Génération et analyse de tests pour les systèmes autonomes

Clément Robert

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 *23/07/2021* **par :**

Clément ROBERT

Génération et Analyse de tests pour les systèmes autonomes

JURY

Pascale Legall	Professeur des Universités	Rapporteur
Arnaud Gotlieb	Professeur des Universités	Rapporteur
Karen Godary Dejean	Maître de conférences	Examinateur
Yves Le Traon	Professeur des Universités	Examinateur
Jérémie Guiochet	Maître de Conférences	Directeur de thèse
Hélène Waeselynck	Directrice de recherche CNRS	Directrice de thèse

---

École doctorale et spécialité :
EDSYS : Informatique 4200018 & Robotique 4200046
Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes
Directrice et directeur de Thèse :
Hélène Waeselynck et *Jérémie Guiochet*
Rapporteurs :
Pascale Legall et *Arnaud Gotlieb*

"We can only see a short distance ahead, but we can see plenty there that needs to be done."(Nous ne pouvons voir qu'à une courte distance, mais nous pouvons y voir beaucoup de choses à faire.)

Alan Turing.

Résumé

Les robots autonomes sont principalement testés par des expérimentations sur le terrain. Cette approche est coûteuse et peut présenter des risques. Une méthode alternative consiste à réaliser une partie des tests sur une plate-forme de simulation. Ceci soulève plusieurs défis : la génération d'environnements virtuels complexes, la vérification automatisée du comportement observé (oracle de test), et la fidélité imparfaite de la simulation par rapport à des tests sur le terrain. Cette thèse est divisée en deux parties axées autour de ces problématiques.

Dans une première partie, nos travaux abordent ces différents points au travers du cas d'étude d'un robot agricole, le robot *Oz* de la société Naïo Technologies. Nous faisons en particulier une analyse comparative des fautes trouvées en simulation et sur le terrain. La contribution principale de cette partie est de démontrer la faisabilité et l'efficacité du test en simulation sur un cas industriel. Malgré leur fidélité imparfaite, les tests en simulation révèlent la plupart des fautes trouvées par les expérimentations avec le robot réel, y compris celles ayant causé la majorité des défaillances sur le terrain. Ils révèlent également une nouvelle faute passée inaperçue sur le terrain mais confirmée par l'industriel. Cependant, l'effort d'implémentation des tests virtuels se révèle important, avec notamment le développement d'un générateur de données spécifiques aux environnements 3D du robot. Ce constat a mené la suite de nos travaux vers l'étude de la génération de tests.

La deuxième partie de nos travaux introduit une nouvelle méthode de génération de test et un outil qui l'implémente. Elle se base sur un modèle de données purement déclaratif, en combinant la résolution de contraintes SMT et l'échantillonnage aléatoire. Bien que le point de départ de ces travaux soit la génération d'environnements virtuels pour les systèmes autonomes, notre méthode de génération s'applique à un domaine bien plus large. Elle permet de produire des données structurées de tailles variables, tout en respectant des propriétés sémantiques et en offrant de la diversité. Nous avons évalué notre méthode sur quatre cas d'étude venant de domaines d'application variés (le robot Oz, un système de gestion de taxe, des images de dégradé de gris et des structures arborescentes). Nos résultats montrent des données générées efficacement et couvrant bien l'espace des solutions. Notre méthode est compétitive au regard des approches existantes auxquelles nous l'avons comparée.

Mots clefs : *test, génération de tests, satisfaction de contraintes, oracle, simulation, systèmes autonomes, indéterminisme, sûreté de fonctionnement.*

Abstract

Autonomous robots are mainly tested through field experiments. This approach is costly and may present risks. An alternative method is to perform part of the tests on a simulation platform. This raises several challenges : the generation of complex virtual environments, the automated verification of the observed behavior (test oracle), and the imperfect fidelity of the simulation compared to real field tests. This thesis is divided into two parts centered around these challenges.

In a first part, our work addresses these different points through the case study of an agricultural robot, the Oz robot from Naïo Technologies. In particular, we make a comparative analysis of the faults found in simulation and in the field. The main contribution of this part is to demonstrate the feasibility and efficiency of simulation testing of an industrial case. Despite their imperfect fidelity, the simulation tests reveal most of the faults found by the experiments with the real robot, including the one that caused the majority of the failures in the field. They also reveal a new fault that was unnoticed in the field but later confirmed by the industry. However, the implementation effort of the virtual tests is important, with in particular the development of a data generator specific to the 3D environments of the robot. This observation led the continuation of our work towards the study of test generation.

The second part of our work introduces a new test generation method and a prototype that implements it. It is based on a purely declarative data model, combining SMT constraint resolution and random sampling. Although the starting point of this work is the generation of virtual environments for autonomous systems, our generation method applies to a much broader domain. It allows the generation of structured data of varying sizes, while respecting semantic properties and providing diversity. We have evaluated our method on four case studies coming from various application domains (the Oz robot, a management system for taxation, color gradient images and tree structures). Our results show that data are generated efficiently and cover the solution space. Our method is competitive with the existing approaches to which we compared it.

Keywords : *test, test generation, constraint satisfaction, oracle, simulation, autonomous systems, indeterminism, dependability.*

Table des matières

Table des figures

Liste des tableaux

Introduction générale

L'automatisation permet aujourd'hui de remplacer les interventions humaines dans beaucoup de tâches pénibles, répétitives, voire impossibles ou trop dangereuses pour l'humain. L'avènement des systèmes autonomes exacerbe ce transfert de responsabilité pour des tâches de plus en plus critiques. Du rover destiné à l'exploration spatiale au drone militaire d'observation, en passant par les automobiles, tous présentent un ensemble de caractéristiques communes aux systèmes autonomes, et posent la question de la confiance que l'on peut leur accorder.

En effet, dans les pires scénarios, une défaillance peut avoir des conséquences catastrophiques (mise en danger de l'utilisateur, destruction de son environnement ou même autodestruction). Cela pose alors la question de la validation de ces systèmes avant leur déploiement. Le domaine de la sûreté de fonctionnement propose un ensemble de techniques pour répondre à ce besoin, dont le test, généralement réalisé en exécutant le système sous test.

Le test d'un système peut être décomposé en trois phases. La sélection / production de cas de test, l'exécution du test et pour finir l'analyse des résultats. Cependant, la mise en place des tests dans le monde réel est laborieuse, car en plus des risques que les tests occasionnent, ils requièrent parfois des infrastructures spécifiques complexes et coûteuses, réduisant d'autant la diversité des cas de test. Ces limitations amènent à considérer le test en simulation comme une activité de plus en plus importante pour la validation et complémentaire aux test sur le terrain. La maturité des simulateurs et la puissance de calcul des ordinateurs actuels permettent des exécutions intensives de tests dans des environnements virtuels, à moindre coût et à risque nul. En revanche, cette approche introduit de nouvelles problématiques pour chaque phase du test. Se posent alors les questions suivantes :

— Comment s'assurer de la pertinence des résultats du test en simulation ? Pendant la phase d'exécution du test en simulation, le comportement observé peut être différent de celui observé dans un environnement réel. La fidélité des simulateurs ne sera jamais équivalente à la réalité, mais il est important d'identifier les apports de la simulation.

— Comment spécifier et générer les cas de test ? Pour la phase de sélection / production de cas de test, il n'est pas envisageable de les générer manuellement. Les données sont souvent trop complexes (proportionnellement à la complexité du système sous test) et il peut exister des interdépendances entre les données d'un cas de test.

— Comment spécifier et développer un oracle pour les systèmes autonomes ? Le test intensif va produire une grande quantité de données à analyser. Il faut donc un oracle automatique capable de déterminer si le comportement du système correspond à celui attendu. La nature non déterministe de ces systèmes rend le problème non trivial.

Le travail présenté dans cette thèse traite des ces problématiques en deux temps. La première partie aborde ces questions de pertinence de la simulation, de la modélisation et de la génération des cas de test, et du type d'oracle, dans le cadre des systèmes autonomes de façon empirique, grâce à une expérimentation sur un robot agricole. Ce travail sert de base pour la mise en place dans la deuxième partie, d'une approche originale pour la génération de cas de test, permettant de garantir à la fois la diversité des cas de test, mais également de satisfaire les contraintes pouvant exister entre les données des ces cas. En plus des contributions théoriques, une plate-forme de test pour un robot agricole et un outil de génération de cas de test contraints et variés ont été développés et seront présentés dans ce manuscrit.

Ce manuscrit de thèse est donc structuré en deux parties, précédées d'un premier chapitre précisant le contexte de nos travaux et l'état de l'art associé. Ce premier chapitre présente notamment les concepts et les travaux autour du test en simulation des systèmes autonomes, et également les approches existantes pour la génération de tests de logiciels.

La première partie détaille nos travaux sur le test en simulation d'un robot agricole industriel développé par Naïo Technologies. Dans cette première partie, le chapitre 2 s'appuie sur les les travaux préliminaires réalisés par [Sotiropoulos 2018] lors de travaux conjoints. Nous y détaillons la modélisation de l'environnement pour la génération de cas de test, la spécification de l'oracle et une première analyse des résultats. L'objectif de ce chapitre est de présenter les différents éléments indispensables à la mise en place d'une plate-forme de test en simulation et d'en détailler les étapes de développement. Puis, le chapitre 3 qui constitue la contribution de cette partie, analyse les résultats du chapitre 2 pour les comparer aux résultats des tests terrain effectués indépendamment par les ingénieurs de chez Naïo. Cette analyse comparative permet d'évaluer la pertinence de notre campagne de test en simulation et de dresser un premier bilan sur le défi majeur qu'est la génération des cas de test abordé dans la seconde partie.

La deuxième partie décrit nos travaux sur le développement d'un générateur générique de données structurées diverses et avec contraintes : *TAF (Testing Automation Framework)* et est composé de trois chapitres. Le Chapitre 4 décrit de manière détaillée le langage de description de modèle pour *TAF*. En plus, d'en montrer les concepts, il sert également de guide à quiconque souhaite utiliser notre approche. Le chapitre suivant (chapitre 5) présente les différents algorithmes au coeur de *TAF*, mettant en avant leurs avantages et leurs inconvénients. Le chapitre 6 évalue les performances et la couverture de *TAF* sur quatre cas d'étude dont la génération de champs de légumes pour le robot agricole vu en première partie. Parmi ces quatre cas d'étude, deux sont également modélisés avec un autre outil (le seul présentant les mêmes caractéristiques que *TAF* à notre connaissance) pour comparer les résultats.

Finalement, la conclusion générale dresse un bilan de nos travaux puis ouvre sur les améliorations et les perspectives de cette thèse.

Publications de la thèse

1. Clément Robert, Jérémie Guiochet, Hélène Waeselynck, Luca Vittorio Sartori. *TAF : A tool for diverse and constrained test case generation.* International Symposium on Software Reliability Engineering (ISSRE), 2021, Wuhan, Chine. **soumission en cours**
2. Adrien Hereau, Karen Godary-Dejean, Jérémie Guiochet, Clément Robert, Thomes Claverie, Didier Crestani. *Testing an Underwater Robot Executing Transect Missions in Mayotte.* Towards Autonomous Robotic Systems Conference (TAROS), 2020, Nottingham, Royaume-Uni. (`https://www.nottingham.ac.uk/conference/fac-eng/taros/index.aspx`)
3. Clément Robert, Jérémie Guiochet, Hélène Waeselynck. *Testing a non-deterministic robot in simulation – How many repeated runs ?* IEEE International conference in Robotic Computing (IRC), 2020, Taochung, Taïwan. (`http://2020.issre.net/`)
4. Clément Robert, Thierry Sotiropoulos, Hélène Waeselynck, Jérémie Guiochet, Simon Vernhes. *The virtual lands of Oz : testing an agribot in simulation.* International journal Empirical Software Engineering, Springer Volume 25, Issue 3, 2020, pp 2025-2054). (`https://link.springer.com/journal/10664/25/3`)
5. Clément Robert. *First insights into testing autonomous robots in virtual worlds.* Student forum at International Symposium on Software Reliability Engineering (ISSRE), 2017, Toulouse, France. (`http://2017.issre.net/`)

Chapitre 1

Contexte et concepts fondamentaux

Sommaire

1.1 Introduction

Le test de systèmes autonomes est devenu un enjeu important pour préparer leur déploiement. Face à l'infinité de situations et de conditions à tester pour ces systèmes, le test en simulation devient stratégique.

Les travaux présentés dans cette thèse s'appuient sur des résultats issus du domaine du test, mais se focalisent sur le test en simulation, et dans le contexte particulier des systèmes autonomes. Ainsi, nous présentons en premier lieu dans la section 1.2 les concepts autour du test en général et de l'autonomie, en insistant sur l'indéterminisme des systèmes autonomes qui n'est pas sans conséquence pour

l'interprétation des résultats des tests. Nous dressons ensuite un état de l'art sur les méthodes et outils pour la simulation des systèmes autonomes dans la section 1.3. Ces simulateurs sont au coeur des suites logicielles permettant de tester virtuellement les systèmes autonomes. La section 1.4 détaille le concept d'oracle de test pour le cas particulier des systèmes autonomes. Elle montre la nécessité d'un oracle automatique et les différents verrous quant à sa réalisation. La contribution de la seconde partie de cette thèse étant sur la génération des entrées de test (ou cas de test), nous présentons les principales approches existantes dans le contexte des systèmes autonomes dans la section 1.5, puis nous présentons un état de l'art sur la génération de données structurées en général (section 1.6) utilisée pour la création de cas de test. Le langage de modélisation puis les aspects liés aux contraintes y sont discutés.

1.2 Notions pour le test de systèmes autonomes

Cette section se concentre sur les notions qui sont au centre de nos travaux. D'abord les notions de test du logiciel et d'autonomie sont présentées. Le problème de l'indéterminisme dans les systèmes autonomes est ensuite discuté.

1.2.1 Test du logiciel

La sûreté de fonctionnement d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre. Le guide de la sûreté de fonctionnement [Laprie 1996] et plus récemment [Avizienis 2004] classent les différentes méthodes en quatre catégories : la prévention de fautes, la tolérance aux fautes, l'élimination de fautes et la prévision de fautes.

Dans cette terminologie, la catégorie « élimination de fautes », regroupe l'ensemble des moyens permettant de réduire la présence (nombre, sévérité) des fautes. Ces approches incluent généralement trois étapes pour leur mise en oeuvre : vérification, diagnostic, correction. L'étape de vérification, qui consiste à confronter un système à un ensemble de propriétés, peut être réalisée avec des approches telles que l'analyse statique, le *model-checking*, la preuve de théorèmes, ou le test. Ce dernier, qui fait l'objet de cette thèse, est un procédé de vérification dynamique (avec activation du système sous test) à l'aide d'entrées valuées, appelées entrées de test (ou cas de test). La sortie est comparée au comportement attendu défini généralement sous forme de propriétés à respecter. Le problème d'élaboration du verdict (acceptation ou rejet) de test est appelé problème d'oracle. Une vue schématique du test est donnée figure 1.1

Selon les objectifs du test, différents types de test existent, comme par exemple le test unitaire, le test d'intégration, le test fonctionnel, le test de performance ou encore le test de non régression [Amman 2016]. Cependant, pour tous ces types de test, le principe reste identique : un cas de test est appliqué au système sous test, et l'oracle détermine un verdict. Parmi les problématiques étudiées dans le domaine

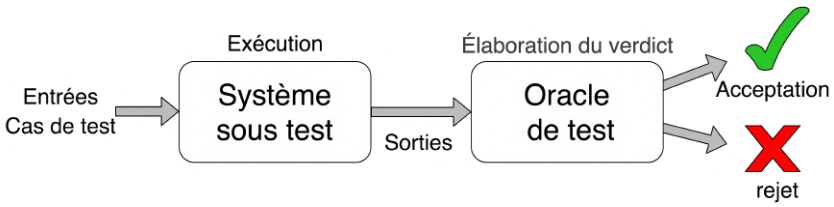


FIGURE 1.1 – Vue schématique du test

du test, nous présentons ici celles qui sont également au coeur de ce manuscrit : 1) la sélection / production d'un ensemble d'entrées représentatives dans un domaine d'entrée infini ; 2) l'élaboration d'un oracle.

L'étape qui consiste à générer des entrées cohérentes (type de fichier, plage de valeurs pour chaque variable d'entrée), que l'on peut appliquer au système sous test, a généré de nombreux travaux et publications. Le terme de génération de test, peut s'utiliser pour regrouper à la fois la création des cas de test cohérents, mais aussi la sélection parmi ces cas de test de ceux qui permettront de répondre à des objectifs de test (ou des critères). Par exemple, dans le cadre de couverture de code, la sélection des cas de test sera guidée par les parties du code que le cas de test doit activer. Cette problématique de sélection / production des entrées de test, ou plus généralement de génération, est centrale. Les sections 1.5 et 1.6 y sont entièrement consacrées.

Le deuxième point concerne l'oracle de test qui permet d'élaborer un verdict d'acceptation ou de rejet des résultats de test. Sa définition est cruciale pour pouvoir repérer tous les comportements défaillants. Une solution très utilisée est de déterminer manuellement la validité des résultats (oracle manuel). Cette solution n'est pas adaptée pour l'automatisation, il faut pouvoir traiter de gros volumes de données rapidement. L'oracle peut fonctionner parallèlement à l'exécution du test (en ligne) ou après, à partir des traces d'exécutions (hors ligne). Les oracles en ligne ont pour avantage de permettre de stopper l'exécution du test dès qu'une défaillance est détectée. Ils sont particulièrement intéressants dans les cas où les tests sont très coûteux en termes de temps d'exécution. A l'inverse, le principal atout des oracles hors ligne est de pouvoir recréer un verdict sans relancer les tests. Cela réduit considérablement le temps de test quand la spécification du système sous test ne permet pas d'établir un oracle définitif. Après chaque modification, seul l'oracle est relancé. Un tel oracle nécessite en revanche des traces d'exécutions les plus détaillées possibles.Le problème d'oracle de test est particulièrement important dans le contexte de la première partie de ce manuscrit, en conséquence nous y accordons plus de détails dans une section dédiée (section 1.4). Les différents types d'oracle et plus particulièrement ceux adaptés au test de systèmes autonomes y sont présentés.

1.2.2 Notion d'autonomie

L'autonomie (auto «soit-même», nomos «loi») est définie dans [Nguyen 2012] comme la liberté d'action sans influence d'une autorité externe. Une définition plus précise donnée dans [Rudd 2008] est «la capacité à raisonner et prendre des décisions pour atteindre un objectif en se basant sur l'état actuel de connaissance et la perception de l'environnement dans lequel le système évolue». Les systèmes autonomes n'ont pas le même degré d'autonomie dite décisionnelle. Un exemple typique est un robot mobile muni d'un système de navigation autonome, qui n'a pas le même niveau d'autonomie que le système d'atterrissage autonome des lanceurs Falcon 9 de chez SpaceX. En revanche, ils présentent des caractéristiques communes. Les auteurs de [Wirsing 2015] donnent quatre grands principes de l'autonomie d'un système :

— **Connaissance** : Le système a des connaissances sur lui-même et son environnement.

— **Adaptation** : Le système s'adapte dynamiquement aux changements d'environnement.

— **Raisonnement** : Le système évalue et raisonne sur son état.

— **Émergence** : Un système complexe émerge de sous-systèmes simples.

Un état des lieux de la littérature sur la notion d'autonomie est proposé dans [Helle 2016].

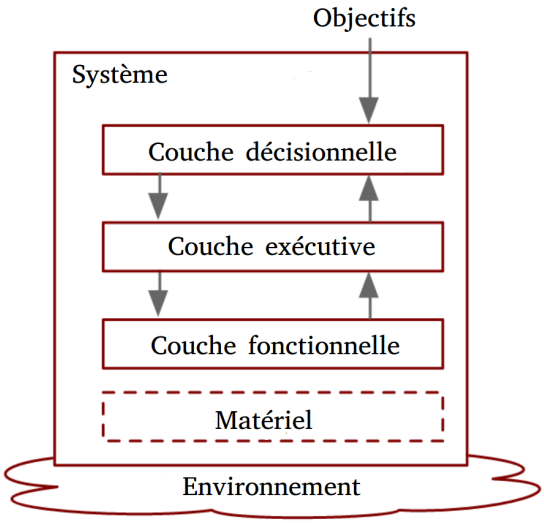


FIGURE 1.2 – Architecture en trois couches pour un système autonome (source : [Guiochet 2017]).

D'un point de vue structurel, ces systèmes sont souvent décomposés en trois couches (figure 1.2) : une couche décisionnelle, une couche exécutive et une couche

fonctionnelle. Les informations relatives à l'environnement sont propagées de la partie matérielle à la couche décisionnelle en traversant toutes les couches. A partir de ces données et des objectifs de mission, une décision est prise et propagée jusqu'à la couche la plus basse.

Le test des systèmes autonomes est complexe, aussi, la majeure partie de l'effort de test se fait sur le terrain à l'exception de quelques tests de fumée en simulation. Actuellement, il n'existe pas de méthodes reconnues pour le test intensif en simulation de ces systèmes. Les deux problématiques du test présentées précédemment sont exacerbées par l'autonomie du système ([Tiwari 2003], [Pecheur 2000] et [Menzies 2005]) :

— la sélection / production de test : le domaine d'entrée est particulièrement difficile à modéliser à cause de la complexité et de la diversité des missions et des environnements. Il en découle également des difficultés pour la modélisation.

— l'élaboration de l'oracle : il n'existe généralement pas de référence à laquelle comparer le comportement du système. Il y a donc un premier problème de spécification. De plus, ces comportements sont difficilement analysables car les systèmes autonomes sont communément non déterministes. Le résultat n'est pas le même d'une exécution à l'autre.

1.2.3 Indéterminisme

Comme nous l'avons vu ci-dessus, une des problématiques du test est la création d'un oracle (voir section 1.4), et dans le cas du test d'un système autonome cette problématique est exacerbée par le non-déterminisme du système : deux exécutions ne donnent pas le même résultat et peuvent donc aboutir à deux verdicts d'oracle différents. Le déterminisme est une notion selon laquelle la succession de chaque évènement est déterminée par le principe de causalité, le passé et les lois de la physique [Laplace 1814]. Selon ce principe, une entrée donnée à un système déterministe produira toujours la même sortie. Un système peut paraître déterministe au premier abord mais des sources de non déterminisme inconnues peuvent rendre sa sortie imprévisible.

Sources d'indéterminisme

Pour les systèmes autonomes, le logiciel embarqué est généralement composé de plusieurs *threads* (fils d'exécution) pour interagir avec l'environnement (souvent un par sous-système). La communication entre les *threads* d'un tel système repose communément sur le principe du *best-effort* (meilleurs efforts). En d'autres termes, la qualité du service de communication est la meilleure possible sans garantie sur le temps de livraison. Il en résulte des comportements non déterministes et asynchrones. Ce type de structure avec plusieurs fils d'exécution (*multithread*) est par exemple utilisé dans *ROS* (*Robot Operating System*) [Quigley 2009]. C'est un *middleware* (intergiciel) spécialement conçu pour la robotique et très utilisé dans le milieu

académique. En plus des communications, certains algorithmes introduisent également de l'indéterminisme. Les fonctions décisionnelles d'un système autonome font régulièrement intervenir ce type d'algorithme pour trouver une solution approximative à un problème lorsque la solution exacte est trop coûteuse ou impossible à obtenir (par exemple, une heuristique pour de la planification de trajectoire).

L'utilisation d'un simulateur pour le test de systèmes autonomes rend le non déterminisme encore plus problématique. En effet, un simulateur est lui-même un système complexe qui peut introduire des sources d'indéterminismes supplémentaires (concurrence des composants du simulateur, utilisation de services distants, appel à l'horloge système, fuite de mémoire, etc).

Approches de prise en compte de l'indéterminisme

Deux solutions sont possibles pour traiter les effets de l'indéterminisme. La première consiste à rendre le système déterministe en retirant les possibles sources d'indéterminisme. La seconde consiste à prendre en compte l'indéterminisme en évaluant son impact sur le système.

Il est possible d'éluder l'indéterminisme ou au moins d'en limiter les effets. Un tel procédé implique des restrictions fortes sur le système et l'environnement de test afin d'en garantir un contrôle total. L'idée est de se ramener à un programme purement séquentiel qui ne fait pas intervenir d'aléatoire. *ESTEREL* [Berry 1992] est un langage de programmation parallèle déterministe spécialement conçu pour les systèmes réactifs. Le compilateur traduit les programmes parallèles en automates séquentiels équivalents. Dans les langages synchrones, on peut également citer *LUSTRE* [Halbwachs 1991] qui est utilisé entre autres dans l'aéronautique et le nucléaire, mais aussi *SIGNAL* [Benveniste 1991]. Cependant, en plus du problème des blocages qui peuvent survenir lors de la transformation en automates séquentiels, certains systèmes ne peuvent pas être réduits en une suite d'opérations atomiques. C'est typiquement le cas des systèmes autonomes qui utilisent des communications asynchrones entre sous-systèmes et entre plusieurs processeurs.

Pour réduire l'indéterminisme introduit par la simulation (tâches concurrentes), une solution préconisée par les standard *HLA* et *DIS* ([HLA 2010] et [DIS 1998]) est d'avoir une gestion centralisée du temps simulé. Tous les messages sont alors échangés par l'intermédiaire d'un noeud central qui s'occupe de la synchronisation. Cette solution ajoute de l'utilisation de ressources de manière non négligeable et est à contre courant des *middleware* (intergiciel) utilisés en robotique comme *ROS*. Une approche décentralisée est donnée dans [Reymann 2019]. Les auteurs proposent un *framework* (infrastructure logicielle) permettant de gérer l'indéterminisme amené par le temps dans des simulations distribuées. Leur approche est basée sur l'ajout d'une couche logicielle qui synchronise le système en gérant les communications entre processus. Cette méthode est bien adaptée pour des simulateurs distribués mais ne permet pas la gestion de l'indéterminisme à l'intérieur même des systèmes et des simulateurs.

Les auteurs de [Sotiropoulos 2016] ont placé le robot *Mana* (robot académique

tout terrain développé au LAAS-CNRS) dans un environnement virtuel. La mission du robot est de se rendre dans le coin opposé d'une zone carré parsemée d'obstacles placés aléatoirement. Pour une même configuration (même point de départ, même point d'arrivée et mêmes positions des obstacles), l'itinéraire choisi par le robot peut être complètement diffèrent. L'écart entre les trajectoires observées se mesure en mètre voir en dizaine de mètres sur les pires cas. Pour prendre en compte le non déterminisme, les missions sont répétées. De manière contre intuitive, le non déterminisme parait totalement aléatoire. Son impact est très variable et ne semble pas corréler aux niveaux de difficulté. La même idée de répétition est utilisée dans [Nguyen 2012] pour le test d'un robot nettoyeur. Les exécutions d'un même cas de test sont parallélisées pour avoir un nombre de données suffisant à l'évaluation statistique d'une fonction de *fitness*. Dans les deux travaux précédents, le nombre de répétition a été choisi arbitrairement (5 répétitions). Ce choix est principalement guidé par les ressources disponibles en faisant un compromis entre le temps de test et l'erreur d'estimation.

Les exécutions multiples sont une bonne approche pour lisser les résultats et donc prendre en compte l'effet de l'indéterminisme. En particulier dans le cas d'un procédé de test itératif avec un calcul de *fitness*. Une limitation est la répétabilité des tests. Prenons l'exemple d'un robot mobile dans un environnement dynamique réel. Entre deux exécutions, le moindre décalage temporel peut alors totalement changer la configuration du robot vis-à-vis des obstacles. De manière générale, les exécutions en simulation sont plus répétables que celles dans le monde réel. Néanmoins, il y subsiste de l'indéterminisme intrinsèque au système sous test et dû à la plate-forme de simulation. Ce dernier, extérieur au système, peut être réduit ou même supprimé mais c'est un exercice difficile. L'évaluation de l'indéterminisme est un problème encore peu étudié. Comment savoir combien de répétitions sont nécessaires ? Comment limiter son impact sans appauvrir la variété des tests ? Des éléments de réponse sont introduits dans le chapitre 3.

1.3 Simulation des systèmes autonomes

Cette section débute par une courte présentation des types de test en simulation. Les notions de complexité et de fidélité des simulateurs sont ensuite abordées. Pour finir, nous proposons un bilan sur les simulateurs.

1.3.1 Les types de simulation

Le nombre de systèmes autonomes et leur complexité est en constante augmentation [Matthews 2020]. La validation de ces systèmes sur le terrain est un véritable défi impliquant des procédures de tests toujours plus longues et coûteuses. Des techniques de modélisation et de simulation sont donc mises en place pour réduire les temps de test et augmenter la couverture, en particulier pour les cas de test difficiles à reproduire de manière fiable.

Plusieurs types de test incluant de la simulation système se distinguent. Ils correspondent aux différents niveaux d'abstraction du système :

— **Model in the Loop (MiL)** : Les tests sont effectués sur le modèle (par exemple un modèle MATLAB/Simulink ou un réseau de pétri) [Plummer 2006]. Les modèles du système et de ses entrées sont simulés sans aucun matériel. Le but est de vérifier la logique algorithmique et de détecter les blocages dès le début du développement.

— **Software in the Loop (SiL)** : Le logiciel du système est intégré tel quel dans un simulateur. Les capteurs et les actionneurs sont simulés. Les travaux de la partie I ont été effectués dans cette configuration.

— **Hardware in the Loop (HiL)** : La vérification s'effectue directement sur le système sous test. Les signaux capteurs sont simulés. Les tests HiL sont notamment très utiles pour tester la connectivité et l'interdépendance des sous-systèmes [Gietelink 2006]. Dans un contexte automobile, il existe également le cas particulier **Vehicule in the Loop (ViL)** ([Bokc 2007]). Dans le cas HiL, la partie matérielle peut concerner un seul calculateur (ECU) qui fonctionne avec le reste d'un véhicule virtuel simulé. Dans le cas ViL, le matériel est le véhicule complet. Le conducteur, la route et/ou le trafic y sont simulés.

1.3.2 Complexité et fidélité des simulateurs

Le système étudié en première partie de ce manuscrit est considéré comme une boite noire. Seul le logiciel est disponible. La vérification de ce système sera donc uniquement basée sur de la simulation SiL. De manière générale, l'approche SiL est très utilisée pour la validation. Les travaux antérieurs diffèrent par la variété des systèmes testés et des simulateurs utilisés. Quelques auteurs proposent des simulateurs développés spécifiquement. Dans [Leurent 2018], une collection d'environnements routiers minimalistes permet le développement d'algorithmes de décision pour les véhicules autonomes. Elle est accompagnée d'un simulateur spécifique développé en Python. L'outil a notamment été utilisé dans [Liu 2020] pour optimiser les stratégies de décision par des méthodes d'apprentissages. Dans [Zou 2014] et [Zou 2016] un simulateur multi agents (*MASON*) est utilisé pour la validation d'un système d'évitement embarqué dans des drones. L'environnement est un rectangle en deux dimensions dans lequel les drones peuvent se déplacer et interagir. Aucune communication entre drones n'est implémentée. Les interactions sont donc uniquement basées sur les algorithmes de type *"sense and avoid"* (détecter et éviter). De façon similaire, un simulateur 2D minimaliste intégrant une carte parsemée d'obstacles est utilisé pour le test de robots autonomes dans [Arnold 2013].

Des simulations plus complexes sont souvent nécessaires (par exemple un environnement 3D avec des forces sur les objets). Pour le testeur, il devient donc difficile de concevoir un simulateur dédié à partir de rien. Heureusement, il existe des moteurs 3D et des moteurs physiques prêts à l'emploi. Ces deux types de moteurs sont

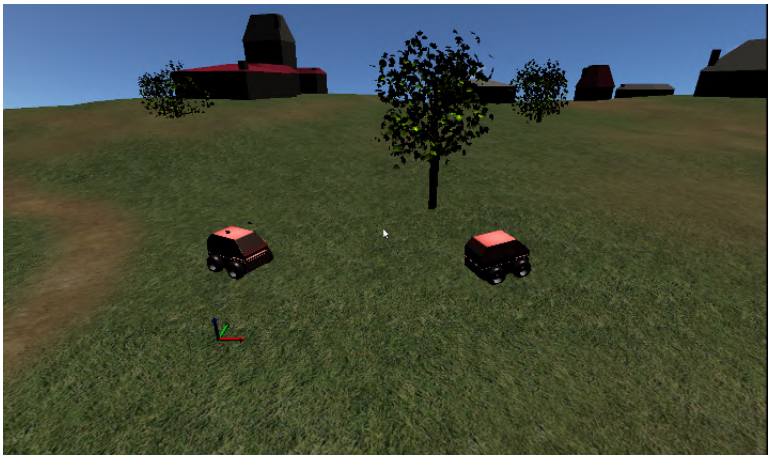


FIGURE 1.3 – Exemple simulateur MORSE (source : [Sotiropoulos 2018]).

souvent regroupés avec d'autres commodités pour former des simulateurs ou des moteurs de jeu. Une évaluation des capacités des différents moteurs physiques est donnée dans [Boeing 2007] et une comparaison des fonctionnalités des moteurs de jeu 3D est présentée dans [Rocha 2010]. Ce type de technologie évolue très rapidement, néanmoins les papiers comparatifs cités précédemment restent pertinents.

Gazebo [Koenig 2004] ou *Morse* [Echeverria 2011] sont des simulateurs conçus pour le prototypage robotique. Ils utilisent respectivement *OGRE* et *Blender* comme moteur 3D. Par exemple, dans [Yao 2015], le simulateur est basé sur *Gazebo* pour simuler une équipe de football robotique (*RoboCup*). Leur simulateur permet de facilement tester les algorithmes de collaboration entre les robots. Pour prendre un autre exemple, *Morse* est utilisé dans [Sotiropoulos 2016] (figure 1.3) pour le test du système de navigation d'un robot autonome.

Pour des raisons de performance, les moteurs de jeu sont de plus en plus utilisés pour la simulation. C'est par exemple le cas du simulateur *Morse* qui en plus d'utiliser le moteur 3D de *Blender*, utilise également son moteur de jeu. Aujourd'hui, les moteurs de jeu *Unity* [uni 2020] et *Unreal* [unr 2020] sont très performants et sont accessibles gratuitement dans un but non commercial.

Pour des systèmes très spécifiques ou nécessitant une fidélité que n'offre pas les outils disponibles, des simulateurs spécialisés sont développés. C'est par exemple le cas de l'automobile. Les récentes avancées dans le domaine des véhicules autonomes ont vu naitre un grand nombre d'outils dont des simulateurs (*Virtual Test Drive* [vir 2020], *PreScan* [pre 2020], *SCANeR* [sca 2020] ou encore *OktalSydac* [okt 2020]). Ils mettent à disposition des modèles de véhicules, des modèles de capteurs réalistes et des fonctionnalités permettant de créer des environnements routiers complexes. Ces environnements comprennent des éléments statiques (infrastructure) et des éléments dynamiques (trafic et piétons). Les rendus graphiques sont suffisamment détaillés pour permettre l'utilisation de systèmes avec caméras (détection de lignes, de panneaux, d'autres véhicules et d'obstacles). Les outils

mentionnés ci-dessus sont tous commerciaux mais il existe également des outils équivalents libres comme par exemple *CARLA* [car 2020].

D'autres domaines ont également leurs simulateurs dédiés. Le simulateur *GraspIt* [Miller 2004] est par exemple spécialement conçu pour l'étude de la préhension de mains robotiques. Il permet entre autre de détecter rapidement les collisions et contacts puis d'évaluer la qualité de la saisie d'un objet de manière réaliste. Un autre exemple est *IPAS* [Parenthoën 2004] qui permet de modéliser un environnement marin réaliste.

1.3.3 Bilan sur les simulateurs

Globalement, le spectre des simulateurs présents dans la littérature est très étendu, allant d'une fidélité minimaliste à un réalisme avancé. Les simulateurs basses fidélités permettent de facilement supporter une flotte de robots avec de l'instrumentation pour le test. Un minimum de ressource est nécessaire à leurs exécutions, le temps de simulation peut donc être accéléré (si le système sous test le permet). Un exemple de simulateur basse fidélité est présenté dans [Arnold 2013] où le terrain est représenté par les cases d'une grille. Les simulateurs proposant un degré de fidélité élevé sont moins flexibles, demandent beaucoup plus de ressources et ne sont pas génériques. Ils sont surtout dédiés aux tests d'éléments spécifiques (par exemple les environnements de mains ou maritimes mentionnés précédemment). La fidélité peut se décomposer en deux aspects : le rendu graphique et le réalisme physique. Par exemple, le simulateur *SIM4CV* pour les applications de *computer vision* (vision par ordinateur) produit des visuels d'environnements photo-réalistes [Müller 2018]. Il est basé sur le moteur de jeu *UNREAL*. Les principales limites à la fidélité d'un simulateur sont les ressources disponibles et le coût de développement. Les simulateurs tel que *Gazebo* ou *Morse* se situent au milieu de ce spectre. Ils sont un compromis entre fidélité et généricité avec une performance passable. Ils sont très utiles pour le prototypage, car relativement rapides à mettre en place grâce à leurs bibliothèques communautaires regroupant de nombreux modèles 3D de robots et leurs capteurs. Le choix d'un simulateur pour le test va donc dépendre du système à tester et du degré de fidélité nécessaire. Les auteurs de [Sotiropoulos 2016] montrent sur un cas d'étude que la plupart des bugs du système de navigation de leur robot autonome peuvent être reproduits avec une simulation de relativement basse fidélité. Les auteurs de [Timperley 2018] arrivent à la même conclusion en étudiant l'historique des bugs du projet *ARDUPILOT* (une collection de librairie open source pour le développement d'autopilote). Les différentes études sur le test en simulation sont aujourd'hui basées sur des projets libres. Dans la première partie de ce manuscrit, *Gazebo* est utilisé pour la validation d'un robot agricole autonome industriel.

1.4 Oracle de test pour les systèmes autonomes

Déterminer si la sortie d'un système est correcte pour une entrée donnée est connue sous le nom de problème d'oracle [Barr 2015]. C'est un problème beaucoup plus difficile qu'il n'y paraît. Pour les systèmes autonomes, l'aspect décisionnel des algorithmes est encore une difficulté supplémentaire au développement d'oracle. "Créer une spécification complète pour un système autonome revient à recréer la logique humaine du contrôle de celui-ci" [Tian 2018].

1.4.1 Les types d'oracle

La majorité de l'effort de recherche sur le test se concentre sur l'automatisation dans le but de rendre les tests moins coûteux, plus rapides et surtout plus fiables. Dans cet objectif d'automatisation, le problème de vérification est aujourd'hui un goulot d'étranglement. A bas niveau, la méthode classique est le test unitaire mais il ne permet pas de tester un système dans son ensemble. A l'échelle d'un système complet le test est beaucoup plus complexe est peut être divisé en deux approches : le test d'un modèle plus simple capturant les caractéristiques du système ou le test direct sur le système. La méthode de vérification de modèle ne permet qu'un haut niveau d'abstraction et devient rapidement limitée par le nombre d'états. De plus, dans le cas général des systèmes cyber-physiques elle est indécidable [Alur 2011]. Dans ce manuscrit, nous nous intéressons uniquement aux approches directes. L'oracle est alors l'un des organes essentiels, il a pour rôle d'établir la validité des résultats de test. La principale difficulté réside dans sa définition. Une mauvaise définition ne permettra pas de détecter les défaillances.

Un oracle automatique basé sur une spécification formelle et/ou sur les traces d'exécutions est réalisable (oracle spécifié / déduit). [Weyuker 1982] classe les difficultés associées en trois grandes catégories :

— Les programmes dont on ne connaît pas le résultat à priori.

— Les programmes qui produisent un volume de sortie trop important.

— Les programmes dont la spécification diffère du problème original.

Les solutions proposées se basent principalement sur de la réduction vers un sous ensemble plus simple à appréhender (seulement les entrées de test dont on connaît le résultat, seulement une sous partie des données produites).

Dans le cas des systèmes autonomes, on se retrouve très souvent dans les 3 catégories définies précédemment. 1) On ne connaît pas le résultat avant l'exécution du test à cause de l'effet boule de neige de chaque décision et du non déterminisme. Plusieurs exécutions d'un même test peuvent aboutir à des résultats complètement différents. 2) La quantité de données nécessaires à l'analyse du comportement du système est souvent énorme. On peut conjecturer qu'elle est liée à la complexité du système sous test. 3) Une spécification précise du comportement attendu par le système est très souvent incomplète ou sujette à interprétation.

Dans [Barr 2015] les oracles sont formalisés en quatre types.

— Les *oracles spécifiés* reposent sur un langage de spécification (par exemple UML / OCL) permettant de définir des expressions logiques qui sont ensuite vérifiées lors des tests. L'oracle présenté dans cette thèse (chapitre 2) est de ce type.

— Les *oracles déduits* basent leurs verdicts sur des informations construites à partir d'éléments divers (spécification, traces d'exécutions, etc.). Ils regroupent les oracles basés sur le test différentiel de deux versions, la comparaisons de différentes exécutions (par exemple le test métamorphique), ou encore les inférences d'invariants et de modèles à partir des traces d'exécution.

— Les *oracles implicites* sont construits à partir de connaissances générales non spécifiques au système sous test (détection de débordement de tampon ou de fuite mémoire). Ils ne nécessitent ni spécification formelle ni expertise concernant le système.

— Les *oracles manuels* consistent en une validation humaine du système sous test. Ce type d'oracle est parfois associé à des outils permettant de mieux cibler les tests et ainsi réduire l'effort de test. C'est la méthode la plus courante mais elle est fastidieuse et source d'erreur.

Les oracles manuels sont encore très utilisés de nos jours. Le plus gros problème est qu'ils nécessitent une expertise humaine et sont donc faillibles par définition. Prenons l'exemple, d'un robot mobile qui navigue dans un environnement donné. Un expert du système peut facilement déceler les écarts de trajectoire mais seulement jusqu'à un certain seuil. Une erreur se manifestant par une petite altération ne sera pas détectée. De plus, beaucoup de programmes ont un domaine d'entrée infini (ou beaucoup trop grand pour un test exhaustif). Pour accroître le niveau de confiance en la validité d'un tel programme, un grand nombre de test est requis. Bien que théoriquement faisable, l'évaluation manuelle des résultats devient alors très fastidieuse, voire même infaisable dans la pratique. Encore une fois le facteur humain est limitant.

Les tests réalisés à l'aide d'oracles spécifiés sont généralement basés sur des modèles ou un ensemble de propriétés. Un modèle n'offre bien souvent qu'une vue partielle et discrète des comportements étudiés. Dans [Mühlbacher 2016], une flotte de robots d'un entrepôt automatisé a pour mission de déplacer des objets. Bien que tous reliés à un serveur central, les robots sont capables de naviguer de manière autonome pour exécuter leurs missions. Un modèle est construit sous forme de système de transition d'états étiquetés. A partir d'un jeu d'étiquettes données en entrée le testeur peut vérifier la cohérence de la sortie avec la spécification. De même, dans [Araiza-Illan 2016] les propriétés d'oracles sont issues de deux standards ISO avant d'être exprimées sous forme de logique temporelle pour la vérification. De manière générale, il est difficile de spécifier un système autonome, il existe alors des approches alternatives reposant sur de l'apprentissage [Tian 2018].

Les sections qui suivent donnent plus de détails sur deux approches candidates intéressantes pour le test de systèmes autonomes : les oracles *spécifiés* basés sur des

invariants (approche retenue pour nos travaux présentés en première partie de ce manuscrit), et les oracles *déduits* basés sur des propriétés métamorphiques.

1.4.2 Oracle basé sur des invariants

Il est toujours possible de construire un oracle basé sur des propriétés invariantes. La difficulté est alors de les identifier, ils sont généralement issus des défaillances critiques propres au système sous test. Dans [Zou 2014], des drones sont simulés afin de tester différents algorithmes d'évitement de type « détecter et éviter ». Le but est de détecter les possibles conflits entre les différents algorithmes. Dans cet exemple, l'absence de collisions est l'unique propriété vérifiée par l'oracle. Dans [Ben Abdessalem 2018] les mêmes types de propriétés (non-collision) sont utilisés pour le test du freinage d'urgence automatique d'un système d'aide à la conduite d'une voiture.

Lorsque cela est possible, un oracle ne doit pas être limité à l'étude des collisions. Il peut exister d'autres propriétés intéressantes spécifiques à chaque système sous test. [Nguyen 2012] présente l'étude d'un robot nettoyeur autonome conçu pour nettoyer le hall d'un aéroport. La mission du robot consiste en l'exploration et le nettoyage d'une surface rectangulaire pouvant comporter des obstacles. Le robot doit être capable de ramasser les déchets et de les amener à la poubelle la plus proche tout en maintenant un niveau de batterie suffisant. Des stations de recharge sont disponibles pour la gestion du niveau de batterie. L'oracle correspondant intègre des propriétés plus variées que dans les travaux précédents. En plus de prendre en compte l'aspect sûreté, des propriétés de performances sont ajoutées pour évaluer l'efficacité et la robustesse. Par exemple, le temps de mission total pour nettoyer toute la zone ou encore la quantité de déchets ramassés par heure. Ces propriétés retournent des valeurs quantifiées, un seuil doit être ajouté pour rendre un verdict de réussite ou d'échec. Elles sont considérées « soft » car il n'y a pas de conséquences catastrophiques liées à leurs échecs.

Toujours dans le but de construire un oracle à partir de propriétés variées, les auteurs de [Sotiropoulos 2017] s'appuient sur leurs résultats pour proposer quelques lignes directrices à la construction d'un oracle. Leurs travaux se concentrent sur le test du système de navigation du robot *Mana*. En plus de démontrer l'efficacité potentielle du test en simulation, les auteurs proposent une classification des défaillances en cinq catégories génériques pour servir de guide à la construction d'un oracle : (i) les propriétés liées aux phases de mission, (ii) les seuils liés aux mouvements du robot, (iii) les évènements catastrophiques, (iv) les propriétés liées aux messages d'erreurs et pour finir (v) les propriétés liées à la perception.

D'autres travaux ne considèrent pas de verdicts binaires mais plutôt une approche par score où la gravité des défaillances observées pondère chaque exécution. Dans [Arnold 2013] des agents mobiles sont simulés sur une grille binaire (0 pour une case accessible et 1 pour un obstacle). Leur mission consiste à suivre un itinéraire défini par une liste ordonnée de points de passage. À partir des traces d'exécutions, l'oracle évalue des propriétés de sûreté et de performance, comme par exemple les

collisions avec un obstacle ou un autre robot, les cas où le robot s'éloigne trop de la trajectoire optimale connue ou encore une proximité dangereuse avec un autre robot. À partir de ces propriétés, un score est attribué à chaque exécution. Par exemple, 10 points sont ajoutés en cas de proximité dangereuse puis un point de pénalité additionnel par seconde dans la zone de danger. Ce système de score incrémental permet d'identifier les résultats de test les plus intéressants pour le testeur. Ces résultats sont ensuite manuellement dépouillés pour en extraire et analyser les comportements défaillants. De manière semblable, des métriques sur la distance entre les véhicules d'un environnement routier simulé sont utilisés pour mettre en évidence les situations les plus critiques [Hallerbach 2018]. Une formalisation des oracles quantifiés pour les systèmes cyber-physiques y est proposée dans [Menghi 2019]. L'idée est ici d'évaluer le degré de satisfaction / violation d'une propriété. Les auteurs définissent un oracle en ligne dont les propriétés sont exprimées dans un fragment de *SFO* (*Signal First Order logic*), une logique du premier ordre qui exprime des propriétés temporelles sur des signaux de valeur réelle [Bakhirkin 2018]. Leur méthode est dédiée aux modèles *Simulink* mais peut être généralisée. Pour faire simple, elle consiste à définir des propriétés sur des signaux. Prenons par exemple la vitesse d'un robot mobile qui doit être maintenu à une certaine valeur v sur un intervalle de temps donné. On peut alors définir une propriété d'oracle vérifiant si la vitesse est constante. Plus la vitesse s'éloignera de la consigne v, plus la sortie d'oracle sera grande. La valeur finale produite par l'oracle est ensuite utilisée pour classer les défaillances par degré de sévérité.

1.4.3 Oracle basé sur des relations métamorphiques

Dans les cas où des propriétés d'oracle satisfaisantes ne peuvent pas être construites, notamment car il est difficile de spécifier un système autonome, une autre approche possible est le test métamorphique [Chen 2015]. L'idée est de vérifier des propriétés (relations métamorphiques) à partir de plusieurs exécutions. Une relation métamorphique nécessite généralement deux exécutions comprenant des entrées différentes. L'exemple classique repose sur un programme hypothétique permettant le calcul très précis d'une valeur de la fonction sinus pour un angle donné. Il est difficile de construire un oracle permettant de vérifier toutes les décimales produites par le programme. En revanche, la périodicité de la fonction sinus permet de construire la relation métamorphique suivante : $sin(x + 2\pi) = sin(x)$. Il est alors possible de vérifier la cohérence des résultats à partir de deux exécutions dont les entrées sont spécifiquement choisies. Toute incohérence indique une erreur dans le programme.

L'approche métamorphique est utilisée dans [Zhang 2018] pour tester le traitement d'images d'un véhicule autonome basé sur un réseau de neurones. D'un point de vue boite noire, le système prend une image de la route (similaire à la vision d'un conducteur) et retourne une prédiction d'angle de braquage. Des images sont artificiellement générées à partir de transformation de clichés originaux. Les nouvelles images sont similaires aux originaux mais peuvent présenter une météo différente

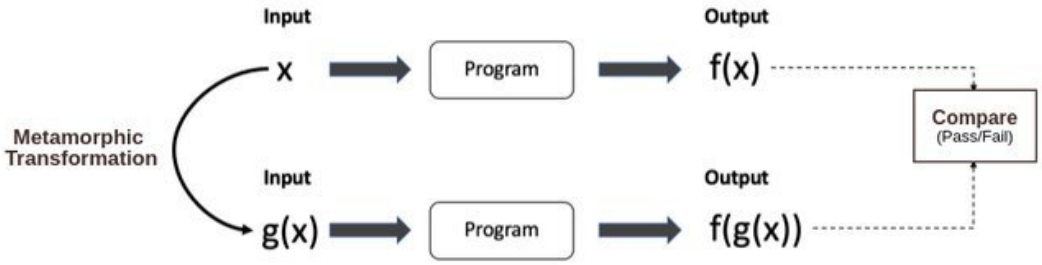


FIGURE 1.4 – Schéma de principe du test métamorphique (source : [Nair 2019]).

ou encore des zones de pixels manquants. Le principe est de comparer les sorties du système de traitement d'images entre les images originales et leurs versions altérées. Un test est alors considéré réussi si les prédictions d'angle de braquage restent dans un intervalle donné. La même idée est présentée dans [Tian 2018] à la différence que leur transformation d'images maximise la couverture de neurones (le ratio de neurones uniques activés sur le nombre total de neurones du réseau). Pour tester des drones autonomes, Les auteurs de [Lindvall 2017] ont manuellement créé des environnements virtuels pour un simulateur de drone basé sous *Unity*. Ces environnements sont conçus spécifiquement pour observer des comportements intéressants. Un algorithme de mutation est ensuite utilisé pour générer des variations de ces cas de test. La totalité des cas de test est ensuite classée dans des groupes de scénarios équivalents. Par exemple, un même scénario orienté vers le sud ou vers l'est doit produire le même résultat (équivalence par rotation). Même chose pour l'équivalence par translation, équivalence du placement des obstacles etc. Les relations métamorphiques pour l'évaluation des résultats sont articulées autour de critères de sécurité (temps dans une zone dangereuse, proximité avec des obstacles etc.). Comparer deux exécutions n'est pas trivial à cause du non déterminisme. En effet, on ne peut pas simplement comparer les valeurs capteurs ou même la trajectoire. La même conclusion est tirée dans [Sotiropoulos 2016] pour le test de la navigation de *Mana* (robot académique tout terrain). Les trajectoires observées pour un même cas de test sont complètement différentes. Déterminer des relations métamorphiques est alors très problématique.

