Test aléatoire de la navigation de robots dans des mondes virtuels

Thierry Sotiropoulos

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :
l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Présentée et soutenue le *04/05/2018* **par :**

SOTIROPOULOS THIERRY

Test aléatoire de la navigation de robots dans des mondes virtuels

---

JURY

LYDIE DU BOUSQUET	Professeur des Universités	Rapporteur
DAVID ANDREU	Maître de conférences	Rapporteur
PATRICK DANÈS	Professeur des Universités	Examinateur
JACQUES MALENFANT	Professeur des Universités	Examinateur
ANTOINE ROLLET	Maître de conférences	Examinateur
FÉLIX INGRAND	Chargé de recherche CNRS	Directeur de thèse
JÉRÉMIE GUIOCHET	Maître de conférences	Directeur de thèse
HÉLÈNE WAESELYNCK	Directrice de recherche CNRS	Directrice de thèse
SIMON VERNHES	Ingénieur à Naïo Technologies	Invité

École doctorale et spécialité :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance
Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes
Directrice et directeurs de Thèse :
Jérémie Guiochet, Félix Ingrand, Hélène Waeselynck

Au lecteur interloqué,
qu'il se rassure : c'était très clair dans ma tête.

It is questionable, if all the mechanical inventions
yet made have lightened the day's toil of any human being.
John Suart Mill, Principes d'économie politique.

Remerciements

Les travaux présentés dans ce manuscrit ont été effectués au sein du Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique (LAAS-CNRS). Je remercie Messieurs Jean Arlat et Liviu Nicu, qui ont successivement assuré la direction du LAAS-CNRS depuis mon entrée, pour m'avoir accueilli au sein de ce laboratoire. Je remercie également Monsieur Mohamed Kaâniche, Directeur de recherche CNRS et responsable de l'équipe Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF), ainsi que Monsieur Rachid Alami, Directeur de recherche CNRS et responsable de l'équipe Robotique et InterractionS (RIS), pour avoir permis la réalisation de mes travaux de thèse dans ces équipes.

La recherche scientifique est un travail d'équipe, aussi je remercie Madame Hélène Waeselynck ainsi que Messieurs Jérémie Guiochet et Félix Ingrand pour m'avoir guidé et soutenu en tant que directeurs de mes travaux de thèse. Ces travaux n'auraient pas pu aboutir sans leur concours, leur disponibilité et leur bienveillance. J'adresse également mes sincères remerciements aux autres membres du jury qui ont accepté de juger mon travail :

— Lydie Du Bousquet, Professeur des Universités, Université Grenoble Alpes, Rapporteur ;

— David Andreu, Maître de conférences, Université de Montpellier, Rapporteur ;

— Patrick Danès, Professeur des Universités, Université de Toulouse, Examinateur ;

— Jacques Malenfant, Professeur des Universités, Sorbonne Université, Examinateur ;

— Antoine Rollet, Maître de conférences, ENSEIRB/MATMECA, Examinateur ;

— Simon Vernhes, Ingénieur à Naïo Technologies, Invité.

Je vous suis très reconnaissant de l'intérêt porté à mes travaux.

Je remercie l'entreprise Naïo Technologies, et plus particulièrement Simon Vernhes, Pascal Schmidt et Gaëtan Séverac pour avoir mis à disposition les moyens de mener à bien nos expériences ainsi que pour avoir répondu à nos nombreuses questions. Je remercie également très chaleureusement Clément Robert pour l'aide apportée sur l'étude de cas Naïo, dans le cadre de son stage de fin d'études d'ingénieur.

Je souhaite remercier chaleureusement le personnel de l'École Doctorale Mathématiques, Informatique et Télécommunications de Toulouse (EDMITT) : Mesdames Agnès Requis et Martine Labruyère pour leur professionnalisme et leur patience.

Je remercie toutes les personnes de l'équipe TSF et RIS et notamment Madame Karama Kanoun, ainsi que les doctorants et stagiaires qui ont participé à faire de mes travaux de thèse une belle aventure humaine et scientifique et en particulier Benoît Morgan, William Excoffon, Clément Robert, Aliénor Damien, Raphaël Lallemand et Carla Sauvanaud avec qui j'ai partagé mes bureaux. J'ai eu l'honneur et

la joie de rencontrer dans ces équipes des personnes que j'ai l'immense privilège de compter parmi mes plus fidèles amis et camarades.

Je remercie également Carla Sauvanaud, pour son soutien indéfectible, pour avoir relu mon manuscrit de thèse à plusieurs reprises et m'avoir écouté présenter mes travaux.

Mes pensées les plus émues vont bien sûr à ma famille. Je veux remercier ici mes grands parents Jean et Cécile Sotiropoulos, ainsi que Chrissanthi et Giorgio Serafini : j'ai eu une chance immense de naître entouré de tels modèles. Je remercie mes parents, sans qui tout cela n'aurait pas été possible, mon père qui m'a donné le goût de la science et de l'esprit critique depuis mon plus jeune âge, et ma mère qui m'a fourni les moyens d'y arriver par son impeccable éducation et son indéfectible amour. Enfin, je souhaite remercier mes deux petites sœurs, Lisa et Julie, qui sont sans aucun doute, les plus belles personnes que je connaisse.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Le robot, du tchèque *robota* qui signifie « travail » ou « labeur », était cantonné à ses débuts aux tâches industrielles, automatiques, dans des environnements structurés et séparés de l'homme. Grâce aux nombreuses avancées dans les domaines de la localisation, de la perception, de la navigation, de la sûreté de fonctionnement et de l'intelligence artificielle, les robots deviennent de plus en plus autonomes et peuvent être déployés dans des environnements moins structurés ainsi qu'être au contact des humains. Ils se déploient dans toutes sortes d'environnements allant de l'espace public à la planète Mars ou encore en support ou en remplacement de l'action humaine dans les industries, les champs ou lors de missions de recherche de survivants à la suite d'événements catastrophiques.

De manière générale, un robot mobile, un véhicule intelligent, un drone, ou plus généralement un système autonome, doivent accomplir des tâches sans ou avec peu de supervision humaine dans des environnements variés. Ce déploiement pose la question de la confiance que l'utilisateur peut placer dans de tels systèmes, dont une défaillance peut avoir des conséquences catastrophiques (collision avec un humain, échec d'une mission critique etc.). Afin de garantir cette confiance, le domaine de la sûreté de fonctionnement propose un ensemble de techniques, notamment pour l'élimination des fautes, dont le test reste une technique prépondérante.

Dans le cadre des systèmes autonomes, pour procéder à des tests en monde réel, il faut que les testeurs déplacent le robot dans un environnement réel particulier, ou dans une installation en laboratoire, et soient présents durant le test. Ces déplacements et ces présences ont un coût. De plus, le test en monde réel ne permet de tester qu'un nombre limité d'environnements. Enfin, si le robot présente une défaillance catastrophique durant le test, il peut se détériorer lui même ou détériorer son environnement, ou pire encore entrer en collision avec un humain. Le test en monde réel s'avère donc coûteux, limité en terme de situations testées et potentiellement dangereux pour l'environnement, le système lui-même et les testeurs.

Le développement des simulateurs 3D donne des moyens pour pallier les problèmes soulevés par le test en monde réel. En plus d'être par nature moins coûteuse et dangereuse, la simulation, couplée avec de la génération automatique d'environnements virtuels, peut s'avérer plus complète en terme de situations testées et par là même permettre un test plus intensif. Toutefois, soumettre un système autonome à du test intensif en simulation, soulève plusieurs problèmes.

En effet, on rencontre les problèmes classiques du test, autrement dit déterminer et générer automatiquement des entrées de test à fournir au système, et implémenter un oracle capable de déterminer, automatiquement, si le comportement du système est en adéquation avec les attentes des développeurs. La richesse des mondes à simuler mène à une explosion combinatoire lors de leur génération. La nature indéterministe des systèmes autonomes rend également plus complexes les problèmes classiques du test. Par exemple, deux trajectoires différentes peuvent être des succès

pour la mission, ce qui peut être complexe à détecter pour un oracle. D'autre part, l'avènement des simulateurs système en robotique pose des problèmes en terme de fidélité et la question de la représentativité des simulations vis-à-vis de la réalité reste ouverte.

Ce manuscrit de thèse rapporte une étude exploratoire sur le test de systèmes autonomes en simulation, et contribue à répondre aux problématiques présentées ci-dessus. En plus des contributions théoriques, deux robots mobiles différents sont étudiés dans nos travaux : le robot Mana et le robot Oz. Une plateforme de test est conçue et déployée sur deux simulateurs afin de mener à bien diverses expériences et tests sur ces robots.

Plan de la thèse

Ce manuscrit de thèse est structuré en quatre chapitres. Le chapitre 1 est dédié à la présentation du contexte de nos travaux ainsi que de l'état de l'art associé. Nous présentons les contextes de la sûreté de fonctionnement, des systèmes autonomes et de leur test, de la simulation et de la génération procédurale dans lesquels s'inscrivent nos travaux. Ce chapitre met en lumière les principales difficultés rencontrées quand on veut tester en simulation des systèmes autonomes soient : la génération d'entrées de test ainsi que la définition d'un oracle de test

Le chapitre 2 présente une première contribution qui s'appuie sur la définition et l'implémentation d'une plateforme expérimentale de test en simulation avec un robot mobile, le robot Mana. Au travers de la mise en place de cette plateforme de test, nous proposons des solutions concrètes aux problèmes soulevés au chapitre précédent. De plus, la contrôlabilité de la génération procédurale est abordée sous l'angle du contrôle de la *difficulté* des mondes générés. Ce premier travail mène à proposer une approche permettant de modéliser les mondes et de définir des niveaux de difficulté de mondes.

Le chapitre 3 présente une étude de la reproductibilité de fautes extraites de la navigation du robot Mana. L'objectif de ce chapitre est d'identifier si les fautes connues et corrigées dans un logiciel de navigation auraient pu être détectées via la simulation. Près de 10 ans de commit du logiciel de navigation (dont le module P3D qui est une version académique d'une technique de planification de trajectoires utilisée par la NASA sur ses rovers martiens) ont été ainsi analysés. Chaque faute relevée est étudiée pour déterminer si elle serait activable en simulation, et l'oracle nécessaire pour la détecter. De nombreuses recommandations sont extraites de cette étude, notamment sur les propriétés de l'oracle à mettre en place pour ce genre de système.

Dans le chapitre 4, les enseignements tirés des deux chapitres précédents sont mis en œuvre dans une étude de cas d'un robot agricole. Le système considéré, fourni par notre partenaire industriel Naïo est celui du robot bineur Oz. La plateforme de test définie et implémentée dans les chapitres précédents est améliorée et modifiée pour convenir au nouveau cas d'étude. Les conclusions des chapitres précédents

concernant la génération de monde et les oracles nécessaires sont validées par une campagne de test intensifs en simulation.

Enfin, la conclusion générale fait le bilan de nos travaux avant d'ouvrir sur les perspectives de nos contributions.

Publications de la thèse

Conférences internationales :

1. Thierry Sotiropoulos, Jérémie Guiochet, Félix, Ingrand, Hélène Waeselynck. *Can robot navigation bugs be found in simulation ? An exploratory study.* IEEE International Conference on Software Quality, Reliability and Security (QRS'17), Jul 2017, Prague, Czech Republic (`http://paris.utdallas.edu/qrs17/`)
2. Thierry Sotiropoulos, Jérémie Guiochet, Félix, Ingrand, Hélène Waeselynck. *Virtual Worlds for Testing Robot Navigation : a Study on the Difficulty Level.* 12th European Dependable Computing Conference (EDCC'16), Sep 2016, Göteborg, Sweden (`http://www.edcc2016.eu/`)

Conférence nationale :

3. Thierry Sotiropoulos, Jérémie Guiochet, Félix, Ingrand, Hélène Waeselynck. *Test de la navigation de systèmes autonomes dans des mondes virtuels.* Control Architectures of Robots (CAR'15), Jun 2015, Lyon, France (`http://www.lirmm.fr/gtcar/webcar/CAR2015/car-conference.fr/index.html`)

Chapitre 1

Contexte et concepts fondamentaux

Sommaire

1.1 Introduction

La robotique, cantonnée à ses débuts aux tâches industrielles, automatiques, dans des environnements structurés et séparés de l'homme, se déploie maintenant dans l'espace public, dans les champs, dans des zones dangereuses ou encore en support de l'action humaine dans les industries ou lors de missions de recherche de personnes à la suite d'événements catastrophiques. Ce déploiement pose la question de la confiance que l'utilisateur peut placer dans de tels systèmes, dont une *défaillance* peut avoir des conséquences catastrophiques. Afin de garantir cette confiance, le domaine de la sûreté de fonctionnement propose un ensemble de techniques, dont le test reste la technique prépondérante pour l'élimination des fautes.

Dans le cadre des systèmes autonomes, procéder à des tests en monde réel s'avère coûteux, limité en terme de situations testées et potentiellement dangereux pour

les testeurs, l'environnement et le système lui même. Le développement des simulateurs donne des moyens de pallier les problèmes soulevés par le test en monde réel. Même si des expérimentations sur le terrain resteront toujours nécessaires, le test en simulation apparaît comme une approche pragmatique pour valider le système sur un plus grand nombre de situations de test, à moindre coût et sans aucun risque physique. Nos travaux s'intéressent ainsi au test de robots autonomes dans des mondes virtuels, en utilisant des moyens de simulation avancés. Les problématiques sous-jacentes sont les problématiques générales du test, mais avec de fortes spécificités introduites par le type des systèmes considérés et leur immersion dans des mondes virtuels.

La présentation du contexte de nos travaux démarre par une introduction générale au test en section 1.2. Les principaux enjeux du test y sont présentés : les problèmes concernant les entrées de test et l'oracle. La section 1.3 précise le type de systèmes auxquels s'appliquent nos travaux. Elle discute d'abord de la notion d'autonomie, de l'architecture des systèmes autonomes, puis décrit un exemple typique de service autonome : la navigation, qui permet à un système mobile d'évoluer dans un environnement a priori inconnu et non structuré. Elle discute également des fautes qui affectent les systèmes autonomes. La section 1.4 présente l'état de l'art du test dans le cadre des systèmes autonomes, en montrant les diverses stratégies spécifiquement déployées pour répondre aux enjeux du test identifiés à la section 1.2. La section 1.5 traite des simulateurs et de leur fonctionnalités avant d'aborder des techniques de génération procédurale de mondes virtuels.

1.2 Test

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 [Laprie 1996]. Divers moyens, qu'il faut utiliser de manière combinée, sont proposés par ce domaine et se classent dans quatre grandes catégories : prévention des fautes, tolérance aux fautes, élimination des fautes et prévision de fautes. Nos travaux considèrent en particulier l'élimination des fautes. Ce moyen comprend trois étapes : vérification, diagnostic et correction. La vérification peut être statique, c'est-à-dire sans activation du système – par exemple une preuve mathématique – ou dynamique, c'est-à-dire avec activation du système. Le test est défini comme la vérification dynamique d'un système effectuée à l'aide d'entrées valuées.

Le test consiste à exécuter un système en lui fournissant des entrées de test, puis à vérifier si les sorties correspondent à celles attendues afin d'élaborer un verdict d'acceptation ou de rejet. L'élaboration de ce verdict est appelé le problème de l'oracle. Le test peut être pratiqué sur toutes les abstractions – modèles exécutables, logiciels ou matériels – et sur toutes les décompositions – du test unitaire où une petite partie, par exemple une procédure ou une classe, du système est testée, jusqu'au niveau système. Le test peut viser à évaluer différentes caractéristiques du système sous test (test de robustesse, test fonctionnel, test de performance, test

de non régression etc.). Pour qu'une faute soit révélée, il faut que les entrées du test activent la faute, que celle-ci se propage sous forme d'erreur jusqu'à affecter le service et le faire dévier du comportement attendu. Cette déviation s'appelle une *défaillance*.

Apparaissent là deux enjeux principaux :

— sauf cas trivial, un ensemble infini d'entrées est possible, il faut donc sélectionner un sous ensemble jugé représentatif ;

— les résultats du test devant être analysés, il faut élaborer un oracle.

1.2.1 Sélection et production des entrées de test

Concernant les entrées de test, une première étape indispensable est de définir le domaine d'entrée, c'est-à-dire le type et la plage de valeurs pour chaque entrée potentielle. Des critères de couverture de test sont ensuite utilisés afin de guider méthodiquement la génération des cas de test. Une vue de ces critères est donnée dans [Ammann 2008]. Ils prennent la forme de critères de couverture (de code, de modèles de comportement du système sous test, de classes d'entrée, de modèles de faute ou de propriétés sur le comportement du système sous test) et sont basés sur l'analyse de la structure du logiciel (*white-box testing*) ou sur l'analyse des fonctions que celui-ci doit réaliser (*black-box testing*). Ces critères se traduisent en un ensemble d'éléments que le test devra couvrir. Afin de les couvrir, deux démarches sont possibles :

— soit chercher à générer *a priori* des cas de tests ;

— soit vérifier cette couverture *a posteriori*.

Générer des cas de tests *a priori* est plus difficile à automatiser et fait appel aux outils de génération de tests dont une vue est donnée dans [Anand 2013]. Certains de ces outils incorporent de la génération aléatoire, voire des procédés de recherche méta-heuristique. Ces procédés sont utiles lorsque l'on ne sait pas produire les entrées en fonction des éléments à couvrir. Dans ce cas, on ramène la génération de tests à un problème d'optimisation [McMinn 2004]. Typiquement, un ou plusieurs cas de tests sont évalués à l'aide d'une fonction objectif (*fitness* en anglais). La méta-heuristique choisie modifiera aléatoirement le ou les cas de test afin d'en améliorer la valeur retournée par la *fitness*, le but étant de trouver l'optimum global.

Les outils de génération de tests restent principalement développés et utilisés dans un cadre académique. La pratique industrielle est encore largement manuelle, avec éventuellement une vérification *a posteriori* à l'aide d'outils d'analyse de couverture, en particulier pour la couverture structurelle. Celle-ci sert alors à mesurer la qualité du test.

1.2.2 Oracle de test

L'oracle de test est le procédé permettant d'émettre un verdict d'acceptation ou de rejet des résultats de test. Le problème de l'oracle est crucial, en effet une mauvaise définition de celui-ci peut mener à ne pas repérer un comportement défaillant

	Niveaux d'interaction avec l'humain				
	Loin	**Proche**	**Contact sans mouvement**	**Contact avec mouvement**	**Support**
Automatique	Bras robotique	Aspirateur robotique	Robot éducatif	Robot d'aide à la personne	Exo-squelette
Autonomie	Robot d'exploration	Robot de surveillance	Robot guide	Robot collègue de travail	Voiture autonome

TABLE 1.1 – Exemples d'applications robotiques par rapport à l'autonomie et à l'interaction (source : [Guiochet 2017]).

qui aurait eu lieu pendant le test. Une solution est de déterminer manuellement les valeurs de sorties attendues. Bien que la plus utilisée, cette solution est fastidieuse et potentiellement source d'erreur. L'automatisation peut se faire à partir d'une spécification formelle complète ou partielle du comportement du système sous test. Une telle automatisation est souvent complexe à mettre en œuvre. Trois raisons à cette complexité sont évoquées dans [Weyuker 1982] :

— le programme donne une réponse non connue préalablement, il est donc difficile de savoir si elle est correcte ou pas ;

— le programme produit un volume trop important de sorties ;

— la spécification fournie au testeur contient des erreurs.

L'auteur évoque ensuite les moyens à notre disposition pour mitiger ces problèmes, par exemple se concentrer sur un sous ensemble des sorties, sur des entrées dont les sorties sont connues, comparer diverses implémentations du même programme (aussi appelé pseudo-oracle). Une vue synthétique plus récente des oracles est donnée dans [Barr 2015].

Le problème de l'oracle, comme celui de la sélection des entrées de test, se pose pour tout type de système sous test. Les spécificités dans le cadre des systèmes autonomes sont détaillées dans la suite de ce manuscrit, mais il est nécessaire au préalable de donner une définition de ces systèmes.

1.3 Systèmes autonomes

Les systèmes autonomes ne sont pas tous robotiques (par exemple les voitures autonomes). Par ailleurs la robotique ne se limite pas aux systèmes autonomes ; en effet la robotique industrielle relève plus souvent de l'automatisme que de l'autonomie (voir table 1.1). Nos travaux portent plus particulièrement sur des systèmes autonomes robotiques.

1.3.1 Notion d'autonomie

L'autonomie, du grec αὐτο qui signifie « soi-même » et νόμος qui signifie « loi », est le fait de se gouverner par ses propres lois, de choisir de son propre chef. Une définition possible de l'autonomie des systèmes est proposée dans [Huang 2008] :

> « 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 ou par un autre système avec lequel le système communique. »

Le même auteur dans [Huang 2007] propose une quantification de l'autonomie à l'aide de trois axes :

— l'indépendance vis-à-vis de l'humain ;

— la complexité de la mission ;

— la complexité de l'environnement.

Ici pas de distinction franche et binaire entre l'automatisme et l'autonomie, mais une échelle continue qui va de l'automate entièrement contrôlé ou pré-programmé par l'humain, au système pleinement autonome qui peut effectuer des missions complexes sans supervision dans un environnement qui lui est hostile. La feuille de route de l'agence européenne pour la recherche en robotique [SPARC 2016] propose 11 niveaux d'autonomie pour les robots. Il est précisé que les facteurs environnementaux, le coût d'une mauvaise décision, le temps durant lequel le robot doit être autonome, ainsi que l'amplitude des décisions qu'il peut prendre, influent sur l'attribution des niveaux d'autonomie d'un système pour une tâche donnée.

De plus un système autonome est pourvu de capteurs afin de percevoir son environnement et d'actionneurs pour agir sur son environnement. La capacité à prendre des décisions, mise en œuvre par des mécanismes décisionnels développés dans le domaine de l'intelligence artificielle, et la nécessité de prendre en compte les incertitudes et l'évolution de l'environnement dans lequel le système accomplit ses missions sont deux aspects fondamentaux des systèmes autonomes. Le niveau d'autonomie d'un système dépendra de ses capacités d'analyse des données obtenues à l'aide de ses capteurs, ainsi que de ses capacités de planification et de décision de ses prochaines actions.

1.3.2 Architecture

Plusieurs types d'architectures sont proposées pour la conception de systèmes autonomes [Kortenkamp 2008] [Ingrand 2014]. Sont distingués conceptuellement, dans l'architecture hiérarchisée, trois niveaux comme illustré sur la figure 1.1 :

— *le niveau décisionnel* reçoit les objectifs, génère des plans afin de les transmettre aux niveaux inférieurs et supervise leur exécution en traitant les problèmes et erreurs soulevés par les niveaux inférieurs. Il raisonne à partir d'une représentation abstraite du système et son environnement ;

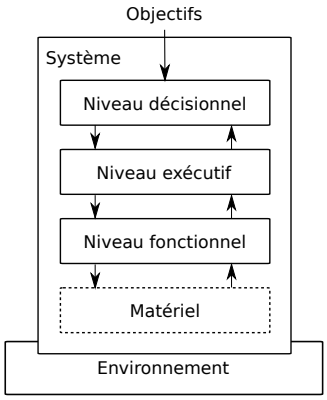

FIGURE 1.1 – Architecture hiérarchisée.

— *le niveau exécutif* convertit les plans qu'il reçoit du niveau décisionnel en actions élémentaires du niveau fonctionnel dont il va superviser et coordonner l'exécution ;

— *le niveau fonctionnel* sert d'interface entre la couche matérielle du système et les niveaux supérieurs. Il contient les fonctions basiques d'action et de perception et est décomposé en fonctions élémentaires qui partagent des données entre elles mais ne partagent pas de représentation globale du système.

Un exemple d'implémentation est donné dans [Ingrand 2007]. Dans cet exemple, le niveau exécutif a un rôle clair mais est intégré dans la couche fonctionnelle. Notons que la figure 1.1 donne une vue conceptuelle de l'architecture hiérarchisée, mais que son implémentation peut être modulée en fonction de besoins spécifiques.

Pour contrôler la complexité inhérente d'un système dans lequel sont embarqués un grand nombre de logiciels, l'approche par composants est très utilisée (voir par exemple [Brugali 2010]). Cette approche permet de réutiliser des composants d'un système à un autre et, par là même, facilite la construction de tels systèmes. L'architecture est donc constituée de divers composants logiciels qui sont par nature hétérogènes, que ce soit du point de vue :

— de la fonction à accomplir ;

— des contraintes de temps associées à celle-ci, classiquement les composants de la couche fonctionnelle doivent être plus réactifs que les composants de la couche décisionnelle qui eux sont plus gourmands en temps ;

— de la représentation de l'environnement sur lequel ils raisonnent.

Les communications entre les composants logiciels sont assurées par des intergiciels. ROS (*Robot Operating System*, [Quigley 2009]) est l'intergiciel le plus connu et utilisé mais il en existe plusieurs dont Yarp (*Yet Another Robot Platform*, [Metta 2006]) et Pocolibs (*POsix COmmunication LIbrary* [1]). Une vue générale ainsi qu'une comparaison des intergiciels les plus utilisés sont données

1. `https://www.openrobots.org/wiki/pocolibs`

dans [Einhorn 2012], qui propose également un nouvel intergiciel MIRA (*Middleware for Robotic Applications*).

Face à la très grande diversité des intergiciels, un outil comme GenoM (*Generator of Modules* [Mallet 2010]) sépare le côté algorithmique de la communication. Cette séparation permet d'avoir des modules réutilisables indépendamment de l'intergiciel considéré. GenoM est un outil de développement qui encapsule des algorithmes dans des modules standardisés. Le développeur doit d'une part détailler les modules dans un fichier de description (.gen) – notamment l'interface de ceux-ci, c'est-à-dire leurs services et ports de données (entrée ou sortie) – et d'autre part fournir les algorithmes à encapsuler écrits dans un langage de programmation (en C par exemple). Ces derniers sont appelés *codels*. GenoM s'occupera de générer la partie communication en fonction du middleware considéré.

1.3.3 Navigation

La navigation est un exemple de service de base d'un robot autonome mobile : il doit être capable d'évoluer dans des environnements inconnus et non structurés. Elle nécessite que le robot construise une représentation adéquate de son environnement, se localise et planifie une trajectoire dans celle-ci. La navigation est considérée conceptuellement comme faisant partie de la couche fonctionnelle. Cette dernière peut contenir plusieurs modes de navigation en fonction de l'environnement auquel est confronté le système autonome (environnement d'intérieur ou d'extérieur, terrain plat ou accidenté), mais aussi en fonction du type de mission (trajet longue distance, mode d'atteinte de cible, etc.). La tâche de navigation concerne la perception de l'environnement et la décision des mouvements d'un système autonome, que nous dissocions de la locomotion qui concerne l'exécution du mouvement. Le problème de la planification du système autonome se décompose en deux sous-problèmes : la planification globale et la planification locale. La planification globale concerne la planification du meilleur chemin possible entre deux points, alors que la planification locale sera utilisée afin d'éviter les obstacles et de manœuvrer, à l'extérieur, sur des terrains accidentés. La manière dont le système autonome va planifier sa trajectoire est liée à sa perception du monde via ses capteurs mais aussi à la nature de la représentation qu'il a de son environnement [Peynot 2006].

Nous détaillons ci-dessous des exemples de formes de représentation de l'environnement, puis des exemples de techniques de localisation et de planification de trajectoire dans cet environnement.

1.3.3.1 Représentation de l'environnement

Afin de représenter son environnement, un robot peut utiliser une carte d'occupation. Cette dernière est une représentation binaire de l'environnement dans laquelle chaque cellule est étiquetée en deux catégories : *traversable* ou *non traversable*. Le modèle qualitatif de l'espace libre est une extension de la carte d'occupation dans laquelle les cellules ne sont plus étiquetées binairement mais de manière

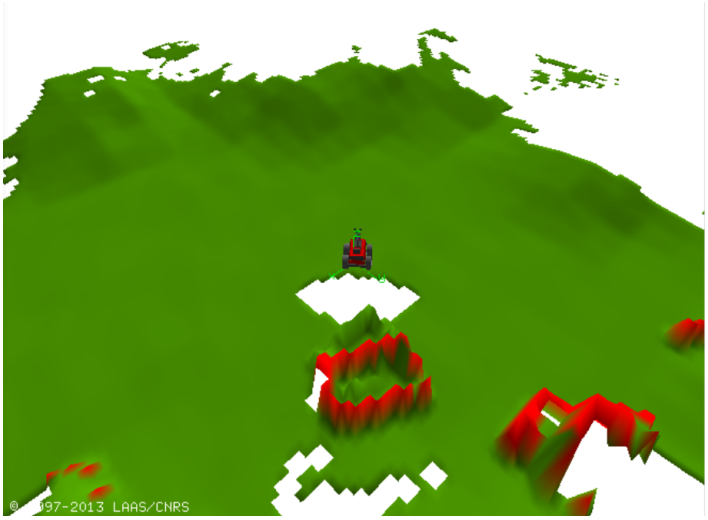

FIGURE 1.2 – Exemple d'un modèle numérique de terrain obtenu avec le module GenoM DTM.

continue à l'aide d'un nombre – probabilité, nombre flou ou encore estimation de rugosité – qui décrit si la cellule d'un terrain est plus ou moins traversable.

Le modèle numérique de terrain est une représentation plus adaptée aux terrains accidentés – typiquement les sols non plats dans des environnements extérieurs – que celles citées au-dessus. Le terrain est représenté sous forme d'un ensemble d'élévations calculées au fur et à mesure de la navigation du robot et sur un repère cartésien. Cette représentation est obtenue à l'aide de la fusion des diverses données en trois dimensions (3D) datées et localisées. Celles-ci peuvent être obtenues à l'aide d'un banc de stéréo-vision, d'une caméra de profondeur ou bien par nappe laser 3D qui bascule afin d'avoir une représentation de l'environnement tout autour du robot (voir figure 1.2). Une description de ces représentations est consultable dans [Lacroix 2002].

1.3.3.2 Localisation

La capacité pour le robot de se localiser dans son environnement est essentielle à la tâche de navigation. La localisation est utilisée pour la construction des représentations internes de l'environnement du robot, et aussi pour la planification locale et globale du robot dans cet environnement. Diverses techniques existent (l'odométrie en intérieur, l'estimation visuelle, la localisation basée sur des repères, le GPS, etc.). Les auteurs de [Ingrand 2007] précisent que l'utilisation d'une seule de ces techniques n'est pas suffisante pour localiser correctement un robot. Il faut plutôt préférer un ensemble complémentaire de celles-ci, puis agréger ces diverses estimations de positions afin d'assurer une meilleure localisation.

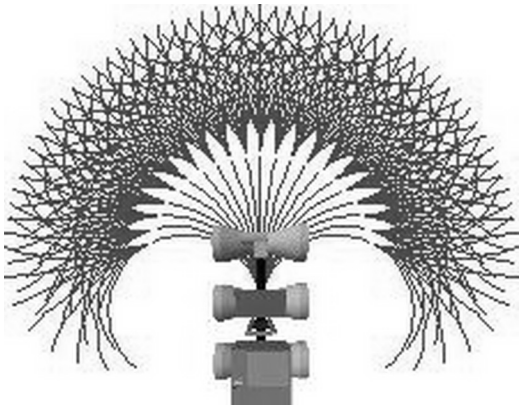

FIGURE 1.3 – Exemple d'arcs de P3D en face du robot *Lama* (source : [Lacroix 2002]).

1.3.3.3 Planification

De nombreux travaux, notamment au LAAS-CNRS, traitent du domaine de la planification locale. Parmi eux l'algorithme présenté dans [Bonnafous 2001] et intégré dans un module GenoM nommé `P3D`. P3D est une implémentation académique qui se base sur les mêmes principes que l'algorithme GESTALT [Biesiadecki 2006] utilisé lors d'explorations martiennes. Le principe général est de choisir le chemin qui optimise un critère *risque/intérêt* calculé le long d'*arcs* générés à partir de la position du robot dans son modèle numérique de terrain (voir figure 1.3). De manière plus détaillée, chaque arc est discrétisé en plusieurs *nœuds*. Ceci permet d'évaluer la valeur du risque lié à la configuration qu'aurait le robot s'il était positionné à cet endroit. L'intérêt dépend du rapprochement vis-à-vis du but. P3D calcule alors le coût lié à l'enchaînement de ces diverses configurations successives et ainsi donne la valeur du critère *risque/intérêt* de l'arc. Le coût est infini si le terrain n'est pas perçu, ce qui est classique des algorithmes de planification locale en environnement extérieur. Cet algorithme raisonne sur un modèle numérique du terrain obtenu à l'aide d'un capteur permettant d'évaluer la surface en trois dimensions. Or ce modèle numérique de terrain est mis à jour depuis la position du robot à une fréquence fixe. Ici deux problèmes sont rencontrés : les occlusions causées par des irrégularités du terrain, ainsi que la baisse de la résolution en fonction de la distance au capteur du robot. Ces problèmes sont visibles sur la figure 1.4 et vont affecter la planification locale. Les occlusions causées par les irrégularités locales sont autant de zones non perçues par le robot (zones blanches sur la représentation que le robot a de son environnement). Pour des raisons de sûreté, le robot ne planifiera pas de trajectoire passant par une zone non perçue (il pourrait en effet s'agir d'un trou).

Il est important de noter que ce type d'algorithme est sensible aux variations dans la perception mais aussi au niveau des ressources disponibles pour l'exécution des tâches du contrôleur. Ainsi, pour une même configuration, deux exécutions

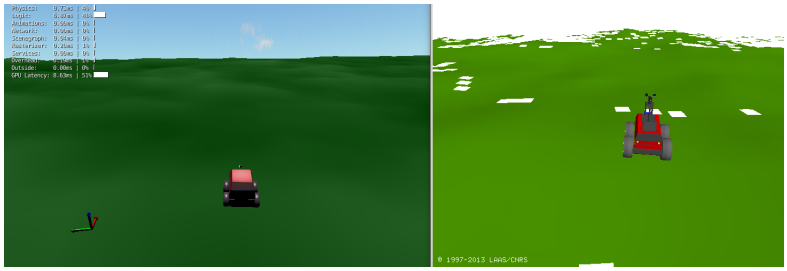

FIGURE 1.4 – A gauche le robot dans le monde simulé par MORSE, à droite la représentation que le robot a de son environnement. Les zones blanches correspondent à des parties non perçues.

en monde réel sont bien souvent différentes, bien que l'algorithme lui même soit déterministe. Dans ce manuscrit, cette caractéristique est mentionnée comme l'indéterminisme de la navigation.

1.3.4 Fautes dans les systèmes autonomes

Un système autonome qui évolue dans un environnement non structuré, ou comprenant des humains, est particulièrement soumis aux fautes externes, également appelées situations adverses ; ces fautes venant s'additionner aux fautes internes de conception. Comme vu section 1.2 une faute, une fois activée, produit une erreur qui peut se propager dans le système et provoquer une défaillance du service.

Avec le déploiement de systèmes autonomes au contact des humains, ces fautes peuvent avoir des conséquences graves. En 2016 en Chine, *Little Chubby*, un robot éducatif, blesse un enfant lors de la foire Hi-Tech. La même année, un robot K5 destiné à la sécurité, renverse un enfant dans un centre commercial aux États-Unis, ne lui laissant heureusement que quelques égratignures. Un an plus tard, le même modèle chute dans une fontaine. Un autre exemple catastrophique est donné par l'accident fatal impliquant une voiture autonome de la marque Tesla en 2016 aux États-Unis. Les causes de ces accidents sont diverses : du problème de la perception pour l'incident de 2016 impliquant le robot K5, au problème couplant la perception et la faute humaine pour l'accident de la voiture autonome, à un incident de communication avec l'interface du robot pour *Little Chubby*. Ces problèmes renvoient aussi à la robustesse, qui est la capacité pour un système de traiter correctement une situation inattendue (par exemple un humain de petite taille dans le cas du robot K5, ou une remorque réfléchissante pour le cas Tesla).

Dans [Carlson 2003] les auteurs ont collecté un total de 97 défaillances sur un ensemble de 13 robots mobiles de 7 modèles différents sur une période de 2 ans, dans des environnements intérieurs et extérieurs. Parmi ces 13 robots, 9 sont des robots industriels commercialisés ; 4 autres étant destinés à la recherche. Les auteurs comptabilisent une défaillance si le robot, ou l'équipement utilisé avec le robot, est dans l'incapacité fonctionner correctement. Le niveau d'autonomie des systèmes considé-

rés n'est pas clairement donné, mais il semble qu'il s'agisse de robots télécommandés avec peu d'autonomie, ce qui explique la définition de défaillance utilisée. Il n'est donc traité que des cas d'arrêt complet du robot ou de dégradation importante dans l'accomplissement du service, et non de problèmes liés à la performance par exemple. Un temps moyen avant défaillance (*Mean Time Before Failure* (MTBF) en anglais) de 8.3 heures en moyenne est obtenu, ce qui implique une confiance assez faible en ces systèmes. Cette mesure est calculée à l'aide de la formule suivante :

$$MTBF = \frac{Nombre\ d'heures\ d'utilisation\ du\ robot}{Nombre\ de\ défaillances}$$

Les auteurs ne donnent pas de description exhaustive des défaillances et fautes, en effet cette étude concerne surtout l'aspect quantitatif des défaillances. Malgré tout, sont évoquées des fautes matérielles (court-circuit de la batterie, chaleur trop importante qui voile une roue) et logicielles (blocage du système de contrôle, perte de la connexion avec une caméra). Il est de plus indiqué que la plupart des défaillances ont pour source une faute venant des actionneurs et du système de contrôle (qui comprend ici l'ordinateur embarqué et ses logiciels ainsi que la station de contrôle utilisée par l'opérateur). Ils observent aussi que les systèmes qui exhibent le plus de défaillances sont ceux qui naviguent dans des environnements extérieurs, bien que ces résultats soient à mitiger par le fait que ces robots soient plus récents et donc moins souvent déployés et testés.

Un an plus tard, dans [Carlson 2004], les mêmes auteurs proposent une taxonomie d'un nouveau corpus de 172 défaillances et observent une augmentation significative du temps moyen avant défaillance qui est maintenant de 24 heures en moyenne. L'étude concerne 15 robots mobiles de 7 modèles différents sur 1 an, dans des environnements intérieurs et extérieurs. Une défaillance est classée par rapport à l'origine de sa faute et la taxonomie de [Laprie 1996] est enrichie à l'aide des catégories définies dans leur précédente étude [Carlson 2003] – actionneurs, capteurs, système de contrôle, alimentation et communication – ainsi que deux autres catégories caractérisant les défaillances dont les fautes viennent d'interactions homme-machine. Dans leur taxonomie, une faute d'interaction homme-machine peut avoir pour origine une mauvaise manipulation de la part de l'humain, qui a mal apprécié une situation par exemple, ou à un mauvais comportement de la machine par rapport à la commande effectuée par l'humain. Cette classification leur a permis de calculer la probabilité de défaillance pour chaque catégorie, menant d'une part à une mesure quantitative plus précise que l'étude précédente, et d'autre part à établir une probabilité de défaillance par types de fautes. Ils réaffirment ici que les robots déployés à l'extérieur défaillent plus que ceux à l'intérieur, et que le système de contrôle et les actionneurs sont les principales sources de fautes.

Dans [Tomatis 2003], les auteurs étudient le déploiement de 11 robots *RoboX* dont la mission est de guider des humains lors d'une exposition. L'étude s'étend sur 5 mois, soit 13 313 heures de temps opérationnel. Une quantité importante de défaillances observées dans cette étude – 96 % des défaillances critiques, c'est-à-dire

celles qui ont nécessité l'arrêt du robot ainsi qu'une intervention humaine – ont pour source une faute logicielle. Le système de navigation est incriminé dans 17 % de ces défaillances critiques du logiciel, contre 79 % pour le logiciel utilisé pour l'interaction homme machine. Le fait que ce dernier soit encore en phase d'essai au début de l'exposition explique ce dernier résultat.

