures, In *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. P. Bonnasso, and R. Murphy editors, MIT/AAAI Press, pages 195-210, 1997.

[Ghallab & Mounir-Alaoui 1989] M. Ghallab & A. Mounir-Alaoui, Relations temporelles symboliques : Représentations et algorithmes, *Revue d'Intelligence Artificielle*, 3 :67–116, 1989.

[Goldman et al. 1997] R. P. Goldman, D. J. Musliner, M. S. Boddy, & K. D. Krebsbach, The CIRCA Model of Planning and Execution, dans *Working Notes of the AAAI Workshop on Robots, Softbots, Immobots : Theories of Action, Planning and Control*, Providence, Rhode Island, 1997, http ://citeseer.nj.nec.com/article/goldman97circa.html.

[Graf et al. 2004] B. Graf, M. Hans, & R. D. Schraft, Mobile Robot Assistants - Issues for Dependable Operation in Direct Cooperation With Humans, *IEEE Magazine on Robotics & Automation*, 11(2) :67–77, 2004.

[Guesnier et al. 1997] G. Guesnier, J. F. Hamelin, & J. M. Peyrouton, Centrales nucléaires N4 : l'informatique au service d'une conduite plus sûre, *Epure*, (56) :59–74, 1997.

[Guiochet & Powell 2006] J. Guiochet & D. Powell, Etude et analyse de systèmes indépendants de sécurité-innocuité de type safety bag, Rapport 05551, LAAS-CNRS, 2006.

[Howe 1995] A. E. Howe, Improving the Reliability of Artificial Intelligence Planning Systems by Analyzing their Failure Recovery, *IEEE Transactions on Knowledge and Data Engineering*, 7(1) :14–25, 1995.

[Howey et al. 2004] R. Howey, D. Long, & M. Fox, VAL : Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning using PDDL, dans *16th IEEE International Conference on Tools with Artificial Intelligence*, Boca Raton, Florida, 2004.

[Huang 2004] H. M. Huang, editor, *Autonomy Levels for Unmanned Systems (ALFUS) Framework*. Numéro NIST Special Publication 1011, 2004.

[Ingrand et al. 2001] F. Ingrand, R. Chatila, & R. Alami, An Architecture for Dependable Autonomous Robots, dans *Proceedings of the first IARP/IEEE-RAS Joint Workshop on Technical Challenge for Dependable Robots in Human Environments*, Seoul, Korea, 2001.

[Jarboui 2003] M. T. Jarboui, *Sûreté de fonctionnement de systèmes informatiques. Étalonnage et représentativité des fautes*, Thèse de doctorat, Institut National Polytechnique de Toulouse, 2003, Rapport LAAS No. 03245.

[Jones et al. 2001] C. Jones, M. O. Killijian, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, & R. Stroud, Revised Version of DSoS Conceptual Model (DC1), Rapport LAAS-CNRS No. 01441, EU IST-1999-11585 DSoS (Dependable Systems of Systems), 2001.

[Joyeux et al. 2005] S. Joyeux, A. Lampe, R. Alami, & S. Lacroix, Simulation in the LAAS Architecture, dans *Proceedings of Principles and Practice of Software Development in Robotics (SDIR2005), ICRA workshop*, Barcelona, Spain, 2005.

[Kim & Welch 1989] K. H. Kim & H. O. Welch, Distributed Execution of Recovery Blocks : An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications, *IEEE Transactions on Computers*, C-38 :626–636, 1989.

[Klein 1991] P. Klein, The Safety-Bag Expert System in the Electronic Railway Interlocking System Elektra, *Expert Systems with Apllications*, 3(4) :499–506, 1991.

[Knight et al. 2001] S. Knight, G. Rabideau, S. Chien, B. Engelhardt, & R. Sherwood, Casper : Space Exploration through Continuous Planning, *IEEE Intelligent Systems*, 16 :70–75, 2001.

[Laprie et al. 1996] J. C. Laprie, J. Arlat, J. P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J. C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac, & P. Thévenod, *Guide de la Sûreté de Fonctionnement (2e édition)*. Cépaduès - Éditions, 1996, (ISBN 2-85428-341-4).

[Laruelle 1994] H. Laruelle, *Planification temporelle et exécution de tâches en robotique*, Thèse de doctorat, Université Paul Sabatier de Toulouse, 1994, Rapport LAAS No. 94189.

[Lemai & Ingrand 2004] S. Lemai & F. Ingrand, Interleaving Temporal Planning and Execution in Robotics Domains, dans *Proceedings of AAAI-04*, pages 617–622, San Jose, California, 2004.

[Lemai-Chenevier 2004] S. Lemai-Chenevier, *IxTeT-eXeC : planification, réparation de plan et contrôle d'exécution avec gestion du temps et des ressources*, Thèse de doctorat, Institut National Polytechnique de Toulouse, 2004, Rapport LAAS No. 04432.

[Marchand et al. 1998] E. Marchand, E. Rutten, H. Marchand, & F. Chaumette, Specifying and Verifying Active Vision-Based Robotic Systems with the SIGNAL Environment, *The International Journal of Robotics Research*, 17(4) :418–432, 1998.