1.5 Génération de tests pour les systèmes autonomes

La génération d'entrée de test est un sujet très étudié dans le milieu académique. Dans le contexte du test de systèmes autonomes ou réactifs c'est l'environnement qui est maître de l'interaction. C'est donc cet environnement qui doit être généré pour leur test. Dans cet objectif, trois approches s'opposent :

— L'approche par reproduction, qui reproduit virtuellement des environnements réels.

— L'approche générative, qui génère des environnements / "scènes" créés de toutes pièces à partir d'un modèle. On appelle "scène" l'arrangement spatio-temporel des différents objets qui forment l'environnement pour un observateur donné [Maurer 2000].

— L'approche par mutation, qui se base sur des cas de test existant et y insèrent des mutations.

Cette dernière approche consistant partiellement en une extension de l'approche générative, nous ne la détaillons pas ci-dessous.

1.5.1 Approche par reproduction de cas réels

Cette approche est principalement utilisée dans le domaine automobile. Les constructeurs ont collecté une grande quantité de données à partir d'essais sur route. Ces données sont ensuite utilisées pour recréer virtuellement les routes et y tester les systèmes d'aide à la conduite (*ADAS*) et les diverses fonctionnalités de conduite autonome. La chaine de transformation des données collectées vers une scène virtuelle est non triviale : elle fait intervenir des méthodes de post-traitement complexes pour extraire tous les paramètres nécessaires pour la génération (topographie, circulation, signalétique etc.).

Dans [Lamprecht 2010], des réseaux routiers réels sont reproduits précisément pour la validation des routes alternatives d'un système de navigation automobile. Les routes virtuelles sont construites à partir d'une base de données et de relevés satellites d'élévations. De façon similaire, dans [Nentwig 2010], une méthode de reproduction virtuelle de cas de test à partir de données réelles est décrite puis la qualité des simulations produites est évaluée. La forme de la route, la trajectoire du véhicule de mesure, les trajectoires et les différents attributs des autres véhicules sont extraits des enregistrements. Le modèle de route est créé en premier à partir des traces GPS et d'une base de données de route. Les véhicules et l'environnement sont ensuite ajoutés puis des rendus sont produits à intervalle d'une seconde. Pour l'évaluation, le même algorithme de détection de véhicule est exécuté sur les images réelles et leur double de synthèse. Les résultats comparables sur les deux types d'images démontrent le potentiel des images générées pour le test de système de vision. Dans [Bach 2017] des cas de test sont extraits d'une large collection de données réelles pour le test des systèmes dépendants de la géolocalisation d'un véhicule. Pour rendre leur campagne de test efficace, les auteurs proposent une méthode de sélection de tests pour maximiser la diversité des scénarios tout en minimisant le nombre de cas de test. Deux étapes se succèdent pour construire l'ensemble de tests final. Les scénarios sont d'abord partitionnés à partir de la spécification en fonction de propriétés abstraites (le type de route, le pays etc.). Dans un second temps, les tests sont évalués un par un pour former le jeu de tests final. Seuls les tests améliorant significativement la couverture des paramètres d'entrée sont ajoutés pour éviter les scénarios redondants.

1.5.2 Approche générative

Cette approche ne nécessite pas de données réelles et permet plus de flexibilité pour la diversité. En revanche, elle implique l'utilisation d'un modèle pour la génération. La spécification de celui-ci est bien souvent non triviale. Dans le cas d'un robot autonome, pour définir un cas de test, il faut obligatoirement définir l'environnement dans lequel le robot doit effectuer sa mission. L'ensemble des environnements possibles est infini et difficile à caractériser. De plus, seul un sous-ensemble est intéressant pour l'exécution de tests. En pratique cet ensemble est déduit à partir de connaissances spécifiques au domaine. Par exemple, pour le test d'un système *ADAS*, on peut imaginer un modèle comprenant un réseau routier avec des intersections, des obstacles statiques et mobiles ou encore différentes conditions météorologiques etc. Plusieurs approches ont été proposées pour la modélisation mais le principe général reste le même : spécifier une vue structurée des différents objets et de leurs relations. Dans [Geyer 2014] et [Ulbrich 2015], des ontologies génériques pour le test en environnement routier sont proposées. Dans [Klueck 2018] des jeux de tests sont générés à partir d'une ontologie similaire. Des paramètres discrets sont associés à chaque facteur d'influence identifié dans l'ontologie. Par exemple, la surface peut être sèche, humide ou gelée. Le trafic sur chaque voie peut être léger, fluide, dense, bouché. Un algorithme de test combinatoire est ensuite utilisé pour générer un jeu de tests minimal comportant toutes les combinaisons de paramètres pour un sous ensemble de cardinal 2 (*pairwise*).

Toujours dans les approches génératives, dans [Sotiropoulos 2016] un diagramme de classe UML est utilisé pour modéliser une carte bosselée et des obstacles. Les auteurs de [Micskei 2012] modélisent les interactions (événements et messages reçus, actions et messages retournés) avec le système sous test par des diagrammes de séquences UML. Une fois un modèle établi et instancié, on peut enfin générer des cas de test concrets. La génération procédurale est une méthode permettant de générer automatiquement des environnements structurés complexes

La génération procédurale est une méthode de création automatique de contenu numérique. Elle est principalement utilisée dans l'édition vidéo pour les effets spéciaux (par exemple du feu ou de la fumée réaliste) et dans le domaine du jeu vidéo pour la création de contenu. Dans le cadre du test des systèmes autonomes, l'idée d'utiliser ce type de technique a été introduite par [Arnold 2013] pour générer des environnements virtuels pour le test d'un petit robot mobile. Initialement, cette technique a permis de résoudre les limitations matérielles des débuts du jeu vidéo. Pour réduire la consommation mémoire, le contenu est généré seulement quand le joueur en a besoin. Par exemple le jeu *Elite* (sorti en 1984) gère 8 galaxies, comprenant chacune 256 systèmes planétaires détaillés, avec seulement une disquette de 20Ko de mémoire. Les auteurs de [Togelius 2011] donnent un tour d'horizon des méthodes de génération procédurales basées sur des métaheuristiques et leurs utilisations de nos jours (création de contenu unique à chaque partie pour faire varier l'expérience de jeu). Plusieurs éléments opposent les différents algorithmes :

— Générer du contenu pendant l'exécution ou a priori.

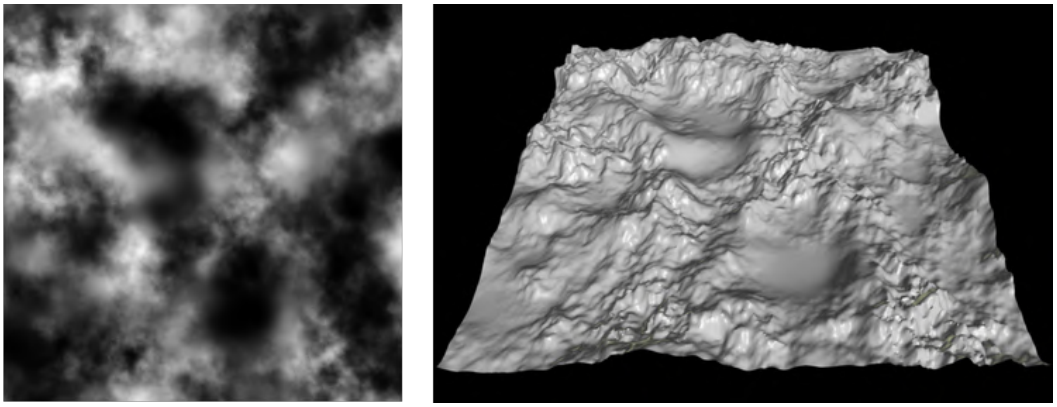


FIGURE 1.5 – Exemple de carte de hauteur et de son rendu 3D.

— La quantité d'information donnée en entrée qui peut aller d'un vecteur de paramètre pour expliciter un modèle précis à un cas plus extrême avec seulement une graine aléatoire (seed).

— Une méthode constructive où les objets générés sont raffinés petit à petit ou une par essais successifs (generate-then-test).

Il est important de noter que les éléments présentés ci-dessus ne sont pas binaires. Un algorithme de génération procédurale peut par exemple générer du contenu avant et aussi pendant l'exécution. Dans les applications liées au test de systèmes autonomes, les objets sont généralement générés a priori à partir d'un vecteur de paramètres. Des variations de ce vecteur permettent ensuite l'exploration du domaine d'entrée.

En résumé, le principe est de décrire un objet à partir d'un ensemble réduit de paramètres qui serviront ensuite au générateur pour créer une instance de cet objet. Le procédé peut être décomposé en trois étapes. D'abord, la création d'un modèle qui contient un ensemble de paramètres abstraits, ensuite l'instantiation des paramètres et pour finir la génération de données concrètes basées sur les paramètres. On peut par exemple imaginer générer un terrain surfacique défini par un paramètre pour la taille et un autre paramètre pour la déformation. Les heightmap (cartes de hauteurs) sont une représentation compacte en mémoire des terrains surfaciques. Elles consistent en une image en niveaux de gris où tous les pixels représentent une valeur d'élévation de la surface (figure 1.5). Pour obtenir un rendu réaliste, l'algorithme de génération doit se baser sur de l'aléatoire tout en conservant une certaine cohérence. Le bruit de Perlin [Perlin 1985] est un algorithme classique permettant de générer ce type d'objet. Dans [Miller 1986], 3 autres techniques dédiées à la génération de cartes de hauteur sont présentées. Une vue plus récente de ces techniques est donnée dans [Shaker 2016].

Dans [Arnold 2013], les techniques de génération procédurale sont utilisées pour tester le système de contrôle d'un robot autonome. Une carte binaire est générée où

chaque case est obstruée ou non. Elle est construite à partir d'un bruit de Perlin dont le résultat passe successivement dans un filtre de pixellisation puis dans un filtre de seuillage. Des robots sont ensuite placés aléatoirement sur des cases non obstruées puis un itinéraire leur est assigné. Une approche de génération procédurale similaire est présentée dans [Sotiropoulos 2016], une carte parsemée d'obstacles est générée pour le test d'un robot autonome. La surface est créée à partir d'un modèle 3D de plan (*mesh*) déformé via *Blender* (logiciel de modélisation 3D). Quatre paramètres permettent la création de celle-ci. D'abord, le *mesh* du plan est ajusté à la taille désirée (*size*). Il est ensuite subdivisé en une grille de taille (subdivision). Pour finir, il est déformé selon l'axe vertical (*deformation*, *smoothness*). Deux types d'obstacles sont ensuite ajoutés : des bâtiments et des arbres. La vision du robot étant basée sur un LIDAR 2D, des cubes de taille adaptée suffisent à aproximer les obstacles. La génération se termine quand le taux d'obstruction désiré (*percentage_obstruction*) est atteint.

1.6 Génération de données structurées

Cette section détaille les différents aspects liés à l'approche générative présentée précédemment. Le champ applicatif est ici élargi au test en général (pas uniquement dans le contexte des systèmes autonomes) et à la génération de données structurées. Les problématiques liées à la génération nous sont parvenues par l'intermédiaire de nos travaux sur les systèmes autonomes (voir chapitre 2) mais ont abouti à des travaux plus globaux, présentés en deuxième partie de ce manuscrit (partie II).

Les données structurées peuvent prendre des formes très variées. Pour garantir des tests de qualité, il faut des entrées (données structurées) conformes au format attendu par le système sous test. En d'autres termes, elles doivent satisfaire un ensemble de contraintes logiques. Par exemple, pour un système qui prend des images au format *bitmap* (*.bmp*) en entrée, le générateur hypothétique associé doit respecter le format et donc faire correspondre l'image à ses méta-données présentes en entête (hauteur, largeur, type d'encodage, offset du début des données etc.). Deux aspects principaux, présentés dans les sous-sections suivantes, entrent en jeu concernant ces générateurs :

— Le langage de modélisation : il doit être le plus expressif possible et permettre de décrire des structures de données complexes comportant des contraintes sémantiques entre les différents éléments.

— La méthode de génération : elle s'occupe de la génération et assure le respect des contraintes. Elle doit être relativement efficace. Dans le cas du test, une génération rapide permet d'allouer plus de temps à l'exécution des tests et donc d'avoir une meilleure couverture.

1.6.1 Langages de modélisation des cas de test

Les langages de modélisation sont presque aussi nombreux que les outils de génération disponibles. Ils ont leurs spécificités et sont plus ou moins expressifs. *QuickCheck* [Claessen 2000] ou encore *GodelTest* [Feldt 2013], [Poulding 2014] sont des *framework* pour la génération de test. Ils assurent une expressivité maximale en reposant sur l'utilisateur pour la programmation de la structure de donnée (avec des langages de programmation à usage général, *Haskell* pour *QuickCheck* et *Ruby* pour *GodelTest*). Ils fournissent des éléments de base pour la génération non déterministe de structures complexes. Ce type de *framework* aide le testeur à exprimer des structures de données arbitraires. En revanche, il n'est pas compatible avec une approche automatique basée sur le modèle, où seulement une description haut niveau entièrement descriptive est donnée.

XML est un méta-langage, il est utilisé pour des applications très variés. Ce formalisme peut également être utilisé pour la description de modèle. Il a l'avantage d'être facilement lisible par une machine et par un humain. *SNOOZE* [Banks 2006] est un *fuzzer* (outil d'injection de données aléatoires dans les entrées d'un programme) spécialement conçu pour le test de protocole. Le langage ne prend pas en charge les contraintes et ne permet donc pas la gestion des aspects liés à la sémantique. *Peach* [PEACH 2020] est un *fuzzer* commercial, son langage de description du modèle est également basé XML. Il incorpore la possibilité d'insérer des références vers des fonctions développées par l'utilisateur. Cette fonctionnalité permet la gestion des relations sémantiques. En contrepartie, elle ajoute de l'effort de développement et d'intégration.

Alloy [Jackson 2019] est un langage purement déclaratif adapté à la spécification de données ayant des structures de données complexes. Ce langage se base sur des relations logiques du premier ordre. Il est particulièrement adapté pour l'analyse automatique et est populaire dans la communauté du test pour la création d'instances de modèle. Il est par exemple utilisé dans le *framework* (infrastructure logicielle) *TestEra* [Marinov 2001] pour le test de programme Java. L'analyse d'un modèle *Alloy* passe par un solveur SAT, il en résulte des limitations d'expressivité, notamment pour les contraintes numériques.

Les grammaires formelles sont une bonne approche pour la spécification de modèles de données structurées. Elles sont utilisées dans le domaine du test depuis plusieurs décennies [Purdom 1972], [Duncan 1981], [Maurer 1990], [Coppit 2005], [Xu 2010], [Pan 2013], [Kifetew 2017], [Cekan 2017]. A l'origine, seules les grammaires sans contexte (type 2) étaient utilisées pour le test. Des extensions ont ensuite été étudiées pour améliorer l'expressivité. Dans [Maurer 1990], l'idée de pondérer les règles de production de la grammaire pour en contrôler la sélection est développée. Des extensions liées au contexte sont également proposées. Elles consistent en des variables et des bouts de code fournis par l'utilisateur (action routines) qui sont associés aux règles de production. Des morceaux de données peuvent alors être générés en fonction du contexte. Dans d'autres travaux, une grammaire attribuée est utilisée pour ajouter des informations sensibles au contexte. C'est par exemple

le cas du *framework Yagg* [Coppit 2005], qui ajoute des vérifications de contrainte aux règles syntaxiques de production des données. Dans [Cekan 2017], une règle peut changer la probabilité d'une autre. Dans [Kifetew 2017], le contexte permet de maintenir des informations sur le type des variables. Cela permet entre autre de générer des expressions valides pour des entiers ou des flottants. Les auteurs de [Pan 2013] emploient une grammaire attribuée d'ordre supérieur pour leur fuzzer. Des mutations sont appliquées directement sur l'arbre syntaxique et les autres noeuds sont mis à jour par du code défini par l'utilisateur.

Dans d'autres travaux sur la génération de cas de test, le modèle est exprimé sous forme de diagrammes de classe UML et de contraintes OCL (*Object Constraint Language* [OCL 2020]). La principale distinction est le degré de couverture de la spécification UML / OCL par le générateur. Par exemple, le générateur dans [Cantenot 2014] prend un modèle défini par un profil UML / OCL, spécialisé pour le test. Les restrictions sur le langage UML sont structurelles (l'héritage et le polymorphisme ne sont pas supportés) et sémantiques (seulement un fragment de OCL est géré et certaines contraintes sont interprétées de manière non standard). A l'inverse, *PLEDGE* [Soltana 2020] est un *framework* qui couvre un large sous ensemble des notations UML / OCL.

Dans la deuxième partie de ce manuscrit (partie II), nous présenterons un prototype de générateur dont les modèles sont basés sur XML. Le langage est purement déclaratif, il incorpore des contraintes et des fonctionnalités d'échantillonnage des variables (chapitre 4).

1.6.2 Génération des tests sous contrainte

Il est assez direct de générer une structure de données sans contraintes. Les contraintes non contextuelles ne posent pas de problème non plus (par exemple une valeur limite maximum pour une variable). En revanche, La présence de contraintes sémantiques dans un modèle est souvent nécessaire pour modéliser des structures de données complexes. La génération peut alors vite devenir problématique. Les solutions utilisées par la plupart des outils sont des variantes de l'approche *generate-then-filter* (générer puis filtrer, c.a.d. générer un grand nombre de tests, puis éliminer ceux qui ne respectent pas les contraintes). C'est par exemple le cas de *Yagg*, un des outils basés sur une grammaire formelle cités dans la sous-section précédente. *Yagg* (*yet another generator generator*) est un outil de génération d'entrée de test [Coppit 2005]. A partir d'une description du format d'entrée basée sur un langage similaire à LEX / YACC, il génère un programme C++ capable de lister toutes les entrées possibles ne dépassant pas une longueur limite fixée par l'utilisateur. L'implémentation consiste à générer un arbre syntaxique de profondeur donnée puis à parcourir toutes les branches. Le fichier de grammaire proche de LEX / YACC permet uniquement d'exprimer la partie sans contexte (*context-free grammar*). Des bouts de code appelés action blocks doivent être ajoutés pour la gestion du contexte. Ils agissent comme un filtre sur les entrées générées sans contexte. *Yagg* n'est pas le plus performant des outils disponibles, son intérêt réside princi-

palement dans son langage proche de LEX/YACC qui facilite l'apprentissage des testeurs initiés.

La principale limitation de l'approche *generate-then-filter* est que la performance dépend fortement du niveau de contrainte. Dans le cas d'un domaine d'entrée très contraint, le générateur a peu de chance de générer un cas de test valide, tout va être filtré. Le nombre de candidats invalides peut être réduit par une gestion de contexte judicieuse. Il faut pour cela ajuster le générateur en y intégrant des bouts de code. Ce code doit être écrit spécifiquement pour le modèle par le testeur. Il doit encoder la récupération des éléments contextuels puis en gérer l'ordre de génération. Dans [Kifetew 2017] des cas de test sont générées pour la validation d'un compilateur/interpréteur *JavaScript*. Les cas de test pour un tel système sont des programmes *JavaScript* et doivent donc en respecter la spécification. La spécification consiste en 331 règles de grammaire et comprend beaucoup d'éléments récursifs. La principale difficulté pour la génération est de cibler certains cas de tests pour respecter certains critères. Les auteurs introduisent un langage d'annotation qui se greffe à la spécification de la grammaire. Ces annotations sont écrites de manière à guider le générateur pour la production de cas de test respectant certaines règles sémantiques. Cette méthode augmente le nombre de cas de test valides parmi tous les tests générés. Une étape de filtrage est toujours nécessaire. Dans [Pan 2013] le modèle est spécifié à l'aide d'une grammaire attribuée d'ordre supérieur. Ce type de grammaire est défini dans [Swierstra 2006] comme une extension des grammaires attribuées classiques en retirant la frontière entre le domaine de l'arbre syntaxique et le domaine des attributs. Les attributs sont calculés en même temps que l'arbre syntaxique. L'évaluation des attributs permet alors la gestion de contraintes sémantiques. La méthode nécessite des échantillons de données. Ces échantillons sont sélectionnés puis utilisés pour construire un arbre syntaxique à partir du modèle. Des noeuds sont ensuite choisis et mutés de manière à maximiser la couverture. Enfin, les autres noeuds sont modifiés afin de respecter les règles des attributs. Il n'est pas forcément possible de respecter toutes les règles. En revanche, cette méthode permet de réduire le nombre de cas de test incorrects ou dupliqués.

Il est également possible d'améliorer le processus de génération en ajustant judicieusement les probabilités de chaque règle pour favoriser les cas de test cohérents. Les générateurs classiques basés sur des grammaires sélectionnent les règles de manière équiprobable (loi uniforme). Les générateurs acceptant des grammaires pondérées ou des modèles stochastiques permettent d'adapter la loi de probabilité pour la sélection des règles. Dans [Kifetew 2014], les cas de test générés pour la validation d'un compilateur / interpréteur *JavaScript* sont issus d'un générateur combinant grammaire stochastique et algorithme génétique. La fonction objectif fait intervenir les pondérations des règles de grammaire de manière à maximiser la couverture. Similairement, dans [Poulding 2014] les pondérations sont optimisées par une variante de recherche arborescente Monte Carlo. Une autre approche consiste à extraire ces pondérations d'un ensemble d'exemples bien construits. Si la grammaire n'est pas ambigüe, alors il existe un unique arbre syntaxique correspondant à chaque exemple. On peut alors compter le nombre d'utilisation de chaque règle

dans l'arbre syntaxique. C'est en répétant l'opération sur l'ensemble des exemples disponibles que l'on déduit les pondérations.

Une autre solution est d'utiliser un solveur de contrainte pour assurer la validité des cas de test. Les contraintes sont alors exprimées dans un formalisme basé sur la logique du premier ordre. L'outil *TestEra* [Marinov 2001] utilise un solveur SAT (Alloy Analyser [Jackson 2000] pour générer exhaustivement les cas de test spécifiés. Dans [Cantenot 2014] le générateur gère une plus grande expressivité du modèle grâce à une description UML et OCL. La génération nécessite alors l'utilisation d'un solveur SMT. A la différence d'un solveur SAT, un solveur SMT permet de combiner des formules de logique du premier ordre sans quantifieur avec des théories supplémentaires telles que les nombres réels ou les tableaux. La principale limitation des générateurs basés sur un solveur est le passage à l'échelle. Il faut limiter le nombre de variables pour éviter l'explosion combinatoire.

Dans le but de résoudre cette limitation, *PLEDGE* [Soltana 2020] implémente une approche générative hybride combinant l'utilisation d'une métaheuristique et d'un solveur SMT (Satisfiabilty Modulo Theories) afin de tirer parti de leur complémentarité. Un outil démonstrateur *PLEDGE* est proposé et les performances sont évaluées sur trois cas d'étude. Leurs modèles sont spécifiés en UML et les contraintes en OCL. La génération se produit en deux temps, les contraintes sont d'abord réécrites sous forme normale négative (FNN) puis la résolution de celles-ci est distribuée entre la métaheuristique et le solveur. Plus précisément, les contraintes impliquant des changements du nombre d'instances d'un élément ou des changements de liens sont données à la métaheuristique et le reste au solveur. La métaheuristique choisie est *AVM* (*Alternating Variable Method*) [Kempka 2015] car elle a été prouvée efficace pour la résolution de contrainte OCL par les auteurs de [Ali 2013]. Le solveur utilisé est Z3.

La méthode de génération présentée en deuxième partie de ce manuscrit est également une méthode hybride dans la même veine que *PLEDGE*. Une comparaison entre les deux outils est proposée dans le chapitre 6.

1.7 Conclusion

Le test des systèmes autonomes est un problème ouvert. Les méthodes utilisées aujourd'hui reposent principalement sur le test empirique en environnement réel. Les cas de test sont exécutés dans l'environnement disponible qui n'est généralement pas modulable. La diversité vient alors principalement des missions et pas de l'environnement dans lequel se déroulent les tests. Dans ce contexte, la simulation est une approche prometteuse pour automatiser le test des systèmes autonomes et permettre plus de diversité à moindre coût. Cependant au regard de notre étude bibliographique trois problèmes principaux restent ouverts :

— La validité des résultats du test en simulation en fonction du niveau de fidélité de la simulation est encore un problème ouvert bien qu'il existe de nombreux simulateurs. Il est donc nécessaire de conduire des expérimentations compa-

rant les résultats pour un même système sous test.

— La définition de l'oracle face à l'indéterminisme induit par le système lui-même et le simulateur ne peut reposer sur de simples comparaisons entre observations et résultats attendus. Les systèmes autonomes sont déployés dans des environnements variés pour la réalisation de missions complexes. Cette autonomie rend impossible de prédire le comportement exact des systèmes à l'avance. De plus, les systèmes autonomes sont construits sur le principe des systèmes concurrents, ce qui induit du non déterminisme. Le comportement observé diffère entre les différentes exécutions. Lors de la validation de ces systèmes, les oracles se limitent donc généralement à des vérifications de propriétés élémentaires comme l'absence de collision. C'est insuffisant pour en évaluer un niveau de confiance et pour remonter jusqu'à l'erreur responsable.

— Il n'existe pas d'outil permettant la génération de cas de test divers et contraints, à partir d'une spécification structurée des données. La qualité des tests générés va directement dépendre de la qualité du modèle. Un élément non modélisé n'apparaîtra jamais dans un cas de test. La définition du modèle pose un problème de spécification, elle demande souvent des connaissances d'expert du système sous test. De plus, l'environnement dans lequel évolue un système autonome est généralement complexe. Sa modélisation requiert un langage d'une grande expressivité capable de décrire des structures de données complexes comportant des contraintes. Générer automatiquement ce type de données est très difficile. Plusieurs approches sont possibles. L'approche de type *generate-and-filter* est courante. En effet, il est difficile de générer une donnée respectant les contraintes mais il est aisé de vérifier si une donnée les respecte. Le principe du *generate-and-filter* et de ses variantes consiste à générer une donnée sans prise en compte des contraintes puis de la filtrer par une analyse des contraintes. Cette méthode montre vite ses limites face à des données très contraintes. Les autres approches reposent soit sur du code inséré par le testeur pour contourner le problème soit sur un solveur. Dans le cas de l'utilisation d'un solveur se pose alors la question du passage à l'échelle et de la diversité des résultats. Des méthodes combinant solveurs et méta heuristiques sont envisageables mais sont encore rares dans la littérature.

Face à ces considérations nous proposons donc de réaliser en premier lieu une étude expérimentale pour estimer la validité du test en simulation par rapport au test sur le terrain. Ce premier travail est présenté dans la première partie de ce manuscrit (chapitres 2 et 3), et se focalise sur l'analyse d'une campagne de test en simulation avec pour cas d'étude un robot agricole autonome. Les résultats obtenus en simulation sont comparés aux résultats des tests sur le terrain et une analyse détaillée des fautes est proposée. Une évaluation de l'impact de l'indéterminisme sur nos résultats est également introduite. Cette partie couvre les grandes problématiques relatives aux tests des systèmes autonomes et propose des éléments de réponse. La deuxième partie du manuscrit (chapitres 4 à 6) est centrée sur la génération d'entrées de test.

La mise en place de la précédente plate-forme de test nous a montré à quel point il peut être laborieux de développer un générateur d'entrée de test propre à un cas d'étude. En effet le générateur doit pouvoir produire des cas de test cohérents et variés. Une grande partie de cette tâche peut s'abstraire du cas d'étude et être automatisée. Dans cet objectif, nous proposons une méthode hybride de génération d'entrée de test à partir d'un modèle purement déclaratif.

Première partie

Campagne de test en simulation et analyse des fautes d'un robot agricole

Présentation générale

Cette partie présente l'étude du test en simulation d'un robot industriel agricole développé par Naïo technologies : le robot *Oz* ([ROBERT 2020d]). Nous avons contribué à ces travaux d'abord dans le cadre d'un stage de fin d'études d'ingénieur, puis dans le cadre de notre thèse.

L'exemple du robot *Oz* permet d'illustrer les problématiques de génération d'environnements virtuels complexes, de spécification du comportement attendu (oracle de test), et de fidélité imparfaite de la simulation par rapport à des tests sur le terrain. Nous expliquerons comment ces problématiques ont été traitées dans le cadre de la conception d'une plate-forme de test en simulation pour *Oz*. La conception de la plate-forme est ici rappelée à titre illustratif, car elle est antérieure au début de notre thèse. Elle a été décrite dans le chapitre 4 du manuscrit de Thierry Sotiropoulos [Sotiropoulos 2018]. Nous y avons néanmoins contribué lors du stage réalisé. Notamment, nous avions proposé une modélisation des cas de test sous forme de grammaire attribuée. La partie hors contexte de la grammaire donnait la structure des éléments à générer, et l'ajout d'attributs et de règles sémantiques permettait de spécifier leurs propriétés. Ces travaux initiaux ont inspiré nos développements ultérieurs sur la génération automatique de données structurées avec des contraintes, qui seront détaillés dans la deuxième partie de ce manuscrit.

Dans le cadre de notre thèse, la contribution principale à l'étude de *Oz* porte sur la comparaison des tests en simulation et sur le terrain. Au début de l'étude, l'opinion qui prévalait chez les ingénieurs de Naïo était que la validation de leurs robots devait s'effectuer en condition réelle par des tests sur le terrain. Les tests en simulation étaient jugés trop simplifiés et insuffisamment réalistes pour révéler les fautes logicielles autres que triviales. Cette opinion sur la nécessité de conditions réelles est, à notre connaissance, largement partagée au sein de la communauté robotique. Au contraire, notre hypothèse était que les ingénieurs de Naïo (et plus généralement les acteurs de la communauté robotique) sous-estimaient l'efficacité potentielle du test en simulation. Pour valider cette hypothèse, l'étude a mené en parallèle des tests virtuels (au LAAS) et réels (par Naïo). Le logiciel sous test était en cours de de développement, et ses fautes n'étaient connues ni de l'industriel, ni de nous. Nous avons réalisé une analyse approfondie des résultats obtenus pour les deux types de test. L'analyse a confirmé notre hypothèse. Malgré leur fidélité imparfaite, les tests en simulation révèlent la plupart des fautes trouvées par les expérimentations avec le robot réel, y compris celle ayant causé la majorité des défaillances sur le terrain. Ils révèlent également une nouvelle faute passée inaperçue sur le terrain mais confirmée par Naïo.

Nous décrivons ci-dessous le robot *Oz* et le simulateur fourni par Naïo Technologies pour nos expérimentations. Nous donnons ensuite une vue d'ensemble des deux chapitres qui vont présenter l'étude du test de ce robot.

Cas d'étude *Oz* : robot et simulateur

Oz est un robot désherbeur autonome spécialisé pour les cultures maraîchères. La figure 1.6 montre le robot *Oz* en fonctionnement. Sa petite taille (75 cm × 45 cm × 55 cm) lui permet de naviguer entre les rangées de légumes du champ. Le désherbage est effectué mécaniquement, à l'aide d'un outil spécifique (sarcleuse) fixé à l'arrière du robot et tiré par celui-ci. Lorsqu'il y a plusieurs inter-rangs à désherber, le robot doit faire un demi-tour à la fin d'une zone de travail pour passer à la suivante. Dans cette partie nous utiliserons indifféremment "inter-rang" ou "zone de travail" pour exprimer l'espace à désherber entre deux rangées de légumes. Dans chaque inter-rang le robot doit également décider si le désherbage se fait en un ou deux passages, en fonction de leur largeur. Ceci est illustré dans l'exemple de mission figure 1.7 : la zone de travail entre les deux premières rangées peut être désherbée en un seul passage, mais celle d'après (entre la deuxième et troisième rangées) est un peu plus large et nécessite deux passages.

FIGURE 1.6 – Le robot agricole *Oz*.

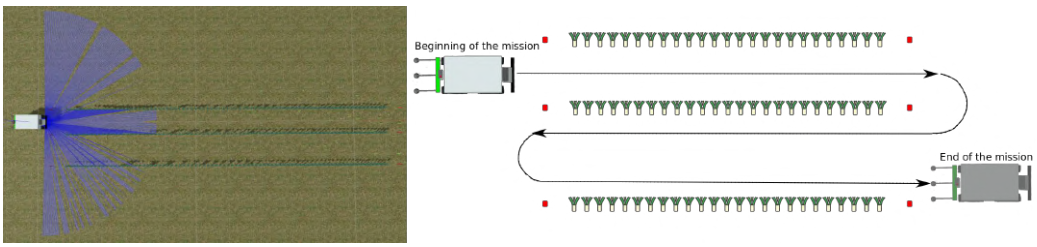


FIGURE 1.7 – Vue en plongée du cas nominal (à gauche : Gazebo, à droite : schéma).

Le robot *Oz* perçoit son environnement à l'aide d'un capteur de type nappe laser pour la télédétection (LiDAR 2D) à l'avant, ainsi que de deux caméras. Dans la version du logiciel étudiée, la gestion de la trajectoire le long des rangées de légumes repose uniquement sur le lidar. Les caméras sont principalement utilisées pour les phases de demi-tours. Des piquets de couleur rouge délimitent les extrémités de chaque rangée de légumes. Ils sont placés à environ 50 cm du premier et du dernier

légume de la rangée. Le robot les identifie à l'aide des caméras et les utilise comme repère pour ses manoeuvres de demi-tour. Les caméras permettent également d'utiliser des techniques d'odométrie visuelle stéréo pour détecter les dérapages excessifs du robot lors des demi-tours. Il est à noter que les dérapages sont inévitables de par la structure du robot. En effet, l'axe des roues est fixe. Le robot tourne en exerçant une vitesse différente de chaque coté à l'instar d'un tank. Enfin, des capteurs de contact (bumpers) déclenchent un arrêt d'urgence en cas de collision. Ces dispositifs de sécurité permettent d'éviter les risques de blessures humaines. Étant donné la faible vitesse maximale de *Oz* ($0,4m.s^{-1}$), son poids modéré (120 kg avec les outils) et le fait que les outils de désherbage ne sont pas dangereux en eux-mêmes, le robot n'est pas considéré dangereux. Il est donc tout à fait acceptable pour l'utilisateur de se situer dans un voisinage proche du robot pendant qu'il effectue sa mission. Néanmoins, le fonctionnement du robot n'est pas sans risque. Envisageons par exemple une trajectoire mal contrôlée : les légumes pourraient être endommagées, ou pire, le robot pourrait atteindre une zone dangereuse en dehors du champ (une route par exemple).

Naïo technologies a développé un simulateur de type SiL (*Software-in-the-Loop*) pour tester le système de navigation de leur robot *Oz*. La figure 1.8 donne un aperçu de l'architecture de test correspondante. La plate-forme est basée sur *Gazebo*, un simulateur fréquemment utilisé dans la recherche en robotique. La figure 1.9 présent des vues d'Oz sous Gazebo. Le logiciel testé, *OzCore*, est une version R&D du contrôleur du robot. Il reçoit les données de ses capteurs simulés et produit des commandes d'actionneurs en conséquence. Les effets de ces actionneurs sont ensuite simulés pour mettre à jour les données des capteurs et ainsi de suite. *OzCore* est écrit en C et C++ avec un total d'environ 151 KLOC. Ce code comprend certaines fonctions de *Oz* qui ne sont pas gérées par le simulateur, comme par exemple le mode téléguidé où le robot est directement contrôlé par l'utilisateur à l'aide d'une télécommande. De plus, l'effet des outils de désherbage n'est pas simulé, donc les commandes de sortie correspondantes sont ignorées. Comme le montre la figure 1.8, le simulateur ne reçoit que les commandes de vitesse des roues du robot. Une autre différence notable est la méthode de programmation d'une mission. Pour utiliser le vrai robot, l'agriculteur doit paramétrer la mission via une interface utilisateur. En simulation, un fichier *.json* contenant tous les paramètres de mission est directement donné au robot.

Le simulateur nécessite trois entrées pour instancier un champ virtuel : une image *.jpg* encodant le relief du terrain et deux fichiers *.sdf* pour les autres éléments du champ (légumes, piquets rouges) et les paramètres de mission.

La figure 1.7 est représentative de la complexité que peut avoir un cas de test compte tenu des performances du simulateur. Le champ virtuel comporte trois rangées de légumes et une faible densité de mauvaises herbes dans les zones de travail. De plus, la longueur des rangées est faible par rapport à celle d'un vrai champ. Malgré un champ de taille limitée, ce cas de test permet de cibler de nombreux aspects importants d'une mission, comme la perception des rangées de légumes en présence de mauvaises herbes, le suivi des lignes, la détection de fin de rangée, deux

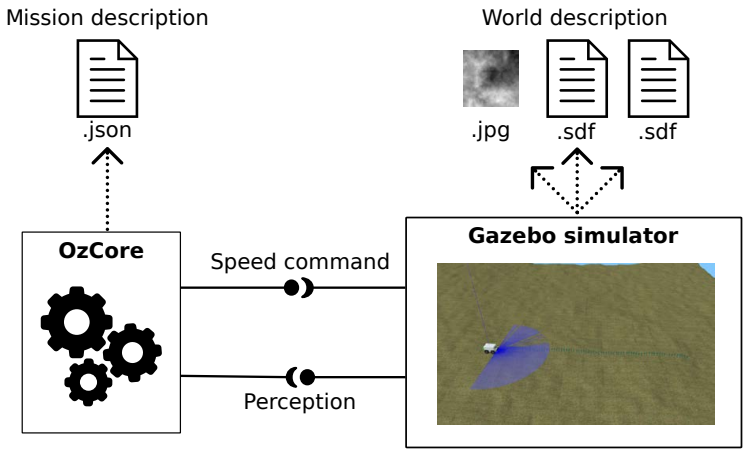


FIGURE 1.8 – Architecture de la plateforme de simulation SiL.

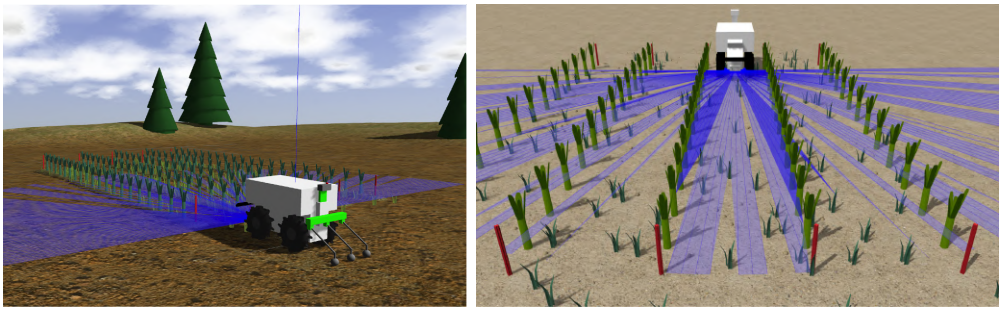


FIGURE 1.9 – Oz dans son champ sous Gazebo

demi-tours avec des configurations différentes, et la décision entre un passage ou deux. Ce cas spécifique est utilisé par Naïo pour repérer les défaillances évidentes en simulation avant d'aller sur le terrain.

Les problèmes de performance ne limitent pas seulement la complexité des cas de test, ils obligent également à faire des compromis sur la physique simulée. Initialement, les ingénieurs de Naïo avaient implémenté un modèle assez fin de l'interaction entre les roues et le sol. Il était jugé important de reproduire l'effet de la boue, des pierres, d'un sol glissant, etc. Cependant, ce modèle s'est avéré trop exigeant en termes de ressources de calcul. Pour donner une idée, une de nos machine équipé de 2 CPU Quad core Intel Xeon E5-2623 v3s à 3,5 GHz et de 64 Go de RAM n'a pas suffit pour exécuter le cas de test de la figure 1.7 avec ce modèle d'interaction entre les roues et le sol. Ce cas de test ne présente pourtant qu'un terrain plat et aucun des éléments stressants susmentionnés. Un modèle alternatif et très simplifié des roues est donc fourni pour les tests : les roues sont abstraites comme de simples cylindres et la physique ignore la friction au sol. Dans cette partie, l'étude des tests en simulation utilise cette version basse-fidélité de la plate-forme.

Présentation des chapitres à venir

Cette partie est découpée en deux chapitres présentant le test du robot *Oz* en simulation. Coté LAAS, le système sous test était une boîte noire, sans accès au code source.

Le chapitre 2 est essentiellement un rappel des travaux réalisés dans la thèse de Thierry Sotiropoulos [Sotiropoulos 2018], auxquels nous avons participé lors d'un stage. Nous y présentons la modélisation de l'environnement pour la génération de cas de test, la spécification de l'oracle et une première analyse des résultats en simulation.

Le chapitre 3 constitue le coeur des contributions de cette partie du manuscrit. Les mêmes résultats y sont d'abord ré-analysés de manière plus approfondie. Nous identifions finement les cas de défaillance en simulation, et faisons l'hypothèse de différents problèmes logiciels susceptibles de les causer. Nous avons ensuite eu accès aux résultats des tests terrain effectués indépendamment par les ingénieurs de Naïo : les défaillances observées, et le diagnostic réalisé par nos partenaires industriels. Une analyse comparative des fautes trouvées permet alors de conclure sur l'efficacité du test en simulation.

Chapitre 2

Oz : Contexte et travaux antérieurs

Sommaire

2.1 Introduction

La plate-forme de test du robot *Oz* se compose de trois éléments :

— Un simulateur du robot fourni par Naïo technologies

— Un générateur de cas de test

— Un oracle de test.

La présentation générale de cette partie I a introduit le simulateur, et a expliqué son caractère basse fidélité pour la physique du robot. Ce chapitre se concentre sur la conception des autres éléments de la plate-forme, qui ont été développés au LAAS. Ils illustrent les problématiques de génération d'environnements 3D complexes, et de vérification automatisée du comportement d'un système autonome.

Pour *Oz*, les cas de test à générer comprennent un champ de légumes virtuel et une mission de désherbage dans ce champ. A un niveau concret, la description d'un champ et d'une mission prend la forme de fichiers dans des formats spécifiques (basés XML, ou .jpg) qui peuvent être compris par le simulateur. La thèse de Thierry Sotiropoulos [Sotiropoulos 2018] propose une génération en deux temps

pour de tels environnements virtuels. Elle part d'un modèle abstrait, qui capture les caractéristiques importantes des éléments à générer (par exemple, le nombre de rangées de légumes, leur longueur, etc...). Dans un premier temps, un ensemble de valeurs de paramètres sont générées à partir du modèle : elles décrivent un cas de test abstrait. Dans un deuxième temps, le cas de test est concrétisé en utilisant des techniques de génération procédurales de contenus de mondes et en formatant les données ainsi produites.

La difficulté de cette approche réside dans la production du modèle de départ, qui spécifie les mondes et missions auxquels le robot peut être confronté. Une telle spécification est difficile dans l'absolu, et va donc se baser sur des cas d'utilisation. Pour *Oz*, la modélisation a nécessité de nombreuses interactions avec les ingénieurs de Naïo, auxquelles nous avons participé. Notre apport conceptuel a été de proposer une modélisation sous forme de grammaire attribuée, qui donne une vue structurée des éléments de mondes et de mission, de leurs paramètres caractéristiques et des propriétés sémantiques liant ces paramètres. L'exercice d'implémentation d'un générateur de cas de test conformes à ce modèle a fortement inspiré la suite de nos travaux : nous en avons tiré une prospective générale sur la génération automatique de données structurées contraintes.

La spécification de l'oracle de test a elle aussi été difficile. Les discussions avec Naïo Technologies ont permis de définir un ensemble de propriétés ne devant jamais être falsifiées durant les tests. Cette spécification a été guidée par les cinq grandes classes de propriétés identifiées par Sotiropoulos et al. [Sotiropoulos 2017] : des propriétés associées aux phases de mission, des invariants basés sur des seuils liés aux mouvements du robot, l'absence d'évènement catastrophique, des propriétés liées aux rapports d'erreur et des propriétés liées à la perception. Par la suite, nous avons pu confirmer la pertinence de ces grandes classes de propriété pour une autre étude de cas (un robot sous-marin) que nous évoquerons brièvement en fin de ce chapitre.

Ce chapitre est structuré comme suit. D'abord, la première section (section 2.2) présente la génération de cas de test et la modélisation correspondante des entrées de test. La spécification de l'oracle est ensuite détaillée dans la section 2.3. La troisième section (section 2.4) présente une première analyse des résultats du test en simulation. Cette analyse sera approfondie au chapitre 3. Pour finir, le section 2.5 montre la même méthode de spécification d'oracle appliquée au test d'un robot sous-marin.

2.2 Génération de cas de test

Le simulateur prend en entrée des fichiers de description d'un champ de légumes (position des légumes et des piquets rouges et relief du sol) et de la mission de désherbage correspondante. Ces fichiers d'entrée sont de bas niveau, ils encodent tous les détails nécessaires pour la création d'un environnement virtuel dans un format compris par le simulateur. Pour générer de tels fichiers, il est pratique d'aborder

le problème de génération à un niveau d'abstraction plus élevé. Cela donne un processus en deux étapes qui génère d'abord des descripteurs de cas de test abstraits à partir d'un modèle, puis produit les fichiers concrets. Le modèle est la base du processus de génération, c'est un élément clé de notre plate-forme de test. Il nous a fallu plusieurs réunions avec les ingénieurs de Naïo pour définir son contenu.

Après un bref aperçu du processus de génération en deux étapes, cette section se concentre sur la façon dont nous avons conçu le modèle du monde. Certains aspects de l'implémentation sont également présentés.

2.2.1 Génération en deux étapes

Pour la génération des entrées de test de bas niveau, nous proposons de mettre en oeuvre des techniques dérivées de la génération procédurale de contenus de monde ([Togelius 2011]). L'idée principale est d'utiliser un ensemble de paramètres de haut niveau (appelé le **génotype**) pour contrôler la production du contenu concret de bas niveau (appelé le **phénotype**). Ce processus est généralement basé sur des fonctions aléatoires, un même génotype peut aboutir à la production de différents phénotypes.

La figure 2.1 montre notre approche de génération en deux étapes avec la génération procédurale. Les paramètres du génotype sont issus du modèle de mondes. Pour illustrer le processus de génération, prenons l'exemple de la génération du terrain avec relief. Notre modèle du terrain prend un paramètre : `roughness` qui représente le degré d'irrégularité du terrain. La première étape consiste à sélectionner aléatoirement la valeur de ce paramètre pour former un génotype. Ensuite, la deuxième étape génère un fichier de terrain concret pour le simulateur : le phénotype. Dans notre exemple, il correspond à une image qui encode une carte d'altitude (*heightmap*), comme montré dans la partie droite de la figure 2.1. Le niveau de gris en un point correspond à l'altitude de ce point. La production de l'image s'effectue par exemple en utilisant une fonction dérivée du bruit de Perlin : le niveau de bruit est alors contrôlé selon la valeur du paramètre `roughness` sélectionnée à l'étape précédente.

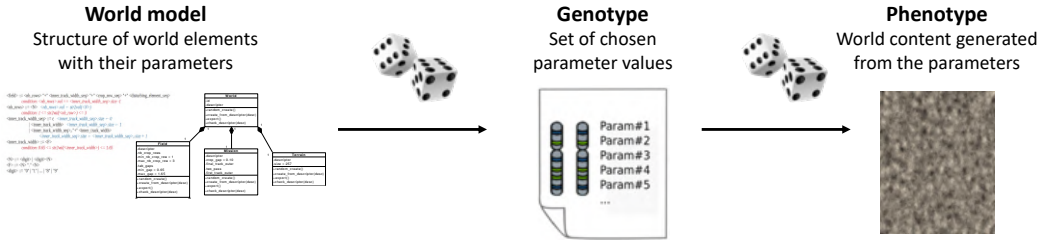


FIGURE 2.1 – Principe de génération en deux étapes. Les dés représentent la part aléatoire à chaque étape : (1) génération de valeurs pour instancier tous les paramètres du modèle, (2) génération de données concrètes.

L'approche en deux étapes permet de maintenir le modèle à un haut niveau

d'abstraction. La modélisation peut alors se concentrer sur l'identification des principaux éléments et de leurs caractéristiques macroscopiques (par exemple, le terrain et son degré de déformation) sans être submergée par la complexité des détails microscopiques (par exemple, la hauteur précise du terrain en chaque point) ou les idiosyncrasies de la plate-forme de simulation (par exemple, le format d'image à produire).

2.2.2 Définition du modèle de cas de test

La modélisation commence par l'identification puis la hiérarchisation des différents éléments qui composent un cas de test. A chaque élément est associé un ou plusieurs paramètres de haut niveau. Une première représentation est réalisée à l'aide de diagrammes de classes UML, qui visualisent la structure proposée. Cette représentation à l'avantage d'être facilement lisible : elle nous a été très utile pour discuter du modèle avec les ingénieurs de Naïo. La structure du modèle a ainsi évolué en fonction des extensions et des simplifications qu'ils nous ont suggérées.

La formalisation du modèle s'est poursuivie en basculant sur une représentation de type grammaire attribuée. Dans cette représentation, le phénotype d'un cas de test est vu comme une chaîne de caractère agrégeant les valeurs de paramètre retenues pour chaque élément. Un mot de la grammaire est ainsi un descripteur d'un monde virtuel. La grammaire attribuée spécifie les descripteurs valides, en prenant en compte à la fois la structure (comment un descripteur complet agrège les descripteurs de ses éléments) et des propriétés sémantiques (les domaines de validité des paramètres et les contraintes de dépendance entre eux).

Appliquée au cas d'étude *Oz*, cette approche de modélisation commence par l'identification des relations structurelles, comme par exemple :

- `Field` (champ) est composé de :
  - `Crop_row` (rangée de légumes) est composé de :
    - `Crop` (légume) qui peut être de type `Cabbage` (chou) ou `Leek` (poireau)
    - Red `Stake` (piquet rouge) placé au début et à la fin des rangées.
  - `Weed_area` (zone de travail) qui contient les mauvaises herbes.

La structure complète est représentée par le diagramme de classes UML de la Figure 2.2. Le modèle agrège les éléments représentatifs d'un cas de test (classes) liés au champ, au terrain 3D et à la mission de désherbage. Cette décomposition a été affinée jusqu'à ce que le niveau de détail soit jugé suffisant. Chaque élément possède des attributs représentant des paramètres de génération. La première version du modèle comportait plus de 30 paramètres, mais plusieurs simplifications ont été suggérées par les experts de Naïo. Par exemple, le modèle initial comportait des paramètres permettant de tenir compte des piquets rouges mal placés ou manquants. En plus d'être hors du cas nominal d'utilisation de *Oz*, cette possibilité a été jugée trop stressante et a donc été supprimée. Le modèle final comporte 15 paramètres, comprenant 14 attributs de classe et le choix du type de légume.

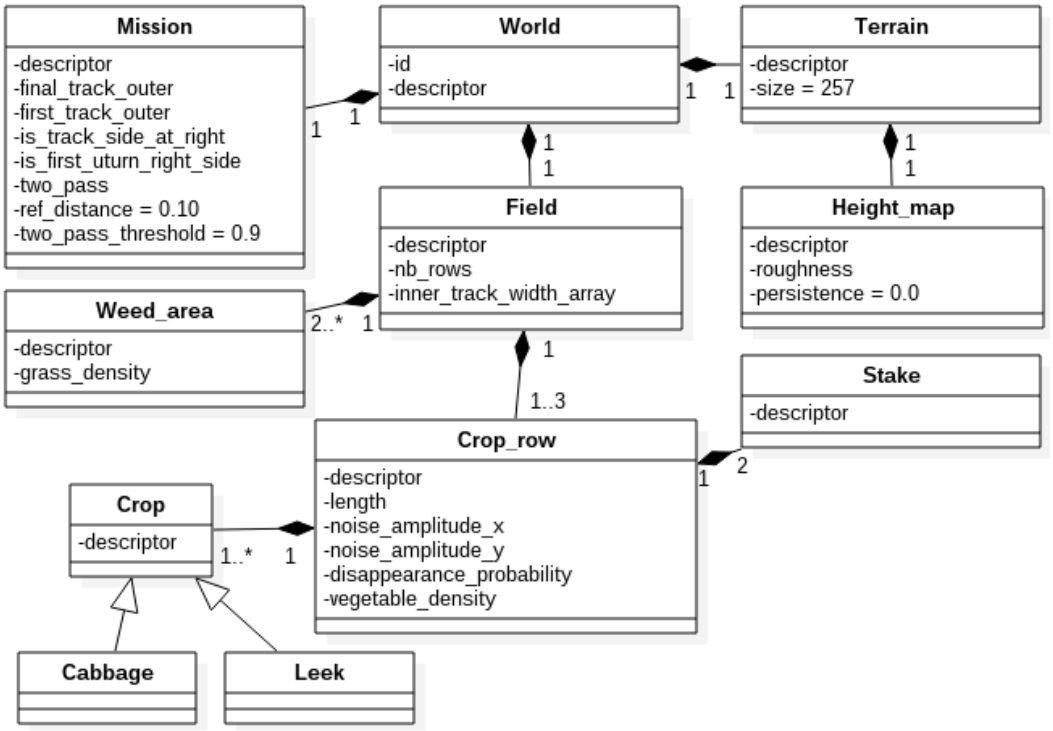


FIGURE 2.2 – Diagramme de classes représentant un cas de test pour *Oz*.

