UNIVERSITÉ DE MONTRÉAL

IDENTIFICATION AUTOMATIQUE DES PROBLÈMES FONCTIONNELS DANS LES SIMULATEURS DE VOL

VINCENT BOISSELLE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
NOVEMBRE 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

IDENTIFICATION AUTOMATIQUE DES PROBLÈMES FONCTIONNELS DANS LES SIMULATEURS DE VOL

présenté par : BOISSELLE Vincent
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. GUÉHÉNEUC Yann-Gaël, Doctorat, président
M. ADAMS Bram, Doctorat, membre et directeur de recherche
M. PETRENKO Alexandre, Ph. D., membre et codirecteur de recherche
M. KHOMH Foutse, Ph. D., membre

DÉDICACE

À mes parents pour m'avoir toujours soutenu dans mes démarches.

À mes amis de Montréal pour leur grand support pendant mes deux années de maîtrise.

REMERCIEMENTS

J'aimerais d'abord remercier mon directeur de recherche, Dr. Bram Adams pour son support exceptionnel tout au long de ma Maîtrise, je tiens également à remercier mon codirecteur de recherche Dr. Alexander Petrenko pour m'avoir fait part de sa grande expérience de recherche dans le domaine des tests.

Je tiens à saluer et à remercier les personnes avec lesquelles j'ai collaboré chez CAE : Patricia Gilbert, Michel Galibois, Marc-André Proulx, Jean-Pierre Rousseau pour leur précieux soutient technique et de coordination, et pour m'avoir fait découvrir cette compagnie à la fine pointe de la technologie. Je salue par le fait même mes anciens collègues du Centre de Recherche en Informatique de Montréal (CRIM), Dr. Giuseppe Destefanis et le directeur Dr. François Labonté pour leur précieuse collaboration dans le projet.

J'aimerais également remercié le Conseil de Recherche en Sciences Naturelles et en Génie du Canada (CRSNG) pour avoir supporté financièrement ce projet avec la Bourse Collaborative en Recherche et Développement dans le cadre d'un projet de recherche industrielle.

Je tiens finalement à saluer mes anciens collègues du MCIS et des autres laboratoires du département pour leur soutient et le dévouement qu'ils ont pour la recherche, on ne peut que s'en inspirer.

RÉSUMÉ

Les simulateurs de vol sont des systèmes composés d'un système logiciel et d'actuateurs mécaniques qui recréés la dynamique d'un vrai vol avec un aéronef et qui sont notamment utilisés par les compagnies de transport aérien pour former leurs futurs pilotes. Ces simulateurs doivent recréer des scénarios de vol permettant d'exécuter des itinéraires réguliers de vol ou de confronter les pilotes à différentes situations d'urgence qui peuvent subvenir lors d'un vrai vol (p. ex., une tempête violente). Puisque les simulateurs de vol ont un impact direct sur la qualité de formation des pilotes, une batterie exhaustive de tests s'impose à chaque fois qu'une nouvelle version d'un simulateur doit être mise en opération. Une bonne partie de ces tests requièrent l'intervention d'un pilote qui stimule le système en réalisant une série d'opérations de contrôle provenant de scénarios de vol préparés par des experts en aéronautique, ce qui est gourmand en temps, en ressources humaines et en ressources financières.

La raison de la nécessité de réaliser tous ces tests est que le logiciel en soit est constitué de plusieurs composants de type boîte noire dont le code source n'est pas disponible, ils sont fournis tels quels par les fournisseurs du composant réel d'origine devant être simulé (p. ex., un composant hydraulique d'un aéronef que nous voulons simuler). Puisque nous n'avons pas pas accès au code source, il n'est pas possible lors de la mise-à-jour de l'un de ces composants de savoir quel sera le changement dans le comportement du simulateur. Dans le meilleur des cas, un défaut dans un composant aura un impact local et, dans le pire des cas, il peut nuire à tous les autres composants du simulateur. Il est dans ce cas nécessaire d'utiliser une stratégie de test d'une large granularité couvrant à la fois le fonctionnement des composants eux-mêmes et l'ensemble des fonctionnalités de vol du simulateur afin d'en assurer sa qualité.