[Minguez & Montano 2004] J. Minguez & L. Montano, Nearness Diagram Navigation (ND) : Collision Avoidance in Troublesome Scenarios, *IEEE Transactions on Robotics and Automation*, 20 :45–59, 2004.

[Monterlo et al. 2006] M. Monterlo, S. Thrun, H. Dahlkamp, D. Stavens, & S. Strohband, Winning the DARPA Grand Challenge with an AI Robot, dans *American Association of Artificial Intelligence 2006 (AAAI06)*, Boston, MA, 2006.

[Morisset 2002] B. Morisset, *Vers un robot au comportement robuste. Apprendre à combiner des modalités sensori-motrices complémentaires*, Thèse de doctorat, Université Paul Sabatier de Toulouse, 2002, Rapport LAAS No. 02617.

[Morisset et al. 2004] B. Morisset, G. Infantes, M. Ghallab, & F. Ingrand, Robel : Synthesizing and Controlling Complex Robust Robot Behaviors, dans *4th International Cognitive Robotics Workshop*, Valencia, Spain, 2004.

[Muscettola et al. 2002] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, & C. Plaunt, IDEA : Planning at the Core of Autonomous Reactive Agents, dans *AIPS 2002 Workshop on On-line Planning and Scheduling*, Toulouse, France, 2002, http ://citeseer.nj.nec.com/593897.html.

[Muscettola et al. 1998] N. Muscettola, P. P. Nayak, B. Pell, & B. C. Williams, Remote Agent : To Boldly Go Where No AI System Has Gone Before, *Artificial Intelligence*, 103(1-2) :5–47, 1998.

[Musliner et al. 1993] D. J. Musliner, E. H. Durfee, & K. G. Shine, CIRCA : A Cooperative Intelligent Real-Time Control Achitecture, *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6) :1561–1574, 1993.

[Musliner et al. 1995] D. J. Musliner, E. H. Durfee, & K. G. Shine, World Modeling for the Dynamic Construction of Real-Time Control Plans, *AI Journal*, 74(1) :83–127, 1995.

[Nesnas et al. 2003] I. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, & W. S. Kim, CLARAty : An Architecture for Reusable Robotic Software, dans *Proceedings of the SPIE Aerosense Conference on Unmanned Ground Vehicle Technology*, Orlando, Florida, 2003, http ://robotics.jpl.nasa.gov/tasks/claraty/overview/publications/.

[Nilsson 1998] N. J. Nilsson, *Artificial Intelligence : A New Synthesis*. Morgan Kaufmann Publishers, 1998, (ISBN 1-55860-467-7).

[Py 2005] F. Py, *Contrôle d'Exécution dans une Architectue Hiérarchisée pour systèmes Autonomes*, Thèse de doctorat, Université Paul Sabatier de Toulouse, 2005, Rapport LAAS No. 05693.

[Py & Ingrand 2004] F. Py & F. Ingrand, Real-Time Execution Control for Autonomous Systems, dans *Proceedings of the 2nd European Congress ERTS, Embedded Real Time Software*, Toulouse, France, 2004.

[Randell 1975] B. Randell, System Structure for Software Fault Tolerance, *IEEE Transactions on Software Engineering*, SE-1 :220–232, 1975.

[Ranganathan & Koenig 2003] A. Ranganathan & S. Koenig, A Reactive Robot Architecture with Planning on Demand, dans *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1462–1468, Las Vegas, Nevada, 2003, www.cc.gatech.edu/ai/robot-lab/online-publications/Ran-Koenig03.pdf.

[Reeves 1997] G. Reeves, What really happened on Mars ?, 1997, document électronique, http ://research.microsoft.com/ mbj/Mars_Pathfinder/Authoritative_Account.html.

[Russell & Norvig 2002] S. Russell & P. Norvig, *Artificial Intelligence, A Modern Approach (2nd edition)*. Prentice Hall, 2002, (ISBN 0-13-790395-2).

[Santa Fe Institute 2001] Santa Fe Institute, document reference RS-2001-009, 2001, Posted 10-22-2001, http ://discuss.santafe.edu/robustness.

[Schneider et al. 1998] S. A. Schneider, V. W. Chen, G. Pardo-Castellote, & H. H. Wang, ControlShell : A Software Architecture for Complex Electromecanical Systems, *The International Journal of Robotics Research*, 17(4) :360–380, 1998.

[Tomatis et al. 2003] N. Tomatis, G. Terrien, R. Piguet, D. Burnier, S. Bouabdallah, K. O. Arras, & R. Siegwart, Designing a Secure and Robust Mobile Interacting Robot for the Long Term, dans *Proceedings of the 2003 IEEE International Conference on Robotics & Automation*, pages 4246–4251, Taipei, Taiwan, 2003.

[Trinquart & Ghallab 2001] R. Trinquart & M. Ghallab, An Extended Functional Representation in Temporal Planning : towards Continuous Chnage, dans *Proceedings of the 6th European Conference on Planning (ECP'01)*, Toledes, Spain, 2001.

[Volpe et al. 2000] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, & H. Das, CLARAty : Coupled Layer Architecture for Robotic Autonomy, Rapport D-19975, NASA - Jet Propulsion Laboratory, 2000.