La figure 2.3 illustre certains des paramètres retenus. L'espace entre deux rangées (`inner_track_width`) et la densité de légumes dans chaque rangée (`vegetable_density`) se rapportent au champ, tandis que les autres paramètres caractérisent la mission. Par exemple, le paramètre `first_track_outer` indique si le robot commence sur une zone de travail en bord exterieur du champ. Le paramètre `final_track_outer` indique si le robot est autorisé à désherber l'autre bord extérieur atteint à la fin de la mission (l'agriculteur peut l'interdire lors de la saisie de la mission).

L'ensemble des paramètres définit le génotype. Pour modéliser explicitement ce concept, chaque classe possède un attribut `descriptor` qui contient son génotype sous forme de chaîne de caractères. Il encode les valeurs de tous les paramètres de cette classe et des classes contenues. Ainsi, le descripteur qui se trouve à la racine intègre le génotype complet d'un cas de test.

Les paramètres dans le génotype ne sont pas indépendants les uns des autres. Si le champ comporte N rangées, nous devons considérer $N - 1$ valeurs d'espacement entre celles-ci. S'il n'a qu'une seule rangée, alors `first_track_outer` doit forcément être vrai. Les rangées doivent avoir des longueurs quasi-identiques et contiennent toutes le même type de légume (il s'agit ici de simplifications suggérées par Naïo). Avec une représentation des génotypes sous forme de chaîne de caractères (descripteur), de telles contraintes peuvent être formalisées par une grammaire attribuée.

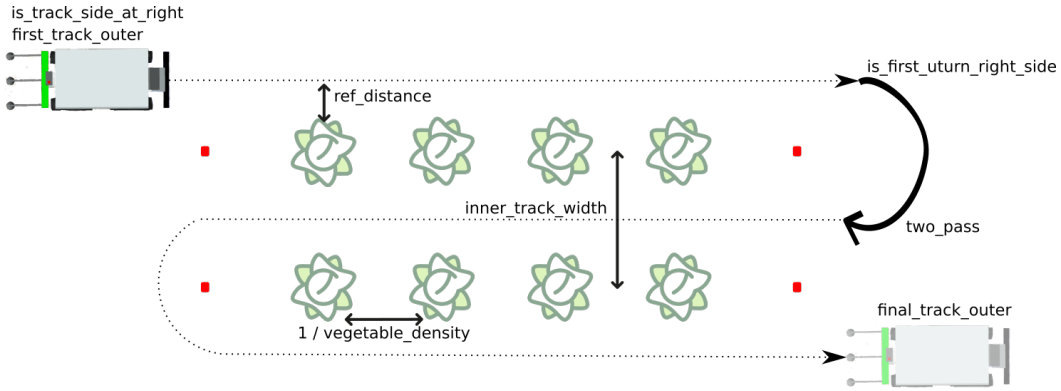


FIGURE 2.3 – Description d'une mission pour *Oz*.

Un génotype est alors valide uniquement s'il est un mot de la grammaire.

La figure 2.4 montre la grammaire d'un génotype pour *Oz*. Les règles syntaxiques (en noir) encodent la structure du diagramme de classes dans les chaînes de caractères. La racine est <world> comme dans le diagramme de classes. De plus, un génotype complet (descripteur de <world>) contient une sous-chaîne pour décrire l'élément <terrain>, une autre pour <field> et une dernière pour <mission>. Les sous-chaînes sont concaténées avec l'ajout d'un caractère séparateur qui permet d'éviter les ambiguïté dans la grammaire.

Les attributs sont associés aux éléments, par exemple *val* est associé à <nb_rows>. Les affectations (en bleu) et les conditions (en rouge) précisent leur valeur et permettent ainsi d'ajouter des informations sémantiques à la syntaxe. A chaque paramètre numérique est associée une contrainte de bornage avec un min et un max. On voit par exemple que le nombre de rangées de légumes (<nb_row>) doit être strictement supérieur à zéro et strictement inférieur à 4 (flèche). Le langage utilisé comprend tous les opérateurs classiques (booléens, relationnels, arithmétiques) plus quelques fonctions utilitaires comme *str2val()* qui permet de convertir une valeur sous forme de chaîne de caractères dans son équivalent numérique. Dans la figure 2.4, la partie grisée se concentre sur la façon dont est spécifiée la contrainte liant le nombre de rangées de légumes (<nb_rows>.val) au nombre d'espacements entre celles-ci (<inner_track_width_seq>.size). Les contraintes sur des attributs multiples sont placées dans le premier ancêtre commun de la hiérarchie, c'est pourquoi cette contrainte apparaît comme une condition attachée à <field>. Le respect de cette condition est déterminé par les affectations effectuées dans les éléments enfants. Il est intéressant de remarquer que la valeur attribuée à l'attribut *size* de <inner_track_width_seq> est définie de manière récursive, en fonction du nombre d'éléments <inner_track_width> produits.

Conceptuellement, la spécification des descripteurs basée sur la grammaire fait du génotype un citoyen de première classe du modèle (*first class citizen*). L'implémentation du générateur de tests suit cette même idée : les opérations de génération créent et manipulent des descripteurs, comme décrit ci-après.

```
<world> ::= <terrain> "-" <field> "-" <mission>
        condition: <field>.nb_rows == 1 → <mission>.first_track_outer == "true"
<terrain> ::= <height_map>
<height_map> ::=  <roughness>
<roughness> ::= <F>
        condition: 0 < str2val(<roughness>) < 1

<field> ::= <nb_rows> "+" <inner_track_width_seq> "+" <crop_row_seq> "+" <weed_area_seq>
            <field>.nb_rows ← <nb_rows>.val
        condition: <nb_rows>.val == <inner_track_width_seq>.size -1 and
                   <nb_rows>.val == <crop_row_seq>.size and
                   <nb_rows>.val == <weed_area_seq>.size +1
<nb_rows> ::= <N>   <nb_rows>.val ← str2val(<N>)
        condition: 1 <= str2val(<nb_rows>) <= 3
<inner_track_width_seq> ::= ε   <inner_track_width_seq>.size ← 0
                | <inner_track_width>   <inner_track_width_seq>.size ← 1
                | <inner_track_width_seq>₂ ":" <inner_track_width>
                    <inner_track_width_seq>.size ← <inner_track_width_seq>₂.size + 1
<inner_track_width> ::= <F>
        condition: 0.65 <= str2val(<inner_track_width>) <= 1.65

<crop_row_seq> ::= <crop_row>       <crop_row_seq>.size ← 1
                                    <crop_row_seq>.length ←<crop_row>.length
                | <crop_row_seq>₂ ":" <crop_row>   <crop_row_seq>.size ← <crop_row_seq>₂.size +1
                        condition: <crop_row_seq>₂.length == <crop_row_seq>.length

<crop_row> ::= <crop> " " <vegetable_density> " " <length> " " <disappearence_probability> " "
                <noise_amplitude_x> " " <noise_amplitude_y>      <crop_row>.length ← str2val(<length>)
        condition: <crop> ==  "l"  → 3 ≤ str2val(<vegetable_density>) ≤ 10
                   <crop> ==  "c"  → 2 ≤ str2val(<vegetable_density>) ≤ 4
<crop> ::= "l" | "c"
<length> ::= <N>
        condition: 5 <= str2val(<length>) <= 15
<vegetable_density> :== <N>
<disappearence_probability> ::= <F>
        condition: 0.0 <= str2val(<disappearence_probability>) <= 0.05
<noise_amplitude_x> ::= <F>
        condition: 0.0 <= str2val(<noise_amplitude_x>) <= 0.1
<noise_amplitude_y> ::= <F>
        condition: 0.0 <= str2val(<noise_amplitude_y>) <= 0.05

<weed_area_seq> ::= <weed_area>   <weed_area_seq>.size ← 1
                        | <weed_area_seq>₂ ":" <weed_area>
                            <weed_area_seq>.size ← <weed_area_seq>₂.size+1
<weed_area> ::= <grass_density>
<grass_density> ::= <N>
        condition: 0 <= str2val(<grass_density>) <= 5

<mission> ::= <two_pass> "+" <final_track_outer> "+" <first_track_outer> "+" <is_track_side_at_right> "+"
            <is_first_uturn_right_side>      <mission>.first_track_outer ← <first_track_outer>
<two_pass> ::= <bool>
<final_track_outer> ::= <bool>
<is_track_side_at_right> ::= <bool>
<is_first_uturn_right_side>  ::= <bool>
<first_track_outer> ::= <bool>

<N> ::= <digit> | <digit><N>
<F> ::= <N> "." <N>
<digit> ::= "0" | "1" | ... | "8" | "9"
<bool> ::= "true" | "false"
```

FIGURE 2.4 – Grammaire attribuée d'un cas de test *Oz*.

2.2.3 Implémentation du générateur

La grammaire attribuée spécifie les génotypes valides, mais n'indique pas comment les produire. En particulier, les règles sémantiques imposent des contraintes sur les valeurs de paramètres dépendants, qu'il va falloir satisfaire. Malheureusement, il n'existe pas de solution satisfaisante pour automatiser la génération de données structurées contraintes. Nous renvoyons le lecteur à la discussion conduite dans le chapitre d'état de l'art (Section 1.6). En particulier, les générateurs basés sur des grammaires soit procèdent par essais successifs (generate-then-filter) ([Coppit 2005]), ce qui peut être très inefficace, soit requièrent l'ajout de code utilisateur pour effectuer la construction des données ([Maurer 1990], [Kifetew 2017], [Pan 2013]]. Dans le cas de l'étude de cas *Oz*, nous avons choisi d'implémenter directement un générateur spécifique.

Comme présenté dans la figure 2.1, la génération du test comporte deux étapes, chacune impliquant des fonctions aléatoires : (1) la production d'un génotype valide, (2) la production d'un phénotype à partir de ce génotype.

Nous avons implémenté ces deux étapes dans un générateur développé en Python en suivant le paradigme orienté objet. La structure du programme est calquée sur celle du modèle UML (figure 2.2) avec en plus l'ajout de quelques méthodes à chaque classe. Les interfaces de chaque classe sont uniformisées. Le code est distribué dans la structure de manière à faciliter les évolutions communes du modèle et de son générateur. Des versions préliminaires basées sur un modèle réduit ont été réalisées pour valider le processus de génération. Le générateur a ensuite été étendu jusqu'à la version finale qui correspond au modèle. Il a été possible d'ajouter ou de supprimer des éléments et des paramètres, avec un impact limité sur le code global.

Un élément du monde possède l'interface suivante pour gérer son génotype :

— `random_create()` : génère aléatoirement un descripteur, c'est-à-dire choisit aléatoirement une configuration valide pour les paramètres de la classe et de ses classes enfants. Si cette méthode est appelée dans la classe world (la racine de la structure), elle appelle récursivement la même méthode de chaque classe enfant pour construire aléatoirement le génotype complet. La structure est instanciée de telle sorte que chaque paramètre prenne la valeur encodée par le descripteur.

— `create_from_descriptor(desc)` : instancie la structure à partir d'un descripteur donné et offre des fonctionnalités de contrôle sur les valeurs des paramètres. Le descripteur donné en entrée peut porter deux types d'éléments : des éléments terminaux (pour forcer des valeurs) ou des opérateurs (pour sélectionner une méthode de génération). Les opérateurs peuvent être vus comme des métacaractères (*wildcard*) qui remplacent une sous-chaîne valide dans un descripteur. L'opérateur *r* est simplement une génération aléatoire par `random_create()`. Par exemple, le descripteur `desc`="0.0-r-r" au niveau de la racine (<terrain>-<champ>-<mission>) signifie que nous voulons créer

tout génotype valide avec un terrain plat (le paramètre `roughness` du terrain est forcée à 0,0 et les paramètres de `Mission` et de `Field` sont générés aléatoirement). D'autres opérateurs m et M ont également été implémentés. Ils sélectionnent respectivement la valeur minimale et maximale d'un paramètre numérique. Ils permettent par exemple de générer un champ avec le nombre maximal de rangées de légumes. La combinaison des terminaux et des opérateurs offre une grande flexibilité pour explorer les régions de l'espace des paramètres.

— `check_descriptor(desc)` : parse un descripteur donné et vérifie s'il est valide. Cette méthode analyse à la fois les contraintes syntaxiques et sémantiques spécifiées dans la grammaire. Par exemple, pour la classe `field`, on vérifie que le nombre de sous-chaînes <inner_track_width> est cohérent avec le nombre de rangées.

Seule la fonction `random_create()` a été utilisé pour générer les cas de test qui ont servi aux expériences présentées de cette partie. Mais il nous a semblé intéressant de présenter aussi la fonction `create_from_descriptor()`, que nous avions conçue pour aller au-delà d'une recherche aléatoire. Nos travaux ultérieurs sur la génération de tests s'en sont inspiré. Dans la solution qui sera présentée dans la deuxième partie du manuscrit, nous avons gardé l'idée de complétion de tests pour lesquels certains paramètres sont déjà définis. Ceci est très utile pour une exploration contrôlée de l'espace des paramètres.

Une fois qu'un génotype valide a été produit, la méthode pour générer un phénotype (un cas de test concret) est la suivante :

— `export()` : la méthode attachée à la classe racine (`World`) génère plusieurs fichiers dans un format adapté au simulateur : un fichier *.json* pour la mission et des fichiers *.sdf* et *.jpeg* pour l'environnement 3D dans *Gazebo* (voir Figure 1.8). La même méthode attachée aux classes filles génère le contenu concret à mettre dans les fichiers. Par exemple, dans le contexte de la classe `Height_map`, on utilise le bruit de Perlin pour générer une carte d'altitude. Dans le contexte de la classe `Crop_row`, la méthode `export()` génère les coordonnées concrètes (x, y) des légumes à partir de leur densité (`vegetable_density`), de leur probabilité de ne pas pousser (`disappearance_probability`) et du bruit sur leur position (`noise_amplitude_X` et `noise_amplitude_Y`). Les modèles 3D des légumes et des piquets rouges sont prédéfinis dans une bibliothèque d'objets (*meshes*), le simulateur se charge de les récupérer puis de les placer à partir d'une liste de coordonnées. Pour finir, dans le contexte de la classe `Mission`, les paramètres sont utilisés pour déterminer la position initiale du robot dans le champ.

Il est à noter que le phénotype est complètement dépendant du simulateur. L'exportation des données est donc très spécifique à chaque étude de cas.

Le framework de génération de tests que nous présenterons dans la deuxième partie de ce manuscrit reprend le principe des deux étapes de génération. Il offre

une solution générique à la génération de données structurées à partir d'un modèle (pour le génotype), basée sur la résolution de contraintes. Il permet à l'utilisateur de rajouter du code spécifique lors de l'exportation de données (pour le phénotype).

2.3 Oracle automatique

Pour les tests sur le terrain et les quelques tests en simulation effectués par Naïo, les verdicts sont déterminés par des testeurs qui vérifient visuellement le comportement du robot. Cette solution manuelle n'est pas envisageable lorsque l'on souhaite tester le logiciel de navigation sur un grand nombre de cas de test générés aléatoirement : Il faut un oracle automatique. On se heurte cependant à la difficulté de la spécification de cet oracle. Pour *Oz*, il n'existait pas de liste d'exigences formalisées. La notion même de comportement défaillant demandait à être clarifiée. L'approche que nous avons utilisée pour surmonter cette difficulté est d'abord introduite puis l'oracle spécifique pour *Oz* est présenté.

2.3.1 Directives pour spécifier l'oracle

La thèse de Thierry Sotiropoulos ([Sotiropoulos 2018]) préconise la spécification d'un oracle basé sur des propriétés (voir section 1.4). L'oracle est basé sur un ensemble de propriétés que les traces de test doivent satisfaire, toute violation étant considérée comme une défaillance. Lorsqu'il n'y a pas de définition claire permettant de différencier un comportement défaillant d'un comportement normal, comme dans le cas de *Oz*, les propriétés doivent être spécifiées à partir de zéro. Afin de guider ce processus de spécification, il est possible de s'inspirer de l'historique des fautes du robot Mana ([Sotiropoulos 2017]), un robot expérimental tout terrain développé au LAAS. Ces travaux antérieurs ont analysé les effets des bugs de navigation de Mana, ils ont permis d'identifier cinq grandes catégories de propriétés à vérifier. Elles correspondent aux divers aspects du comportement d'un robot autonome. Ces grandes catégories de propriétés sont listées ci-après :

1. **Propriétés liées aux phases de mission.** Une mission se compose généralement d'une série de phases, avec certaines attentes sur le comportement du robot (ce que le robot doit ou ne doit pas faire dans chaque phase).
2. **Seuils liés aux mouvements du robot.** Ici, l'objectif est de détecter des valeurs anormales de variables cinématiques ou cinétiques.
3. **Évènements catastrophiques.** par exemple une collision.
4. **Propriétés liées aux messages d'erreur.** Un robot est capable de surveiller son fonctionnement et de signaler les erreurs. Des exigences peuvent être associées au traitement de ces erreurs.
5. **Propriétés liées à la perception.** On cible ici les inadéquations entre la perception du robot et les valeurs réelles correspondantes.

Phase de mission	P1	Demi-tour en plus de 5 et moins de 7 manoeuvres
	P2	Le robot maintient une distance de référence (`ref_distance`) avec les rangées de légumes
	P3	La séquence des zones de travail parcourues correspond à la mission
Seuil lié aux mouvements	P4	*Oz* maintient sa vitesse en dessous d'un seuil
Évènements catastrophiques	P5	Pas de collisions avec les légumes et les piquets rouges
	P6	Le robot ne sort pas des limites du champ
Perception	P7	*Oz* se localise avec une précision n'excédant pas un seuil
Messages d'erreur	P8	En cas d'erreur, la distance d'arrêt d'urgence ne doit pas dépasser un seuil

TABLE 2.1 – Propriétés d'oracle pour *Oz*.

Ces cinq grandes catégories de propriétés ont servi de guide pour structurer les discussions avec Naïo Technologies, et ainsi arriver à la spécification d'un oracle pour *Oz*.

2.3.2 Oracle spécifique pour *Oz*

Le tableau 2.1 présente la liste des propriétés identifiées pour *Oz*. Une mission est vue comme une séquence de phases de désherbage et de demi-tour. Une mauvaise exécution de la mission (par exemple, si le robot emprunte une mauvaise zone de travail) donne un verdict de rejet (**P3**). Une mission incomplète n'est pas nécessairement synonyme de défaillance. En effet, on accepte que le robot puisse décider d'abandonner la mission à tout moment, s'il considère qu'elle ne peut plus être réalisée. Il signale alors une erreur et doit effectuer un arrêt d'urgence en toute sécurité (**P8**). Les défaillances les plus critiques concernent les dommages causés à l'environnement (**P5**) et les sorties des limites du champ (**P6**).

Deux des propriétés (**P2**, **P7**) ont particulierement donné lieu à discussion. Il n'était pas clair si leur satisfaction devait être exigée, ou simplement souhaitée dans un objectif de performance. Par exemple, le maintien d'une distance de référence par rapport aux légumes (**P2**) correspond à une trajectoire idéale de désherbage, ce que Naïo ne considère pas comme une exigence forte, tant qu'il n'y a pas de collision. De même, la précision de l'auto-localisation (**P7**) n'est pas une exigence forte, puisque le robot est guidé par sa perception des rangées de légumes et des piquets rouges. Une mauvaise perception des rangées de légumes et des piquets serait plus pertinente à prendre en compte. Cependant, il a été jugé trop lourd d'ajouter l'instrumentation et les moyens d'analyse manquants dans la plate-forme de test.

Après discussion, il a été décidé d'implémenter les huit détecteurs correspondant

logs du robot	logs du simulateur
Position perçue x, y	Position x, y, z
Lacet (*yaw*) perçu	Quaternion X, Y, Z, W
Rapport de succès de la mission	
Rapport d'erreur	
Commandes moteur	

TABLE 2.2 – Données horodatées collectées pendant les exécutions

aux propriétés de la table 2.1, mais d'exclure ceux des propriétés **P2** et **P7** de l'élaboration du verdict du test. Ces deux détecteurs pourraient être utiles pour de futures expériences (de test de non régression par exemple).

Le tableau 2.2 affiche les données horodatées collectées par la plate-forme de test sous forme de trace d'exécution. Ces données sont ensuite mises à disposition pour nos détecteurs : le verdict est élaboré hors ligne, à partir de la trace d'exécution. Une partie des données est enregistrée à l'interface avec le robot : elle comprend les sorties (commandes moteurs, rapports d'erreur et rapport de succès de la mission) ainsi que certaines données internes rendues observables par les fonctions de log intégrés dans *Ozcore* (position et angle de lacet perçus par le robot). Le reste des traces provient du simulateur afin de retracer la position et l'orientation 3D réelles du robot (à l'inverse de celles perçues). A partir des données disponibles, la mise en oeuvre de **P4** et **P7** est assez directe. En revanche, les autres détecteurs ont demandé plus d'efforts car les données brutes doivent être analysées en prenant en compte le contexte (la configuration du champ, la mission correspondante et la phase de mission courante). Par exemple, la position du robot peut être à l'intérieur d'une zone de travail, à l'intérieur d'une zone de demi-tour en fin de rangée, ou encore à l'extérieur du périmètre du champ. Pour faciliter l'analyse des traces d'exécutions, les zones d'intérêts sont automatiquement pré-calculées pour chaque cas de test.

Le détecteur de **P3** consiste à vérifier que les positions réelles du robot traversent les zones d'intérêt dans le bon ordre. Le détecteur de **P1** se concentre sur les zones de demi-tour, et compte le nombre de changements de direction du robot. Le détecteur de **P2** calcule la distance par rapport à la rangée de légumes suivie. Les collisions (**P5**) sont détectées en fonction de la position des objets dans l'environnement. Pour simplifier les calculs, tous les détecteurs font abstraction de la forme du robot et ne considère que son point central.

Des exécutions préliminaires pour débuguer les détecteurs ont révélé de nombreuses violations transitoires de la propriété **P4**, indiquant des vitesses excessives du robot. Des analyses complémentaires nous ont permis d'en déterminer la cause : il s'agissait d'un artefact de la simulation lié à la physique de faible fidélité. En effet, la simulation ignore la force de frein moteur et surestime la vitesse sur les petites pentes descendantes dues aux irrégularités du terrain. En conséquence, nous avons

décidé de désactiver la vérification de **P4**, pour éliminer ces défaillances parasites qui ne se produiraient pas en conditions réelles. Les résultats présentés dans la section suivante (section 2.4) concernent donc les verdicts basés sur **P1**, **P3**, **P5**, **P6** et **P8**. Une analyse plus en profondeur de ces résultats est donnée dans le chapitre suivant (chapitre 3)

2.4 Résultats

La plate-forme de test décrite dans ce chapitre a été utilisée afin de soumettre le robot *Oz* à une campagne de test en simulation. 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.

Le système sous test a été confronté à 80 cas de test générés aléatoirement à partir du modèle (figures 2.2 et 2.4). Pour prendre en compte l'indéterminisme de la navigation du robot, il est procédé à 5 exécutions par cas de test. L'ensemble donne un total de 400 exécutions, ce qui correspond à une durée de campagne de test d'environ 24 heures.

Les résultats sont assez surprenants mais témoignent de la version prototype du système sous test. En effet, près d'une exécution sur deux s'est vue attribuer un verdict de rejet : 192 exécutions présentent au moins un comportement défaillant alors que 208 n'en présentent aucun.

P1	P3	P5	P6	P8
0	70	142	57	14

Table 2.3 – Décompte des défaillances par type

Le tableau 2.3 donne en détail le nombre d'occurrence des différentes violations de propriétés. Le nombre total de défaillances qui y figurent dépasse 192 car certaines exécutions violent plusieurs propriétés. Dans le cas où la même propriété est violée à plusieurs reprise, elle n'est comptée qu'une fois. C'est notamment le cas pour les collisions qui sont souvent multiples (le robot écrase plusieurs légumes). On observe que 35.5% des exécutions 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 14% des cas, le robot sort des limites du champ (**P6**). A de plus rares occasions (environ 3.4% des cas), le robot ne s'arrête pas directement après avoir levé une erreur. Aucune violation de la propriété **P1** (demi-tour en 5 à 7 manoeuvres) n'a été observée.

Cette campagne de test stresse efficacement le système sous test en provoquant un nombre important de défaillances. Sachant que quelques tests préalables en simulation ont été effectués chez Naïo, ce nombre de défaillances peut paraître surprenant. En y regardant de plus près, les tests effectués chez Naïo sont uniquement pour détecter les défaillances les plus évidentes avant d'aller sur le terrain (test de

fumée). Un cas de test nominal est sélectionné parmi un petit corpus (le plus souvent le cas décrit sur la figure 1.7) puis exécuté sur le simulateur. Cette méthode ne permet pas au testeur d'observer facilement toutes les défaillances car le cas nominal ne semble pas assez stressant. De plus, la validation manuelle ne permet pas de vérifier certaines propriétés comme la sur-vitesse ou la sortie des limites du champ. Les travaux d'analyse présentés dans [Sotiropoulos 2018] poursuivent ensuite par quelques considérations sur l'indéterminisme puis par une aide au diagnostic. Nous reprenons en profondeur l'analyse des résultats au chapitre suivant (chapitre 3).

2.5 Spécification d'oracle de test d'un robot sous-marin

Dans cette section, nous faisons une courte parenthèse sur des travaux auxquels nous avons collaborés [Hereau 2020]. Ces travaux ré-utilisent les mêmes principes que nos travaux sur *Oz* pour spécifier un oracle pour le test d'un robot autonome sous-marin, mais cette fois sur le terrain (au large de Mayotte). En les présentant, notre but est de montrer une application des grandes classes de propriétés d'oracle identifiées précédemment, dans un cadre plus large.

2.5.1 Présentation du robot

Une équipe de roboticiens du LIRMM (LIRMM, CNRS, Montpellier, France) travaille sur le développement d'un robot sous-marin pour analyser la biodiversité. Par exemple, pour analyser l'abondance des différentes espèces de poisson dans une zone donnée. Une telle mesure nécessite des observations très fréquentes ce qui les rend infaisables par des plongeurs.

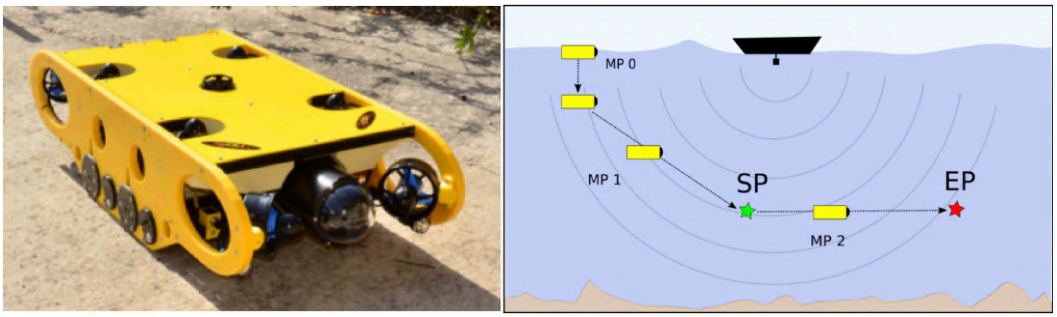


FIGURE 2.5 – Le robot *REMI* et une vue schématique d'une mission type (source : [Hereau 2020]).

Le système est composé de deux éléments principaux : un appareil en surface embarqué à bord d'un bateau et le robot sous-marin appelé *REMI*. La communication entre les deux éléments se fait par un câble ethernet. La figure 2.5 montre le robot (à gauche) et une mission type (à droite). La mission consiste à parcourir un transect, c'est à dire une ligne virtuelle que l'on met en place pour étudier un phénomène dont on comptera les occurrences. Lors de son parcours le robot effectue

des enregistrements vidéo qui peuvent ensuite être utilisés par des biologistes. La mission se décompose en 3 phases (notées MPi sur la figure 2.5). Lors de la première (MP 0), le robot plonge verticalement pour capter un signal USBL émis depuis le bateau. Ce signal est nécessaire pour la localisation. La deuxième phase (MP 1) consiste à rejoindre le début du transect (Start Point). Enfin la dernière phase (MP 2) consiste à parcourir le transect jusqu'au bout (End Point).

2.5.2 Spécification de l'oracle

Comme pour *Oz*, les logs contiennent un maximum de données pour ensuite servir à l'évaluation des propriétés de l'oracle (état estimé, valeurs capteurs, commandes envoyées et les différents évènements). Le tableau 2.4 donne un aperçu de la liste des 20 propriétés identifiées par les auteurs de [Hereau 2020] pour le robot *REMI*.

Phase de mission	P1	Le signal de localisation est capté dans la première phase de mission (MP 0)
	P2, P3	Le début du transect (Start Point) est atteint (position et attitude correctes)
	P4 - P10	La fin du transect (End Point) est atteinte (position, attitude et vitesse correctes)
	P11 - P13	Chaque phase de mission ne dépasse pas un temps limite
Évènements catastrophiques	P14	Pas de fuite d'eau
	P15	La température interne ne doit pas dépasser un seuil
	P16	Le robot doit maintenir son altitude au dessus d'un seuil
	P17	La pression ne doit pas dépasser un seuil
	P18	La communication avec le robot doit être maintenue
Matériel	P19	Le niveau de batterie ne doit jamais passer sous un seuil
Perception	P20	La précision de la localisation estimée doit rester au dessus d'un seuil

TABLE 2.4 – Propriétés d'oracle pour le robot sous marin *REMI*.

Les propriétés 1 à 13 sont relatives aux phases de mission. La deuxième phase (MP 1) se termine lorsque le robot est suffisamment proche du point de départ du transect (**P2** et **P3**). De la même manière, la dernière phase (MP 2) se termine lorsque le robot atteint la fin du transect (**P8** et **P9**). Pendant cette dernière phase, les propriétés se rapportent à la distance par rapport à la ligne (**P4**), à l'attitude du robot (**P5**), à la vitesse du robot (**P6** et **P7**) et à une mesure de couverture de la zone parcourue par rapport à celle spécifier dans la mission (**P10**). Les propriétés 11 à 13 concernent le temps maximal d'exécution de chaque phase de mission. Ce

temps dépend de la mission, il est calculé à partir de la distance et d'une vitesse moyenne minimale.

On remarque qu'il n'y a aucune propriété dans la classe "Seuil lié aux mouvements". Sur ce cas d'étude, les propriétés pouvant entrer dans cette catégorie sont également dépendantes de la phase de mission. Les auteurs ont alors préféré les classer dans la catégorie "Phase de mission".

Une propriété concernant le niveau de batterie ne rentre pas dans les classes que nous avions identifiées initialement. Une catégorie "matériel" a donc été ajoutée. Cela n'invalide en rien la pertinence de nos classes de propriétés. En effet, la classification ciblait uniquement le système logiciel de navigation. Ici les tests sont effectués sur le terrain, le cadre est donc plus large et fait intervenir des éléments matériels (*hardware*). La classification peut simplement être étendue au gré des besoins.

2.6 Conclusion

Ce chapitre présente la mise en place de la plate-forme de test de *Oz* et la spécification de ses différents éléments : (i) d'abord le générateur de cas de test avec sa spécification en UML puis avec une grammaire attribuée, (ii) ensuite l'oracle, sous forme d'un ensemble de propriétés déterminé en suivant les acquis de travaux antérieurs (cinq grandes catégories de propriétés à considérer). Un court détour sur un autre cas d'étude montre la conception d'un autre oracle basé sur des propriétés.

Le développement de la plate-forme de test a été fait lors d'une thèse précédant la nôtre ([Sotiropoulos 2018]). Nous étions intervenu en support à ces travaux, dans le cadre d'un stage. Ce chapitre contient tout de même des éléments nouveaux, en détaillant davantage la formalisation du modèle à l'aide d'une grammaire attribuée, et en montrant un autre exemple d'oracle de test pour le robot sous-marin *REMI*. Les résultats de test présentés dans ce chapitre sont issus d'une première analyse faite avec peu d'information sur le système sous test. Nous ne savons pas si les défaillances observées révèlent des fautes réelles, ou sont des défaillances parasites induites par la simulation (comme dans le cas des survitesses parasites qui ont conduit à désactiver la propriété **P4**). Nous ne savons pas non plus quelles fautes, révélées par des tests en conditions réelles, seraient manquées par les tests virtuels. Pour aller plus loin, l'analyse doit être approfondie en prenant en compte les résultats des tests sur le terrain menés par Naïo et leur expertise du système sous test. Cette analyse approfondie, ainsi qu'un retour d'expérience global, font l'objet du chapitre suivant (chapitre 3). Ils constituent une contribution de notre thèse. Nos autres contributions ont également bénéficié de ces premiers travaux sur le robot *Oz*. La problématique de génération automatique de données de test, qui sera abordée en deuxième partie de ce mémoire, découle directement de notre expérience dans la conception et l'implémentation du générateur pour *Oz*. La modélisation des environnements virtuels du robot revient à spécifier des structures de données riches, comportant un nombre variable de sous-éléments, et devant satisfaire des propriétés

sémantiques. Dans le générateur que nous avons développé pour *Oz*, les interdépendances entre les données sont gérées en dur dans le code et pourraient être remise en cause par une modification du modèle. Une telle solution spécifique est clairement insatisfaisante. Cela a motivé notre recherche d'une solution générique, capable de produire automatiquement les données à partir d'un modèle purement déclaratif de leurs propriétés structurelles et sémantiques. La solution à laquelle nous avons abouti offre cette généricité, tout en gardant des caractéristiques intéressantes du générateur spécifique initial :

— la capacité à produire des données multiples et diverses, via l'incorporation de procédés aléatoires,

— une fonctionnalité de complétion de tests, permettant de guider l'exploration du domaine d'entrée,

— la séparation entre cas de tests abstraits et concrets, avec des fonctions d'exportation spécifiques qui peuvent être ajoutées par l'utilisateur.

Avant d'arriver à la partie qui présentera cette solution, le chapitre suivant poursuit l'étude de cas *Oz* en donnant une analyse comparative détaillée des tests en simulation et sur le terrain.

Chapitre 3

Comparaison des tests en simulation et sur le terrain

Sommaire

3.1 Introduction

La simulation est une approche prometteuse pour réduire les tests coûteux sur le terrain. Elle donne aussi plus de souplesse pour explorer des situations de test diverses. Néanmoins, elle ne reproduit qu'imparfaitement ces situations, notamment en ce qui concerne les interactions avec le monde physique. On peut donc se demander si les simplifications induites par la simulation entravent la capacité à révéler des fautes.

De fait, la validation de robots autonomes utilise peu la simulation. Les développeurs de robots considèrent très important que les tests s'effectuent en conditions réelles, ou tout au moins le plus proche du réel possible. A cet égard, l'exemple de Naïo Technologies est typique. De façon antérieure à notre étude, cette société avait commencé à introduire des tests virtuels de type SiL (*software-in-the-loop*) pour soumettre des missions de désherbage à leur robot. Mais leur expérience initiale

s'était avérée décevante. Les ingénieurs avaient cherché à développer une simulation haute fidélité, mais avait été limités par des problèmes sévères de performance, les obligeant à produire la version basse fidélité utilisée par nos expériences. Cette version étant jugée trop éloignée de la réalité, Naïo utilisait en fait très peu la simulation pour la validation de niveau mission. Les ingénieurs se contentaient de tester quelques cas virtuels – le plus souvent un seul (voir la figure 1.7) – comme "test de fumée" (*smoke test*) avant de démarrer les expérimentations sur le terrain.

A rebours de cette vue restrictive de la simulation, des travaux menés au LAAS ([Sotiropoulos 2017]) suggéraient qu'il n'était pas nécessaire d'avoir une haute fidélité pour trouver une grande partie des fautes actuellement révélées sur le terrain. Les auteurs avaient analysé un historique de 33 fautes du logiciel de navigation du robot Mana du LAAS. Ils avaient étudié les conditions de reproductibilité des fautes, c'est-à-dire les conditions pour activer les fautes et observer leurs effets. Parmi les fautes analysées, une seule nécessitait une simulation haute fidélité du monde physique.

Dans ce chapitre, nous présentons une nouvelle étude de l'efficacité du test en simulation. Nous approfondissons l'analyse des tests de *Oz* présentés dans le chapitre précédent, et comparons leurs résultats avec ceux des tests menés sur le terrain. Nous cherchons à déterminer la capacité des tests virtuels à révéler une partie des fautes trouvées en conditions réelles, voire à en révéler de nouvelles. Pour être complet, il nous faut aussi considérer les cas de défaillances parasites induits par la simulation.

La question de recherche est donc la suivante. Quels sont les avantages et les inconvénients de test en simulation en ce qui concerne :

— les fautes trouvées lors des test sur le terrain,

— les fautes non trouvées lors des test sur le terrain,

— les problèmes parasites causés par des artefacts de simulation ?

L'exemple de *Oz* permet d'étudier cette question sur un système en cours de développement dans l'industrie. Au début de l'étude, les expérimentations sur le terrain n'avaient pas commencé et les fautes affectant le logiciel étaient inconnues. Les interactions avec Naïo pour modéliser les cas de test et spécifier l'oracle n'ont donc pu être biaisées par la connaissance des fautes à trouver. Les tests en simulation et sur le terrain ont ensuite été menés en parallèle. Nous n'avons eu aucune information sur les résultats sur le terrain avant la comparaison finale. Cette comparaison a eu lieu plusieurs mois après la fin des tests, pour laisser le temps à Naïo de diagnostiquer les problèmes observés sur le terrain.

Le diagnostic des fautes trouvées en simulation n'a pu être finalisé que lors de cette phase de comparaison. Rappelons que les tests virtuels s'effectuaient en boîte noire, sans accès au code source. L'expertise de Naïo était nécessaire pour confirmer les causes des cas de défaillance. Un grand nombre d'exécutions de tests (192) présentant au moins une défaillance, il n'était pas possible d'examiner chacune d'entre elles avec les ingénieurs. Nous avons donc mené une analyse préalable de ces exécutions pour préparer la comparaison finale. Les cas de défaillance ont été catégorisés

sous forme de problèmes de plus haut niveau, susceptibles d'être causés par la même faute. Pour chaque problème, nous avons identifié quelques tests typiques à analyser avec les ingénieurs.

Le chapitre est structuré comme suit. Avant d'aborder l'analyse approfondie des résultats de test, nous revisitons un point délicat de la mise en oeuvre des tests virtuels : la prise en compte du non déterminisme (section 3.2). En effet, pour un même cas de test, le comportement observé peut être différent d'une exécution à l'autre. Comme indiqué dans le chapitre précédent, ceci a été pris en compte en répétant cinq fois l'exécution de chaque test. Nous évaluons l'impact du non-déterminisme sur les détections de défaillance, et étudions la pertinence de cette stratégie de répétition. La section suivante (section 3.3) effectue une analyse fine des 192 exécutions de test défaillantes, pour préparer la phase de comparaison finale avec Naïo. Les résultats obtenus sur le terrains sont ensuite présentés, ainsi que la liste des fautes connues à l'issue des deux types de test (section 3.4). Pour finir, tous les éléments de réponse sont réunis pour répondre à notre question de recherche, dans la section 3.5. Un retour sur les enseignements tirés de la mise en place de la plate-forme de test et une discussion sur la validité de nos résultats y sont également détaillés.

3.2 Impact de l'indéterminisme sur les résultats

Dans cette section nous reprenons les résultats du chapitre précédent (2). Nous évaluons l'impact de l'indéterminisme sur les occurrences de défaillance. D'abord, l'indéterminisme de nos résultats est quantifié en observant la variabilité de nos résultats. Ensuite, une évaluation de l'efficacité du test est donnée selon les critères d'exploration (un nouveau cas de test est soumis) et d'exploitation (l'exécution d'un cas de test existant est répétée) des cas de test. Cette évaluation a fait l'objet d'une publication [ROBERT 2020c].

3.2.1 Variabilité des verdicts de test

L'ensemble de tests générés se compose des 80 cas de test (environnements virtuels différents avec leur mission correspondante). Pour chaque cas de test, nous avons effectué 5 exécutions pour prendre en compte l'effet de l'indéterminisme. Le choix du nombre d'exécutions est arbitraire. A priori, 5 semble un bon compromis pour parcourir les différents cas de test tout en limitant le temps d'exécution global (environ 24h pour les 400).

L'analyse du chapitre 2 montre que les environnements que nous avons générés sont très stressants pour le robot car un verdict de rejet survient dans pas moins de 48% des exécutions (192/400). Toutes les propriétés, à l'exception de P1 (demi-tour en 5-7 manoeuvres), sont violées. Le tableau 3.1 rappelle les propriétés vérifiées à chaque exécution et le tableau 3.2 donne le nombre d'exécutions violant chaque propriété. Puisqu'une même exécution peut violer plusieurs propriétés, la somme est supérieure aux 192 exécutions présentant un verdict de rejet. La figure 3.1 illustre

Détecteurs inclus dans l'oracle	**P1** :	Demi-tour en plus de 5 et moins de 7 manoeuvres
	P3 :	La séquence des zones de travail parcourues correspond à la mission
	P5 :	Pas de collisions avec les légumes et les piquets rouges
	P6 :	Le robot ne sort pas de la limite du champ
	P8 :	En cas d'erreur, la distance d'arrêt d'urgence ne doit pas dépasser un seuil
Détecteurs désactivés	**P2** :	Le robot maintient la distance référence (`ref_distance`) avec les rangées de légumes
	P4 :	*Oz* maintient sa vitesse en dessous d'un seuil
	P7 :	*Oz* se localise avec précision n'excédant pas un seuil

Table 3.1 – Détecteurs associés aux propriétés (rappel du chapitre 2)

P1	P3	P5	P6	P8
0	70	142	57	14

Table 3.2 – Décompte des défaillances par type (rappel du chapitre 2)

une exécution avec plusieurs violations. Cette vue est générée à partir d'un outil de visualisation des sorties de test que nous avons développé au LAAS. On peut y voir le robot (représenté par un rectangle noir) en cours de mission sur un côté extérieur du champ, le long d'une rangée de poireaux (représentée par des cercles verts). Sa trajectoire est représentée par une ligne noire. Plus précisément, elle correspond à la position du centre du robot tout au long de sa mission. En faisant demi-tour à la fin de la rangée, le robot franchit les limites spécifiées du champ (indiquées par un trait en pointillé) et entre ensuite en collision avec le piquet rouge et trois poireaux. Cette exécution compte à la fois pour **P6** (sortie de champ) et **P5** (collision). Globalement, la collision est le type de défaillance le plus fréquent, on l'observe dans 35,5% des exécutions de test.

Le tableau 3.3 donne la proportion de cas de test donnant n verdicts de rejet parmi les cinq exécutions répétées (un verdict est catégorisé comme "rejet" lorsqu'au moins une propriété **Px** est violée). Un cas de test a un résultat non déterministe quand au moins une (mais pas toutes) de ses exécutions répétées échoue, c'est-à-dire lorsque $n \in 1, 2, 3, 4$. Un peu moins de 29% des cas de test (23 sur 80) donnent un tel verdict variable d'une exécution à l'autre. De plus, les verdicts de rejet cohérents ($n = 5$) ne signifient pas que le mauvais comportement détecté est le même pour toutes les exécutions.

La figure 3.2 illustre différents mauvais comportements sur le même cas de test. La partie supérieure de la figure montre une vue globale de la même exécution présentée dans la figure 3.1. Comme nous l'avons vu précédemment, le premier

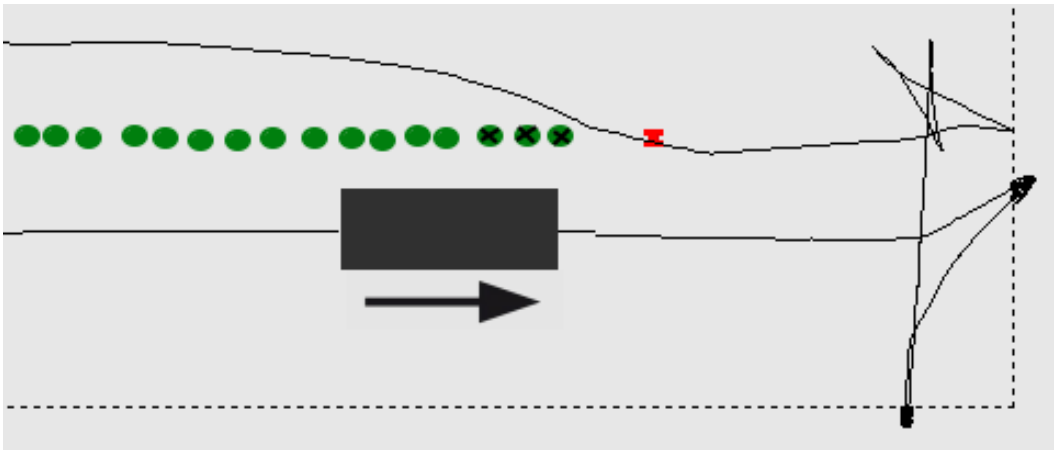


FIGURE 3.1 – Exemple présentant plusieurs défaillances.

FIGURE 3.2 – Deux résultats différents lors de deux exécutions d'un même cas de test.

# Verdict de rejet (sur 5 exécutions)	# Cas de test (total : 80)
0	32
1	5
2	4
3	2
4	12
5	25

TABLE 3.3 – Proportion des cas de test avec n verdicts de rejet sur les 5 exécutions répétées.

demi-tour donne lieu à plusieurs défaillances : l'une des manoeuvres dépasse les limites du champ et il y a des collisions lors de l'entrée dans l'inter-rang suivant. Le reste de la mission se déroule correctement. La partie inférieure de la figure 3.2 visualise une autre exécution d'exactement le même cas de test. Cette fois, une nouvelle défaillance s'ajoute aux autres : après le demi-tour, le robot rate l'approche vers la prochaine zone de travail et retourne dans une zone déjà désherbée (violation de **P3**). A la fin de son trajet retour, le robot détecte que quelque chose ne va pas et s'arrête avec un rapport d'erreur. Ces deux exécutions ont toutes deux un verdict de rejet mais présentent des schémas de violation de propriété différents. Ce cas de test n'est qu'un exemple parmi d'autres.

En considérant le grand nombre d'exécutions menant à un verdict diffèrent, ainsi que les exécutions concluant au même verdict de rejet mais avec des violations différentes, on en conclut que le non-déterminisme a un impact important sur la détection de défaillances pour un cas de test donné.

La prochaine partie évalue la pertinence de la stratégie de répétition des cas de test. Pour le test de notre système est-il préférable de diminuer le nombre de cas de test et d'augmenter le nombre de répétition pour avoir des résultats plus fiables ? Ou au contraire, est il préférable de réduire le nombre de répétitions et d'augmenter le nombre de cas de test pour favoriser l'exploration ?

3.2.2 Répétition vs exploration

La répétition des exécutions est un moyen classique de tenir compte de l'effet de l'indéterminisme. Comme chaque exécution prend quelques minutes, ajouter des exécutions répétées pour chaque cas de test est très coûteux sur le budget de temps alloué à la campagne de test. On peut alors se demander si le choix d'un plus petit nombre de répétitions aurait pénalisé l'efficacité de la campagne de test. En partant de l'ensemble complet des exécutions $X_{80,5}$ (80 cas de test x 5 exécutions répétées) qui sont déjà réalisées et analysées au chapitre 2, nous extrayons aléatoirement des sous-ensembles $X_{t,r}$ pour simuler des stratégies avec t cas de test répétés r fois, $t \in 1, 80$ et $r \in 1, 5$.

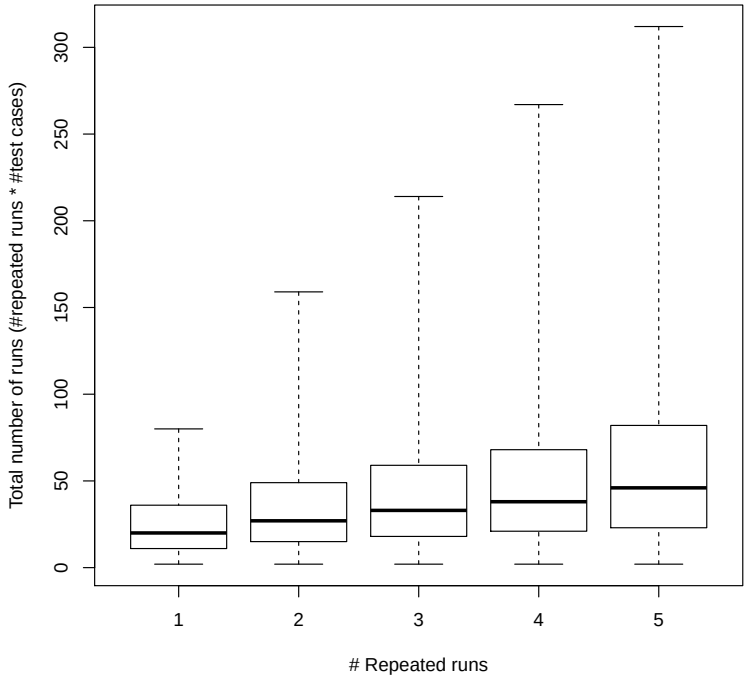


FIGURE 3.3 – Nombre total d'exécution pour observer tous les types de défaillance ($P\_3P\_5P\_6P\_8$) au moins une fois.

D'abord, une première étude concerne le nombre de cas de test nécessaires pour observer au moins une défaillance de chaque type (**P3**, **P5**, **P6**, **P8**). On considère r exécutions par cas de test. On ajoute ensuite des nouveaux cas de test de manière incrémentale en sélectionnant à chaque fois un sous-ensemble de r exécutions parmi les 5 disponibles. L'opération est répétée jusqu'à ce que chaque propriété soit violée au moins une fois. Le cardinal de l'ensemble d'exécution de test ainsi créé est alors $t*r$. Les cas de test et les exécutions sont sélectionnés aléatoirement. Pour permettre une analyse statistique, 10 000 ensembles sont créés (comme décrit ci-dessus) pour chaque r. La figure 3.3 montre le résultat de ce processus dans un diagramme boite à moustache (*box plot*). Avec une seule exécution par cas de test (la première boîte $r = 1$), le cardinal médian de l'ensemble des exécutions de test permettant de violer au moins une fois chaque propriété est de 20 ($t = 20$ et $r = 1$). Ce cardinal est plus de deux fois supérieur (46) si nous choisissons 5 exécutions par cas de test. Pour nos résultats, il semble donc plus efficace de favoriser la diversité des cas de test que la répétition des exécutions : les stratégies avec $r > 1$ ralentissent la détection de nouveaux types de défaillance.

Ce résultat montre également que le nombre de cas test pourrait être réduit. Avec un ensemble plus petit, on peut s'attendre à trouver toutes les défaillances. Cependant, avec un ensemble plus petit de cas, on peut perdre l'information des fréquences relatives des types de défaillance. Or, dans un contexte industriel, il peut être intéressant d'avoir un classement des défaillances par fréquence pour corriger en priorité les plus fréquentes. Un trop petit ensemble de tests ne permet pas forcément cette évaluation.