Dans [Steinbauer 2013], sont présentés les résultats d'une étude réalisée dans le cadre de la compétition *RoboCup*. Les auteurs impliquent les participants à la compétition, qui sont invités à répondre à un questionnaire. Des informations concernant les robots développés par 17 équipes différentes ayant participé à la compétition entre sa création et novembre 2010 sont ainsi récoltées. Les robots concernés sont des robots autonomes mobiles, déployés dans une des trois épreuves majeures proposées dans *RoboCup* : un match de foot faisant s'affronter deux équipes de 6 robots, des missions de recherche de survivants à la suite d'événements catastrophiques ainsi que des missions d'aide à domicile. Les questions concernent les causes des fautes observées, les parties du système affectées par ces fautes, les défaillances causées ainsi que leur impact et fréquence. L'étude montre que les fautes logicielles sont plus fréquentes que les fautes matérielles.

Peu d'études sur les causes et les effets des fautes réelles affectant des systèmes autonomes sont disponibles. Elles permettraient pourtant d'améliorer les mécanismes de sûreté de fonctionnement (test, moniteur, etc.). Les études évoquées montrent que la confiance en ces systèmes autonomes doit être améliorée et que la partie logicielle est très souvent la source des défaillances. De plus, elles pointent sur le fait que les systèmes naviguant dans un environnement extérieur sont plus susceptibles de défaillir que ceux se déplaçant à l'intérieur d'un bâtiment. Le test est étudié, dans ce manuscrit, comme moyen d'améliorer la confiance en de tels systèmes.

1.4 Test des systèmes autonomes

La pratique la plus répandue consiste à effectuer quelques tests *ad hoc* en simulation, et de mettre l'accent sur des campagnes de test sur le terrain. Il n'existe pas actuellement de méthode reconnue pour le test intensif en simulation. Cependant, ce sujet suscite un intérêt croissant au sein de la communauté du test.

Lorsqu'il s'agit d'éliminer les fautes, les systèmes autonomes soulèvent des problèmes spécifiques évoqués dans [Tiwari 2003] [Pecheur 2000] [Menzies 2005] et détaillés dans cette section :

— sélection des entrées : le domaine d'entrée est particulièrement difficile à caractériser. Le système doit accomplir des missions dans une grande variété d'environnements, et il n'est pas facile les modéliser (section 1.4.1) et de les générer (section 1.4.2) ;

— oracle : on ne dispose généralement pas d'un comportement de référence détaillé à comparer à celui observé durant le test. Pour une mission arbitraire, déterminer la faisabilité de celle-ci et les décisions à prendre est problématique.

De plus, le système exhibe typiquement un comportement non déterministe d'une exécution à l'autre de la mission. Ces limitations sont étudiées dans la section 1.4.3.

1.4.1 Modèles des entrées de test

La littérature exhibe deux grandes approches de modélisation des entrées de test : une première est centrée sur les interactions avec des entités de l'environnement, et l'autre est centrée sur les caractéristiques de l'environnement dans lequel se déroule la mission.

Lorsque les interactions sont au centre de la modélisation, l'accent est mis sur l'identification des événements et actions correspondant à ces interactions. En pratique, des modèles comportementaux décrivant les protocoles d'interaction sont construits. Ces protocoles peuvent prendre simplement la forme d'une liste d'actions possibles. C'est l'approche adoptée dans [Araiza-Illan 2016b] et [Araiza-Illan 2015], où les entrées sont ici une séquence de fragments de code que doivent exécuter l'humain, l'environnement, les capteurs ainsi que les actionneurs. Le simulateur Gazebo [Koenig 2004] est ici utilisé afin de tester un transfert d'objet entre un humain et un torse et bras robotiques : sont simulés l'environnement complet, le robot ainsi qu'un humain.

Les systèmes de transitions étiquetés sont aussi utilisés, dans [Mühlbacher 2016], afin de modéliser et tester le comportement du dispositif de manutention de charge d'un robot industriel transporteur. Cette approche a permis de repérer des fautes qui n'avaient pas été détectées lors de tests manuels effectués au préalable.

Dans le cas général, le système interagit avec plusieurs entités de l'environnement. Aussi la modélisation structurelle de celles-ci s'ajoute au modèle comportemental. Afin de tester la robustesse d'un aspirateur robotique, les auteurs de [Micskei 2012] utilisent le langage UML (*Unified Modeling Language*) pour spécifier deux méta-modèles d'entrées qui représentent :

— le contexte, qui englobe les objets et leurs attributs ainsi que leurs événements dynamiques ;

— les scénarios, un scénario est exprimé en un diagramme de séquence qui décrit les conditions de son déclenchement ainsi que le comportement exigé par le système sous test.

Une autre approche mixte structure/comportement est proposée dans [Andrews 2016], où un modèle UML capture les aspects structurels des entités et un réseau de Petri les aspects dynamiques.

En conclusion, lorsque les interactions sont au centre de la modélisation, on va retrouver des problématiques classiques de test basé sur des modèles comportementaux, avec éventuellement une vue structurelle complémentaire pour représenter les entités impliquées.

Dans la deuxième grande approche de modélisation, les interactions avec l'environnement sont implicites, via la boucle perception/décision/action. L'accent est

mis ici sur les caractéristiques de l'environnement dans lequel cette boucle s'exécute. C'est la démarche utilisée dans [Zendel 2013] qui traite de la génération d'entrées de test dans le domaine de la vision par ordinateur. Les entrées sont des images avec des objets à percevoir, selon différentes dispositions spatiales et avec différents éclairages. Deux structures sont utilisées : un modèle de domaine qui décrit les objets possibles, leurs propriétés et les contraintes entre ceux-ci ainsi que les différents cas d'illumination de l'environnement, et un ensemble de criticités (effet miroir, texture uniforme, obstruction etc.). Ces criticités ont été préalablement identifiées grâce à une technique d'analyse de risque appelée *HAZOP* (*HAZard OPerability*), modifiée pour convenir au domaine de la vision par ordinateur [Zendel 2015].

La fonction de navigation se prête bien à une approche par modélisation de l'environnement, car la trajectoire se construit en fonction de la perception de l'environnement par le système. Un exemple est donné dans [Nguyen 2009], où les auteurs cherchent à valider un agent autonome de nettoyage. L'agent doit nettoyer une pièce en récoltant des déchets, qu'il doit amener dans une poubelle, le tout en évitant des obstacles et en maintenant sa batterie au-dessus d'un seuil fixé. Cet environnement est schématiquement considéré et modélisé comme une matrice, avec des cases vides et des cases occupées soit par un déchet, une poubelle ou encore une station de chargement. Le travail de [Arnold 2013] propose de tester la navigation d'un robot mobile en immergeant celle-ci dans un environnement en deux dimensions (2D), exhibant des obstacles fixes et mobiles. L'utilisation d'algorithmes de génération procédurale de mondes permettent de représenter l'environnement simplement par quelques variables. C'est une contribution capitale car elle établit la connexion entre le test et la génération procédurale de mondes, utilisée notamment dans le domaine du jeu vidéo [Togelius 2011], qui est une perspective très prometteuse.

De façon transverse aux deux grandes approches de modélisation, [Alexander 2015] a mis en avant la notion de *situation*, qui permet d'identifier des cas typiques et critiques en considérant les entités, leurs relations et interactions. La définition précise de la notion de situation n'est pas donnée, mais on pourrait considérer que les scénarios de [Micskei 2012] (centrés sur les événements d'interaction), ou les cas abstraits issus des modèles de domaine de [Zendel 2013] (centrés sur les caractéristiques de l'environnement à percevoir) correspondent à des situations. Dans [Zou 2014] les auteurs identifient un ensemble de situations pertinentes pour guider le test d'une fonction d'anti-collision de drones. D'autres travaux portent sur le même type de fonction. Un système similaire, dans [Murschitz 2016], doit dans un premier temps détecter la présence d'intrus dans son flux vidéo, puis donner des commandes afin d'éviter la collision. Les auteurs testent ici la fonction de détection. Afin de dériver le domaine d'entrée, ils spécifient, avec l'aide de groupes de testeurs et de développeurs, l'objectif du système et son domaine. Une discussion sur la définition d'un domaine d'entrée est donnée. Les auteurs reconnaissent que celui-ci est par essence moins riche que le monde réel dans lequel le robot va opérer, mais il permet de savoir exactement par rapport à quoi le système est testé et donc de connaître les limites de la campagne de test effectuée. Le système sous

test peut se trouver entre 200 et 2000 m d'altitude, sa vitesse est comprise entre 100 et 200 $km.h^{-1}$, la vue est dégagée, le ciel peut-être encombré de 0 à 3 autres appareils qui sont choisis dans une liste d'objets volants prédéfinie, l'horizon du ciel peut-être de divers types, eux aussi choisis dans une liste, tout comme le sol en dessous du système sous test. Les auteurs définissent aussi 6 types de scénarios pour guider les tests. Ceux-ci décrivent divers types de rencontres aériennes. Ces scénarios permettent de contraindre encore plus la génération d'entrées, en excluant les situations inutiles du point de vue test – typiquement celles où au moins deux appareils sont présents mais ne se croisent pas.

1.4.2 Procédés de génération des entrées de test

La génération de tests basée sur des modèles comportementaux est très étudiée [Utting 2010]. Certaines des approches centrées sur la modélisation des interactions mettent en œuvre des algorithmes typiques de ce domaine. Par exemple dans [Andrews 2016], les entrées abstraites de test sont générées à partir de modèles UML et de réseaux de Petri en choisissant des chemins dans ces modèles. Ces chemins sont choisis afin de satisfaire des critères de couverture. Une démarche similaire est déployée dans [Mühlbacher 2016], mais ici les chemins sont choisis en utilisant un algorithme de marche aléatoire.

Dans la grande majorité des travaux cités dans la section 1.4.1, la génération de tests a recours à des procédés aléatoires, ou à des approches sophistiquées incorporant de l'aléatoire tels que les méta-heuristiques. Dans [Araiza-Illan 2015] et [Araiza-Illan 2016b] trois critères de couverture sont utilisés afin de générer des tests pour tester le passage d'objet entre un humain et le robot BERT2 : critères de couverture de code, de situation et de propriété. Ces critères permettent de guider la génération de tests et de mesurer l'efficacité de plusieurs méthodes de génération que les auteurs jugent complémentaires :

— non contrainte, c'est-à-dire aléatoire ;

— contrainte, ici la génération sera guidée afin de vérifier une propriété particulière ;

— basée sur un modèle des actions du robot et de son environnement. Ici les auteurs utilisent des automates probabilistes temporisés afin de capturer des actions incertaines telles que l'annulation d'une tâche.

La génération d'entrées de test se fait en deux temps, d'abord une entrée abstraite est générée à partir d'une liste d'actions possibles – c'est-à-dire ici une séquence de fragments de code que doivent exécuter l'humain, l'environnement, les capteurs ainsi que les actionneurs – qui va ensuite être concrétisée – c'est-à-dire que la séquence va être traduite en actions paramétrées qui devront être effectuée par l'humain (par exemple une séquence vocale pré-enregistrée). Ce travail est prolongé dans [Araiza-Illan 2016a] où les auteurs reprennent le même cas d'étude en plus d'un cas de robot d'aide à la personne et comparent deux manières de générer des entrées : une basée sur des modèles et l'autre en utilisant des agents Croyance-Désir-

Intention (*Belief-Desire-Intention* en anglais). Ces derniers raisonnent en termes de croyances (connaissances à propos du monde et des agents), de désirs (objectifs à remplir) et d'intentions (plans à exécuter) pour combler les désirs en fonction des croyances. Les auteurs montrent l'utilité et la supériorité, pour leurs deux cas d'études, de l'utilisation de ces agents et la simplicité d'utilisation de ceux-ci dans le cas du test d'interaction homme-machine.

Dans [Zendel 2013], les entrées sont générées dans le but de couvrir une liste de criticités ainsi que des classes d'entrée. Pour ce faire, les auteurs échantillonnent l'espace des possibles puis utilisent des techniques de *clustering* afin de supprimer les environnements trop similaires. Une démarche comparable est déployée dans [Murschitz 2016].

L'utilisation d'algorithmes de génération procédurale dans [Arnold 2013] est une solution pseudo aléatoire adaptée à la génération d'entrées de test tel qu'un environnement 2D complet. En effet, lorsqu'il s'agit d'un environnement détaillé – typiquement un environnement d'extérieur – la génération procédurale permet de ne pas s'en occuper manuellement (en plus d'économiser de l'espace de stockage).

Plusieurs travaux ont exploré l'utilisation de méta-heuristiques pour exhiber des tests qui instancient des scénarios à couvrir, ou qui induisent des comportements dangereux (typiquement, des collisions). Dans le premier cas, on trouve les travaux de [Micskei 2012], où la fonction de fitness dépend de la couverture des méta-modèles de contexte et de scénarios. Le deuxième cas correspond à la recherche de tests stressants non connus *a priori*, et la fonction fitness va mesurer la dangerosité du comportement observé sous test.

Pour ce deuxième cas d'utilisation des méta-heuristiques, on peut notamment citer les travaux précurseurs de [Wegener 2004], portant sur un système autonome qui effectue des créneaux pour automobiles. Une comparaison de deux fonctions de fitness est proposée pour guider la recherche de collisions. Une première stratégie consiste à prendre en compte la distance entre le véhicule et la zone de collision, l'autre l'aire entre le véhicule et cette même zone. La deuxième stratégie s'avère plus efficace à guider le test que la première, menant plus rapidement à des situations dans lesquelles le système de créneau mène le véhicule à une collision.

Plus récemment, les travaux de [Zou 2014] s'intéressent à un algorithme d'évitement de collision pour des drones. Les auteurs définissent diverses situations de collision ainsi qu'un algorithme génétique afin de guider le test. Les auteurs ne simulent pas l'ensemble de l'appareil, deux drones sont représentés par des points dans un environnement simulé en 2D, et le système sous test est l'algorithme d'évitement de collision. Les cas de test qui exhibent la plus importante proximité entre les deux drones sont sélectionnés. Leur approche est validée en la comparant à du test aléatoire, montrant qu'elle révèle des fautes que le test aléatoire met plus de temps à trouver. Les mêmes auteurs prolongent leurs travaux dans [Zou 2016] avec pour objectif de trouver des situations adverses pour un système d'évitement de collision plus complexe.

Une autre approche de test utilisant un algorithme génétique est développée dans [Nguyen 2009]. Dans un premier temps, un ensemble de propriétés vis-à-vis

du comportement du système sous test est défini : le robot doit maintenir une charge de batterie supérieure à 10 % et éviter les obstacles. Ces propriétés sont utilisées afin de définir la fonction objectif. Ici uniquement l'exigence d'évitement des obstacles est prise en compte : la fitness dérivée est basée sur distance minimale entre ces obstacles et le système sous test. Deux expériences sont présentées, la première permet d'exhiber deux fautes qui provoquent des collisions et la deuxième montre la supériorité de leur approche par rapport à une approche de test aléatoire. Un apport important de ces travaux est de prendre en compte le fait que lorsque l'on teste des systèmes autonomes, un même cas d'entrée peut provoquer des comportements différents de la part du système sous test (indéterminisme). Le nombre de fois que le test doit être rejoué afin de converger statistiquement sur une valeur donnée par la fonction objectif est calculé. A notre connaissance, ce sont les seuls travaux à considérer explicitement l'indéterminisme inhérent aux systèmes autonomes, qui impacte la mise en œuvre des méta-heuristiques.

Réduire le test à un problème d'optimisation est tentant, et ces travaux montrent la possibilité d'une telle réduction. Il est à noter toutefois qu'il s'agit là de problèmes exhibant un nombre relativement petit de paramètres et que ces approches nécessitent la manipulation d'un grand nombre de cas de test, soit autant de simulations – donc de temps de calcul. A notre connaissance une telle approche n'a jamais été tentée sur un monde et un système sous test complets – par exemple un environnement d'extérieur avec des objets statiques et dynamiques, ainsi qu'un système incluant tous les actionneurs et capteurs.

1.4.3 Oracle de test

Lorsque l'approche est centrée sur la modélisation des interactions, l'oracle est basé sur la vérification de la conformité du comportement observé du système sous test par rapport au comportement spécifié. Par exemple, [Mühlbacher 2016] vérifie la conformité par rapport à une spécification complète sous forme de systèmes de transitions étiquetés. Dans [Araiza-Illan 2015] et [Araiza-Illan 2016b], les propriétés à vérifier durant le test sont dérivées du standard ISO 13482:2014 et ISO 10218:2011 (par exemple : *le robot doit prendre une décision dans un temps imparti*, ou encore *si le robot se trouve à moins de 10 cm d'une personne, la vitesse de la pince du robot doit être inférieure à* 250 $mm.s^{-1}$). Celles-ci sont ensuite exprimées formellement en logique temporelle et un automate moniteur vérifie que le robot ne les viole pas.

Pour le test de la vision déployé dans [Murschitz 2016], un oracle spécifique est déployé. Afin de comparer la sortie du système sous test et la réalité terrain, la mesure de performance *Multi Object Maximum Overlap Matching* est utilisée.

Pour les autres travaux cités dans cette section, l'oracle ne vérifie que quelques propriétés de base, comme l'absence de collision. Cette notion d'oracle est très restrictive. Même si, dans le cas des systèmes autonomes, on ne peut exiger d'avoir une spécification complète de référence, il serait souhaitable d'élargir le type des propriétés vérifiées pour détecter des comportements anormaux. Comment choisir ces propriétés reste un problème ouvert.

1.5 Mondes virtuels et simulateurs

Les travaux décrits dans la section précédente ont une vue très simplifiée de l'environnement, et l'exécution de tests ne fait pas appel à des moyens de simulation sophistiqués. Seul [Arnold 2013] considère l'immersion dans un monde virtuel complet 2D.

Pourtant, le domaine robotique dispose d'outils de simulation avancés, certains basés sur des moteurs de jeux vidéos. La section 1.5.1 traite des simulateurs qui permettent de plonger un système autonome dans un environnement complet. La génération de mondes est également un sujet très étudié et développé dans le cadre des jeux vidéos. Nous aborderons celle-ci dans la section 1.5.2

1.5.1 Simulateurs

Les simulateurs sont utilisés en robotique pour la conception ou lors de tests de mise au point préliminaires à des expérimentations sur le terrain. Pour la conception, la simulation permet dans la plupart des cas un prototypage rapide de robot grâce à des ensembles d'actionneurs et de capteurs déjà implémentés, mais aussi des avatars complets de robots (voir par exemple la bibliothèque de composants du simulateur MORSE [Echeverria 2011]). Un autre exemple est fourni par la bibliothèque communautaire de Gazebo [Koenig 2004].

Lorsqu'il s'agit de simuler un ou plusieurs robots, les simulateurs utilisent un moteur 3D pour plonger le ou les robots dans un monde en 3 dimensions. La complexité de ce monde, la rapidité de rendu ainsi que sa qualité dépendent du moteur utilisé. Il existe plusieurs moteurs 3D dont OGRE, utilisé dans le simulateur Gazebo, ou encore *Blender Game Engine*, utilisé par MORSE [Echeverria 2011]. L'utilisation du *Blender Game Engine* permet de créer des environnements réalistes, ce qui est nécessaire si l'on souhaite correctement simuler la vision robotique par exemple et appréciable dans le cas où l'on veut faire du test.

La plupart des simulateurs intègrent dans des environnements complets la gravité, les frottements et autres phénomènes physiques. Pour ce faire ils utilisent des moteurs physiques en temps réel tel que l'*Open Dynamic Engine* (ODE) [Smith 2005] ou le *Bullet Engine*[2]. La précision de la simulation des divers phénomènes physiques dépend du moteur choisi. Une évaluation de ceux-ci est proposée dans [Boeing 2007]. Par exemple *Graspit !* [Miller 2004] est spécialisé dans la simulation de mains robotiques. Il intègre donc un modèle physique complexe et évolué afin de correctement simuler les actions de saisie d'objet. Afin de simuler l'environnement d'un robot sous-marin ou marin, [Parenthoën 2004] intègre un modèle physique de mer qui permet d'obtenir en simulation des conditions proches de celles rencontrées en monde réel.

Certains simulateurs sont utilisés uniquement pour un système ou composant en particulier, c'est le cas du simulateur du robot *iCub* [Tikhanoff 2008] qui est spécialement développé dans le but de simuler l'humanoïde *iCub* et ses modules.

2. `http://bulletphysics.org/wordpress/`

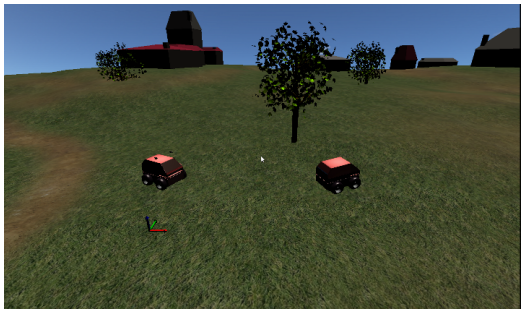

FIGURE 1.5 – Exemple de vue 3D avec MORSE.

C'est aussi le cas pour l'humanoïde du projet japonais *Humanoid Robotics Project* (HRP) [Kanehiro 2002] et pour le simulateur de *Molecube robots* un robot modulaire [Zykov 2009].

D'autres sont plus génériques et permettent de simuler divers robots (terrestres, aériens, marins/sous-marins, bras robotiques, humanoïdes etc.) dans leurs environnements respectifs comme c'est le cas pour MORSE, Gazebo ou encore Webots [Michel 1998], mais au prix d'une précision moindre.

La plupart des simulateurs cités précédemment évaluent les mêmes modules robotiques dans le monde simulé que ceux qui sont déployés dans le monde réel. Ce type de simulation s'appelle *Software-in-the-Loop* (SIL). Lorsque que la simulation interagit directement avec le système physique, on parle alors de simulation *Hardware-in-the-loop* (HIL). Par exemple, [Louis 2015] interagit avec le système physique d'un robot sous-marin.

L'immersion d'un système dans un environnement complet pose la question de la fabrication de cet environnement. Il est possible, et c'est souvent le cas, que cet environnement soit entièrement modélisé à la main. Lorsqu'il s'agit de faire du prototypage, une telle technique est suffisante. Mais lorsque qu'il est nécessaire de plonger le système dans un grand nombre d'environnements, il peut-être utile d'utiliser d'autres méthodes. Un parallèle est à faire avec le domaine du jeu vidéo, dans lequel peuvent être déployées les techniques de génération procédurales de monde [Peytavie 2010]. Celles-ci, utilisées initialement dans le jeu vidéo *Rogue* (ce qui a donné les *Rogue-like*) ou encore dans le célèbre jeu *Minecraft*, présentent un double avantage : d'une part elles permettent de réduire les coûts de la création d'un monde – que ce soit du point de vue informatique ou du travail humain – d'autre part elles peuvent permettre d'insérer une dimension aléatoire dans le contenu obtenu.

1.5.2 Génération procédurale de mondes

L'appellation « génération procédurale » englobe un nombre conséquent de techniques permettant de générer de manière automatique une partie ou la totalité d'un contenu, souvent d'un jeu vidéo ou d'une scène 2D/3D. Dans le jeu vidéo, ces techniques sont utilisées à plusieurs reprises pour générer des terrains, des cartes, des

niveaux, des arcs narratifs, des dialogues, des missions, des personnages, des règles du jeu, des dynamiques ainsi que des armes, ou encore des aspects plus décoratifs comme les textures, les éclairages ainsi que la conception sonore [Togelius 2011]. L'idée principale est d'avoir en amont quelques paramètres – description compacte du monde (ou génotype) – décrivant le monde qui permettront en aval de générer le contenu désiré – description étendue du monde (ou phénotype). Cette génération peut avoir lieu hors ligne ou en ligne, c'est-à-dire durant le développement ou durant le temps de jeu. La génération peut aussi être personnalisée et dépendre de la manière dont l'utilisateur joue [Yannakakis 2011].

Les auteurs de [Togelius 2011] font la distinction entre les techniques constructives et par essais successifs. Les techniques constructives produisent une solution de manière gloutonne en une seule passe ou avec très peu de retours en arrière. Cette catégorie comprend par exemple le classique bruit de Perlin [Perlin 1985], les fractales [Miller 1986], et les grammaires. Une vue de ces techniques est présentée dans [Shaker 2015].

Les techniques par essais successifs comprennent des approches simples, par exemple les algorithmes utilisés par le jeu *Dwarf Fortress* dans lequel diverses cartes sont générées avant d'être ou non gardées en fonction de leur capacité à se conformer au regard de critères, et comprennent des approches plus sophistiquées basées sur des méta-heuristiques. [Togelius 2010] présente une démarche de recherche dans les espaces d'état appliquée à la génération procédurale. Ici il ne s'agit plus de rejeter ou d'accepter une solution, mais d'adopter une approche de type évolutionniste afin d'optimiser une population de solutions potentielles à l'aide d'une fonction objectif à multi-objectifs.

1.6 Conclusion

Les fautes présentes dans les systèmes autonomes empêchent d'avoir une confiance en leur capacité à rendre des services. Peu d'études sur la nature de ces fautes, ainsi que sur les conditions et les effets de leur déclenchement sont disponibles. Le test fait partie des moyens d'obtenir une confiance justifiée dans un système, mais il soulève deux problèmes principaux : la sélection des entrées et l'oracle. Ces problèmes sont amplifiés par la nature des systèmes autonomes.

Les systèmes autonomes sont typiquement confrontés à un grand nombre de situations opérationnelles une fois déployés. Les défaillances peuvent survenir lors de situations complexes ou imprévues. Non seulement il est difficile de définir des critères de sélection propres à révéler des fautes, mais de plus la notion même de domaine d'entrée de test est problématique à définir. Dans certains cas, il est possible de focaliser sur des événements d'interaction avec l'environnement, et de se ramener à un cadre classique de test basé sur des modèles comportementaux. Mais dans les cas où les interactions avec l'environnement sont implicites via la boucle de perception/décision/action, l'espace des possibles est mal défini. Par exemple, pour tester une fonction de navigation, le domaine d'entrée est un ensemble hypothétique

de mondes dans lesquels le système est susceptible d'évoluer, ainsi que l'ensemble de toutes les missions de navigation possibles dans ces mondes. De même, tester une fonction de vision requiert de donner une définition constructive des scènes que le système sera amené à percevoir.

La définition d'un oracle de test est également un vrai défi. En effet bien souvent on ne peut connaître avec exactitude et à l'avance les actions que devra effectuer un système autonome face à un environnement et une mission donnés. En pratique, l'oracle se limite alors à la vérification de quelques propriétés critiques comme l'absence de collision, ce qui est insuffisant. De plus, les systèmes peuvent exhiber un comportement qui diffère d'une exécution à l'autre pour les mêmes entrées : cet indéterminisme est rarement pris en compte dans la littérature du test des systèmes autonomes.

Enfin, d'un point de vue technique, nous avons pu constater que le test des systèmes autonomes tire insuffisamment partie des moyens de simulation avancés utilisés pour le prototypage des robots. La pratique met plutôt l'accent sur les campagnes de test sur le terrain. Les travaux de recherche considèrent typiquement des simulations à un haut niveau d'abstraction, pouvant conduire à une simplification excessive de l'environnement de test. Notre point de vue est que l'immersion dans des mondes virtuels complets, permise par les technologies de simulation actuelles, constitue une piste intéressante pour la validation intensive des fonctions robotiques. Comme souligné par certains auteurs, la génération des entrées de test peut alors s'inspirer de techniques issues des jeux vidéo telles que la génération procédurale de mondes, avec son aspect aléatoire mais contrôlé. Notons d'ailleurs que certains simulateurs sont basés sur des moteurs de jeux.

Dans la suite de ce manuscrit, nous nous intéressons donc au test de robots dans des mondes virtuels, générés selon des procédés aléatoires, les expériences de test étant réalisées à l'aide de moyens de simulation avancés. La démarche suivie par nos travaux sera exploratoire, centrée sur des études de cas de complexité réaliste. Comme objet d'étude, nous choisissons un exemple de fonction typique des robots autonomes : la navigation en extérieur. La navigation permet d'illustrer la complexité des interactions implicites avec l'environnement via la boucle perception/décision/action. De plus, les environnements extérieurs sont par nature non structurés et plus complexes que ceux intérieurs. Deux études de cas seront successivement abordées. La première est la fonction de navigation d'un robot terrestre académique, que nous étudierons dans les chapitres 2 et 3. Les enseignements tirés de cette étude seront ensuite transférés à une deuxième étude de cas, un robot industriel agricole, permettant d'approfondir les solutions proposées (chapitre 4).

Chapitre 2

Premières expériences et étude des niveaux de difficulté

Sommaire

2.1 Introduction

Notre première étude de cas provient du robot Mana, développé au LAAS et servant de support aux recherches menées sur les robots terrestres en extérieur. Nous nous intéressons au test de la navigation de ce robot. Le robot est placé dans un environnement extérieur inconnu et reçoit les coordonnées d'un point objectif à atteindre. Il doit alors planifier et réaliser ses déplacements jusqu'à l'objectif, en s'aidant de fonctions pour cartographier l'environnement avoisinant et connaître sa localisation à chaque instant. Le simulateur choisi pour exécuter les tests est MORSE (Modular OpenRobots Simulation Engine) [Echeverria 2011], que nous avons brièvement mentionné dans notre discussion des simulateurs au chapitre précédent. MORSE est un simulateur robotique open-source basé sur le moteur de jeu Blender. Un récent article de synthèse [Cook 2014] le met en avant comme l'un des simulateurs les plus adéquats vis-à-vis de la navigation (un autre simulateur

adéquat étant Gazebo [Koenig 2004], que nous utiliserons dans une deuxième étude de cas au chapitre 4).

Nous concevons et réalisons une plateforme de test pour le logiciel de navigation de Mana, en simulation sous MORSE. Cette réalisation nous confronte à différents problèmes de conception du test évoqués au chapitre précédent, ainsi qu'à des problèmes techniques liés avec l'interfaçage avec des moyens de simulation avancés. Nous sommes ainsi amenés à considérer les problèmes suivants :

— la définition du domaine d'entrée de test, qui implique de préciser l'espace hypothétique des mondes dans lequel Mana peut être déployé ;

— la génération procédurale de mondes dans cet espace, en cherchant à exploiter des fonctionnalités disponibles sous Blender ;

— la collecte de traces de test pendant les simulations ;

— l'analyse des données collectées, qui inclut la procédure d'oracle, ainsi que d'autres types d'analyse que nous pourrons juger utiles.

L'objectif de l'exercice est de mieux cerner ces problèmes, de proposer des premières solutions concrètes, et de tirer des enseignements pour des développements futurs.

Le principe de la génération procédurale de mondes, étudiée ici à travers l'exemple de la navigation du robot Mana, est de générer du contenu à partir de quelques paramètres haut niveau en incorporant de l'aléatoire. L'utilisation d'une telle approche pose la question du calibrage des paramètres de génération et de la contrôlabilité offerte vis-à-vis d'objectifs de test. Nous abordons ces questions sous l'angle du contrôle de la *difficulté* des mondes vis-à-vis de missions de navigation. Lors de la génération de test, on peut souhaiter rester dans une enveloppe de fonctionnement du robot et éviter les mondes trop difficiles dans lesquels le robot a du mal à se mouvoir. On peut au contraire vouloir mettre l'accent sur les cas difficiles, dans un objectif de test de robustesse. Ou encore, on peut vouloir échantillonner des cas de test correspondant à des niveaux de difficulté divers, en commençant par des cas simples pour aller progressivement vers des cas de plus en plus difficiles. Ces différentes stratégies de sélection supposent de pouvoir contrôler le niveau de difficulté en ajustant les valeurs des paramètres de génération.

Associer un niveau de difficulté à des valeurs de paramètres de génération n'est pas trivial. Pour cela nous proposons une approche expérimentale pour le réglage des paramètres, que nous mettons en œuvre dans la plateforme de test. Du fait de la part aléatoire de la procédure de génération, ainsi que de l'indéterminisme du comportement du robot, l'étude de chaque configuration de paramètres nécessite plusieurs phases de simulation. Une mesure de la difficulté est proposée, à partir de la faisabilité observée des missions et de l'effort demandé afin de les mener à bien. Nous classons ensuite les configurations de paramètres en niveaux de difficulté, et observons les plages de variation des paramètres correspondant à ces niveaux. Nous évaluons également l'indéterminisme de la navigation et son évolution en fonction des niveaux de difficulté. Enfin, ces expériences sont l'occasion d'étudier une première faute. Il s'agit d'une faute de configuration présente dans une version prélimi-

naire de la plateforme, que nous avions rapidement corrigée. Nous ré-injectons cette faute pour observer l'effet des niveaux de difficulté sur le comportement erroné.

Ce chapitre est organisé de la manière suivante. Le système sous test est présenté section 2.2. Puis la conception de la plateforme de test, basée sur le simulateur MORSE est exposée section 2.3. Cette plateforme est utilisée pour aborder un ensemble de questions expérimentales concernant la notion de difficulté présentée dans la section 2.4. Les résultats sont présentés et discutés dans la section 2.5.

2.2 Modules robotiques sous test et simulés

Le logiciel du robot Mana est organisé conceptuellement suivant une architecture hiérarchisée en trois niveaux (voir les différentes architectures en section 1.3.2) : un niveau décisionnel, un niveau exécutif et un niveau fonctionnel. Nous nous concentrons ici sur le niveau fonctionnel des services de navigation. Dans le cas de Mana, il correspond à six modules robotiques. Trois d'entre eux correspondent au pilotage des capteurs et actionneurs du robot, et les trois autres implémentent les fonctions plus haut niveau de la navigation.

La figure 2.1 représente l'articulation entre ces modules, dans le cadre d'une plateforme de simulation MORSE représentative de ce que les développeurs utilisent à des fins de prototypage et de mise au point. Le pilotage des capteurs et actionneurs est géré par MORSE, en remplaçant les modules réels par des modules de simulation. Ainsi, les modules `Rflex` (contrôleur de roues et odométrie), `Velodyne` (capteur laser 3D) et `Pose` (capteur de position) sont simulés. Les autres modules fonctionnels sont exactement ceux déployés sur le robot réel et vont constituer notre système sous test (SUT, d'après l'abréviation anglaise). Il s'agit de : `DTM` (cartographie 3D), `POM` (agrégateur de position) et `P3D` (planification locale présentée section 1.3.3.3). Ces trois modules sont développés avec GenoM (pour une description de GenoM voir section 1.3.2). Ils communiquent à l'aide de l'intergiciel `pocolibs`, via des `posters` qui sont des espaces de mémoire partagée. Le système sous test, comportant ces trois modules et les diverses librairies dont ils ont besoin, représente environ 35 000 lignes de code.

Ce système est réaliste du point de vue de la complexité d'un service de navigation locale qui planifie une trajectoire pour atteindre un point objectif en évitant les éventuels obstacles, trous et terrains trop accidentés pour le robot. Pour rappel, le module P3D reprend les principes d'un algorithme développé par la NASA pour l'exploration martienne. Il considère un ensemble d'arcs projetés à partir de la position perçue du robot dans son modèle numérique de terrain et évalue ces arcs selon un critère risque/intérêt. L'algorithme est mature et ce service de navigation a été utilisé lors de nombreuses expériences sur terrain extérieur par des chercheurs en robotique du LAAS. Le système sous test est également réaliste du point de vue du caractère indéterministe de la navigation. L'algorithme de sélection d'arcs est déterministe, mais son intégration au sein d'une architecture fortement concurrente, typique des robots complexes, induit de l'indéterminisme à l'exécution. Tout chan-

gement temporel au niveau de l'exécution des processus de perception, décision et action peut entraîner des changements drastiques sur la trajectoire construite itérativement. Dans le monde réel, on a de plus des aspects incontrôlables de l'environnement (par exemple du vent dans la végétation, qui impacte la carte perçue). Ainsi, il n'est pas possible de reproduire à l'identique l'exécution d'une mission donnée, que ce soit dans le monde réel ou en simulation.

En plus des modules robotiques, deux fichiers principaux de configuration sont nécessaires dans la plateforme. Le premier spécifie les limites physiques du robot utilisé afin que P3D puisse calculer les trajectoires (`Robot model` sur la figure 2.1). Le deuxième est utilisé par MORSE (`Simulation config`), en spécifiant notamment l'avatar du robot, ses capteurs et en mettant à disposition le chemin où est stocké le monde virtuel à simuler (`world.blend`). L'avatar du robot Mana, ainsi que les modules de simulation des capteurs et actionneurs, correspondent à une approche de simulation *basse fidélité* du point de vue des aspects physiques. On ne simule pas les réactions entre les roues du robot et le terrain, l'inertie du robot ou encore des zones glissantes. L'avatar reçoit des consignes en vitesse de P3D et se déplace directement en fonction de celles-ci sans passer par le mouvement des roues. Le moteur de jeu de Blender, sur lequel est basé MORSE, peut proposer une simulation plus fine des interactions physiques, mais à un coût de calcul et de développement plus important des modules de simulation. En pratique, le prototypage de la navigation sous MORSE a adopté une approche basse fidélité, que nous conservons dans le cadre de notre étude sur le test. Nous reviendrons ultérieurement, dans le chapitre 3, sur l'impact de ce choix pour la reproductibilité des fautes.

Enfin, dans la plateforme, un script contrôle l'exécution de la simulation (*Execution controller* sur la figure 2.1). Il initialise, transmet la mission au système sous test et communique avec lui lors de la simulation.

Nous allons partir de cette plateforme de prototypage et l'enrichir pour construire une plateforme de test.

2.3 Plateforme de test

Pour identifier les extensions à apporter à la plateforme précédente, nous considérons une vue conceptuelle du test en simulation, donnée figure 2.2. Le test peut être découpé en plusieurs phases :

1. Génération : production des entrées dans un format d'échange directement compréhensible par le simulateur et le système sous test. La génération des entrées peut être guidée par un critère de test et peut également prendre en compte les résultats des tests précédents. Cette dernière possibilité induit une boucle de test, typique des approches telles que le test basé sur des métaheuristiques de recherche ;
2. Simulation : phase d'exécution des tests durant laquelle le système sous test va évoluer dans des environnements simulés. Durant cette phase, diverses données sont collectées ;

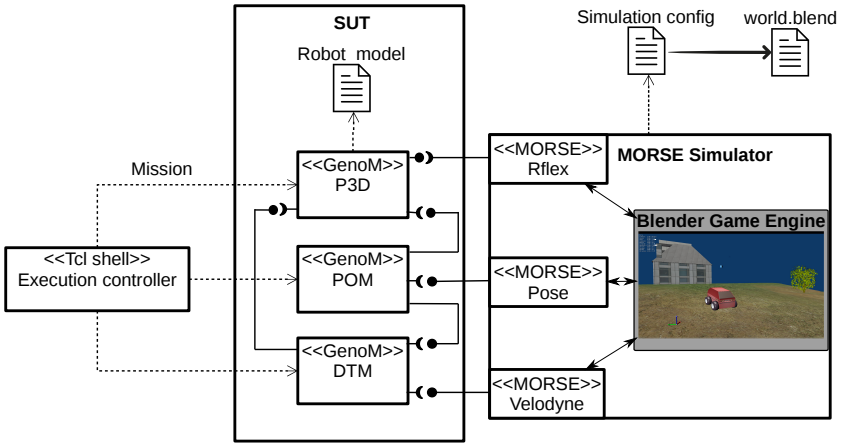

FIGURE 2.1 – Diagramme simplifié représentant l'articulation entre les modules robotiques sous test et ceux simulés par MORSE.