Dans le cadre de ces tests, afin de réduire le temps requis pour vérifier le bon comportement d'une nouvelle version d'un simulateur, nous proposons dans ce mémoire d'automatiser l'analyse des résultats de test en modélisant le comportement normal du système logiciel en utilisant de l'apprentissage automatique. Un tel modèle tente de recréer le comportement stable du système en bâtissant des règles reliant ses entrées à chacune de ses sorties. Les entrées sont constituées de métriques provenant des contrôles qui stimulent le système et les sorties sont constituées de métriques que nous pouvons observer. Nous produisons donc un modèle initial d'une version fiable du simulateur et nous validons par la suite le bon comportement des versions subséquentes en comparant leur comportement avec celui du modèle initial. Nous cherchons à voir s'il y a un écart trop important avec le modèle initial, ce nous

considérons comme étant une déviation.

Le but final de notre recherche est de pouvoir appliquer notre méthodologie d'analyse des résultats de test dans l'industrie de la simulation. Afin d'avoir plus de flexibilité dans nos expérimentations, nous commençons d'abord par valider l'approche avec un simulateur plus simple et ouvert. Nous utilisons donc le simulateur de vol à source libre FlightGear comme cas d'étude pour évaluer notre approche de test. Ce simulateur utilise l'engin de vol très populaire JSBsim et la librairie SimGear pour modéliser les aéronefs et simuler leur comportement dynamique dans un environnement de vol. JSBsim est notamment utilisé par la Nasa à des fins de recherche. Nous réalisons d'abord un vol manuel en utilisant des périphériques de contrôle d'un ordinateur tout en enregistrant les données contenues dans le simulateur. Le vol doit être composé de toutes les opérations régulières de pilotage afin que le modèle que l'on en extrait représente fidèlement le comportement normal du simulateur. Nous réalisons par la suite plusieurs répétitions du même vol afin d'identifier des niveaux de seuils robustes pour déterminer au delà de quels écarts un métrique d'une nouvelle version doit être considéré comme déviant par rapport au modèle. Nous élaborons à cet effet cinq scénarios de mutation du comportement de la version initiale de notre simulateur afin de générer différentes versions ayant des métriques qui respectent le comportement normal du sytème ou présentant une déviation du comportement normal reliée à un problème fonctionnel. En sachant d'avance quelles mutations présentent des déviations et avec l'aide d'experts identifiant précisément les métriques qui dévient de leur comportement normal, nous pouvons construire un oracle avec lequel nous évaluons la précision et le rappel de notre approche de détection.

En particulier, dans le cadre de notre étude empirique sur FlightGear, nous modifions une version initiale du simulateur en y injectant cinq différentes mutations qui n'ont pas d'impact grave ou qui engendrent des problème fonctionnels avec lesquels nous évaluons notre approche. Les résultats montrent qu'en choisissant des niveaux de seuil initiaux lors d'une première calibration, notre approche peut détecter les métriques ayant des déviations avec un rappel majoritairement de 100% (un seul cas de mutation à 50%) et une précision d'au moins 40%. En ce qui a trait aux mutations ne changeant pas le comportement normal du simulateur, notre approche a un taux de fausses alarmes de 60%. Nous montrons également qu'avec une base de données plus large, il est possible de calibrer notre approche afin d'améliorer sa performance. Avec des niveaux de seuil optimaux qui sont calibrés pour chacune des mutations, nous obtenons un taux de rappel de 100% pour tous les cas, une précision d'au moins 50% pour les versions ayant des problèmes fonctionnels et un taux de fausses alarmes de 0% pour les versions n'ayant pas de problèmes fonctionnels.

Afin d'ouvrir la voie pour des améliorations futures, nous utilisons notre méthodologie pour

faire une petite expérimentation connexe sur JSBsim afin de détecter les déviations dans les transitions entre les états stables de chacun des métriques. Nous utilisons la même modélisation du simulateur. Les résultats montrent que nous pouvons battre une classification aléatoire des déviations. Cette nouvelle approche n'en n'est qu'à ces débuts et nous sommes convaincu que nous pouvons obtenir des résultats plus significatifs avec de futurs travaux sur le sujet.