On s'intéresse donc aussi à l'étude des fréquences relatives des violations de propriétés. La figure 3.4 montre l'histogramme pour l'ensemble $X_{80,5}$. Les données sont divisées en 16 cases, correspondant à chaque combinaison potentielle de violation de propriété (encodées par un compteur binaire) : le *bin* 0 donne le pourcentage d'exécutions ne violant aucune propriété, et le *bin* 15 correspond aux exécutions violant les quatre propriétés. Nous utilisons cet histogramme $X_{80,5}$ comme référence pour étudier l'impact de la variation de t et de r. L'écart des résultats entre l'ensemble $X_{80,5}$ et un autre ensemble $X_{t,r}$ est mesuré en calculant la distance de *Manhattan* des fréquences de chaque combinaison :

$$D = 0.5 * \sum_{i=0}^{15} |f_i - f_{Refi}|$$

Le coefficient 0.5 sert simplement à normaliser la distance sur l'intervalle $[0, 1]$, 15 est l'indice maximal des bin, f_{Refi} est la fréquence relative du *bin* i de $X_{80,5}$ (par exemple, $f_{Ref0} = 52\%$ dans la figure 3.4) et f_i est la fréquence relative du *bin* i de l'ensemble considéré $X_{t,r}$. L'objectif est d'évaluer pour chaque sous-ensemble la distance D par rapport à la référence $X_{80,5}$ et d'identifier quels r et t sont suffisants pour obtenir une distribution proche de la référence. Pour chaque paire (r, t), on sélectionne aléatoirement 10 000 ensembles $X_{t,r}$ et on mesure leur distance à la référence ($X_{80,5}$). Lorsqu'il y a moins de 10 000 possibilités, comme c'est le cas pour les

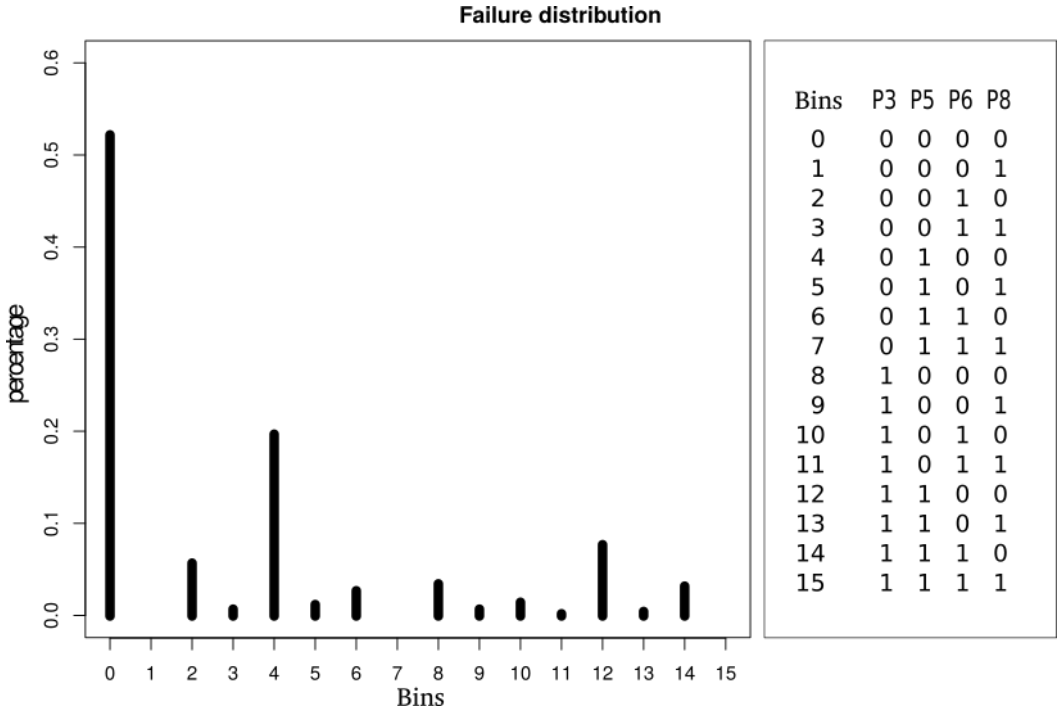


FIGURE 3.4 – Fréquence relative des violations de propriété évalué sur les 400 exécutions ($X_{80,5}$).

petites valeurs de t et r, on considère exhaustivement tous les ensembles possibles. La figure 3.5 montre les distances médianes obtenues. Leur valeur est représentée par un niveau de gris sur une échelle linéaire, où le blanc signifie $D = 0$ et le noir signifie $D = 1$. Comme on peut le voir pour tous les choix de t, les fréquences de violation de défaillance ont tendance à devenir de plus en plus similaires à mesure que le nombre de cas de test augmente (le ton devient plus clair). Un point intéressant est de regarder les iso-distances. Par exemple, les lignes rouges de la figure 3.5 délimitent les configurations (t, r) avec une distance $D = 0,1$ par rapport à la référence. On observe qu'il faut beaucoup plus de temps pour s'approcher de cette distance avec des exécutions répétées : le cardinal de l'ensemble des exécutions de test varie de 56 ($56 * 1$) à 165 ($33 * 5$). Intuitivement, cela suggère que les fréquences finales observées pour $X_{80,5}$ sont plus déterminées par la diversité des cas de test que par leurs exécutions répétées.

Nous arrivons ainsi à une conclusion surprenante. Bien que le non-déterminisme de l'exécution ait un impact élevé sur l'occurrence ou non des défaillances, il semble plus rentable de simplement ignorer le problème et d'utiliser le budget de test pour exécuter plus de cas de test divers sans répétition. Bien sûr, cette conclusion est relative à notre cas d'étude. Mais nous avons donné une méthode d'évaluation de l'indéterminisme qui pourrait par exemple permettre de rendre les prochaines campagnes de test plus efficaces pour ce système. De manière plus générale, un

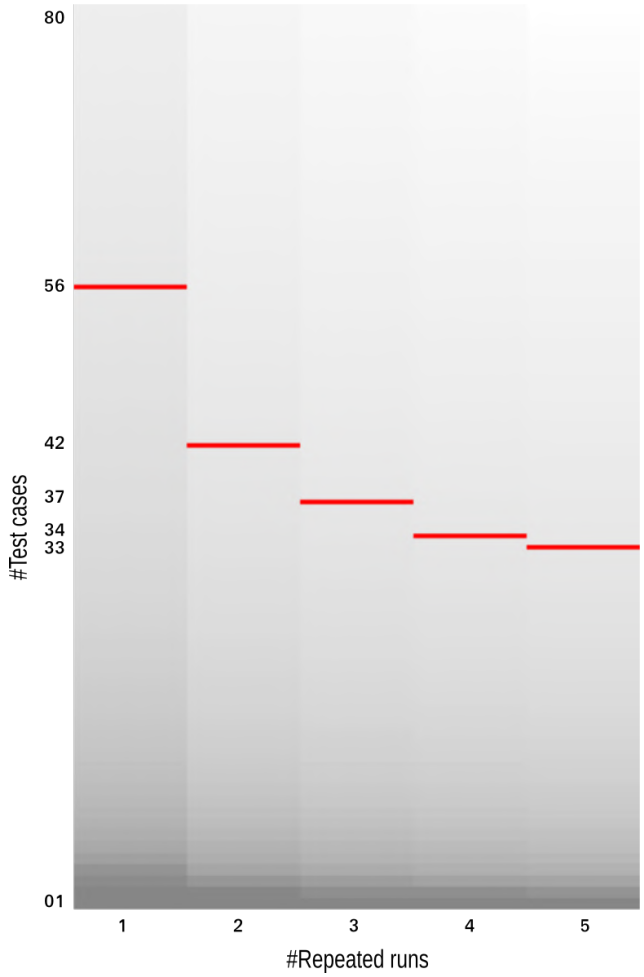


FIGURE 3.5 – Distance médiane avec l'histogramme figure 3.4. En rouge, la ligne iso-distance $D = 0.1$.

résultat intéressant est de démontrer que les exécutions répétées ne sont pas toujours la meilleure approche pour tester des applications non déterministes.

Pour ce qui suit, nous retiendrons que le choix de 80 cas de test et 5 répétitions semble suffisant pour notre analyse des cas de défaillance. Les différents cas de violation et leur fréquence relative auraient été similaires avec des nombres plus petits, montrant une certaine stabilisation des résultats. Nous n'avons donc pas cherché à étendre le jeu de test existant en rajoutant des cas ou des exécutions supplémentaires. L'analyse va porter sur les 400 exécutions de test déjà à notre disposition, et plus particulièrement sur les 192 présentant un verdict de rejet.

3.3 Analyse approfondie des défaillances en simulation

Nous présentons maintenant une analyse qualitative des différents cas de défaillance observés en simulation. Le but est ici de préparer la comparaison avec les résultats de test de notre partenaire industriel, réalisés sur le terrain. Ainsi, nous avons passé en revue les 192 exécutions présentant au moins une défaillance et pour chacune de celles-ci, nous avons extrait des informations pour en caractériser le contexte. Nous obtenons la liste de sous-cas listés dans la figure 3.6. A partir de cette liste détaillée, nous cherchons à effectuer des regroupements et à identifier des causes communes de ces défaillances, en termes de fonctions fautives du robot. Rappelons que notre analyse s'effectue sans accès au code source. Elle se base sur des inspections manuelles de chacune des exécutions de test, éventuellement à l'aide d'un outil de visualisation de données.

3.3.1 Un problème majeur de gestion du demi-tour

À partir des sous-cas de la figure 3.6, il est évident que le demi-tour est le contexte de défaillance le plus répandu. Cela est tout à fait compréhensible car les phases de demi-tours sont bien plus délicates que les phases de suivi de ligne. Ce sont les phases de mission les plus difficiles. Dans 75% des cas présentant des comportements défaillants, au moins une violation de propriété est liée au demi-tour. Cela suggère un problème majeur dans la gestion des manoeuvres correspondantes. Par la même occasion, cela permet d'expliquer le taux élevé de verdict de rejet que nous obtenons.

Pendant le demi-tour, le robot peut heurter un piquet rouge (**P5__2**). De plus à la fin de la phase de demi-tour, les manoeuvres laissent souvent le robot dans une position et une orientation inadéquates pour entrer dans l'inter-rang suivant. En conséquence, le robot entre dans la mauvaise rangée (**P3__1**, **P3__2**, **P3__3**) et/ou entre en collision avec le piquet et les légumes lors de son entrée dans l'inter-rang (**P5__1**). La trajectoire présentée précédemment dans la figure 3.1 illustre ce type de mauvais comportement. Lorsque le demi-tour a lieu à l'extrémité du champ, la mauvaise orientation du robot entraîne parfois une sortie en dehors des limites du champ (**P6__1**). La figure 3.7a montre un exemple de ce type de trajectoire. Dans ce type de défaillance, le scénario est toujours le même, le robot lève d'abord une

P3 : Mauvaise séquence de passage dans les zones de travail (70 exécutions) :
◇ P3_1 : Le robot reprend le même inter-rang après son demi-tour (55/70)
◇ P3_2 : Le robot reprend l'inter-rang précédent après son demi-tour (4/70)
◇ P3_3 : Le robot saute un inter-rang après son demi-tour (9/70)
◇ P3_4 : Le robot quitte l'inter-rang courant pendant son passage (2/70)

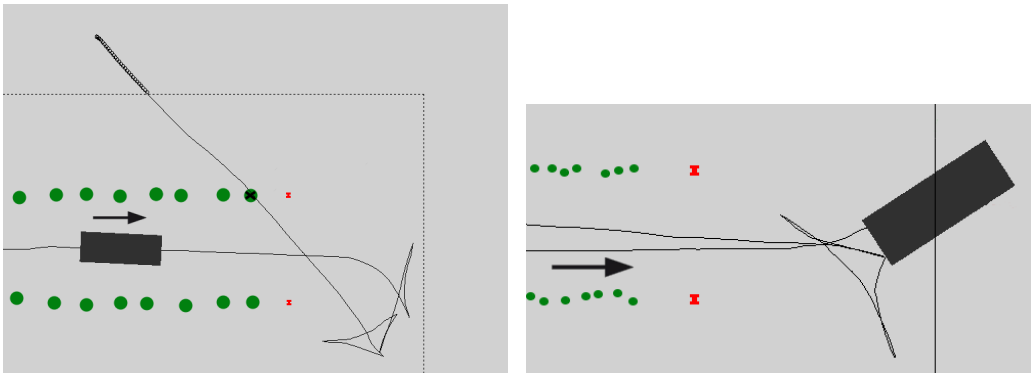
P5 : Collisions avec des piquets rouges et des légumes (142 exécutions) :
◇ P5_1 : Après le demi-tour (91/142)
◇ P5_2 : Pendant le demi-tour (18/142)
◇ P5_3 : A la fin d'un inter-rang (11/142)
◇ P5_4 : Au milieu d'un inter-rang (22/142)

P6 : Dépassement des limites du champ (57 exécutions) :
◇ P6_1 : Après le demi-tour (6/57)
◇ P6_2 : Pendant le demi-tour (49/57)
◇ P6_3 : A la fin d'un inter-rang (2/57)

P8 : L'arrêt du robot n'est pas immédiat après la levée d'une erreur (14 exéc.) :
◇ P8_1 : Après une erreur de type *"Blind course"* (7/14)
◇ P8_2 : Après une erreur de type *"Steersman"* (1/14)
◇ P8_3 : Après une erreur de type *"Invalid markers spacing"* (6/14)

FIGURE 3.6 – Sous-cas des violations de propriété.

erreur puis s'arrête trop tard (après avoir franchi les limites du champ). Comme spécifié dans les conditions d'utilisation de *Oz*, ces limites sont situées à 1,5 mètre après les piquets rouges et autour des rangées de légumes externes de chaque côté du champ.



(a) Sortie de champ après un demi-tour raté. (b) Sortie de zone pendant un demi-tour

FIGURE 3.7 – Défaillances de sortie de champ liées au demi-tour.

Toujours en relation avec le demi-tour, un autre sous-cas de violation de **P6** (**P6_2**) est illustré par la figure 3.7b. Le demi-tour peut être réussi (le robot est

aligné sur le bon inter-rang à la fin du demi-tour) mais prend plus d'espace que prévu. Lors des manoeuvres quelques points dépassent légèrement les limites du champ, ce qui entraîne une position temporairement hors du champ. À chaque fois, la vérification de l'oracle signale un ou deux centimètres de dépassement de la limite. Cette mesure concerne le point central du robot, le dépassement est donc en réalité de l'ordre de quelques dizaines de centimètres.

3.3.2 Un problème de gestion de l'arrêt d'urgence

Les autres violations de propriété ne sont pas directement liées au demi-tour. Un exemple est l'arrêt trop tardif après avoir émis une erreur (**P8_1**, **P8_2**, **P8_3**). Il y a trois types d'erreurs qui ont été levées lors des tests. Elles nécessitent toutes une intervention humaine en situation réelle. Ces erreurs se classent en deux catégories :

1. les erreurs récupérables (*recoverable error*), elles permettent à l'opérateur de relancer la mission là où elle s'est arrêtée ;
2. les erreurs fatales (*fatal error*), elles nécessitent un redémarrage du robot et donc une reprogrammation de la mission.

L'erreur de type *"Blind course"* (trajectoire à l'aveugle) est récupérable, elle est levée lorsque le robot a parcouru environ 3 mètres sans percevoir la rangée de légumes qu'il est censé suivre. L'erreur de type *"Steersman"* est également récupérable, elle est levée lorsque le robot considère que son angle de lacet est suspect pendant une phase de suivi de ligne. La direction de la trajectoire actuelle diffère trop de celle des 10 derniers mètres (La différence d'angle de lacet est supérieure à 15 degrés). Finalement, l'erreur de type *"Invalid markers spacing")* (espacement des marqueurs non valide) est fatale, elle est levée lorsque le robot détecte un piquet rouge manquant ou mal placé. Dans les cas de test générés, les piquets sont toujours au bon endroit. Cette erreur fatale est levée lorsque le robot désherbe une mauvaise zone de travail sur une extrémité du champ et ne voit donc pas le nombre attendu de piquets à la fin de cette zone.

Comme on peut le voir dans les sous-cas **P8_1**, **P8_2**, **P8_3** de la figure 3.6, l'arrêt tardif n'est pas lié à un type d'erreur spécifique : il se produit pour les trois types d'erreur. De plus, cette défaillance n'est pas systématique : dans l'ensemble des exécutions, nous avons observé de nombreux arrêts corrects, et ce après chaque type d'erreur (respectivement 46, 3 et 26 arrêts corrects). Cela suggère un problème de concurrence, qui affecte de manière non déterministe le traitement générique des erreurs.

3.3.3 Des problèmes possiblement liés à la perception de la situation

En ce qui concerne les collisions non liées au demi-tour (**P5_3**, **P5_4**), le robot s'approche trop près de la rangée de légumes qu'il suit pendant sa phase de désherbage. Dans deux des exécutions, le robot traverse même la rangée, ce

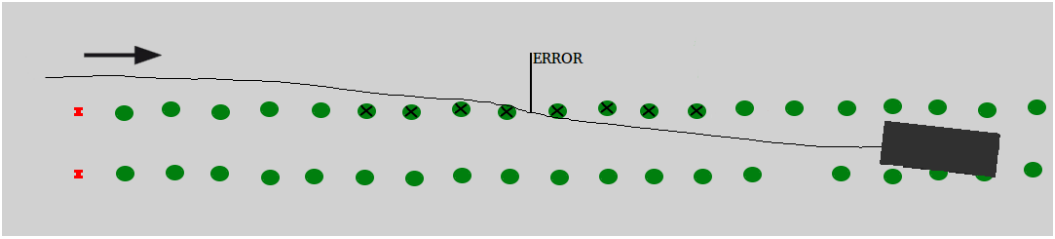
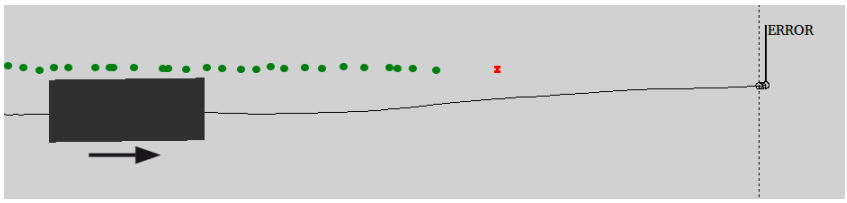
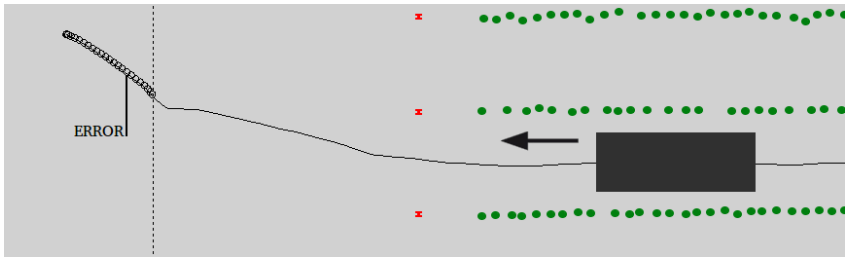


FIGURE 3.8 – Le robot traverse la rangée de légumes pendant sa phase de suivi de ligne. S'ajoute à cela, une défaillance de la procédure d'arrêt d'urgence après une erreur.



(a) Erreur de type "*Blind course*" puis arrêt d'urgence immédiat



(b) Erreur de type "*Steersman*" avec arrêt d'urgence tardif

FIGURE 3.9 – Deux exécutions où le robot ne perçoit pas la sortie de rang et sort des limites du champ.

qui entraîne les deux violations **P3_4** que nous avons observées. L'une de ces exécutions est illustrée à la figure 3.8. Il est intéressant de noter que le robot signale une erreur de type *"Blind course"* (trajectoire à l'aveugle) au moment de la collision. C'est également le cas dans environ 25% des collisions qui se produisent au milieu d'une phase de suivi de ligne. Cela suggère qu'une partie de ces collisions pourrait être due à une perte de perception des rangs lors du désherbage. Dans l'exécution présentée figure 3.8, une violation de la propriété **P8_1** s'ajoute, le robot poursuit sa trajectoire dans la rangée voisine au lieu de s'arrêter immédiatement.

En ce qui concerne les défaillances de sortie de champ, le sous-cas **P6_3** (dépassement en fin de zone de travail) suggère également un problème de mauvaise perception. Les deux parcours défaillants (voir figure 3.9) entraînent respectivement une erreur de type *"Blind course"* et une erreur de type *"Steersman"* plus de 1,5 mètre après le passage des piquets rouges. Cela indique que le robot n'a pas correctement perçu la de fin de zone de travail : il tente encore d'effectuer un suivi de

Défaillances	# occurrences
D1 - Mauvaise trajectoire d'approche d'inter-rang après un demi tour	11
D2 - Mauvaise trajectoire de sortie d'inter-rang avant un demi tour	2
D3 - Le robot saute une zone de travail après un demi-tour	2
D4 - Le robot perd le cap pendant le suivi de ligne	2
D5 - Déviation de la trajectoire lors de légumes manquants	2
D6 - Le robot s'arrête inopinément lors de la détection des piquets rouges	2
D7 - Mauvaise trajectoire en début de zone de travail laissant une large zone non désherbée	1
D8 - Le robot s'arrête en dehors des limites du champ en fin de mission	1

TABLE 3.4 – Défaillances de la navigation observées pendant les tests sur le terrain.

ligne longtemps après être sorti de l'inter-rang.

3.3.4 Résumé de l'analyse

En résumé, l'analyse qualitative des exécutions comportant des défaillances donne les problèmes hypothétiques suivants :

— Un problème majeur de demi-tour, affectant la capacité à effectuer une mission en toute sécurité lorsqu'il y a plus d'une rangée à désherber.

— Un problème de concurrence retardant l'arrêt du robot en cas d'erreur.

— Un problème de suivi de ligne, peut-être en relation avec une perte de perception de rangée.

— Un problème de détection de fin de rangée survenant dans de rares cas.

3.4 Comparaison avec les résultats de test sur le terrain

Pour cette phase comparative, nous avons eu des réunions conjointes et effectué une visite d'une semaine dans les locaux de Naïo Technologies. Cela nous a permis de rencontrer les personnes clés en charge des tests et du développement de *Ozcore*. Nous avons également pu consulter les notes rapportant les différentes sessions de tests sur le terrain. Nous avons discuté avec les ingénieurs un ensemble de traces d'exécutions défaillantes, avec l'aide d'un outil de rejeu de données développé par Naïo comme aide au diagnostic.

Parallèlement et indépendamment des tests en simulation, les tests sur le terrain ont également révélé des problèmes dans le logiciel *Ozcore*. Les rapports des cinq sessions de test correspondant à la même version du logiciel nous ont été partagés pour cette étude. Chacune de ces sessions nécessite un certain temps pour charger le robot dans la camionnette, se rendre sur le site expérimental, décharger le robot, initialiser les expériences et faire de même en sens inverse pour le retour. La durée effective des tests est généralement d'une ou deux heures pour une session

Fautes (*Issues*)	Tests sur le terrain	Tests en simulation	Réelle / Parasite
I1 - Demi-tour	✓	✓	Réelle
I2 - Marge de manoeuvre pour le demi-tour	–	✓	Réelle
I3 - Heuristique lors d'une perte de perception transitoire	✓	✓	Réelle
I4 - Traitement d'image des piquets rouges	✓	–	Réelle
I5 - Alignement en début d'inter-rang	✓	(✓) (avec P2')	Réelle
I6 - Patinage / Odométrie	✓	–	Réelle
I7 - Détection d'une fin de rangée	?	?	?
I8 - Arrêt d'urgence lors d'une erreur	–	✓	Parasite
I9 - Vitesse en pente descendante	–	✓	Parasite

Table 3.5 – Comparaison des fautes révélées par les tests sur le terrain et en simulation : "✓" = révélée, "–" = non révélée, " ?" = pas de conclusion par manque de diagnostic.

d'une demi-journée. Le tableau 3.4 est construit à partir des notes prises par les opérateurs de tests, il énumère les défaillances de navigation observées. Les procédures de test et l'oracle sont manuelles. Par exemple, l'opérateur entre une mission de désherbage via l'interface utilisateur. À un moment donné, il détermine visuellement qu'une trajectoire d'entrée de rang est inadéquate et arrête le robot avant qu'une collision ne se produise.

La comparaison avec les tests en simulation a commencé environ quatre mois après la cinquième et dernière session de test de Naïo. Les ingénieurs de Naïo avaient donc du recul sur le diagnostic des problèmes logiciels ainsi que sur leur résolution. Cela nous a ensuite permis de nous assurer que les défaillances observées en simulation révélaient les mêmes problèmes que ceux trouvés grâce aux tests sur le terrain. En raison du grand nombre de défaillances en simulation, il n'a pas été possible de procéder à une analyse conjointe de chacun d'entre eux. La comparaison s'est plutôt appuyée sur l'analyse préparatoire de l'ensemble des défaillances (présentée dans la section précédente) et sur une analyse plus approfondie de quelques traces d'exécutions particulièrement intéressantes.

Les résultats finaux de la comparaison sont présentés dans le tableau 3.5. Ils comprennent des fautes réelles, et des problèmes parasites ou non diagnostiqués, dont les détails sont développés ci après. Dans la discussion détaillée, nous expliquons le lien que nous faisons entre chaque faute et les défaillances sur le terrain (Di, Table 3.4) ou en simulation (violation de propriété Pi_j, Figure 3.6) qu'elle a pu causer.

— **Faute réelle I1** : Tout comme les tests en simulation, les tests sur le terrain

ont révélé un problème majeur lors du demi-tour causant la majorité des défaillances. Il existe une bonne correspondance entre les défaillances observées par chaque type de campagne de test : le robot entre dans la mauvaise rangée (**D3** vs. **P3_1**, **P3_2**, **P3_3**). Le robot a un positionnement et/ou un angle inadéquat pour son entrer dans l'inter-rang (**D1** vs. **P5_1**, **P6_1**). D'après l'analyse de Naïo, le problème du demi-tour provoque également de mauvaises trajectoires de sortie de rangée (**D2**). En effet, lorsque le robot perçoit les piquets rouges, il commence à préparer le demi-tour et la trajectoire d'approche peut entrer en collision avec la fin de la ligne. Il est donc plausible qu'une partie des collisions en bout de ligne observées en simulation (**P5_3**) soient également dues au demi-tour et pas seulement à des problèmes de perception. Cela n'a toutefois pas pu être démontré. Le problème du demi-tour a été considéré comme critique par les développeurs de Naïo et beaucoup d'efforts ont été déployés pour le résoudre. N'étant pas en mesure d'identifier une partie de code fautive en particulier, les développeurs ont entièrement redéveloppé la fonctionnalité à partir de zéro en se basant sur une autre stratégie de contrôle du demi-tour.

— **Faute réelle I2** : Le dépassement des limites du champ par les manoeuvres de demi-tour en simulation (**P6_2**) n'a pas été constaté lors des tests sur le terrain. L'opérateur en charge des tests n'a pas vérifié l'amplitude précise des manoeuvres, un dépassement a donc pu se produire mais rester inaperçu. Après analyse, le problème a été confirmé par Naïo. La planification interne des manoeuvres ne prend pas une marge suffisante pour éviter de dépasser la limite dans tous les cas. Exiger des manoeuvres plus courtes rendrait le demi-tour encore plus difficile et donc plus délicat à implémenter. L'entreprise a donc préféré réviser les conditions d'utilisation du robot afin de prévoir plus d'espace pour le demi-tour.

— **Faute réelle I3** : Les défaillances **D4** et **D5** révèlent un problème dans la gestion de la navigation pendant les pertes de perception transitoire. Dans le cas de la défaillance **D4**, la perte était due à un aléa de détection. Cette situation est causée par le bruit de mesure sur les capteurs ou par une déformation du terrain qui change soudainement l'attitude du robot. Dans le cas de la défaillance **D5**, la perte de perception était due à un "trou" dans la rangée de légumes (plusieurs plantes consécutives n'ont pas poussé). Lorsque les rangées de légumes repères sont perdues, le robot considère heuristiquement une direction probable basée sur son comportement passé. L'objectif est de maintenir le robot sur sa trajectoire jusqu'à ce qu'il perçoive à nouveau les rangées de légumes et ré accroche la fonctionnalité de suivi ou jusqu'à ce qu'il signale une erreur de type "*blind course*" (trajectoire sans visibilité). Cette heuristique de maintien de trajectoire est risquée et peut se tromper en envoyant le robot vers des légumes. Ce diagnostic est cohérent avec nos observations en simulation où l'on peut voir des collisions au milieu des rangées de légumes et des erreurs de type "*blind course*" associées. Pour vraiment confir-

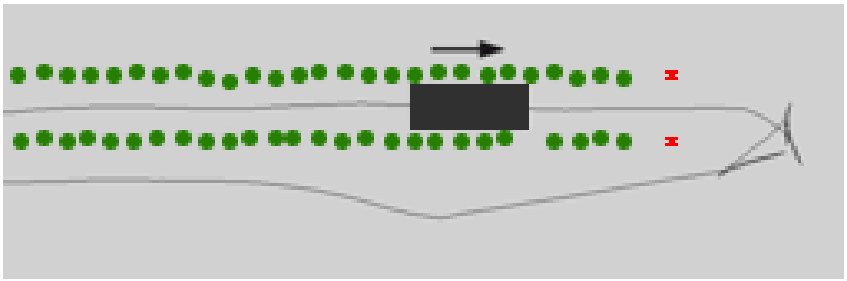
mer l'observation du problème en simulation, nous avons aussi sélectionné une exécution comportant une collision au milieu d'une rangée mais pas d'erreur "*blind course*". La trace a été analysée à l'aide d'un outil de rejeu développé par Naïo. Le cas de test choisi est un terrain relativement accidenté, ce qui est difficile pour la perception basée sur le LIDAR : les faisceaux laser ont des inclinaisons erratiques lorsque le robot monte et descend.

Le rejeu des données nous a confirmé une perte de perception transitoire des plantes voisines, et la modification de trajectoire qui en résulte. En conclusion, la faute dans l'heuristique de maintien de trajectoire est bien décelée par nos tests en simulation, pour des traces avec ou sans levée d'erreur "*blind course*". Cependant, nos tests n'ont reproduit que les scénarios comportant des aléas de détection sur des terrains irréguliers, et non les scénarios avec un grand trou dans une rangée. Les rangées générées présentent des trous de 2 ou 3 plantes manquantes au maximum, ce qui n'est pas assez stressant en comparaison avec un vrai champ. A l'inverse, les irrégularités du terrain virtuel sont plus stressantes que dans le monde réel et induisent de nombreux aléas de détection.

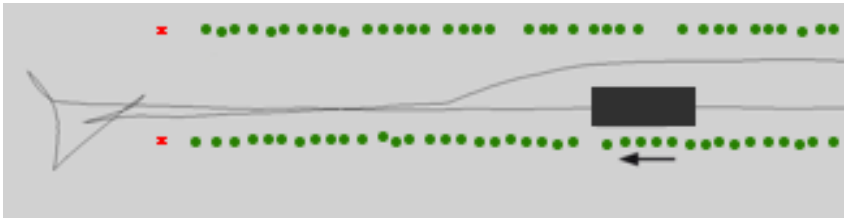
Naïo étudie actuellement des heuristiques alternatives pour améliorer la tolérance aux pertes de perception transitoires, mais aucune d'entre elles n'est encore intégrée à la plate-forme *Oz*. En ce qui concerne les trous (plusieurs légumes successifs manquants) dans une ligne, il est conseillé aux agriculteurs de mettre des éléments de substitution (par exemple, des piquets) pour réparer la ligne.

— **Faute réelle I4** : La défaillance **D6** est causée par le traitement des images pour la détection des piquets rouges. Des artefacts dans le traitement des images peuvent induire la double détection d'un seul et même piquet, ce qui conduit le robot à lever une fausse erreur (faux positif) de type "*invalid markers spacing*". Cette erreur correspond au cas où le robot détecte une incohérence dans le placement des piquets rouges. Naïo a retravaillé le code du traitement d'images pour résoudre ce problème. Du coté de nos tests en simulation, l'oracle n'est pas prévu pour évaluer la pertinence des erreurs : il vérifie uniquement que le robot s'arrête après la levée erreur. Nous avons inspecté manuellement toutes les exécutions se terminant par une erreur de type "*invalid markers spacing*" afin de déterminer s'il y avait des faux positifs. Mais ce n'était pas le cas : dans toutes ces exécutions, le robot était entré dans une mauvaise rangée après un demi-tour, puis avait correctement détecté l'incohérence du nombre de piquets rouges à la fin de la rangée. Nous concluons que ce problème de détection double n'est pas reproduit en simulation. En effet, les images venant des caméras simulées sont très nettes et contrastées par rapport aux images réelles. Les piquets rouges se distinguent parfaitement du sol unicolore et du ciel, alors que la vision réelle peut subir des reflets ou d'autres effets non désirés.

— **Faute réelle I5** : La défaillance **D7** révèle une faute affectant l'alignement initial de la trajectoire de désherbage avec les rangées de légumes. En effet, il



(a) Mauvais alignement dans une zone de travail externe du champ.



(b) Mauvais alignement dans une zone de travail interne du champ (mission aller-retour).

FIGURE 3.10 – Problème d'alignement en début de zone de travail (**D7**)

arrive que le robot mette du temps à rejoindre un alignement correct et par conséquent, que le début d'un inter-rang ne soit pas bien désherbé. L'oracle utilisé pour notre campagne de test en simulation ne détecte pas ce type de mauvais comportement. Pour rappel, dans la conception initiale de l'étude, nous avions désactivé le détecteur correspondant à **P2** (maintien de la distance de référence avec la rangée de légumes courante). Pour déterminer si la situation n'a pas été détectée par l'oracle mais s'est quand même produite dans nos simulations, nous avons introduit a posteriori une version modifiée du détecteur ciblant **P2**. Pour ce faire, le seuil de détection a été modifié pour se concentrer uniquement sur les grands écarts de distances et nous contrôlons également le taux de variation de la distance. Avec ce détecteur **P2'**, nous avons pu observer le problème d'alignement initial en simulation. La figure 3.10a présente un scénario virtuel proche de celui observé sur le terrain : le robot s'écarte de la ligne en début de zone de travail puis met du temps à rectifier sa trajectoire, laissant les 2 à 3 premiers mètres non désherbés. Les tests en simulation ont également permis la découverte de nouveaux scénarios de mauvais alignement. L'un d'eux est intéressant, il concerne le désherbage en deux passages (figure 3.10b) : le robot commence parfois à désherber du mauvais côté de la zone pendant 3 mètres avant de rejoindre le bon côté. Nous avons également observé un cas avec des oscillations sur les 4 premiers mètres avant que la trajectoire ne se stabilise. Il est à noter que la longueur d'une rangée est généralement supérieure à 100 mètres dans les champs réels. Les zones non désherbées en début de rangée ne représentent donc qu'une toute petite portion de la zone de travail. De ce fait, Naïo considère ce problème

occasionnel comme un problème mineur et n'a pas cherché à corriger la faute.

— **Faute réelle I6** : La défaillance **D8** correspond à un arrêt standard hors champ, c'est à dire que le robot n'est pas en erreur et s'arrête à la fin de sa mission à plus de 1,5 mètre après les piquets rouges (hors des limites du champ). D'après l'analyse de Naïo, cette défaillance est dû à un dérapage excessif et à une odométrie erronée. Un correctif a ensuite été introduit pour mieux gérer le dérapage. Dans le simulateur, l'interaction des roues avec le sol du champ virtuel n'est pas précise en ce qui concerne les dérapages ou les glissements. Cela explique sans doute pourquoi cette défaillance (**D8**) n'a pas été observée en simulation.

— **Problème non diagnostiqué I7** : Dans nos tests en simulation, deux arrêts hors champ semblaient être dus à une mauvaise perception de la fin de la rangée (**P6_3** et Figure 3.9). Cela n'a jamais été observé au cours des cinq sessions de test sur le terrain que nous avons analysées. Cependant, Naïo a depuis observé de rares cas où le robot ne détecte pas la fin d'une rangée. Le problème n'ayant pas été diagnostiqué jusqu'à présent, il n'est pas possible de conclure à la concordance des tests.

— **Problèmes parasites I8 and I9** : Les dernières entrées du tableau 3.5 sont des problèmes intempestifs uniquement détectés en simulation, ou autrement dit des artefacts de simulation. L'arrêt d'urgence tardif suite à une erreur (violation de **P8**) n'a jamais été observé sur le terrain. Après analyse, cette défaillance est due à un problème dans les scripts de simulation, et non dans le logiciel testé. Par souci d'exhaustivité, le tableau 3.5 mentionne également la sur-vitesse parasite que nous avons observée dans des cas de pentes descendantes. Il s'agit en fait d'un artefact de la physique basse fidélité du simulateur, qui ne prend pas en compte le frein moteur. Ce constat avait entraîné la désactivation des détecteurs associés à la propriété **P4**.

3.5 Bilan et validité des résultats

Nous sommes maintenant en mesure de répondre à la question explorée par l'étude, et d'identifier les leçons apprises. L'étude pourrait être affectée par des menaces à la validité, que nous abordons également.

3.5.1 Réponse à notre question de recherche

Quels sont les avantages et les inconvénients de test en simulation en ce qui concerne :

— les fautes révélées lors des test sur le terrain,

— les fautes non révélées lors des test sur le terrain,

— les problèmes parasites causés par des artefacts de simulation ?

L'étude confirme que le potentiel du test en simulation pour trouver des fautes est sous exploité. Le test en simulation révèle des fautes qui, dans le processus de validation industriel, ne sont trouvées que lors des tests sur le terrain. La faute de conception majeure affectant le demi-tour en est un parfait exemple. Elle n'a pas été trouvée lors des test de fumée et a par la suite causé la plupart des défaillances observées sur le terrain. Pourtant, le problème n'a pas été difficile à déceler lors des tests aléatoires car la fréquence de ses occurrences est élevée en simulation. La simulation a également permis de reproduire le traitement inapproprié des pertes de perception transitoires, ainsi que les difficultés d'alignement en début de rangée.

La faute affectant la gestion du dérapage semble difficile à trouver avec une physique simplifiée : elle nécessiterait une reproduction précise de l'interaction entre les roues et le sol. De même, la faute affectant le traitement des images n'a pas été trouvée en simulation par manque de réalisme. Mais pour ce dernier cas, il serait possible d'ajouter des effets de perturbation simulés sur les images caméras (voir par exemple les travaux connexes suivants : [Zendel 2013] et [Zendel 2017]).

En plus de révéler des fautes, les tests virtuels fournissent divers exemples de mauvais comportements causés par ces problèmes. Cela est utile car les fautes trouvées lors des tests sur le terrain ne sont pas des bugs de bas niveau. Elles remettent en question les algorithmes et les heuristiques clefs sur lesquels reposent les capacités de perception, de décision et de contrôle du mouvement du robot. Pour de telles fautes, il n'y a généralement pas de corrections évidentes. Il est alors très utile pour les développeurs d'avoir un aperçu des différents effets indésirables pour pouvoir les corriger. Par exemple, les tests en simulation nous ont permis d'observer des défaillances de demi-tour additionnelles par rapport à celles observées sur le terrain : le cas où le robot repasse par une rangée déjà désherbée ou encore le cas où le robot entre en collision avec un piquet rouge pendant les manoeuvres de demi-tour. De même, dans les tests en simulation, les problèmes d'alignement en début de rangée n'ont pas seulement été observés sous la forme d'un large détour, mais également sous la forme d'une oscillation sur les premiers mètres ou encore d'un alignement du mauvais côté d'une zone de travail nécessitant deux passages. Ces diverses défaillances peuvent être obtenues en une seule fois, en effectuant une série de tests (typiquement pendant la nuit pour avoir les résultats disponibles dès le lendemain). A l'inverse, sur le terrain, les tests s'étendent sur une période plus longue, impliquant plusieurs sessions sur des jours différents.

Les tests en simulation ont même permis de découvrir une faute non trouvée lors des tests sur le terrain : l'espace prévu pour le demi-tour en extrémité des rangées est insuffisant. Trouver des fautes différentes n'était pas notre objectif premier, et les tests n'avaient pas été conçus pour explorer des cas limite ("corner case") difficiles à atteindre sur le terrain. La génération de test avait procédé par simple échantillonnage aléatoire du domaine d'entrée défini par les conditions d'utilisation du robot. Il est impossible de dire si des sorties hors des limites du champ se sont produites lors des tests sur le terrain mais n'ont pas été vues par l'opérateur. Quoi qu'il en soit, cette violation s'est produite en simulation et a été capturée par les détecteurs de l'oracle.

Du coté des inconvénients, la simulation peut introduire des problèmes parasites. Dans cette étude, nous en avons observé deux types. Le premier était dû à la basse fidélité du simulateur qui a induit un comportement impossible dans le monde réel : des sur-vitesses transitoires sur les pentes descendantes. Le deuxième était causé par une faute dans le code du simulateur qui retardait de manière non déterministe l'arrêt effectif du robot.

La nécessité de développer le simulateur doit être ajoutée comme un inconvénient du test en simulation. Le simulateur spécifique pour *Oz* est assez complexe en lui même. Les développements basés sur *Gazebo* sont techniquement non triviaux. Il n'est donc pas surprenant que le code de la simulation puisse présenter des problèmes. Au-delà de l'effort de développement initial, la maintenance s'est également avérée préoccupante en raison de la dépendance à une technologie externe et instable. Pour l'anecdote, juste avant notre visite chez Naïo, une mise à jour de *Gazebo* a cassé le simulateur. Il ne fonctionnait plus avec la nouvelle version.

3.5.2 Enseignements tirés de cette campagne de tests en simulation

L'ensemble des expériences présentées dans cette partie nous a permis de dégager quelques idées et recommandations pour le développement de campagne tests automatisés en simulation.

1. **Conception évolutive et extensible du générateur de test et de l'oracle.** Il est difficile de spécifier les environnements virtuels et les propriétés que l'oracle doit vérifier. En conséquence, la première version plate-forme de test aura très probablement des lacunes. Les différents éléments de la plate-forme de test doivent être continuellement améliorés au fur et à mesure que l'on acquiert de l'expérience sur le système ou sur des systèmes similaires. Pour *Oz*, les retours de notre partenaire industriel concernant les tests sur le terrain nous ont fourni des indications sur les améliorations à apporter. Le détecteur **P2** de l'oracle a été transformé en **P2'** car la condition de détection du mauvais alignement du robot était initialement inadaptée. Il est à noter que le détecteur **P2'** s'inspire d'un problème spécifique détecté sur le terrain, mais est plus général que celui-ci. Notre modèle pour la génération doit également être amélioré pour prendre en compte les grands trous dans une rangée (plusieurs légumes successifs manquants) et les aléas visuels affectant les images. A l'inverse, la génération actuelle est peut-être un peu trop stressante en ce qui concerne les irrégularités du terrain par rapport aux champs réels. Pour permettre des améliorations continues, le générateur de test et l'oracle doivent être conçus pour être évolutifs dès le départ. Pour le générateur, cela signifie qu'une solution générique est fortement souhaitable : on devrait pouvoir produire automatiquement de nouveaux cas de tests à chaque mise à jour du modèle, sans avoir à introduire du code spécifique. La recherche d'une telle solution fera l'objet de la deuxième partie de nos travaux.
2. **Spécification bien structurée du modèle de données.** La génération de tests doit être basée sur un modèle du monde bien structuré, pour faciliter

l'ajout, la suppression ou la modification des éléments. Dans cette étude, la modélisation structurelle a démarré en UML. L'implémentation a conservé cette même structure avec une interface uniformisée pour chaque élément. D'après l'expérience acquise avec le cas d'étude *Oz*, l'approche de modélisation doit également tenir compte des contraintes sémantiques sur les paramètres attachés aux diffèrents éléments. Pour cette étude, nous avons complété le modèle structurel par une grammaire attribuée spécifiant les configurations valides des valeurs des paramètres.Toute solution générique de génération devra ainsi prendre en entrée des modèles de données présentant à la fois un caractère structuré et contraint. La modélisation peut être maintenue à un haut niveau d'abstraction en utilisant des procédures spécifiques pour concrétiser les tests.

3. **Séparer clairement l'enregistrement des données (en ligne) et l'analyse (hors ligne).** L'oracle de test est basé sur des propriétés. Il est composé d'un ensemble de détecteurs (au moins un par propriété) permettant de déceler les mauvais comportements. Lorsque la spécification de l'oracle part de zéro, les cinq aspects du comportement que nous avons identifiés peuvent servir de guide pour déterminer les propriétés à vérifier. Afin de faciliter les modifications et les extensions de l'oracle, nous recommandons une analyse hors ligne des traces de test. Il devrait y avoir une séparation stricte entre l'enregistrement des données brutes, qui est effectué pendant les simulations, et l'analyse potentiellement complexe des données qui doit être effectuée par la suite. De cette façon, les détecteurs peuvent être désactivés, révisés ou ajoutés sans modifier le code de simulation. De plus, des versions alternatives de l'oracle (par exemple, avec le détecteur **P2'** au lieu du **P2**) peuvent être étudiées sans avoir à ré-exécuter les tests. Cette méthode est donc encore plus avantageuse lorsque la phase d'exécution des tests est longue.

   L'enregistrement des données brutes doit être aussi complet que possible, au-delà de l'ensemble de données minimal requis par les détecteurs prévu au moment de l'exécutions. Il faudrait idéalement prendre en compte autant de données que possible pour couvrir les besoins futurs. L'instrumentation doit être placée à la fois sur le système sous test et sur le simulateur. L'objectif est de capturer le point de vue "subjectif" du robot (par exemple, ses positions perçues, les commandes qu'il envoie et les erreurs qu'il soulève) et la situation objective de la simulation (par exemple, les positions réelles et les événements de collision). La complexité de certains détecteurs s'accompagne de la nécessité de prendre en compte le contexte. Les données brutes doivent donc être analysées en relation avec l'environnement du robot et sa mission.

4. **Gérer la fidélité et la complexité du simulateur.** Le retour d'expérience sur la plate-forme de simulation utilisée dans cette étude est assez mitigé. D'une part, la simulation basée sur *Gazebo* a démontré son efficacité pour révéler des fautes réelles, dont une majeure. Elle offre donc la perspective d'alléger l'étape de test sur le terrain en lui déléguant une partie des tests.

D'autre part, le développement et la maintenance du simulateur se sont avérés lourds pour une petite entreprise comme Naïo. Puisque nos résultats montrent une efficacité dans un environnement virtuel dont la fidélité est limitée, on peut alors se demander si une plate-forme de simulation bien plus légère et beaucoup plus facile à maintenir ne serait pas suffisante. De plus, une telle plate-forme de simulation serait plus rapide, ce qui permettrait de réaliser un plus grand nombre de tests en moins de temps.

3.5.3 Obstacles éventuels à la validité

L'étude de cas présentée dans cette partie est exploratoire, elle est basée sur une analyse qualitative (les types de défaillances et les problèmes qui les causent).

La *validité de construction* concerne l'identification correcte des problèmes trouvés par chaque type de test. Le principal risque relatif aux tests sur le terrain provient du fait que la version du logiciel embarqué n'était pas exactement la même d'une session de test à l'autre. Le code était globalement le même mais des petits patchs (correctifs) y ont été ajoutés. La comparaison avec nos tests en simulation est donc basée sur le recul dont disposent nos partenaires de Naïo sur le diagnostic des problèmes. Par exemple, la première session de test sur le terrain a abouti à l'observation de défaillances qui sont rétrospectivement attribuées au problème majeur du demi-tour et au problème d'alignement initial du robot. Les autres problèmes ont fait surface plus tard, mais nous estimons qu'ils étaient bien présents dès le début. En ce qui concerne le problème majeur du demi-tour, les mêmes défaillances ont persisté dans toutes les sessions de test malgré les tentatives de patch. Hors du cadre de cette étude, ces défaillances ont même subsisté à travers plusieurs versions ultérieures avant que le développement du demi-tour ne soit décidé.

A l'égard des problèmes décelés lors des tests en simulation, il n'a pas été possible d'examiner toutes les exécutions présentant des défaillances avec les ingénieurs de Naïo. Bien qu'il y ait une forte probabilité que les défaillances observées soient toutes causées par les problèmes décrits dans le tableau 3.5, il reste tout à fait possible qu'une partie de ces défaillances soient de causes inconnues. Si celles-ci étaient des artefacts de simulation, cela diminuerait le bénéfice des tests en simulation. Au final, nous ne pouvons pas exclure l'existence de problèmes inconnus, mais nous avons veillé à ce que tous les cas de défaillances soient cohérents avec les problèmes identifiés dans cette l'étude.

Le risque de compréhension différente des problèmes et donc de mauvais diagnostic a été atténué par les interactions établies au cours de l'étude. Les réunions multiples ainsi que le séjour sur place avec les développeurs ont permis des discussions approfondies sur les problèmes. En particulier, le séjour au sein de l'entreprise a été suivie d'un débriefing entre pairs avec d'autres chercheurs. Nous sommes ensuite restés en contact avec les équipes de Naïo pour éclaircir certains points techniques rencontrés pendant nos analyses. Finalement, avant la publication des résultats, la liste des problèmes a été examinée et rediscutée conjointement avec nos partenaires industriels de chez Naïo.

Dans ces travaux, la *validité interne* concerne la limitation des biais dans la conception du processus tests en simulation, en particulier les biais dus à une certaine connaissance préalable des fautes dans le logiciel. Au moment de la conception, les problèmes **I1** à **I9** nous étaient tous inconnus. Ils étaient même inconnus pour Naïo. Nous ne connaissions pas non plus les résultats des tests sur le terrain lorsque nous avons analysé les cas défaillants pour en extraire un ensemble de problèmes hypothétiques.

La *validité externe* concerne la généralisation des résultats au-delà du cas d'étude *Oz*. Le contexte de l'étude est celui d'un logiciel robotique développé par une petite entreprise. Les technologies de simulation utilisées par Naïo sont représentatives de ce contexte : la plateforme *Gazebo* et le middleware (intergiciel) ROS sont très répandus en robotique. Les difficultés rencontrées au cours de l'étude (problèmes de spécification, instabilité de la technologie de simulation, fautes parasites dues à la physique simplifiée ou à des défaillances dans le code de simulation) ne sont pas spécifiques à l'exemple du cas d'étude *Oz*. Les résultats concernant l'efficacité des tests peuvent quant à eux être plus dépendants de l'application. Cependant, nous considérons qu'il est utile d'étudier l'efficacité de notre approche sur un exemple industriel à l'échelle réelle. Ces résultats viennent compléter d'autres résultats empiriques sur des exemples académiques, par exemple, Sotiropoulos et al. (2017) [Sotiropoulos 2017] ; Timperley et al. (2018) [Timperley 2018]. Enfin, les approches adoptées pour générer des tests et détecter les mauvais comportements sont générales, leur principe peut être réutilisé en dehors de notre étude de cas.

3.6 Conclusion

Face à la complexité croissante des systèmes autonomes, les tests deviennent l'un des plus grands défis. Ce chapitre et plus globalement cette partie, traite de la validation haut niveau de ces systèmes (c'est-à-dire au niveau de la mission). Après avoir spécifié le modèle d'entrée et l'oracle, nous avons étudié les avantages et les inconvénients des tests basés en simulation en comparaison avec ceux sur le terrain. Enfin, nous avons fourni quelques recommandations pratiques, basées sur notre expérience, pour le déploiement des tests en simulation. Le robot *Oz*, développé par l'entreprise Naïo Technologies, a servi de cas d'étude pour mener nos travaux à échelle réelle. Ces travaux ont été réalisés à la fois en laboratoire et en entreprise, Ils ont été accompagnés de nombreuses interactions pour la conception de l'étude de cas et l'interprétation des résultats.

La génération des cas de test (incluant des champs de légumes en 3D et des missions de désherbage) est basée sur un modèle structurel en UML et une grammaire attribuée pour exprimer les contraintes. Les mots valides de la grammaire représentent les configurations valides des paramètres du modèle (le génotype), à partir desquelles est produit le contenu concret des cas de test (le phénotype). L'oracle de test a été conçu en utilisant cinq grandes catégories de propriétés et implémenté

pour analyser automatiquement les traces de test. Le non-déterminisme des résultats de test a été anticipé, ce qui a donné lieu à plusieurs exécutions des mêmes cas de test.