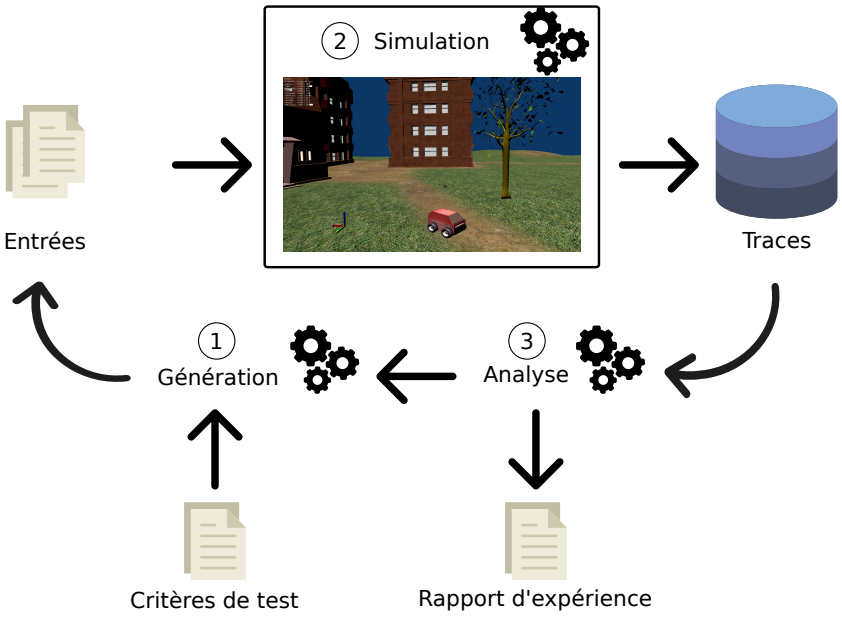

FIGURE 2.2 – Vue générale du test en simulation.

3. Analyse : les données collectées sont analysées afin de produire un rapport d'expérience ou éventuellement de guider la génération suivante.

La génération des entrées, la collecte de données en cours de simulation et l'analyse des résultats du test sont de nouvelles fonctionnalités à intégrer dans la plateforme de prototypage.

Pour la génération, nous supposons une stratégie simple de test aléatoire, basée sur un modèle de monde et implémentée à l'aide des fonctionnalités offertes sous Blender. Nous ne considérons donc pas le rebouclage des résultats de test sur la génération suivante. Nous mettons l'accent sur la définition du domaine d'entrée (décrite section 2.3.1) et sur la construction de mondes appartenant à ce domaine (décrite section 2.3.2).

Nous choisissons de séparer clairement la collecte de données brutes effectuée en ligne (voir section 2.3.3) et l'analyse réalisée hors ligne (décrite section 2.3.4). Cette séparation offre plus de flexibilité pour effectuer divers types d'analyse (par exemple élaboration d'un verdict, analyse de performance, de robustesse, aide au diagnostic, etc.) sans avoir à changer les instrumentations et à rejouer les tests. Par contre, l'enjeu est d'identifier un ensemble raisonnablement complet de données brutes à collecter, couvrant les besoins de types d'analyse potentiellement différents.

2.3.1 Définition du domaine d'entrée de test

Une première étape nécessaire à la génération automatique d'entrées est de modéliser le domaine d'entrée.

Le domaine d'entrée doit capturer le plus d'éléments significatifs de l'environnement réel dans lequel le système autonome est amené à opérer. Il borne et spécifie la portée du test en déterminant l'ensemble des éléments auquel pourra être confronté le système durant la campagne de test (il peut, par exemple, donner un nombre d'obstacles maximum). Pour toute campagne de test, la définition du domaine d'entrée est capitale. Si celle-ci est incomplète, il peut en résulter l'impossibilité ou la difficulté pour le test de déclencher certaines fautes du système sous test et donc de ne pas repérer les défaillances qui y sont liées. Pour autant, le domaine d'entrée ne peut représenter toutes les caractéristiques du monde réel et doit idéalement ne retenir que celles qui sont pertinentes.

Pour guider la définition du domaine d'entrée et limiter l'espace des possibles, nous proposons de partir de l'analyse de cas d'utilisation du robot. Ces cas servent de support pour extraire des informations pertinentes selon plusieurs aspects :

— l'environnement : dans quel environnement le robot va-t-il être déployé ?

— les capteurs : comment le robot perçoit-il son environnement ?

— les actionneurs : comment le robot va-t-il interagir avec son environnent ?

— la mission : que doit faire le robot dans son environnement ?

Comme exemple de cas d'utilisation, nous avons pu obtenir une image en trois dimensions représentant un environnement réel dans lequel le robot a été déployé à des fins d'expérimentation. La figure 2.4 en donne une représentation en deux

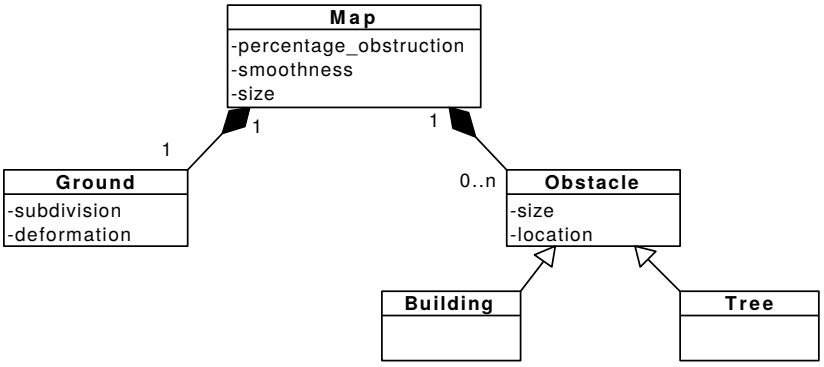

FIGURE 2.3 – Diagramme de classes UML représentant le modèle de monde virtuel dérivé du cas d'étude du robot Mana.

dimensions. Pour ce cas, on peut considérer qu'un monde est composé d'un terrain et de divers obstacles posés sur celui-ci. Ces derniers peuvent être des arbres ou des bâtiments. Le terrain n'exhibe pas une topologie accidentée (montagne, canyon, etc.) mais est plutôt lisse avec des irrégularités locales.

Le robot est équipé d'un capteur laser (Lidar 3D Velodyne) lui permettant de percevoir son environnement. Les caractéristiques telles que la couleur ou la texture ne sont pas perçues. Elles peuvent donc être ignorées dans notre modélisation des entrées, alors qu'elles seraient pertinentes si le robot se servait d'une caméra. Mana ne distingue pas non plus les routes et chemins et ne cherchera pas spécifiquement à les emprunter – son seul critère est l'optimisation des valeurs risque/intérêt attachées aux arcs. Nous ne considérons pas de tracé routier.

Il n'y a pas d'actionneur autre que ceux nécessaires à la locomotion. Les interactions possibles avec l'environnement sont uniquement : se déplacer sur le terrain et éventuellement entrer en collision avec un obstacle (interaction catastrophique). Si on se limite à des obstacles statiques, il n'y a pas de caractéristique particulière à considérer autre que la forme générale du terrain et des obstacles.

Une mission de navigation peut être décomposée en trois phases. Dans une première phase d'initialisation, le robot se déplace à l'aveugle sur un mètre en ligne droite depuis son point de départ pour commencer l'acquisition et la cartographie de son environnement. La deuxième phase est celle de la navigation en direction du point de destination prescrit, en évitant les obstacles et les parties de terrain trop pentues. Enfin la troisième phase est l'approche de la destination : le robot décélère pour s'arrêter sur le point de destination. Ainsi, une mission sera trivialement infaisable si la zone de départ est obstruée (empêchant l'initialisation de la cartographie) ou si le point de destination est obstrué (empêchant l'atteinte de ce point). Nous excluons ces cas du domaine d'entrée de test.

A partir de l'analyse ci-dessus, nous dérivons un modèle de monde présenté en figure 2.3. Ce modèle peut être enrichi en considérant des objets dynamiques (c'est-à-dire les obstacles mobiles), du bruit sur les capteurs du robot etc. Toutefois, avant

FIGURE 2.4 – Image d'un environnement réel dans lequel Mana a été déployé.

de considérer de telles extensions, il est utile d'étudier dans quelle mesure les éléments statiques impactent la navigation. Les diagrammes de classe UML capturent bien ces aspects statiques, c'est donc le langage choisi afin de modéliser la structure des mondes. On y retrouve les éléments identifiés sur le cas d'utilisation : un monde (`Map`) est composé d'un terrain et d'obstacles posés sur celui-ci, qui peuvent être des arbres ou des bâtiments. Le modèle est enrichi par des attributs et des contraintes sur leurs valeurs. Par exemple, la classe `Map` est caractérisée par une taille, un pourcentage d'obstruction et un degré de déformation (attribut `Map.smoothness`). Le pourcentage d'obstruction dépend du nombre d'obstacles présents et de leur taille, comparativement à la taille globale du monde.

En complément au modèle de monde, notre modèle de mission consiste en une simple paire de coordonnées correspondant respectivement au point de départ du robot et au point objectif à atteindre. Des contraintes sont ajoutées afin que ces deux points soient situés dans des zones libres d'obstacles. Comme expliqué plus haut, leur obstruction conduirait à des missions trivialement infaisables.

2.3.2 Génération des mondes et missions

Une fois la structure des mondes définie, notre approche est d'associer des fonctions de génération (c'est-à-dire des constructeurs incorporant de l'aléatoire) à chaque classe du modèle. Ainsi, la génération d'un monde va lancer la génération d'un terrain, puis celle d'un certain nombre d'obstacles, et enfin va poser ces obstacles sur le terrain selon des positions aléatoirement choisies. La figure 2.5 donne une vue d'ensemble des différentes étapes, que nous allons détailler. Notons que lors du choix des fonctions de génération, des attributs et contraintes peuvent être ajoutés au modèle. Par exemple, l'utilisation d'un algorithme de génération procédurale peut nécessiter l'introduction de paramètres particuliers à celui-ci. De façon

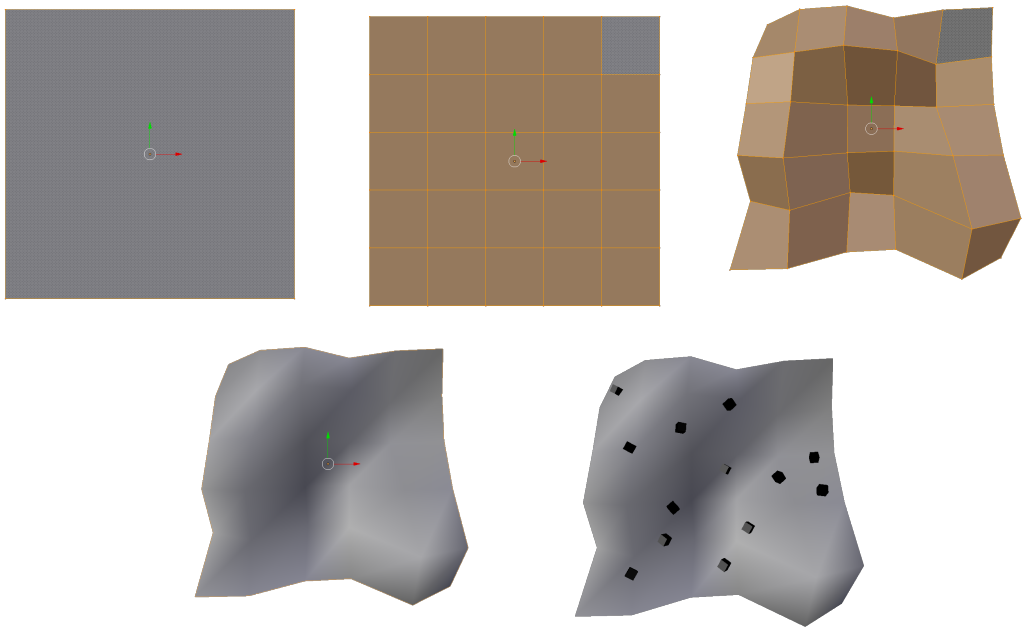

FIGURE 2.5 – Étapes de génération d'un monde peuplé d'arbre avec comme paramètres $subdivision = 4$, $deformation = 3$ et $percentage\_obstruction = 0.13$.

générale, nos fonctions de génération vont tirer partie des fonctionnalités offertes par Blender. Leur implémentation est réalisée par des scripts en langage Python, en utilisant l'API Blender. Elle exploite un ensemble d'objets 3D de base (appelés *mesh*) proposés nativement dans Blender, ainsi que des fonctions de manipulation et de transformation de ces objets. L'ensemble des objets de base peut être enrichi via des scripts, dont un certain nombre est proposé par la communauté d'utilisateurs de Blender.

La génération de la topologie du terrain part d'un mesh carré de longueur de côté `Map.size`. Nous lui appliquons d'abord une fonction du logiciel Blender permettant de subdiviser les faces du maillage d'un objet 3D afin d'obtenir une grille plus précise qui contrôle la géométrie de celui-ci. La fonction `fractal transformation` est ensuite utilisée afin de déplacer les sommets de cette grille dans des directions pseudo-aléatoires. Afin de contrôler l'action de ces deux étapes, nous avons besoin de paramètres s et d qui correspondent respectivement au nombre de subdivisions et à l'amplitude de la déformation de la fonction `fractal transformation`. Ces paramètres permettent d'obtenir des terrains plus ou moins irréguliers, déterminant ainsi la caractéristique globale de `smoothness` du modèle de monde. Dans notre cas, le déplacement des sommets le long de l'axe z est contraint afin d'obtenir un résultat moins chaotique et plus réaliste. La figure 2.5 montre le résultat de cette transformation appliquée à un mesh initialement plan. Ainsi déformé, et après lissage, ce mesh représente le terrain. Le choix d'une mission de navigation correspond alors à

la sélection de deux points dans ce terrain.

Le nombre d'obstacles à placer sur le terrain (cardinalité n sur la figure 2.3) est calculé en fonction du pourcentage d'obstruction souhaité et de la taille relative du monde et des obstacles. Par exemple, en supposant un seul type d'obstacle à générer occupant une surface au sol de $Obstacle.size^2$, le calcul s'effectue de la manière suivante :

$$n = round\left(\frac{Map.percentage\_obstruction \times Map.size^2}{Obstacle.size^2 \times 100}\right)$$

La fonction `round` permet de garder uniquement la partie entière et donc de ne considérer que des obstacles complets.

Pour la génération de chacun des n obstacles, nous avons d'abord cherché à produire des arbres et des bâtiments réalistes. Par exemple, des modèles d'arbres complexes, générés de manière procédurale à l'aide du script *Sapling Tree* [1] ont été intégrés à nos tests. Toutefois, nous avons rencontré des problèmes de performance sévères en simulation. En pratique, on constate que MORSE ne permet pas l'instanciation d'un grand nombre d'objets complexes (c'est-à-dire complexe du point de vue de la physique, ou encore du maillage de l'objet). Nous avons donc dû écarter ce type de solution. Cela n'a pas été trop gênant pour notre étude de cas car la télédétection par laser du robot Mana ne perçoit que la forme générale des obstacles qui l'entourent. Aussi, des formes rectangulaires et cubiques peuvent être utilisées pour représenter un bâtiment ou un arbre. Nous avons finalement opté pour une représentation schématique de ceux-ci sous forme de mesh cubiques.

Leur placement sur le terrain doit satisfaire des contraintes, ainsi les obstacles :

— doivent être entièrement placés à l'intérieur du monde virtuel, afin que le robot puisse potentiellement les percevoir et tenter de les éviter ;

— ne doivent pas se chevaucher entre eux, afin de calculer le pourcentage d'obstruction ;

— ne doivent pas occuper les zones autour du point de départ et d'arrivée de la mission, ainsi le robot peut s'initialiser correctement et nous ne générons pas de mission trivialement infaisable où le robot ne pourrait pas quitter son point de départ, ni occuper physiquement la place du point d'arrivée ;

— doivent reposer sur le sol en tenant compte des irrégularités (creux et bosses), cela permet d'avoir des obstacles correctement disposés sur le sol et ainsi d'obtenir un environnement réaliste.

Les trois premières contraintes conditionnent le choix aléatoire des coordonnées (x, y) d'un obstacle, alors que la dernière détermine sa hauteur z. Il est intéressant de discuter comment nous avons traité ce cas du placement sur un sol irrégulier. Dans un premier temps, il s'effectuait à l'initialisation de la simulation, en « lâchant » l'obstacle au dessus de la carte et en utilisant la gravité pour qu'il se place. Toutefois cette stratégie s'avère coûteuse en temps de calcul et, à partir d'un certain nombre

1. http ://wiki.blender.org/index.php/Extensions :2.6/Py/Scripts/Curve/Sapling_Tree

d'obstacles, la simulation devient impossible. En conséquence, une autre stratégie a été appliquée : aucune gravité n'est ajoutée aux obstacles et un calcul est fait par le script de génération afin de déterminer la hauteur des obstacles dans le monde virtuel en fonction de leur position le long des axes des abscisses et ordonnées.

Au final, nous disposons d'une fonction de génération de mondes permettant de s'interfacer avec MORSE. Ses paramètres de contrôle correspondent aux attributs de l'élément racine `Map` : la taille du monde, le pourcentage d'obstruction souhaité, et le degré de déformation du terrain (*smoothness*), défini par un couple de valeurs $(subdivision, deformation)$ à transmettre au constructeur de terrain. La génération de mondes prend ces paramètres en entrée, construit aléatoirement un monde et retourne un fichier de type `mon_monde.blend` dans un format directement compréhensible par le simulateur.

2.3.3 Collecte des données

Les mondes/missions générés constituent des cas de test, exécutés en simulation. Pour analyser le comportement sous test, des données d'observation doivent être collectées en cours d'exécution. Nous avons cherché à identifier un ensemble de données pertinentes pour un logiciel de navigation, et idéalement susceptibles de répondre aux besoins de différents types d'analyse.

Il nous paraît intéressant de distinguer deux catégories de données :

— prises du point de vue du robot, c'est-à-dire relatives à la perception qu'a le logiciel du déroulement de sa mission ;

— reflétant la vérité terrain, c'est-à-dire relatives au déroulement réel de la mission tel que perçu par un observateur externe.

L'appartenance à l'une ou l'autre des deux catégories influe sur la stratégie d'obtention de la donnée. Le point de vue du robot s'obtient par instrumentation du logiciel sous test, tandis que la vérité terrain nécessite des observateurs intégrés au simulateur. Pour un logiciel de navigation, les données pertinentes vont notamment inclure la trajectoire du robot, c'est-à-dire une succession de positions horodatées, perçues et réelles. Nous les complétons par des données relatives à des erreurs (ex : erreur interne, collision).

Les données que nous proposons de collecter du point de vue du robot sont les suivantes :

— les positions perçues. Chaque position est horodatée et inclut l'attitude complète du robot, c'est-à-dire les valeurs du lacet, du tangage et du roulis (correspondant aux angles d'Euler ψ, ϕ et θ) ainsi que ses coordonnées (x, y, z) relativement à un repère de référence ;

— les messages d'erreur produits par le logiciel de navigation, indiquant l'impossibilité perçue de poursuivre la mission.

— le dernier modèle numérique de terrain perçu par le robot à la fin d'une simulation, qui peut être utile pour comprendre les cas d'abandon de mission.

Ces données sont pertinentes pour tout logiciel de navigation, mais leur facilité d'accès va dépendre de l'implémentation testée. Pour le logiciel étudié, la collecte de données est grandement facilitée par les utilitaires de *log* nativement fournis par l'interface des modules GenoM. En particulier, tout module GenoM peut être configuré pour enregistrer les valeurs de ses `posters` (espaces de mémoire partagée permettant la communication entre les modules GenoM) à chaque mise à jour de ceux-ci ou bien à la demande. Ces facilités suffisent pour les données souhaitées et nous évitent d'avoir à instrumenter le code manuellement. Les positions perçues correspondent au `poster` du module POM, configuré pour un enregistrement à chaque mise à jour : elles sont donc capturées au taux de rafraîchissement de POM. Le modèle numérique de terrain correspond à un `poster` du module DTM, configuré pour un enregistrement à la demande.

Les données de vérité terrain que nous proposons de collecter sont les suivantes :

— les positions réelles. On conserve le même ensemble de données et le même format que pour les positions perçues (attitude et coordonnées horodatées) ;

— un événement de collision ;

— un événement d'expiration du temps alloué (*timeout*). En effet, si l'exécution d'une mission excède un certain temps, on considère qu'elle ne terminera pas et on force son arrêt.

Le système sous test n'est pas mis à contribution pour obtenir ces données : elles sont prises d'un point de vue externe à ce système. MORSE fournit des capteurs de position et de collision, qui permettent d'obtenir les informations nécessaires à l'élaboration de telles données. Ces informations sont publiées régulièrement sur des sockets. Pour les enregistrer, nous avons développé un composant *moniteur* comportant autant de processus légers qu'il y a de sockets à lire. Le moniteur lit, formate et sauvegarde périodiquement les données durant toute l'exécution de la mission. S'il repère une collision, il stoppe directement la simulation. La détection et l'arrêt d'une mission qui ne termine pas fonctionne différemment et ne sont pas sous la responsabilité du moniteur. Ce sont les scripts de test qui lancent un compteur de temps avant chaque exécution et qui récupèrent l'événement de *timeout* pour le traiter et l'enregistrer.

Toutes ces données, prises du point de vue du robot et de la vérité terrain, sont des données brutes qui doivent être traitées afin de produire une vue synthétique des résultats de test. Le traitement est réalisé hors ligne par des fonctions d'analyse.

2.3.4 Analyse des données

L'analyse des données brutes peut servir différents objectifs : élaboration d'un verdict (oracle de test), mais aussi évaluation de robustesse, de performance, aide au diagnostic, etc. Nous avons intégré à la plateforme de test quelques outils de base pour traiter et visualiser les données d'exécution. De façon générale, notre expérience met en évidence la difficulté d'interpréter ces données automatiquement.

Quelle que soit l'analyse réalisée, un besoin fondamental est de pouvoir distinguer automatiquement les exécutions correspondant à un succès ou à un échec de la mission de navigation. Nous avons développé un utilitaire pour classer les exécutions de test selon ce point de vue. On considère que la mission est un *succès* lorsque la simulation n'exhibe pas de *timeout*, ni d'erreur venant de modules GenoM, ni de collision, et également lorsque la dernière position réelle correspond aux coordonnées du point objectif à une tolérance près. Dans tous les autres cas, la mission est un *échec*. Un échec peut être divisé en plusieurs catégories : *Fail-Collision*, *Fail-TimeOut*, *Fail-Error* ou *Fail-Other*, selon que la simulation exhibe une collision, un *timeout*, une erreur ou autre. Si la phase de simulation exhibe plusieurs problèmes, par exemple à la fois une collision et un message d'erreur, alors l'échec est classé dans la catégorie la plus sévère, qui est ici *Fail-Collision*. On calcule également la distance entre la position réelle à laquelle le robot se retrouve à la fin de la phase de simulation et la position du point objectif.

Ici, il est important de comprendre qu'un échec de la mission n'est pas nécessairement le signe d'une défaillance et n'implique pas que le logiciel de navigation soit fautif. Le robot peut s'arrêter à tout moment en cours de navigation s'il considère que la mission est infaisable. Il renvoie alors un rapport d'erreur. Comme P3D est un algorithme de navigation locale, cela se produit lorsque le robot entre dans une impasse ou une zone en forme de U. Il se trouve alors dans l'incapacité de sélectionner un arc qui le fasse sortir de la zone tout en le rapprochant du point de destination. Ainsi, un classement dans la catégorie *Fail-Error* ne suffit pas à déterminer un verdict *Fail* par l'oracle de test. Il faudrait analyser plus en détail la configuration du terrain et des obstacles autour du robot, ce qui paraît difficile à automatiser. Un classement en *Fail-TimeOut* n'implique pas nécessairement un verdict de rejet. Si l'impasse ou la zone en U est suffisamment large pour que le robot puisse évoluer à l'intérieur, il est connu que le robot pourra être amené à tourner en rond indéfiniment sans trouver la sortie, jusqu'à épuisement du temps imparti à la mission. La possibilité de ce comportement doit là encore être acceptée comme une conséquence du choix d'une heuristique locale telle que P3D. La seule classe conclusive du point de vue d'une défaillance logicielle est *Fail-Collision* : une collision révèle toujours une faute.

On retrouve là une illustration des difficultés à élaborer un oracle pour ce type de système. À part pour les défaillances catastrophiques, on ne sait pas distinguer un comportement défaillant d'un comportement inhérent à la résolution heuristique de problèmes complexes. À une granularité plus fine que le succès et l'échec de la mission, la comparaison détaillée par rapport à une trajectoire de référence ne serait pas non plus adéquate pour détecter les défaillances. Tout d'abord, exhiber une telle trajectoire serait problématique dans le cas de mondes et missions arbitraires, tels que ceux que nous générons aléatoirement. Ensuite, même si nous connaissions une ou plusieurs trajectoires optimales, nous devrions accepter la possibilité de trajectoires suboptimales ou d'échec de la mission. Et enfin, nous devrions prendre en compte l'indéterminisme à l'exécution : la répétition d'une même mission pourrait produire des trajectoires différentes d'une exécution à l'autre sans que ce soit une

anomalie. Clairement, à ce premier stade de nos travaux, le problème de l'oracle reste un point dur.

Si notre classement en succès et échec de la mission est insuffisant, il constitue néanmoins une base utile pour des analyses plus poussées, comme l'analyse manuelle de cas d'échec ou des évaluations expérimentales sur un ensemble de cas de test.

Dans notre plateforme, l'analyse manuelle est aidée par la possibilité de visualiser le dernier modèle numérique de terrain perçu par le robot. Cela peut permettre de comprendre pourquoi P3D considère qu'aucun arc ne peut être sélectionné. Nous avons également intégré un script développé au LAAS pour dessiner des trajectoires dans un monde virtuel 3D dans Blender, à partir d'ensembles de positions. Il permet de visualiser la trajectoire réelle prise par le robot durant l'exécution et de comparer celle-ci avec la trajectoire perçue par le robot. Une autre utilisation est de visualiser plusieurs trajectoires différentes du robot sur le même monde virtuel avec la même mission. La figure 2.6 correspond à ce cas d'utilisation et montre le comportement indéterministe du robot : selon les décisions prises en cours d'exécution, la mission pourra être un succès ou un échec. La figure illustre également l'effet des zones en U sur le robot : les boucles exhibées par certaines trajectoires correspondent à des moments où le robot a tourné en rond jusqu'à trouver une sortie.

Les sections suivantes détaillent une utilisation de la plateforme à des fins d'évaluation expérimentale. Le classement en échec et succès est exploité dans le cadre d'analyses plus poussées du niveau de difficulté des mondes générés.

2.4 Conception des expériences sur les niveaux de difficultés

Nous définissons la difficulté d'un monde en fonction de la faisabilité d'une mission de navigation dans ce monde, et de l'effort demandé au robot pour l'accomplir. Intuitivement, il est facile de naviguer dans un monde plat et sans obstacle, alors qu'un monde accidenté et obstrué sera difficile.

La notion de difficulté est utile pour définir les plages de variation des paramètres de génération de test. Par exemple, on fixera un pourcentage d'obstruction maximum pour les mondes générés, au-delà duquel la navigation devient trop difficile. La notion de difficulté peut aussi être exploitée plus finement dans le cadre de stratégies spécifiques, comme la production de mondes de difficulté croissante, ou un test de robustesse cherchant à stresser le logiciel de navigation.

Dans ce qui suit, plusieurs questions de recherche relatives à la notion de difficulté sont posées et des réponses y sont apportées expérimentalement grâce à la plateforme de test précédemment décrite.

2.4.1 Questions de recherche

La première question concerne le contrôle de la difficulté par notre génération procédurale de mondes. Rappelons que les mondes virtuels dans lesquels le système

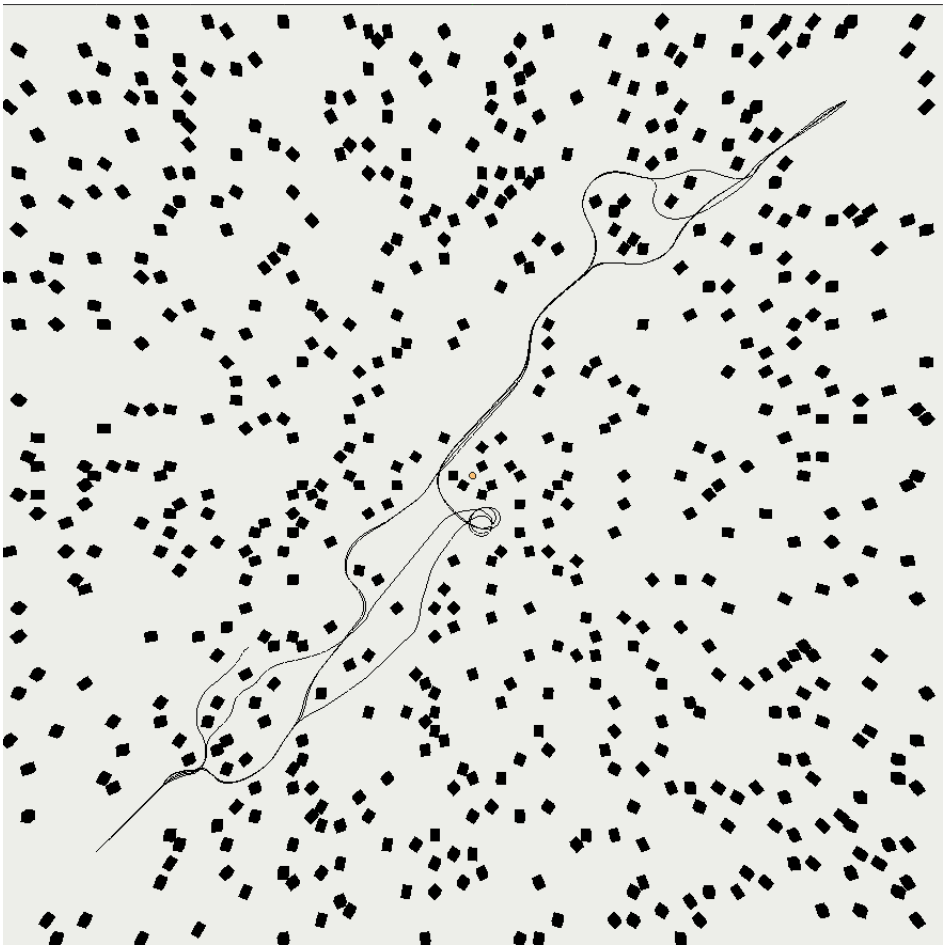

FIGURE 2.6 – Cinq trajectoires pour la même mission et dans un même monde présentant 6 % d'obstruction.

est testé sont obtenus à l'aide de quelques paramètres de haut niveau. Ces paramètres de génération sont principalement le degré de déformation et le pourcentage d'obstruction. Ils vont avoir un impact sur la difficulté des mondes, mais il n'est pas évident de savoir dans quelle mesure ils la déterminent du fait de la part d'aléatoire, et donc non maîtrisée, que comporte la génération. On souhaiterait idéalement identifier des plages de valeurs de paramètres telles que, par exemple, la plage 1 engendre des mondes *faciles*, la plage 2 engendre des mondes *difficiles* et la plage 3 engendre des mondes *très difficiles*. C'est cette correspondance entre plages de valeurs et niveaux de difficulté que nous voulons étudier expérimentalement, dans un objectif de calibrage de la génération.

Étant donné un monde, nous ne savons pas calculer sa difficulté *a priori*. Nous devons l'estimer *a posteriori*, en lançant plusieurs exécutions d'une même mission pour prendre en compte l'indéterminisme de la navigation. Ainsi, la difficulté d'un monde est évaluée à l'aide des mesures suivantes :

— le taux de succès des exécutions de la mission ;

— le temps médian pris par le robot pour rejoindre le point objectif ;

— la *tortuosité* médiane de la trajectoire empruntée.

Ces mesures sont agrégées afin d'obtenir une classification en plusieurs niveaux de difficulté (par exemple *facile*, *difficile* et *très difficile*). Nous cherchons à déterminer si les niveaux de difficultés obtenus correspondent à des plages de valeurs des paramètres de génération.

Q1 - *À quel point est-il possible d'identifier et de contrôler les niveaux de difficulté à l'aide des paramètres de génération ?*

La seconde question concerne l'indéterminisme de la navigation. Pour un monde virtuel donné et une mission sur celui-ci, les différentes exécutions de la mission peuvent donner des résultats très différents, comme présentés figure 2.6. Cet indéterminisme doit être évalué et pris en compte dans les méthodes de test, notamment lors de l'utilisation de techniques de génération itératives. Ainsi, pour un algorithme de recherche méta-heuristique, la valeur de *fitness* d'un cas de test ne peut plus être évaluée sur une unique exécution (cf. les travaux de [Nguyen 2009] mentionnés au chapitre 1). De la même manière, la mise en œuvre d'opérateurs de voisinage peut être entravée. Considérons par exemple une production incrémentale de cas de test voisins dans lesquels des obstacles seraient graduellement ajoutés sur la trajectoire prise par le robot : un fort degré d'indéterminisme compromettrait l'utilisation efficace de cette production.

Nous proposons une mesure de l'indéterminisme basée sur les différences de trajectoires lors de plusieurs exécutions de la même mission. Nous supposons que ce degré d'indéterminisme pourrait dépendre du niveau de difficulté. Une observation empirique de celui-ci en fonction des niveaux de difficulté est donc nécessaire.

Q2 - *Comment l'indéterminisme de la navigation évolue-t-il en fonction du niveau de difficulté ?*

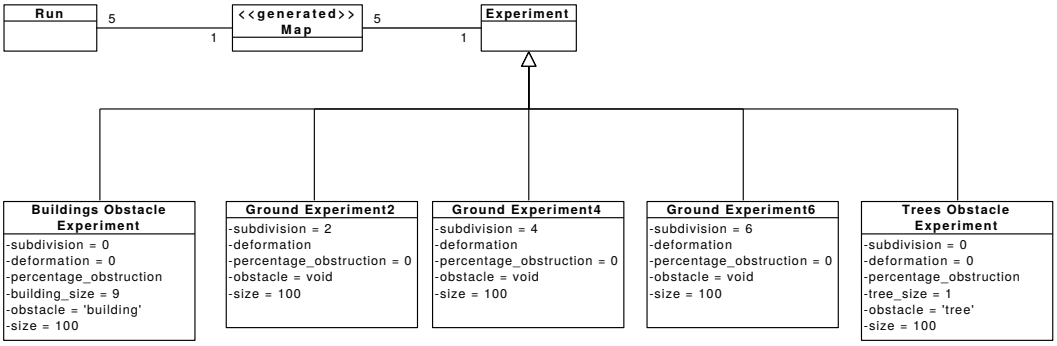

FIGURE 2.7 – Les cinq classes d'expérience.

La troisième question concerne une faute spécifique, à laquelle nous avons été confrontés au début de nos travaux. En effet, une faute de configuration était présente dans la plateforme de simulation initiale : le modèle de robot utilisé pour configurer P3D ne correspondait pas à l'avatar simulé. Les dimensions spatiales différaient, de sorte que P3D considérait une plateforme de taille inférieure à celle de l'avatar. Cette configuration incorrecte s'est traduite par de nombreuses collisions lors de simulations préliminaires, ce qui nous a permis de détecter le problème et de le corriger.

La configuration incorrecte de P3D fournit un exemple de faute induisant des défaillances catastrophiques détectables par l'oracle de test. Il nous a paru intéressant de ré-injecter la faute pour l'étudier dans l'optique d'un test basé sur des niveaux de difficulté. Par exemple, les mondes faciles suffisent-ils pour révéler la faute, ou faut-il favoriser les mondes difficiles et très difficiles ? Dans ce dernier cas, une trop grande difficulté pourrait s'avérer contre-productive, si le robot se trouve rapidement bloqué et doit renoncer à poursuivre la mission.

Nous mesurons le taux de collisions de la navigation fautive et observons son évolution en fonction de niveau de difficulté.

Q3 - *La détection de la faute de configuration injectée dépend-elle du niveau de difficulté d'un monde virtuel ?*

2.4.2 Description des expériences

On considère, afin de répondre à ces questions, plusieurs exécutions de mission, sur différents mondes virtuels dont la taille est fixée à $100\,m \times 100\,m$. Pour maximiser la portion de carte traversée par le robot, le point de départ et le point objectif des missions sont fixés respectivement en bas à gauche et en haut à droite du terrain. Une trajectoire directe, sur un terrain plat, représente un déplacement d'environ $140\,m$. Les expériences successives ne considèrent qu'un paramètre de génération à la fois. Par exemple, si la déformation varie, aucun obstacle n'est ajouté au monde. Réciproquement, si le pourcentage d'obstruction est considéré, le terrain est gardé plat. Ainsi, cinq classes d'expérience sont étudiées et présentées figure 2.7.

Trois expériences font varier la déformation d pour un nombre fixé de subdivisions $s = 2$, 4 et 6. Deux autres expériences concernent l'étude de l'obstruction, en considérant un seul type d'obstacle à la fois. Les obstacles, arbres ou bâtiments, sont placés aléatoirement sur le terrain jusqu'à obtenir le pourcentage d'obstruction désiré. Les bâtiments sont représentés par des cubes de 9 $m \times 9$ m et les arbres par des cubes de 1 $m \times 1$ m. Aussi, pour un pourcentage d'obstruction donné, le monde peut contenir soit peu d'obstacles larges, soit beaucoup de petits obstacles.

Pour chaque classe d'expérience, les valeurs des paramètres de génération étudiés sont échantillonnées avec un incrément fixe. Les différents échantillonnages de chaque classe permettent d'obtenir un ensemble de *configurations*. Pour une configuration d'une classe d'expérience donnée, 5 terrains différents sont générés et 5 exécutions de mission sont effectuées sur chaque monde virtuel, pour un total de 25 exécutions par configuration. Par exemple, 5 mondes virtuels différents avec des arbres représentant 6 % d'obstruction sont générés. Puis, 5 exécutions de la même mission sont effectuées sur chaque monde. Chaque exécution de mission dure quelques minutes. Le nombre de générations et d'exécutions de mission est fixé à 5 afin de prendre en compte l'aspect aléatoire de la génération, ainsi que l'indéterminisme de la navigation, tout en restant raisonnable du point de vue du temps pris par ces nombreuses exécutions. Ces diverses exécutions de la même mission sur un même monde virtuel permettent d'observer l'indéterminisme du système sous test ainsi que l'aléatoire relatif à la génération procédurale de monde.

Les données collectées durant les exécutions des missions sont utilisées afin d'attribuer à chaque configuration un niveau de difficulté (Q1). Cette attribution dépend du pourcentage de succès des 25 exécutions de mission, de la durée et de la tortuosité médianes des exécutions réussies. La tortuosité est introduite afin de caractériser les courbes d'une trajectoire, c'est-à-dire les détours que le robot a pris pour atteindre le point objectif. Elle correspond à la valeur de l'angle absolu moyen entre les positions successives du robot. La tortuosité est calculée à partir des N valeurs réelles des lacets (correspondant à l'angle d'Euler ψ, en degrés) observés du robot à l'aide la formule suivante :

$$tortuosité = \frac{\sum_{n=0}^{N-1}(|\psi_{n+1} - \psi_n|)}{N}$$

La durée et la tortuosité ne concernent que les exécutions réussies. Une première étape consiste donc à extraire les configurations pour lesquelles aucun ou très peu de succès ont été observés. Ces configurations reçoivent le niveau de difficulté *très difficile*. Une deuxième étape consiste à partitionner les configurations restantes à l'aide de l'algorithme de partitionnement *kmeans*[2]. Le cluster qui contient la configuration pour laquelle le monde exhibe un terrain plat sans obstacle est estampillé *facile*, alors que les autres configurations sont considérées *difficiles*.

L'indéterminisme (Q2) se produit quand, sur un même monde virtuel et une

2. Dans ce travail nous avons utilisé l'implémentation en langage R, voir `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/kmeans.html`

même mission, le robot prend des chemins différents. L'indéterminisme est observé en comparant les positions horodatées réelles issues de plusieurs exécutions de la même mission sur un même monde virtuel. L'algorithme 1 est utilisé afin de calculer une valeur représentant l'indéterminisme à partir des positions réelles obtenues lors de multiples exécutions d'une même mission sur un même monde. Cette mesure est basée sur la distance Euclidienne maximale entre les positions pour chaque temps t.