Nous proposons également dans les améliorations futures de descendre le niveau de granularité de notre modélisation du simulateur au niveau de chacun de ses composants. Cela permettrait d'isoler exactement qu'elle composant présente une déviation, et ainsi trouver la source du problème.

ABSTRACT

Flight simulators are systems composed of software and mechanical actuators used to train crews for real flights. The software is emulating flight dynamics and control systems using components off-the-shelf developed by third parties. For each new version of these components, we do not know what parts of the system is impacted by the change and hence how the change would impact the behaviour of the entire simulator. Hence, given the safety critical nature of flight simulators, an exhaustive set of manual system tests must be performed by a professional pilot. Test experts then rely on the test pilot's feedback and on the analysis of output metrics to assess the correctness of a simulator's behaviour, which consumes time and money, even more these tests must be repeated each time one of the components is updated by its vendor.

To automate the analysis of test results from new simulator versions, we propose a machine learning-based approach that first builds a model that mimics the normal behaviour of a simulator by creating rules between its input and output metrics, where input metrics are control data stimulating the system and output metrics are data used to assess the behaviour of the system. We then build a behavioural model of the last-known correctly behaving version of the simulator, to which we compare data from new versions to detect metric deviations. The goal of our research is to first build a model of the simulator, then detect deviations of new simulator versions using that model, and finally develop an automatic approach to flag deviations when there are functional problems.

Our case study uses the open-source flight simulator Flight Gear, based on the well-known JSBsim flight dynamic engine currently used by the Nasa. Applying our approach, we first drive a manual basic flight using computer peripherals while recording metrics in the simulator. We then use the recorded metrics to build a model of this basic execution scenario. After repeating several times our flight we have enough data to identify robust metric deviation thresholds to determine when a metric in a new version is deviating too far from the model.

We generate five different versions of FlightGear by injecting harmless and harmful modifications into the the simulator, then perform again our initial flight multiple times for each of those versions while recording input and output metrics. We rely on experts identifying which output metric should be deviating in each scenario to build an oracle. Using an initial threshold, our approach can detect most of the deviations in problematic versions with a near perfect recall (only one case at 50%) and a reasonable precision of at least 40%. For the versions without harmfull deviations we get a false alarm rate of 60%. By optimizing the

threshold for each version, we get a perfect recall of 100%, a precision of at least 50%, and a false alarm rate of 0% for the good behaving versions.

To open the path for future work, we perform a case study on JSBsim using a variant of our basic deviation detection approach to detect transient deviations using the same model. Our results show that we can beat a random classification for one kind of deviation, however for other deviations our models do not perform as well. This new approach is just a first step towards transient deviation detection, and we are convinced that we can get better results in future work. We also propose to use finer-grained models for each component to be able to isolate the exact component causing a functional problem.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

TCP-IP	Transmission Control Protocol/Internet Protocol
MTA	Manuel de Tests d'Acceptation
GEQ	Guide d'Essais de Qualification
FOQA	Flight operations quality assurance
CSV	Comma-Separated Values
UDP	User Datagram Protocol
RAM	Random Access Memory
ROC	Receiver Operating Characteristic
XML	Extensible Markup Language