Le principal résultat de l'étude a été de montrer que des fautes logicielles majeures pouvaient être révélés par la simulation plutôt que sur le terrain. Les tests générés de manière aléatoire ont même permis de découvrir une nouvelle faute qui n'avait pas été trouvée lors des tests sur le terrain. Le test en simulation présente beaucoup d'atouts mais il faut tout de même considérer les défaillances parasites et des problèmes de maintenance de la plate-forme de simulation.

Le degré de fidélité du simulateur requis pour une campagne de tests est une question ouverte. On peut s'attendre à ce qu'il ait un fort impact sur le rapport coût / efficacité des tests. Par exemple, une haute fidélité peut être nécessaire pour détecter les problèmes liés au traitement des images caméras ou à l'interaction physique entre les roues et le sol. Dans une telle situation, le temps et les ressources de calcul peuvent augmenter et devenir prohibitifs (cela a d'ailleurs été le cas avec le simulateur pour *Oz*). De plus, un code de simulation complexe peut être bugué et difficile à maintenir. À l'inverse, un simulateur plus léger risque de laisser beaucoup plus de défaillances non détectées avant les essais sur le terrain (et aussi d'introduire des problèmes parasites), En revanche, il nécessite moins de ressources, accélère la simulation et est plus facile à maintenir. La question de la fidélité de la simulation n'a pas de réponse simple, et il est intéressant de mentionner comment la stratégie employée chez Naïo a évolué depuis le début de cette étude. A l'époque, leur effort consistait à rendre la simulation aussi réaliste que possible avec les ressources disponibles. Par la suite, ils ont décidé d'abandonner le simulateur basé sur *Gazebo* au profit d'un simulateur beaucoup plus léger, développé en interne. Les tests en simulation sont également plus systématiques qu'à l'époque de l'étude, et portent désormais sur un ensemble de cas de test variés.

Les travaux qui suivent (voir partie II) portent sur la génération de cas de test. Ils conservent les principes du générateur spécifique pour *Oz* (génotypes bien structurés formant un objet de première classe, vérification et manipulation des éléments du génotype, génération mêlant éléments fixes et libres) afin de développer un générateur de cas de test plus générique et réutilisable.

Deuxième partie

Génération aléatoire de données structurées variées et contraintes

Présentation générale

Beaucoup d'applications nécessitent des cas de test constitués de données riches et structurées. Le cas d'étude de la partie précédente (le robot autonome agricole *Oz*) en est un parfait exemple. Pour le tester en simulation, des environnements agricoles cohérents ont été construits. Dans le contexte plus général du test de robots mobiles, un cas de test inclus communément la description d'une scène 3D dans laquelle le robot virtuel est placé. Pour un outil 3D quelconque, cette scène est typiquement représentée par des données structurées composées d'éléments de types variés dont le nombre et les attributs doivent satisfaire des contraintes sémantiques. Dans un contexte encore plus global, les données structurées de taille variable avec des éléments interdépendants sont assez courantes lors du test de systèmes de traitement d'information, de protocoles de communication etc.

La production de données synthétiques pour le test de ces systèmes implique généralement l'utilisation de générateurs spécifiques. Les travaux présentés dans cette partie traitent d'une approche plus générique et basée sur un modèle des données. L'objectif est permettre à l'utilisateur de déclarer des structures de données ainsi que des contraintes entre ces données, puis de générer des données valides automatiquement. Les approches similaires détaillées chapitre 1 section 1.6 demandent à l'utilisateur d'insérer des bouts de code permettant d'ajuster la génération pour générer des données cohérentes au regard des contraintes. A l'inverse, notre approche se veut purement déclarative, l'utilisateur ne doit pas spécifier comment construire les différents éléments du modèle pour en satisfaire les contraintes.

Pour atteindre les objectifs fixés par cette approche, l'utilisation de solveurs de contraintes vient naturellement à l'esprit. Il existe un certain nombre de solveurs très performants disponibles sur étagère, notamment des solveurs SMT (*Satisfiability Modulo Theories*) pouvant traiter des contraintes sur des réels. Cependant, deux raisons font qu'aucun de ces solveurs n'est satisfaisant au regard de notre approche. Premièrement, ils resolvent des problèmes avec des structures de données à taille fixe : ils ne s'adaptent pas aux structures de taille variable, lorsque l'ensemble des contraintes évolue dynamiquement au fur et à mesure que des instances de données sont créées ou supprimées. Deuxièmement, les solveurs sont déterministes et renvoient une unique solution. Cette solution est pour le solveur la plus "évidente" permettant de satisfaire toutes les contraintes (par exemple toutes les variables d'un problème à zéro). Certains outils basés sur des solveurs permettent de faire une liste exhaustive des solutions jusqu'à une taille maximum. Mais la taille doit rester très petite, et ce sont des problèmes pour lesquels le nombre de solutions est énumérable (par exemple, le cas où on ne considère pas de variables numériques réelles). Ceci n'est pas adapté lorsque l'on souhaite traiter des types de données plus riches, et générer de multiples cas de tests et obtenir de la diversité pour couvrir l'espace des solutions des valides.

Cette partie développe la solution que nous avons construite pour pallier ces problèmes. Nous proposons une nouvelle façon d'exploiter la résolution des contraintes

en assurant la diversité des solutions, en taille et en valeur des variables. Notre solution met dynamiquement à jour les contraintes à mesure que des instances de données sont créées ou supprimées et combine des tirages aléatoires avec de la résolution de contraintes pour forcer la diversité des solutions. Plus précisément, nous injectons des valeurs aléatoires dans les requêtes envoyées au solveur.

TAF (*Testing Automation Framework*) est l'outil développé dans le cadre de cette thèse qui implémente cette approche et dont la vue d'ensemble est présentée ci-dessous.

Vue d'ensemble de *TAF*

TAF permet à l'utilisateur de spécifier un modèle dans un langage basé sur XML. Il peut ensuite utiliser *TAF* pour générer des données, compléter des données partielles ou encore mettre en forme les données générées vers un format propre à l'application ciblée.

Pour illustrer les fonctionnalités qu'offrent *TAF*, nous proposons de reprendre le cas d'étude de la première partie dans une version simplifiée. Il s'agit du test en simulation d'un robot autonome agricole nommé *Oz*, qui a pour mission de désherber des champs de cultures maraîchères. Un cas de test pour ce système consiste en une mission de désherbage (un ensemble de paramètres à fournir au robot) dans un champ de culture virtuel (voir Figure 3.11).

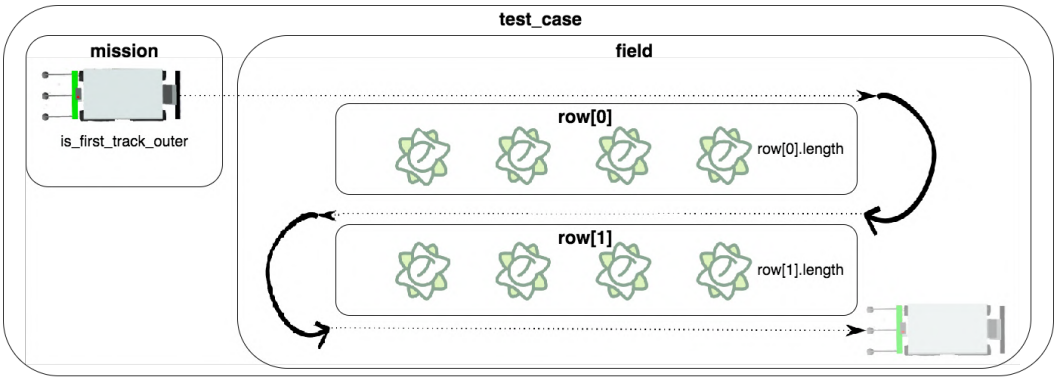


FIGURE 3.11 – Le cas de test virtuel et ses paramètres pour le robot *Oz*

Pour rendre cette partie indépendante de la première, une courte description du cas d'étude est donnée ci-après. Un champ est composé d'un nombre arbitraire de rangées de légumes. Pour mener à bien sa mission de désherbage, le robot doit naviguer entre ces rangées. Lorsqu'il arrive à la fin d'une rangée, il fait demi-tour et accède à la suivante. On suppose un champ de monoculture, c'est-à-dire contenant un seul type de légumes, dans notre cas des choux ou des poireaux. L'un des paramètres de la mission est *is_first_track_outer*, qui est une variable booléenne qui indique si le robot commence ou non sa mission sur un bord extérieur du

champ. Cette variable est à trivialement vraie lorsque le champ n'est composé que d'une rangée.

La génération de ces cas de test virtuels peut se faire en deux étapes. La première étape génère des cas abstraits à partir d'un modèle de données, tandis que la seconde produit des entrées de test concrètes qui peuvent ensuite être introduites dans la plate-forme de test. Cette séparation en deux étapes distinctes permet de concentrer la modélisation sur les éléments de données de haut niveau, sans se laisser submerger par la complexité des détails concrets ou des problèmes liés au format final des données. Par exemple, un cas de test abstrait pour le robot *Oz* ne spécifie qu'un nombre de rangées de légumes et un type de légume. Les entrées de test sont ensuite créees avec pour notre exemple un fichier de description de scène 3D dans un format compatible avec le simulateur. Ces données comportent par exemple la position exacte de chaque légume sur le champ.

Notre outil *TAF* se charge de ce processus de génération de tests en deux temps (voir figure 3.12). Pour cela, il fournit un langage basé sur XML permettant de spécifier des *modèles*, c'est-à-dire des modèles structurés des éléments composant un cas de test abstrait. La première étape de *TAF* est de générer des cas de test abstraits à partir du modèle. Cette étape constitue le coeur de cette partie du manuscrit et est détaillée et évaluée dans les chapitres suivants. Ces données sont ensuite transformées en entrées de test concrètes lors de la seconde étape. Cette étape n'est pas automatisable mais est facilitée par un *squelette de code* également créé à partir du modèle. L'utilisateur peut ensuite y définir les fonctions d'exportation personnalisées correspondants aux différents éléments de données. Ces fonctions d'exportation sont vides par défaut, une fois complétées elle produisent les fichiers dont le format dépend de la plate-forme de test. Cette seconde étape (`export`) ne présente que peu d'intérêt scientifique et ne sera pas donc pas détaillée dans ce manuscrit.

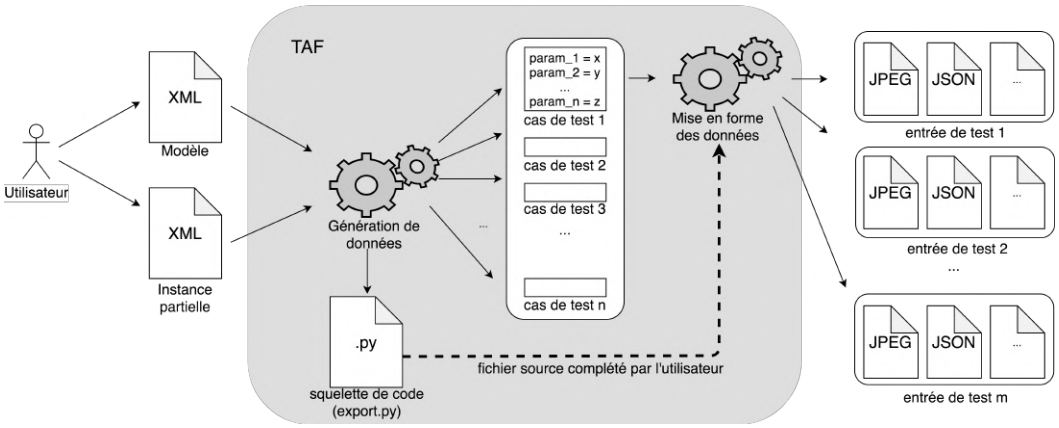


FIGURE 3.12 – Vue globale de l'outil *TAF*

L'une des principales caractéristiques de *TAF* est sa capacité à prendre en compte des contraintes. Par exemple, si le champ ne comporte qu'une seule

rangée, la mission de désherbage commence nécessairement sur un côté extérieur du champ. Un paramètre de mission doit alors être assigné en conséquence ("is_first_track_outer"). Une deuxième contrainte est que les rangées consécutives doivent avoir pratiquement la même longueur (à +/- 10%). Pour compliquer encore le problème, nous ajoutons la contrainte que les longueurs de la première et de la dernière rangée soient également proches de +/- 10%. De telles contraintes peuvent être exprimées dans le modèle et rendent la génération de tests difficile.

Pour ce type de modèle contraint, une stratégie possible consiste à générer de nombreux cas de test puis à ne retenir que ceux satisfaisant les contraintes (générer-puis-filtrer). Cependant cette approche est peu efficace. Supposons que nous ayons des dizaines de rangées : il faudrait alors un nombre important d'essais avant d'obtenir un ensemble de longueurs de rangées cohérentes. Il faut donc intégrer de la résolution de contraintes dans la génération. Il est important de noter que l'utilisation classique d'un solveur ne renvoie qu'une seule solution, bien souvent triviale. Par exemple, on va obtenir une solution avec toutes les rangées de légumes à la même longueur, cette longueur étant la borne minimale de son ensemble de définition. Pour contourner ces problèmes, *TAF* combine la génération aléatoire avec la résolution de contraintes, ce qui permet de produire efficacement des cas de test divers tout en satisfaisant les contraintes. *TAF* offre des fonctions de génération aléatoire réglables pour chaque type de données de base, et utilise le solveur SMT Z3 ([de Moura 2008]) pour la satisfaction des contraintes.

L'utilisateur a également la possibilité de fournir un *cas de test partiellement instancié*. Ce fichier d'entrée est facultatif. Il permet de définir les valeurs de tout sous-ensemble d'éléments de données du modèle. Par exemple, l'utilisateur peut demander un champ avec un nombre spécifique de rangées de légumes, ou encore demander à la fois ce même nombre de rangées de légumes avec un type de légumes particulier. *TAF* va alors générer les éléments de données manquants pour produire un cas de test complet et valide. Cette fonctionnalité de complétion offre une grande flexibilité pour explorer des sous-ensembles de l'espace d'entrée ou couvrir des cas extrêmes. En outre, le choix de formats de fichiers basés sur XML vise à faciliter la mise en oeuvre de stratégies de test intégrant *TAF*. Ainsi, le modèle et les fichiers d'instances partielles peuvent être manipulés pour demander des valeurs spécifiques, ajouter ou supprimer des contraintes, voire même changer la fonction de génération par défaut pour un élément donné.

Présentation des chapitres à venir

Cette partie est découpée en trois chapitres présentant chacun un aspect différent de *TAF*. Le premier (chapitre 4) décrit de manière détaillée le langage de description de modèle pour *TAF*. Il sert également de guide pour les utilisateur souhaitant expérimenter avec l'outil. Le chapitre suivant (chapitre 5) présente les différents algorithmes au coeur de *TAF*. À titre comparatif le principe de fonc-

tionnement de *PLEDGE* (le seul outil académique à notre connaissance ayant des fonctionnalités proches) est également décrit. Finalement le chapitre 6 évalue les performances et la couverture de *TAF* sur 4 cas d'étude : des champs de légumes virtuels pour le robot agricole *Oz*, des populations de contribuables pour un logiciel de gestion d'impôt, des images bitmap avec un dégradé de gris et des structures d'arbre dont la profondeur et la taille varient. Ce sont des exempes pour lesquels une approche de type générer-puis-filtrer s'avère très inéfficace. Parmi ces 4 cas d'étude 2 sont également modélisés sous *PLEDGE* pour comparer les résultats.

Chapitre 4

Langage spécifique au domaine (*XML-TAF*)

Sommaire

4.1 Introduction

Ce chapitre décrit le langage d'entrée pour *TAF*. Ce langage est utilisé dans deux fichiers comme présenté sur la figure 4.1. Le fichier **modèle** est requis pour la génération, il spécifie le modèle de données structurées et les contraintes. Le fichier **instance partielle** est quant à lui optionnel, il permet de guider le générateur en lui donnant une instance partielle des données. Nous avons basé notre langage sur XML car il permet une extensibilité de l'outil et une analyse syntaxique (parsing) simple. XML est un standard pour la sauvegarde et l'échange de données, il assure donc une interface classique. Le langage *XML-TAF* intègre trois concepts important pour la modélisation : des relations de composition permettant de hiérarchiser les éléments du modèles, des éléments avec un nombre d'instances variable, des contraintes dont la syntaxe est proche du solveur Z3. Un modèle simplifié du cas d'étude sur le robot agricole industriel *Oz* sert à illustrer ces différents concepts tout au long de ce chapitre. On cherchera ici à générer un champs de n rangées de légumes ainsi que les réglages du robot pour assurer sa mission sur ce champ (avec $n \in [1, 100]$).

Une présentation d'un modèle est proposée en section 4.2, allant de la description des structures des données jusqu'aux paramètres et leurs types associés. La section 4.3 se concentre sur l'expression des contraintes. Pour finir la section 4.4

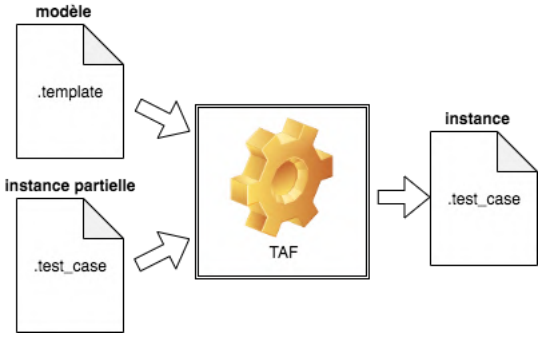


FIGURE 4.1 – fichiers d'entrées et de sorties de *TAF*.

présente le format des instances partielles et les possibilités qu'elles offrent. En plus, de fournir une description précise du langage de modélisation, ce chapitre s'appuie sur de nombreux exemples dans le but de servir de guide aux lecteurs souhaitant tester leurs propres modèles. *TAF* est disponible sur la page [Robert 2020a].

4.2 Structure du modèle : les noeuds et les paramètres

Cette section décrit le langage de modélisation basé sur XML spécifique à *TAF*. Plus précisément, elle présente les 2 principaux éléments structurels de nos modèles : <node> et <parameter>. Pour faciliter la lecture de ce chapitre, les noms de balise sont repérés par des chevrons (<...>) et les attributs associés à ces balises sont soulignés. La grammaire définissant la syntaxe *XML-TAF* est distribuée dans les différentes sections (figures 4.4, 4.5, 4.8, 4.4, 4.10 et 4.12).

4.2.1 Élément de type <node>

Pour comprendre dans quel contexte s'utilise les éléments de type <node>, il est nécessaire de partir d'un point de vue plus large. Un Template est composé de quatre balises XML différentes : <root>, <node>, <parameter> et <constraint> (Un exemple simple est donné à droite de la figure 4.3). La figure 4.2 présente le méta-modèle du langage. Ces balises contiennent toutes un attribut name. Elles sont imbriquées pour former un arbre dont la balise <root> est la racine. Mis à part la racine, un nœud non terminal de cet arbre est forcément représenté par une balise <node>. Chaque balise <node> peut donc avoir des enfants de type <node>, <parameter> ou <constraint>. Les balises <constraint> sont volontairement ignorées dans cette section qui se concentre sur les aspects structurels. Une description approfondie est donnée dans la section suivante (4.3).

Le langage *XML-TAF* est purement déclaratif. Sa structure est calquée sur celle des données que l'on souhaite modéliser. La figure 4.3 est un exemple de modélisation d'un environnement simplifié pour *Oz*. Elle décompose le processus de modélisation en trois étapes. La première étape consiste à identifier et isoler les dif-

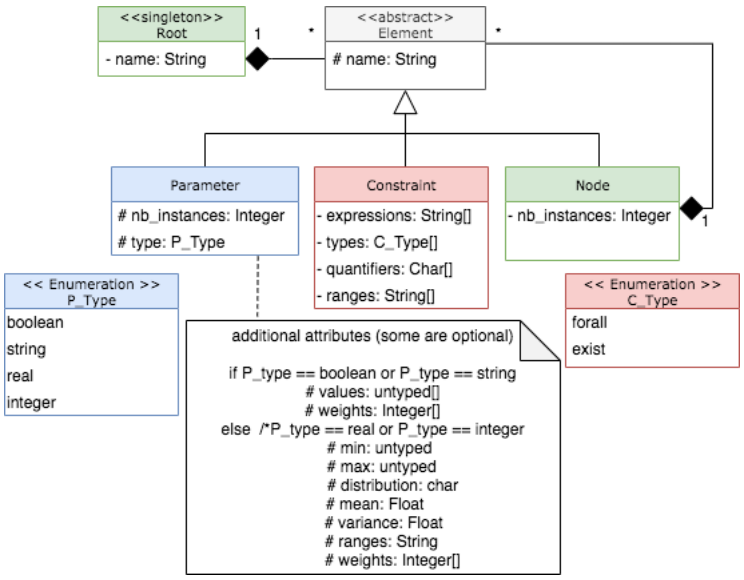


FIGURE 4.2 – Ontologie du langage spécifique au domaine à partir de digramme de classe UML.

férents éléments que l'on souhaite faire apparaître dans notre modèle (étape 1) : un champ avec des rangées de légumes. Ces éléments sont ensuite nommés et organisés hiérarchiquement en suivant les relations de composition (étape 2). Le champ (field) contient des rangées (row) de légumes ayant chacune une longueur (length). Ce découpage assure une structure d'arbre pour l'étape suivante. La dernière étape (étape 3) consiste à traduire cette structure d'éléments en une description équivalente dans la syntaxe propre aux fichiers modèle (étape 3). De manière très simplifiée, on utilise des <node> pour les objets parents (le champs ou encore les rangées) et des paramètres de base pour les définir (length pour définir la longueur en mètres d'un objet rangée). Seule cette étape 3 est détaillée dans la suite de ce chapitre. De plus, le terme **Template** y est utilisé en référence à ce modèle en *XML-TAF*.

Les balises <node> et <parameter> ont un attribut nb_instances (nombre d'instances) permettant de gérer la multiplicité en fixant le nombre d'instances. Pour le cas plus particulier de la balise <node>, on ne souhaite pas forcément que le nombre d'instances d'un élément soit prédéterminé avant la génération. Par exemple, on souhaite un nombre de rangées ("row") de légumes variable entre les différents cas de test. Le nombre d'instances de l'élément "row" doit donc pouvoir être un paramètre de génération. Pour cela, il suffit de définir son intervalle à l'aide des attributs min et max. Il est possible de déclarer une valeur minimale à 0 pour signifier la possibilité d'absence (ou le caractère optionnel) de ce <node>. Ce noeud et tous ses enfants sont alors ignorés si la valeur 0 est retenue en cours du processus de génération. Si aucun attribut ne spécifie explicitement la multiplicité, le nombre d'instances est fixé à 1 par défaut. Le Template simplifié à droite de la figure 4.3 illustre ce concept structurel. Le nombre d'instances du <node> "field" est forcé à

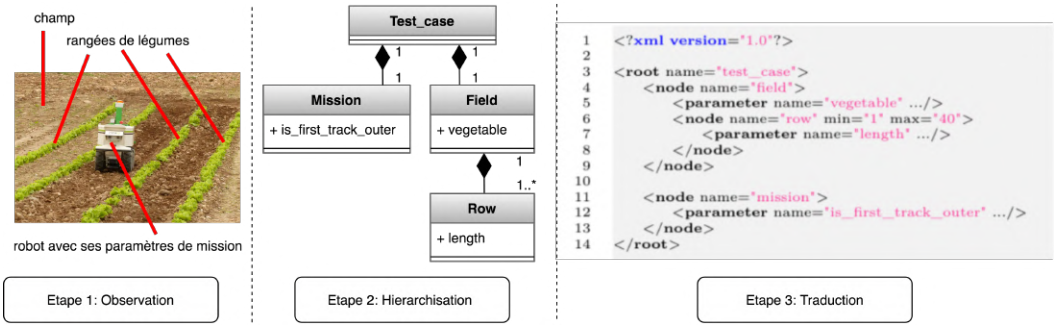


FIGURE 4.3 – Exemple du processus de modélisation pour *Oz*.

```
<float> ::= <integer> "." <integer>
<integer> ::= <non_nul_digit><digit_string>

<digit_string> ::= <digit_string><digit_string>
                 | <digit>
<string> ::= <string><string>
           | <letter>
           | <digit>
           | "_"
<non_nul_digit> ::= "1" | "2" | "3" .... "9"
<digit> ::= "0"
          | <non_nul_digit>
<letter> ::= "a" | "A" | "b" | "B" | "c" | "C" .... "y" | "Y" | "z" | "Z"
```

FIGURE 4.4 – Règles de production des types de base.

1 (L4) alors que le nombre d'instances du <node> "row" est défini dans l'intervalle $[\![1, 40]\!]$ (L6). Nous verrons dans la section suivante chaque instance de "row" qu'il est possible de désigner en utilisant la notation C : "row[indice]".

Les listing 4.4 et 4.5 fournissent un extrait de la définition du langage de modélisation au format BNF (*Backus-Naur Form*). La syntaxe BNF utilise les chevrons comme méta-symbole et peut donc prêter à confusion avec les éléments XML de ce chapitre. Dans le format BNF, les chevrons représentent les non-terminaux. On y remarque par exemple que l'attribut name doit obligatoirement commencer par une lettre et être constitué uniquement de caractère alphanumérique ou de soulignement (underscore). Les règles de production pour les non terminaux <parameter> et <constraint> seront traitées plus tard dans ce chapitre. Les règles de production des non terminaux <integer>, <float> et <string> données en figure 4.4 sont réutilisées dans les prochaines figures. De même les non-terminaux <name>, <nb_instances>, <integer_min> et <integer_max> sont réutilisés dans la définition de <parameter>. De manière générale, le langage *XML-TAF* est plus permissif que celui décrit par la grammaire. En effet, l'ordre des attributs n'a pas d'importance en XML.

```
<root> ::= "<root " <name> ">" <node> <parameter> <constraint> "</root>"
<node> ::= "<node " <name><count>  ">" <node> <parameter> <constraint> "</node>"
             | <node> <node>

<name> ::= "name="" <letter> <string> "" "
<count> ::=  <nb_instances>
           | <integer_min> <integer_max>
<nb_instances> ::= "nb_instances="" <integer> "" "
                 | ε
<integer_min> ::= "min="" <integer> "" "
<integer_max> ::= "max="" <integer> "" "
```

FIGURE 4.5 – Règle de production des éléments <node>.

4.2.2 Élément de type <parameter>

Les paramètres sont par défaut de multiplicité 1. Comme pour les <node>, l'attribut nb_instances peut être ajouté auquel cas le paramètre sera traité comme un tableau. Les paramètres peuvent être de 4 types (P_type sur la figure 4.2) : boolean, string, real, integer, regroupés en 2 catégories, les paramètres énumérés et les paramètres numériques.

— Les types de paramètres énumérés sont boolean et string. Ils sont définis par un attribut values permettant de lister les différentes possibilités. Pour la syntaxe, l'ensemble des valeurs possibles sont concaténées avec des points virgules comme séparateur pour former une longue chaine de caractères.

  Exemple :

```
<parameter name="vegetable" type="string" values="cabbage;leek"/>
<parameter name="is_first_track_outer" type="boolean"/>
```

  Ci-dessus, le <parameter> vegetable peut prendre soit la valeur “cabbage” soit la valeur “leek”. Le <parameter> is_first_track_outer ne comporte pas d'attribut supplémentaire car l'attribut values est implicite pour les paramètres de type boolean. A l'issue de la génération, les valeurs des paramètres énumérés sont disponibles sous forme de chaînes de caractères.

— Les types de paramètres numériques sont real et integer. Ils sont déclarés à l'aide de deux attributs obligatoires min et max permettant de définir un intervalle. Ces attributs doivent obligatoirement être des entiers pour la déclaration de paramètre de type integer. Notre implémentation de *TAF* arrondit les paramètres de type real à 5 chiffres après la virgule. Il est néanmoins possible de gérer des paramètres plus petits à l'aide de coefficients multiplicateurs (x1000 par exemple). La gestion de ces coefficients est laissée à l'utilisateur.

Exemple :

```
<parameter name="length" type="real" min="10.0" max="100.0"/>
<parameter name="integer_example" type="integer" min="−5" max="5"/>
```

Ci-dessus, le <parameter> length est un paramètre de type real qui peut prendre des valeurs réelles comprises entre 10.0 et 100.0. Similairement le <parameter> integer_example donne un exemple de définition du type integer défini entre -5 et 5.

Chaque paramètre a ainsi un domaine de définition qui est soit explicitement énuméré soit borné.

TAF associe un générateur à chaque paramètre de la structure. Le but est de produire des valeurs variées en spécifiant des lois de probabilités pour le tirage des valeurs des paramètres. Trois types de lois de probabilité ont été implémentées dans *TAF* :

— loi uniforme (“u”)

— loi discrète (“i”)

— loi normale (“n”)

Le choix se fait à partir de l’attribut distribution qui peut prendre trois valeurs sous forme d’un caractère, “u” pour uniforme, “n” pour normale et “i” pour intervalle. Quand rien n’est spécifié (comme dans les exemples précédents), les valeurs attribuées à chaque paramètre sont sélectionnées par un générateur uniforme “u” (générateur par défaut). Chacune des valeurs possibles d’un intervalle ou d’une énumération a alors la même probabilité d’être sélectionnée.

Le générateur par défaut peut être changé par l’utilisateur, en définissant soit une loi discrète (tout type de paramètre) soit une loi normale (paramètres de type numérique). Dans le cas des paramètres de type énumérés (boolean ou string), la loi discrète peut être sélectionnée et paramétrée à l’aide de l’attribut weights. Ce dernier permet d’associer un poids à chaque valeur possible de paramètre. Les poids sont obligatoirement des entiers nuls ou positifs séparés par des points virgules. Pour une définition valide, il est requis qu’une pondération soit associée à chaque valeur possible, le poids pouvant être nul. De plus, au moins une pondération doit être non nulle.

Exemple :

```
<parameter name="vegetable" type="string" values="cabbage;leek" weights="5;7"/>
<parameter name="is_first_track_outer" type="boolean" values="true;false" weights="0;1"/>
```

Ci-dessus les poids 5 et 7 sont respectivement associés aux valeurs “cabbage” et “leek” du <parameter> vegetable. A la fin du procédé de génération de *TAF*, le paramètre vegetable a donc plus de chance de prendre la valeur “leek”. Dans le deuxième exemple le paramètre is_first_track_outer prendra toujours la valeur

false car c'est la valeur correspondant à la seule pondération non nulle (cet exemple est donné à titre illustratif car il n'a bien sûr aucun intérêt pour la modélisation)

Dans le cas des paramètres de type numérique (integer ou real), *TAF* propose deux générateurs en plus du générateur uniforme par défaut. Le premier utilise une loi normale tandis que le deuxième permet de définir des intervalles pondérés.

Le générateur "n" selon une loi normale peut-être réglé à l'aide de la valeur moyenne et de la variance désirées (respectivement les attributs mean et variance). Lorsqu'une valeur n'appartient pas à l'intervalle strictement défini par les attributs min et max la valeur est tirée à nouveau (loi normale tronquée). Il est à noter que la valeur moyenne doit impérativement appartenir à l'intervalle et que la variance doit être positive ou nulle. Une variance nulle donne un générateur qui renvoie toujours la valeur moyenne. A l'inverse, une variance très grande fait tendre la densité de probabilité vers une loi uniforme sur l'intervalle.

Pour finir, lorsque la moyenne et la variance ne sont pas définis, les réglages de la figure 4.6 sont appliqués. Ces valeurs particulières permettent de centrer la densité de probabilité sur l'intervalle et d'assurer 95% des tirages dans l'intervalle.

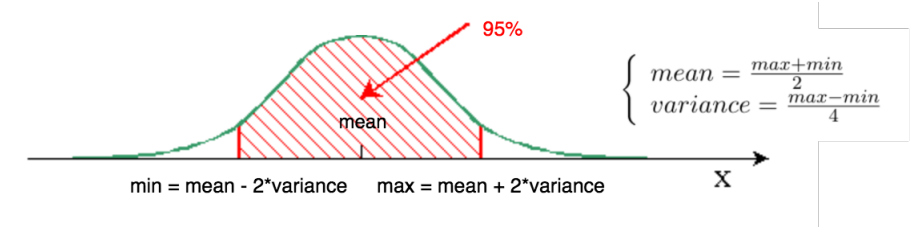


FIGURE 4.6 – Paramètres par défaut du générateur loi normale.

Le générateur permettant de définir des intervalles pondérés "i" est basé sur le générateur uniforme des paramètres énumérés. A la place de lister les valeurs possibles, on liste les intervalles avec l'attribut ranges. Il n'y a pas de restriction sur les intervalles tant que ceux-ci sont inclus dans l'intervalle de base du paramètre ($[min, max]$). Ils peuvent donc être disjoints ou avoir des intersections non nulles. La syntaxe est une chaîne de caractère contenant les intervalles séparés par des points-virgules. Les bornes doivent être encadrées par des crochets et séparées par une virgule. De manière similaire au générateur des paramètres énumérés, les pondérations associées à chaque intervalle sont définies par l'attribut weights. Si cet attribut est absent un poids de 1 est attribué par défaut à chaque intervalle (tirage uniforme). Lors d'un tirage, un intervalle est d'abord sélectionné en fonction des pondérations puis une valeur est choisie uniformément dans celui-ci.

Exemples :

Dans les exemples ci-dessus, le paramètre length est spécifié avec le générateur uniforme (le générateur par défaut). Le paramètre normal est défini avec le générateur loi normale avec le réglage par défaut (mean=55 et variance=22.5). Même chose pour le paramètre normal_2 mais avec un réglage personnalisé. Le paramètre

```
<parameter name="length" type="real" min="10.0" max="100.0" distribution="u"/>
<parameter name="normal" type="real" min="10.0" max="100.0" distribution="n"/>
<parameter name="normal_2" type="real" min="10.0" max="100.0" distribution="n"
                    mean="30.0" variance="50.0"/>
<parameter name="normal_3" type="real" min="10.0" max="100.0" distribution="n"
                    mean="10.0" variance="0.0"/>
<parameter name="interval" type="real" min="10.0" max="100.0" distribution="i"
                    ranges="  [10.0,50.0];[40.0,70.0];[80.0,90.0]  " weights="1;2;3"/>
```

normal_3 prendra toujours la valeur 10 car sa variance est nulle. Enfin, le paramètre interval est spécifié avec le générateur d'intervalles pondérés. 3 intervalles y sont décrits. Concernant les pondérations, le troisième intervalle a trois fois plus de chance d'être tiré que le premier.

La figure 4.8 détaille la règle de production de <parameter> ainsi que toutes celles sous-jacentes.

4.3 Expression des contraintes

Pour rappel, le modèle pour le cas d'étude *Oz* simplifié est donné figure 4.7. Le champ décrit peut être composé de 1 à 40 rangées de choux ou de poireaux (cabbage ou leek). Chaque rangée peut être d'une longueur (length) entre 10 à 100 mètres. On ajoute au modèle un paramètre de mission pour le robot (is_first_track_outer). Il s'exprime sous la forme d'un booléen, vrai si le robot est initialement positionné sur un bord extérieur de la première rangée, faux sinon.

```
1  <?xml version="1.0"?>
2
3  <root name="test_case">
4      <node name="field" nb_instances="1">
5          <parameter name="vegetable" type="string" values="cabbage;leek" weights="5;7"/>
6          <node name="row" min="1" max="40">
7              <parameter name="length" type="real" min="10.0" max="100.0"/>
8          </node>
9      </node>
10
11     <node name="mission" nb_instances="1">
12         <parameter name="is_first_track_outer" type="boolean"/>
13     </node>
14 </root>
```

FIGURE 4.7 – Modèle non contraint du cas d'étude simplifié *Oz*.

Si on donne ce modèle à *TAF*, on se retrouve avec des champs irréalistes comportant des longueurs de rangées absurdes (Par exemple, une rangée de 10 mètres entre deux rangées de 100 mètres). Pour avoir un champ cohérent, on souhaite des rangées de légumes de longueurs proches. Cette propriété peut s'exprimer à l'aide de contraintes. C'est l'objet de cette section où l'utilisation de contraintes dans un modèle pour *TAF* y est détaillée. Le même modèle y est ensuite complété à titre illustratif.

```
<parameter> ::= "<parameter " <name> <p_arg> <nb_instances> "/>"
                | <parameter> <parameter>
                | ε

<p_arg> ::= "type="boolean" "
          | "type="boolean" " <boolean_values>
          | "type="boolean" " <boolean_values> <boolean_weights>
          | "type="string" " <string_values>
          | "type="string" " <string_values_and_weights>
          | "type="integer" " <integer_min> <integer_max> <integer_attributes>
          | "type="real" " <real_min> <real_max> <real_attributes>

<boolean_values> ::= "values="[True;False]" "
                   | "values="[False;True]" "
<boolean_weights> ::= "weights="[" <integer> ";" <integer> "]" "

<string_values> :== "values="[" <val> "]" "
<val> ::= <string>
        | <string> ";" <string>

<string_values_and_weights> ::= "values="[" <string> <vnw> <integer> "]" "
<vnw> ::= ";" <string> <vnw> <integer> ";"
        | "]" weights="["

<real_min> ::= "min="" <float> "" "
             | "min="" <integer> "" "
<real_max> ::= "max="" <float> "" "
             | "max="" <integer> "" "

<integer_attributes> ::= <numerical_optional_attributes>
                       | <integer_optional_attributes>
<real_attributes> ::= <numerical_optional_attributes>
                    | <real_optional_attributes>
<numerical_optional_attributes> ::= ε
                                  | "distribution="u" "
                                  | "distribution="n" "
                                  | "distribution="n" " <mean>
                                  | "distribution="n" " <variance>
                                  | "distribution="n" " <mean> <variance>
<mean> ::= "mean="" <float> "" "
<variance> ::= "variance="" <float> "" "
<integer_optional_attributes> ::= "distribution="i" " <integer_interval>
                                | "distribution="i" " <integer_interval_and_weights>
<real_optional_attributes> ::= "distribution="i" " <real_interval>
                             | "distribution="i" " <real_interval_and_weights>

<integer_interval> ::= "ranges="" <integer_inter_list> "" "
<integer_inter_list> ::= <integer_inter> ";" <integer_inter>
                       | <integer_inter>
<integer_inter> ::= "[" <integer> "," <integer> "] "

<integer_interval_and_weights> ::= "ranges="" <integer_inter> <int_inw> <integer> "]" "
<int_inw> ::= ";" <integer_inter> <int_inw> <integer> ";"
            | "" weights="["

<real_interval> ::= "ranges="" <real_inter_list> "" "
<real_inter_list> ::= <real_inter> ";" <real_inter>
                    | <real_inter>
<real_inter> ::= "[" <float> "," <float> "] "
<real_interval_and_weights> ::= "ranges="" <real_inter> <real_inw> <integer> "]" "
<real_inw> ::= ";" <float_inter> <real_inw> <integer> ";"
             | "" weights="["
```

FIGURE 4.8 – Règles de production des éléments <parameter>.

Les contraintes permettent de raffiner le modèle en contraignant la génération à un sous ensemble ciblé. Elles assurent la cohérence des données générées tout en laissant place à la diversité des instances générées. Cette section débute par une description de la syntaxe des contraintes ainsi que leurs portées. Une seconde partie détaille l'utilisation de contraintes quantifiées.

4.3.1 Syntaxe et portée des contraintes

Les contraintes sont des expressions permettant de spécifier des propriétés sémantiques de manière descriptive. *XML-TAF* permet de définir des contraintes à l'aide d'éléments <constraint> qui viennent s'insérer dans le modèle. En *XML-TAF*, elles sont déclarées au niveau d'un noeud et sont spécifiées en complétant l'attribut expressions des balises <constraint> par une ou plusieurs expressions séparées par des points virgules. La grammaire relative aux contraintes est disponible en fin de section (figure 4.10).

```
1  <?xml version="1.0"?>
2
3  <root name="test_case">
4      <node name="field" nb_instances="1">
5          <parameter name="vegetable" type="string" values="cabbage;leek" weights="5;7"/>
6          <node name="row" min="1" max="40">
7              <parameter name="length" type="real" min="10.0" max="100.0"/>
8              <constraint name="interval_2"
9                          expressions="row[0]\length INFEQ 1.1*row[row.nb_instances - 1]\length;
10                                      row[0]\length SUPEQ 0.9*row[row.nb_instances - 1]\length"/>
11         </node>
12     </node>
13 </root>
```

FIGURE 4.9 – Modèle du cas d'étude simplifié *Oz* avec un exemple de contrainte.

Une expression peut faire intervenir des opérateurs logiques, des opérateurs arithmétiques ou encore des opérateurs de comparaison. Lors de son évaluation, elle renvoie une valeur booléenne. Le rôle du solveur est d'attribuer des valeurs à chaque paramètre de manière à ce que toutes les expressions soient évaluées à vrai. La figure 4.9 présente la contrainte "interval_2" et ses deux expressions. Elle exprime le fait que la longueur de la première rangée (row[0]) vaut +/- 10% de la longueur de la dernière rangée (row[nb_instances - 1]). La syntaxe des expressions est la plus proche possible de la syntaxe du module Z3 pour python (binding avec la bibliothèque originale C++). En particulier, les opérateurs booléens sont préfixés.

Un exemple d'expression simple fait uniquement intervenir un opérateur de comparaison entre deux variables :

$$var1\ operateur\_de\_comparaison\ var2$$

Les opérateurs de comparaison INFEQ et SUPEQ sont utilisés dans la contrainte "interval_2" du modèle figure 4.9. Au total, 6 opérateurs de comparaison diffèrents sont disponibles (table 4.1). La table présente également la notation classique qui

TABLE 4.1 – Opérateurs de comparaison

Dénomination	Notation classique	Notation *TAF*
Opérateur d'égalité	==	EQ
Opérateur d'infériorité stricte	<	INF
Opérateur d'infériorité	<=	INFEQ
Opérateur de supériorité stricte	==	SUP
Opérateur de supériorité	==	SUPEQ
Opérateur de différence	!=	DIF

est utilisée en langage C par exemple. La notation de *TAF* diffère pour des raisons pratiques. En effet, les caractères "<" et ">" sont réservés pour la syntaxe XML.

Les opérateurs arithmétiques ("+", "-", "*", "%") et logiques sont plus conventionnels. Les opérateurs logiques sont au nombre de 3 (NOT, OR, AND). Leur syntaxe est préfixé et l'utilisation de parenthèses est nécessaire car les opérateurs logiques ne sont pas binaire. L'opérateur NOT prend un unique opérande alors que les opérateurs OR et AND prennent au minimum 2 opérandes.

Exemple d'utilisation des différents opérateurs logiques , où p et q sont des expressions :

```
<constraint name="example_not"
            expressions="NOT(p)"/>
<constraint name="example_or"
            expressions="OR(p, q, ...)"/>
<constraint name="example_and"
            expressions="AND(p, q, ...)"/>
<constraint name="example_implies"
            expressions="IMPLIES(p, q)"/>
<constraint name="example_implies_2"
            expressions="OR(NOT(p), q)"/>
```

A l'image du module python Z3, nous avons ajouté un opérateur spécifique pour exprimer les implications (IMPLIES). Cet opérateur n'est pas indispensable, il peut être réécrit à partir des autres opérateurs logiques. Même si les deux notations sont logiquement équivalentes (voir les deux dernières contraintes de l'exemple), il est préférable d'opter pour la première pour une meilleure lisibilité mais surtout pour profiter de potentielles optimisations lors de la résolution par Z3.

Dans la suite de cette section, nous expliquons comment spécifier les opérandes, c'est à dire comment faire référence aux différents paramètres du modèle. La notation choisie s'appuie sur de la structure arborescente, elle consiste en un chemin d'accès relatif à l'emplacement de déclaration de la contrainte.

La syntaxe des chemins d'accès utilise la notation propre aux systèmes *Windows* avec des "\" comme séparateurs. Ce choix a pour but de lever toute ambiguïté avec le symbole de division. De plus, "." réfère au noeud courant et ".." au noeud

parent. Les chemins d'accès peuvent également inclure des indices pour cibler une instance particulière d'un noeud. L'instance zéro est sélectionnée par défaut lorsque aucun indice n'est spécifié. Il est également possible de faire référence à un nombre d'instances (noeud ou paramètre) en post-fixant ".nb_instances" au chemin d'accès (voir exemple juste après).

L'expression de la contrainte "interval_2" (figure 4.9) est rappelée ci-dessous :

```
<constraint name="interval_2"
            expressions="row[0]\length INFEQ 1.1*row[row.nb_instances − 1]\length;
                         row[0]\length SUPEQ 0.9*row[row.nb_instances − 1]\length"/>
```

Ici, "row[0]\length" fait référence à la longueur de la première rangée et "row.nb_instances" au nombre de "row. row[row.nb_instances - 1]\length" fait donc référence à la longueur de la dernière ligne. La prochaine section complète les contraintes par l'ajout de quantificateur.

4.3.2 Contraintes quantifiées

Lorsque l'on souhaite exprimer une contrainte qui se répète avec la multiplicité des éléments concernés ou pour exprimer une relation entre plusieurs éléments d'une structure ordonnée, il faut nécessairement une fonctionnalité permettant d'indexer et de parcourir un ensemble de variables automatiquement. C'est le rôle des quantificateurs. Par exemple : une relation entre la longueur de la rangée i et celle de la rangée $i + 1$. Énumérer manuellement ce type de contrainte serait trop fastidieux et ne permettrait pas de s'adapter aux multiplicités variables.

Pour déclarer une contrainte faisant intervenir des quantificateurs, il faut ajouter les 3 attributs suivants :

— quantifiers : les variables quantifiées sont déclarés dans cet attribut sous forme d'une chaîne de caractère séparée par des points-virgules. Leurs noms sont réduits à une lettre (i,j,k,...), pour faciliter l'analyse syntaxique en enlevant les potentielles ambiguïtés.

— types : un type est associé à chaque quantificateur. Ils peuvent être universels ou existentiels et se déclarent respectivement par les mots clefs "forall" et "exist". Ils doivent être déclarés dans le même ordre que les noms des variables quantifiées et également sous la forme d'une chaîne de caractère séparée par des points-virgules.

— ranges : Dans *TAF*, les quantificateurs fonctionnement uniquement sur des structures finies. A chaque quantificateur doit donc être associé un intervalle. Ils sont notés sous forme d'intervalles fermés $[\![borne\_min, borne\_max]\!]$ dans le même ordre que la déclaration des quantificateurs et sont séparés par des points-virgules.

Terminons par un exemple concret sur notre cas d'étude *Oz*. Pour avoir un champ à la forme cohérente, on souhaite limiter la différence de longueur entre

```
<constraint> ::= "<constraint " <name> <types_quantifiers_and_ranges> <expressions> "/>"
                 | <constraint> <constraint>
                 | ε
<expressions> ::= " expressions="" <expression> """
<expression> ::= <expr>
                  | <expression> ";" <expression>
<expr> ::= "(" <expr> ")"
          | "AND(" <expr> "," <expr_list> ")"
          | "OR(" <expr> "," <expr_list> ")"
          | "NOT(" <expr> ")"
          | "IMPLIES(" <expr> ","<expr> ")"
          | <term> <comparison_operator> <term>
<expr_list> ::= <expr>
                | <expr> "," <expr_list>
<comparison_operator> ::= "SUP"
                          | "SUPEQ"
                          | "INF"
                          | "INFEQ"
                          | "EQ"
                          | "DIF"
<term> ::= <term> "+" <term>
          |  <term> "-" <term>
          | <term> "*" <term>
          | <term> "/" <term>
          | <term> "%" <term>
          | <tree_path>
          | <integer>
          | <float>

<tree_path> ::= <tree_path_beginning> <tree_path_element> <tree_path_ending>
<tree_path_beginning> ::= ε
                          | "./"
                          | "../"
                          | <tree_path_beginning> <tree_path_beginning>

<tree_path_element> ::= <name> "[" <term> "]"
                        | <name>
                        | <tree_path_element> "/" <tree_path_element>
<tree_path_ending> ::= ε
                       | ".nb_instances"
```

FIGURE 4.10 – Règles de production des éléments <constraint>.

deux rangées successives à +/- 10%. Pour cela il faut parcourir toutes les rangées à l'aide d'un quantificateur universel. Cette contrainte est nommée "interval" (L8) dans la figure 4.11. L'intervalle de variation de "i" est défini entre 1 et n (avec n le nombre de rangée) (L12) et la contrainteest sur les longueurs de "row[i]" et "row[i-1]". La gestion des erreurs de *TAF* est assez sommaire pour le moment, les intervalles doivent donc être définis avec soin. Pour avoir un modèle simplifié cohérent, la contrainte "first_track" (L21) a également été ajoutée sur le paramètre de mission "is_first_track_outer". Elle force ce paramètre à vrai si le champ ne comporte qu'une rangée (L22). Du point de vue de la mission, on spécifie ici que le robot commence sa mission sur le bord extérieur du champ et non pas entre deux rangées.

```
1  <?xml version="1.0"?>
2
3  <root name="test_case">
4      <node name="field" nb_instances="1">
5          <parameter name="vegetable" type="string" values="cabbage;leek" weights="5;7"/>
6          <node name="row" min="1" max="40">
7              <parameter name="length" type="real" min="10.0" max="100.0"/>
8              <constraint name="interval" types="forall"
9                          expressions="row[i]\length INFEQ 1.1*row[i-1]\length;
10                                      row[i]\length SUPEQ 0.9*row[i-1]\length"
11                          quantifiers ="i"
12                          ranges="[1, row.nb_instances-1]"/>
13             <constraint name="interval_2"
14                         expressions="row[0]\length INFEQ 1.1*row[row.nb_instances - 1]\length;
15                                     row[0]\length SUPEQ 0.9*row[row.nb_instances - 1]\length"/>
16         </node>
17     </node>
18
19     <node name="mission" nb_instances="1">
20         <parameter name="is_first_track_outer" type="boolean"/>
21         <constraint name="first_track"
22                     expressions="IMPLIES(..\field\row.nb_instances EQ 1, .\is_first_track_outer EQ
       True)"/>
23     </node>
24 </root>
```

FIGURE 4.11 – Modèle contraint du cas d'étude simplifié *Oz*.

Bien que non montré sur cette exemple, il est possible d'imbriquer les quantificateurs universels ("forall") et existentiels ("exist"). Un exemple de modèle est disponible chapitre 6 pour le cas d'étude "tax payer". L'algorithme de traitement des contraintes avec quantificateur est quant à lui donné au chapitre 5.

Pour conclure cette sous-section, la figure 4.12 complète la grammaire relative aux contraintes en y intégrant les quantificateurs. Notons que la règle de production <types_quantifiers_and_ranges> ne respecte pas strictement le format BNF. En effet, il s'agit ici d'exprimer une règle de la forme $a^n b^n c^n, n \geq 1$, ce qui est impossible dans le cadre d'une grammaire sans contexte. Pour des questions de lisibilité, nous avons pris la liberté de simplement ajouter un exposant n pour exprimer cette règle.

```
<types_quantifiers_and_ranges> ::= ε
        | "types="" (<type>";")ⁿ <type> "" quantifiers="" (<letter>";")ⁿ <letter>
          "" ranges=""(<range>";")ⁿ <range> """ <expressions>
<type> ::= "forall"
         | "exist"
<range> ::= "[" <expr> "," <expr>"]"
```

FIGURE 4.12 – Complément au règle de production des contraintes intégrant des quantificateurs.

4.4 Instance complète ou partielle

Cette dernière section se concentre sur le format instance et sur la possibilité d'utiliser un fichier d'instance partielle (voir figure 4.13). Nous donnons en premier lieu une description rapide de ce format, puis quelques exemples applicatifs. Le principe général d'instance partiel est de sélectionner des valeurs de paramètres de manière externe à *TAF* pour guider la génération vers un sous-ensemble souhaité.