Algorithm 1 Calcul de l'indéterminisme

Input : *tab_trajectories* : tableau contenant les trajets observés (tableaux de positions) pour chaque exécution d'une même mission sur un même monde

```
tab_max_dists = [ ]
for traj1 in tab_trajectories do
    for traj2 in tab_trajectories do
        // Si ce n'est pas la même trajectoire
        if traj1 != traj2 then
            len = min(len(traj1), len(traj2))
            tab_dists_tmp = [ ]
            for i in range(0, len) do
                // tab.append(x) ajoute à la fin du tableau tab l'élément x
                tab_dists_tmp.append(euclidean_distance(traj1[i], traj2[i]))
            end for
            tab_max_dists.append(max(tab_dists_tmp))
        end if
    end for
end for
return max(tab_max_dists)
```

Les mondes virtuels produits pour l'étude du paramètre obstruction à l'aide des obstacles de type bâtiment ou arbre sont réutilisés dans des expériences où une version intentionnellement fautive des services de navigation est utilisée (Q3). La faute injectée consiste en la configuration de P3D pour un robot légèrement plus petit ($0,84\ m \times 0,45\ m$) par rapport à l'avatar utilisé par le simulateur ($1,14\ m \times 0,67\ m$). P3D sous-estime donc l'espace nécessaire pour le robot, ce qui peut mener à des collisions durant la phase de simulation. Comme précédemment, 5 exécutions de mission sont effectuées sur chaque monde virtuel. Durant ces exécutions, le nombre de collisions est relevé et donne un aperçu de la capacité de chaque configuration de paramètres de génération à révéler cette faute particulière.

2.5 Résultats

Toutes les expériences ont été effectuées à l'aide d'un ordinateur personnel équipé d'un processeur Intel Core i7-4800MQ cadencé à 2.70GHz, et de 16 GB de mémoire vive. La table 2.1 présente le nombre d'exécutions de mission pour chaque classe d'expérience et leur durée totale.

Classe d'expérience	Nombre total d'exécutions de mission	Temps total
Bâtiments	900	48h30min
Arbres	570	23h53min
Terrain s = 2	375	16h39min
Terrain s = 4	300	14h35min
Terrain s = 6	200	8h10min

TABLE 2.1 – Effort expérimental.

2.5.1 Contrôlabilité des niveaux de difficulté (Q1)

Les figures 2.8, 2.9 et 2.10 montrent les affectations des niveaux de difficulté aux configurations pour trois expériences : l'étude de l'influence du paramètre obstruction à l'aide d'obstacles de type arbre (figure 2.8), de type bâtiment (figure 2.9) et enfin l'étude de l'influence du paramètre déformation sur un terrain de subdivision $s = 2$ (figure 2.10). Chaque figure est composée de trois graphiques correspondant aux mesures utilisées afin d'affecter un niveau de difficulté à une configuration, soit : le taux de succès de mission (*Success*), la durée (*Duration*) et la tortuosité (*Tortuousness*) médianes des exécutions de mission réussies. Le graphique du taux de succès indique aussi les niveaux de difficulté attribués. Le symbole ● est utilisé pour représenter le niveau facile, ▲ pour le difficile ■ pour le très difficile.

Le seuil retenu sur le taux de succès en dessous duquel une configuration est estampillée très difficile est de 25 % de succès. En effet, en dessous de ce seuil, le nombre trop faible de mission réussies empêche un calcul représentatif de la durée et de la tortuosité (qui, pour rappel, ne sont calculées que pour les missions réussies). L'algorithme *kmeans* utilise un poids de 0.5 pour le taux de succès et 0.25 pour la durée et la tortuosité afin de partitionner les configurations qui ne sont pas estampillées très difficiles. Ces poids, choisis empiriquement, permettent de donner autant d'importance au taux de succès qu'à la durée et la tortuosité réunies. En effet, ces deux grandeurs rendent compte d'une réalité proche comme on peut le constater en comparant les graphiques en bas des figures 2.8, 2.9 et 2.10.

Les expériences concernant les bâtiments (figure 2.9) n'exhibent pas de configuration très difficile. Au delà d'une obstruction de 28.25 %, le placement aléatoire des bâtiments échoue à trouver assez d'espace libre afin d'ajouter les obstacles sans que ceux-ci se chevauchent. L'expérience s'arrête donc à cette valeur du taux d'obstruction.

Aucune collision n'a été observée durant ces expériences. La grande majorité des échecs de mission est due au fait que P3D considère que la continuation de la mission est impossible (*Fail-Error*) : il arrête le robot et lève un message d'erreur. Tous les autres cas correspondent à des exécutions de mission interrompues car ayant dépassées le temps alloué à celles-ci (*Fail-TimeOut*). Cela arrive lorsque le robot se trouve bloqué par des obstacles ou des pentes abruptes qui l'empêchent de rejoindre le point objectif, tout en le bloquant dans un minimum local. Dans ce cas de figure, le robot tourne en rond indéfiniment.

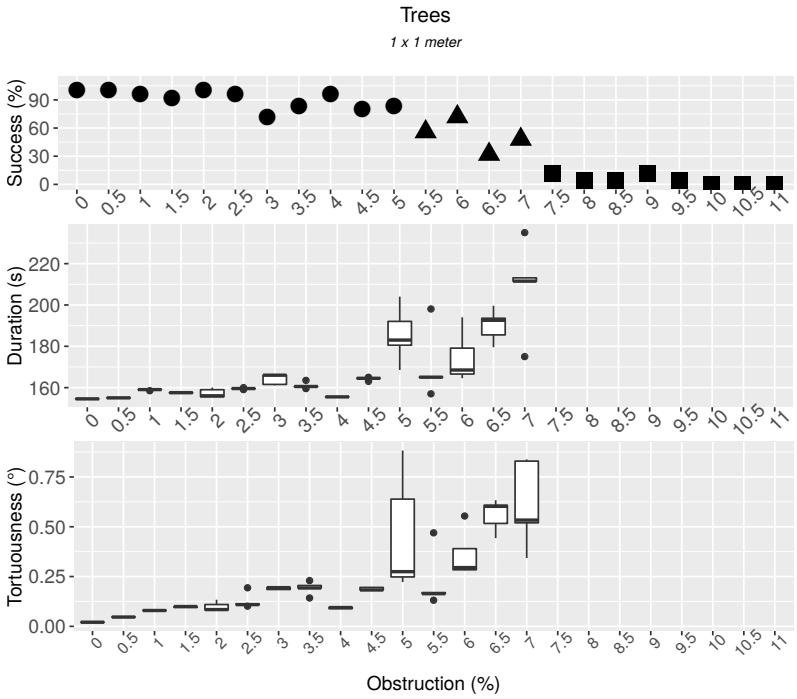

FIGURE 2.8 – Taux de succès, durées et tortuosités observées pour des configurations considérant le type d'obstacle arbre sur un terrain plat.

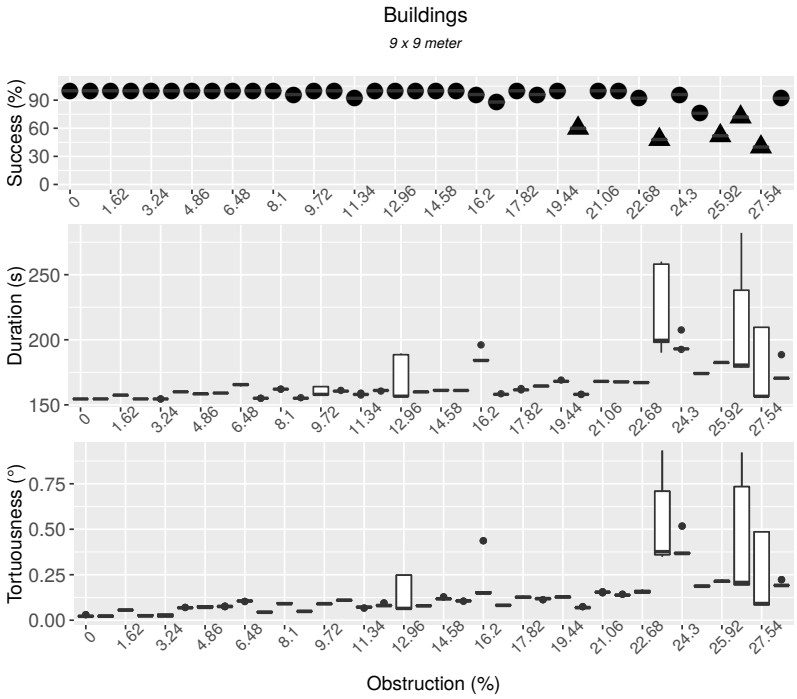

FIGURE 2.9 – Taux de succès, durées et tortuosités observées pour des configurations considérant le type d'obstacle bâtiment sur un terrain plat.

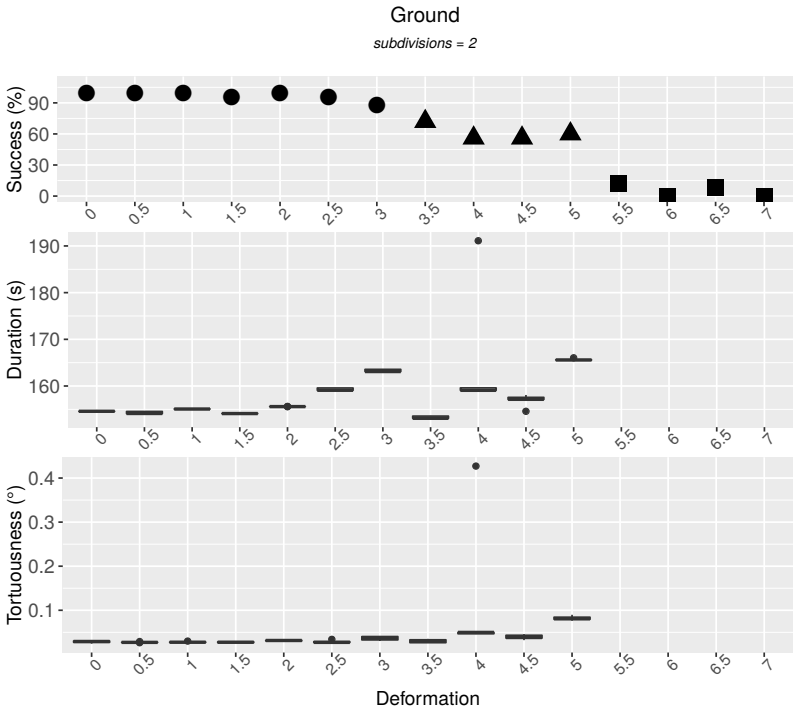

FIGURE 2.10 – Taux de succès, durées et tortuosités observées pour des configurations considérant une subdivision $s = 2$ sans obstacle.

De manière générale, une correspondance entre l'augmentation des valeurs des paramètres de génération d'un côté, et l'augmentation des niveaux de difficulté de l'autre, est observable. Par exemple, sur la figure 2.8, la configuration est simple jusqu'à 5 % d'obstruction, difficile entre 5 % et 7 % et très difficile au-dessus de 7 %. Ces seuils peuvent néanmoins être imprécis, ce qui est observable sur la figure 2.9 pour l'expérience concernant l'obstruction à l'aide de bâtiments. Dans cette expérience, on trouve à la fois des configurations faciles et difficiles entre 19 % et 28 % d'obstruction à l'aide de bâtiments. Ces seuils imprécis s'observent aussi pour une classe d'expérience concernant le degré de déformation pour une subdivision $s = 4$ avec une frontière mal définie entre les configurations estampillées faciles et difficiles, puis une autre frontière imprécise entre les configurations difficiles et très difficiles. Il faut rappeler qu'un nombre limité de cinq mondes virtuels par configuration est considéré, ce qui peut être insuffisant pour une détermination précise des seuils. Néanmoins, on a une bonne idée de l'ordre de grandeur.

Un autre résultat significatif s'observe en comparant les valeurs de seuils obtenues lors de différentes expériences d'une même classe (c'est-à-dire pour les expériences concernant le même paramètre de génération). En effet, on constate que la taille des obstacles influe sur la difficulté, lorsque l'on compare les résultats obtenus pour les arbres et ceux obtenus pour les bâtiments. Un monde avec 8 % d'obstruction par des arbres sera considéré comme très difficile, alors qu'un monde avec

le même pourcentage d'obstruction par des bâtiments sera facile. Il est donc plus stressant pour le système sous test de slalomer entre beaucoup de petits obstacles, plutôt que d'éviter quelques obstacles plus gros. De même, la comparaison des résultats concernant le degré de déformation montre que la taille de la surface déformée (définie par le nombre de subdivisions), compte. Plus la taille de cette surface est petite (c'est-à-dire plus le nombre de subdivisions sera élevé), plus la déformation aura un impact. Sur la figure 2.10, pour deux subdivisions, les configurations très difficiles commencent à $d = 5.5$. Ces configurations se retrouvent pour des valeurs de déformation plus petites pour 4 subdivisions, soit $d = 3.5$ et pour 6 subdivisions, soit $d = 2.5$.

Pour conclure, l'approche expérimentale que nous avons définie est pertinente pour aider à mettre en œuvre un contrôle basique de la difficulté des mondes, à l'aide de paramètres de génération. Nous avons eu des retours intéressants sur notre choix de paramètres. En particulier, nous avons vu que le pourcentage d'obstruction global n'offre pas un contrôle suffisant, car la difficulté dépend fortement de la taille des obstacles. Il faudrait donc inclure des paramètres supplémentaires pour préciser la pondération des différents types d'obstacles (par exemple poids ajustable des bâtiments et des arbres), voire pour contrôler la variation de taille au sein d'un même type (par exemple les arbres ne sont pas tous identiques). Pour les irrégularités de terrain, il est important de prendre en compte à la fois l'amplitude de la déformation et l'échelle à laquelle on l'applique (par exemple le nombre de subdivisions de terrain). Si l'on envisageait de combiner différents types d'irrégularité, plus ou moins amples et à différentes échelles, il faudrait là aussi enrichir les paramètres de génération pour pouvoir ajuster leur pondération et leur variabilité intra-type.

2.5.2 Indéterminisme (Q2)

Les figures 2.11 et 2.12 présentent les résultats de la comparaison des positions horodatées observées pour plusieurs exécutions d'une même mission sur un même monde. La figure 2.11 présente les résultats obtenus pour le paramètre obstruction sur le type d'obstacle arbre, et la figure 2.12 présente les résultats obtenus pour l'expérience concernant la déformation pour une subdivision $s = 2$. Chaque figure contient deux graphiques : celui qui se trouve en haut concerne les exécutions de mission réussies, et celui du bas concerne les exécutions de mission qui ont échoué. Les distances maximales intra-mondes au temps t sont calculées, puis la distance maximale moyenne est calculée pour les différents mondes virtuels d'une configuration de paramètres donnée. La comparaison des trajectoires réussies se concentre sur les configurations faciles et difficiles, car les configurations très difficiles ont trop peu de succès. De même, la comparaison des trajectoires infructueuses ne se fait pas pour les configurations faciles.

L'indéterminisme de la navigation est observable pour les cinq classes d'expérience exécutées (présentées figure 2.7). Le robot peut suivre différents chemins pour atteindre le point objectif (voir par exemple la figure 2.6), ou peut échouer dans diverses zones du monde virtuel. La différence entre les trajectoires est significative,

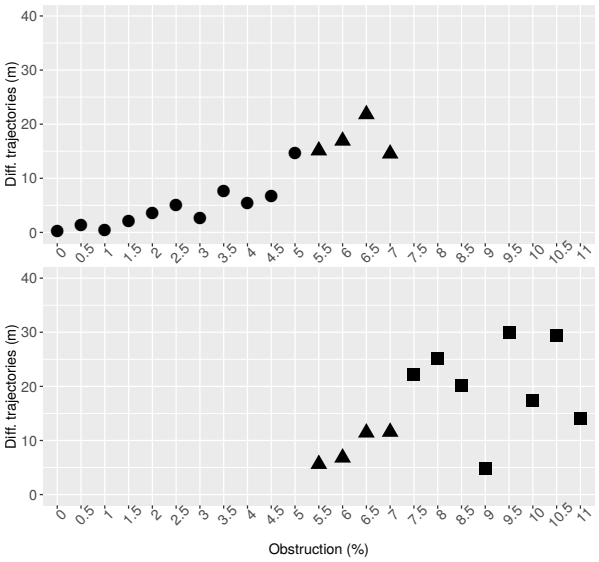

FIGURE 2.11 – Indéterminisme pour l'expérience concernant l'obstruction à l'aide d'arbre. Graphique du haut : exécutions de missions réussies, graphique du bas : exécutions de missions échouées.

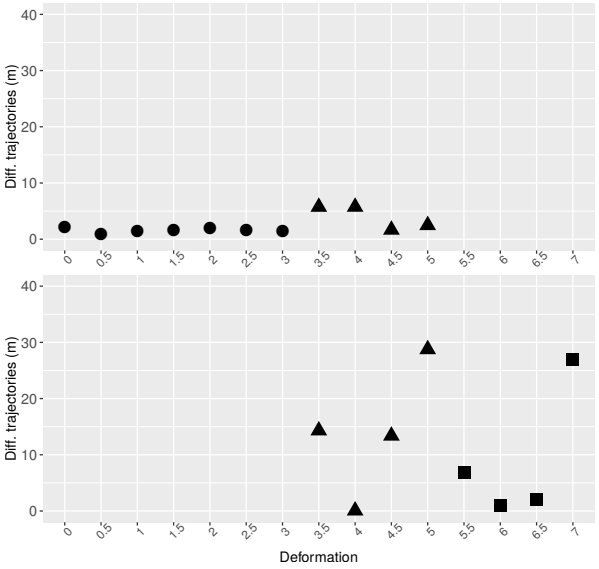

FIGURE 2.12 – Indéterminisme pour l'expérience concernant la déformation pour $s = 2$. Graphique du haut : exécutions de missions réussies, graphique du bas : exécutions de missions échouées.

de l'ordre de quelques mètres et même des dizaines de mètres. Pour les configurations faciles et pour les exécutions de mission réussies, l'indéterminisme reste stable ou est croissant. Dès que le taux de succès se dégrade, ces tendances n'apparaissent plus (voir par exemple le graphique du bas de la figure 2.12).

En conclusion, un comportement indéterministe apparaît à tous les niveaux de difficulté et le degré d'indéterminisme semble non prévisible, en tout cas pour les mondes difficiles et très difficiles. L'imprévisibilité rend d'autant plus problématique la prise en compte efficace de ce phénomène dans les méthodes de test, par exemple pour minimiser le nombre de rejeux nécessaires. Notons que l'indéterminisme peut également rendre complexe le test de situations dynamiques. Par exemple, si l'on souhaite intégrer dans le test un obstacle mobile coupant la trajectoire du robot, il conviendra d'implémenter des procédures de contrôle à la volée. Il est en effet impossible de savoir précisément à l'avance par où et quand le robot passera durant l'exécution de sa mission.

2.5.3 Étude d'une version fautive de la navigation (Q3)

La version fautive du système sous test considère que la plateforme physique du robot est plus petite que sa taille réelle. Cette faute implique que les trajectoires planifiées passent plus près des obstacles, ce qui augmente la probabilité d'une collision. On rappelle qu'aucune collision n'a été observée avec la version corrigée utilisée pour répondre aux questions Q1 & Q2. La figure 2.13 présente le nombre de collisions observées avec la version fautive pour les deux types d'obstacles.

De manière générale, les résultats correspondent aux observations précédentes : beaucoup de petits obstacles sont plus stressants que quelques plus larges. En résulte un plus grand nombre de collisions observées lorsque l'obstruction est effectuée à l'aide d'arbres que lorsque les obstacles sont des bâtiments, et cela pour tous les niveaux de difficulté.

Les configurations faciles tendent à exhiber moins de collisions que les difficiles ou très difficiles : elles révèlent donc moins bien la faute. Pour les bâtiments, la plupart des configurations faciles ne permettent pas d'observer des collisions. Ces expériences ne permettent donc pas de révéler la faute. Pour les arbres, le nombre moyen de collisions est de 3.82 pour les configurations faciles contre 9.75 pour les difficiles et 9.25 pour les très difficiles. Donc, du point de vue du test, les configurations faciles sont moins efficaces, par rapport à cette faute, mais il n'existe pas de différence significative entre les configurations difficiles et très difficiles.

Les résultats ne confortent pas l'hypothèse selon laquelle une trop grande difficulté nuirait à l'efficacité du test, le robot se trouvant rapidement bloqué. En fait, comme nous venons de le voir, les configurations très difficiles sont aussi efficaces que les configurations difficiles pour cette faute particulière. Une explication pourrait être que l'algorithme de navigation testé est robuste et que même dans des mondes virtuels très difficiles, le robot arrive à naviguer sur une distance suffisamment longue pour rencontrer diverses situations d'obstruction.

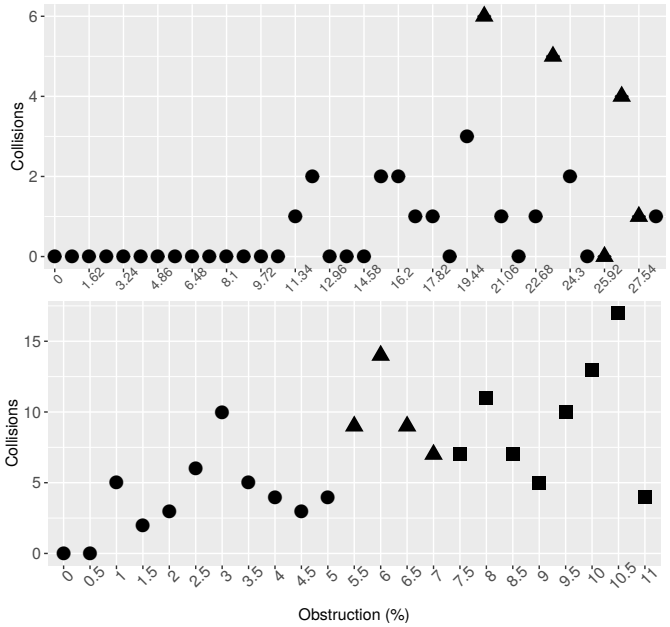

FIGURE 2.13 – Nombre de collisions. En haut : pour les mondes virtuels avec des bâtiments, en bas : pour les mondes virtuels avec des arbres.

2.6 Conclusion

Ce chapitre présente une plateforme de test capable de conduire des campagnes de test des services de navigation du robot Mana. Pour ce faire, cette plateforme génère les entrées de test, c'est-à-dire un monde virtuel et une mission conformément à deux paramètres macroscopiques de génération, qui sont l'obstruction et la déformation. Puis elle procède à des exécutions de mission dans le monde virtuel durant lesquelles diverses données sont collectées puis analysées.

La réalisation de cette plateforme a nécessité de traiter différents problèmes. Pour définir le modèle de monde à générer, notre proposition est de partir de cas d'utilisation. L'analyse de ces cas aborde différents aspects, incluant les caractéristiques de l'environnement, mais aussi la perception qu'en a le robot, les interactions qu'il a avec lui, et les caractéristiques de la mission à accomplir. Nous en déduisons les éléments du monde à considérer (par exemple, la présence d'obstacles de différents types), mais aussi ceux que l'on peut ignorer (par exemple, le tracé routier, la couleur des obstacles). L'analyse produit un diagramme de classe représentant la structure du monde. L'étape suivante consiste à associer des fonctions de génération à chaque classe du diagramme. Nous avons montré que l'API Blender s'avérait très utile pour implémenter ces fonctions, à travers les objets de base (*mesh*) proposés et les fonctions applicables à ces objets. En pratique, nous avons cependant été limités par des problèmes de performance en simulation, par exemple pour la gestion d'arbres de forme complexe, ou pour l'utilisation de la gravité dans le placement

des objets.

Notre plateforme sépare la collecte de données brutes, effectuée en ligne, de leur analyse effectuée hors ligne. Nous avons défini un ensemble de données pertinentes pour un logiciel de navigation et susceptibles de répondre aux besoins de différents types d'analyse. L'ensemble inclut à la fois des données prises du point de vue du robot et des données reflétant la vérité terrain. À titre d'exemple d'analyse, nous avons intégré plusieurs outils de base pour traiter et visualiser les données, dont un outil qui classe automatiquement les exécutions de test en fonction du succès ou de l'échec de la mission. Dans le cadre d'une étude expérimentale réalisée à l'aide de la plateforme, nous avons également élaboré des outils pour analyser la difficulté des problèmes de navigation soumis au robot et mesurer l'indéterminisme de sa trajectoire.

La difficulté d'un problème de navigation dans un monde est caractérisée par différentes mesures : sa faisabilité (mesurée par le taux de réussite observé) et l'effort pour le résoudre (temps, détours). Des niveaux de difficulté sont déterminés, basés sur ces mesures et mènent à une classification des mondes virtuels générés. Cette classification est appliquée pour étudier le contrôle de la difficulté à l'aide des deux paramètres de génération de notre modèle de monde : le pourcentage d'obstruction et le degré de déformation du terrain. Pour l'obstruction, notre étude montre que la taille des obstacles est un facteur déterminant de la difficulté. Donc, ce paramètre doit être réglé en ajoutant des poids sur les divers types d'obstacles.

Tandis que les résultats obtenus sont spécifiques au système sous test, la notion choisie de difficulté et l'approche expérimentale pour classer les mondes virtuels sont assez générales pour être utilisées pour d'autres algorithmes de navigation. De plus, cette approche peut être réutilisée pour étudier la contrôlabilité de la difficulté par des paramètres de génération différents de l'obstruction et de la déformation. Le contrôle de la difficulté peut ensuite être exploité dans le cadre de stratégies de test telles que la production de mondes de difficulté croissante, ou des tests cherchant à stresser le robot. À titre d'exemple, pour une faute de configuration que nous avons étudiée, les niveaux difficile et très difficile sont plus efficaces à révéler la faute que le niveau facile.

L'indéterminisme est observé au travers de la différence entre les trajectoires en comparant leurs positions horodatées. Une indétermination importante, de quelques mètres à plusieurs dizaines de mètres, a été mesurée pour tous les niveaux de difficulté ainsi que pour des missions réussies et infructueuses. Pour une architecture robotique fortement concurrente telle que celle du robot Mana, il semble impossible de mener des procédures de test sans faire face à ce phénomène, même pour un algorithme de planification déterministe. En conséquence, les résultats du test doivent être évalués en fonction des résultats de plusieurs exécutions de mission.

A l'issue ce premier travail, un important problème reste en suspens : le problème de l'oracle de test. L'oracle implémenté ne peut détecter qu'un ensemble restreint de défaillances, celles correspondant à des événements catastrophiques (collisions). Nous avons échoué à identifier d'autres types de défaillance. Un autre point négatif concerne l'adoption d'une approche basse fidélité de simulation, qui est discutable.

Enfin, il serait utile d'avoir des retours sur notre modélisation du domaine d'entrée, qui a pu laisser passer des caractéristiques importantes des mondes et des missions.

Pour avancer sur ces points, le chapitre suivant complète l'étude de la navigation de Mana par une analyse de fautes ayant affectées ce système dans le passé. Les conditions d'activation et les effets des fautes sont étudiés, ainsi que leur reproductibilité en simulation basse fidélité.

Chapitre 3

Étude de la reproductibilité de fautes en simulation

Sommaire

3.1 Introduction

Malgré l'évolution des simulateurs il existe un écart entre la simulation et le monde réel, notamment en ce qui concerne la prise en compte des lois physiques. La pertinence du test en simulation est par là-même discutable. En effet, il se pourrait que la plupart des fautes nécessitent des conditions réelles pour être révélées, c'est-à-dire pour activer ces fautes et observer leur(s) potentiel(s) effet(s). Des études concernant la reproduction de fautes en simulation sont donc nécessaires afin d'attester de la pertinence du test en simulation. Ce chapitre fait état d'une telle étude exploratoire appliquée à un logiciel académique pour la navigation de robots d'extérieur.

La plateforme de test présentée au chapitre précédent est représentative d'une approche de simulation basse fidélité. Elle est utilisée comme base de notre étude. Nous cherchons à déterminer si un test exécuté sur ce type de plateforme peut suffire à révéler des fautes de navigation qui sont actuellement trouvées par des expérimentations sur le terrain. Les fautes sont extraites, à l'aide d'une analyse manuelle, du suivi de versions du logiciel de navigation du robot Mana. L'échantillon de fautes

considéré est de taille raisonnable (33 fautes) permettant une analyse qualitative approfondie de chacune d'entre elles. Cette analyse manuelle représente un effort de 6 mois de travail. L'effort déployé, le nombre de versions du logiciel ainsi que de fautes analysées sont du même ordre de grandeur que dans des travaux similaires dans la littérature (voir [Jin 2012], [Abal 2014] et [Cavezza 2014] pour des exemples d'analyse approfondie de fautes dans d'autres domaines que le nôtre). Dans notre cas, il s'agit d'étudier la reproductibilité des fautes en simulation, par analyse détaillée de leurs conditions d'activation (appelées dans le suite *déclencheurs*) et de leurs effets observables. Pour certaines fautes, l'analyse va jusqu'à confirmer expérimentalement ces déclencheurs et effets dans des simulations MORSE.

L'analyse approfondie des fautes de Mana apporte de nouvelles connaissances sur les entrées de test, les données d'observation et la procédure d'oracle propres à révéler ces fautes. Au-delà de la question de la reproductibilité, elle permet aussi des retours sur la conception du test du logiciel de navigation, telle que nous l'avions mise en œuvre dans nos premières expériences.

Dans le chapitre précédent, un modèle de monde virtuel et de mission a été élaboré à partir d'un cas d'utilisation. En ce basant sur ce modèle, il est possible de générer un monde complet incluant un terrain présentant des irrégularités locales et parsemé d'obstacles, ainsi qu'une mission de navigation constituée de deux points : un point de départ et un point objectif. L'analyse des fautes permet d'effectuer un retour critique sur la pertinence de ce modèle et sa complétude. Nous pouvons ainsi déterminer si l'ensemble des déclencheurs identifiés est déjà présent dans notre modèle, ou si des éléments sont manquants.

Le problème de l'oracle est celui pour lequel nous avons le plus besoin de retour. C'est un problème complexe et ouvert, pour lequel la solution la plus rencontrée dans la littérature se cantonne à la détection de défaillances catastrophiques. L'analyse détaillée des effets des fautes offre l'opportunité d'identifier un spectre de défaillances plus large. Nous espérons ainsi recueillir des indications utiles pour opérer une discrimination entre comportements acceptables et comportements anormaux d'un logiciel de navigation. Dans une moindre mesure, nous cherchons aussi à évaluer la complétude de l'ensemble de données collectées. Rappelons que l'élaboration d'un verdict s'effectue hors ligne, par post-traitement des données brutes d'exécution. Cette approche permet de considérer différents types de traitements sans avoir à changer les instrumentations ni à rejouer les tests, mais suppose un ensemble d'observations raisonnablement complet. La connaissance des effets des fautes peut nous renseigner sur d'éventuelles omissions dans les données à collecter.

L'archivage du logiciel de navigation ciblé ainsi que son suivi de versions sont décrits section 3.2. La conception de l'étude est détaillée section 3.3. La section 3.4 analyse les résultats, émet des recommandations et discute les obstacles à la validité de cette étude.

Nom du module	.c	.h	.gen	Total
dtm-genom	2256	232	434	2922
pom-genom	1929	340	435	2704
p3d-genom	1984	677	592	3253
LibP3D	7526	1115	0	8641
Total	13695	2364	1461	17520

TABLE 3.1 – Nombre de lignes de code pour tous les modules du logiciel de navigation ciblé en fonction du type de fichiers considérés

3.2 Archivage et suivi de versions du logiciel de navigation de Mana

Le logiciel de navigation de Mana fait partie du dépôt *OpenRobots*[1]. Ce dernier contient principalement des logiciels développés au LAAS pour l'étude et la conception de plateformes robotiques comprenant des robots d'extérieur, d'intérieur ainsi que des drones. L'infrastructure *Robotpkg* est utilisée pour l'installation et la compilation des composants *OpenRobots* et de leurs dépendances. Mana est une plateforme robotique spécialisée dans la navigation en environnement extérieur. Le paquet *Mana meta-package*[2] dans *OpenRobots* contient tous les logiciels nécessaires pour prendre en charge la localisation, la cartographie, l'acquisition d'image et son traitement, le matériel (liens avec les drivers du robot), la planification de trajectoire et son exécution, les opérations mathématiques ainsi que la communication.

Cette étude se focalise sur le service de navigation comprenant, comme présenté dans le chapitre précédent, la cartographie gérée par DTM (voir module `dtm-genom`[3]), la localisation gérée par POM (voir module `pom-genom`[4]), et la planification locale (voir module `p3d-genom`[5]). Tous ces modules sont des modules GenoM.

Cette étude couvre les versions successives (appelées *commits* dans l'outil de suivi de versions) du logiciel entre 2005 et 2015. Les développements initiaux datent d'une période antérieure à 2005, mais les *commits* avant cette date n'ont pas été conservés. En 2005, ce logiciel avait déjà atteint un niveau relativement mature. Durant la période s'étalant entre 2005 et 2015, le service de navigation a été utilisé pour toutes les expériences menées par les chercheurs du LAAS en environnement d'extérieur, dans le cadre de divers projets collaboratifs. Les évolutions du logiciel sont dictées par les besoins des projets, et les actions correctives font suite aux problèmes rencontrés lors des expériences sur le terrain.

1. https ://www.openrobots.org/wiki
2. http ://robotpkg.openrobots.org/robotpkg/meta-pkgs/mana/index.html
3. http ://trac.laas.fr/git/robots/dtm-genom.git
4. http ://trac.laas.fr/git/robots/pom-genom.git
5. git ://trac.laas.fr/git/robots/p3d-genom

Le système sous test comprend $35K$ lignes de code, incluant les fichiers sources des modules DTM, POM et P3D, une bibliothèque pour le support d'exécution des modules, ainsi que des bibliothèques fonctionnelles dédiées à un module spécifique (par exemple, LibP3D) ou à un ensemble de modules (par exemple, la bibliothèque générique LibT3D pour la géométrie 3D). Les modules et bibliothèques sont développés et archivés dans des dépôts Git [6] séparés.

Pour que l'étude manuelle des *commits* soit faisable, la collecte et la documentation des fautes se concentrent sur un sous-ensemble de ces dépôts. Ce sous-ensemble est détaillé dans la table 3.1. Il est construit de manière à être représentatif des modules les plus pertinents liés aux problèmes de navigation. Il contient les trois modules de base et la bibliothèque fonctionnelle de P3D, comme le montre la table 3.1, pour un total de $17K$ lignes de code. Un module GenoM est construit à l'aide d'un fichier `.gen` qui décrit la structure du module, et de fichiers en langage C (`.c`, `.h`) qui décrivent les algorithmes encapsulés dans le module. Il est important de noter que ces modules sont génériques et peuvent être déployés pour d'autres robots que Mana.

Les modules sélectionnés ont été développés par des doctorants et post-doctorants qui n'exercent plus au laboratoire. De plus, aucun outil de suivi de faute n'a été utilisé durant le développement et la maintenance des modules. Les seules informations disponibles concernant l'évolution de ces modules sont les lignes de code ajoutées et supprimées entre deux *commits*, ainsi que le commentaire écrit par le développeur associé à chaque *commit*. La figure 3.1 montre deux exemples de commentaires, l'un corrigeant une faute et l'autre introduisant une simple mise à jour du code. Ces commentaires sont représentatifs des commentaires analysés dans cette étude : aucun formatage n'est utilisé et relativement peu de détails sont donnés. Entre deux *commits*, le code des modules peut être corrigé, du nouveau code ajouté ou du code existant supprimé. Toute modification ne correspond pas forcément à la corretion d'une faute mais peut avoir différentes motivations, telles que le nettoyage ou la re-factorisation du code, la mise-à-jour de bibliothèques afin d'ajouter des fonctionnalités, etc. L'analyse manuelle de ces modifications doit déterminer quels *commits* correspondent à la correction de fautes et en quoi ces fautes consistent.

3.3 Conception de l'étude

L'objectif de cette étude est d'avoir un retour sur la pertinence du test en simulation, comparé à du test en monde réel. Dans ce but, nous posons dans un premier temps plusieurs questions de recherche. Les réponses apportées visent à évaluer la pertinence de notre plateforme de test et à proposer des améliorations pour celle-ci. L'approche utilisée pour répondre à ces questions est décrite dans un second temps. Cette approche consiste en une analyse qualitative approfondie d'un ensemble de fautes extraites de l'historique des *commits* du logiciel considéré. L'analyse est complétée par des expériences de test en simulation.

6. https ://git-scm.com/

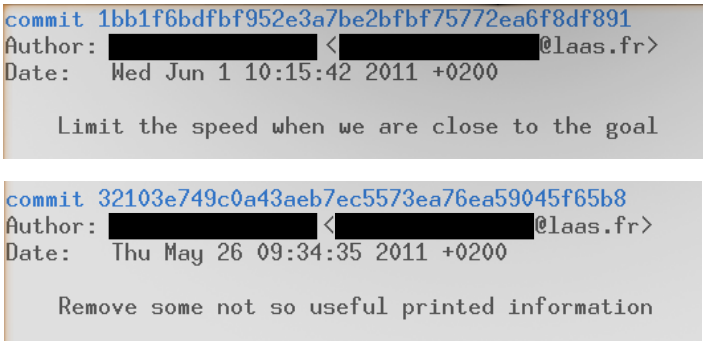

FIGURE 3.1 – Exemple de commentaires. En haut : un commentaire d'un *commit* corrigeant une faute ; en bas : un commentaire d'un *commit* non relié à une faute.

3.3.1 Questions de recherche

Nous prenons pour base de recherche le service de navigation du robot Mana et la plateforme de test décrite dans le chapitre précédent. La simulation dans cette plateforme est représentative d'une simulation *basse fidélité* de la physique du robot. Nous laissons pour le moment cette simulation dans l'état où elle nous a été fournie : nous ne simulons pas l'inertie du robot, les réactions entre les roues du robot et le terrain (potentiellement rocheux) ainsi que des zones glissantes. Nous émettons l'hypothèse qu'une simulation basse fidélité suffit à reproduire un grand nombre de fautes. Par exemple, cette simulation nous a permis d'observer les effets d'une faute de configuration injectée dans le système (voir section 2.5.3). Ici nous souhaitons aller au-delà de cette observation singulière, en confrontant notre hypothèse à un échantillon de fautes réelles. Nous nous posons donc une première question de recherche :

RQ1 - *Les déclencheurs des fautes issues de la couche de navigation peuvent-ils être reproduits et les effets correspondant peuvent-ils être révélés, dans une simulation basse fidélité ?*

En outre, il est utile, à partir de l'analyse détaillée des déclencheurs et des effets, d'émettre des recommandations sur la conception et la mise en œuvre de tests basés sur la simulation, c'est-à-dire sur la définition du domaine d'entrée, les données d'observation à collecter et les procédures d'oracle pour automatiser leur analyse.