CHAPITRE 1 INTRODUCTION

Les simulateurs de vol reproduisent la dynamique d'un vol et sont associés à un éventail de scénarios de vols afin d'entraîner les pilotes à piloter de vrais aéronefs. Il faut donc s'assurer que le sytème est conforme aux normes de l'industrie aéronautique et ne présente aucun problème fonctionnel pour que les pilotes aient une formation de qualité, ce qui est essentiel pour assurer la sécurité des vrais vols commerciaux. Pour cette raison, les simulateurs doivent subir une batterie exhaustive de tests à chaque fois qu'une nouvelle version du simulateur est mise en opération. Comme la plupart des composants constituant le logiciel du simulateur sont fournis par des tierces parties qui ne dévoilent pas leur code source, le manufacturier du simulateur s'occupe de connecter les composants entre eux par le biais de leur interface de connexion et fait appel à des pilotes professionnels pour réaliser différentes routines de pilotage sur le simulateur. Par la suite, l'opinion des pilotes est collectée par l'équipe de test pour valider le fonctionnement correct du simulateur. Les testeurs peuvent aussi analyser les données de vol enregistrées pendant les tests afin de valider que leur enveloppe temporelle est conforme à un gabarit de test. Comme l'intervention humaine rend le processus de test coûteux, nous proposons dans ce mémoire une approche automatique d'analyse des résultats de test dans les simulateurs de vol. Nous utilisons à cet effet une technique d'apprentissage automatique pour obtenir un modèle d'une version du simulateur ne présentant pas de problème fonctionnel avec laquelle nous comparons le comportement de d'autres versions du simulateur afin de détecter les possibles déviations menant à des problèmes fonctionnels.

1.1 Concepts et définitions

Dans cette section, nous décrivons les concepts de base utilisés tout au long du mémoire.

Composant boîte noire dans un système logiciel

Un composant logiciel de type boîte noire se définit seulement par ses entrées et sorties et ne dévoile pas son code source. On réalise l'assemblage d'un système en connectant les entrées et sorties de plusieurs composants étant chacun responsable d'une fonctionnalité. Le système expose également une interface d'entrée et de sortie aux acteurs externes voulant interagir avec ce dernier. La figure 1.1 montre l'exemple d'un système composé de boîtes noires, où les entrées I1, I2, et I3 acheminent des données provenant de l'extérieur au Composant 1 et la sortie O3 achemine des données du Composant 1 vers l'extérieur du système. On interconnecte les composants 1 et 2 en associant les sorties O1 et O2 avec les entrées I4 et

I5. Le Composant 2 possède également la sortie O4 qui se dirige vers l'extérieur du système. L'échange d'information entre les entrées et sorties des composants peut se faire soit par le biais d'une mémoire partagée ou en utilisant un protocole de transport (p. ex., *Transmission Control Protocol/Internet Protocol (TCP-IP)).*

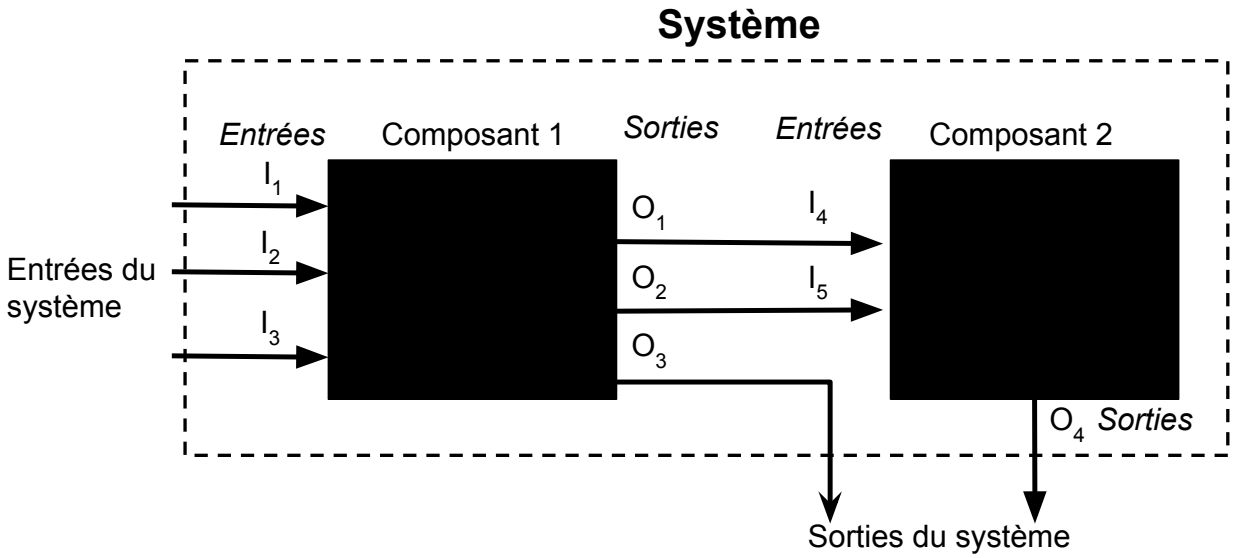


Figure 1.1 Composants boîtes noires qui forment un système.