4.4.1 Langage de description des cas de tests générés

Les cas de test générés (extension .test_case dans l'outli *TAF*) ont la même structure que le modèle de donnée vu dans les sections précédentes. On y retrouve les <node> et les <parameter> mais sans les contraintes et les attributs spécifiant les types de générateurs. Pour le cas de *Oz*, ce cas de test contient un certain nombre d'éléments "row" ainsi que la valeur des paramètres length associés.

La figure 4.13 donne un exemple d'instance générée à partir du modèle de l'exemple précédent (figure 4.11). Dans cet exemple, un champ de choux (cabbage)(L6) comportant seulement 2 rangées (L7 et L10) a été créé. La structure du modèle est réutilisée mais les attributs des éléments XML sont beaucoup moins nombreux. Les attributs name sont conservés pour identifier les différents éléments. Les <parameter> ont un attribut values stockant la valeur générée. Dans le cas d'un paramètre ayant une multiplicité supérieur à 1, toutes les valeurs sont concaténées dans l'attribut values avec un point virgule comme séparateur. Il n'y a pas de notion d'ordre des éléments en XML. Un attribut instance est ajouté aux <node> pour les ordonner. Il consiste en une chaîne de caractères indiquant l'indice du <node> et le nombre total de <node> du même type. L'indexage utilisé commence à zéro, il correspond à celui d'un tableau dans l'implémentation. Ainsi, sur l'exemple figure 4.13, l'attribut instance est noté "0/1" pour la première rangée et "1/1" pour la deuxième.

Le dernier élément qui diffère avec le fichier du modèle est la balise <seed>. Elle contient uniquement un attribut value qui sert de graine pour pour la génération aléatoire. Elle est choisie aléatoirement avant la génération du cas de test. La réutilisation de cette graine mène exactement à la même génération des paramètres.

```
1  <?xml version="1.0"?>
2
3  <root name="test_case">
4      <seed value ="QldFtnbU4s"/>
5      <node name="field" instance="0/0">
6          <parameter name="vegetable" values="cabbage"/>
7          <node name="row" instance="0/1">
8              <parameter name="length" values="31.75546"/>
9          </node>
10         <node name="row" instance="1/1">
11             <parameter name="length" values="29.25197"/>
12         </node>
13     </node>
14     <node name="mission" instance="0/0">
15         <parameter name="final_track_outer" values="False"/>
16     </node>
17 </root>
```

FIGURE 4.13 – Exemple d'instance (fichier .test_case) pour le modèle simplifié de *Oz*.

4.4.2 Déclaration de cas de test partiellement instancié

Un cas de test partiellement instancié permet de forcer des paramètres ou un nombre d'instances à une valeur spécifique. Cette fonctionnalité vise à réduire l'espace des solutions à un sous-ensemble sans modifier le modèle. Avant chaque génération, *TAF* va d'abord vérifier si un fichier d'instance partielle existe déjà. Si ce n'est pas le cas alors une instance complète va être créée de toute pièce. Si c'est le cas et que la structure du fichier correspond bien au modèle en cours, il va être lu et chaque valeur de paramètre va être redistribuée au bon endroit. *TAF* va ensuite compléter la génération de données manquantes. La déclaration d'instances partielles peut se faire manuellement. Pour faciliter cette tâche, il est possible de tirer profit du fait que les instances générées automatiquement sont au même format que les instances partielles. On peut alors faire une première génération puis modifier le fichier complet généré.

```
1  <?xml version="1.0"?>
2
3  <root name="test_case">
4      <seed value ="customSeed"/>
5      <node name="field" instance="0/0">
6          <node name="row" instance="1/9">
7              <parameter name="length" values="25.00000"/>
8          </node>
9      </node>
10 </root>
```

FIGURE 4.14 – Exemple de cas de test partiellement instancié.

La figure 4.14 montre un exemple de fichier .test_case partiellement instancié. Il y est spécifié un champs de 10 rangées dont la deuxième (1/9, d'indice 1 en partant de zéro) à une longueur de 25 mètres (L7). Les 9 autres rangées, le type de

légume et le paramètre de mission seront générés par *TAF*.

Exemple :

```
<node name="row" instance="1">
    <parameter name="length" values="25.00000"/>
</node>
```

Il est également possible de ne pas spécifier le nombre de rangées. Dans le fragment d'exemple ci-dessus, l'attribut instance vaut 1. Le cas de test généré aura au minimum deux rangées, la deuxième prendra la valeur de 25 mètres. Les cas de test partiellement instanciés doivent respecter les intervalles et les contraintes du modèle pour permettre à la génération de s'effectuer sans encombre. Dans le cas contraire, *TAF* retourne une erreur.

Pour faciliter la création de cas de test partiellement instanciés quelques mots clés sont disponibles. Ils sont spécifiques au type du paramètre auquel il sont associés.

— Pour les paramètres de type string, le mot clé "first" force la valeur à la première valeur définie dans l'énumération. Au contraire "last" force la sélection de la dernière. Le mot clé "wmin" sélectionne le choix possédant la plus petite pondération et "wmax" sélectionne le choix opposé. Dans le cas où plusieurs pondérations sont égales, la première dans l'ordre de déclaration est sélectionnée.

— Pour les paramètres de type numérique (integer ou real), le mot clé "min" forcera le paramètre à prendre la borne inférieure de l'intervalle spécifié dans le modèle. A l'inverse, "max" force la sélection de la borne supérieure de l'intervalle. Dans le cas où le paramètre est de type real, le mot clé "mean" permet de forcer la valeur moyenne.

Enfin, considérons le cas d'un paramètre de multiplicité supérieur à 1. La syntaxe est similaire à celle de la déclaration dans le modèle ("template"). Prenons par exemple un paramètre de type integer de multiplicité 5 appelé "integer_m5". L'exemple ci-dessous montre une possible instanciation partielle de ce paramètre. La première valeur est forcée au minimum et la quatrième au maximum. La deuxième est forcée à 8, la troisième et cinquième (∅) sont laissées libres au générateur. Autrement dit, elles sont produites aléatoirement selon le générateur par défaut.

Exemple :

```
<parameter name="integer_m5" values="min;8;;max;r"/>
```

Il est important de remarquer que la création des cas de test partiellement instanciés peut facilement être automatisée. La fonctionnalité est particulièrement

adaptée pour integrer *TAF* dans une suite de test. Un programme peut par exemple générer des cas de test partiellement instanciés en suivant une stratégie de test donnée (par exemple du *pairwise testing*) puis utiliser *TAF* pour les compléter. L'unique condition est de respecter le langage spécifique de *TAF*.

4.5 Conclusion

Ce chapitre décrit le langage spécifique *XML-TAF* : . C'est un langage purement déclaratif permettant de modéliser des données structurées ainsi que leurs contraintes. Le langage est basé sur XML ce qui assure une manipulation facile coté utilisateur et coté machine. De plus, XML est un format standard pour l'échange et la sauvegarde de données. Les autres forces de *XML-TAF* sont la spécification simple de générateurs pour chaque type de paramètre et les expressions de contrainte avec des variables quantifiées. Les contraintes ont une syntaxe permettant de les exporter facilement vers le solveur Z3. Globalement, le langage de modélisation *XML-TAF* est conçu pour être le plus simple possible tout en garantissant une bonne expressivité (exemple au chapitre 6). Ce langage nous a permis de prototyper rapidement *TAF* pour expérimenter notre approche de génération combinant sélection aléatoire et résolution de contrainte. Il contient un ensemble suffisant de concepts pour l'expression des modèles qui nous intéressent : des relations de composition pour structurer les données, des types de base avec des fonctions de génération par défaut, la notion d'instance et de multiplicité, et enfin des contraintes sémantiques qui peuvent impliquer des expressions numériques.

Néanmoins, pour les futurs développement de *TAF* plusieurs axes d'améliorations sont à considérer. Par exemple, il n'existe pas de notation graphique pour nos modèles. Il est aisé de faire un modèle équivalent en UML à partir d'un modèle *TAF*, mais l'opération inverse et plus délicate. En effet, *TAF* n'est pour le moment pas compatible avec cenrtains concepts fondamentaux du paradigme orienté objet. Plus précisément, il n'est pas possible de représenter directement les concepts d'héritage et d'association. Il existe tout de même des solutions permettant de contourner ces problèmes. Dans le cas de l'héritage, on peut multiplier les éléments dans le modèle. Cela à pour effet d'augmenter la taille du modèle car il n'y a plus de classe qui factorise les éléments communs. Cette solution est simple et suffisante dans la plupart des cas. Dans le cas du concept d'association, il est possible de l'exprimer avec des contraintes. Cela peut s'avérer lourd. L'expression de certaines contraintes pourrait aussi être simplifiée par l'ajout de l'opérateur "self" dans les chemins d'accès aux variables.

A l'issue de ce travail sur la spécification des modèles, on peut déjà entrevoir les améliorations à apporter à notre langage. Toutefois, la version actuelle de *TAF* est pleinement opérationnelle pour de nombreux cas. Une démonstration approfondie est proposée au chapitre 6. Le chapitre suivant décrit comment le modèle est traité ainsi que les algorithmes clefs pour la résolution des contraintes tout en maintenant la diversité des solutions.

Chapitre 5

Concepts et algorithmes centraux de *TAF*

Sommaire

5.1 Introduction

Pour pouvoir générer des données structurées complexes, *TAF* repose sur l'utilisation intensive d'un solveur SMT (Z3). L'utilisation directe de ce type de solveur pose deux problèmes. D'abord, les solveurs fonctionnent à partir de modèles statiques, c'est à dire avec une structure qui ne varie pas pendant la résolution. En effet, un changement dans la structure implique une évolution dynamique des contraintes à chaque création ou suppression d'un élément des données. Pour la suite de ce chapitre, on appellera structure variable, une structure dont le nombre et le type d'éléments n'est pas fixe. Le deuxième problème concerne la diversité des solutions retournées par le solveur. Un solveur retourne en général la première solution qu'il trouve, celle-ci elle est bien souvent triviale (par exemple toutes les variables à 0). Ces limitations rendent les solutions des solveurs insuffisantes pour le test. Les outils de génération pour le test ne correspondent pas à nos besoins où ne passent pas à l'échelle. Le testeur doit alors développer un générateur spécifique pour son système sous test. A l'inverse, nous proposons une méthode de génération à partir d'un modèle de données purement déclaratif. Cette approche est basée sur le découpage d'un problème en sous problèmes pour permettre la génération de structures variables. A chaque sous problème nous ajoutons des contraintes qui injectent des valeurs aléatoires pour forcer de la diversité dans les solutions. L'introduction de

structures variables requiert une mise à jour permanente des contraintes et l'utilisation de quantificateurs pour le parcours des structures concernées. Cette approche est implémentée par le prototype *TAF* et ce chapitre en décrit les algorithmes clefs.

La première section (section 5.2) dévoile le coeur de notre approche avec les algorithmes de découpage en sous-problèmes et l'ajout de diversité. Ensuite la gestion des contraintes est détaillée dans la section 5.3. Pour finir, la dernière section (section 5.4) explique une approche concurrente dans un but comparatif. À notre connaissance cette approche est la seule à avoir les caractéristiques que nous désirons : purement déclarative, structure variable, multiplicité des solutions retournées et passage à l'échelle.

5.2 Algorithmes de génération

Cette section détaille les algorithmes centraux au processus de génération avec entre autres le découpage en sous problèmes et l'hybridation du solveur avec de l'échantillonnage aléatoire.

5.2.1 Principe de découpage du modèle

La génération adopte une approche descendante et stratifiée qui suit la structure du modèle XML. L'analyse syntaxique du modèle permet de construire une structure arborescente. Elle contient des espaces réservés pour les différents éléments définis dans le modèle en respectant leur hiérarchie. L'arbre initial ainsi créé est mis à jour au fur et à mesure de la génération en y insérant les instances d'éléments créés et en attribuant des valeurs à leurs paramètres. L'exemple simplifié du robot agricole utilisé au chapitre 4 est réutilisé dans ce chapitre. La structure arborescente des données (figure 5.1) a un noeud "test_case" pour racine. Ce noeud a pour fils un noeud "field" qui a lui même pour fils un noeud "row". Comme un champ est rarement composé d'une seule rangée, l'élément "row" peut être dupliqué et inséré aux fils du noeud "field".

La figure 5.1 illustre l'approche descendante sur l'exemple *Oz* en montrant le découpage de la structure en différents niveaux de profondeur. Le noeud racine ("test_case") se trouve à la profondeur 0 de l'arbre (P0), ses noeuds fils "mission" et "field" sont à la profondeur 1 (P1), et ainsi de suite. Pour la génération, le principe est de générer tous les éléments d'une profondeur donnée p dans une même étape. Ils sont générés après les éléments de profondeur $d-1$ et avant ceux de profondeur $d+1$.

Le nombre d'instances à créer à chaque couche est un paramètre $<element\_name>.nb\_instances$ qui appartient à la couche immédiatement supérieure, celle du noeud parent. Cela garantit que le nombre d'instances est toujours défini avant de construire les objets correspondants. Si le modèle laisse ce paramètre implicite, il prend la valeur par défaut 1. Dans l'exemple en cours, le nombre d'instances de missions (element "mission") et de champs (element "field") est fixé à 1,

d'où la construction d'une instance de chacune d'entre elles. En revanche, le modèle spécifie un nombre quelconque de rangées (element "row") dans un intervalle $[\![min, max]\!]$ donné. Le nombre d'instances sera determiné lors du traitement de la profondeur 1, lorsque le noeud parent "field" sera traité.

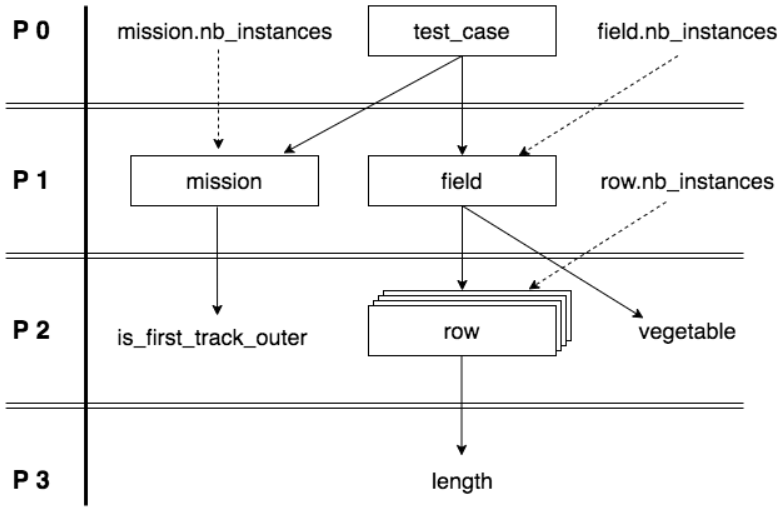


FIGURE 5.1 – Découpage en couche pour la génération.

À chaque contrainte définie dans le modèle est également attribuée une profondeur : elle prend la profondeur maximale de tous les paramètres auxquels elle se réfère. Dans notre exemple *Oz*, il existe une contrainte entre $row.nb\_instances$ et $is\_first\_track\_outer$ qui assure que le robot est bien paramétré dans le cas où il y aurait une seule rangée de légumes. En effet, si le champ ne contient qu'une rangée, alors la mission commence nécessairement sur une zone de travail extérieure du champ ($is\_first\_track\_outer = True$). Les paramètres intervenant dans cette contrainte sont respectivement de profondeur 1 et 2. Selon la règle précédente, il s'agit donc d'une contrainte de profondeur 2. Cela signifie que $row.nb\_instances$ sera généré en premier, et que par la suite la génération tentera de trouver une valeur de $is\_first\_track\_outer$ qui satisfasse la contrainte. Cette approche peut évidemment nous conduire à une impasse car les paramètres générés à partir des couches précédentes peuvent produire des contraintes insatisfiable. C'est pourquoi la génération utilise un système de backtracking.

5.2.2 Algorithme de génération par couche

L'algorithme 1 montre la génération par couches avec backtracking. Il analyse d'abord le modèle, crée le squelette du code d'exportation et prépare l'arborescence initiale avec les espaces reservés et les contraintes attachées à la profondeur appropriée (L1 à L3). La boucle principale (L6 à L18) itère jusqu'à ce que la génération soit terminée ou que le budget maximal pour le backtracking soit atteint (MAX_B). À chaque itération, la fonction $generate\_depth()$ tente la génération des éléments à la profondeur actuelle. Si elle réussit, la génération se poursuit à la

profondeur suivante (L8). Si elle échoue, les choix effectués à la profondeur actuelle sont annulés et la génération retourne à la couche précédente (L11 à L13). Après avoir quitté la boucle, l'outil rend compte de son échec (L20) ou de sa réussite (L23). Dans ce dernier cas, une version XML du scénario de test est créée et sauvegardée au format ".test_case" décrit chapitre 4.4. C'est à partir de ces données que peuvent ensuite être créées des entrées de tests propres à l'application ciblée, par des fonctions d'exportation spécifiques.

Algorithm 1 main

Require: *path*
1: $tree \leftarrow parse\_template(path)$
2: $create\_export\_code\_skeleton\_file(tree)$
3: $preprocess(tree)$
4: $counter \leftarrow 0$
5: $depth \leftarrow 0$
6: **while** $counter < MAX\_B\ AND\ depth \leq tree.max\_depth()$ **do**
7: **if** $generate\_depth(tree, depth)$ **then**
8: $depth \leftarrow depth + 1$
9: **else**
10: **if** $depth > 0$ **then**
11: $counter \leftarrow counter + 1$
12: $depth \leftarrow depth - 1$
13: $tree.reset(depth)$
14: **else**
15: **return** $False$
16: **end if**
17: **end if**
18: **end while**
19: **if** $counter == MAX\_B$ **then**
20: **return** $False$
21: **end if**
22: $create\_XML\_Test\_case\_file(tree)$
23: **return** $True$

L'algorithme 2 décrit comment une couche donnée est générée. Tout d'abord, il extrait tous les paramètres p et toutes les contraintes c de la profondeur courante. Ensuite, les paramètres sont triés en deux groupes, les paramètres dépendants et ceux indépendants. Pour cela, on regarde pour chaque paramètre s'il est impliqué ou non dans l'une des contraintes (L3 à L4). Les paramètres indépendants sont simplement générés à l'aide des générateurs aléatoires par défaut (L5 à L7) décrit chapitre 4.2. S'il n'y a pas de contrainte, la génération de la couche est terminée (L8 à L10). Sinon, un objet *problem* est créé avec tous les paramètres dépendants et les contraintes (L12). Si le problème échoue au test de satisfiabilité dans Z3, la couche n'est pas générée et la fonction renvoie $False$ (L14). Cela entraîne un retour en arrière dans l'algorithme principal. En revanche, si le problème passe

le test de satisfiabilité avec succès, les paramètres dépendants sont traités (L16). Leur génération mélange la résolution de contraintes et l'échantillonnage aléatoire afin d'assurer la diversité des solutions. Ce processus est décrit dans l'algorithme 3 présenté juste après. Pour finir, avant de renvoyer le succès de la génération, l'algorithme prépare la couche suivante en fonction des valeurs nouvellement générées des paramètres $<\textit{élément\_nom}>.\textit{nb\_instances}$ (L9 ou L17). Le nombre approprié d'instances d'éléments est alors créé et inséré dans l'arbre.

Algorithm 2 generate_depth

Require: $tree, depth$

1: $p \leftarrow extract\_parameters(tree, depth)$
2: $c \leftarrow extract\_constraints(tree, depth)$
3: $dependent\_p \leftarrow filter\_dp(p, c)$
4: $independent\_p \leftarrow filter\_indp(p, c)$
5: **for** p' *in* $independent\_p$ **do**
6: $\quad p'.default\_generate()$
7: **end for**
8: **if** c *is empty* **then**
9: $\quad create\_next\_depth\_instances(p)$
10: $\quad$ **return** $True$
11: **end if**
12: $problem \leftarrow init\_problem(dependent\_p, c)$
13: **if** $NOT\_problem.check\_sat()$ **then**
14: $\quad$ **return** $False$
15: **else**
16: $\quad solve\_with\_diversity(problem)$
17: $\quad create\_next\_depth\_instances(p)$
18: $\quad$ **return** $True$
19: **end if**

5.2.3 Algorithme de génération avec ajout de diversité

L'algorithme 3 prend un objet *"problem"* comme entrée et le résout avec ajout de diversité. Nous savons que le problème est satisfiable (d'après le test dans l'algorithme 2 en L13) mais nous voulons forcer le solveur à produire une solution différente de celle qu'il retournerait normalement. Pour cela, le principe est d'attribuer des valeurs aléatoires à autant de paramètres que possible avant de résoudre les contraintes. Dans la boucle interne (L5 à L12) nous continuons à ajouter de nouvelles contraintes de forme $p = v$ jusqu'à ce que le problème devienne insatisfaisable. Cela donne une nouvelle version du problème qui peut être résolue, dans laquelle certains paramètres prennent des valeurs aléatoires. Le processus est répété plusieurs fois selon un budget de diversité MAX_D (boucle externe L3). À chaque itération, nous gardons une trace du meilleur problème c'est-à-dire celui avec le plus grand nombre de paramètres choisis au hasard (L14). Enfin, ce problème est résolu

et tous les paramètres de la structure sont mis à jour en conséquence (L17 et L18).

Algorithm 3 solve_with_diversity

```
Require: problem
 1: current_best ← problem
 2: counter ← 0
 3: while counter < MAX_D do
 4:    new_problem ← copy(problem)
 5:    while new_problem.check_sat() do
 6:       p ← select_parameter(problem)
 7:       p.default_generate()
 8:       v ← p.value
 9:       p.reset()
10:       c ← build_constraint(p, v)
11:       new_problem.add_constraint(c)
12:    end while
13:    remove_last_constraint(new_problem)
14:    current_best ← choose_best(current_best, new_problem)
15:    counter ← counter + 1
16: end while
17: current_best.solve()
18: current_best.update_parameters()
```

TAF implémente les algorithmes de génération de ce chapitre en Python. Dans un but de simplification, la présentation s'est concentrée sur la génération basée sur des modèles, en omettant la fonctionnalité d'instances partiellement instanciées. En pratique, ce cas n'a d'incidence que sur le pré-traitement de la structure arborescente dans l'algorithme 1. Il ajoute de nouvelles contraintes pour forcer les valeurs des paramètres souhaités, puis la génération par couches se déroule comme décrit dans ce chapitre. L'utilisateur peut également demander la génération de n cas de test, plutôt qu'un seul. Les budgets de backtracking et de diversité (MAX_B et MAX_D) peuvent être facilement configurés, si leur valeur par défaut est insuffisante au bon fonctionnement de notre approche de génération sur un problème donnée.

Il est également important de remarquer que notre approche suppose un modèle dont les contraintes sont correctement agencées. En effet, il est possible de créer des modèles qui sont très difficiles voir impossibles à résoudre par *TAF* en distribuant des contraintes sur plusieurs couches de manière à épuiser le budget de backtracking. Dans la pratique, nous n'avons pas rencontré de cas d'étude où ce type de modélisation est inévitable. Dans la liste des futures fonctionnalités figure un détecteur propre à cette situation.

5.3 Gestion des contraintes

Cette section se concentre sur la gestion des contraintes avec notamment l'usage de quantificateurs. L'implémentation correspondante dans *TAF* conclut cette section.

5.3.1 Principe général de résolution

Replaçons d'abord la gestion des contraintes dans le contexte de la méthode génération détaillée précédemment (section 5.2). Les contraintes sont attachées à des noeuds dans le modèle et sont définies dans un attribut *expressions* (voir chapitre 4). Elles sont utilisées et traitées lors du traitement des couches (figure 2). En particulier, les contraintes qui correspondent aux variables de la couche en cours sont extraites (L2) et une mise à jour des variables qui les composent est effectuée lors de l'initialisation du problème (L12).

Les contraintes sont exprimées dans un format le plus proche possible du binding Z3 et python (voir chapitre 4). Les quelques différences sont forcées par les restrictions du langage XML. Une étape de mise en forme est donc nécessaire avant l'envoi des contraintes vers le simulateur : c'est le rôle de la fonction *"buildZ3query"*.

Une autre différence est l'ajout de quantificateurs. L'imbrication des quantificateurs est nécessaire pour parcourir les structures de données et exprimer des propriétés reliant des éléments différents. Pour faciliter l'analyse syntaxique (*parsing*) des contraintes et l'implémentation de l'algorithme pour déplier les quantificateurs, nous avons choisi de sortir les quantificateurs des formules. Plus spécifiquement dans le modèle *TAF*, une contrainte est représentée à l'aide d'un noeud et des attributs *expressions* pour les expressions, *quantifiers* pour le nom des variables sur lesquelles sont appliqués les quantificateurs, *types* pour spécifier leur type (universel ou existentiel) et *ranges* pour leur domaine de définition. Cette séparation en plusieurs champs exclut la possibilité d'exprimer des quantificateurs dans des sous-formules. On se limite donc aux expressions avec quantificateurs externes. Par exemple :

$$\forall x \in [\![a, b]\!], \exists y \in [\![x, b]\!], P(x, y)$$

Comme le montre l'exemple précèdent, le domaine de définition d'une variable peut dépendre du contexte des quantificateurs qui précèdent. Le domaine de définition de la variable y dépend de la valeur de la variable x. En conséquence, la traduction des formules vers leur forme sans quantificateur, doit suivre l'ordre des quantificateurs, de la gauche vers la droite (autrement dit, en partant du quantificateur le plus externe). Un exemple de ce procédé est donné ci-dessous en prenant $a = 0$ et $b = 2$:

Un vision plus complète et plus formelle du depliement des quantificateurs est donnée figure 5.3. Lorsque qu'il y a plusieurs variables quantifiées le procédé est répété récursivement jusqu'à la mise à plat complète de l'expression. Notons que,

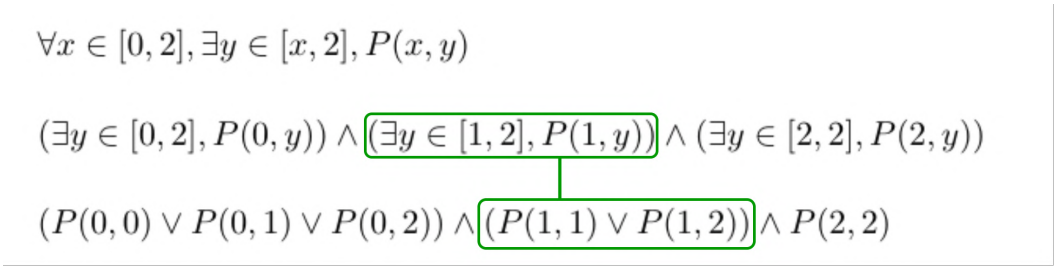


FIGURE 5.2 – Exemple de dépliage des quantificateurs.

les domaines de valeurs des variables liées sont entiers et connus lorsque le quantificateur est déplié. En effet, les expressions de *min* et *max* ne doivent dépendre que de paramètres choisis dans les couches supérieures à celle attribuée à la contrainte. Si tel n'est pas le cas, *TAF* retourne un message d'erreur.

$Process(P)$	$=$	$buildZ3query()$	Si P non quantifiée.
$Process(\forall x \in [min, max], P)$	$=$	$\begin{cases} ERROR \\ FALSE \\ \bigwedge_{i=val(min)}^{val(max)} Process([x := i]P) \end{cases}$	Si $val(min)$ ou $val(max)$ indéfinis, Si $val(min) > val(max)$, Si $val(min) \leq val(max)$.
$Process(\exists x \in [min, max], P)$	$=$	$\begin{cases} ERROR \\ FALSE \\ \bigvee_{i=val(min)}^{val(max)} Process([x := i]P) \end{cases}$	Si $val(min)$ ou $val(max)$ indéfinis, Si $val(min) > val(max)$, Si $val(min) \leq val(max)$.

FIGURE 5.3 – Principe de l'implémentation avec Z3.

Nous allons maintenant décrire comment nous avons implémenté ce traitement.

5.3.2 Algorithmes

Pour simplifier la description de l'algorithme, nous décrirons une version dans laquelle une contrainte ne contient qu'une seule expression. Il est en réalité possible de définir plusieurs expressions (séparées par des points virgules dans une même déclaration de contrainte). Le principe est de pouvoir factoriser la déclaration des quantificateurs et de leurs intervalles quand ils sont communs à plusieurs expressions. Les expressions sont alors traitées indépendamment lors de la phase de mise à plat des quantificateurs.

Par exemple : $\forall i, \exists j\ (P_{i,j}; Q_{i,j})$
sera interprété comme : $\forall i, \exists j\ (P_{i,j});\ \forall i \exists j\ (Q_{i,j})$

On prend pour point de départ une classe "contrainte" avec pour attribut un objet qui contient l'expression (de type "expression") et les quantificateurs externes sous la forme d'une liste d'objets de type "quantifier". Le type "quantifier" est une structure de données pure qui ne contient que 4 attributs au format chaîne de caractère ("name", "min", "max", "type"). Pour finir, les utilitaires suivants sont

disponibles dans l'environnement de la classe "contrainte" :

— **int <− str2int(string)**, prend une expression sous forme de chaîne de caractère et retourne l'entier correspondant. Retourne une erreur si la traduction n'est pas possible. Elle correspond à la fonction *val* de la figure 5.3.

— **expr <− substitute(expr, name, string_value)**, prend une expression et produit une nouvelle expression où toutes les occurrences de la variable name sont remplacées par leur valeur au format chaine de caractère (string_value). Elle permet de réaliser les substitutions $[x := i]P$ montrées dans la figure 5.3.

— **quantifier_list <− update_range_quantifier(quantifier_list, name, string_value)**, prend une liste de quantificateurs et met à jour tous les intervalles faisant intervenir la variable name en utilisant la fonction substitute().

— **expr <− evaluate(expr)**, prend une expression et retourne une expression dans laquelle toutes les variables connues (car choisies lors du traitement des couches supérieures) sont remplacées par leur valeur.

— **expr <− build_Z3_query(expr)**, prend une expression et retourne une version mise en forme pour Z3.

La classe "contrainte" a une méthode "process()" qui va appeler une méthode récursive "process_quantified()" uniquement s'il y a des quantificateurs (algorithme 4).

Algorithm 4 process

```
1: expr ← ""
2: if self.quantifier_list is empty then
3:    expr ← self.expression
4: else
5:    expr ← process_quantified(self.quantifier_list, self_expression)
6: end if
7: expr ← evaluate_expression(expr)
8: return build_Z3_query(expr)
```

L'algorithme permettant de déplier les expressions quantifiées est présenté dans l'algorithme 5. Il est divisé en quatre parties. D'abord la déclaration des variables locales (L1 à L7). On y voit par exemple, "first" qui est de type quantifier et "remainder" qui est une liste de quantifier. Ces deux variables servent au parcours des quantificateurs. La deuxième partie (L9 à L11) extrait le premier quantificateur de la liste puis en évalue les bornes. La troisième partie (L13 à L29) gère la récursion avec l'initialisation et la condition de fin. Pour finir, la quatrième partie (L31 à L46) concerne le dépliage de l'expression pour toutes les valeurs du quantificateur. On y trouve l'appel récursif (L35).

L'unicité ($\exists!x$) est un concept qui peut être très utile dans la modélisation. Il est tout à fait possible de l'intégrer comme un troisième type de quantificateur avec une fonction spécifique à son dépliage. La fonctionnalité a été retirée de *TAF* lors de la

Algorithm 5 process_quantified

Require: *q_list*, *expression*

quantifier first
quantifier_list remainder
string q
int min, *max*
boolean multiple
expression expr ← ""
expression sub_expr ← ""

split q_list into first and remainder
min ← *str2int*(*evaluate*(*first.min*))
max ← *str2int*(*evaluate*(*first.max*))

if *min* > *max* **then**
  if *first.type* == "*forall*" **then**
    return *True*
  else if *first.type* == "*exists*" **then**
    return *False*
  end if
end if
if *min* = *max* **then**
  multiple ← *False*
else
  multiple ← *True*
  if *first.type* == "*forall*" **then**
    expr ← "*AND*("
  else if *first.type* == "*exists*" **then**
    expr ← "*OR*("
  end if
end if

for *q* ← *min to max* **do**
  sub_expr ← *substitute*(*expression*, *first.name*, *q*)
  if *remainder is not empty* **then**
    remainder ← *update_quantifier_range*(*remainder*, *first.name*, *q*))
    sub_expr ← *process_quantified*(*remainder*, *sub_expr*)
  end if
  if *multiple* **and** *q* < *max* **then**
    expr ← *expr* + *sub_expr* + ", "
  end if
  if *multiple* **and** *q* == *max* **then**
    expr ← *expr* + *sub_expr* + ")"
  end if
  if *not multiple* **then**
    expr ← *sub_expr*
  end if
end for
return *expr*

mise au propre du code mais figure sur la liste des prochains ajouts. La réécriture à l'aide de quantificateurs universels et existentiels requiert l'utilisation de deux variables quantifiées dont une qui est définie dans une sous-formule. Quelques manipulations algébriques sont nécessaires pour pouvoir traiter le dépliage de ce cas particulier. La marche à suivre est donnée ci-dessous puis dans un exemple concret figure 5.4.

$\exists! i \in [\![m, n]\!], P(i)$
$\exists i \in [m, n], (P(i) \wedge (\forall j \in [\![m, n]\!], P(j) \implies j = i))$
$\bigvee_{i=m}^{n} (\overline{P(i)} \bigwedge_{j=m}^{n, j \neq i} P(j))$

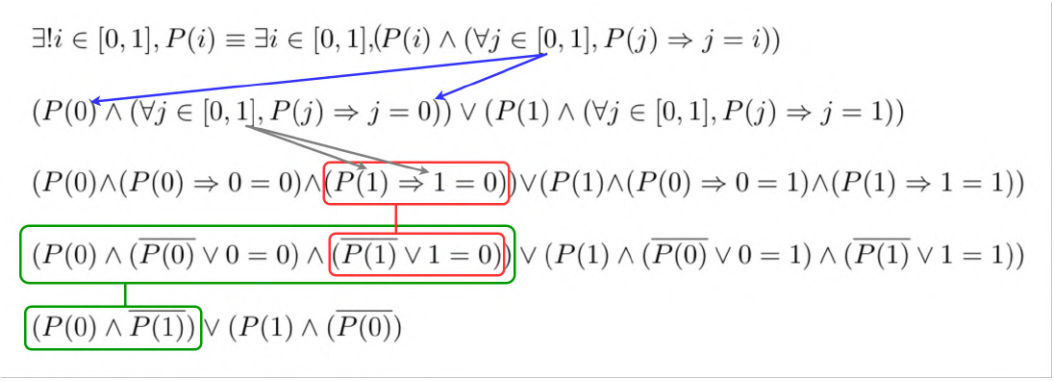


FIGURE 5.4 – Méthode de dépliage du cas "il existe un unique"

5.4 Approche concurrente : *PLEDGE*

Pour finir ce chapitre, l'approche concurrente la plus proche de nos travaux à notre connaissance est décrite. Cette approche est une solution hybride entre un solveur et une heuristique. Le solveur utilisé est le même que pour *TAF* (c'est-à-dire Z3). Cette approche répond au même cahier des charges que *TAF* : Elle prend en entrée un modèle purement déclaratif (basé UML / OCL), elle permet de gérer les structures variables et enfin elle propose des solutions différentes satisfaisant les contraintes du modèle. Cette approche est implémentée dans l'outil *PLEDGE* [Soltana 2020].

L'approche de *PLEDGE* repose sur l'utilistation d'un solveur SMT et d'une métaheuristique de manière à tirer profit des avantages de chaque méthode. Le problème est découpé puis partiellement résolu par le solveur ou l'heuristique. La diversité des résultats dépend ici intégralement de l'utilisation de la métaheurisque, ainsi un bon compromis doit être trouvé pour le découpage du problème et sa distribution vers les deux techniques de résolution.

La métaheuristique choisi est AVM (**A**lternating **V**ariable **M**ethod) [Kempka 2015]. Son fonctionnement est similaire à la méthode plus

classique de la descente de gradient mais pour des variables discrètes. Plus précisément, la méthode débute à partir d'une solution arbitraire puis tente de l'améliorer de manière incrémentale. L'algorithme s'arrête lorsque plus aucune amélioration ne peut être trouvée. Il n 'y a donc pas de garantie de trouver une solution, on peut se retrouver piégé dans l'équivalent discret d'un maximum local. Dans le contexte de *PLEDGE*, AVM débute avec des données aléatoires et probablement invalides. L'algorithme va ensuite le faire évoluer vers une solution respectant les contraintes OCL. Pour savoir si une nouvelle variation est meilleure que la précédente, il faut définir une fonction objectif (*fitness*). Cette fonction est décrite dans Ali et al [Ali 2013]. Elle effectue une évaluation quantitative de la distance qui sépare les données de la satisfaction des contraintes OCL. Les données à faire évoluer sont représentées sous forme d'un vecteur de n variable $V = (v_1, v_2, ..., v_n)$ où chaque variable peut représenter un objet (une instance de classe), une association ou encore des attributs. L'algorithme peut ajouter ou retirer des éléments de V, la taille du vecteur n'est pas fixe. Pour l'optimisation du vecteur V, les variables sont sélectionnées séquentiellement. Pour chaque variable, une étape d'exploration est effectuée selon un schéma IPS (**I**terated **P**attern **S**earch) [Kempka 2015]. De petites variations sont opérées pour cibler la direction la plus prometteuse. Cette direction est ensuite explorée jusqu'à trouver un optimum. Dans le cas ou la notion de direction n'a pas de sens, la méthode IPS ne peut pas être utilisée. En remplacement, dix possibilités choisies aléatoirement sont explorées. Une fois que toutes les variables du vecteur ont été traitées, on considère une itération terminée. On repart alors à la première variable, on multiplie les itérations jusqu'à atteindre une solution ou la condition d'arrêt (nombre maximal d'itérations). Pour éviter d'être piégé dans un maximal local, lorsqu'une itération n'aboutit à aucune amélioration et ne satisfait pas les contraintes, le vecteur est réinitialisé aléatoirement.

Avant de répartir la résolution entre AVM et le solveur SMT, le modèle est d'abord mis sous forme normale négative NNF (**N**egation **N**ormal **F**orm). Dans ce format l'opérateur de négation est appliqué uniquement aux variables booléennes et les seuls opérateurs booléens autorisés sont le "et" et le "ou" logique. Les sous formules faisant intervenir des contraintes structurelles (ajout et suppression d'objet) sont distribuées pour une résolution par méta-heuristique. Au contraire, les sous formules faisant uniquement intervenir des types primitifs sont envoyées au solveur. Ce découpage est réalisé de manière déterministe à partir de l'arbre de la syntaxe abstraite (AST, **A**bstract **S**yntax **T**ree). Cette procédure associe un label à chaque noeud pour en spécifier le traitement. Cette étape est complexe, elle fait du parcours en profondeur et repose sur un ensemble de règles ad hoc. Un exemple est donné figure 5.5

Sur cet exemple, nous nous contentons de donner une vue intuitive du traitement de la contrainte, sans en présenter les détails. Il est à noter que certains cas nécessitent à la fois l'utilisation du solveur (label SMT) et de AVM (label search). Ces cas sont signalés par des labels "both" sur la figure. En effet, en général il n'est pas possible de complètement partitionner les sous formules, il y a des inter-

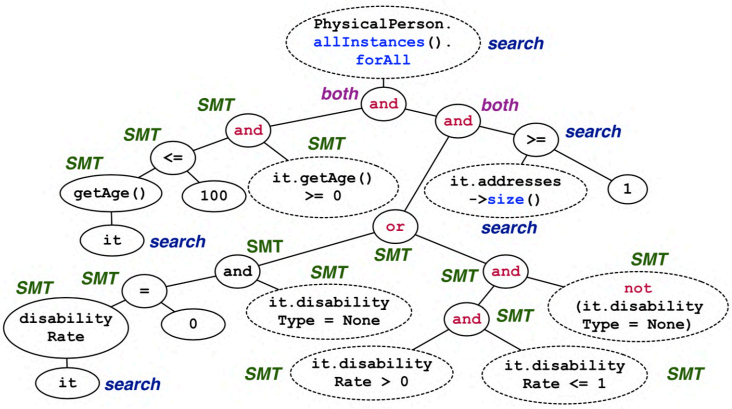


FIGURE 5.5 – Exemple de découpage dans *PLEDGE* (source : [Soltana 2020]).

sections. Une instance partagée est alors traitée conjointement par le solveur et la méta-heuristique AVM. Nous verrons qu'en pratique, cette méthode n'offre que peu de diversité des données générées et a des performances inférieures à la stratégie plus simple que nous avons définie (chapitre 6).

5.5 Conclusion

Ce chapitre présente les algorithmes centraux de notre méthode de génération de données structurées diverses avec le découpage en couches, la gestion d'expressions de contrainte avec et sans quantificateur, et leur résolution avec ajout de diversité. Il se conclut par une brève présentation d'une méthode concurrente.

Le principal intérêt de la découpe en couches de notre approche est de pouvoir gérer des éléments avec un nombre d'instances non fixé, et de manière relativement simple. À cet effet, les noeuds et leurs paramètres sont générés couche par couche de manière descendante, ce qui assure que le nombre d'instances d'un élément soit déterminé au moment de sa génération. De plus, lors de la génération de chaque couche un algorithme force la diversité en ajoutant des contraintes aléatoires.

Notre méthode repose sur de nombreux appels au solveur. Le nombre d'appels est réglable par l'intermédiaire des budgets de backtracking et de diversité (MAX_B et MAX_D). Ces deux degrés de liberté permettent d'adapter l'algorithme à la difficulté du problème et au temps disponible pour la génération des tests. On peut par exemple donner un budget de diversité plus grand sur un problème simple pour forcer le plus d'échantillonnages aléatoires possibles. L'efficacité est grandement liée à la performance du solveur. Le prochain chapitre propose une évaluation de notre approche sur 4 cas d'étude venant de domaines d'applications différents et une comparaison avec *PLEDGE*.

Chapitre 6

Cas d'étude

Sommaire

6.1 Introduction

Ce chapitre montre l'utilisation de *TAF* sur 4 cas d'étude venant de différents domaines d'applications :

— **Oz** : champ de légumes et ses paramètres de mission correspondants pour le robot agricole *Oz* ;

— **Tax payer** : population d'individus avec des sources de revenus différents pour le test d'un logiciel de gestion d'imposition au Luxembourg ;

— **Bitmap** : image carrée au format bitmap représentant un dégradé en niveau de gris ;

— **Tree** : structure arborescente de noeuds indexés.

Ces exemples nous permettent de considérer différents types de propriétés structurelles et sémantiques. Notamment, les contraintes considérées vont de simples propriétés booléennes à des propriétés plus complexes, nécessitant des quantificateurs

imbriqués et des calculs arithmétiques. Nous évaluons *TAF* selon deux critères : sa performance et la diversité des données générées.

La performance de *TAF* est mesurée par le temps d'exécution nécessaire pour générer 100 cas de test valides pour chacun des quatre cas d'étude. La génération est répétée 10 fois et les temps médians, min et max sont rapportés. La diversité des données est analysée en termes de couverture de l'espace de données. Son évaluation se base sur un ensemble de cas à couvrir que nous avons manuellement dérivés pour chacun des cas d'étude. Pour cela, nous avons pris en compte à la fois les domaines de définition des paramètres et les aspects spécifiques à l'application liées à la sémantique des données. Nous relevons le pourcentage cumulé de cas couverts et identifions les cas manquants à la fin de la génération.

Nous comparons également *TAF* à des approches concurrentes. Parmi celles existant dans la littérature pour traiter des données structurées avec des contraintes (voir la section 1.6), nous pouvons identifier trois grandes catégories :

— Des approches basées sur de multiples essais successifs, selon un algorithme de type "générer-puis-filtrer". Des exemple typiques sont *QuickCheck* [Claessen 2000] et *Yagg* [Coppit 2005]. Dans *QuickCheck*, on cherche à tester des propriétés : les données sont générées aléatoirement, et on ne retient que celles qui satisfont les préconditions des propriétés ciblées. *Yagg* est un outil de génération basé sur des grammaires attribuées : les données sont produites à partir de la grammaire hors contexte, puis elles sont vérifiées selon les conditions dépendantes du contexte.

— Des approches nécessitant un réglage fin du processus de génération. Ce réglage peut prend la forme de code utilisateur attaché au modèle de données, comme dans le fuzzer *Peach* [PEACH 2020], ou dans diverses approches basées sur des grammaires formelles ([Maurer 1990], [Pan 2013], [Kifetew 2017]). Dans le cas de modèles stochastiques, l'utilisateur peut également régler les probabilités des choix non-déterministes pris lors de la construction des données. Le réglage peut s'effectuer manuellement [Cekan 2017], par recherche métaheuristique [Feldt 2013] ou encore par apprentissage sur un jeu de données valides [Kifetew 2014].

— Des approches basées sur la résolution de contraintes. Celles-ci se focalisent souvent sur des instances de données de petites tailles. Par exemple, l'outil *TestEra* [Marinov 2001] exploite l'analyseur de *Alloy* [Jackson 2000] pour énumérer toutes les instances de données différentes (en fait, non isomorphes) jusqu'à une certaine taille. *UMLtoCSP* [Cabot 2007] utilise la programmation par contraintes pour créer des instances de données de taille bornée, les auteurs recommandant de prendre une petite borne. Par contre, l'outil *PLEDGE* [Soltana 2020] (présenté au chapitre précédent) vise à passer à l'échelle pour la production de données de taille réaliste, par une stratégie hybride combinant recherche métaheuristique et résolution de contraintes SMT.

Nous écartons la deuxième catégorie d'approches, qui procèdent à un réglage

fin de la génération. Les approches de la catégorie "générer-puis-filtrer" sont plus pertinentes pour la comparaison, offrant un moyen simple et générique de produire automatiquement des données à partir d'un modèle. L'efficacité de ces approches peut cependant être faible, du fait d'un trop fort taux de données invalides avant filtrage. Considérer ce type d'approche nous permet d'une part d'évaluer la difficulté des cas d'études pour la génération (une bonne performance du "générer-puis-filtrer" indiquant des cas faciles), et d'autre part d'évaluer la capacité de *TAF* à surmonter les cas difficiles. Pour réaliser l'évaluation, nous choisissons d'implémenter le "générer-puis-filtrer" sur les cas d'étude comme suit : nous retirons les contraintes du modèle de données en *XML-TAF*, procédons à la génération aléatoire avec *TAF*, puis filtrons les données invalides. Les résultats sont ensuite comparés à ceux de *TAF* avec le modèle contraint.

Notre outil *TAF* relevant de la troisième catégorie d'approches mentionnées ci-dessus (basées sur la résolution de contraintes), il est intéressant de le comparer aussi avec l'état de l'art dans cette catégorie. L'approche la plus significative en termes de passage à l'échelle semble être *PLEDGE* [Soltana 2020]. Les auteurs ont montré que leur stratégie hybride est capable de produire des données de plus grande taille que *Alloy* et *UMLtoCSP*. De plus, la stratégie hybride s'avère plus performante qu'une stratégie basée sur la métaheuristique de recherche seule. Nous retenons donc *PLEDGE* pour la comparaison. L'évaluation comparative reprend les deux critères de performance et de diversité.

La chapitre est structuré comme suit.Les différents cas d'étude et leur modélisation respective sont introduits dans la section 6.2. Pour chacun des cas d'étude des critères de couverture sont définis section 6.3 pour quantifier la diversité des données générées. Enfin (section 6.4), les résultats sont présentés et comparés avec les stratégies alternatives retenues : 1) générer puis filtrer, 2) *PLEDGE*. Tous les résultats présentés dans ce chapitre sont disponibles sur le git [Robert 2020a].

6.2 Modèles *XML-TAF* des cas d'étude

Cette section se concentre sur la description et la modélisation en *XML-TAF* de quatre cas d'étude (*Oz*, *Tax payer*, *Bitmap* et *Tree*). Comme dans la partie I, nous proposons d'abord une première modélisation sous forme de diagramme de classe *UML*. Cette modélisation donne un support visuel pour définir la structure avant de détailler la spécification des différents éléments en *XML-TAF*.

6.2.1 Cas d'étude *Oz*

Le cas d'étude *Oz* est très proche du cas simplifié qui a servi d'exemple au chapitre 4. Il comporte toutefois plus de variables et de contraintes. Pour rendre ce chapitre indépendant, le cas d'étude est quand même décrit intégralement. Il est très complet d'un point de vue modèle car il balaye une grande partie des possibilités qu'offrent *TAF* avec avec des contraintes sur des valeurs de paramètres mais aussi

sur des nombres d'instances. Un autre intérêt de ce cas d'étude pour le test de notre outil est la présence de dépendances cycliques.

Oz est un cas d'étude industriel de robot agricole qui a pour mission de désherber des champs de légumes. Pour son test en simulation nous souhaitons générer des champs et des paramètres de mission à la fois variés et cohérents.

FIGURE 6.1 – Exemple de champ pour *Oz*.

On souhaite modéliser un champ de plusieurs rangées légumes de nombre et de longueurs variables (figure 6.1). En plus des longueurs, on attribue les variables "vegetable_density", "noise_X" et "noise_Y" à chaque rangée. Ces variables permettent de modéliser le nombre de légumes par mètre et le bruit en centimètre sur leur position selon l'axe x et y. Dans un champ, il est fréquent que des légumes manquent sur certaines zones. En conséquence, une variable "disappearance_probability" modélise la probabilité d'absence de chaque légume au sein de chaque rangée. Les paramètres considérés sont ceux de la figure 6.2.

Dans le cas où il y a plusieurs rangées, on s'assure que deux rangées successives n'ont pas trop d'écart de longueur (10%) pour garantir une forme de champ réaliste. On définit également une relation similaire entre la première et la dernière rangée formant ainsi un cercle de dépendance. Ces contraintes sont respectivement définis ci-dessous :

$$\text{contrainte "interval"} = \begin{cases} \forall i \in [\![1, n]\!], row[i].length > 0.9 * row[i-1].length \\ \forall i \in [\![1, n]\!], row[i].length < 1.1 * row[i-1].length \end{cases}$$

$$\text{contrainte "interval\_2"} = \begin{cases} row[0].length > 0.9 * row[n].length \\ row[0].length < 1.1 * row[n].length \end{cases}$$

Le modèle complet modélisé en *XML-TAF* est disponible figure 6.3. Du point de vue des contraintes, deux paramètres sont intéressants. Le premier représente l'espacement entre chaque rangée (gap L26) et le deuxième la densité de mauvaises herbes qui y est présente (grass_density L23). Le nombre de variables d'espacement inter-rangée doit être de $n-1$ (avec n le nombre de rangées). Similairement, le nombre de paramètres pour la densité de mauvaises herbes doit être de multiplicité $n+1$. En effet, on considère les $n-1$ zones d'inter-rangée ainsi que les deux zones de part et d'autre du champ. Ces deux contraintes sont exprimées entre les lignes 28 et 31.

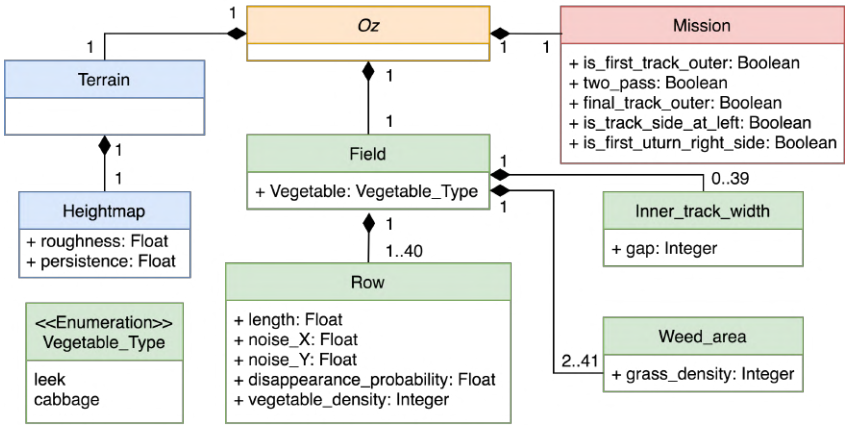


FIGURE 6.2 – Diagramme de classes UML du cas d'étude *Oz*.