La définition du domaine d'entrée devrait idéalement inclure toutes les caractéristiques des mondes et missions propres à révéler les fautes du logiciel de navigation. Ceci nous amène à la deuxième question :

RQ2 - *Quelles recommandations concernant les mondes virtuels et des missions de tests peut-on émettre à partir de l'analyse des déclencheurs et de leurs effets ?*

Les mondes virtuels et missions ne sont pas les seules données à fournir pour une exécution de test. Rappelons que le simulateur ainsi que le système sous test s'appuient chacun sur un fichier de configuration. Ces fichiers décrivent respectivement la configuration du robot simulé (par exemple le nombre et le placement de ses capteurs) et les paramètres des algorithmes de navigation (par exemple, le nombre d'arcs explorés par P3D et les limites physiques de la plateforme robotique utilisée). Malheureusement, les développeurs n'ont pas archivé les configurations matérielles et logicielles utilisées pour les expériences sur le terrain. Pourtant, ces configurations peuvent jouer un rôle dans les conditions d'activation des fautes. Ceci nous amène à la troisième question :

RQ3 - *Dans quelle mesure les fichiers de configurations peuvent-ils participer au déclenchement des fautes ?*

Enfin, comme vu section 2.3.4 l'oracle est un problème ouvert. Il est très difficile de distinguer un comportement défaillant d'un échec de mission inhérent à la résolution heuristique de problèmes complexes. Dans la plateforme de test présentée au chapitre précédent, seule la collision est interprétée comme une défaillance. L'analyse de l'effet de fautes réelles peut fournir des connaissances sur les comportements défaillants, ainsi que sur la façon de les détecter automatiquement. Ceci nous amène à la quatrième question :

RQ4 - *À partir de l'analyse des effets, quelles données d'observation sont utiles à la détection de fautes et quelles procédures pour l'oracle peuvent être utilisées pour les révéler automatiquement ?*

3.3.2 Détails de l'approche pour répondre aux questions de recherche

L'approche adoptée pour analyser les fautes comprend trois étapes :

— extraction des fautes à analyser, par examen des modifications de code dans l'ensemble des *commits* ;

— analyse approfondie de chaque faute et documentation dans un formulaire ;

— pour un sous-ensemble des fautes, confirmation expérimentale de l'analyse par des tests en simulation conçus pour révéler la faute.

Ces étapes permettent de recueillir les informations requises par nos questions de recherche : reproductibilité de chaque faute, déclencheurs et effets, éventuelles contraintes sur les fichiers de configuration.

3.3.2.1 Extraction des fautes

Les *commits* des modules P3D, DTM et POM ainsi que de la librairie LibP3D sont analysés manuellement depuis leur dépôts respectifs. Leur nombre raisonnable (moins de 400 *commits*) rend possible cet examen manuel de chacun d'eux. Dans des cas où cette stratégie n'est pas envisageable de par le trop grand nombre de *commits*, des stratégies permettant de sélectionner les *commits* les plus susceptibles

de proposer une correction de faute (aussi appelé *patch* en anglais) peuvent être employées. Ceci peut être fait par exemple, en isolant les *commits* présentant le plus de modifications dans le code [Nakamura 2006].

Pour chaque *commit*, une première lecture du commentaire écrit par son auteur ainsi que l'analyse des lignes de code ajoutées ou supprimées est faite. Le but de cette première étape est de déterminer si la modification de code correspond à la correction d'une faute. Si nous jugeons que c'est le cas, le *commit* est retenu afin de procéder à une analyse plus approfondie.

3.3.2.2 Analyse et documentation de chaque faute

L'analyse approfondie de chaque faute est guidée par un formulaire à remplir, consultable figure 3.2. Le formulaire utilisé s'inspire de celui proposé par les auteurs de [Nakamura 2006]. Comme nous, ces auteurs déploient une méthode d'analyse manuelle de fautes extraites de l'historique des évolutions de code. Nous reprenons la structure de leur formulaire en l'adaptant à nos questions de recherche : un champ sur la reproductibilité en simulation y est ajouté.

Notre formulaire d'analyse de *commits* comprend six champs. Son premier champ, **Localisation** contient les données factuelles qui identifient la faute en fournissant les informations nécessaires pour exactement retrouver les lignes de code concernées. La sous-section Ligne(s) sauvegarde la (les) ligne(s) corrigeant une faute, c'est-à-dire celle(s) qui correspond(ent) à un *patch*.

Le champ **Faute** contient du texte libre qui décrit la faute. Comme relevé dans [Nakamura 2006], le succès d'une construction ascendante de connaissance dépend de la quantité d'information de bas niveau contenue dans les rapports de faute. Par conséquent, la description de la faute doit être suffisamment détaillée pour indiquer clairement ce qui ne va pas dans le code. Par exemple, « la variable `uTurning` n'est pas réinitialisée lorsqu'un nouveau point objectif est transmis en paramètre » est préférable à la simple indication d'un « défaut d'initialisation ».

De la même manière, le champ **Défaillance** contient une description en texte libre de la défaillance, avec un degré suffisant de détails.

Temps pris pour correction rend compte des dates de l'insertion d'une faute et de sa correction. La date de correction correspond à celle du *commit* analysé. Une analyse manuelle des *commits* le précédant est effectuée afin de récupérer la date d'insertion de la faute. Il est à noter que, dans le cadre du logiciel de navigation considéré, la durée de vie d'une faute n'est pas significative. En effet, ce logiciel académique est utilisé de façon sporadique, selon les besoins des projets de recherche. Par exemple, une faute grossière empêchant P3D de démarrer est restée neuf mois inaperçue : cela pourrait signifier l'absence d'expérience en extérieur prévue pendant ces neuf mois, ou encore que les expériences effectuées ont utilisé une version P3D non archivée dans le dépôt. Bien que la durée ne soit pas pertinente, ce champ est gardé au cas où il faudrait analyser l'ordre séquentiel des fautes.

— **Localisation** : *localisation du patch*
  — Version :
  — Id. commit :
  — Id. faute :
  — Fichier(s) :
  — Ligne(s) :
  — Fonction(s) :
— **Faute** : *qu'est-ce qui n'allait pas dans le code ?*
— **Défaillance** : *comment la faute se manifestait ?*
— **Temps avant correction** : *temps entre l'insertion de la faute et sa correction*
  — Faute insérée :
  — Faute fixée :
— **Reproductibilité** : *comment la faute peut être déclenchée et la défaillance observée en simulation ?*
  — Jugement général : *non reproductible/reproductible/reproduite*
  — Contrainte(s) sur la fidélité de la simulation :
  — Contrainte(s) sur le monde virtuel/la mission :
  — Contrainte(s) sur les données de configuration :
  — Donnée(s) brute(s) à observer :
  — Post-traitement à effectuer pour détecter la défaillance :
— **Description** : *autres découvertes et précisions sur le contexte*

FIGURE 3.2 – Formulaire à remplir pour chaque faute rencontrée.

Reproductibilité est le champ central du formulaire. Ce dernier synthétise les résultats de l'analyse d'une faute par rapport à nos questions de recherche. Sur la base de la compréhension de la faute, les conditions de déclenchement, devant être remplies par les entrées de test (monde virtuel/mission) ainsi que par les données de configuration, sont identifiées. De plus, les données d'observation ainsi que leur traitement pour détecter la défaillance sont déterminés. Si la reproductibilité en simulation d'une faute donnée le requiert, les éventuelles contraintes sur la fidélité physique du simulateur sont aussi indiquées.

Pour finir, un jugement général sur la reproductibilité est donné. Une mention *non reproductible* est prononcée si on juge que la faute nécessiterait une trop grande fidélité physique ; elle est donc inobservable en simulation. Une mention *reproductible* est donnée si la simulation basse fidélité est jugée suffisante. Idéalement, cette dernière mention devrait rester temporaire : l'étape suivante consiste à confirmer expérimentalement que la faute peut être révélée avec notre plateforme de test basse fidélité. La mention passe alors de *reproductible* à *reproduite*. En pratique, il n'a été possible de tester qu'un sous-ensemble des fautes jugées reproductibles, pour les raisons qui sont expliquées dans la section suivante. Le jugement final garde donc les trois possibilités : *non reproductible*, *reproductible* ou *reproduite*.

3.3.2.3 Confirmation expérimentale de la reproductibilité des fautes

Cette dernière étape vise à confirmer les résultats de l'analyse précédente, pour chaque faute jugée *reproductible*. On conçoit des cas de test devant révéler la faute, à partir des déclencheurs identifiés. Ces tests sont exécutés sur la plateforme basse fidélité basée sur MORSE, et on observe si les effets de la faute sont bien ceux attendus. La faute est alors considérée comme *reproduite*.

Pour les expériences de test ciblant une faute, deux options sont envisageables :

1. Recréer le logiciel à la version précédant immédiatement le *commit* considéré ;
2. Injecter la faute dans la version courante du logiciel.

La première option n'est pas retenue. Un premier obstacle à celle-ci est le fait que les développeurs n'ont pas systématiquement archivé toutes les versions des divers modules et bibliothèques, et n'ont pas du tout archivé les fichiers de configuration. Étant donnée une date de *commit* pour un module, même en considérant les versions des autres modules correspondant à cette date, il n'est pas certain de recréer le logiciel qui a été exécuté lorsque la faute a été révélée. De plus, un second obstacle à cette option est que les scripts de test compatibles avec la version actuelle du système sous test ne le sont pas avec des versions plus anciennes ; beaucoup d'efforts seraient nécessaires pour modifier ces scripts pour chaque version à tester.

La seconde option est retenue, de sorte que la plateforme de test actuelle peut être utilisée. Cela nous permet également d'étudier les fautes une par une, en injectant une seule d'entre elles à la fois. Cependant, sur le plan technique, l'injection d'une faute peut s'avérer difficile. Dans certains cas, la fonction affectée par la faute n'existe plus dans la version actuelle du logiciel. Dans d'autres, le code affecté est

tellement modifié que l'injection ne consiste plus seulement à modifier les lignes corrigées par le *patch* : il est nécessaire de revenir à d'anciennes versions de plusieurs parties du logiciel. Identifier celles-ci peut s'avérer complexe. Ainsi, il n'est pas toujours possible d'injecter la faute dans le logiciel actuel. Pour ces cas là, une discussion avec un ingénieur de recherche ayant contribué à l'architecture logicielle robotique du LAAS, et plus particulièrement aux modules GenoM, a permis de valider la compréhension de la faute.

3.4 Résultats empiriques

356 *commits* sont analysés et détaillés ci-dessous :

— P3D : 69 *commits* ;

— LibP3D : 154 *commits* ;

— DTM : 50 *commits* ;

— POM : 83 *commits*.

Un total de 33 fautes est extrait de l'ensemble des *commits*. Dans la suite, nous détaillons l'analyse de ces fautes et répondons à nos trois questions de recherche.

3.4.1 Vue d'ensemble des fautes et de leur reproductibilité (RQ1)

La table 3.2 donne la répartition des types de fautes entre les divers composants du logiciel. La classification orthogonale des défauts (*Orthogonal Defect Classification* en anglais, ou ODC) [Chillarege 1992] est utilisée, avec une classe additionnelle concernant la gestion de la mémoire. Nous remontons cette sous-catégorie de faute au premier niveau, afin de souligner son nombre d'occurrences élevé dans le logiciel étudié. Rentrent typiquement dans cette catégorie les fautes dues à l'oubli de libération d'espace dynamiquement alloué, entraînant des fuites de mémoire. Les autres types de fautes concernent, dans l'ordre des colonnes de la table 3.2 : les affectations de variables, les vérifications sur des valeurs de variables, l'implémentation incorrecte d'algorithmes, les problèmes temporels, la conception incorrecte de fonctions et enfin les problèmes d'interface en raison de la gestion de paramètres dans différentes unités de mesure ou dans différents formats de stockage.

La table 3.3 présente le jugement général sur la reproductibilité des fautes en simulation. Une seule faute est jugée impossible à reproduire dans une simulation de basse fidélité (colonne *non reproductible*). Elle est l'unique faute de P3D classée dans catégorie *fonction* de la table 3.2. La fonction correspondante est celle qui permet l'auto-rotation. Le robot Mana est une plateforme à quatre roues motrices et elle peut tourner sur place en fournissant des consignes en vitesse de signes opposés aux roues gauches et aux roues droites. Cependant, cette rotation provoque des vibrations mécaniques de la structure physique, affectant les capteurs de telle sorte que le robot peut perdre sa localisation et corrompre le modèle numérique de terrain (géré par DTM). Cette fonctionnalité (l'auto-rotation) n'est plus présente dans la version actuelle de P3D : c'est la couche décisionnelle qui suspend la construction

Type de faute	P3D	LibP3D	DTM	POM	Total
Gestion mémoire	0	9	1	0	10
Affectation	2	2	2	1	7
Vérification	2	0	0	0	4
Algorithme	4	3	0	0	7
Timing	1	0	0	0	1
Fonction	1	0	0	0	1
Interface	2	0	0	3	3
Total	12	14	3	4	33

TABLE 3.2 – Types de faute selon la classification ODC.

Nom	non reproductible	reproductible	reproduite	total
P3D	1	4	7	12
LibP3D	0	9	5	14
DTM	0	1	2	3
POM	0	4	0	4
Total	1	18	14	33

TABLE 3.3 – Compilation des jugements généraux concernant la reproductibilité des fautes.

du modèle numérique de terrain lors d'une auto-rotation. Révéler ce problème durant la simulation nécessite une reproduction très précise des interactions entre les roues et le terrain provoquant des vibrations dont la propagation dans la structure mécanique du robot est également à prendre en compte.

Aucune des autres fautes ne dépend de la reproduction d'interactions physiques complexes. L'inertie est la seule contrainte physique jugée utile à ajouter à la plateforme de test. Cet ajout a permis d'observer une des fautes de P3D. Celle-ci provoque l'arrivée trop rapide du robot au point objectif, ce qui le force à freiner abruptement. La plateforme de test sans cette amélioration est suffisante pour observer le freinage abrupt, mais simule un arrêt immédiat irréaliste du robot. L'inertie a été introduite grâce à la modification de l'avatar du robot afin que MORSE simule le mouvement du robot via la rotation de ses roues. Ainsi, le *Bullet Engine* (le moteur physique utilisé par MORSE) peut gérer l'inertie et la friction des roues sur le terrain pendant la simulation et l'effet du freinage n'est plus instantané. Notons que la prise en compte de la friction des roues reste rudimentaire, de sorte que la simulation améliorée doit encore être considérée comme étant de basse fidélité.

Comme visible sur la table 3.3, la reproductibilité est confirmée pour un sous-ensemble de 14 fautes. Les autres (colonne *reproductible*) correspondent aux cas suivants :

— dix fuites de mémoire sont trouvées dans LibP3D et DTM. Ces fautes ne sont pas spécifiques au service de navigation, elles sont donc considérées en

dehors du champ de cette étude. Elles peuvent être traitées par des tests dédiés à l'aide d'outils d'analyse dynamique tels que Valgrind [Bond 2007]. Mais en théorie, elles pourraient également être reproduites par des tests de la navigation, en simulation ou en monde réel, à condition que les exécutions de mission soient suffisamment longues pour épuiser la mémoire ;

— quatre fautes de P3D affectent la fonction d'auto-rotation (indépendamment du problème de vibration de la structure déjà mentionné). Comme cette fonction n'existe plus dans la dernière version du logiciel, il est impossible de les injecter. Toutefois, elles ne sont pas difficiles à comprendre (par exemple une variable non réinitialisée), ce qui autorise une bonne confiance dans l'analyse faite quant aux déclencheurs et effets ;

— trois fautes de POM sont reliées au traitement de données de capteur. Ces fautes affectent la logique de conversion de l'un des formats de données que peut traiter POM. Malheureusement, la bibliothèque de composants de simulation MORSE ne comprend pas de capteur avec le format de sortie adéquat, et nous n'avons pu reproduire la faute. Néanmoins, il est clair que sa reproduction ne dépend pas de la fidélité physique de la simulation. Pour toute configuration avec un capteur adéquat, l'effet probable est que les données mal converties rendent le robot incapable de déterminer sa position ;

— une faute trouvée dans POM correspond à un débordement de tableau (faute commune pouvant être repérée à l'aide d'un outil d'analyse dédié). Il s'avère que le tableau est intégré dans une structure de données de type `union`, dont l'un des membres possède une taille supérieure à celle du tableau. Comme la taille d'une `union` a la taille maximale de ses membres, on dispose d'un espace mémoire supplémentaire dans lequel les accès hors borne peuvent avoir lieu. Cette faute ne peut donc provoquer aucune défaillance, que ce soit dans le monde réel ou, comme nos expériences le confirment, en simulation. Mais si la définition de l'`union` venait à changer, une erreur se produirait systématiquement. Par conséquent, cette faute est considérée comme reproductible aussi bien en simulation que dans le monde réel.

Cette analyse confirme la pertinence des tests en mondes virtuels. Bien que certaines fautes exigent des tests dans des conditions réelles (comme le problème de vibration que nous avons mentionné), de nombreuses autres peuvent être révélées à l'aide d'une simulation de basse fidélité.

3.4.2 Entrées : mondes, missions et données de configuration (RQ2, RQ3)

Sept fautes sur les 33 analysées n'ont pas besoin de condition de déclenchement spécifique en ce qui concerne les entrées de monde virtuel/mission ainsi que les données de configuration. Un cas de test basique suffit à les révéler, autrement dit une mission en ligne droite sur un terrain plat sans obstacle. La table 3.4 présente

la liste des conditions identifiées afin de déclencher les fautes restantes. Elles sont détaillées dans la suite.

	Conditions de déclenchement
Mission	**CD1** : Le point objectif localisé derrière le robot au début de la mission
	CD2 : Le point de départ et le point objectif n'ont pas la même valeur de la coordonnée Y
	CD3 : Longue distance entre le point de départ et le point objectif
	CD4 : Mission contenant plusieurs points de passage ou plusieurs missions en séquence
	CD5 : Annulation et remplacement de la mission
Monde	**CD6** : Impasse
	CD7 : Trou
	CD8 : Grand terrain
Config.	**CD9** : Paramètre profondeur de P3D > 1
	CD10 : Faible tolérance à l'objectif (testé avec 0.1)
	CD11 : Capteur spécifique
	CD12 : Arc le plus petit pouvant être sélectionné par P3D (testé avec 1 m)
	CD13 : Paramètres incorrects

Table 3.4 – Entrées et configurations utilisées pour déclencher les fautes.

Plusieurs conditions sur les missions sont identifiées. Elles peuvent concerner simplement l'emplacement du point objectif (**CD1**, **CD2**, **CD3**), mais également la nécessité de proposer une séquence de points objectifs (**CD4**) ou encore l'annulation de la mission et son remplacement par une autre (**CD5**). Les travaux présentés au chapitre 2 ne prenaient pas en compte ces aspects concernant la mission. Nous avions sous-estimé l'impact de ces aspects sur le système : nous pensions que seul le monde virtuel était intéressant du point de vue du test et qu'une mission définie par deux points fixes suffisait.

Certaines fautes requièrent une combinaison de conditions sur les missions afin d'être déclenchées. Par exemple, une faute de P3D affecte la gestion d'un nouveau point objectif, quand celui-ci est transmis au robot alors qu'il est en train d'effectuer une auto-rotation. Les conditions de déclenchement identifiées par notre analyse manuelle, consistent en la transmission d'un premier point objectif situé derrière le

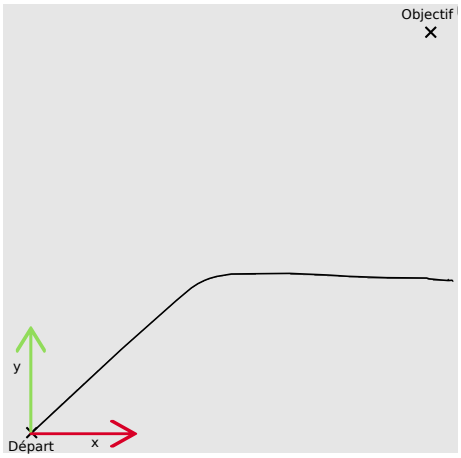

FIGURE 3.3 – Reproduction d'un bug affectant DTM.

robot afin qu'il commence son auto-rotation (**CD1**), puis le remplacement par un nouveau point alors que le robot tourne encore (**CD5**).

Un autre exemple est une faute de DTM qui affecte une mauvaise valeur à la coordonnée Y du robot dans sa carte perçue. Le déclencheur de cette faute combine deux conditions sur la localisation du point objectif : il ne doit pas avoir la même valeur absolue Y que le point de départ (**CD2**) et doit être suffisamment loin pour assurer la dégradation progressive du modèle numérique de terrain du robot (**CD3**), jusqu'à ce que la capacité de navigation soit affectée. Cette dégradation a pour conséquence que le robot perçoit de moins en moins l'environnement qui l'entoure. Ceci est illustré par la trajectoire du robot représentée figure 3.3 qui commence correctement mais qui, après une certaine distance sur un terrain plat et sans obstacle, dévie sans raison apparente et n'atteint pas le point objectif.

D'autres déclencheurs exigent que le robot soit exposé à des éléments spécifiques du monde tout en accomplissant sa mission. Par exemple, le fait d'être piégé dans une impasse (**CD6**) ou d'avoir une trajectoire qui traverse un trou (**CD7**). Les impasses permettent de déclencher trois fautes de P3D qui affectent la logique du rapport d'erreur `P3D_Blocked`. Ce rapport est émis lorsque le robot se retrouve dans une impasse, bloqué à cause d'obstacles autour de lui, qu'un arc est sélectionné par P3D mais que le premier obstacle le long de l'arc est trop proche du robot. Pour activer les fautes, il est nécessaire de modifier la configuration utilisée par P3D (**CD12**) afin que le robot n'émette pas une autre erreur, par exemple relative à l'impossibilité de sélectionner un arc. De plus, l'impasse doit être suffisamment étroite pour forcer le robot à s'arrêter et à lever l'erreur. En effet, si l'impasse n'est pas assez étroite, le robot tournera à l'intérieur de la zone afin de trouver une sortie. On a donc une combinaison de conditions liées à la configuration logicielle de P3D et à la présence d'une impasse de la bonne taille. Une des trois fautes de P3D évoquées ci-dessus nécessite en plus que l'impasse encercle le point objectif afin que, lorsque le robot atteint ce dernier, il signale une erreur au lieu du succès de la mission.

Les travaux présentés dans le chapitre 2 n'incluent pas explicitement les impasses dans le modèle du monde virtuel passé en entrée de la plateforme. Toutefois, le réglage du pourcentage d'obstruction peut produire des mondes virtuels avec des impasses de différentes tailles, et des situations dans lesquelles le robot se piège dans une impasse étroite ont été déclenchées. Par contre, le cas spécifique avec l'impasse entourant le point objectif ne pouvait pas se produire, car nous avions choisi de laisse une zone libre autour des points de départ et d'objectif (contrainte introduite afin d'éviter des missions infaisables, voir section 2.3.1).

Les trous n'étaient pas explicitement représentés dans notre modèle de monde virtuel ; nous considérions uniquement les dépressions locales du terrain. Pourtant, des trous plus profonds sont intéressants à ajouter car ils correspondent à des zones non perçues par le robot. Ces zones sont utiles afin de déclencher une des fautes de LibP3D, qui conduit le robot à choisir un arc dont un nœud est non perçu.

Les conditions sur les données de configuration d'entrée sont les plus difficiles à déterminer, les fichiers de configuration n'étant pas archivés avec le code source. Dans certains *commits*, il est clair que la faute est liée à des configurations logicielles incorrectes (**CD13**) : la modification de code inclut une vérification de la cohérence des données de configuration, ou le commentaire laissé par l'auteur porte sur la valeur d'un paramètre de configuration. D'autres cas sont plus complexes et nécessitent la lecture attentive des lignes de code modifiées dans le *commit*. Par exemple, il nous a fallu déduire qu'augmenter la profondeur de l'exploration arc (**CD9**), ou diminuer la valeur de tolérance de la distance au but (**CD9**), était nécessaire pour déclencher une faute. On peut noter que la configuration pour déclencher les fautes liées au rapport d'erreur `P3D_Blocked` (**CD12**) a été particulièrement difficile à trouver. De tels déclencheurs sont signalés dans un souci d'exhaustivité, mais ne constituent pas pour nous des entrées de test à proprement parler ; nous n'envisageons pas de tester génériquement la robustesse du logiciel par rapport à un ensemble de configurations, correctes ou incorrectes. De même, notre étude montre que la configuration matérielle (par exemple, celle de capteurs) peut participer au déclenchement des fautes (**CD11**), mais le type de système testé ne justifie pas l'exploration systématique de variantes de configuration. Par contre, ces fautes montrent que les données de configuration sont aussi importantes que le code source. Deux recommandations en découlent : une première est de toujours archiver les fichiers de configurations avec le code source, et une deuxième est de procéder à des tests de non régression si la configuration évolue, comme on le ferait pour une modification du code.

En conclusion, le logiciel analysé fournit divers exemples de déclencheurs pour les fautes de navigation. Leur étude donne des informations pour améliorer le modèle d'entrée élaboré section 2.3.1, en ce qui concerne la modélisation des missions et l'introduction de trous sur le terrain de mondes virtuels. Pour les missions, il est également intéressant de remarquer que nos expériences du chapitre 2 n'exploitaient pas toutes les possibilités du modèle : les coordonnées des points de la mission étaient fixées et non choisies aléatoirement. Cela restreint la diversité des missions pour les fautes qui dépendent des positions relatives de ces points. De façon générale, les

déclencheurs identifiés soulignent l'intérêt d'une génération procédurale de mondes et missions laissant une part d'aléatoire et donc de richesse dans l'instantiation des entrées de test. L'exemple de l'impasse discuté dans cette section est parlant. Même si cette situation n'a pas été explicitement identifiée dans le modèle de mondes, le placement aléatoire d'obstacles permet de la trouver. De plus, il est important de laisser les tests explorer différentes tailles d'impasse : pour les fautes analysées, il fallait exhiber la bonne largeur pour que le robot reste bloqué à l'intérieur.

3.4.3 Données d'observation et procédures d'oracle (RQ4)

Les défaillances induites par les fautes analysées sont présentées table 3.5. Les effets vont des défaillances évidentes telles que des crashs, ou l'impossibilité d'un module à démarrer, jusqu'à des problèmes de performance tels que des trajectoires sous-optimales dues à une mauvaise orientation initiale vers le point objectif, ou tels que l'échec d'une mission (alors que celle-ci est facile à réaliser). On observe aussi des défaillances qui présentent un réel danger si le test est effectué en monde réel (le robot tombe dans un trou, les commandes en vitesse ne sont pas rafraîchies et conservent leur dernière valeur pour toujours).

Les données d'observation collectées par la plateforme de test de base sont pertinentes vis-à-vis des effets des fautes. Les seules données manquantes sont les commandes en vitesse envoyées aux roues du robot (afin de détecter des à-coups au niveau de celles-ci). On peut donc dire que notre ensemble d'observations est raisonnablement complet.

Le problème le plus délicat n'est pas la collecte de données brutes, mais bien leur interprétation afin de détecter automatiquement les défaillances. La diversité des comportements défaillants constitue un défi pour l'oracle de test. De plus, il est difficile de faire la différence entre un problème affectant la performance du robot, et un comportement légitime. Cette distinction ne peut se faire qu'à l'aide d'une comparaison entre un comportement observé et une référence. Par exemple, un échec de mission ne peut être considéré comme une défaillance que si le testeur sait que la mission devrait être facilement réalisée par le robot. La trajectoire figure 3.3 n'est absurde que parce qu'on sait que le robot doit aller tout droit, et non pas dévier vers la droite. Même tourner en rond peut ne pas être le signe d'une défaillance : c'est un comportement connu et accepté de l'algorithme de planification utilisé. Ceci est dû au fait que P3D est un algorithme de planification locale. Ainsi, quand le robot est piégé dans une impasse, il est incapable de s'en sortir si le fait de se diriger vers la sortie augmente sa distance au point objectif (voir section 2.2).

Pour faire face à la diversité des défaillances observées, nous préconisons la création de multiples *détecteurs d'erreur*, chacun se concentrant sur une propriété simple. D'après l'analyse des fautes de Mana, nous pouvons identifier cinq classes de propriétés à couvrir par ces détecteurs (colonne de gauche de la table 3.5) :

— *Des propriétés liées aux phases de mission.* Pour le logiciel étudié, une mission peut être divisée en trois phases : initialisation, transit vers le point objectif, arrivée au point objectif. On pourra chercher à associer des propriétés spé-

Indicateurs pour la détection	Exemple de défaillance rencontrée
Phase de mission	P3D ne démarre pas
Phase de mission	Échec de l'orientation vers le point objectif
Phase de mission	Le robot tourne en rond autour du point objectif jusqu'à l'expiration du temps alloué
Phase de mission	Le robot arrive correctement au point objectif mais se considère comme bloqué
Phase de mission	Le robot freine trop tard quand il arrive au point objectif
Phase de mission	Échec inattendu de la mission évaluée comme facile (mission triviale ou test de non-régression)
Seuil lié aux mouvements	À-coup au niveau des commandes en vitesse angulaire
Seuil lié aux mouvements	Auto-rotation infinie
Événements catastrophiques	Le robot tombe dans un trou
Événements catastrophiques	Les commandes en vitesse ne sont pas rafraîchies et conservent leur dernière valeur pour toujours
Perception	Le robot a une trajectoire absurde
Messages d'erreur	Arrêt non immédiat du robot après une erreur
–	Crash du logiciel

TABLE 3.5 – Liste des modes de défaillance rencontrés.

cifiques à chacune de ces phases. Par exemple, une trajectoire sous-optimale est difficilement détectable en tant que telle, mais il est facile de détecter une mauvaise orientation par rapport au point objectif à la fin de la phase d'initialisation. Pour ce faire, l'orientation correcte du robot doit être identifiée comme une exigence à vérifier pour cette phase de mission. De la même manière, l'arrivée au point objectif peut être soumise à d'autres exigences, comme le signalement du succès de la mission lorsque l'objectif est atteint, ou le non dépassement du point objectif (directement identifiable à l'aide des observations collectées), etc. ;

— *Des seuils liés aux mouvements du robot*, comme la variation maximale des commandes en vitesse (qui permet de détecter des à-coups), ou l'angle maximal d'une auto-rotation (il ne doit pas dépasser 180°, ce qui permet de détecter une auto-rotation sous-optimale ou infinie) ;

— *Des événements catastrophiques*, tels que la collision avec un obstacle ou la chute dans un trou. Une éventuelle chute est détectable, par exemple, à partir des positions des trous sur la carte et des données d'observation ;

— *Des propriétés liées aux messages d'erreur*, comme le fait que le robot doit s'arrêter immédiatement quand il émet un message d'erreur ;

— *Des propriétés liées à la perception*, comme avoir une bonne perception de sa position (à un paramètre de tolérance près). Détecter que le robot a une mauvaise perception de son environnement est plus difficile. Pour la faute conduisant à la trajectoire erronée de la figure 3.3, le pourcentage élevé de zones inconnues dans la carte perçue est un détecteur possible.

L'idée est donc de se baser sur ces grandes classes de propriétés pour spécifier un ensemble de détecteurs pour le système de navigation considéré. L'ensemble peut être enrichi au fur et à mesure que le testeur acquiert une plus grande expérience des fautes du système (ou de systèmes similaires), notamment celles trouvées par des tests sur le terrain.

La détection de problèmes additionnels liés à la performance, qui ne sont capturés par aucun des détecteurs d'erreur, est un problème ouvert. Une solution partielle consiste à commencer par des cas de test pour lesquels un comportement de référence peut être prédéterminé. Par exemple, le robot se voit attribuer une mission triviale qu'il doit réussir, et sa trajectoire ne doit pas trop s'écarter de celle précalculée (typiquement, une ligne droite). Mais ce ne sera pas suffisant pour les fautes avec des déclencheurs complexes. Une solution peut alors être d'utiliser les niveaux de difficulté tels que définis dans le chapitre 2 afin de faire du test de non régression. Par exemple, on suspectera l'introduction d'une faute lorsqu'un monde virtuel classé comme *facile* avec une version antérieure du logiciel devient *difficile* ou *très difficile* avec la version la plus récente (c'est-à-dire le pourcentage d'échec de la mission augmente, les missions prennent plus de temps et impliquent plus de détours).

3.4.4 Obstacles à la validité de cette étude

Les résultats présentés dans ce chapitre découlent de l'analyse d'un logiciel de navigation académique spécifique, ce qui peut constituer une menace pour la validité en dehors de cette étude. De plus, le « temps avant correction » (champ du formulaire de description de *commits*) des fautes trahit une utilisation non continue du logiciel ciblé. Elle peut refléter des périodes de non-activité ou simplement des trous dans le processus d'archivage. Par conséquent, l'analyse temporelle de ces données a peu de signification. Ce logiciel est cependant représentatif de nombreux services de navigation, que ce soit en termes des algorithmes qui le composent ou de complexité du code. Au cours de la période couverte par l'étude (soit environ 10 ans), ce logiciel a été utilisé dans plusieurs applications réelles, dont certaines ont été déployées en dehors du LAAS où il a été développé.

Comme dans toutes les études similaires basées sur des corrections de fautes, cette analyse ne porte que sur les fautes trouvées. Cela peut induire une surreprésentation des fautes faciles à révéler par rapport à des fautes plus difficiles et encore inconnues. En outre, l'ensemble des données analysées est composé de fautes révélées à l'aide de procédures de test non formalisées, et en monde réel, qui sont donc contraintes en termes d'entrées de test (quelques scénarios seulement sont testés). La surreprésentation de fautes faciles est un obstacle à l'obtention de résultats quantitatifs, mais gêne moins la collecte de connaissances qualitatives auxquelles nous nous sommes intéressés. Outre les fautes qui peuvent être révélées par des cas « faciles » (c'est-à-dire un monde plat, sans obstacle et une mission en ligne droite), des exemples intéressants de déclencheurs ont également été trouvés et différents modes de défaillance identifiés.

En ce qui concerne la question de la reproductibilité, les résultats sont suffisants pour attester que des tests basés sur la simulation constituent une approche à considérer. Supposons qu'il manque quelques fautes difficiles à révéler dans nos données expérimentales et que ces fautes ne soient pas reproductibles en simulation. Il n'en reste pas moins vrai que de nombreuses fautes actuellement trouvées lors d'expériences en monde réel peuvent également être révélées par des tests en simulation.

En ce qui concerne la validité interne de cette étude expérimentale, certaines fautes ont pu ne pas être identifiées par l'analyse manuelle. C'est un problème récurrent pour les études de faute centrées sur une analyse manuelle, mais cela ne réduit pas la pertinence des recommandations tirées pour les fautes identifiées. Un autre problème discutable est la méthode de reproduction des fautes. Nous avons opté pour l'injection de faute dans la version actuelle du logiciel plutôt que le test d'une version obsolète complète. La stratégie d'injection choisie pourrait induire des différences pour les déclencheurs et effets, qui ne correspondraient pas à ceux dans la version obsolète. Néanmoins, nous avons effectué une analyse manuelle approfondie de chaque faute dans le contexte de sa version. Pour les fautes reproduites avec succès à l'aide de la version actuelle du logiciel, les déclencheurs et les effets étaient ceux attendus d'après l'analyse manuelle. Pour les fautes dont la reproduction n'a

pas été faite, nos résultats ont été validés avec un ingénieur de recherche travaillant sur les logiciels robotiques étudiés et impliqué dans le développement du logiciel considéré.

3.5 Conclusion

Dans ce chapitre, la reproductibilité en simulation de fautes affectant le logiciel de navigation d'un robot d'extérieur est étudiée. Ces fautes sont collectées à l'aide d'une analyse manuelle de l'historique de développement du logiciel visé.

L'analyse approfondie des fautes fournit des informations utiles sur des déclencheurs et des comportements défaillants spécifiques au domaine. Elle suggère également des recommandations pour les développeurs de modules robotiques ainsi que pour leurs testeurs. L'une d'entre elles est de considérer les fichiers de configuration comme partie intégrante du logiciel. En effet, plusieurs déclencheurs de faute sont liés à la configuration, que ce soit du matériel (par exemple, les capteurs du robot) ou du logiciel (par exemple, les paramètres de l'algorithme de planification de chemin). Il est donc très important d'archiver la configuration et de valider les changements de configuration de la même manière que les modifications de code le seraient. Cela n'a pas été fait dans le cas du logiciel académique étudié. Une autre recommandation générale est de traiter séparément du reste les fautes de programmation classiques non spécifiques au domaine, comme les fuites de mémoire et les fautes d'indexation de tableau. Elles représentent un tiers des fautes analysées. Une analyse spécifique à l'aide d'outils comme Valgrind est conseillée afin de détecter ces fautes avant de procéder à la validation de la navigation.

Dans l'ensemble, l'étude soutient l'hypothèse selon laquelle la révélation de nombreuses fautes de navigation ne nécessite pas la reproduction de phénomènes physiques complexes. Une seule faute est liée à des aspects difficilement simulables de la physique. Il s'agit d'une incompatibilité des fonctions d'auto-rotation et de cartographie, cette dernière étant altérée par des vibrations mécaniques de la plateforme durant l'auto-rotation. Les déclencheurs et les effets des autres fautes analysées sont bien reproductibles dans une simulation dite de *basse fidélité*. En ce qui concerne la physique, la seule amélioration suggérée de la plateforme de test est de tenir compte de l'inertie. Cet ajout permet de constater que le robot peut freiner trop tard lors de son arrivée au point objectif. Il est intéressant de noter que quelques fautes induisent un comportement dangereux (le robot tombe dans un trou, ne rafraîchit plus ses commandes en vitesse), ce qui confirme la pertinence de procéder à des tests en simulation pour des soucis de sécurité.

Les déclencheurs identifiés suggèrent des améliorations du modèle d'entrée considéré dans les expériences du chapitre 2. Le modèle de mission qui y est exploité est incomplet en termes de position relative des points de départ et objectif, du nombre de points de passage à atteindre ou des aspects de la gestion de la mission, comme l'abandon et le remplacement de mission. De plus, le modèle de monde n'intègre pas d'éléments tels que des trous que le robot doit considérer comme inconnus et po-

tentiellement dangereux (donc à éviter). Quelques déclencheurs combinent plusieurs conditions sur les missions, les éléments du monde ou l'état courant du robot (par exemple, le remplacement de la mission pendant que le robot tourne sur place pour s'aligner sur un point derrière lui). Ils peuvent être reliés à la notion de scénario ou *situation* [Alexander 2015], mise en avant par plusieurs auteurs pour guider le test. Certains cas peuvent être difficiles à trouver, même avec un test guidé, comme ceux qui requièrent une largeur d'impasse spécifique. Ils appuient l'importance de l'utilisation d'algorithmes de génération procédurale qui permettent la conservation d'une part d'aléatoire durant l'élaboration des entrées de test.

Les effets observés des fautes sont diversifiés et certains ne sont pas facilement différentiables de comportements normaux. Des comportements de références seraient nécessaires. Dans les faits, un tel comportement de référence (une trajectoire par exemple) ne peut être déterminé que pour les cas les plus simples. L'insertion de nombreux détecteurs de comportement défaillant, qui seront utilisés de manière générique pour toutes les missions de navigation testées, peut être une manière d'atténuer ce problème. L'analyse des fautes nous a permis de définir des catégories que ces détecteurs doivent au moins couvrir : des propriétés associées aux phases de mission, des invariants basés sur des seuils liés aux mouvements du robot, l'absence d'événement catastrophique, des propriétés liées aux rapports d'erreur et des propriétés liées à la perception.

Le chapitre suivant, présente une étude de cas industriel : un robot agricole naviguant dans des rangées de légumes. Cette étude permet de développer et transférer les connaissances obtenues à partir de l'analyse des fautes présentée dans ce chapitre.

Chapitre 4

Application au cas d'un robot agricole

Sommaire

4.1 Introduction

Dans ce chapitre, les leçons tirées des chapitres précédents sont utilisées afin de mener une campagne de test sur un robot industriel agricole : le robot Oz. Ce cas d'étude est fourni par l'entreprise toulousaine Naïo qui est spécialisée dans le développement et la commercialisation de robots agricoles depuis 2011. Les principes de la plateforme de test développée au chapitre 2 sont repris et adaptés au robot Oz ainsi qu'au simulateur Gazebo qui est utilisé par l'entreprise partenaire. Notre étude exploratoire de Oz revisite les points cruciaux du test de systèmes autonomes en simulation, c'est-à-dire la génération des entrées, la gestion de la phase de simulation ainsi que l'élaboration automatique de verdicts par un oracle de test.