L'avion et ses phases de vol

Il est essentiel d'avoir des notions de base sur les avions afin de comprendre le fonctionnement des simulateurs de vol et pour comprendre les informations que l'on en extrait. Nous montrons sur la figure 1.2 les composants de la structure d'un avion (aussi appelés surfaces) qui contrôlent ses trois axes (figure 1.3). Les ailes sont les surfaces principales portant l'avion dans les airs. Les turboréacteurs propulsent l'appareil dans la direction désirée et la vélocité qui en découle assure la portance de l'appareil. On utilise les ailerons pour contrôler le roulis, le gouvernail de profondeur pour contrôler le tangage et le gouvernail de direction pour contrôler le lacet. Le roulis, le tangage et le lacet sont les angles de rotation d'un aéronef autour de chacun de ses trois axes (voir figure 1.3). On utilise les volets d'atterrissage pour donner plus de surface aux ailes pendant les opérations de décollage et d'atterrissage. D'autres composants sont également présents dans la structure de l'avion (tel que montré sur la figure 1.2), mais nous nous concentrons sur ceux que nous utilisons pour contrôler l'aéronef dans ce mémoire (nous pourrions utiliser tous les composants si nécessaire).

Figure 1.2 Les différents composants structurels d'un avion.

source : `http://www.infovisual.info/05/img_fr082%20Avion%20de%20ligne.jpg/`

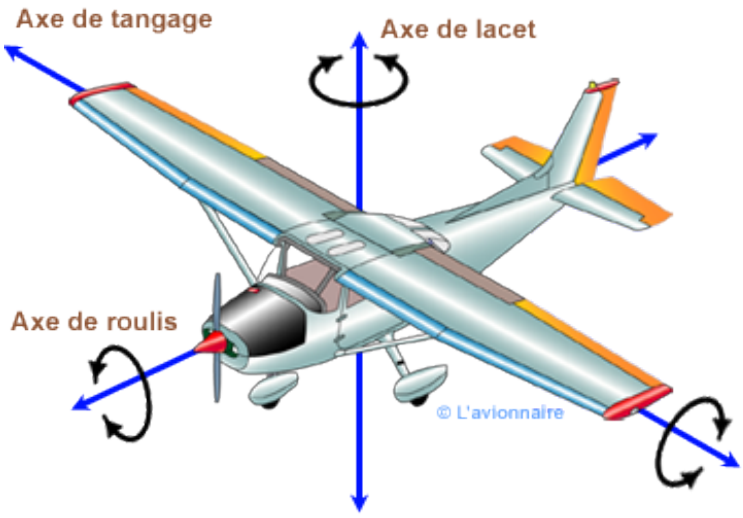


Figure 1.3 Les axes d'un aéronef.

source : `http://www.lavionnaire.fr/CelluleGouvernes.php`

La figure 1.4 montre l'ensemble des phases de vol d'un avion, il est possible de les grouper en trois grandes phases, soient : le décollage, la croisière et l'atterrissage. Au décollage, l'avion prend de l'altitude par rapport au sol pour atteindre une altitude de croisière dans laquelle se déroule la phase de croisière. Le retour au sol pour atteindre la destination se fait par

la suite durant l'atterrissage. Chaque phase nécessite d'exécuter une série d'opérations, le pilote doit porter une attention particulière durant les phases critiques telles que le décollage et l'atterrissage.