Finalement, on ajoute également 5 paramètres booléen de mission (L36 à L41). Ils comportent les informations nécessaires pour que le robot puissent effectuer sa mission. Le paramètre "is_first_track_outer" est à vrai lorsque le départ de mission s'effectue sur le coté du champ. Dans le cas d'un champ composé d'une unique rangée, ce paramètre doit trivialement prendre la valeur Vrai (contrainte "first_track").

Les paramètres de terrain sont non-contraints ("roughness" et "persistence"). ils sont utilisés lors de la phase de mise en forme des données (export) pour générer le relief du champ. Les paramètres de bruit sur la position des légumes et celui de probabilité de légumes manquants sont aussi traités lors cette même étape. C'est un exemple concret de cas où un même cas de test donne par la suite différentes entrées de test. Autrement dit, ce sont des méta-paramètres servant par la suite à la génération procédurale de contenu de monde.

Pour ce cas d'étude, l'étape d'exportation des cas de test crée 3 fichiers : un fichier .world permettant de définir la position des différents éléments de la scène 3D pour le simulateur, un fichier .json avec les paramètres de mission à donner au robot pour l'exécution de la mission et une image .jpeg représentant le relief du terrain pour le simulateur.

6.2.2 Cas d'étude *Tax payer*

Le cas d'étude *Tax payer* est extrait de l'article de référence sur l'outil *PLEDGE* [Soltana 2020]. Il est modéliser en UML / OCLL et comporte des contraintes intéressantes pour le test de notre outil (par exemple : le cas de quantificateurs imbriqués).

Ce cas d'étude consiste en une population d'individus pour le test d'un logiciel d'imposition au Luxembourg. Chaque individu peut avoir plusieurs adresses, plusieurs sources de revenus ou encore plusieurs enfants à charge. Il peut également avoir un handicap ainsi qu'un taux d'invalidité correspondant. La figure 6.4 donne

```
<?xml version="1.0"?>

<root name="test_case">
    <node name="field" nb_instances="1">
        <parameter name="vegetable" type="string" values="cabbage;leek"/>
        <node name="row" min="1" max="100">
            <parameter name="length" type="real" min="10.0" max="100.0"/>
            <parameter name="noise_X" type="real" min="0.0" max="5.0"/>
            <parameter name="noise_Y" type="real" min="0.0" max="5.0"/>
            <parameter name="disappearance_probability" type="real" min="0.0" max="30.0"/>
            <parameter name="vegetable_density" type="integer" min="1" max="5"/>

            <constraint name="interval" types="forall"
                        expressions="row[i]\length INFEQ 1.1*row[i-1]\length;
                                     row[i]\length SUPEQ 0.9*row[i-1]\length"
                        quantifiers="i"
                        ranges="[1, row.nb_instances-1]"/>
            <constraint name="interval_2"
                        expressions="row[0]\length INFEQ 1.1*row[row.nb_instances - 1]\length;
                                     row[0]\length SUPEQ 0.9*row[row.nb_instances - 1]\length"/>
        </node>
        <node name="weed_area" min="2" max="101">
            <parameter name="grass_density" type="integer" min="0" max="5"/>
        </node>
        <node name="inner_track_width" min="0" max="99">
            <parameter name="gap" type="integer" min="55" max="165"/>
        </node>
        <constraint name="nb_weed_area"
                    expressions=".\weed_area.nb_instances EQ .\row.nb_instances + 1"/>
        <constraint name="nb_inner_track_width"
                    expressions=".\inner_track_width.nb_instances EQ .\row.nb_instances - 1"/>

    </node>

    <node name="mission" nb_instances="1">
        <parameter name="two_pass" type="boolean"/>
        <parameter name="is_first_track_outer" type="boolean"/>
        <parameter name="final_track_outer" type="boolean"/>
        <parameter name="is_track_side_at_left" type="boolean"/>
        <parameter name="is_first_uturn_right_side" type="boolean"/>
        <constraint name="first_track"
                        expressions="IMPLIES(..\field\row.nb_instances EQ 1, .\is_first_track_outer EQ True)"/>
    </node>
    <node name="terrain" nb_instances="1">
        <node name="heightmap" nb_instances="1">
            <parameter name="roughness" type="real" min="0.0" max="1"/>
            <parameter name="persistence" type="real" min="0.0" max="0.7"/>
        </node>
    </node>

</root>
```

FIGURE 6.3 – Modèle *XML-TAF* pour le cas d'étude *Oz*.

une vue schématique de ce cas d'étude. Une vue UML, telle qu'elle est décrite dans l'article présentant l'outil *PLEDGE*, est donnée figure 6.5, elle permet de faire le parallèle avec le modèle pour *TAF* donné un peu plus loin (figure 6.6).

Le Luxembourg est un pays comptant de nombreux travailleurs ressortissants des pays frontaliers. La fiscalité y est donc complexe est doit prendre en compte la diversité des travailleurs ainsi que l'origine de leurs revenus. Le modèle UML montre deux paramètres clefs de ce cas d'étude. Le premier est un booléen associé à chaque contribuable pour signifier si celui-ci réside au Luxembourg ("is_resident") et le deuxième associe cette même notion de localité à chaque source de revenus ("is_local").

Le modèle inclut 5 contraintes OCL explicitées ci-dessous. Elle sont valables pour tout les contribuables.

— C1 : L'année de naissance d'une personne est comprise entre 1920 et 2020.

— C2 : Si une personne n'a pas de handicap (type : None) alors son taux doit

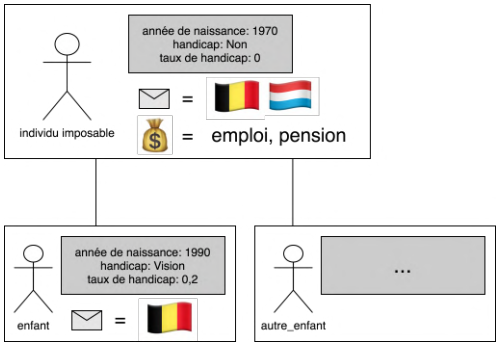


FIGURE 6.4 – Exemple d'un individu de la population de contribuables (*Tax payer*).

être nul. En revanche si une personne a un handicap alors son taux doit être supérieur à zéro et plus petit que 1.

— C3 : Un contribuable ayant au moins une adresse au Luxembourg ("LU") est considéré comme résident

— C4 : Si au moins un revenu d'un contribuable est local, mais aucune adresse de ce contribuable n'est au Luxembourg, alors il n'est pas résident.

— C5 : Si une source de revenu est de type "OTHER", alors aucune carte de taxe ("Tax_card") ne lui est associée. Pour les autres types de revenu une carte de taxe est associée.

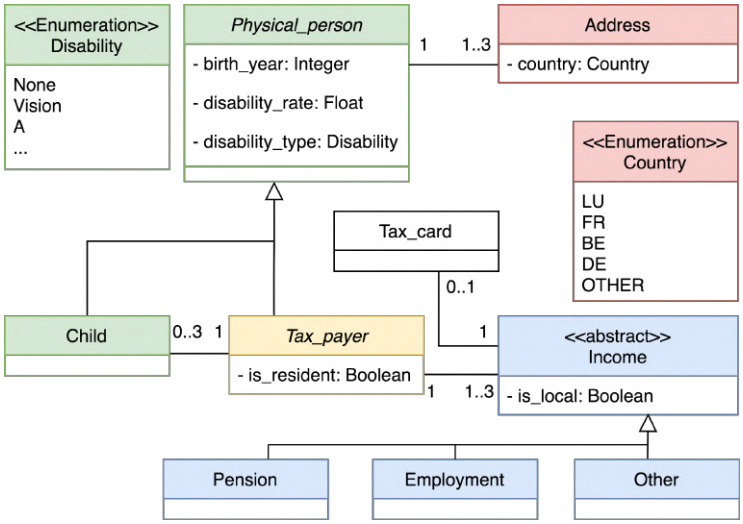
Ce modèle permet de générer des données peu logiques comme par exemple un enfant à charge dont l'âge et supérieur à celui du contribuable parent. Ce modèle correspond à celui présenté dans le papier de *PLEDGE*. Nous le jugeons suffisant pour le test de *TAF* et dans un but comparatif nous nous limitons à cette version.

Le modèle en *XML-TAF* est disponible figure 6.6. La traduction depuis un modèle UML / OCL est délicate car la structure des modèles *TAF* est structurellement différente. La principale difficulté est d'exprimer un modèle équivalent sous forme d'arbre.

De manière générale il est possible de traduire un modèle UML / OCL en un modèle *TAF*. Le résultat est toutefois moins compact car il manque les concepts d'héritage et d'association. Pour exprimer l'équivalence d'un héritage, il faut redistribuer les variables dans les classes filles faisant apparaître des définitions multiples. Pour les associations, il faut ajouter des variables d'indexation permettant de faire référence à la classe associée. Il faut également ajouter un système de contraintes pour assurer la bidirectionnalité de l'association.

Pour le modèle équivalent *TAF*, nous avons remplacé l'association entre les classes "Tax_payer" (L3) et "Child" (L13) par une composition. De la même façon, la classe "Income" est devenue les nodes "income_pension" (L26), "income_employment" (L31) et "income_other" (L36).

Les contraintes sont ensuite exprimées en conséquence. La contrainte "C1" n'a pas besoin d'être définie à l'aide d'un élément <contrainte> car le paramètre



(a) Modèle UML du cas d'étude *Tax payer*

```
1  context Physical_person inv C1:
2  self.birth_year >=1920 and self.birth_year <2020
3
4  context Physical_person inv C2:
5  if (self.disability_type = Disability::None)
6  then (self.disability_rate = 0)
7  else (self.disability_rate > 0 and
8  self.disability_rate <= 1.0) endif
9
10 context Tax_payer inv C3:
11 not self.address->forAll(a:Address |
12         a.country <> Country::LU)
13 implies self.is_resident=true
14
15 context Tax_payer inv C4:
16 self.income->exists(inc:Income | inc.is_local=true) and
17 not self.address->exists(a:Address |
18         a.country = Country::LU)
19 implies self.is_resident=false
20
21 context Income inv C5:
22 if(self.oclIsTypeOf(Other)) then
23 self.tax_card->size()=0 else
24 self.tax_card->size()=1 endif
```

(b) Modèle OCL du cas d'étude *Tax payer*

FIGURE 6.5 – Modèle UML / OCL du cas d'étude *Tax payer* extrait de [Soltana 2020].

"birth_year" (L5 et L14) est borné dans sa définition. La contrainte "C2" est déclarée en deux parties pour reproduire l'héritage de la classe "Physical_person" ("c2" dans le noeud "tax_payer" et "c2a" dans le noeud "child" L43 et L20). La contrainte "C3" est exprimée de la même manière, elle n'est pas impactée par la traduction du modèle. La contrainte "C4" est divisée dans chaque type de revenu ("c4_pension" L51, "c4_employment" L55 et "c4_other" L59). La contrainte "c5" n'est pas exprimée dans le modèle *TAF*. Elle est toujours vraie par construction. La séparation de

```
1  <?xml version="1.0"?>
2
3  <root name="test_case">
4      <node name="tax_payer" min="1" max="100">
5          <parameter name="birth_year" type="integer" min="1920" max="2020"/>
6          <parameter name="disability_rate" type="real" min="0" max="1"/>
7          <parameter name="disability_type" type="string" values="None;Vision;A"/>
8          <parameter name="is_resident" type="boolean"/>
9      <node name="address" min="1" max="3">
10             <parameter name="country" type="string" values="LU;FR;BE;DE;OTHER"/>
11         </node>
12
13         <node name="child" min="0" max="3">
14             <parameter name="birth_year" type="integer" min="1920" max="2020"/>
15             <parameter name="disability_rate" type="real" min="0" max="1"/>
16             <parameter name="disability_type" type="string" values="None;Vision;A"/>
17             <node name="address" min="1" max="3">
18                 <parameter name="country" type="string" values="LU;FR;BE;DE;OTHER"/>
19             </node>
20             <constraint name="c2a" types="forall"
21                     expressions="IMPLIES(child[i]\disability_type EQ None, child[i]\disability_rate EQ 0);IMPLIES(
   child[i]\disability_rate EQ 0, child[i]\disability_type EQ None)"
22                     quantifiers="i"
23                     ranges="[0, child.nb_instances-1]"/>
24         </node>
25
26         <node name="income_pension" min="0" max="3">
27             <parameter name="is_local" type="boolean"/>
28             <node name="tax_card" nb_instances="1">
29             </node>
30         </node>
31         <node name="income_employment" min="0" max="3">
32             <parameter name="is_local" type="boolean"/>
33             <node name="tax_card" nb_instances="1">
34             </node>
35         </node>
36         <node name="income_other" min="0" max="3">
37             <parameter name="is_local" type="boolean"/>
38         </node>
39     <constraint name="income" types="forall"
40                 expressions="tax_payer[i]\income_pension.nb_instances + tax_payer[i]\income_employment.
   nb_instances + tax_payer[i]\income_other.nb_instances INFEQ 3;tax_payer[i]\income_pension.nb_instances +
   tax_payer[i]\income_employment.nb_instances + tax_payer[i]\income_other.nb_instances SUP 0"
41                     quantifiers="i"
42                     ranges="[0, tax_payer.nb_instances-1]"/>
43         <constraint name="c2" types="forall"
44                 expressions="IMPLIES(tax_payer[i]\disability_type EQ None, tax_payer[i]\disability_rate EQ 0);
   IMPLIES(tax_payer[i]\disability_rate EQ 0, tax_payer[i]\disability_type EQ None)"
45                 quantifiers="i"
46                 ranges="[0, tax_payer.nb_instances-1]"/>
47     <constraint name="c3" types="forall;forall"
48                 expressions="IMPLIES(tax_payer[i]\address[j]\country EQ LU, tax_payer[i]\is_resident EQ True)"
49                 quantifiers="i;j"
50                 ranges="[0, tax_payer.nb_instances-1];[0, tax_payer[i]\address.nb_instances-1]"/>
51     <constraint name="c4_pension" types="forall;forall;exist"
52                 expressions="IMPLIES(AND(tax_payer[i]\income_pension[j]\is_local EQ True, NOT(tax_payer[i]\
   address[k]\country EQ LU)), tax_payer[i]\is_resident EQ False)"
53                 quantifiers="i;j;k"
54                 ranges="[0, tax_payer.nb_instances-1];[0, tax_payer[i]\income_pension.nb_instances-1];[0, tax_payer[
   i]\address.nb_instances-1]"/>
55     <constraint name="c4_employment" types="forall;forall;exist"
56                 expressions="IMPLIES(AND(tax_payer[i]\income_pension[j]\is_local EQ True, NOT(tax_payer[i]\
   address[k]\country EQ LU)), tax_payer[i]\is_resident EQ False)"
57                 quantifiers="i;j;k"
58                 ranges="[0, tax_payer.nb_instances-1];[0, tax_payer[i]\income_employment.nb_instances-1];[0,
   tax_payer[i]\address.nb_instances-1]"/>
59     <constraint name="c4_other" types="forall;forall;exist"
60                 expressions="IMPLIES(AND(tax_payer[i]\income_pension[j]\is_local EQ True, NOT(tax_payer[i]\
   address[k]\country EQ LU)), tax_payer[i]\is_resident EQ False)"
61                 quantifiers="i;j;k"
62                 ranges="[0, tax_payer.nb_instances-1];[0, tax_payer[i]\income_other.nb_instances-1];[0, tax_payer[i]\
   address.nb_instances-1]"/>
63
64     </node>
65 </root>
```

FIGURE 6.6 – Modèle *XML-TAF* pour le cas d'étude *Tax payer*.

l'objet "Income" en trois noeuds permet de placer des noeuds "tax_card" directement au bon endroit (dans "income_pension" L28 et "income_employment" L33). Pour finir, la contrainte "income" (L39) est ajoutée pour gérer le nombre de sources

de revenus. Dans le modèle initial, on a entre une et trois sources de revenus. Ce qui peut se traduire par : $0 < \Sigma(income\_n) \leq 3$ avec n les types de revenu.

Nous ne connaissons pas le format requis pour le logiciel d'imposition. Il n'y a donc pas de mise en forme des données générées, elles sont conservées dans le format brut de *TAF*. Un exemple d'exportation sous forme csv est néanmoins donné dans [Robert 2020a].

6.2.3 Cas d'étude *Bitmap*

Le cas d'étude *Bitmap* est construit dans le but de tester notre outil avec un résultat visuellement intéressant. Il a la particularité de contenir un grand nombre de contraintes simples et permet d'en évaluer la compatibilité avec notre approche basée sur des appels répétés au solveur. Il permet également de montrer l'utilisation du fichier *export.py* pour l'étape de transformation des données générées vers un format propriétaire bitmap.

Ce cas d'étude consiste en la génération d'images matricielles carrées représentant un dégradé en niveaux de gris. Le dégradé est choisi diagonal avec la zone la plus foncée dans le coin inférieur gauche et la zone la plus claire dans le coin supérieur droit.

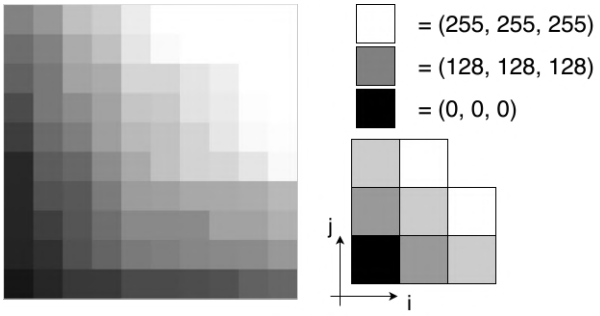


FIGURE 6.7 – Exemple d'image bitmap avec dégradé.

Une image matricielle ou carte de points (bitmap) est une image constituée d'une matrice de points colorés ou autrement dit constituée d'une grille ou chaque case contient une couleur qui lui est propre. Le format bitmap (BMP) est un format ouvert pour les images matricielles, il a l'avantage d'être simple et d'être lisible par quasiment tous les lecteurs d'images.

La figure 6.7 est un exemple créé manuellement au format bitmap classique. Le système de codage des couleurs utilisées est RGB (**R**ed **G**reen **B**lue). Pour chaque pixel, on attribue 3 valeurs comprises entre 0 et 255 (1 octet) donnant dans l'ordre la composante rouge, verte puis bleue de la couleur que l'on souhaite exprimer. Dans ce système, exprimer un niveau de gris revient à forcer ces trois composantes à la même valeur. On a 0,0,0 pour l'absence de couleur (noir) et 255,255,255 pour la présence maximale de toutes les couleurs (blanc). Dans notre modèle nous considérons une unique valeur par pixel. Un traitement des données générées lors de l'étape de mise

en forme (export) transforme ces valeurs en triplet RGB pour former une image valide.

Pour définir les contraintes permettant d'assurer un dégradé diagonal, on définit des relations entre le niveau de gris d'un pixel et celui des ses voisins. Soit un repère dans le coin en bas à gauche de l'image avec i l'axe horizontal et j l'axe vertical tel que $i, j \in \mathbb{N}; 0 \leq i, j \leq width - 1$. i et j représentent les coordonnées d'un pixel, ils sont tous les deux définis sur le même intervalle car l'image est carrée. Cette notation permet d'écrire les relations suivantes :

$$\forall i, j \in [\![1, width - 1]\!], pixel[i][j].niveau\_de\_gris \leq pixel[i+1][j].niveau\_de\_gris$$
$$\forall i, j \in [\![1, width - 1]\!], pixel[i][j].niveau\_de\_gris \leq pixel[i][j+1].niveau\_de\_gris$$

Ces relations décrivent le minimum nécessaire pour spécifier ce cas d'étude. Néanmoins, l'espace des solutions est encore trop vaste pour avoir un résultat satisfaisant visuellement. Dans le modèle final (figure 6.9), nous avons ajouté un écart maximal entre les valeurs de deux pixels consécutifs pour obtenir un dégradé. Pour ce modèle, l'utilisation standard d'un solveur (sans passer par *TAF*) donne une image monochrome (la solution triviale respectant toutes les contraintes). A l'inverse, la diversité des solutions générées par *TAF* est ici facilement visible et démontre l'intérêt de notre approche (section 6.4).

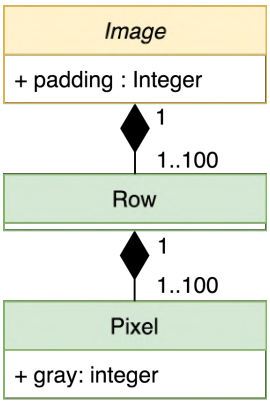


FIGURE 6.8 – Diagramme de classes UML du cas d'étude *Bitmap*.

La figure 6.9 est la modélisation en *XML-TAF* de ce cas d'étude (le modèle de structure en UML est donné figure 6.8). On se limite à des images carrées de 10 à 100 pixels de coté (L4 et L5). La contrainte ("nb_pixel" L9) assure que chaque ligne contient le même nombre de pixels, elle contraint également ce nombre pour qu'il égale le nombre de lignes forçant ainsi une image carrée. Les deux contraintes suivantes ("horizontal_gradient" L13 et "vertical_gradient" L17) correspondent aux relations pour le dégradé décrit précédemment. On utilise le fait que l'image soit carrée pour simplifier les expressions des intervalles (ranges L16 et L20) pour parcourir les pixels.

```
1  <?xml version="1.0"?>
2
3  <root name="test_case">
4      <node name="row" min="10" max="100">
5          <node name="pixel" min="10" max="100">
6              <parameter name="gray" type="integer" min="0" max="255"/>
7          </node>
8
9          <constraint name="nb_pixel" types="forall"
10                 expressions="row[i]\pixel.nb_instances EQ row.nb_instances"
11                     quantifiers ="i"
12                 ranges="[0, row.nb_instances-1]"/>
13         <constraint name="horizontal_gradient" types="forall;forall"
14                 expressions="row[i]\pixel[j]\gray INFEQ row[i]\pixel[j+1]\gray"
15                     quantifiers ="i;j"
16                 ranges="[0, row.nb_instances-1];[0, row.nb_instances-2]"/>
17         <constraint name="vertical_gradient" types="forall;forall"
18                 expressions="row[i]\pixel[j]\gray INFEQ row[i+1]\pixel[j]\gray"
19                     quantifiers ="i;j"
20                 ranges="[0, row.nb_instances-2];[0, row.nb_instances-1]"/>
21     </node>
22     <parameter name="padding" type="integer" min="0" max="3"/>
23     <constraint name="nb_padding"
24             expressions="(4-(3*test_case\row.nb_instances)%4)%4 EQ test_case\padding"/>
25 </root>
```

FIGURE 6.9 – Modèle *XML-TAF* pour le cas d'étude *Bitmap*.

Dans un but illustratif, nous avons également intégré au modèle un paramètre prenant la taille du *padding* (remplissage) à ajouter (L22). Dans le format bitmap une image est décrite ligne par ligne. Pour optimiser la lecture chaque ligne doit être un multiple de 4 octets. Pour atteindre un tel multiple, il peut être nécessaire d'ajouter une suite de zéro (*padding*). Prenons par exemple une ligne de 30 pixels, chaque pixel est codé sur 3 octets (RGB) ce qui donne 90 octets. 90 n'est pas un multiple de 4, il faut ajouter 2 octets de zéro pour arriver à 92. C'est la contrainte "nb_padding" (L23) qui a ce rôle. Elle utilise un opérateur de modulo.

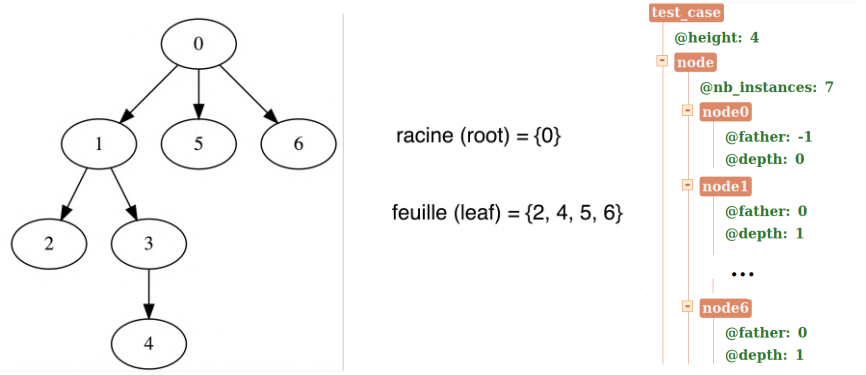
L'étape de mise en forme (`export`) est assez directe. il suffit de suivre la spécification bitmap ([Murray 1996]) et de former le fichier binaire correspondant. On y ajoute une entête de fichier avec entre autres la taille du fichier et les deux octets de signature propres au bitmap (424D -> "BM"). On ajoute également l'entête d'information avec par exemple la palette de couleur et toutes les options. Le fichier se termine par les données en commençant par la ligne du bas de l'image.

6.2.4 Cas d'étude *Tree*

Le cas d'étude *Tree* est intéressant car c'est une structure de données très commune. Un arbre est un cas particulier de graphe, plus précisément un graphe connexe sans cycle muni d'une racine. La spécification du modèle est relativement simple et compacte car elle fait uniquement intervenir des relations de descendance. Elle laisse beaucoup de liberté à la génération en intégrant le moins de connaissance possible. Il en résulte un ensemble de contraintes pouvant être complexe à traiter.

Ce cas d'étude consiste à générer des arbres de taille et de hauteur variées. Chaque noeud de l'arbre comporte un index unique. Mis à part la racine qui porte l'index 0, les noeuds peuvent être placés arbitrairement.

La figure 6.10 est un exemple d'arbre de taille 7 (c'est-à-dire, comportant 7 noeuds). On définit la racine comme le noeud sans parent c'est-à-dire le noeud



(a) Exemple d'arbre

(b) Structure *XML* associée

FIGURE 6.10 – Exemple d'arbre.

ancêtre de toute la structure (noeud 0). Les feuilles (ou noeuds terminaux) sont les noeuds sans descendant (noeuds 2, 4, 5 et 6). Tout noeud qui n'est pas une feuille est un noeud interne. On définit la profondeur (*depth*) d'un noeud comme la longueur du chemin qui le relie à la racine. Ainsi dans notre exemple, la racine a pour profondeur 0, les noeuds 1, 5 et 6 ont une profondeur de 1, les noeuds 2 et 3 ont une profondeur de 2 et le noeud 4 a une profondeur de 3. La hauteur (*height*) d'un arbre est la profondeur maximale de ses noeuds.

La figure 6.11 montre une modélisation classique de la structure d'arbre en UML. Les noeuds sont reliés entre eux par des relations parent-enfant, un noeud parent pouvant avoir un nombre arbitraire d'enfants. Cette structure ne peut pas être reproduite telle quelle en *XML-TAF*. Le langage n'autorise pas les définitions récursives : on ne peut pas faire apparaître les noeuds enfant comme sous-éléments de leur noeud parent. La modélisation de *Tree* va donc contenir une liste "à plat" de noeuds, et il nous faut coder la relation parent-enfant par un paramètre spécifique et des contraintes. La figure 6.10b montre une instance XML d'arbre correspondant à la figure 6.10a. Chaque noeud contient un paramètre spécifique parent, qui donne l'index de celui-ci.

Pour qu'un arbre soit correctement défini chaque noeud doit avoir un unique parent (sauf la racine qui n'en a pas). Il existe donc un unique chemin entre une feuille et la racine. De plus il doit exister un chemin reliant un noeud à n'importe quel autre noeud (connexité). Ce chemin est unique à cause de la propriété précédente. La propriété de connexité implique l'unicité de la racine.

La spécification d'un arbre pour *TAF* est donné dans la figure 6.12. Elle comprend un ensemble de contraintes sur les paramètres parent, *depth* et *height* pour avoir un arbre valide.

Les deux premières contraintes (L7-L12) assurent l'unicité de la racine. La contrainte "root_parent" définit le noeud 0 comme la racine. La valeur de son parent est forcée à -1. C'est une valeur arbitraire qui se différencie des valeurs d'in-

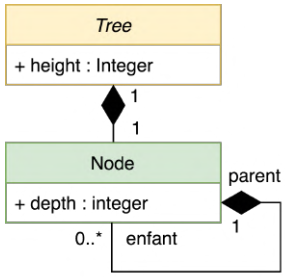


FIGURE 6.11 – Diagramme de classes UML du cas d'étude *Tree*.

```
1  <?xml version="1.0"?>
2
3  <root name="test_case">
4      <parameter name="height" type="integer" min="0" max="50"/>
5      <node name="node" min="1" max="50">
6          <parameter name="parent" type="integer" min="-1" max="50"/>
7          <constraint name="root_parent"
8                  expressions="node[0]\parent EQ -1"/>
9          <constraint name="other_parent" types="forall"
10                 expressions="node[i]\parent SUPEQ 0; node[i]\parent INF node.nb_instances"
11                 quantifiers ="i"
12                 ranges="[1,node.nb_instances-1]"/>
13
14         <parameter name="depth" type="integer" min="0" max="50"/>
15         <constraint name="inductive_depth_0"
16                 expressions="node[0]\depth EQ 0"/>
17         <constraint name="inductive_depth_1" types="forall;forall"
18                 expressions="IMPLIES(node[i]\parent EQ j, node[j]\depth+1 EQ node[i]\depth)"
19                 quantifiers ="i;j"
20                 ranges="[1,node.nb_instances-1];[0,node.nb_instances-1]"/>
21         <constraint name="height_depth_1" types="forall"
22                 expressions="node[i]\depth INFEQ ..\height"
23                 quantifiers ="i"
24                 ranges="[1, node.nb_instances-1]"/>
25         <constraint name="height_depth_2" types="exist"
26                 expressions="node[i]\depth EQ ..\height"
27                 quantifiers ="i"
28                 ranges="[0,node.nb_instances-1]"/>
29     </node>
30
31     <constraint name="tree_height_1"
32             expressions="test_case\height INFEQ test_case\node.nb_instances-1"/>
33     <constraint name="tree_height_2"
34             expressions="IMPLIES(test_case\height EQ 0, test_case\node.nb_instances EQ 1)"/>
35 </root>
```

FIGURE 6.12 – Modèle en *XML-TAF* pour le cas d'étude tree.

dex possibles pour signifier que la racine n'a pas de parent. La deuxième contrainte ("other_father" L9) est valable pour tous les autres noeuds et spécifie que l'index du noeud parent doit correspondre à un noeud existant (entre 0 et $nb\_instances - 1$).

On définit ensuite inductivement les valeurs de profondeur (*depth*) de chaque noeud. La contrainte "inductive_depth_0" (L15) donne la profondeur 0 à la racine de l'arbre. La contrainte "inductive_depth_1" donne à tout autre noeud la profondeur de son parent incrémentée de 1.

Il ne reste plus qu'à spécifier ce qu'est une hauteur et à donner les relations sémantiques qui lient hauteur, nombre de noeuds et profondeur de ces noeuds. Deux contraintes assurent que la profondeur de chaque noeud est inférieure à la hauteur de l'arbre et qu'il existe au moins un noeud de profondeur égale à la hauteur désirée

(respectivement "height_depth_1" L21 et "height_depth_2" L25). Ensuite, les contraintes "tree_height_1" (L31) et "tree_height_2" (L33) assurent la cohérence de la hauteur de l'arbre et de sa taille. La première contraint la hauteur à une valeur strictement inférieure au nombre de noeuds. En effet il est impossible de produire un arbre de hauteur n avec n noeuds. Dans le cas particulier où la hauteur de l'arbre correspond à $n-1$, il forme un arbre linéaire (c'est l'unique arrangement des noeuds possible). La contrainte "tree_height_2" gère un autre cas particulier : l'arbre de hauteur nulle. Il est unique et n'est formé que du noeud racine. En conséquence, la contrainte spécifie qu'une hauteur de 0 implique un arbre composé d'un seul noeud.

La modélisation d'arbre que nous avons donné ci-dessus reste très déclarative. Elle laisse beaucoup de liberté à la génération : par exemple, on n'indique pas de procédure pour construire les relations de parent de façon cohérente avec une valeur de hauteur globale. Il en résulte un ensemble de contraintes pouvant être complexes à traiter.

L'étape d'exportation des cas de test est facilitée par l'utilisation des outils *Graphviz*. Ils permettent la manipulation et la visualisation de graphe à partir du langage *DOT*. La mise en forme se résume donc à traduire les données générées vers un équivalent au format *DOT*.

6.3 Critères de couverture pour les différents cas d'études

La figure 6.13 donne un exemple des données générées pour les cas d'études de la section précédente. On a de gauche à droite : un champ pour *Oz*, une population de contribuables (*Tax payer*), une image bitmap en dégradé de gris (*Bitmap*) et un arbre (*Tree*). L'exemple pour le cas d'étude *Tax payer* présente simplement le nombre d'individus de la population car les données sont brutes et ne présentent pas d'intérêt visuel.

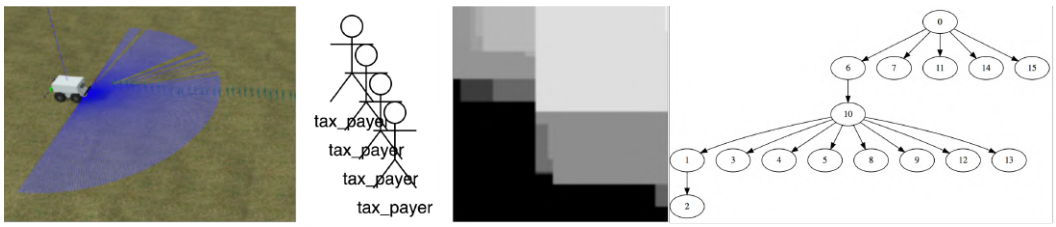


FIGURE 6.13 – Quatre exemples de données générées par *TAF*.

La visualisation des données comme dans la figure précédente n'est pas suffisante pour justifier rigoureusement la diversité des données générées. Dans cette section est présentée la méthode permettant d'en évaluer la variété et des critères de couverture spécifique à chaque cas d'étude sont définis.

6.3.1 Principes généraux

Ces critères suivent les 3 principes suivants :

— Dans un modèle *TAF*, chaque paramètre numérique ou nombre d'instances est borné par un minimum et un maximum. Nous divisons systématiquement ces intervalles en trois sous-intervalles : faible, moyen et élevé (noté **L**ow, **M**edium et **H**igh). Pour déterminer à quel sous-intervalle appartient une valeur, on commence par la normaliser $(valeur - min)/(max - min)$. On a ensuite : $L \in [0, 0.333[, M \in [0.333, 0.667[, H \in [0.667, 1]$.

On considère qu'un sous-intervalle est couvert s'il existe une instance du paramètre qui a une valeur dans ce sous intervalle. Un paramètre numérique est couvert si est seulement si ses trois sous-intervalles sont couverts

— Un paramètre de type énuméré est couvert lorsque chaque valeur possible est couverte.

— Des critères dérivés des contraintes et des relations sémantiques entre les paramètres sont construits spécifiquement pour chaque cas d'étude.

6.3.2 Critères de couverture pour le cas d'étude *Oz*

Pour chaque cas d'étude, il y a beaucoup de possibilités pour construire des critères sur les relations sémantiques entre les paramètres. Les critères que nous avons choisis d'évaluer sont décrits ci-dessous. La liste exhaustive des critères pour chaque cas d'études est disponible en annexe (A).

Lorsque l'on donne une contrainte à un solveur, celui-ci renvoie généralement une solution triviale. Par exemple, avec la contrainte $x > 0$ le solveur va renvoyer une solution du type $x = 1$. Dans le cas d'étude *Oz*, les longueurs des rangées de légumes successives sont contraintes de manière à ce que l'écart de longueur ne dépasse pas +/-10% (contrainte "interval"). Une solution triviale est alors d'attribuer la même longueur à chaque rangée ou exactement 10% d'écart. Nous ajoutons donc un critère permettant d'évaluer la couverture de l'écart entre deux rangées. La couverture est évaluée sur les bases des paramètres numériques avec une division en trois sous-intervalles (L-M-H).

Cette même idée est également appliquée sur l'écart de longueur entre la première et la dernière rangée du champ (contrainte "interval_2"). En revanche, ce critère n'est évalué que dans le cas où le champ comporte plus de deux rangées (c'est-à-dire si la première et la dernière rangées ne sont pas consécutives).

Pour finir, on s'intéresse aux valeurs prises par la première rangée (L-M-H). En cas de non-couverture, on peut s'attendre à une certaine monotonie sur les longueurs et donc à des champs de formes similaires.

6.3.3 Critères de couverture pour le cas d'étude *Tax payer*

Dans le modèle pour *TAF*, les sources de revenus sont définis comme trois éléments différents. Pour avoir l'équivalence avec le modèle UML / OCL. La contrainte

"income" permet de borner le nombre de sources de revenus entre 1 et 3 en sommant le nombre d'instances des différents éléments. De la même manière (en sommant les instances), on peut définir un critère de couverture sur le nombre de sources de revenus.

Pour le critère relatif à la contrainte "C3" (contrainte entre les adresses et le statut de résident), nous considérons trois cas. Le contribuable n'a pas d'adresse au Luxembourg, au moins une adresse au Luxembourg et une autre ailleurs, ou toutes les adresses sont au Luxembourg. Le critère est simplement la couverture de ces trois cas.

Le critère sur la contrainte "C4" (contrainte entre la localité des revenus et le statut de résident) est traité de manière analogue. On souhaite couvrir les cas d'au moins une adresse au Luxembourg associée à chacun des types de revenus.

6.3.4 Critères de couverture pour le cas d'étude *Bitmap*

Le cas d'étude est constitué de contraintes entre les valeurs d'un pixel et celle des ses voisins ("horizontal_gradient" et "vertical_gradient") et d'une contrainte sur le padding (remplissage) propre au format ("padding").

La relation de voisinage entre les pixels est une simple inégalité. Elle permet tous les dégradés possibles allant du dégradé nul (image monochrome) et un dégradé fort (image qui va d'un extrême de la palette à l'autre). Pour évaluer la diversité, en plus de couvrir la plage de valeur d'un pixel, nous allons également nous intéresser aux valeurs prises par le pixel le plus sombre et le pixel le plus clair. Ainsi si le pixel le plus clair est très sombre, on déduit que l'image est quasiment monochrome. De même si le pixel le plus sombre est très clair.

On définit également des critères sur l'écart entre pixels voisins à l'aide de sous-intervalles. Un écart dans le sous-intervalle L signifie que la couleur ne change pas alors qu'un écart dans H correspond à un changement brusque de couleur. Par soucis de cohérence avec les contraintes, nous découpons ce critère en deux et différencions les écarts selon l'axe horizontal et vertical de l'image. De plus, deux critères similaires sont ajoutés pour évaluer les variations de couleur au sein d'une ligne ou d'une colonne entière de pixels. Ils sont construits en mesurant la différence entre le premier et le dernier pixel.

La contrainte sur le padding donne le nombre d'octets à ajouter à la fin de chaque ligne de l'image pour avoir un multiple de 4 octets. Les 4 cas possibles (on ajoute 0 octet, on ajoute 1 octet, ...) forment le critère à couvrir pour la contrainte de padding. Cette mesure peut paraître redondante avec celle sur la taille des lignes, mais ce n'est pas le cas. On peut par exemple couvrir tous les cas de padding en restant dans le sous-intervalle L pour la taille des lignes.

6.3.5 Critères de couverture pour le cas d'étude *Tree*

Nous définissons quatre critères pour le cas d'étude *Tree*. En premier lieu, nous souhaitons évaluer la diversité dans le rapport entre la hauteur et le nombre de

noeuds pour des cas allant d'un arbre peu profond avec tous les noeuds sous la racine, à un arbre linéaire avec la hauteur maximale possible. Ce critère est défini ci dessous :

$$height\_vs\_size = \frac{height-1}{size-2}$$

Il n'est mesuré que pour une taille d'arbre supérieure ou égale à 4 de manière à avoir un minimum de 3 possibilités pour la hauteur (pour le découpage en sous-intervalles).

Les autres critères évaluent la diversité dans le caractère équilibré ou non des arbres. Nous retenons trois définitions différentes de l'équilibre (provenant de [Cha 2012]), chacune se décomposant en sous-intervalle L, M, et H à couvrir.

Leaf-balancedness : étant donné une hauteur h, nous mesurons le taux de feuilles qui sont soit à la profondeur h ou $h-1$.

$$leafb\_rate = \frac{(nombre\ de\ feuilles\ \text{à}\ la\ profondeur\ h\ ou\ h-1)-1}{nombre\ total\ de\ feuilles-1}$$

Il y a obligatoirement au moins une feuille de profondeur égale à la hauteur de l'arbre. Pour que le ratio soit bien défini entre 0 et 1, il faut enlever le compte de cette feuille au numérateur et au dénominateur. Cette propriété est intéressante à mesurer pour des arbres de hauteur strictement supérieure à 2 et dont la taille est supérieure à la hauteur plus un. On en déduit que la taille doit être supérieure à 4. Les arbres de taille inférieure ne devront pas être comptés pour la couverture des cas L, M, H de la métrique. La figure 6.14 présente des arbres avec leur métrique t "leafb_rate" avec à gauche un arbre pas du tout équilibré, à droite un arbre parfaitement équilibré et quelques arbres intermédiaire.

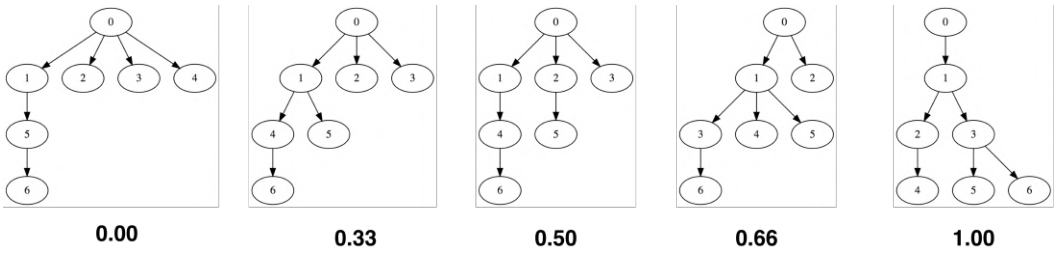


FIGURE 6.14 – Exemple de la mesure d'équilibre *Leaf-balance*.

height-balancedness est le taux de noeuds internes qui introduisent des sous-arbres de hauteur similaire (+/- 1). La métrique *Size-balancedness* est identique à la différence qu'on observe la taille des sous-arbres à la place de la hauteur.

$$heightb\_rate = \frac{nombre\ de\ noeuds\ internes\ avec\ des\ sous-arbres\ de\ hauteur\ similaire-1}{nombre\ total\ de\ noeuds\ internes-1}$$

$$sizeb\_rate = \frac{nombre\ de\ noeuds\ internes\ avec\ des\ sous-arbres\ de\ taille\ similaire-1}{nombre\ total\ de\ noeuds\ internes-1}$$

On considère des noeuds à k enfants, un sous ensemble d'enfants pouvant éventuellement être l'arbre nul de hauteur -1 et de taille 0. Si un noeud n'a qu'un enfant, on le compare à l'arbre nul. Ces deux critères sont intéressants à mesurer pour un arbre de hauteur supérieure à 1 car tous les arbres de hauteur 1 sont trivialement équilibrés selon la hauteur et la taille ("height-balanced" et "size-balanced"). Aussi, il y a au moins un noeud de profondeur $height - 1$ dans l'arbre. Celui-ci introduit un sous arbre trivialement équilibré. Comme pour la métrique "leaf-balancedness", on retire ce cas au numérateur et au dénominateur pour avoir un ratio bien défini entre 0 et 1. Les figures 6.15 et 6.16 montrent quelques exemples ces critères.

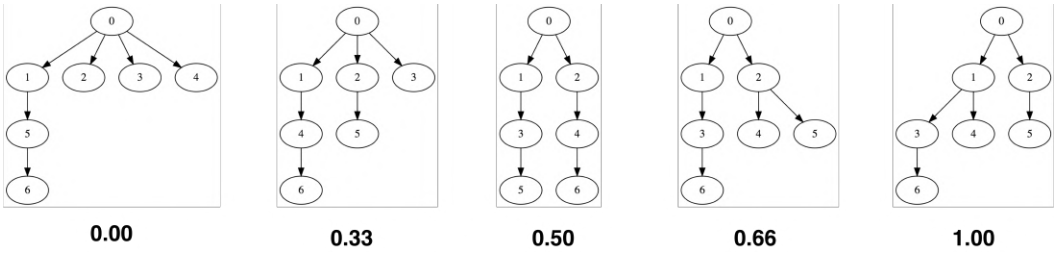


FIGURE 6.15 – Exemple de la mesure d'équilibre *Height-balance*.

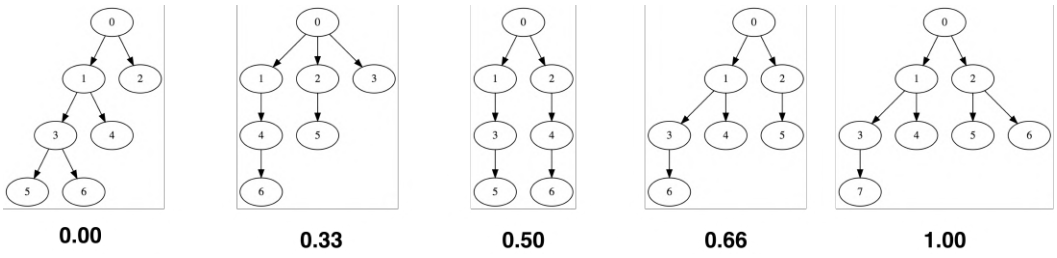


FIGURE 6.16 – Exemple de la mesure d'équilibre *Size-balance*.

Pour exclure les cas de couverture dégénérée et homogénéiser l'évaluation des critères différents d'équilibre, nous les calculons uniquement pour les arbres ayant au moins 5 noeuds et une hauteur supérieure à 2 et dont la taille est supérieure à la hauteur plus un.

6.4 Analyse des résultats

Cette section présente l'analyse de performance de l'outil *TAF* ainsi que la diversité des cas de test générés. L'étape de génération consiste en la génération de 100 cas de tests valides pour chaque cas d'étude. Elle est ensuite répétée 10 fois pour solidifier statistiquement nos mesures. Nous commençons cette section par l'analyse des performances avant de finir sur l'analyse de couverture.

6.4.1 Analyse de performance

La performance de *TAF* dépend fortement de la performance du solveur utilisé. Le bon côté est que le solveur ne traite qu'une partie des données à chaque appel. En effet, notre approche par couches décompose le problème de génération en sous-problèmes. De plus, dans une couche, seuls les paramètres liés à des contraintes doivent être résolus. En revanche, un point négatif de notre approche est que l'ajout de contrainte pour forcer la diversité implique de multiples appels au solveur, ce qui peut être très coûteux. Notre approche est théoriquement intéressante, la question est de voir si les points positifs et les points négatifs s'équilibrent et rendent l'approche utilisable dans la pratique avec un solveur comme Z3.

Le tableau 6.1 indique le temps de génération de 100 cas de test pour chaque étude de cas. Les expériences sont réalisées sur un MacBookPro14,2 2017 double coeur (3,5 GHz Intel Core i7, RAM 16 Go). Les budgets alloués au backtracking et à la diversité sont tous deux fixés à 10. La taille des données générées peut varier dans chaque cas d'essai et la limite supérieure est indiquée dans la deuxième colonne du tableau. Par exemple, pour *Oz*, le nombre de rangées de légumes est compris entre 1 et 100 (nombre d'instances maximal est 100).

TABLE 6.1 – Temps de génération des données avec *TAF* (10 runs, 100 cas de test par run)

Cas d'étude	Nombre d'instances maximal	Tmin	Tmax	Tmedian
Oz	100 rows	16.36 s	18.79 s	17.62 s
Bitmap	100x100 px	636.7 s	1554 s	746.6 s
Tree	100 nodes	>2 h	>2 h	>2 h
	50 nodes	1961 s	2995 s	2664 s
Tax payer	100 taxp.	123.9 s	161.9 s	133.9 s

Nous arrêtons une exécution si elle dure plus de deux heures. La validité des cas de test générés est confirmée par un oracle externe, que nous avons développé pour chaque étude de cas. Il nous a permis de vérifier que les modèles encodent les propriétés voulues et que le processus de génération ne présente pas de bugs.

L'arbre est l'étude de cas la plus difficile pour le solveur. Avec un nombre d'instances maximal à 100, il n'arrive pas à produire les 100 cas de test en deux heures. Il se retrouve vite bloqué sur un arbre trop complexe avec un trop grand nombre de possibilités : c'est l'explosion combinatoire. Dans notre approche de génération par couches, la hauteur globale de l'arbre est d'abord décidée, puis les décisions locales sur les paramètres des noeuds doivent s'y conformer. D'après notre analyse, les longs temps de résolution sont ceux où les valeurs de parent et de profondeur injectées au hasard donnent un problème insatisfiable (unsat), mais le solveur ne le détecte pas et continue de parcourir les possibilités. En prenant une taille maximale

de 50 nœuds, la génération est possible en moins de 50 minutes. Les études de cas les plus faciles sont *Oz* et *Tax payer*, pour lesquels 100 cas tests sont produits avec un temps médian de respectivement 17,62 et 133,9 secondes. Pour *Oz*, la taille des données est réaliste puisque les champs de culture dans lesquels le robot réel opère ont généralement moins de 100 rangées de légumes. En ce qui concerne *Tax payer*, On remarque que les différents individus de la population que nous générons sont faiblement couplés. Pour générer efficacement une grande population, il est donc possible de générer plusieurs petites populations puis de les fusionner.

TABLE 6.2 – Temps de génération des données avec Z3 (10*100 cas de test par run.) sans *TAF* (nombre d'instances fixées)

Cas d'étude	Nombre d'instances	Tmin	Tmax	Tmedian
Oz	100 rows	18.70s	25.23s	20.11s
Bitmap	100x100 px	619.8s	997.8s	727.1s
Tree	100 nodes	340.5s	392.6s	380.8s
	50 nodes	86.61s	91.51s	87.64s
Tax payer	100 taxp.	87.77s	121.5s	95.40s

Le tableau 6.2 est donné dans un but comparatif. Il contient les temps de génération des données pour les différents cas de test avec l'utilisation du solveur z3 mais sans *TAF*. On fixe les nombres d'instances à leurs valeurs maximales pour rendre le modèle statique, il peut alors être résolu directement par le solveur en un seul appel. Le temps de génération est bien plus faible mais les solutions obtenues ne sont pas variées : le solveur retourne toujours la même solution, qui est la première. Par exemple, pour l'arbre, on obtient un arbre plat : tous les noeuds ont la racine pour parent. Ces chiffres permettent de donner un ordre de grandeur sur l'impact qu'a notre approche sur le temps de génération.

Également à des fins de comparaison, nous avons évalué les performances de deux autres stratégies pour la génération de données : une stratégie dite "générer-puis-filtrer", et une deuxième implémentée dans l'outil *PLEDGE*. Pour la premiere stratégie "générer-puis-filtrer", il suffit de supprimer toutes les contraintes des modèles de données et de générer aléatoirement les données. La génération est très rapide car il n'y a aucun appel au solveur. On utilise ensuite un oracle (ce même oracle qui nous permet de vérifier le bon fonctionnement de notre outil) pour filtrer les données en fonction de leur validité. Un premier cycle de génération de 2 heures n'a pas permis de produire de données valides, sauf pour le cas d'étude *Tax payer*, qui a produit 46 données valides (sur 33 290 essais). Les essais supplémentaires pour *Tax payer* n'ont jamais réussi à obtenir les 100 cas valides désirés en deux heures (le meilleur essai a obtenu 55 cas). En outre, tous les cas valides ne consistaient qu'en des populations d'un seul contribuable. La stratégie "générer puis filtrer" est donc très inefficace pour nos études de cas.