Concernant la génération des entrées, la méthode utilisée dans le chapitre 2 est maintenant appliquée à un cas plus complexe. Comme précédemment, nous n'avons pas à notre disposition de documentation spécifique pour spécifier les entrées : le domaine d'entrée est déduit de cas d'utilisation. Un domaine d'entrée riche, comportant un grand nombre de paramètres, est ainsi obtenu. Pour faciliter la gestion de ces paramètres, nous étendons notre approche de modélisation structurelle des mondes en combinant les diagrammes de classes UML avec une grammaire formelle. Un mot de la grammaire représente le génotype d'un monde, agrégeant les informations sur ses paramètres de génération. Cette approche offre une grande souplesse pour créer, manipuler et vérifier des configurations de valeurs de paramètres, avant de lancer la production de contenus de mondes dans un format compréhensible par l'environnement de simulation.

La gestion de la simulation concerne les compromis à réaliser pour des besoins de performance. Gazebo exhibe des problèmes de performance similaires à ceux que nous avons rencontrés avec MORSE. Pour économiser le temps de calcul et l'espace mémoire, nous sommes amenés à considérer les mêmes types de compromis que pour l'étude de cas Mana : simplification des formes des objets présents dans l'environnement du robot, simulation basse fidélité du point de vue des lois de la physique. Nous avons vu dans le chapitre 3 que de nombreuses fautes ne nécessitent pas une grande fidélité de la physique pour être révélées. Ce résultat est encourageant pour faire face aux limitations des environnements de simulation, comme celles des deux environnements que nous avons expérimentés.

Les enseignements tirés du chapitre 3 sont également utilisés afin de guider la spécification de l'oracle de test. Nous avons vu qu'un oracle possible prenait la forme d'une liste de détecteurs couvrant cinq grandes classes de propriétés : des propriétés associées aux phases de mission, des invariants basés sur des seuils liés aux mouvements du robot, l'absence d'événement catastrophique, des propriétés liées aux rapports d'erreur et des propriétés liées à la perception. Cette classification est issue de l'étude des fautes du robot Mana et nous étudions son application au robot Oz. Les détecteurs précis à mettre en œuvre dépendent bien sûr du robot testé, mais les cinq grandes classes identifiées offrent un cadre sur lequel s'appuyer pour éliciter les comportements anormaux.

Le chapitre est structuré comme suit. Le cas d'étude est présenté en section 4.2. La section 4.3 traite de l'environnement de simulation et des modifications apportées à la simulation de l'entreprise partenaire. Les sections 4.4 et 4.5 détaillent les méthodes utilisées afin de mettre en place la génération automatique des entrées de test et l'oracle. Les résultats du test effectué sur le robot Oz à l'aide de notre plateforme de test en simulation sont montrés et discutés dans la section 4.6.

4.2 Présentation du robot

Oz est un robot bineur autonome dédié aux exploitations maraîchères. La photographie reproduite en figure 4.1 illustre Oz en train de désherber un champ. Une

FIGURE 4.1 – Robot Oz travaillant dans un champ avec ses outils. À gauche : vue de face ; à droite : vue de dos.

exploitation maraîchère est organisée en une, ou plusieurs rangées de légumes. Cet ensemble de rangées de légumes constitue un champ. La mission du robot consiste à se déplacer entre les rangées de légumes composant le champ. L'opération de désherbage est effectuée mécaniquement à l'aide d'outils spécifiques attachés à l'arrière du robot et tractés par celui-ci. Lorsqu'il y a plusieurs rangées de légumes, le robot effectue des demi-tours à la fin de chacune de celles-ci afin de toutes les parcourir.

Les petites dimensions du robot Oz ($75\ cm \times 45\ cm \times 55\ cm$) lui permettent de se faufiler entre les rangées de légumes à désherber. L'acquisition de son environnement se fait à l'aide d'une télédétection par laser (LiDAR 1D) à l'avant, ainsi que deux caméras. La télédétection permet de repérer les rangées de légumes afin de guider le robot entre elles. Les caméras servent à repérer les piquets rouges qui marquent la fin et le début de chaque rangée de légumes. Elles permettent aussi l'utilisation de techniques de stéréo-odométrie afin de détecter d'éventuels patinages du robot pendant les demi-tours. Des capteurs de contacts sont ajoutés au châssis afin d'arrêter d'urgence le robot en cas de collision. Le robot se déplace à une allure de $0.4\ m.s^{-1}$ et pèse $110\ kg$ sans outil.

D'un point de vue logiciel, les composants du robot sont développés en langage C et C++. Le code représente un total de 151 556 lignes de code. Ce code inclut quelques fonctions non prises en compte par notre étude telles que l'interface utilisateur et la commande des outils. Le système sous test considéré pour cette étude est la navigation du robot Oz.

4.3 Environnement de simulation

Naïo fournit, en plus du système sous test, une première version de la simulation dont une vue schématique est donnée figure 4.2. Cette simulation est basée sur le simulateur Gazebo, dont nous avons parlé au chapitre 1 et qui est très utilisé en robotique. Le simulateur lit trois fichiers (une image `.jpg` et deux fichiers `.sdf`) afin d'instancier un monde virtuel. Le système sous test **OzCore**, aussi représenté

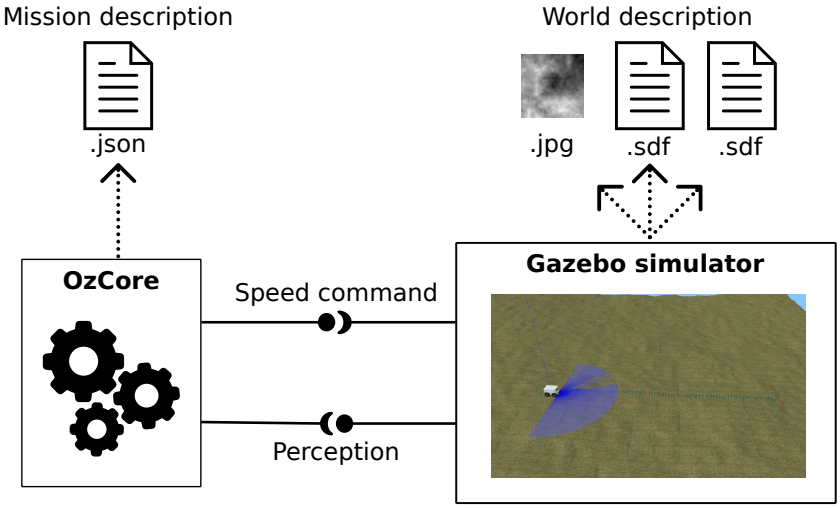

FIGURE 4.2 – Architecture de départ.

sur ce schéma, perçoit le monde virtuel et fournit les commandes en vitesse afin d'effectuer la mission décrite et fournie par un fichier `.json`.

Gazebo se sert de l'image `.jpg` afin de simuler un terrain présentant des irrégularités locales. En effet, contrairement à MORSE utilisé chapitre 2, Gazebo ne propose pas nativement d'outil avancé qui simplifie la génération et la simulation de terrain présentant des irrégularités locales. De manière générale et d'après notre expérience, Gazebo se prête moins que MORSE à de la génération procédurale, du fait que MORSE profite de l'API de Blender. Toutefois, Gazebo peut déformer un plan à partir d'une image en niveau de gris (aussi appelée *carte de profondeur*) afin de simuler un tel terrain. Le simulateur interprète alors les valeurs de chaque pixel de la carte de profondeur comme autant de hauteurs et déforme le maillage d'un plan en fonction de celles-ci. Un exemple de vue du robot dans l'environnement 3D est donné figure 4.3.

Afin de faire du test de manière intensive et dans des environnements plus riches, nous avons procédé à diverses modifications. Ces modifications affectent la fidélité de la physique et la modélisation des légumes.

Dans la simulation fournie par Naïo, les roues du robot sont modélisées avec une haute fidélité physique, se rapprochant des conditions du monde réel. Or, dans le chapitre 3 nous avons vu qu'un grand nombre de fautes ne nécessitent pas une grande fidélité physique afin d'être révélées. De plus, cette modélisation précise implique un grand nombre de ressources et de temps de calcul que nous ne sommes pas en mesure de fournir. Ce modèle avancé a donc été désactivé dans notre environnement de simulation.

Les maillages 3D utilisés par la simulation sont aussi simplifiés afin de présenter moins de sommets, et ainsi économiser des ressources durant la phase de simulation. Un exemple est donné figure 4.4 pour des légumes de type chou.

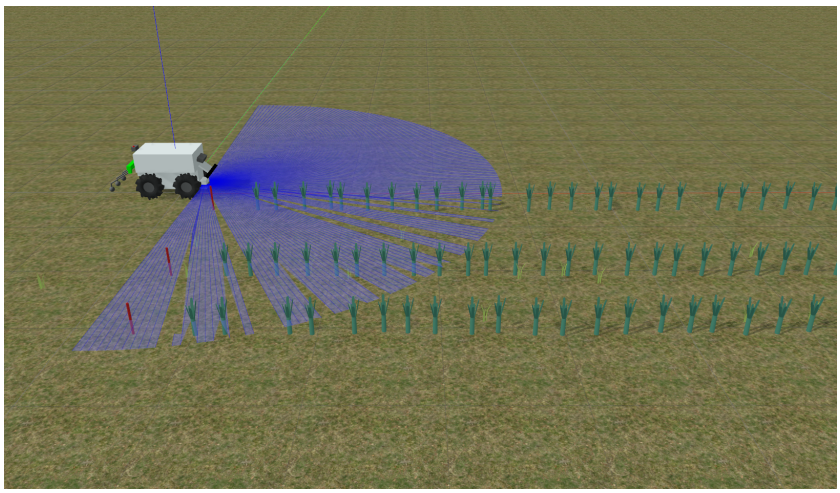

FIGURE 4.3 – La représentation 3D du robot Oz sous Gazebo.

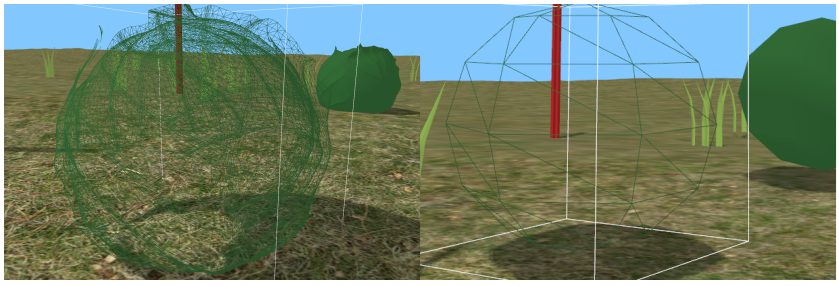

FIGURE 4.4 – Exemple simplification de maillage. À gauche : maillage du chou fourni par Naïo (21081 sommets) ; à droite : maillage du chou simplifié (132 sommets).

Nous avons également modifié la simulation fournie par Naïo au niveau de sa gestion des légumes. Dans la simulation utilisée par l'entreprise, les légumes sont posés sur un terrain plat et sont représentés en 3D, sous la forme d'un solide à base cylindrique intraversable et soumis à la gravité. Dans ce cas, si le robot heurte un légume, ce dernier va se renverser et potentiellement rouler. De plus, cette première simulation ne permet que l'utilisation d'un terrain complètement plat. En effet, si ce dernier présente des irrégularités locales, ou encore une pente, les légumes, qui ne sont plus posés sur une surface plane, vont se renverser et rouler dès le lancement de la simulation. Nous avons donc développé un module logiciel supplémentaire afin de poser un légume sur n'importe quel type de terrain à l'instar des obstacles du chapitre 2.

Partant de cette architecture de simulation de départ, nous construisons une plateforme de test pour Oz, selon les mêmes principes que celle réalisée pour Mana. La plateforme de test permet de générer automatiquement des entrées, de gérer l'exécution de tests, ainsi que de collecter et d'analyser des données d'observation. La génération d'entrées de test est décrite section 4.4. Un modèle de mondes et missions est défini à partir de l'analyse de cas d'utilisation, et implémenté par une arborescence de classes auxquelles sont associées des fonctions de génération. La section 4.5 présente les données d'observation et leur analyse hors ligne par un oracle de test. Cet oracle vérifie un ensemble de propriétés, issues des cinq grandes classes de détecteurs identifiées par l'analyse des fautes de Mana (voir chapitre 3).

4.4 Les entrées

Cette section détaille la méthode utilisée afin de mettre en place la génération automatique des entrées de test. La démarche utilisée est similaire à celle développée dans le chapitre 2 et se découpe en deux étapes :

— définir et modéliser le domaine d'entrée de test ;

— choisir les fonctions de génération à associer aux éléments du domaine d'entrée.

Nous reprenons le principe d'une modélisation du domaine d'entrée basée sur des cas d'utilisation. Par rapport à la méthode développée dans le chapitre 2, la modélisation UML est enrichie par l'ajout d'une grammaire qui précise les règles de production des attributs des classes. Le choix des fonctions de génération dépend des possibilités offertes dans l'environnement considéré. Nous verrons par exemple que la génération de la topologie du terrain en interface avec Gazebo procède différemment d'avec MORSE.

4.4.1 Analyse de cas d'utilisation

Afin de modéliser le domaine d'entrée, nous nous appuyons sur une trentaine de cas d'utilisation fournis par les développeurs du robot Oz sous forme de fichiers de

```
{
"ITK": { "crop_spacing": 0.250000, "crop_width": 0.200000 }, // espacement inter-légumes et largeur des légumes

"inner_tracks_widths": [ { "d" : 1.4 } ],  // espacement inter-rangée

"is_first_uturn_right_side": true,  // vrai si le premier demi-tour est effectué vers la droite du robot
"is_track_side_at_right": true, // vrai si la première rangée à longer se situe à droite du robot

"first_track_outer": true,   // vrai si le robot est situé en dehors de la première rangée
"final_track_outer" : true,  // vrai si le robot finit sa mission en dehors de sa dernière rangée

"two_pass": false,    // vrai si le robot fait, quand cela est possible, deux passages par inter-rangée
"two_pass_threshold": 0.9,   // seuil sur espacement inter-rangée au-delà duquel le robot effectue, si two_pass vaut
    vrai, deux passages
"crop_gap": 0.170000  // espacement voulu entre le robot et les rangées longées

}
```

FIGURE 4.5 – Exemple d'un fichier `json` de mission fourni par Naïo.

simulation. Ces exemples constituent un corpus de mondes virtuels et de missions de désherbage dans ces mondes.

Un monde est constitué d'un terrain sur lequel se trouve un champ de légumes. Un champ est composé d'au moins une rangée de légumes. Une rangée est formée par un ensemble de légumes disposés approximativement en ligne. Deux piquets rouges marquent obligatoirement le début et la fin d'une rangée. Un champ peut présenter des éléments gênants pour le robot (mauvaises herbes). Les légumes peuvent être de deux sortes, des poireaux ou des choux. Le terrain sur lequel sont disposés les poireaux ou les choux peut être plat, ou présenter des irrégularités locales.

Comme expliqué dans la section 4.2, le robot est équipé de caméras qui lui permettent de percevoir les piquets délimitant chaque rangée de légumes. La couleur de l'environnement est donc perçue par le robot et doit être simulée, notamment le rouge des piquets.

Dans notre simulation, nous nous concentrons sur la navigation du robot. Ne sont donc pris en compte que les actionneurs correspondant à la locomotion. Les interactions simulées avec l'environnement sont uniquement : se déplacer sur le terrain et entre les rangées de légumes et éventuellement entrer en collision avec un légume ou un piquet (interaction catastrophique).

Le robot a besoin, afin de procéder au désherbage d'un champ de légumes, de diverses informations concernant ce dernier. Ces informations lui sont transmises à l'aide d'un fichier de mission, dont un exemple est donné figure 4.5. La figure 4.6 donne une vue schématique de la mission exécutée qui correspond aux valeurs des paramètres contenus dans ce fichier. Une fois le fichier de mission dûment rempli et transmis au robot, l'utilisateur n'interagit plus avec ce dernier : le robot effectue alors sa tâche de manière autonome.

Une description complète des paramètres contenus dans les fichiers de mission est donnée en Annexe A.1. Leur étude permet de les classer en trois catégories :

— les paramètres relatifs au champ considéré :
  — l'espacement inter-légumes et la largeur des légumes (paramètre `ITK` dans la figure 4.5) ;
  — l'espacement inter-rangée (paramètre `inner_tracks_widths`) ;

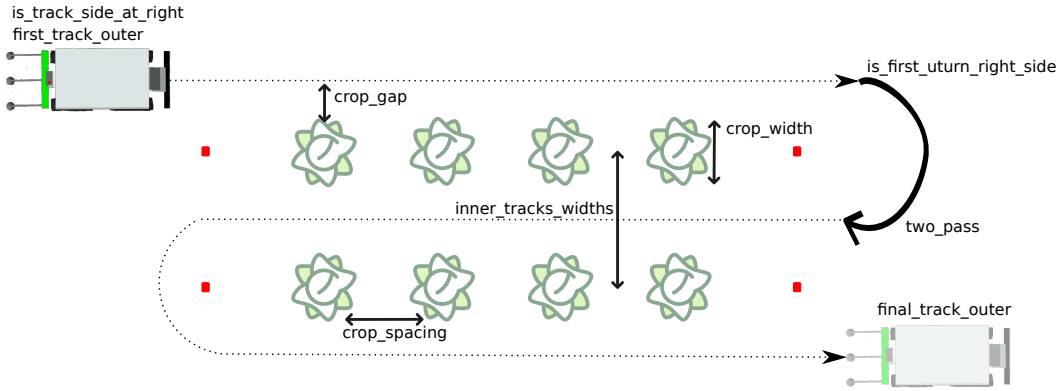

FIGURE 4.6 – Schéma d'une mission en fonction du fichier figure 4.5.

— les paramètres fixes pour configurer le logiciel du robot :
  — Naïo précise que généralement la distance que le robot conserve quand il longe une rangée (paramètre `crop_gap`), est fixée à 10 *cm* ;
  — la valeur du paramètre `two_pass_threshold` dépend quant à elle de la taille du robot utilisé. Si la taille du robot est petite par rapport à l'espacement inter-rangée, le déherbage peut nécessiter deux passages. Pour Oz, le seuil est fixé à 0.9 *m.* ;
— les paramètres relatifs à la mission proprement dite :
  — le paramètre `final_track_outer` décrit si le robot doit finir sa mission en longeant la dernière rangée du champ par l'extérieur (comme dans la figure 4.6). Le paramètre pourra être faux lorsque, par exemple, le champ est dans une serre. Dans le cadre de nos expériences, nous ne considérons que des champs en extérieur, mais laissons la possibilité de mettre ce paramètre à vrai ou faux ;
  — les paramètres `is_first_uturn_right_side`, `first_track_outer` et `is_track_side_at_right` dépendent du placement initial du robot par rapport au champ ;
  — le paramètre restant, `two_pass`, permet d'autoriser deux passages si la largeur inter-rangée le nécessite.

Ce dernier ensemble de paramètres peut être repris pour modéliser les missions. Par contre, dans le cadre d'un modèle complet, il manque les paramètres relatifs au terrain (qui peut être plus ou moins bosselé) et ceux relatifs au champ doivent être enrichis. Par exemple, les espacements inter-rangées et inter-légumes peuvent varier, l'alignement des légumes peut présenter des irrégularités, des légumes ont pu mal pousser induisant des trous dans une rangée, etc.

Une première modélisation a conduit à un ensemble de 31 paramètres de génération, que nous avons finalement réduit à 15 pour nos expériences, après discussion avec Naïo. Des contraintes sont attachées à certains de ces paramètres. Ainsi, les paramètres numériques sont bornés par des valeurs *min* et *max*. D'autres contraintes

lient plusieurs paramètres. Par exemple, la longueur doit être approximativement la même pour toutes les rangées de légumes constituant le champ. Ou encore, si nous avons choisi de générer un champ à N rangées, il faudra générer $N-1$ espacements inter-rangées.

Une telle modélisation, avec un ensemble riche de paramètres que l'on peut souhaiter étendre ou restreindre, nécessite une gestion maîtrisée des configurations de génération. Notre proposition est alors de combiner deux approches de modélisation, respectivement basées sur un diagramme de classes UML et sur une grammaire formelle.

4.4.2 Modélisation : diagramme de classes et grammaire formelle

Le diagramme en figure 4.7 donne une vue structurelle des mondes à générer. Chaque élément du monde est capturé dans une classe. L'agencement de ces classes exhibe une structure arborescente. Comme vu précédemment, une mission est en fait très dépendante des caractéristiques du champ. On regroupe donc sous l'appellation « monde » les trois éléments principaux qui constituent le domaine d'entrée soit : un terrain, un champ et une mission. Le champ est lui-même décomposé en rangées de légumes et en éléments perturbateurs. Ces derniers sont constitués des mauvaises herbes qui peuvent perturber la perception qu'a le robot des rangées de légumes.

La liste des 15 paramètres de génération est détaillée en Annexe A.2. Ils sont distribués dans 5 classes, et apparaissent comme des attributs de ces classes :

— la classe `Field` contient les paramètres généraux du champ : le nombre de rangées et leur écartement ;

— la classe `Crop_row` contient les paramètres spécifiques à une rangée de légumes : sa longueur, l'amplitude du bruit appliqué à l'alignement des légumes, la densité de légumes, la probabilité de légumes manquants et le type de légumes (poireaux ou choux) ;

— la classe `Disturbing_element` contient le paramètre de densité de mauvaises herbes ;

— la classe `Mission` contient les paramètres détaillés section 4.4.1, relatifs aux passages doubles, à la possibilité de longer une rangée par l'extérieur etc. ;

— la classe `Height_map_generator_function` contient les paramètres de génération du terrain irrégulier avec une certaine amplitude de bosselage.

Afin de manipuler et stocker ces paramètres de génération, une grammaire est mise en place. Cela se traduit dans le modèle par l'ajout d'un attribut `descriptor` et de méthodes associées : `random_create()`, `create_from_descriptor(desc)`, `check_descriptor(desc)` et `export()`.

L'attribut `descriptor` contient un mot synthétisant les valeurs des paramètres de génération pour cette classe et ses sous-classes. Ainsi, le descripteur de la classe `World` contient le génotype complet d'un monde, alors que le descripteur de `Field` ne contient que la partie relative au champ de légumes. Le mot stocké dans un descripteur doit satisfaire un ensemble de règles syntaxiques données en figure 4.8,

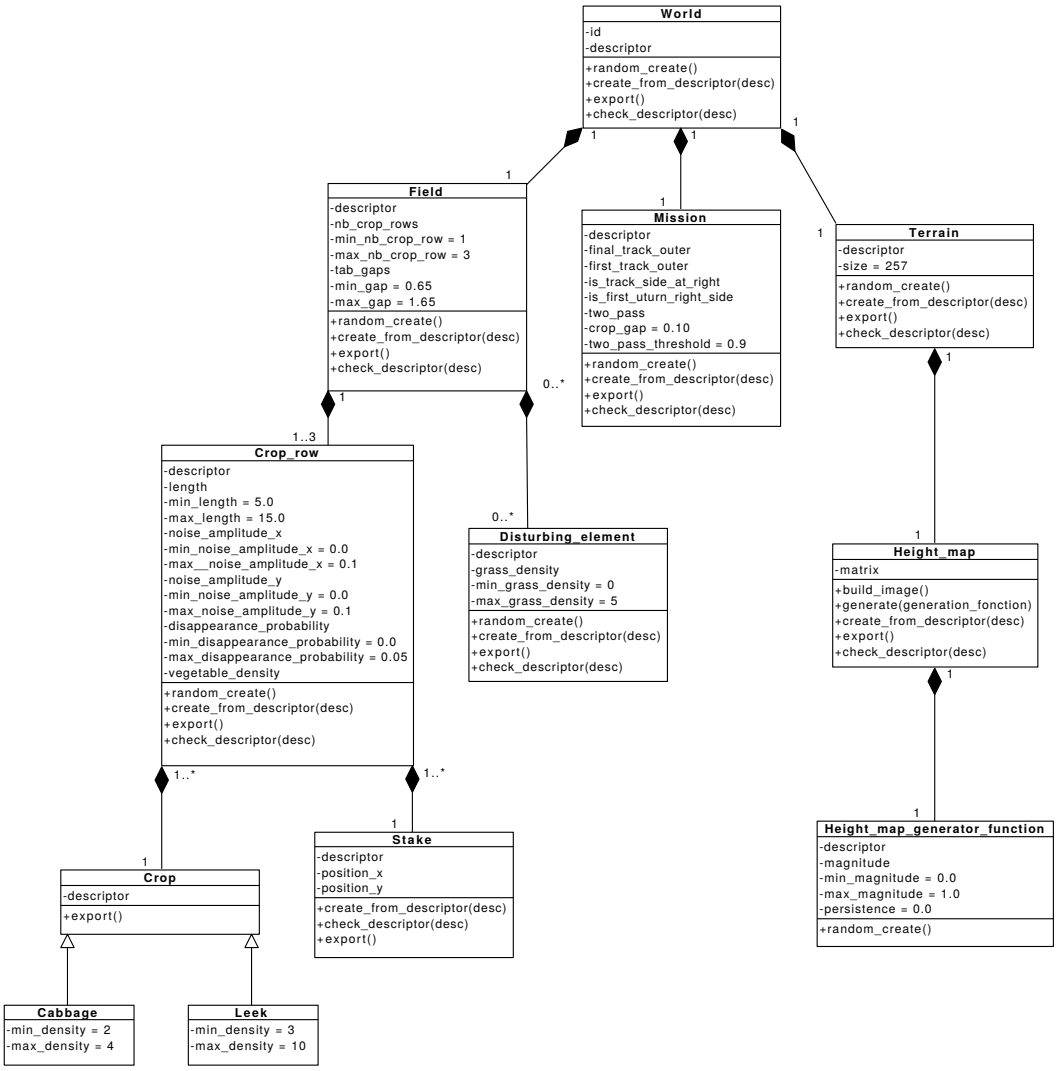

FIGURE 4.7 – Arborescence des classes des entrées sous forme d'un diagramme UML.

auxquelles sont adjointes des règles sémantiques qui correspondent aux contraintes numériques. Par exemple, les règles sémantiques pour le non-terminal *field* imposent notamment que la valeur de *nb_crop_row* soit comprise entre ses bornes minimum et maximum, et que *crop_row* présente le bon nombre de rangées de légumes. La méthode `check_descriptor(desc)` associée à la classe `Field` implémente ces vérifications et détermine si un descripteur de champ de légumes est bien reconnu par la grammaire. Notons qu'il y a une correspondance directe entre la structure définie par les règles syntaxiques (comportant par exemple le non-terminal *field*) et la structure du diagramme de classes (comportant par exemple la classe `Field`). Cela permet de distribuer les règles de production et les vérifications de la grammaire au sein de l'architecture de classes.

La méthode `random_create()` permet de générer un mot aléatoirement, c'est à dire de choisir n'importe quelle configuration valide de paramètres de génération pour la classe concernée et ses sous-classes. Si elle est appelée depuis la classe `World`, `World` commence par s'instancier lui même, c'est-à-dire tirer aléatoirement les valeurs de ses paramètres afin de générer son mot (à partir des méthodes privées correspondantes à ses paramètres), puis demande à `Terrain`, `Field` et `Mission` de faire de même. Ces trois derniers appellent leur méthode `random_create()`, puis ordonnent à leurs sous-classes de faire de même et ainsi de suite jusqu'à ce que toutes les classes concernées aient instancié leurs attributs. On obtient alors un descripteur complet de monde, ne contenant que des éléments terminaux.

La méthode `create_from_descriptor(desc)` offre plus de contrôle sur la génération de mots. Le descripteur passé en paramètre peut être composé de deux types d'éléments : des *terminaux*, fixant la valeur de certains paramètres de génération et des *opérateurs* correspondant à des méthodes prédéfinies pour produire des valeurs de paramètres. Un de ces opérateurs est « r », correspondant à une production aléatoire selon la méthode prédéfinie `random_create()`. Dans le mot, il peut être vu comme un *joker* susceptible d'être remplacé par n'importe quelle chaîne valide de terminaux. En plus de cet opérateur, nous avons mis en place des opérateurs permettant de forcer un attribut à sa valeur minimum « m » ou maximum « M ». Cet ensemble d'opérateurs pourrait bien sûr être étendu. Pour illustrer l'utilisation de `create_from_descriptor(desc)`, supposons que cette méthode soit appelée depuis la racine `World` et considérons différentes possibilités pour le descripteur `desc` passé en paramètre.

D'après les règles syntaxiques de la grammaire (figure 4.8), un descripteur de monde est la concaténation de descripteurs de terrain, mission et champ avec un séparateur « - ». Le contenu de `desc` peut prendre différentes formes :

— `desc` ne comporte que des terminaux. Chaque classe va mettre récursivement à jours ses attributs aux valeurs spécifiées, sous réserve que le contenu de `desc` soit reconnu par la grammaire ;

— `desc` comporte un mélange de terminaux et d'opérateurs, comme « 0.00-r-r ». Dans cet exemple, la configuration de génération récursivement créée doit correspondre à un terrain plat (sous-mot « 0.00 »), et les paramètres de

```
<monde> ::= <terrain> "-" <mission> "-" <field>
<terrain> ::= <height_map>
<height_map> ::= <height_map_generator_function>
<height_map_generator_function> ::= <magnitude>
<magnitude> ::= "0."<digit><digit>
<mission> ::= <two_pass> "+" <final_track_outer> "+" <first_track_outer>
<twoPass> ::= <bool>
<final_track_outer> ::= <bool>
<first_track_outer> ::= <bool>
<field> ::= <nb_crop_row> "+" <gap> "+" <crop_row> "+" <disturbing_element>
<nb_crop_row> ::= "1" | "2" | "3"
<gap> ::= "0.65" | "0.70" | "0.75" | ...| "1.55" | "1.60" | "1.65" | <gap> "+" <gap>
<crop_row> ::= <crop> "␣" <length> "␣" <disappearence_probability> "␣" <noise_amplitude_x> "␣" <noise_amplitude_y> |
    <crop_row> "+" <crop_row>
<crop> ::= "l" | "c"
<length> ::= "5" | "6" | ... | "14" | "15"
<disappearence_probability> ::= "0.01" | "0.02" | "0.03" | "0.04" | "0.05"
<noise_amplitude_x> ::= "0.00" | "0.01" | "0.02" | ... | "0.09" | "0.10"
<noise_amplitude_y> ::= "0.00" | "0.01" | "0.02" | ... | "0.09" | "0.10"
<disturbing_element> ::= <grass_density>
<grass_density> ::= "0" | "1" | "2" | "3" | "4" | "5"
<digit> ::= "0" | "1" | ... | "8" | "9"
<bool> ::= "true" | "false"
```

FIGURE 4.8 – Règles syntaxiques correspondant à notre monde d'entrée.

génération de la mission et du champ sont choisis aléatoirement (« r ») ;

— `desc` ne comporte que des opérateurs, comme simplement « r », ou encore « r-r-M+r+r+r ». Le premier cas est équivalent à un appel direct à `random_create()` : n'importe quelle configuration valide peut être créée. Dans le deuxième cas, tous les paramètres seront instanciés aléatoirement, sauf le nombre de rangées du champ qui doit être pris à sa valeur maximale (« M »).

On le voit, cette approche offre beaucoup de souplesse pour cibler des sous-ensembles du domaine d'entrée. Dans les expériences décrites par la suite, nous n'avons pas exploité toutes ces possibilités : nous avons choisi de procéder uniquement à de la génération aléatoire depuis la racine `World`. Néanmoins, ces possibilités existent et sont utilisables si l'on souhaite faire d'autres campagnes de test.

Après appel de `random_create()` ou `create_from_descriptor(desc)` depuis la classe racine, tous les attributs de génération sont à jour et correspondent à une configuration acceptée par la grammaire. L'appel à `export()` permet alors de finaliser la génération de contenu de monde et de l'exporter dans un format compréhensible par le simulateur. Cet aspect sera présenté dans la section suivante.

Pour conclure sur la modélisation, l'association du diagramme de classes UML et de la grammaire formelle s'avère intéressante tant d'un point de vue conceptuel que pratique. Elle permet de gérer des configurations riches, impliquant de nombreux paramètres qui peuvent être liés par des contraintes. Elle donne de la souplesse pour fixer certains paramètres et en laisser d'autres libres. Le fait d'avoir une approche bien structurée facilite également les évolutions : par exemple les modifications pour ajouter ou retirer un paramètre de génération vont rester confinées à une partie du modèle (les classes et règles de la grammaire directement impactées). Enfin, la manipulation de descripteurs pourra être très utile si l'on souhaite aller vers des stratégies de test basées sur des méta-heuristiques de recherche, qui requièrent de modifier ou combiner des configurations d'entrée existantes pour explorer de nouvelles configurations.

4.4.3 Génération et mise en forme des entrées de test

Une configuration de paramètres de génération correspond à un ensemble de mondes virtuels potentiels. Par exemple, l'amplitude des irrégularités du terrain a été choisie, mais il reste à produire un relief comportant ces irrégularités.

On rappelle que Gazebo a besoin d'une carte de profondeur au format `.jpg` afin de simuler un terrain présentant des irrégularités locales (voir la section 4.3). Cette carte de profondeur, est obtenue à l'aide du bruit de Perlin [Perlin 1985] qui est un algorithme très largement utilisé pour procéder à de la génération procédurale. Il est lui même contrôlé par le paramètre de génération `magnitude` qui définit l'amplitude du bosselage désiré. La classe `Height_map` du modèle d'entrée (voir figure 4.7) est une sous-classe privée qui est introduite afin d'exporter la carte de profondeur. Cette dernière est en effet stockée dans l'attribut `matrix`, qui est une matrice carrée, dont chaque valeur correspond à la valeur d'un pixel.

Les autres fonctions de génération de contenu consistent notamment à déterminer la position de chaque rangée de légumes en fonction de la position initiale du robot, puis de fixer la position exacte de chaque légume en fonction de l'amplitude du bruit sur leur alignement dans la rangée. On fixe également la position des piquets à 50 *cm* de part et d'autre de chaque rangée.

La méthode `export()` de chaque classe prend en charge la phase de génération de contenu et exporte les données dans les formats requis par la plateforme de simulation (les fichiers `.json`, `.jpg` et `.sdf` de la figure 4.2).

Durant la phase de simulation, le simulateur lit les fichiers `.sdf` et instancie le monde virtuel correspondant. Il déforme d'abord le maillage du terrain en fonction de la carte de profondeur dont le chemin sur le disque dur, ainsi que la force avec laquelle il doit appliquer la carte de profondeur, lui sont indiqués dans un premier fichier `.sdf`. Il place ensuite les éléments du monde conformément aux informations lues dans le deuxième fichier `.sdf` en piochant ces éléments dans une bibliothèque d'objets préétablie. L'algorithme de placement des objets s'assure que ceux-ci soient bien posés sur le sol.

4.5 Données d'observation et oracle

Comme pour le chapitre 2, nous considérons un ensemble de données brutes d'observation à collecter, aussi bien prises du point de vue du robot que reflétant la vérité terrain. Ces données sont détaillées dans la section 4.5.1. Nous prenons aussi en compte les leçons du chapitre 3 concernant la mise en place de l'oracle. Celle-ci est détaillée section 4.5.2.

4.5.1 Données d'observation

Nous considérons l'ensemble de données d'observation prises du point de vue du robot suivant :

— les positions perçues capturées à intervalle régulier. Chaque position est horodatée et inclut les valeurs du lacet (correspond à l'angle d'Euler ψ) en plus de la position du robot relativement à un repère de référence décrit par les coordonnées x et y ;

— les états de l'automate interne du robot (dans cette étude ils sont utilisés pour les états d'erreur et de fin de mission).

L'obtention de ces données d'observation est très simple : elle se fait via des fonctions de *log* du robot qui tracent automatiquement et à intervalle régulier ses diverses positions ainsi que ses états internes.

Concernant les données d'observation reflétant la vérité terrain, nous ne considérons que les positions réelles du robot. Ces dernières sont récoltées à l'aide d'un nœud ROS que nous avons implémenté. Durant la phase de simulation ce dernier se connecte à Gazebo afin de récupérer et enregistrer à intervalle régulier dans un fichier texte les positions réelles du centre du robot. Chaque position est horodatée. Elle inclut l'attitude complète du robot décrite en quaternions ainsi que les coordonnées x et y.

4.5.2 Oracle

La mise en place de l'oracle est guidée par les recommandations concernant l'élaboration de détecteurs de comportements défaillants présentés dans le chapitre 3. On rappelle que l'analyse des fautes a permis de définir des catégories que ces détecteurs doivent au moins couvrir : des propriétés associées aux phases de mission, des seuils liés aux mouvements du robot, des événements catastrophiques, des propriétés liées aux rapports d'erreur et des propriétés liées à la perception. Une série de discussions avec Naïo est engagée à partir d'une première version de l'oracle ainsi obtenue. Ces discussions nous ont permis d'affiner notre compréhension du système sous test et de valider l'oracle. La version de l'oracle présentée dans cette section est l'aboutissement de ces échanges.

La table 4.1 présente l'ensemble des propriétés couvertes par l'oracle. Certaines de ces propriétés conditionnent le verdict de l'oracle. Un test entraînera un verdict de rejet si le robot viole une des propriétés P1, P3, P4, P5, P6 ou P8. Dans le cas contraire, un verdict d'acceptation sera formulé. Les autres propriétés (P2, P7) sont considérées comme relatives à la performance et pourront, par exemple, être utilisées dans des futurs tests de non régression. En effet, l'entreprise partenaire ne précise pas d'exigence forte sur la localisation du robot ou sur sa position par rapport à la rangée, tant que le robot n'entre pas en collision avec les piquets ou les légumes (P5) et que celui reste dans un périmètre borné autour du champ (P6). Toute violation est reportée de façon similaire aux propriétés déjà évoquées, mais

Catégorie	Num.	**Propriété**	**Observable**
Phase de mission	P1	Demi-tour en plus de 5 et moins de 7 manœuvres	Positions réelles du robot durant les phases de demi-tour
Phase de mission	P2	Respect de `crop_gap`	Positions réelles du robot durant les phases de travail
Phase de mission	P3	Oz n'effectue pas une autre mission que celle demandée	Positions réelles du robot durant les phases de travail
Seuil lié aux mouvements	P4	Oz maintient sa vitesse en dessous d'un seuil	Positions réelles robot
Événements catastrophiques	P5	Pas de collision	Positions réelles du robot
Événements catastrophiques	P6	Pas de sortie du champ	Positions réelles du robot
Perception	P7	Oz se localise avec une précision n'excédant pas un seuil	Positions perçues et réelles du robot
Messages d'erreur	P8	En cas d'erreur, le robot doit s'arrêter avant un seuil	Positions réelles et états de l'automate interne du robot

TABLE 4.1 – Propriétés de l'oracle par catégories ainsi que les données d'observation associées.

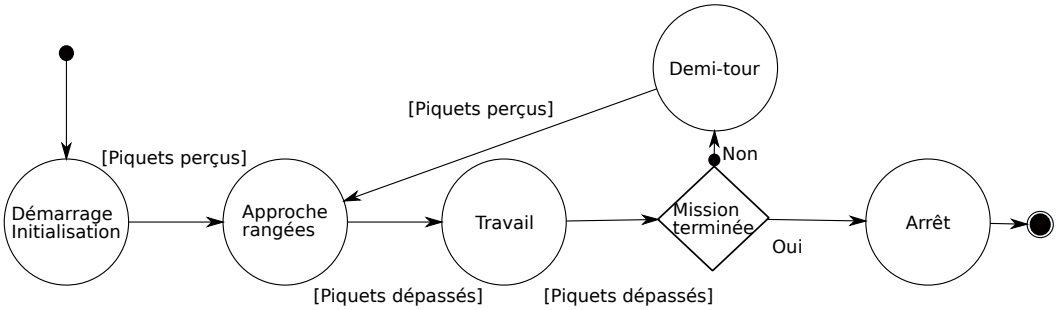

FIGURE 4.9 – Vue sous forme de diagramme d'état transition des phases de mission.

une exécution de mission présentant une violation au niveau de ces propriétés de performance n'est pas sanctionnée par un verdict de rejet.

Un premier ensemble de détecteurs est dérivé des propriétés de l'oracle et implémenté dans la plateforme de test. Bien que ces détecteurs soient d'une granularité fine, leur implémentation n'est pas triviale. Afin de formuler des propriétés associées aux phases de mission, il faut d'abord déterminer celles-ci. Une vue des diverses phases est donnée figure 4.9. Un pré-traitement des positions réelles brutes récoltées est nécessaire à la vérification des propriétés correspondant à des phases de mission. Ce pré-traitement consiste à segmenter ces positions conformément aux phases de mission. Cette segmentation est effectuée en fonction de la zone du champ qui correspond à chaque position. On obtient ainsi, l'ensemble de positions réelles pour chaque phase de mission. Nous détaillons ci-dessous les détecteurs mis en place pour chaque propriété :

P1 Détecter une violation de la propriété P1 nécessite un raisonnement géométrique sur les angles observés durant la phase de demi-tour, sur une abstraction de la trajectoire. Il est possible alors de déduire du nombre d'angles observés le nombre de manœuvres que le robot a effectué afin de faire un demi-tour.

P2 Pour vérifier que P2 est respectée, il faut procéder à un raisonnement logique à partir des paramètres de mission et de la position des rangées, afin d'obtenir la séquence de rangées de travail que le robot va devoir longer. Nous calculons alors les distances entre les positions réelles du robot durant les phases de travail et ces rangées. On peut alors comparer ces distances et `crop_gap`.

P3 Pour vérifier que P3 est respectée, il faut procéder à un raisonnement logique (comparable à celui effectué pour P2) à partir des paramètres de mission et du nombre de rangées, afin d'obtenir la séquence de zones de travail correspondant à la mission attendue. Ensuite, la mission réellement effectuée est obtenue à partir de la comparaison entre les positions réelles du robot durant

ses phases de travail et la position des rangées de légumes [1]. On obtient ainsi deux séquences de zones de travail sous forme de chiffres : une séquence qui représente la mission attendue, et une autre la mission effectuée par le robot (par exemple : « 0, 1, 1, 2 » et « 0, 1, 1 »). On compare alors une à une les zones de travail qui correspondent à la mission réellement effectuée est celles qui correspondent à la mission attendue. Si nous obtenons un couple de deux zones de travail différentes, alors la propriété P3 est violée (c'est le cas par exemple pour les séquences « 0, 1, 2 » et « 0, 1, 1 »). Si la séquence qui correspond à la mission réellement effectuée est plus grande que la séquence qui correspond à la mission attendue, alors P3 est violée (par exemple : « 0, 1 » et « 0, 1, 1 »). Autrement, P3 est respectée.

P4 Lors d'expériences préliminaires, nous avons observé la violation ponctuelle de la propriété P4. Une discussion avec Naïo nous a permis d'en déduire que ces violations étaient les produits d'un artefact de simulation. En effet, Oz n'est pas en roue libre dans le monde réel, mais le frein moteur n'est pas pris en compte dans la simulation. Dès lors, nous avons préféré ne plus prendre en compte cette propriété et l'ignorer.

P5 Une collision avec un légume est détectée lorsque la position du centre de la base d'un légume se trouve sur la surface occupée par le robot. Il en va de même pour détecter d'éventuelles collisions avec les piquets. Notre estimation de collision n'est pas sévère : si le robot touche un légume sans passer pas son centre, la collision n'est pas détectée.

P6 De même, l'estimation des sorties de champ est aussi laxiste : le robot doit dépasser un périmètre autour du champ – pré-calculé à partir des positions des rangées de légumes – avec son centre de gravité pour qu'une sortie de champ soit détectée.

P7 Comme pour P3, les positions issues du simulateur sont enregistrées et comparées à celles perçues par Oz.

P8 Les seuils sont réglés à partir de discussion avec l'entreprise partenaire. Un échec d'arrêt après une erreur (P8) est ainsi effectif si le robot se déplace pendant plus de 50 *cm* malgré le fait qu'il ait détecté une erreur.

4.6 Résultats du test aléatoire

La plateforme de test décrite section 4.3 est utilisée afin de soumettre le robot Oz à une campagne de test. Cette campagne est effectuée à l'aide du même ordinateur personnel utilisé pour les expériences du chapitre 2. Les défaillances sont automatiquement détectées à l'aide des procédures d'oracle détaillées section 4.5. Le système sous test a été confronté à un ensemble de 80 mondes virtuels qui sont

1. En réalité, toutes les positions réelles correspondant à une même phase de travail ne sont pas considérées. Uniquement une seule de celles-ci est sélectionnée au milieu du tableau des positions réelles du robot pour une même phase de travail et est comparée à la position des rangées de légumes.

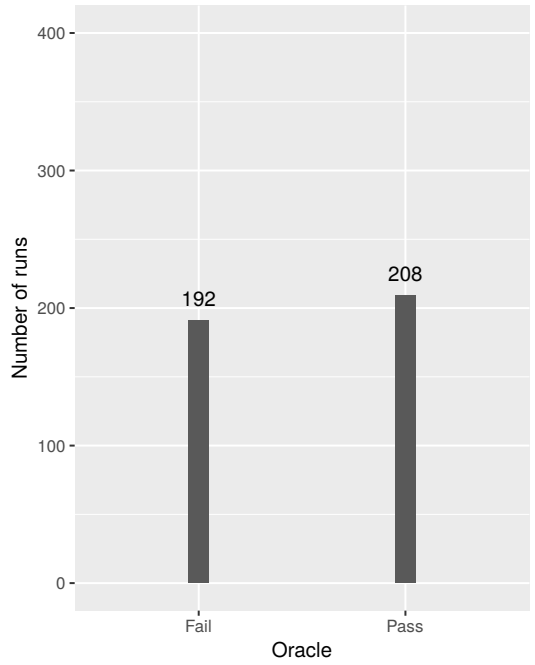

FIGURE 4.10 – Nombre d'exécutions de mission en fonction du verdict de l'oracle. À gauche : *Fail* (verdict de rejet) ; à droite : *Pass* (verdict d'acceptation).

générés aléatoirement. Cette génération s'appuie sur le modèle de monde dont le diagramme est présenté figure 4.7. Nous procédons à 5 exécutions de mission par monde virtuel, soit un total de 400 exécutions. Ces exécutions répétées sur un même monde permettent de prendre en compte l'indéterminisme de la navigation du robot Oz. Dans ces conditions, une campagne de test dure environ 24 heures.

4.6.1 Vue globale

La figure 4.10 présente les résultats de l'oracle sur la campagne de test menée. Cette figure présente le nombre d'exécutions de mission pour lesquelles l'oracle a formulé un verdict de rejet (*Fail*) et celles correspondant à un verdict d'acceptation (*Pass*). 192 exécutions de mission présentent au moins un comportement défaillant, pour 208 exécutions n'en présentant aucun.

La figure 4.11 détaille le nombre de propriétés violées lors des 400 exécutions de mission. Sur cette figure, plusieurs occurrences d'une violation d'une même propriété, pour une même exécution de mission (par exemple plusieurs collisions) ne sont comptabilisées qu'une fois. 35.5 % des exécutions de mission présentent au moins une collision (P5), ce qui fait de la collision la défaillance la plus observée durant la campagne de test. Dans 17.5 % des cas, le robot effectue une mission erronée (P3) et dans environ 14 % des cas le robot outrepasse les limites du champ (P6). Dans de rares occasions (environ 3.5 % des cas), le robot ne s'arrête pas directement après avoir levé une erreur (P8). Aucune violation de la propriété P1 (le

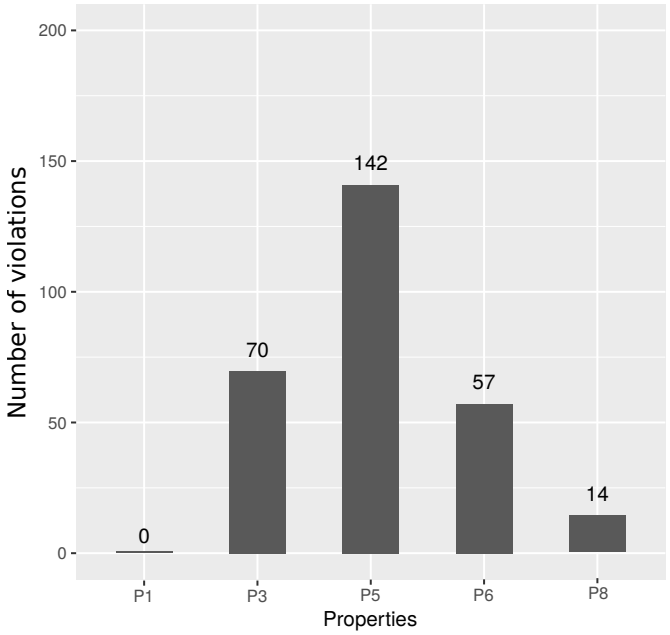

FIGURE 4.11 – Distribution des types de défaillances.

robot effectue son demi-tour en plus de 5 et moins de 7 manœuvres) n'est observée.

Une exécution de mission peut également violer plusieurs propriétés. La figure 4.12 présente un exemple, vu du dessus, de plusieurs violations sur un seul monde virtuel. Sont représentés sur cette figure les piquets rouges (carrées rouges), les rangées de poireaux (cylindres verts), la trajectoire effectuée par le robot (ligne noire), les limites du champ (cadre noir), les collisions (ronds bleus) ainsi que les positions qui sont en dehors des limites du champ (cercles noirs). Le robot est aussi représenté sur cette figure à sa position de départ, sous la forme d'un rectangle bleu. La mission attendue ici consiste en un passage à l'extérieur de la rangée en bas du champ, puis à deux passages entre les deux rangées. Sur cet exemple, le robot effectue une mission erronée (le robot repasse par la même rangée de légumes : P3 est violée), le robot s'arrête à la fin de la rangée et lève une erreur (le robot s'arrête à temps : P8 n'est pas violée). Oz présente une grande difficulté à effectuer son demi-tour, ce qui engendre des collisions après le premier demi-tour (P5 est violée). De plus, le robot va légèrement outrepasser les limites du champ durant son demi-tour (P6 est violée).

Cette campagne de test stresse efficacement le système et provoque un nombre important de défaillances : environ une exécution de mission sur deux présente un verdict de rejet. Ce grand nombre de défaillances est étonnant. En effet, Naïo a déjà procédé manuellement à des tests Oz en simulation, et le logiciel est considéré comme mature. Afin d'apporter une explication à ce résultat surprenant, nous procédons à une autre campagne de test.Celle-ci s'appuie sur un monde nominal choisi parmi le corpus de cas d'utilisation fourni par Naïo.

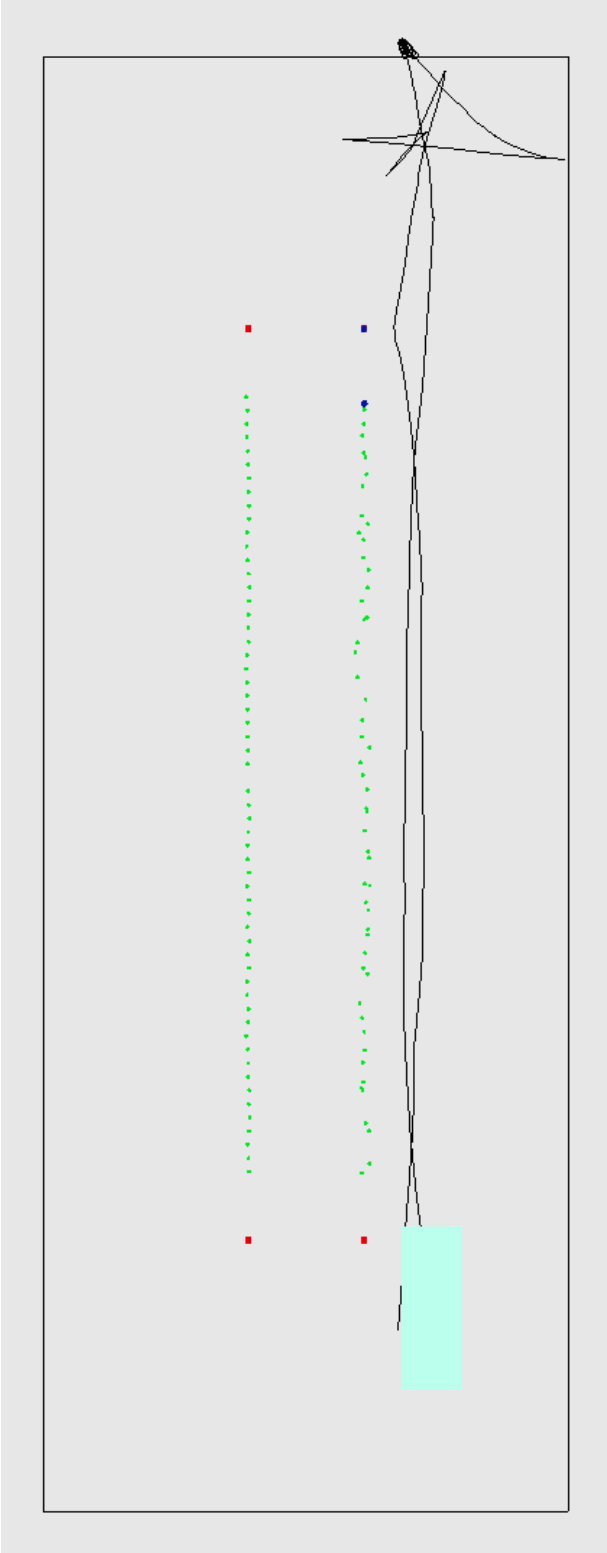

FIGURE 4.12 – Vue du dessus d'une scène Blender reconstituant la trajectoire du robot et les multiples défaillances rencontrées pour un même monde virtuel et une même mission.

FIGURE 4.13 – Vue schématique de la mission attendue correspondant au cas nominal.

4.6.2 Comparaison test nominal et test aléatoire

Naïo a développé une première version de simulation présentée en section 4.3 utilisée pour des tests ponctuels lors du prototypage (mise au point pendant le développement). Ces tests utilisent un ensemble limité de mondes virtuels. La génération de mondes supplémentaires est facilitée par un script simple mais non automatisé, ce qui nécessite la supervision d'un humain. L'entreprise ne procède pas à du test intensif à l'aide de la génération procédurale et certains éléments comme un terrain non plat ne sont pas considérés dans leurs tests (voir discussion sur la gestion de la physique des légumes section 4.3).

Pour tenter de comprendre pourquoi nous observons autant de défaillances durant la campagne de test utilisant la génération aléatoire, une seconde campagne est menée. Cette dernière utilise la même plateforme de test décrite section 4.3. Les mêmes propriétés d'oracle détaillées section 4.5 sont vérifiées pour chaque exécution de mission. Les deux expériences ne diffèrent que du point de vue des entrées de test. Cette campagne consiste à confronter le système sous test à un cas de test nominal non trivial utilisé par l'entreprise partenaire. Il s'agit d'un champ présentant trois rangées sur un terrain plat parsemé de brins d'herbe (voir figure 4.3). La figure 4.13 montre une vue schématique de la mission attendue sur ce cas de test nominal.

La campagne de test nominal procède donc à 400 exécutions d'une même mission sur un même monde virtuel (contre 5 exécutions de mission pour 80 mondes virtuels différents pour la campagne de test aléatoire).

La figure 4.14 présente les résultats de l'oracle pour les deux campagnes (monde nominal et mondes aléatoires) menées avec à chaque fois, le nombre d'exécutions de mission présentant un verdict de rejet (*Fail*) et celles présentant un verdict d'acceptation (*Pass*).

Malgré le fait que le cas nominal ne soit pas un cas trivial (pour rappel trois rangées de légumes et une mission incluant 2 demi-tours), le test aléatoire se montre

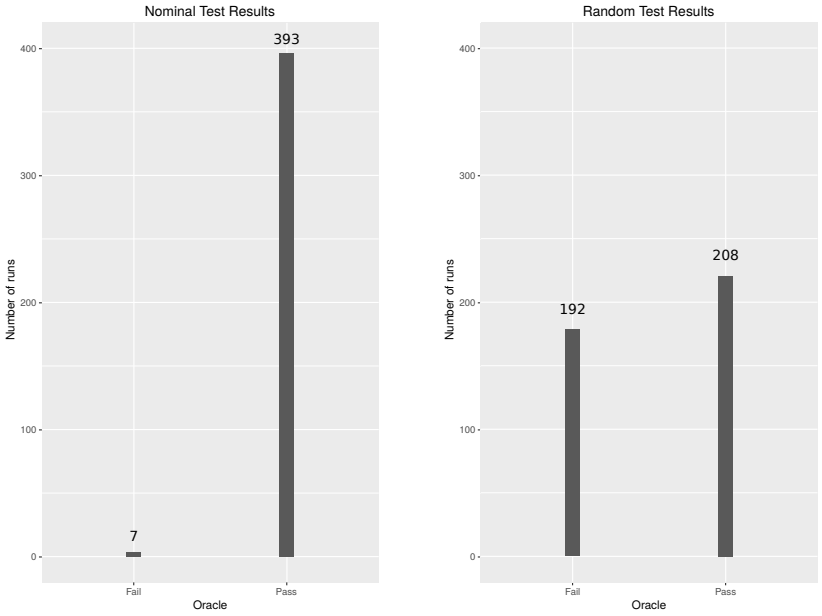

FIGURE 4.14 – Nombre d'exécutions de mission en fonction du verdict de l'oracle pour les deux campagnes de test. À gauche : la campagne de test nominal ; à droite : la campagne de test aléatoire.

nettement plus efficace à déclencher des défaillances. En effet, seulement 7 exécutions de mission (soit 1.75 %) de la campagne nominale présentent un verdict de rejet, contre 192 exécutions de mission de la campagne aléatoire. Cette différence est donc clairement reliée à la richesse du domaine d'entrée utilisé dans la campagne aléatoire. Notamment, le cabossage du terrain semble largement influencer le déclenchement des violations.

Il apparaît que la génération aléatoire est un outil important pour mener des campagnes de test intensif. Cependant, nous avons pu identifier que la spécification du domaine d'entrées (mondes admissibles) est un élément fondamental pour que les résultats des tests soient pertinents. Cette première campagne de test a montré que définir de façon précise la spécification des entrées du test est non trivial pour ce type de mission.

4.6.3 Indéterminisme

Dans les chapitres précédents, nous avons vu que l'indéterminisme était inhérent au comportement des systèmes robotiques, même en simulation. La mission de navigation considérée au chapitre 2 est particulièrement soumise à l'indéterminisme. Bien que dans le cas de Oz, sa trajectoire soit contrainte par les rangées de légumes, l'indéterminisme de cette trajectoire reste observable. Il a de façon évidente un impact direct sur l'exécution des missions et sur la violation des propriétés.

Par exemple, la figure 4.15 présente l'effet de l'indéterminisme pour le robot Oz, avec 2 exécutions sur des mondes/missions similaires. La mission attendue consiste en un passage à l'extérieur de la rangée en bas du champ, puis à deux passages entre les deux rangées. La partie haute de la figure présente une exécution de cette mission durant laquelle le système effectue la mission attendue, la partie basse de la figure présente une exécution durant laquelle Oz effectue la mission de façon erronée. Cette vue simplifiée présente les piquets rouges de début de rangée (carrées rouges), les rangées de poireaux (cylindres verts), la trajectoire effectuée par le robot (ligne noire), les limites du champ (cadre noir), les collisions (ronds bleus) ainsi que les positions qui sont en dehors des limites du champ (cercles noirs). Le robot est représenté sous la forme d'un rectangle bleu. Dans le cas où le robot effectue une mission erronée, le robot s'arrête à la fin de la rangée et lève une erreur. On peut voir que le robot a une grande difficulté à effectuer son demi-tour, ce qui engendre des collisions dans les deux cas après le premier demi-tour. De plus, dans les deux cas le robot va légèrement outrepasser les limites du champ.

On peut ainsi imaginer que pour 2 exécutions sur un même monde/mission, on obtiendrait une exécution sans violation et une autre erronée. Pour étudier cette tendance, nous avons analysé la répétabilité des rejets pour chaque monde généré (avec 5 exécutions pour un même monde).

La figure 4.16 présente le nombre de mondes ayant n exécutions sanctionnées par un verdict de rejet. On remarque que l'ensemble des mondes pour lesquels la totalité des exécutions de mission ont le même verdict (soit acceptation avec $n = 0$, soit rejet avec $n = 5$), représente environ 71 % des cas. Quasiment un tiers des mondes générés provoque un indéterminisme au niveau des verdicts. Cette proportion importante illustre bien que le comportement indéterministe du robot a des conséquences sur le test, et valide le fait qu'il est nécessaire d'effectuer plusieurs exécutions par monde. Une analyse similaire, mais plus détaillée, de l'indéterminisme relatif aux propriétés de l'oracle va confirmer cette première observation.

Afin d'étudier ce phénomène, nous avons également analysé si certaines propriétés étaient violées de façon systématique ou non selon les mondes. Pour cela la figure 4.17, présente les proportions de mondes violant les propriétés P3, P5, P6 et P8, en considérant pour chaque propriété le nombre de rejets sur les 5 exécutions. Par exemple, le diagramme P3 présente le nombre de mondes en fonction du nombre d'exécutions de mission qui violent la propriété P3, sur 22 mondes violant P3 au total. Il en va de même pour les propriétés P5, P6 et P8 avec respectivement 40, 24 et 6 mondes. La propriété P8 (arrêt après message d'erreur) est déclenchée par 6 mondes différents seulement. C'est le détecteur le plus difficile à déclencher. Si l'on regarde de plus près, un seul monde parvient à violer à chaque fois cette propriété, même si, dû au tout petit corpus de mondes disponible, nous ne pouvons parvenir à une observation statistiquement correcte. Cette défaillance était non connue par Naïo. En effet, elle semble difficile à repérer sans l'aide du détecteur adéquat.

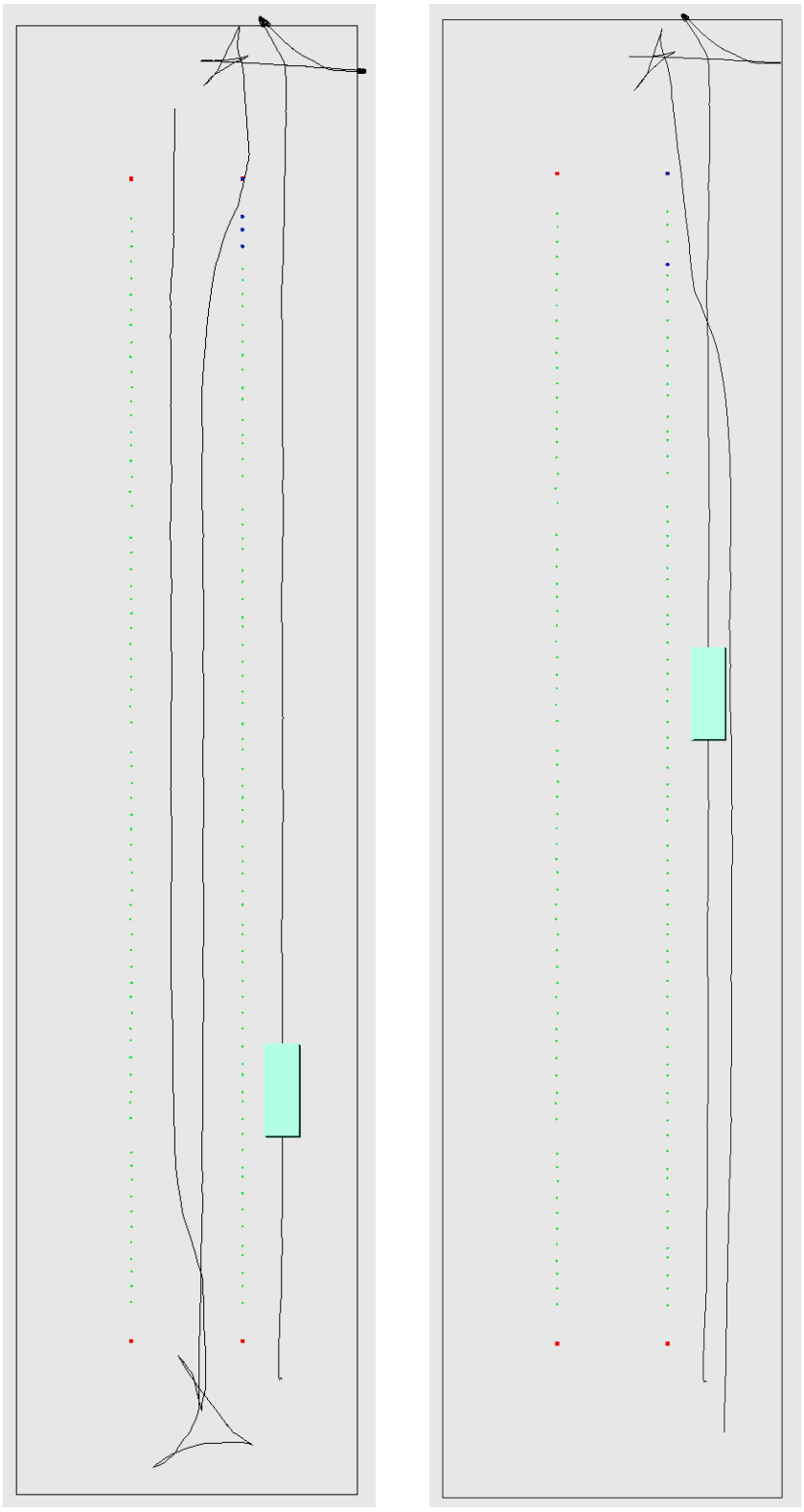

FIGURE 4.15 – Les trajectoires de deux exécutions de mission sur une même carte. À gauche : Oz effectue la mission attendue (P3 n'est pas violée), mais des collisions sont détectées (P5 violée) et le robot sort du champ (P6 est violée). À droite : Oz effectue une mission erronée (P3 violée), P5 et P6 sont également violées.

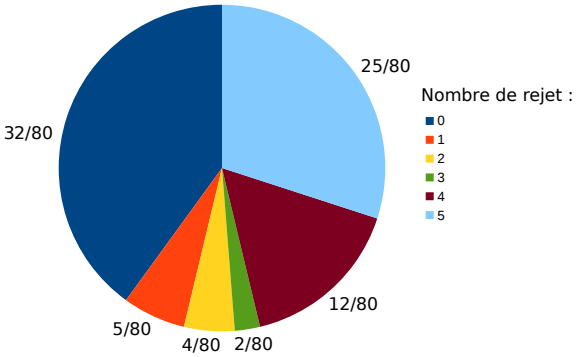

FIGURE 4.16 – Nombre de mondes ayant n verdicts de rejet sur les 5 exécutions sur les 80 mondes au total. Avec $n \in \{0, 1, 2, 3, 4, 5\}$.

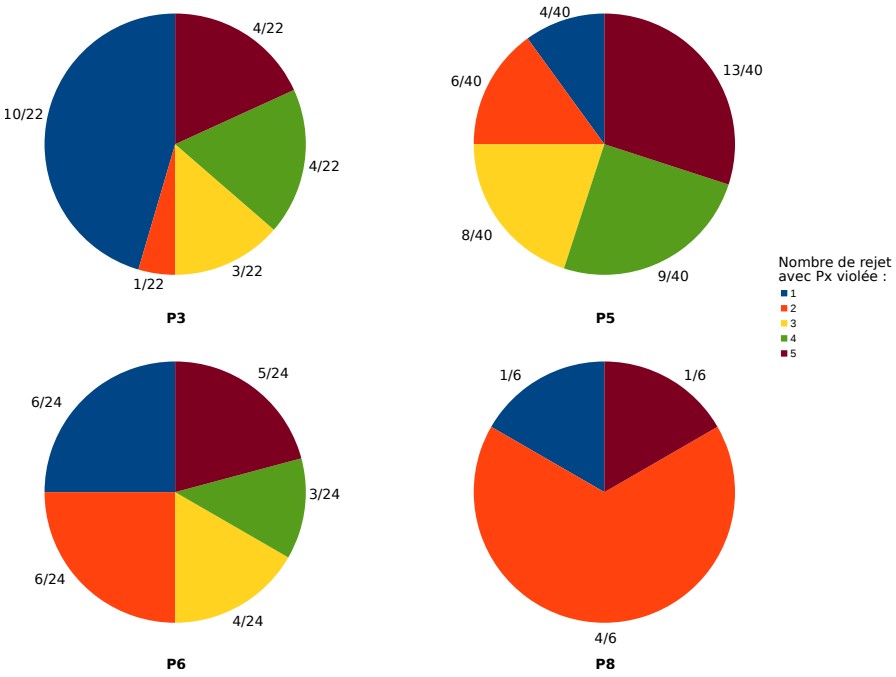

FIGURE 4.17 – Quatre diagrammes circulaires présentant, respectivement pour les propriétés P3, P5, P6 et P8, le nombre de mondes ayant n exécutions de mission violant la propriété considérée, sur le nombre de mondes ayant au moins une exécution de mission qui viole cette même propriété. Avec $n \in \{1, 2, 3, 4, 5\}$.

Concernant les autres propriétés, nous avons un corpus plus important de mondes qui déclenchent les détecteurs associés. Ces violations de propriétés correspondent à des comportements déjà observés par Naïo, mais qui étaient non connus par nous avant la campagne de test.

Pour la propriété P5 (collision), seulement 32.5 % des mondes déclenchent systématiquement le détecteur associé à celle-ci, et 22.5 % des mondes déclenchent le détecteur dans 4 cas sur 5. Le comportement relatif aux propriétés de l'oracle est complètement indéterministe : les violations n'étant pas révélées à coup sûr dans la grande majorité des cas.

Les conséquences sur le test de l'indéterminisme n'avaient pas été observées dans les chapitres précédents. Il aurait été possible de croire que la nature plus restrictive de la mission du robot Oz permettrait d'amoindrir les effets de l'indéterminisme sur le test. Ici, cette campagne de test montre que pour un même monde virtuel et une même mission, une défaillance peut être déclenchée ou non. Ces résultats attestent de la nécessité de procéder à des exécutions de mission répétées sur un même monde virtuel, même si celui-ci n'est pas un monde ouvert.

4.6.4 Aide au diagnostic

L'étude des traces des 192 exécutions de mission correspondant à au moins une violation représente une charge importante de travail. En effet, il s'agit d'identifier les causes des violations, qui peuvent être issues de :

— fautes au niveau du logiciel Ozcore ;
— fautes dans les fichiers de configuration de l'environnement de test ;
— spécification hors des limites acceptables des entrées de test ;
— effet de bord de l'utilisation des ressources logicielles par le système de simulation.

L'objectif de cette thèse n'est pas de poursuivre les investigations pour chacune des propriétés violées mais de fournir une approche pour aider cette tâche. Nous avons donc catégorisé les contextes dans lesquels ces violations de propriété apparaissaient. Pour cela, nous avons choisi de nous appuyer principalement sur les détecteurs de l'oracle et de considérer chacune des violations de manière séparée. Pour les raisons évoquées précédemment nous nous intéressons ici aux propriétés P3, P5, P6 et P8.

P3 Pour la propriété P3 (erreur dans la mission effectuée), nous avons observé 76 erreurs dans la mission sur 22 mondes présentant au moins une exécution de mission violant la propriété P3. Nous avons regroupé les défaillances observées en fonction des zones de travail par lesquelles Oz passe de manière erronée ci dessous :

$P3MD1$:	Le robot repasse par la même zone de travail	77.6 %
$P3MD2$:	Le robot repasse par une ancienne zone de travail	10.5 %
$P3MD3$:	Le robot saute une zone de travail	11.8 %

Une zone de travail correspond à une zone délimitée par les piquets rouges, en dehors ou entre deux rangées de légumes. Par exemple, dans la figure 4.12, nous pouvons voir trois zones de travail : une en dessous du champ par où le robot passe, une située entre les deux dernières rangées et une dernière située au-dessus du champ.

L'analyse des traces nous montre que Oz peut effectuer une mission erronée de trois manières différentes : en repassant par la zone de travail qu'il vient de longer, en repassant par une ancienne zone de travail ou encore en sautant une zone de travail. Le comportement le plus représenté (environ 77.6 %) dans notre corpus d'exécution de mission est celui qui implique que le robot repasse par la même zone de travail. Cette observation illustre que l'analyse et la correction du code correspondant au demi-tour pour un changement de rangée doit être faite en priorité par les développeurs.

P5 Pour la propriété P5 (collisions), les résultats sont statistiquement significatifs car 40 mondes présentent au moins une exécution de mission violant P5, pour un total de 213 collisions. On rappelle que, si une seule collision suffit à violer la propriété P5 pour une exécution de mission, il peut y avoir plusieurs collisions pour une même exécution de mission. Cela est valable pour toutes les propriétés que nous étudions (sauf pour la propriété P8). Les collisions sont regroupées en fonction de la phase de le mission durant lesquelles elles ont eu lieu et la localisation approximative par rapport à la rangée comme présenté ci-dessous :

$P5MD1$:	Après le demi-tour en début de rangée	64.4 %
$P5MD2$:	Pendant le demi-tour en début de rangée	12.7 %
$P5MD3$:	Pendant la phase de travail en fin de rangée	7.5 %
$P5MD4$:	Pendant la phase de travail en milieu de rangée	15.5 %

Nous remarquons tout d'abord que la grande majorité des collisions, soit environ 77 %, ont lieu juste après ou pendant un demi-tour et en début de rangée. Comme pour P3, cela illustre que le demi-tour semble être l'action la plus délicate que le robot Oz effectue. Cependant, environ 23 % des collisions ont lieu durant la phase de travail. Après une collision en milieu de rangée, Oz peut ou non traverser la rangée concernée. Dans certains cas observés après avoir traversé la rangée, le robot s'arrête au bout de quelques mètres et lève un message d'erreur (*Blind course guard triggered* : ce message est levé quand le robot estime avoir parcouru 3 mètres alors qu'il a perdu la localisation des rangées de légumes). En analysant la position du robot 3 mètres en amont de la position correspondant au message d'erreur, nous avons diagnostiqué que le robot perdait la localisation de la rangée avant d'entrer en collision avec celle-ci. Dans d'autres cas, le robot arrive en fin de la rangée préalablement

traversée avec collision. Il s'arrête alors soit en levant une erreur (*Invalid markers spacing*), soit décrète qu'il a correctement fini sa mission (ce qui n'est pas vrai puisqu'il n'est pas dans la bonne zone de travail). Nous considérons ici, que la collision serait due à une perte de localisation, mais plus temporaire que dans le cas précédant, le robot ayant réussi à se « raccrocher » à une rangée après sa collision. Nous n'avons pas poussé plus loin les investigations, mais le problème de perte de la localisation des rangées est un élément important à étudier par les concepteurs.

P6 L'étude des violations de la propriété P6 (Oz outrepasse les limites du champ), permet de réaliser trois groupes à partir des 66 sorties du champ sur les 24 mondes dont au moins une exécution de mission viole la propriété P6. Les contextes dans lesquels cette propriété est violée sont présentés ci-dessous :

$P6MD1$:	Après demi-tour	10.6 %
$P6MD2$:	Pendant demi-tour	86.4 %
$P6MD3$:	Fin de rangée	3.0 %

On retrouve ici le problème majeur du demi-tour. Le nombre réduit de sortie en fin de rangée est également à étudier, dans un second temps, car c'est un risque majeur pour le système (notamment si le robot sort de la parcelle pour se placer sur une route par exemple).

P8 La campagne de test aléatoire montre que lorsque la propriété P8 est violée (le robot ne s'arrête pas en cas d'erreur), le robot peut être dans l'état *RecoverableErrorState* ou *FatalErrorState* et émettre des messages d'erreur comme présenté ci-dessous :

$P8MD1$:	RecoverableErrorState/Blind course guard triggered	50 %
$P8MD2$:	RecoverableErrorState/Steersman guard triggered	7.1 %
$P8MD3$:	FatalErrorState/Invalid markers spacing	42.9 %

— le message *Blind course guard triggered* signifie que le robot a perdu la localisation des rangées de légumes et s'est déplacé de plus de 3 m à l'aveugle ;

— si le robot a une orientation qui fait un angle de plus de 15° par rapport à la ligne droite formée par les 10 derniers mètres parcourus, il se met en sécurité et enregistre alors le message d'erreur *Steersman guard triggered* ;

— le message *Invalid markers spacing* est levé quand le robot perçoit que les piquets rouges de droite et de gauche ne sont pas alignés l'un avec l'autre ou ne présentent pas le bon écartement.

Cette distribution est donnée à titre indicatif, en effet, sur notre campagne de test seulement 14 exécutions de mission violent P8 sur 6 mondes virtuels différents. À ce niveau, nous n'avons pu diagnostiquer les causes et les résultats ont été transmis à l'entreprise partenaire qui étudie les cas.

4.6.5 Corrélations avec paramètres de génération

Afin de déduire quelles valeurs des paramètres de génération influent sur le verdict de l'oracle, nous calculons les corrélations entre le verdict de l'oracle et les paramètres de génération de mondes. La figure 4.18 présente les valeurs des coefficients de corrélation de Pearson [2] obtenus en utilisant la fonction `rcorr` [3] du langage R. Les coefficients ont une valeur comprise entre -1 et 1. Un coefficient

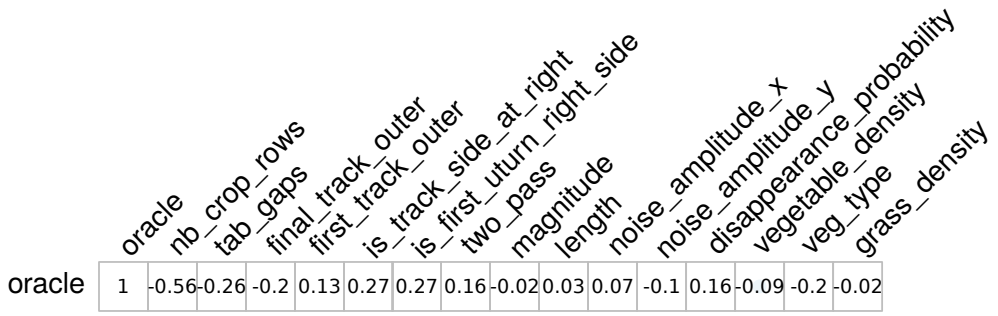

FIGURE 4.18 – Corrélations entre le verdict de l'oracle et les paramètres de génération.

compris entre -1 et 0 indique une corrélation négative, qui sera d'autant plus forte qu'elle est proche de -1 ; de la même manière un coefficient compris entre 0 et 1 indique une corrélation positive, qui sera d'autant plus forte qu'elle est proche de 1. Si une corrélation positive lie un paramètre de génération donné et le verdict de l'oracle, alors plus la valeur du paramètre sera forte, plus l'oracle aura de chance d'être une acceptation. De la même manière, si une corrélation négative lie un paramètre de génération donné et le verdict de l'oracle, alors plus la valeur du paramètre sera forte, plus l'oracle aura de chance d'être un rejet.

Le calcul des corrélations nécessite que certaines valeurs des paramètres de génération et du verdict de l'oracle soient converties. Par exemple, le paramètre `veg_type` vaut 1 pour *poireau* et 0 pour *chou*. De manière générale, la valeur 0 est choisie pour représenter le verdict de rejet ou la valeur booléenne `faux` et la valeur 1 est choisie pour représenter un verdict d'acceptation ou la valeur

2. Pour de plus amples détails sur la corrélation de Pearson : https ://libguides.library.kent.edu/SPSS/PearsonCorr

3. Bibliothèque Hmisc, documentation : `https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf`

booléenne `vrai`. Les paramètres de génération `tab_gaps`, `noise_amplitude_x`, `noise_amplitude_y`, `disappearance_probability` et `grass_density` correspondent à des ensembles de valeurs pour un même monde virtuel (à partir du moment où ce dernier présente plusieurs rangées de légumes). Nous calculons donc la moyenne de ces paramètres pour un monde virtuel avant de calculer la corrélation entre cette moyenne et le verdict de l'oracle.