La deuxième méthode est celle implémentée dans l'outil

PLEDGE [Soltana 2020], elle représente une stratégie similaire à la nôtre. *PLEDGE* utilise le même solveur que *TAF* (c'est-à-dire Z3) mais l'exploite d'une manière différente. La stratégie de génération combine une recherche métaheuristique et le solveur SMT dans le but d'utiliser chaque technique là où elle est la plus efficace. Elle distribue les tâches à l'une ou l'autre des techniques quand cela est possible. Au contraire, *TAF* fait en permanence un usage combiné de générateurs aléatoires et d'un solveur de contraintes. Nous avons modélisé *Oz* et *Tax payer* en UML pour les fournir à *PLEDGE*. Le tableau 6.3 indique les temps de génération correspondants. La génération est nettement plus longue qu'avec *TAF* malgré la réduction du nombre d'instances pour *Tax payer*. Les 2 autres cas d'étude n'ont pas été modélisés dans *PLEDGE* car l'outil est encore un prototype et admet donc de nombreux problèmes. De plus, *PLEDGE* n'est pas un outil open source et nous avons du interagir avec l'un des développeurs (G. SOLTANA) pour le modèle de *Oz*.

TABLE 6.3 – Temps de génération des données avec *PLEDGE* (10 runs, 100 cas de test par run)

Cas d'étude	Nombre d'instances maximal	Tmin	Tmax	Tmedian
Oz	100 rows	1073 s	1327 s	1108 s
Tax payer	100 taxp	>2 h	>2 h	>2 h
	50 taxp	1640 s	2651 s	1804 s

Pour conclure, le solveur Z3 est suffisamment efficace pour que notre approche de résolution avec diversité proposée soit utilisable dans la pratique. Malgré les multiples appels au solveur, le temps nécessaire pour générer des données contraintes de taille moyenne reste acceptable. Notre approche nous a permis d'aborder quatre études de cas pour lesquelles une stratégie plus simple de type "générer puis filtrer" a échoué. La comparaison avec un autre outil également basé sur une méthode hybride, mêlant méta-heuristique et solveur SMT, montre que *TAF* peut être compétitif en termes de performance.

6.4.2 Analyse de couverture

Le tableau 6.4 indique le nombre total de cas à couvrir pour chacun des cas d'étude ainsi que les résultats de la couverture. Une liste exhaustive des cas à couvrir est disponible en annexe.

Par exemple, pour *Oz*, 100% des 51 cas sont couverts par les 10 ensembles de données générées. La couverture complète est atteinte après qu'un nombre médian de 16,5 données ait été généré (8 données suffisent dans le meilleur cas alors que 49 sont nécessaires dans le pire cas). Comme on peut le voir, *TAF* parvient à fournir une couverture élevée pour toutes les études de cas.

Nous considérons le cas d'étude *Tree* comme le plus difficile à couvrir. Les propriétés mesurées sur la structure arborescente ne ressortent qu'indirectement de l'instanciation des paramètres. Il n'y a pas de notion d'équilibre dans le modèle, et pourtant *TAF* a réussi à générer des arbres avec des mesures d'équilibre diverses. Le seul cas d'étude pour lequel la couverture complète n'est pas atteinte est celle de *Bitmap*. Les cas manquants concernent des valeurs élevées pour le pixel le plus sombre d'une image (ce qui donne une image presque entièrement blanche), et des valeurs faibles pour le pixel le plus clair (image presque noire). De tels cas extrêmes sont très peu probables dans le processus de génération automatisée. Il est préférable de les couvrir en créant des cas tests partiellement instanciés dont le pixel inférieur gauche ou le pixel supérieur droit ont les valeurs souhaitées.

TABLE 6.4 – Couverture pour 100 cas de test générés par *TAF*

	# cases to cover	% coverage	sample size to reach final coverage
Oz	51 cases	100%	8-49 (median 16.5)
Bitmap	28 cases	89.3%-96.4%	10-80 (median 28)
Tree	18 cases	100%	8-47 (median 22.5)
Tax payer	70 cases	100%	2-8 (median 2.5)

En comparaison, *PLEDGE* ne fournit que très peu de diversité dans les données qu'il génère. Les dix ensembles de données générées fournissent une couverture de 52% pour *Oz* et 51% pour *Tax payer*. L'outil gère bien la génération d'un nombre variable d'instances des classes, mais leurs attributs prennent pour valeur la solution la plus simple retournée par le solveur. Par exemple, pour *Oz*, les champs générés ont un nombre variable de rangée de légumes, mais toutes ont une longueur identique de 10 mètres (le minimum de l'intervalle). Autre exemple, le bruit sur la position des légumes est nul, etc. Le même constat est établi pour les données générées pour le cas d'étude *Tax payer*, la population de contribuables manque de diversité. Par exemple, tous les contribuables sont nés en 1920 (la date de naissance minimale), leurs adresses ne sont jamais en Belgique ou en Allemagne (BE ou DE). Le manque de diversité est reconnu par les auteurs [Soltana 2020] : ils affirment que la diversité peut être efficacement mise en oeuvre en actualisant dynamiquement les contraintes OCL avec de nouvelles inégalités. Nous pensons que cette solution n'est pas aussi satisfaisante que d'avoir la gestion de la diversité directement intégrée à l'outil. En outre, l'ajout de contraintes en OCL n'offre qu'une diversité limitée. Par exemple, le fait de forcer un paramètre vers un sous-ensemble de valeurs conduira Z3 à toujours sélectionner la valeur minimale dans ce sous-ensemble. Pour *Oz*, nous avons essayé d'ajouter une contrainte OCL selon laquelle la première et la dernière rangée ne doivent pas avoir la même longueur : les rangées de légumes générées ont alors toutes une longueur de 10 mètres, sauf la dernière qui a une longueur de 10,5 mètres.

Pour conclure, *TAF* a réussi à générer des données contraintes tout en assurant une couverture élevée de l'espace de données. À notre connaissance, il n'existe aucun outil comparable qui offre des fonctionnalités similaires avec un solveur SMT.

6.5 Conclusion

Ce chapitre présente les analyses de performance et de couverture, et conclut la partie sur notre méthode de génération de données structurées variées et son prototypage dans *TAF*. Cet outil génère automatiquement des entrées de test à partir d'un modèle de données de haut niveau basé sur XML. Il introduit une nouvelle façon de générer diverses instances d'éléments de données dans des structures de données de taille variable. La caractéristique principale de *TAF* est sa fonctionnalité "résoudre avec diversité", qui mélange la résolution de contraintes avec de l'échantillonnage aléatoire. Elle attribue des valeurs aléatoires à autant de paramètres que possible avant de résoudre les contraintes, afin de forcer de la diversité dans les solutions obtenues. La fonction fait partie d'un algorithme clé qui utilise une approche de génération par couches avec backtracking. La décomposition descendante en couches garantit que les éléments de données ne sont générés qu'une fois que leur nombre d'instances est décidé.

Lors de nos expérimentations, nous avons pu identifier les limites de *TAF*. Premièrement, la structure arborescente des modéles *TAF-XML* n'est pas directement compatible avec les concepts des modèles orientés objet. Il est difficile de produire un modèle équivalent pour *TAF* à un modèle complexe orienté objet. Deuxièmement, *TAF* repose sur une utilisation intensive du solveur. Il est donc totalement dépendant de ses performances.

TAF a été appliqué sur quatre études de cas. Il a permis de générer avec succès des données valides et variées pour chacune d'entre elles. La comparaison avec un autre outil utilisant le même solveur a montré que *TAF* est compétitif par rapport au temps de génération, tout en permettant une bien meilleure couverture de l'espace des données. Au delà de ce travail, *TAF* a été utilisé pour les tests en simulation du nouveau robot de Naïo (*Dino*), et est actuellement déployé dans une autre application industrielle avec l'entreprise Allemande SICK.

TAF est un logiciel libre disponible en ligne [Robert 2020b]. L'ensemble des codes sources des exemples et des résultats de ce chapitre sont également téléchargeables sur le dépot [Robert 2020a].

Conclusion générale et perspectives

Pour conclure ce manuscrit dédié à la génération et à l'analyse des tests pour les systèmes autonomes, un bilan de nos travaux est proposé, il s'ensuit une mise en avant des principales contributions. Pour finir nous ouvrons sur les perspectives possibles pour la poursuite de nos travaux.

Bilan

Les systèmes autonomes sont de plus en plus présents dans un contexte non académique. La variété des applications et les tâches de plus en plus critiques qui leur sont déléguées induisent un effort de validation important. Les tests sur le terrain sont largement utilisés, mais présentent des coûts et des difficultés de mise en oeuvre. La simulation, associée à une génération de cas de test de bonne qualité, devrait permettre de réaliser une partie des tests avec un coût bien moindre, un danger nul et une meilleure diversité des cas de test.

La première partie du manuscrit étudie le test en simulation au travers du cas d'étude de *Oz*, un robot agricole développé par la société Naïo Technologies. La contribution principale vient de l'analyse comparative que nous avons effectuée entre les résultats de notre campagne de test en simulation et ceux de la campagne de test sur le terrain menée par Naïo. A notre connaissance, il n'existe aucune comparaison de ce type dans la littérature, pour un système réel en cours de développement dans l'industrie. Nos résultats s'avèrent très encourageants. Malgré la fidélité imparfaite de la simulation, les tests virtuels permettent de reproduire la plupart des défaillances observées sur le terrain. Ils révèlent également une nouvelle faute, qui n'avait pas été trouvée par l'industriel. Ainsi, notre étude confirme l'efficacité de ce type de tests, encore trop sous-exploités dans la validation des robots autonomes. La simulation, associée à une génération de cas de test diversifiés, devrait permettre de réduire l'effort de validation sur le terrain.

Automatiser la génération des cas de tests reste cependant problématique. Pour des systèmes autonomes tels que *Oz*, le domaine d'entrée inclut les mondes virtuels et les missions considérés par le test. Les données correspondantes à générer sont structurées, comportent des éléments de nombre et de type variables, et doivent satisfaire des propriétés sémantiques. Dans le cas de *Oz*, nous avons développé un générateur spécifique : les procédés pour construire des données valides sont codés en dur dans ce générateur. Cette solution spécifique n'est clairement pas satisfaisante. D'une part, l'effort de développement est à recommencer pour toute nouvelle application. D'autre part, le modèle de données est susceptible d'évoluer au fur et à mesure que l'on acquiert de l'expérience sur les environnements pertinents pour le test du système, ce qui induit des problèmes de maintenance du code de

génération. Une solution plus satisfaisante doit être générique et capable de produire des données directement à partir d'un modèle déclaratif.

La deuxième partie de ce manuscrit s'attaque à ce problème. Elle part du constat qu'il n'existe pas de solution sur étagère permettant de générer des données structurées avec des contraintes et de la diversité à l'échelle d'un cas d'étude industriel comme le robot *Oz*. En particulier, les approches basées sur la résolution de contraintes gèrent mal les structures de données de taille et de type variables, et n'offrent pas de diversité. La solution que nous proposons lève ces limitations. Notre outil *TAF* prend en entrée le modèle de données, et met en oeuvre une approche de génération hybride qui combine échantillonnage aléatoire et résolution de contraintes. Il met dynamiquement à jour les ensembles de contraintes lorsque des instances de données sont créées ou supprimées. Il injecte des valeurs aléatoires dans les requêtes envoyées au solveur, pour forcer de la diversité.

Nous avons appliqué *TAF* à quatre cas d'étude venant de domaines d'application variés, et l'avons comparé avec deux approches alternatives : "générer puis filtrer" et *PLEDGE*. Les résultats montrent que notre outil est compétitif en termes de performance, tout en couvrant bien l'espace des solutions. *TAF* offre aussi des fonctionnalités de complétion de test, qui peuvent être exploitées pour cibler des parties de cet espace. Il peut être utilisé non seulement pour la génération d'environnements de test virtuels, mais aussi pour produire tout autres types de données riches, comme par exemple des trames réseaux.

Contributions principales

Les contributions des travaux présentées dans ce manuscrit sont ainsi les suivantes :

— la modélisation de cas de test virtuels basée sur une grammaire attribuée (section 2.2) ;

— la spécification d'un oracle à partir de grandes classes de propriétés (travaux conjoints présentés sections 2.3 et surtout 2.5) ;

— une méthode pour analyser l'impact de l'indéterminisme sur les résultats (section 3.2) ;

— une analyse comparative des résultats des campagnes en simulation et sur le terrain (section 3.4) ;

— des recommandations pour la mise en oeuvre de tests en simulation (section 3.5) ;

— un outil de génération de données diverses et avec contraintes (partie II) ;

— une méthode de génération hybride mêlant échantillonage aléatoire et résolution de contraintes SMT (section 5.2) ;

— une méthode pour gérer dynamiquement les contraintes, lorsque des instances de données sont créées ou supprimées (section 5.2).

Perspectives de recherche

Lors de ces travaux, nous avons pu identifier des pistes pour aller au delà de ces travaux. Ces différentes perspectives sont détaillées ci-après :

1. Notre plate-forme de simulation permet uniquement de générer des environnements statiques. Autrement dit, les positions de tous les objets sont fixes à l'exception de celle du système sous test. L'ajout d'objets dynamiques pour le test est une perspective intéressante. Il rendrait possible des scénarios qu'on ne peut pas exécuter actuellement. D'un point de vue pratique, la modélisation d'objets dynamiques n'est pas simple. Il ne suffit pas de définir une trajectoire et une courbe de vitesse. En effet, l'indéterminisme du système sous test implique la création de déclencheurs pour synchroniser les mouvements des éléments dynamiques et ainsi tester un scénario en particulier. La trajectoire de ces objets peut également dépendre de celle du système sous test. Pour pouvoir intégrer à notre plate-forme des objets dynamiques répondant à nos contraintes, il faut pouvoir les générer à la volée. Des travaux sur la modélisation de ces objets et de leur déclencheur seraient une continuation intéressante des travaux présentés en première partie de ce manuscrit.
2. L'oracle se base sur un ensemble de propriétés défini à partir de cinq grandes classes de propriétés identifiées lors de travaux précédents. Pendant l'implémentation de l'oracle pour le robot *Oz* ou pour le robot *REMI*, nous avons remarqué que chacune des propriétés doit être traitées différemment. Elles sont dépendantes du contexte. Certaines dépendent de la phase de mission ou des états et des choix précédents du système. Pour faciliter les implémentations des futurs oracles et tendre vers une solution plus générique, il peut être intéressant de travailler sur la formalisation du contexte relatif aux propriétés.
3. Nos travaux ont prouvé l'efficacité du test en simulation sur un cas d'étude industriel. Malgré tout, quelques défaillances n'ont pas pu être observées à cause d'un réalisme limité. Pour des questions de performance, de coût de développement et de maintenabilité, beaucoup d'aspects de la simulation sont simplifiés. Un exemple frappant est la gestion de la friction pour l'interaction entre les roues et le sol. Nos partenaires de Naïo avaient pour stratégie initiale d'augmenter la fidélité du simulateur le plus possible. La complexité de la tâche et les résultats satisfaisants fournis par le simulateur actuel ont fait évoluer cette stratégie. Aujourd'hui, un simulateur très basse fidélité a été développé par Naïo. Il serait intéressant de poursuivre la comparaison avec les tests sur le terrain, en considérant les différentes versions de simulateurs et leur niveau de fidélité. Le but serait d'identifier les fonctionnalités nécessaires à la détection de certaines défaillances et ainsi de mieux appréhender le niveau de fidélité minimum requis pour une campagne de test efficace.
4. La deuxième partie de ce manuscrit présente notre outil de génération *TAF*. Le langage dédié à la modélisation pour *TAF* est décrit sans le chapitre 4. Il se base sur XML car c'est un format standard pour l'échange de donnée.

Il a l'avantage d'être facilement lisible par les machines et les utilisateurs. En conséquence, nos modèles XML-TAF sont spécifiques à notre outil, ils ne sont pas basés sur un langage de modélisation standard. Une amélioration serait le support des modèles basés sur UML / OCL (ou au moins un sous-ensemble). Cela permettrait une prise en main plus simple de l'outil mais aussi une simplification de la spécification de certaines contraintes pour l'utilisateur. Notamment, les contraintes découlant de la structure du diagramme de classes UML s'avèrent actuellement lourdes à exprimer dans le langage d'entrée de *TAF*.

5. En interne, la gestion du solveur est très dépendante du binding entre python et le solveur z3, un changement de solveur est alors difficile et demande beaucoup de modification dans le code. Comme le solveur est une ressource externe, il devrait idéalement être interchangeable. Pour cela, une amélioration serait de traduire les contraintes dans un langage plus standard après leur traitement par *TAF*. Des recherches sur les possibilités qu'offrent SMTLIB ([Barrett 2016]) seraient un premier pas dans cette direction.
6. Enfin, les fonctionnalités de *TAF* ouvrent la possibilité de mettre en oeuvre des stratégies plus sophistiquées que l'échantillonnage aléatoire. Par exemple, *TAF* pourrait être utilisé dans le cadre de tests basés sur des métaheuristiques de recherche, ou encore des tests combinatoires qui explorent des combinaisons de valeurs de paramètres. Dans ce cas, les services de génération et de complétion de tests de *TAF* seraient appelés par un programme frontal qui mettrait en oeuvre la stratégie souhaitée. Le format d'entrée XML de *TAF* devrait faciliter cette mise en oeuvre, se prêtant bien à la manipulation de contenus de fichiers. Le programme frontal pourrait ainsi facilement manipuler le modèle des données (pour ajouter/enlever des contraintes, changer les générateurs par défaut des types de données) et/ou les tests partiellement instanciés (pour changer les valeurs proposées). De telles implémentations de stratégies complexes basées sur *TAF* sont actuellement à l'étude au LAAS, dans le cadre d'une nouvelle thèse.

Annexe A

Annexe

Ci-après la liste exhaustive des diffèrent cas à couvrir pour les différents cas d'étude. Les pourcentage de couverture présenté section 6.4 sont calculés à partir de cette liste. Les résultats disponibles sur le git [Robert 2020a] suivent l'indexation des différents cas.

A.1 Cas à couvrir pour le cas d'étude *Oz*

field\vegetables :
1. cabbage
2. leek
field\row.nb_instances :
3. [0, 100/3[
4. [100/3, 200/3]
5.]200/3, 255]
field\row[]\length :
6. [10, 40[
7. [40, 70]
8.]70, 100]
field\row[]\noise_X :
9. [0, 5/3[
10. [5/3, 10/3]
11.]10/3, 5]
field\row[]\noise_Y :
12. [0, 5/3[
13. [5/3, 10/3]
14.]10/3, 5]
field\row[]\disappearance_probability :
15. [0, 10[
17.]20, 30]
16. [10, 20]
field\row[]\vegetable_density :
18. [1, 2]
19. [3]
20. [4, 5]
Ratio entre les rangées successives (contrainte "interval"), **field\row[i]\length / field\row[i-1]\length**. Ce critère s'applique uniquement aux champs comportant

un minimum de deux rangées :
21. [0.9, 0.9 + 0.2/3[
22. [0.9 + 0.2/3, 0.9 + 0.4/3]
23.]0.9 + 0.4/3, 1.1]
Ratio entre les deux rangées extérieures du champ (contraite “interval_2”), **field\row[0]\length / field\row[nb_instances - 1]\length**. Ce critère s'applique uniquement aux champs comportant un minimum de trois rangées. On s'assure ainsi que les rangées extérieures ne sont pas consécutives :
24. [0.9, 2.9/3[
25. [2.9/3, 3.1/3]
26.]3.1/3, 1.1]
On s'assure également que les rangées extérieures sont de longueurs variés dans les différents champs générés **field\row[0]\length** :
27. [10.0, 40.0[
28. [40.0, 70.0]
29.]70.0, 100.0[
field\weed_area\grass_density :
30. [0, 5/3[
31. [5/3, 10/3]
32. [10/3, 5]
field\inner_track_width\gap :
33. [55, 55 + 110/3[
34. [55 + 110/3, 55 + 220/3]
35. [55 + 220/3, 165]
mission\two_pass :
36. true
37. false
mission\is_first_track_outer :
38. true
39. false
mission\final_track_outer :
40. true
41. false
mission\is_track_side_at_left :
42. true
43. false
mission\is_first_uturn_right_side :
44. true
45. false
terrain\heightmap\roughness :
46. [0.0, 1/3[
47. [1/3, 2/3]
48.]2/3, 1.0]
terrain\heightmap\persistence :

49. [0.0, 0.7/3[
50. [0.7/3, 1.4/3]
51.]1.4/3, 0.7]

A.2 Cas à couvrir pour le cas d'étude *Tax payer*

tax_payer.nb_instances :
1. [1, 1 + 99/3[
2. [1 + 99/3, 1 + 198/3]
3.]1 + 198/3, 100[

tax_payer[]\birth_year :
4. [1920, 1920 + 100/3[
5. [1920 + 100/3, 1920 + 200/3]
6.]1920 + 200/3, 2020]

tax_payer[]\disability_rate :
7. [0, 1/3[
8. [1/3, 2/3]
9.]2/3, 1[

tax_payer[]\disability_type :
10. None
11. Vision
12. A

tax_payer[]\is_resident :
13. true
14. false

tax_payer[]\address.nb_instances :
15. 1
16. 2
17. 3

tax_payer[]\address :
18. LU
19. FR
20. BE
21. DE
22. OTHER

tax_payer[]\child.nb_instances :
23. 0
24. 1
25. 2
26. 3

tax_payer[]\child[]\birth_year :
27. [1920, 1920 + 100/3[

28. [1920 + 100/3, 1920 + 200/3]
29.]1920 + 200/3, 2020]

tax_payer[]\child[]\disability_rate :
30. [0, 1/3[
31. [1/3, 2/3]
32.]2/3, 1[

tax_payer[]\child[]\disability_type :
33. None
34. Vision
35. A

tax_payer[]\child[]\address.nb_instances :
36. 1
37. 2
38. 3

tax_payer[]\child[]\address :
39. LU
40. FR
41. BE
42. DE
43. OTHER

tax_payer[] income_pension.nb_instances :
44. 0
45. 1
46. 2
47. 3

tax_payer[]\income_employment.nb_instances :
48. 0
49. 1
50. 2
51. 3

tax_payer[]\income_other.nb_instances :
52. 0
53. 1
54. 2
55. 3

tax_payer[]\income_pension[]\is_local :
56. true
57. false

tax_payer[]\income_employment[]\is_local :
58. true
59. false

tax_payer[]\income_other[]\is_local :
60. true
61. false

Pour la contrainte "income" on a income_pension.nb_instances + income_employment.nb_instances + income_other.nb_instances :
62. 1
63. 2
64. 3
Pour la contrainte "C3" on compte le nombre d'adresse au Luxembourg :
65. No address in Luxembourg
66. Address in Luxembourg and elsewhere
67. All addresses in Luxembourg
Pour "C4" :
68. income_pension\is_local is true and at least one address in Luxembourg
69. income_employment\is_local is true and at least one address in Luxembourg
70. Income_othert\is_local is true local and at least one address in Luxembourg

A.3 Cas à couvrir pour le cas d'étude *Bitmap*

row.nb_instances :
1. [10, 10 + 90/3[
2. [10 + 90/3, 10 + 180/3]
3.]10 + 180/3, 50[
row[]\pixel[]\gray :
4. [0, 255/3[
5. [255/3, 510/3]
6.]510/3, 255]
padding :
7. 0
8. 1
9. 2
10. 3
min(row[]\pixel[]\gray) :
11. [0, 255/3[
12. [255/3, 510/3]
13.]510/3, 255]
max(row[]\pixel[]\gray) :
14. [0, 255/3[
15. [255/3, 510/3]
16.]510/3, 255]
horizontal_border (row[nb_instances - 1]\pixel[]\gray - row[0]\pixel[]\gray) :
17. [0, 255/3[
18. [255/3, 510/3]
19.]510/3, 255]

vertical_border (row[]\pixel[nb_instances - 1]\gray - row[]\pixel[0]\gray) :
20. [0, 255/3[
21. [255/3, 510/3]
22.]510/3, 255]
horizontal_interval (row[]\pixel[i+1]\gray - row[]\pixel[i]\gray) :
23. [0, 255/3[
24. [255/3, 510/3]
25.]510/3, 255]
vertical_interval (row[i+1]\pixel[]\gray - row[i]\pixel[]\gray) :
26. [0, 255/3[
27. [255/3, 510/3]
28.]510/3, 255]

A.4 Cas à couvrir pour le cas d'étude *Tree*

node.nb_instances :
1. [1, 1 + 49/3[
2. [1 + 49/3, 1 + 98/3]
3.]1 + 98/3, 50]
height :
4. [1, 1 + 48/3[
5. [1 + 48/3, 1 + 96/3]
6.]1 + 96/3, 49]
height_vs_size :
7. [0.0, 1/3[
8. [1/3, 2/3]
9.]2/3, 1.0]
leaf_balanced :
10. [0.0, 1/3[
11. [1/3, 2/3]
12.]2/3, 1.0]
height_balanced :
13. [0.0, 1/3[
14. [1/3, 2/3]
15.]2/3, 1.0]
size_balanced :
16. [0.0, 1/3[
17. [1/3, 2/3]
18.]2/3, 1.0]

Bibliographie

[Ali 2013] S. Ali, M.Z. Iqbal, A. Arcuri et L. Briand. *Generating Test Data from OCL Constraints with Search Techniques.* IEEE Transactions on Software Engineering, vol. 39, pages 1376–1402, 2013. (Cité en pages 31 et 124.)

[Alur 2011] R. Alur. *Formal verification of hybrid systems.* Dans Ninth ACM International Conference on Embedded Software (EMSOFT), Taipei, Taiwan, pages 273–278, 2011. (Cité en page 19.)

[Amman 2016] P. Amman et J. Offutt. Dans Introduction to software testing, second edition, ISBN 9781107172012, 2016. (Cité en page 10.)

[Araiza-Illan 2016] D. Araiza-Illan, A. Pipe et K. Eder. *Model-based Test Generation for Robotic Software : Automata versus Belief-Desire-Intention Agents.* ArXiv, vol. abs/1609.08439, 2016. (Cité en page 20.)

[Arnold 2013] J. Arnold et R. Alexander. *Testing autonomous robot control software using procedural content generation.* Dans 32nd International Conference on Computer Safety, Reliability, and Security (SAFECOMP), Toulouse, France, pages 33–44, 2013. (Cité en pages 16, 18, 21, 25 et 26.)

[Avizienis 2004] A. Avizienis, J-C Laprie, B. Randell et C. Landwehr. *Basic Concepts and Taxonomy of Dependable and Secure Computing.* IEEE Transactions on Dependable and Secure Computing, vol. 1, page 11–33, 2004. (Cité en page 10.)

[Bach 2017] J. Bach, J. Langner, S. Otten, M. Holzäpfel et E. Sax. *Test Scenario Selection for System-Level Verification and Validation of Geolocation-Dependent Automotive Control Systems.* Dans IEEE International Conference on Engineering, Technology and Innovation/ International Technology Management Conference (ICE/ITMC), Madeira Island, Portugal, volume 23, 2017. (Cité en page 24.)

[Bakhirkin 2018] A. Bakhirkin, T. Ferrère, T. Henzinger et D. Nickovic. *The First-Order Logic of Signals.* Dans International Conference on Embedded Software (EMSOFT), Torino, Italy, 2018. (Cité en page 22.)

[Banks 2006] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer et G. Vigna. *SNOOZE : Toward a Stateful NetwOrk prOtocol fuzZEr.* Dans 9th International Conference on Information Security, Samos, Greece, pages 343–358, 2006. (Cité en page 28.)

[Barr 2015] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz et S. 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 19.)

[Barrett 2016] C. Barrett, P. Fontaine et C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB).* `https://www.SMT-LIB.org`, 2016. (Cité en page 154.)

[Ben Abdessalem 2018] R. Ben Abdessalem, S. Nejati, L. Briand et T. Stifter. *Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms*. Dans IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, pages 1016–1026, 2018. (Cité en page 21.)

[Benveniste 1991] A. Benveniste, P. Le Guernic et C. Jacquemot. *Synchronous programming with events and relations : the SIGNAL language and its semantics*. Science of Computer Programming, vol. 16, no. 2, pages 103 – 149, 1991. (Cité en page 14.)

[Berry 1992] G. Berry et G. Gonthier. *The ESTEREL Synchronous Programming Language : Design, Semantics, Implementation*. Sci. Comput. Program., vol. 19, no. 2, pages 87–152, 1992. (Cité en page 14.)

[Boeing 2007] A. Boeing et T. Bräunl. *Evaluation of real-time physics simulation systems*. Dans 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia (GRAPHITE), Perth, Australia, 2007. (Cité en page 17.)

[Bokc 2007] T. Bokc, M. Maurer et G. Farber. *Validation of the Vehicle in the Loop (VIL) ; A milestone for the simulation of driver assistance systems*. Dans IEEE Intelligent Vehicles Symposium, Istanbul, Turkey, pages 612–617, 2007. (Cité en page 16.)

[Cabot 2007] J. Cabot, R. Clarisó et D. Riera. *UMLtoCSP : A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming*. Dans Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE), Atlanta, Georgia, USA, page 547–548, 2007. (Cité en page 128.)

[Cantenot 2014] J. Cantenot, F. Ambert et F. Bouquet. *Test Generation with Satisfiability Modulo Theories Solvers in Model-Based Testing*. Software Testing, Verification & Reliability, vol. 24, no. 7, page 499–531, 2014. (Cité en pages 29 et 31.)

[car 2020] *CARLA*. `https://carla.org/`, 2020. Accessed : 2021-05-22. (Cité en page 18.)

[Cekan 2017] O. Cekan et Z. Kotasek. *A Probabilistic Context-Free Grammar Based Random Test Program Generation*. Dans Euromicro Conference on Digital System Design (DSD), Vienna, Austria, pages 356–359, 2017. (Cité en pages 28, 29 et 128.)

[Cha 2012] Sung-Hyuk Cha. *On Integer Sequences Derived from Balanced K-Ary Trees*. Dans Proceedings of the 6th WSEAS International Conference on Computer Engineering and Applications, and Proceedings of the 2012 American Conference on Applied Mathematics (AMERICAN-MATH 2012/CEA 2012), Harvard, Cambridge, Massachusetts, USA, page 377–381, 2012. (Cité en page 144.)

[Chen 2015] T. Chen. *Metamorphic Testing : A Simple Method for Alleviating the Test Oracle Problem.* Dans Proceedings of the 10th International Workshop on Automation of Software Test (AST), Florence, Italy, pages 53–54, 2015. (Cité en page 22.)

[Claessen 2000] K. Claessen et J. Hughes. *QuickCheck : A Lightweight Tool for Random Testing of Haskell Programs.* Dans 5th International Conference on Functional Programming (ICFP), Montreal, Canada, 2000. (Cité en pages 28 et 128.)

[Coppit 2005] D. Coppit et J. Lian. *Yagg : An easy-to-use generator for structured test inputs.* Dans 20th IEEE/ACM International Conference on Automated Software Engineering (ASE), Long Beach, California, USA, pages 356–359, 2005. (Cité en pages 28, 29, 50 et 128.)

[de Moura 2008] L. de Moura et N. Bjørner. *Z3 : An Efficient SMT Solver.* Dans Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary, pages 337–340, 2008. (Cité en page 92.)

[DIS 1998] DIS. *IEEE Standard for Distributed Interactive Simulation – Application Protocols*, 1998. (Cité en page 14.)

[Duncan 1981] A.G. Duncan et J.S. Hutchison. *Using Attributed Grammars to Test Designs and Implementations.* Dans Proceedings of the 5th International Conference on Software Engineering (ICSE), San Diego, California, USA, page 170–178, 1981. (Cité en page 28.)

[Echeverria 2011] G. Echeverria, N. Lassabe, A. Degroote et S. Lemaignan. *Modular open robots simulation engine : Morse.* Dans IEEE International Conference on Robotics and Automation (ICRA), Shanghai, China, pages 46–51, 2011. (Cité en page 17.)

[Feldt 2013] R. Feldt et S. Poulding. *Finding test data with specific properties via metaheuristic search.* Dans IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), Pasadena, California, USA, pages 350–359, 2013. (Cité en pages 28 et 128.)

[Geyer 2014] S. Geyer, M. Baltzer, B. Franz, S. Hakuli, M. Kauer, M. Kienle, S. Meier, T. Weissgerber, K. Bengler, R. Bruder, F. Flemisch et H. Winner. *Concept and development of a unified ontology for generating test and use-case catalogues for assisted and automated vehicle guidance.* IET Intelligent Transport Systems, vol. 8, no. 3, pages 183–189, 2014. (Cité en page 25.)

[Gietelink 2006] O. Gietelink, J. Ploeg, B. De Schutter et M. Verhaegen. *Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations.* Vehicle System Dynamics, vol. 44, no. 7, pages 569–590, 2006. (Cité en page 16.)

[Guiochet 2017] J. Guiochet, M. Machin et H. Waeselynck. *Safety-critical advanced robots : A survey.* Robotics and Autonomous Systems, vol. 94, pages 43–52, 2017. (Cité en pages 1 et 12.)

[Halbwachs 1991] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud. *The synchronous data flow programming language LUSTRE*. Proceedings of the IEEE, vol. 79, no. 9, pages 1305 – 1320, 1991. (Cité en page 14.)

[Hallerbach 2018] S. Hallerbach, Y. Xia, U. Eberle et F. Koester. *Simulation-Based Identification of Critical Scenarios for Cooperative and Automated Vehicles*. Dans WCX World Congress Experience, rapport technique SAE, 2018. (Cité en page 22.)

[Helle 2016] P. Helle, W. Schamai et C. Strobel. *Testing of Autonomous Systems - Challenges and Current State-of-the-Art*. Dans 26th Annual INCOSE International Symposium (IS), Edinburg, Scotland, UK, 2016. (Cité en page 12.)

[Hereau 2020] A. Hereau, K. Godary-Dejean, J. Guiochet, C. ROBERT, T. Claverie et D. Crestani. *Testing an Underwater Robot Executing Transect Missions in Mayotte*. Dans 21st Towards Autonomous Robotic Systems Conference (TAROS), virtual, Nottingham, UK, 2020. (Cité en pages 1, 56 et 57.)

[HLA 2010] HLA. *IEEE Standard for Modeling and Simulation (M&S) High LevelArchitecture (HLA) – Federate Interface Specification*, 2010. (Cité en page 14.)

[Jackson 2000] D. Jackson. *Automating First-Order Relational Logic*. Dans ACM Symposium on the Foundations of Software Engineering (SIGSOFT), San Diego, California, USA, page 130–139, 2000. (Cité en pages 31 et 128.)

[Jackson 2019] D. Jackson. *Alloy : a language and tool for exploring software designs*. Dans Communications of the ACM, volume 62, pages 66–76, 2019. (Cité en page 28.)

[Kempka 2015] J. Kempka, P. McMinn et D. Sudholt. *Design and Analysis of Different Alternating Variable Searches for Search-Based Software Testing*. Theoretical Computer Science, vol. 605, pages 1–20, 2015. (Cité en pages 31, 123 et 124.)

[Kifetew 2014] F.M. Kifetew, R. Tiella et P. Tonella. *Combining Stochastic Grammars and Genetic Programming for Coverage Testing at the System Level*. Dans Search-Based Software Engineering (SSBSE), Fortaleza, Brazil, pages 138–152, 2014. (Cité en pages 30 et 128.)

[Kifetew 2017] F.M. Kifetew, R. Tiella et P. Tonella. *Generating valid grammar-based test inputs by means of genetic programming and annotated grammars*. Empirical Software Engineering, vol. 22, no. 2, pages 928–961, 2017. (Cité en pages 28, 29, 30, 50 et 128.)

[Klueck 2018] F. Klueck, Y. Li, M. Nica, J. Tao et F. Wotawa. *Using Ontologies for Test Suites Generation for Automated and Autonomous Driving Functions*. Dans IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Memphis, Tennessee, USA, pages 118–123, 2018. (Cité en page 25.)

[Koenig 2004] N. Koenig et A. Howard. *Design and use paradigms for gazebo, an open-source multi-robot simulator*. Dans IEEE/RJS 17th International

Conference on Intelligent Robots and Systems (IROS), Sendai, Japan, volume 3, pages 2149–2154, 2004. (Cité en page 17.)

[Lamprecht 2010] A. Lamprecht et T. Ganslmeier. *Simulation Process for Vehicle Applications Depending on Alternative Driving Routes Between Real-World Locations*. Dans Advanced Microsystems for Automotive Applications, pages 377–386, 2010. (Cité en page 24.)

[Laplace 1814] P.S. Laplace. *Essai philosophique sur les probabilités*, 1814. (Cité en page 13.)

[Laprie 1996] J-C Laprie, J. Arlat, J-P Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J-C Fabre, H. Guillermain, M. Kaâniche, K. Kanoun*et al. Guide de la sûreté de fonctionnement*. Toulouse : Cépaduès, 1996. (Cité en page 10.)

[Leurent 2018] E. Leurent. *An Environment for Autonomous Driving Decision-Making*. Dépot GitHub : `https://github.com/eleurent/highway-env`, 2018. Accessed : 2020-09-25. (Cité en page 16.)

[Lindvall 2017] M. Lindvall, A. Porter, G. Magnusson et C. Schulze. *Metamorphic Model-based Testing of Autonomous Systems*. Dans 2nd International Workshop on Metamorphic Testing (MET), Buenos Aires, Argentina, pages 35–41, 2017. (Cité en page 23.)

[Liu 2020] T. Liu, X. Mu, X. Tang, B. Huang, H. Wang et D. Cao. *Dueling Deep Q Network for Highway Decision Making in Autonomous Vehicles : A Case Study*. CoRR, vol. abs/2007.08343, 2020. (Cité en page 16.)

[Marinov 2001] D. Marinov et S. Khurshid. *TestEra : a novel framework for automated testing of Java programs*. Dans 16th Annual International Conference on Automated Software Engineering (ASE), San Diego, USA, pages 22–31, 2001. (Cité en pages 28, 31 et 128.)

[Matthews 2020] G. Matthews, P. Hancock, J. Lin, A-R Panganiban, L. Reinerman-Jones, J. Szalma et R. Wohleber. *Evolution and revolution : Personality research for the coming world of robots artificial intelligence, and autonomous systems*. ELSEVIER Personality and Individual Differences, vol. 169, no. 3, page 109969, 2020. (Cité en page 15.)

[Maurer 1990] P.M. Maurer. *Generating Test Data with Enhanced Context-Free Grammars*. IEEE Software, vol. 7, no. 4, pages 50–55, 1990. (Cité en pages 28, 50 et 128.)

[Maurer 2000] M. Maurer. *EMS-vision : knowledge representation for flexible automation of land vehicles*. Dans IEEE 4th Intelligent Vehicles Symposium, Dearborn, Michigan, USA, 2000. (Cité en page 24.)

[Menghi 2019] C. Menghi, S. Nejati, K. Gaaloul et L. Briand. *Generating Automated and Online Test Oracles for Simulink Models with Continuous and Uncertain Behaviors*. Dans 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tallinn, Estonia, page 27–38, 2019. (Cité en page 22.)

[Menzies 2005] T. Menzies et C. Pecheur. *Verification and validation and artificial intelligence*. Advances in computers, vol. 65, pages 153–201, 2005. (Cité en page 13.)

[Micskei 2012] Z. Micskei, Z. Szatmári, J. Oláh et I. Majzik. *A Concept for Testing Robustness and Safety of the Context-Aware Behaviour of Autonomous Systems*. Dans Agent and Multi-Agent Systems. Technologies and Applications (AMSTA), Dubrovnik, Croatia, pages 504–513, 2012. (Cité en page 25.)

[Miller 1986] G. Miller. *The definition and rendering of terrain maps*. Dans 13th annual conference on Computer graphics and interactive techniques (SIGGRAPH), Dallas, Texas, USA, volume 20, pages 39–48, 1986. (Cité en page 26.)

[Miller 2004] A. Miller et P. Allen. *Graspit ! a versatile simulator for robotic grasping*. IEEE Robotics and Automation Magazine, vol. 11, no. 4, pages 110–122, 2004. (Cité en page 18.)

[Murray 1996] J. D Murray et W. VanRyper. Dans Encyclopedia of graphics file formats (pp 55-74), ISBN 1565921615, 1996. (Cité en page 138.)

[Mühlbacher 2016] C. Mühlbacher, S. Gspandl, M. Reip et G. Steinbauer. *Improving dependability of industrial transport robots using model-based techniques*. Dans 2016 IEEE International Conference on Robotics and Automation (ICRA), Stockholm, Sweden, 2016. (Cité en page 20.)

[Müller 2018] M. Müller, V. Casser, J. Lahoud, N. Smith et B. Ghanem. *Sim4CV : A Photo-Realistic Simulator for Computer Vision Applications*. International Journal of Computer Vision, vol. 126, no. 9, pages 902–919, 2018. (Cité en page 18.)

[Nair 2019] A. Nair, K. Meinke et S. Eldh. *Leveraging Mutants for Automatic Prediction of Metamorphic Relations using Machine Learning*. Dans 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), Tallin, Estonia, page 1–6, 2019. (Cité en pages 1 et 23.)

[Nentwig 2010] M. Nentwig et M. Stamminger. *A method for the reproduction of vehicle test drives for the simulation based evaluation of image processing algorithms*. Dans 13th International IEEE Conference on Intelligent Transportation Systems (ITSC), Funchal, Madeira Island, Portugal, pages 1307–1312, 2010. (Cité en page 24.)

[Nguyen 2012] C. D. Nguyen, A. Perini, P. Tonella, S. Miles, M. Harman et M. Luck. *Evolutionary testing of autonomous software agents*. Autonomous Agents and Multiagent Systems, vol. 25, pages 260–283, 2012. (Cité en pages 12, 15 et 21.)

[OCL 2020] OCL. *Object Constraint Language*. `https://www.omg.org/spec/OCL/`, 2020. Accessed : 2020-09-25. (Cité en page 29.)

[okt 2020] *Oktal Sydac*. `https://www.oktalsydac.com/`, 2020. Accessed : 2021-05-22. (Cité en page 17.)

[Pan 2013] F. Pan, Y. Hou, Z. Hong, L. Wu et H. Lai. *Efficient Model-based Fuzz Testing Using Higher-order Attribute Grammars.* Journal of Software, vol. 8, 2013. (Cité en pages 28, 29, 30, 50 et 128.)

[Parenthoën 2004] M. Parenthoën, T. Jourdan, J. Tisseau*et al. IPAS : Interactive phenomenological animation of the sea.* Dans The Fourteenth International Offshore and Polar Engineering Conference (ISOPE), Toulon, France, 2004. (Cité en page 18.)

[PEACH 2020] PEACH. *Peach Fuzzer.* `https://www.peach.tech/`, 2020. Accessed : 2020-09-25. (Cité en pages 28 et 128.)

[Pecheur 2000] C. Pecheur. *Verification and validation of autonomy software at NASA.* 2000. (Cité en page 13.)

[Perlin 1985] K. Perlin. *An image synthesizer.* ACM Computer Graphics (SIGGRAPH), vol. 19, no. 3, page 287–296, 1985. (Cité en page 26.)

[Plummer 2006] A-R Plummer. *Model-in-the-loop testing.* Proceedings of the Institution of Mechanical Engineers, Part I : Journal of Systems and Control Engineering, vol. 220, no. 3, 2006. (Cité en page 16.)

[Poulding 2014] S. Poulding et R. Feldt. *Generating structured test data with specific properties using nested Monte-Carlo search.* Dans 16th Genetic and Evolutionary Computation Conference (GECCO), Vancouver, BC, Canada, page 1279–1286, 2014. (Cité en pages 28 et 30.)

[pre 2020] *PreScan Simulation platform for ADAS.* `https://tass.plm.automation.siemens.com/prescan`, 2020. Accessed : 2020-09-25. (Cité en page 17.)

[Purdom 1972] P. Purdom. *A sentence generator for testing parsers.* BIT Numerical Mathematics, vol. 12, no. 3, pages 366–375, 1972. (Cité en page 28.)

[Quigley 2009] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler et A. Ng. *ROS : an open-source Robot Operating System.* Dans ICRA Workshop on Open Source Software, Kobe, Japan, volume 3, 2009. (Cité en page 13.)

[Reymann 2019] C. Reymann, M. Foughali et S. Lacroix. *Repeatable Decentralized Simulations for Cyber-Physical Systems.* Dans International Conference on Software Quality, Reliability and Security (QRS 2019), Sofia, Bulgaria, pages 240–247, 2019. (Cité en page 14.)

[Robert 2020a] C. Robert, J. Guiochet et H. Waeselynck. *TAF - Testing Automation Framework.* `https://redmine.laas.fr/projects/taf`, 2020. Accessed : 2021-04-20. (Cité en pages 96, 129, 136, 150 et 155.)

[Robert 2020b] C. Robert, J. Guiochet et H. Waeselynck. *TAF - Testing Automation Framework.* `https://www.laas.fr/projects/taf/`, 2020. Accessed : 2021-04-20. (Cité en page 150.)

[ROBERT 2020c] C. ROBERT, J. Guiochet et H. Waeselynck. *Testing a non-deterministic robot in simulation - How many repeated runs ?* Dans The

fourth IEEE International Conference on Robotic Computing (IRC 2020), Taichung, Taiwan, 2020. (Cité en page 63.)

[ROBERT 2020d] C. ROBERT, T. Sotiropoulos, H. Waeselynck, J. Guiochet et S. Vernhes. *The virtual lands of Oz : testing an agribot in simulation.* Empirical Software Engineering, vol. 25, no. 3, pages 2025–2054, 2020. (Cité en page 37.)

[Rocha 2010] RV. Rocha et RB Araújo. *Selecting the Best Open Source 3D Games Engines*, 2010. (Cité en page 17.)

[Rudd 2008] L. Rudd et H. Hecht. *Certification techniques for advanced flight critical systems, WPAFB.* Dans Technical report, 2008. (Cité en page 12.)

[sca 2020] *SCANeR.* `https://www.avsimulation.com/`, 2020. Accessed : 2021-05-22. (Cité en page 17.)

[Shaker 2016] N. Shaker, J. Togelius et M. Nelson. *Fractals, noise and agents with applications to landscapes.* Dans Procedural Content Generation in Games, ISBN 978-3-319-42714-0, 2016. (Cité en page 26.)

[Soltana 2020] G. Soltana, M. Sabetzadeh et L. Briand. *Practical Constraint Solving for Generating System Test Data.* ACM Transactions on Software Engineering and Methodology, vol. 29, no. 2, page 1–48, 2020. (Cité en pages 2, 29, 31, 123, 125, 128, 129, 131, 134, 148 et 149.)

[Sotiropoulos 2016] T. Sotiropoulos, J. Guiochet, F. Ingrand et H. Weaselynck. *Virtual Worlds for Testing Robot Navigation : a Study on the Difficulty Level.* Dans 12th European on Dependable Computing Conference (EDCC), Gothenburg, Sweden, pages 153–160, 2016. (Cité en pages 14, 17, 18, 23, 25 et 27.)

[Sotiropoulos 2017] T. Sotiropoulos, J. Waeselynck H.‘and Guiochet et F. Ingrand. *Can robot navigation bugs be found in simulation ? An exploratory study.* Dans 17th IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, 2017. (Cité en pages 21, 44, 52, 62 et 85.)

[Sotiropoulos 2018] Thierry Sotiropoulos. *Test aléatoire de la navigation de robots dans des mondes virtuels.* mémoire de thèse, Université Paul Sabatier - Toulouse III, 2018. (Cité en pages 1, 6, 17, 37, 41, 43, 52, 56 et 58.)

[Swierstra 2006] S. Swierstra et H. Vogt. *Higher Order Attribute Grammars.* Dans Attribute Grammars, Applications and Systems. ISBN 978-3-540-54572-9, pages 256–296, 2006. (Cité en page 30.)

[Tian 2018] Y. Tian, K. Pei, S. Jana et B. Ray. *DeepTest : Automated Testing of Deep-Neural-Network-Driven Autonomous Cars.* Dans 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, page 303–314, 2018. (Cité en pages 19, 20 et 23.)

[Timperley 2018] C.S. Timperley, A. Afzal, D.S. Katz, J.M. Hernandez et C. Le Goues. *Crashing Simulated Planes is Cheap : Can Simulation Detect*

Robotics Bugs Early ? Dans 11th IEEE International Conference on Software Testing, Verification and Validation (ICST), Västerås, Sweden, pages 331–342, 2018. (Cité en pages 18 et 85.)

[Tiwari 2003] A. Tiwari et P. Sinha. *Issues in v&v of autonomous and adaptive systems.* Dans IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), Montreal, QC, Canada, volume 2, pages 1339–1342, 2003. (Cité en page 13.)

[Togelius 2011] J. Togelius, G.N. Yannakakis, K.O. Stanley et C. Browne. *Search-Based Procedural Content Generation : A Taxonomy and Survey.* IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, no. 3, pages 172–186, 2011. (Cité en pages 25 et 45.)

[Ulbrich 2015] S. Ulbrich, T. Menzel, A. Reschka, F. Schuldt et M. Maurer. *Defining and Substantiating the Terms Scene, Situation, and Scenario for Automated Driving.* Dans IEEE 18th International Conference on Intelligent Transportation Systems (ITST), Copenhagen, Denmark, pages 982–988, 2015. (Cité en page 25.)

[uni 2020] *Unity.* `https://https://unity.com/`, 2020. Accessed : 2020-09-25. (Cité en page 17.)

[unr 2020] *Unreal Game Engine.* `https://www.unrealengine.com/`, 2020. Accessed : 2020-09-25. (Cité en page 17.)

[vir 2020] *Virtual Test Drive.* `http://www.mscsoftware.com/product/virtual-test-drive`, 2020. Accessed : 2020-09-25. (Cité en page 17.)

[Weyuker 1982] EJ. Weyuker. *On testing non-testable programs.* The Computer Journal, vol. 25, no. 4, pages 465–470, 1982. (Cité en page 19.)

[Wirsing 2015] M. Wirsing, M. Hölzl, N. Koch et P. Mayer. Dans Software Engineering for Collective Autonomic Systems - The ASCENS Approach, ISBN 978-3-319-16309-3, 2015. (Cité en page 12.)

[Xu 2010] Z. Xu, L. Zheng et H. Chen. *A Toolkit for Generating Sentences from Context-Free Grammars.* Dans 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM), Pisa, Italy, pages 118–122, 2010. (Cité en page 28.)

[Yao 2015] W. Yao, W. Dai, J. Xiao, H. Lu et Z. Zheng. *A simulation system based on ROS and Gazebo for RoboCup Middle Size League.* Dans 15th IEEE International Conference on Robotics and Biomimetics (ROBIO), Zhuhai, China, pages 54–59, 2015. (Cité en page 17.)

[Zendel 2013] O. Zendel, W. Herzner et M. Murschitz. *VITRO - Model based vision testing for robustness.* Dans IEEE International Symposium on Robotics (ISR 2013), Seoul, Korea, pages 1–6, 2013. (Cité en page 81.)

[Zendel 2017] O. Zendel, M. Murschitz, M. Humenberger et W. Herzner. *How Good Is My Test Data ? Introducing Safety Analysis for Computer Vision.* International Journal of Computer Vision, vol. 125, no. 1, pages 95–109, 2017. (Cité en page 81.)

[Zhang 2018] M. Zhang, Y. Zhang, L. Zhang, C. Liu et S. Khurshid. *DeepRoad : GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems.* Dans 33rd ACM/IEEE International Conference on Automated Software Engineering, (ASE), Montpellier, France, pages 132–142, 2018. (Cité en page 22.)

[Zou 2014] X. Zou, R. Alexander et J. McDermid. *Safety Validation of Sense and Avoid Algorithms Using Simulation and Evolutionary Search.* Dans 33rd International Conference on Computer Safety, Reliability, and Security (SAFECOMP), Firenze, Italy, pages 33–48. 2014. (Cité en pages 16 et 21.)

[Zou 2016] X. Zou, R. Alexander et J. McDermid. *On the Validation of a UAV Collision Avoidance System Developed by Model-Based Optimization : Challenges and a Tentative Partial Solution.* Dans IEEE/IFIP 46th International Conference on Dependable Systems and Networks Workshop (DSN Workshop), Toulouse, France, pages 192–199, 2016. (Cité en page 16.)