Nous avons relevé que les corrélations les plus importantes (`nb_crop_rows`, `tab_gaps`, `is_track_side_at_right` et `is_uturn_right_side`) correspondaient à des tendances que nous avions observées de façon empirique. Nous en présentons une analyse préliminaire ci-dessous.

Tout d'abord, le verdict de l'oracle est impacté par `nb_crop_rows` avec une corrélation négative. Cela vient du fait que l'augmentation du nombre de rangées de légumes dans un monde virtuel, augmente le nombre de demi-tours que doit réaliser le robot Oz. Or comme nous l'avons vu dans la section précédente, les demi-tours sont la source de beaucoup d'échecs. Il s'agit donc d'un résultat déjà observé précédemment.

La corrélation entre `is_track_side_at_right` (ou `is_first_uturn_right_side` qui a systématiquement la même valeur) et le verdict de l'oracle est en revanche nouvelle. Cette corrélation positive indique qu'une exécution de mission a plus de chance d'être sanctionnée par un verdict de rejet si le robot a le champ situé sur sa gauche. Ce défaut a été observé par les concepteurs, et corrigé depuis notre étude dans une nouvelle version du contrôleur.

Pour les autres corrélations présentées sur cette figure, il peut être remarqué, par exemple, que plus les écarts de rangées d'un monde virtuel sont élevés (paramètre `tab_gaps`), plus le verdict a de chances d'être un rejet. Ou encore le type de légumes impacte le système sous test : si les légumes sont des choux (légumes plus larges que des poireaux), alors le verdict de l'oracle a plus de chances d'être un rejet. Les informations liées à la non corrélation sont aussi une source d'information. Pas exemple le fait que la densité d'herbe (`grass_density`) ou les déformations du terrain (paramètre `magnitude`) ne semblent pas voir d'impact sur le verdict de l'oracle est contre intuitif et requiert donc une étude approfondie pour déterminer si cela provient du calcul de corrélation lui même, ou de la manière dont sont gérés ces paramètres dans la simulation. L'objectif de cette thèse n'étant pas de réaliser et corriger le contrôleur du robot Oz, nous avons donc transmis ces informations aux concepteurs du robot Oz de l'entreprise partenaire.

4.6.6 Recommandations issues des tests

Outre le fait que nous avons fourni un ensemble de mondes dans lesquels le robot violait des propriétés, nous résumons ci-dessous les principales recommandations des analyses préliminaires que nous avons pu mener :

— en premier lieu, la logique d'arrêt du robot lorsqu'un état d'erreur est levé par le contrôleur de robot est à corriger. En effet, les distances d'arrêt provoquent des collisions ou des sorties de champs, alors qu'une erreur était levée ;

— l'analyse jointe des violations de P5, P3 et P6, ainsi que l'étude des corrélations entre les paramètres d'entrée et l'oracle soulignent la criticité des demi-tours. Beaucoup de violations dépendent directement de l'orientation et de la position que le robot obtient à la suite d'un demi-tour. En effet, une mauvaise orientation peut avoir pour conséquences catastrophiques : des collisions avec les piquets et/ou les légumes en début de rangée, une sortie du champ ou encore de diriger le robot vers une zone de travail incorrecte. De plus, certains demi-tours sont parfois effectués avec une trop grande amplitude ce qui induit, durant le demi-tour, des sorties de champ ou encore des collisions avec les piquets. Les marges entre les piquets et la limite du terrain fixées au début de notre étude par l'entreprise ne sont donc pas adaptées au robot. Il est donc nécessaire soit de reprendre l'algorithme calculant les demi-tours en prenant en compte la distance à la limite du champ et aux piquets, soit d'augmenter les marges (cela revient, pour un même champ délimité à réduire la longueur des rangées) ;

— l'analyse des violations de P5 souligne la responsabilité de la perte de localisation, par le robot, des rangées de légumes. Nous pensons que cette perte de localisation peut être aussi impliquée dans la mauvaise orientation du robot après un demi-tour.

4.7 Conclusion

Ce chapitre présente une campagne de test effectuée sur un robot autonome destiné à l'agriculture : le robot Oz. La plateforme de test présentée au chapitre 2 est modifiée, en collaboration avec le partenaire industriel Naïo, afin de gérer le déroulement du test pour ce nouveau système.

Les enseignements tirés du cas d'étude Mana dans les chapitres précédents sont utilisés pour ce nouveau système. Un large domaine d'entrée est spécifié, modélisé en une structure arborescente et des fonctions de génération sont choisies afin de procéder à de la génération procédurale d'entrée. L'approche utilisée permet de gérer les paramètres de génération de façon structurée à l'aide d'une grammaire, en donnant de la flexibilité pour étendre le modèle et de générer des mondes en fixant certains paramètres pour réduire l'espace d'entrée.

La simulation fournie par Naïo, utilisant le simulateur Gazebo, est modifiée afin de soumettre le système à du test dans des environnements plus riches. Ces modifications touchent la fidélité physique des roues ainsi que la gestion des légumes composant le champ. Des procédures d'oracle, ainsi que les données collectées sur lesquelles elles s'appuient, sont mises en place conformément aux enseignements issus de l'étude de la reproductibilité de fautes en simulation présentée chapitre 3.

Les résultats obtenus lors d'une campagne de test montrent l'efficacité de la génération aléatoire ainsi que des procédures d'oracle mises en place pour révéler un nombre important de défaillances. Les détecteurs implémentés permettent aussi de guider l'analyse des résultats du test, qui sans cela, serait plus coûteuse. L'étude

approfondie des défaillances observées permet de catégoriser celles-ci afin de guider les développeurs du robot dans leur diagnostic. Cette étude révèle notamment la difficulté qu'a le robot à maîtriser ses demi-tours et la nécessité d'étudier sa localisation par rapport aux rangées.

Ce dernier chapitre, ainsi que les retours positifs de l'entreprise, atteste du passage à l'échelle de notre démarche, et de la pertinence de celle-ci dans un cadre industriel.

Conclusion générale et perspectives

En conclusion de ce manuscrit dédié au test de systèmes autonomes en simulation, nous faisons le bilan de nos travaux et de leurs principales contributions, avant d'ouvrir sur des perspectives possibles dans le prolongement de nos recherches.

Bilan

Avec le déploiement de plus en plus important de systèmes autonomes dans des environnements variés, non structurés et à proximité de l'homme, un des défis majeurs est de justifier la confiance que l'on peut placer dans ces systèmes. Le test en tant que technique d'élimination des fautes contribue à augmenter cette confiance. La simulation permet de tester un système autonome pour un coût moindre, un danger nul et d'explorer un ensemble d'environnements plus important que ne le permet le test en monde réel.

L'étude exploratoire présentée dans ce manuscrit permet d'aborder les problématiques principales du test, c'est-à-dire la génération des entrées de test (mondes et missions) et la gestion de l'oracle (propriétés observées et verdicts). Pour mener ces réflexions nous nous sommes appuyés sur les robots Mana et Oz, ainsi que sur les simulateurs Morse et Gazebo.

Le premier constat que nous avons fait dans le chapitre 1, est que cette problématique est encore peu explorée. En effet très peu de travaux allient à la fois une simulation 3D complète et des techniques de tests appliquées à la robotique autonome. Nous avons également conclu que pour faire du test dans ce contexte il était fondamental de pouvoir utiliser la génération procédurale de monde comme cela est utilisé dans certains jeux vidéos.

Le deuxième chapitre a permis de nous confronter aux problématiques principales de la mise en œuvre de ce type de tests en simulation et nous avons identifié qu'un modèle de monde était nécessaire, et qu'il était possible de définir des niveaux de difficulté. Nous avons également mesuré et quantifié l'indéterminisme dans le suivi de trajectoire.

Le troisième chapitre qui s'est concentré sur l'analyse de fautes et de leur correctif, a permis de conclure que la plupart des fautes corrigées sur 10 ans de développement étaient reproductibles en simulation sous conditions. Ces dernières étant en réalité un ensemble de recommandations que nous avons formulées pour la génération de mondes, pour l'expression et l'évaluation de l'oracle ainsi que pour l'environnement de simulation.

Dans le dernier chapitre, nous avons montré comment mettre en œuvre ces différents aspects du test en simulation pour le cas d'un robot agricole pour le désherbage. La complexité des mondes nous a amené à proposer une approche de généra-

tion de mondes basée sur une grammaire et une implémentation orientée objet. Le test aléatoire réalisé a permis d'illustrer que de nombreuses déviations pouvaient être observées en comparaison au test dans des mondes prédéfinis. Les résultats sont actuellement analysés par les développeurs de l'entreprise Naïo, qui ont déjà intégré nos remarques et produit de nouvelles versions du logiciel de contrôle d'Oz. Cette expérience nous a également permis d'observer les conséquences sur l'oracle de test de l'indéterminisme.

Le travail présenté dans cette thèse a donc permis d'établir pour ce nouveau thème de recherche un ensemble d'observations et de recommandations qui pourront être utilisées pour du test en simulation de systèmes autonomes. En effet, nous nous sommes focalisés ici sur la navigation, mais tout mécanisme décisionnel pourrait être mis dans ce type de boucle de test en simulation (par exemple un planificateur de tâches). Le test aléatoire tel que nous l'avons défini et expérimenté est une première manière de faire mais qui doit être évidemment étendue par d'autres techniques de test.

Contributions principales

Les contributions majeures de ces travaux sont les suivantes :

— l'identification et l'illustration des problématiques majeures liées au test en simulation de systèmes autonomes (notamment l'indéterminisme) ;

— une approche pour définir des niveaux de difficulté de monde (et qui intègre des mesures d'indéterminisme de suivi de trajectoire et de tortuosité) ;

— des recommandations pour la mise en œuvre de tests en simulation ;

— une approche de génération de monde basée sur une grammaire et sur une implémentation objet ;

— une étude sur un cas industriel qui montre les bénéfices et les limites du test en simulation.

Perspectives de recherche

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

1. Les modèles de mondes que nous avons développés se concentrent sur les aspects statiques des environnements des systèmes sous test. Nous avons aussi vu que l'intégration d'objets dynamiques (par exemple des obstacles mobiles) est compliquée par la nature indéterministe des systèmes sous test considérés. En effet, il est difficile de prédire la trajectoire que prendra le système sous test, et donc de placer l'obstacle dynamique afin que le système le rencontre. Du fait de cet indéterminisme, instancier un ou plusieurs obstacles mobiles dans une scène de manière aléatoire n'est donc pas suffisant. Une perspective serait donc de définir une stratégie de test intégrant ces objets dynamiques

dans un modèle d'entrée. Il serait intéressant de définir, d'implémenter et d'étudier des procédures de génération à la volée d'obstacles mobiles.

2. Nous avons vu dans le chapitre 2 qu'il est possible, à partir de paramètres de génération de haut niveau, de définir des plages de valeurs correspondant à divers niveaux de difficulté. Nous avons réutilisé cette approche pour le test aléatoire. Une prochaine étape doit consister à guider grâce à des heuristiques, la génération et la sélection de mondes.
3. La mise en place de techniques de sélection mentionnées ci-dessus se heurte au fait que pour converger vers une solution, une méta-heuristique a besoin d'évaluer beaucoup de cas. Or une simulation au niveau système, telles que celles utilisées dans nos travaux, dure souvent 2 ou 3 minutes, ce qui est très long. De plus, dû à l'indéterminisme inhérent aux systèmes autonomes, il faut plusieurs exécutions de mission sur un même monde pour pouvoir correctement évaluer un cas donné. La question de l'accélération des simulations se pose, et face à un logiciel multi-processus communiquant avec un simulateur possédant sa propre horloge, c'est un problème encore ouvert.
4. Dans le chapitre 4, l'analyse des résultats du test, bien que guidée par l'oracle, se fait manuellement. Une perspective stimulante serait d'investiguer la possibilité d'automatiser le plus possible le dépouillement de tels résultats. Pour ce faire, il est possible de s'inspirer des travaux concernant la comparaison de trajectoires [Pelekis 2007] par exemple, ou encore chercher une représentation des résultats (par exemple, sous forme d'images) qui permettrait de les classifier automatiquement.
5. Dans ce manuscrit, nous nous sommes concentrés sur la navigation robotique dans une boucle perception/décision/action. De ce fait, le modèle de mission correspond à une mission de navigation dans un environnement donné. Des travaux sont nécessaires pour évaluer si notre approche est généralisable pour des missions impliquant d'autres types de missions autonomes (par exemple, trouver et prendre des objets).

Annexe A

Annexe

A.1 Paramètres du fichier de mission du robot Oz

Le robot Oz utilise, afin de configurer sa mission, les paramètres suivants :

— `ITK` décrit l'espacement entre les légumes plantés ainsi que la largeur de ces derniers ;

— `inner_tracks_widths` décrit l'espacement entre deux rangées consécutives. Ce paramètre contient un tableau qui a autant de valeurs qu'il y a de zones inter-rangées ;

— `is_first_uturn_right_side` est une variable booléenne qui vaut `vrai` si le premier demi-tour en bout de rangée est effectué vers la droite du robot. Si `first_track_outer` vaut `faux`, alors le robot démarre en face d'une inter-rangée (c'est-à-dire entre deux rangées). Dans ce cas, `is_first_uturn_right_side` sert au robot à savoir si la prochaine rangée est sur sa droite ou sur sa gauche et donc éventuellement de choisir laquelle des deux rangées il doit longer en premier ;

— `is_track_side_at_right` est une variable booléenne qui vaut `vrai` si la première rangée à longer est sur la droite du robot. Oz ne tient pas compte de ce paramètre s'il est situé entre deux rangées de légumes (c'est-à-dire quand le paramètre `first_track_outer` vaut `faux`). Dans ce cas, soit Oz passe au milieu de la rangée, soit il longe une des deux rangées de légumes situées immédiatement à sa droite ou sa gauche. Le choix est notamment conditionné par le paramètre `two_pass` décrit plus bas ;

— `first_track_outer` est une variable booléenne qui décrit la position du robot au démarrage de la mission. Si `first_track_outer` vaut `vrai`, alors le robot est dit « en dehors de la rangée », c'est-à-dire qu'il n'est pas en face d'une inter-rangée. Dans ce cas, la première rangée à longer se situe à la droite ou la gauche du robot. Le robot déduit la position de la rangée à longer de la valeur de `is_track_side_at_right`. Si `first_track_outer` vaut `faux`, alors le robot est positionné en face d'une inter-rangée ;

— `final_track_outer` est une variable booléenne qui décrit la position du robot relativement à la fin de la mission. Si `final_track_outer` vaut `vrai` c'est que le robot doit, à la fin de sa mission, se trouver en dehors du champ. Dans ce cas, la dernière rangée se trouve à la droite ou à la gauche du robot et, si le champ présente plus d'une rangée de légumes, il n'a pas de zone d'inter-rangée derrière lui. C'est le cas sur la figure 4.6. Le robot, à la fin de sa mission, se

positionne en dehors du champ, une rangée de légumes sur sa gauche. Si le champ ne présente qu'une rangée de légumes, le robot longera cette rangée deux fois, une fois au-dessus d'elle, une fois en dessous ou vice-versa en fonction de la valeur du paramètre `is_track_side_at_right`. Si `final_track_outer` vaut `faux`, le robot se retrouve en fin de mission au bout de la dernière inter-rangée rencontrée, où s'il n'y a qu'une seule rangée de légumes à désherber, au bout de celle-ci ;

— `two_pass` est une variable booléenne qui vaut `vrai` si l'on veut que le robot procède, quand cela est possible, à deux passages par inter-rangée. Un double passage est effectué quand le robot est confronté à une inter-rangée et que la distance entre les deux rangées de légumes est plus grande que le seuil `two_pass_threshold`. Dans ce cas, le robot longe la rangée sur sa droite si le paramètre `is_first_uturn_right_side` vaut `faux`, sinon il longe la rangée sur sa gauche ;

— `two_pass_threshold` est un seuil au-delà duquel le robot effectue, quand les conditions sont réunies, un double passage ;

— `crop_gap` contient l'espacement (en mètres) voulu par l'utilisateur entre le robot et les rangées longées.

A.2 Description complète des paramètres de génération utilisés chapitre 4

Les 15 paramètres de génération sont distribués dans 5 classes :

— `Field` contient :
  — `nb_crop_rows` qui stocke le nombre de rangée de légumes que présente le champ ;
  — `tab_gaps` qui est un tableau stockant les `nb_crop_rows` − 1 écarts entre les diverses rangées de légumes ;

— `Mission` contient les paramètres `final_track_outer`, `first_track_outer`, `is_track_side_at_right`, `is_first_uturn_right_side` et `two_pass` qui sont détaillés section 4.4.1 ;

— `Height_map_generator_function` contient les paramètres de génération concernant la génération du terrain, soit :
  — `magnitude` détermine l'amplitude des irrégularités du terrain ;

— `Crop_row` contient :
  — `length` correspond à la taille de la rangée de légumes ;
  — `noise_amplitude_x` et `noise_amplitude_y` stockent l'amplitude du bruit (le long des axes des abscisses et ordonnées) appliqué aux positions des légumes dans une rangée ;
  — `disappearance_probability` stocke la probabilité de disparition d'un légume ;

  - `vegetable_density` correspond à la densité des légumes contenus dans une rangée. Cette densité est tirée au sort entre `min_density` et `max_density` en fonction du type de légume composant le champ (poireaux ou choux) ;
  - `veg_type` qui correspond au type de légume présent dans la rangée ;
- `Disturbing_element` contient le paramètre `grass_density` qui renseigne sur la densité de l'élément perturbateur *herbe*. La classe `Field` a un tableau stockant `nb_crop_rows` + 1 densités d'herbes.

Bibliographie

[Abal 2014] Iago Abal, Claus Brabrand et Andrzej Wasowski. *42 variability bugs in the linux kernel : a qualitative analysis.* Dans Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014. (Cité en page 60.)

[Alexander 2015] Rob Alexander, Heather Rebecca Hawkins et Andrew John Rae. *Situation coverage–a coverage criterion for testing autonomous robots.* 2015. (Cité en pages 22 et 79.)

[Ammann 2008] Paul Ammann et Jeff Offutt. Introduction to software testing. Cambridge University Press, 2008. (Cité en page 11.)

[Anand 2013] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn*et al.* *An orchestrated survey of methodologies for automated software test case generation.* Journal of Systems and Software, vol. 86, no. 8, pages 1978–2001, 2013. (Cité en page 11.)

[Andrews 2016] Anneliese Andrews, Mahmoud Abdelgawad et Ahmed Gario. *World Model for Testing Autonomous Systems Using Petri Nets.* Dans IEEE 17th International Symposium on High Assurance Systems Engineering HASE, 2016. (Cité en pages 21 et 23.)

[Araiza-Illan 2015] Dejanira Araiza-Illan, David Western, Anthony Pipe et Kerstin Eder. *Coverage-Driven Verification.* Dans Haifa Verification Conference, pages 69–84. Springer, 2015. (Cité en pages 21, 23 et 25.)

[Araiza-Illan 2016a] Dejanira Araiza-Illan, Anthony G Pipe et Kerstin Eder. *Model-based Test Generation for Robotic Software : Automata versus Belief-Desire-Intention Agents.* arXiv preprint arXiv :1609.08439, 2016. (Cité en page 23.)

[Araiza-Illan 2016b] Dejanira Araiza-Illan, David Western, Anthony G Pipe et Kerstin Eder. *Systematic and realistic testing in simulation of control code for robots in collaborative human-robot interactions.* Dans Conference Towards Autonomous Robotic Systems, pages 20–32. Springer, 2016. (Cité en pages 21, 23 et 25.)

[Arnold 2013] James Arnold et Rob Alexander. *Testing autonomous robot control software using procedural content generation.* Dans International Conference on Computer Safety, Reliability, and Security, pages 33–44. Springer, 2013. (Cité en pages 22, 24 et 26.)

[Barr 2015] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz et Shin Yoo. *The oracle problem in software testing : A survey.* IEEE transactions on software engineering, vol. 41, no. 5, pages 507–525, 2015. (Cité en page 12.)

[Biesiadecki 2006] Jeffrey J Biesiadecki et Mark W Maimone. *The mars exploration rover surface mobility flight software driving ambition.* Dans IEEE Aerospace Conference, 2006. (Cité en page 17.)

[Boeing 2007] Adrian Boeing et Thomas Bräunl. *Evaluation of real-time physics simulation systems.* Dans Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia, 2007. (Cité en page 26.)

[Bond 2007] Michael D Bond, Nicholas Nethercote, Stephen W Kent, Samuel Z Guyer et Kathryn S McKinley. *Tracking bad apples : reporting the origin of null and undefined value errors.* Dans ACM SIGPLAN Notices, 2007. (Cité en page 70.)

[Bonnafous 2001] David Bonnafous, Simon Lacroix et Thierry Siméon. *Motion generation for a rover on rough terrains.* Dans Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2001. (Cité en page 17.)

[Brugali 2010] Davide Brugali et Azamat Shakhimardanov. *Component-based robotic engineering (part 1).* Dans IEEE Robotics & Automation Magazine, vol. 17, no. 1, pages 100–112, 2010. (Cité en page 14.)

[Carlson 2003] Jennifer Carlson et Robin R Murphy. *Reliability analysis of mobile robots.* Dans IEEE International Conference on Robotics and Automation (ICRA), volume 1, pages 274–281, 2003. (Cité en pages 18 et 19.)

[Carlson 2004] Jennifer Carlson, Robin R Murphy et Andrew Nelson. *Follow-up analysis of mobile robot failures.* Dans IEEE International Conference on Robotics and Automation (ICRA), volume 5, pages 4987–4994, 2004. (Cité en page 19.)

[Cavezza 2014] Davide G Cavezza, Roberto Pietrantuono, Javier Alonso, Stefano Russo et Kishor S Trivedi. *Reproducibility of environment-dependent software failures : An experience report.* Dans IEEE 25th International Symposium on Software Reliability Engineering ISSRE, 2014. (Cité en page 60.)

[Chillarege 1992] Ram Chillarege, Inderpal S Bhandari, Jarir K Chaar, Michael J Halliday, Diane S Moebus, Bonnie K Ray et M-Y Wong. *Orthogonal defect classification-a concept for in-process measurements.* IEEE Transactions on software Engineering, 1992. (Cité en page 68.)

[Cook 2014] Daniel Cook, Andrew Vardy et Ron Lewis. *A survey of AUV and robot simulators for multi-vehicle operations.* Dans Proceedings of the IEEE/OES Conference on Autonomous Underwater Vehicles AUV, 2014. (Cité en page 31.)

[Echeverria 2011] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote et Séverin Lemaignan. *Modular open robots simulation engine : Morse.* Dans IEEE International Conference on Robotics and Automation ICRA, 2011. (Cité en pages 26 et 31.)

[Einhorn 2012] Erik Einhorn, Tim Langner, Ronny Stricker, Christian Martin et Horst-Michael Gross. *MIRA-middleware for robotic applications.* Dans Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, pages 2591–2598. IEEE, 2012. (Cité en page 15.)

[Guiochet 2017] Jérémie Guiochet, Mathilde Machin et Hélène Waeselynck. *Safety-critical advanced robots : A survey.* Robotics and Autonomous Systems, vol. 94, pages 43–52, 2017. (Cité en pages 3 et 12.)

[Huang 2007] Hui-Min Huang. *Autonomy levels for unmanned systems (ALFUS) framework : safety and application issues.* Dans Proceedings of the 2007 Workshop on Performance Metrics for Intelligent Systems, pages 48–53. ACM, 2007. (Cité en page 13.)

[Huang 2008] Hui-Min Huang. *Autonomy Levels for Unmanned Systems (ALFUS) Framework - Volume I : Terminology.* Dans Special Publication 1011-I-2.0, Nationnal Institute of Standards and Technology (NIST), 2008. (Cité en page 13.)

[Ingrand 2007] Félix Ingrand, Simon Lacroix, Solange Lemai-Chenevier et Frederic Py. *Decisional autonomy of planetary rovers.* Journal of Field Robotics, vol. 24, no. 7, pages 559–580, 2007. (Cité en pages 14 et 16.)

[Ingrand 2014] Félix Ingrand et Malik Ghallab. *Deliberation for autonomous robots : A survey.* Artificial Intelligence, 2014. (Cité en page 13.)

[Jin 2012] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz et Shan Lu. *Understanding and detecting real-world performance bugs.* ACM SIGPLAN Notices, 2012. (Cité en page 60.)

[Kanehiro 2002] Fumio Kanehiro, Kiyoshi Fujiwara, Shuuji Kajita, Kazuhito Yokoi, Kenji Kaneko, Hirohisa Hirukawa, Yoshihiko Nakamura et Katsu Yamane. *Open architecture humanoid robotics platform.* Dans Proceedings of IEEE International Conference on the Robotics and Automation ICRA, 2002. (Cité en page 27.)

[Koenig 2004] Nathan Koenig et Andrew Howard. *Design and use paradigms for gazebo, an open-source multi-robot simulator.* Dans International Conference on Intelligent Robots and Systems (IROS), 2004. (Cité en pages 21, 26 et 32.)

[Kortenkamp 2008] David Kortenkamp et Reid Simmons. *Robotic systems architectures and programming.* Dans Springer Handbook of Robotics, pages 283–306. Springer, 2008. (Cité en page 13.)

[Lacroix 2002] Simon Lacroix, Anthony Mallet, David Bonnafous, Gerard Bauzil, Sara Fleury, Matthieu Herrb et Raja Chatilla. *Autonomous rover navigation on unknown terrains : Functions and integration.* The International Journal of Robotics Research, 2002. (Cité en pages 1, 16 et 17.)

[Laprie 1996] Jean-Claude Laprie, Jean Arlat, JP Blanquart, A Costes, Y Crouzet, Y Deswarte, JC Fabre, H Guillermain, M Kaâniche, K Kanoun*et al. Guide de la sûreté de fonctionnement.* Toulouse : Cépaduès, 1996. (Cité en pages 10 et 19.)

[Louis 2015] Louis Louis, David Andreu, Karen Godary-Dejean et Lionel Lapierre. *HIL Simulator for AUV with ContrACT.* 2015. (Cité en page 27.)

[Mallet 2010] Anthony Mallet, Cédric Pasteur, Matthieu Herrb, Séverin Lemaignan et Félix Ingrand. *GenoM3 : Building middleware-independent robotic components.* Dans IEEE International Conference on Robotics and Automation (ICRA), pages 4627–4632, 2010. (Cité en page 15.)

[McMinn 2004] Phil McMinn. *Search-based software test data generation : a survey.* Software testing, Verification and reliability, vol. 14, no. 2, pages 105–156, 2004. (Cité en page 11.)

[Menzies 2005] Tim Menzies et Charles Pecheur. *Verification and validation and artificial intelligence.* Advances in computers, vol. 65, pages 153–201, 2005. (Cité en page 20.)

[Metta 2006] Giorgio Metta, Paul Fitzpatrick et Lorenzo Natale. *Yarp : Yet another robot platform.* International Journal of Advanced Robotic Systems, vol. 3, no. 1, page 8, 2006. (Cité en page 14.)

[Michel 1998] Olivier Michel. *Webots : Symbiosis between virtual and real mobile robots.* Dans Virtual Worlds, 1998. (Cité en page 27.)

[Micskei 2012] Zoltán Micskei, Zoltán Szatmári, János Oláh et István Majzik. *A concept for testing robustness and safety of the context-aware behaviour of autonomous systems.* Dans KES International Symposium on Agent and Multi-Agent Systems : Technologies and Applications, pages 504–513. Springer, 2012. (Cité en pages 21, 22 et 24.)

[Miller 1986] Gavin SP Miller. *The definition and rendering of terrain maps.* Dans ACM SIGGRAPH Computer Graphics, volume 20, pages 39–48, 1986. (Cité en page 28.)

[Miller 2004] Andrew T Miller et Peter K Allen. *Graspit ! a versatile simulator for robotic grasping.* Robotics and Automation Magazine, IEEE, 2004. (Cité en page 26.)

[Mühlbacher 2016] Clemens Mühlbacher, Stephan Gspandl, Michael Reip et Gerald Steinbauer. *Improving dependability of industrial transport robots using model-based techniques.* Dans 2016 IEEE International Conference on Robotics and Automation (ICRA), 2016. (Cité en pages 21, 23 et 25.)

[Murschitz 2016] Markus Murschitz, Oliver Zendel, Martin Humenberger, Christoph Sulzbachner et Gustavo Fernández Domınguez. *An Experience Report on Requirements-Driven Model-Based Synthetic Vision Testing.* 2016. (Cité en pages 22, 24 et 25.)

[Nakamura 2006] Taiga Nakamura, Lorin Hochstein et Victor R Basili. *Identifying domain-specific defect classes using inspections and change history.* Dans Proceedings of the ACM/IEEE international symposium on Empirical software engineering, 2006. (Cité en page 65.)

[Nguyen 2009] Cu D Nguyen, Anna Perini, Paolo Tonella, Simon Miles, Mark Harman et Michael Luck. *Evolutionary testing of autonomous software*

agents. Dans Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1, pages 521–528. International Foundation for Autonomous Agents and Multiagent Systems, 2009. (Cité en pages 22, 24 et 46.)

[Parenthoën 2004] Marc Parenthoën, Thomas Jourdan, Jacques Tisseau et others. *IPAS : Interactive phenomenological animation of the sea*. Dans The Fourteenth International Offshore and Polar Engineering Conference, 2004. (Cité en page 26.)

[Pecheur 2000] Charles Pecheur. *Verification and validation of autonomy software at NASA*. 2000. (Cité en page 20.)

[Pelekis 2007] N. Pelekis, I. Kopanakis, G. Marketos, I. Ntoutsi, G. Andrienko et Y. Theodoridis. *Similarity Search in Trajectory Databases*. Dans 14th International Symposium on Temporal Representation and Reasoning TIME, 2007. (Cité en page 115.)

[Perlin 1985] Ken Perlin. *An image synthesizer*. ACM Siggraph Computer Graphics, 1985. (Cité en pages 28 et 93.)

[Peynot 2006] Thierry Peynot. *Sélection et contrôle de modes de déplacement pour un robot mobile autonome en environnements naturels*. PhD thesis, Institut National Polytechnique de Toulouse-INPT, 2006. (Cité en page 15.)

[Peytavie 2010] Adrien Peytavie. *Génération procédurale de Monde*. PhD thesis, Université Claude Bernard-Lyon I, 2010. (Cité en page 27.)

[Quigley 2009] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler et Andrew Y Ng. *ROS : an open-source Robot Operating System*. Dans ICRA workshop on open source software, volume 3, page 5. Kobe, 2009. (Cité en page 14.)

[Shaker 2015] Noor Shaker, Julian Togelius et Mark Nelson. *Fractals, noise and agents with applications to landscapes (DRAFT)*. Dans Procedural Content Generation in Games : A Textbook and an Overview of Current Research. 2015. (Cité en page 28.)

[Smith 2005] Russell Smith et others. *Open dynamics engine*. Rapport technique, 2005. (Cité en page 26.)

[SPARC 2016] SPARC. *Robotics 2020 multi-annual roadmap for robotics in europe, Horizon 2020 Call ICT-2017 (ICT-25, ICT-27 and ICT-28), Release B*. 02/12/2016. (Cité en page 13.)

[Steinbauer 2013] Gerald Steinbauer. *A Survey about Faults of Robots used in RoboCup*. Dans RoboCup Robot Soccer World Cup XVI. 2013. (Cité en page 20.)

[Tikhanoff 2008] Vadim Tikhanoff, Angelo Cangelosi, Paul Fitzpatrick, Giorgio Metta, Lorenzo Natale et Francesco Nori. *An open-source simulator for cognitive robotics research : the prototype of the iCub humanoid robot simulator*. Dans Proceedings of the 8th workshop on performance metrics for intelligent systems, 2008. (Cité en page 26.)

[Tiwari 2003] Ashish Tiwari et Purnendu Sinha. *Issues in v&v of autonomous and adaptive systems.* Dans IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), volume 2, pages 1339–1342, 2003. (Cité en page 20.)

[Togelius 2010] Julian Togelius, Mike Preuss et Georgios N Yannakakis. *Towards multiobjective procedural map generation.* Dans Proceedings of the Workshop on Procedural Content Generation in Games, 2010. (Cité en page 28.)

[Togelius 2011] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley et Cameron Browne. *Search-based procedural content generation : A taxonomy and survey.* IEEE Transactions on Computational Intelligence and AI in Games, 2011. (Cité en pages 22 et 28.)

[Tomatis 2003] Nicola Tomatis, Gregoire Terrien, Ralph Piguet, Daniel Burnier, Samir Bouabdallah, Kai Oliver Arras et Roland Siegwart. *Designing a secure and robust mobile interacting robot for the long term.* Dans Proceedings of IEEE International Conference on Robotics and Automation ICRA, 2003. (Cité en page 19.)

[Utting 2010] Mark Utting et Bruno Legeard. Practical model-based testing : a tools approach. Morgan Kaufmann, 2010. (Cité en page 23.)

[Wegener 2004] Joachim Wegener et Oliver Bühler. *Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system.* Dans Genetic and Evolutionary Computation Conference, pages 1400–1412. Springer, 2004. (Cité en page 24.)

[Weyuker 1982] Elaine J Weyuker. *On testing non-testable programs.* The Computer Journal, vol. 25, no. 4, pages 465–470, 1982. (Cité en page 12.)

[Yannakakis 2011] Georgios N Yannakakis et Julian Togelius. *Experience-driven procedural content generation.* IEEE Transactions on Affective Computing, 2011. (Cité en page 28.)

[Zendel 2013] Oliver Zendel, Wolfgang Herzner et Markus Murschitz. *VITRO-Model based vision testing for robustness.* Dans 44th International Symposium on Robotics ISR, 2013. (Cité en pages 22 et 24.)

[Zendel 2015] Oliver Zendel, Markus Murschitz, Martin Humenberger et Wolfgang Herzner. *CV-HAZOP : Introducing Test Data Validation for Computer Vision.* Dans Proceedings of the IEEE International Conference on Computer Vision, 2015. (Cité en page 22.)

[Zou 2014] Xueyi Zou, Rob Alexander et John McDermid. *Safety Validation of Sense and Avoid Algorithms Using Simulation and Evolutionary Search.* Dans Computer Safety, Reliability, and Security. 2014. (Cité en pages 22 et 24.)

[Zou 2016] Xueyi Zou, Rob Alexander et John McDermid. *On the Validation of a UAV Collision Avoidance System Developed by Model-Based Optimization : Challenges and a Tentative Partial Solution.* Dans IEEE/IFIP International

Conference on Dependable Systems and Networks Workshop, 2016. (Cité en page 24.)

[Zykov 2009] Victor Zykov, Phelps Williams, Nicolas Lassabe et Hod Lipson. *Molecubes Extended : Diversifying Capabilities of Open-Source Modular Robotics*. Rapport technique, 2009. (Cité en page 27.)

Résumé

Un des défis majeurs pour le déploiement de systèmes autonomes dans des environnements variés, non structurés et à proximité de l'homme, est d'établir la confiance entre ces systèmes et leurs utilisateurs. En effet, les fautes internes du système, les incertitudes sur la perception, ou encore les situations non prévues, sont des menaces importantes qui pèsent sur cette confiance. Nos travaux se concentrent sur les robots autonomes qui font partis des systèmes autonomes. La validation du logiciel de navigation embarqué dans les robots est généralement centrée sur des tests sur le terrain, qui sont coûteux et potentiellement risqués pour le robot lui-même ou son environnement. De plus, il n'est possible de tester le système que dans un sous-ensemble restreint de situations. Une approche alternative consiste à effectuer des tests basés sur la simulation en immergeant le logiciel dans des mondes virtuels.

L'objectif de cette thèse est d'étudier les possibilités et les limites qu'offre le test en simulation du logiciel embarqué dans les systèmes autonomes. Nos travaux traitent particulièrement du test en simulation de la couche de navigation de systèmes autonomes mobiles.

Le premier chapitre présente les contextes de la sûreté de fonctionnement, des systèmes autonomes et de leur test, de la simulation et de la génération procédurale de mondes. Les problématiques liées au test des systèmes autonomes en simulation, telles que la définition et la génération des entrées ainsi que l'expression de l'oracle, sont identifiées et discutées. La génération procédurale de mondes utilisée dans les jeux vidéos est retenue comme piste pour répondre au problème de la générations des entrées de test (mondes et missions).

Une première contribution est proposée dans le 2^e chapitre qui s'appuie sur la définition et l'implémentation d'une première plateforme expérimentale de test en simulation avec un robot mobile. Le logiciel de navigation utilisé est intégré dans le framework Genom et testé avec le simulateur MORSE. À travers cette expérimentation, des premières conclusions sont établies sur la pertinence de la génération procédurale de mondes, et sur l'oracle à considérer. Des mesures comme la tortuosité ou l'indéterminisme de la navigation sont définies. Ce premier travail mène également à proposer une approche permettant de définir des niveaux de difficulté de mondes.

L'objectif du 3^e chapitre est d'identifier si les fautes connues et corrigées dans un logiciel de navigation auraient pu être détectées via la simulation. Près de 10 ans de *commits* du logiciel de navigation (dont le module P3D qui est une version académique d'un planificateur de trajectoire utilisé par la NASA) ont été ainsi analysés. Chaque faute relevée est étudiée pour déterminer si elle serait activable en simulation, et l'oracle nécessaire pour la détecter. De nombreuses recommandations sont extraites de cette étude, notamment sur les propriétés de l'oracle à mettre en place pour ce genre de système.

Dans le quatrième chapitre, les enseignements tirés des deux chapitres précédents sont mis en œuvre dans une étude de cas d'un robot industriel. Le système considéré, fourni par notre partenaire industriel Naïo est celui du robot agricole bineur Oz. Les conclusions des chapitres précédents concernant la génération de monde et les oracles nécessaires sont validées par une campagne de test intensifs en simulation.

Mots clefs : *systèmes autonomes, simulation, test, sûreté de fonctionnement.*

Abstract

One of the major challenges for the deployment of autonomous systems in diverse, unstructured and human shared environments, is the trust that can be placed in those systems. Indeed, internal faults in those systems, uncertainties on the perception, or even unforeseen situations, threat this confidence. Our work focus in autonomous robots, which are part of autonomous systems. The validation of the navigation software embedded in robots typically involves test campaigns in the field, which are expensive and potentially risky for the robot itself or its environment. These tests are able to test the system only in a small subset of situations. An alternative is to perform simulation-based testing, by immersing the software in virtual worlds. The aim of this thesis is to study the possibilities and limits offered by simulation-based testing of embedded software in autonomous systems. Our work deals particularly with simulation-based testing of the navigation layer of autonomous mobile robots.

The first chapter introduce the contexts of dependability, autonomous systems and their testing, simulation and procedural generation of worlds. We identify and discuss the issues related to autonomous systems simulation-based testing, such as the definition and generation of inputs as well as the oracle. The procedural generation of worlds used in video games is retained as a way to answer the problem of the generation of test inputs (worlds and missions).

A first contribution is presented in the second chapter, which is based on the definition and implementation of a first experimental simulation-based testing framework with a mobile robot. The navigation software used is integrated into the Genom framework and tested with the MORSE simulator. Through this experiment, first conclusions are drawn on the relevance of the procedural generation of worlds, and on the oracle to be considered. Measures such as tortuousness or indeterminism of navigation are defined. This first work also leads to propose an approach to define levels of difficulty of worlds.

The purpose of the third chapter is to identify whether faults known and corrected in a academic navigation software could have been detected through simulation-based testing. Nearly 10 years of commits of the navigation software (including the P3D module which is an academic version of a trajectory planner used by NASA) were thus analyzed. Each fault detected is studied to determine the oracle necessary to detect it whether it could be activated in simulation. Many recommendations are extracted from this study, especially on the properties of the oracle to set up for this type of system.

In the fourth chapter, lessons learned from the previous two chapters are implemented for the case of an industrial robot. The considered system, provided by our industrial partner Naïo is the agricultural robot Oz. The conclusions of the preceding chapters regarding the world generation and the oracles are validated by an intensive test campaign in simulation.

Key words : *autonomous systems, simulation, test, safety.*