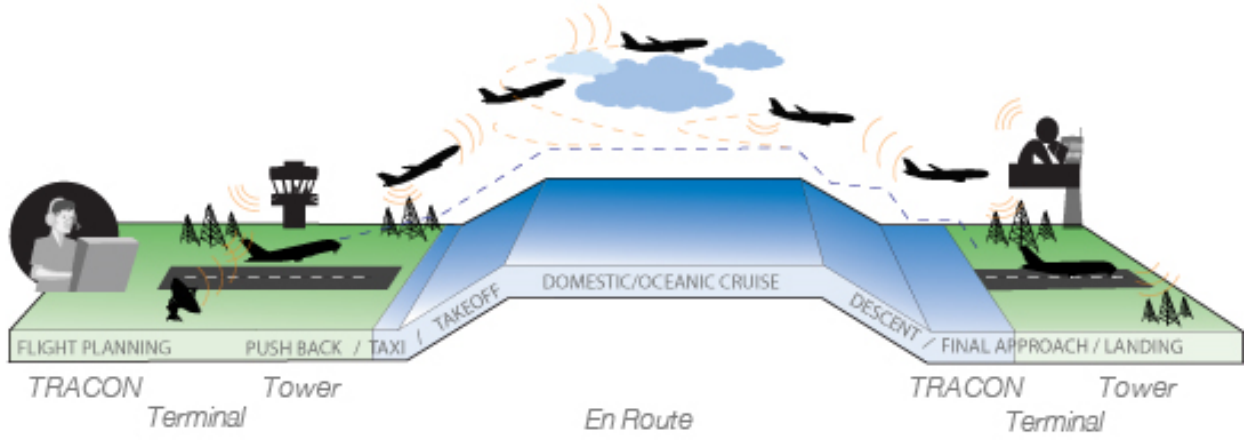


Figure 1.4 Les phases de vol.

source : `http://www.allweatherinc.com/programs/flexids/`

Simulateur de vol

Les simulateurs de vol complet reproduisent la dynamique d'un vol en utilisant conjointement un système logiciel et des actuateurs mécaniques intégrés à un habitacle de simulation. La figure 1.5 montre le degré de réalisme de vol qu'apporte un simulateur. Le simulateur se contrôle à partir d'un vrai cockpit qui est connecté au système logiciel et on utilise des projecteurs afin d'afficher le rendu graphique de l'environnement de vol tout en offrant une vue panoramique. La partie logicielle du simulateur est composée de divers composants échangeant de l'information par le biais de leur interface d'entrée et de sortie. Chacun des composants possède une spécialité, par exemple un composant du logiciel peut recréer l'ensemble du système hydraulique d'un avion en utilisant des équations hydrauliques.

Figure 1.5 Simulateur de vol.

source :
`https://commons.wikimedia.org/wiki/File:Full_Flight_Simulator_(5573438825).jpg`

Opérations et métriques

Dans les tests des systèmes logiciels, on exécute des scénarios composés de plusieurs opérations pendant lesquels on enregistre des données du système. Chaque opération applique un stimulus sur un système qui affecte potentiellement son comportement. Pendant l'exécution de ces opérations, on qualifie comme métrique toute variable qui est utilisée pour stimuler le système ou pour mesurer les effets engendrés par le stimulus.

Afin d'entraîner des modèles expliquant le comportement du système, nous séparons les métriques d'entrée et de sortie. Les métriques d'entrée sont les données envoyées à un composant du système pendant des opérations afin de le stimuler, tandis que les métriques de sortie sont les données observables qui peuvent être utilisées pour analyser le comportement d'un composant ou du système en entier. Sur la figure 1.1, les métriques d'entrée pour le Composant 1 sont I1, I2 et I3, tandis que ses métriques de sortie sont O1, O2 et O3. Par exemple, dans un simulateur de vol, les données reliées à la vitesse et la position de l'aéronef seraient des métriques de sortie, tandis que les données reliées au contrôle des moteurs et au contrôle des ailerons seraient des métriques d'entrée.

1.2 Problématique

1.2.1 Automatisation de l'analyse des résultats de test

Les systèmes logiciels dans les simulateurs de vol sont composés de plusieurs composants boîtes noires dont une grande partie est développée par des tierces parties ne dévoilant pas leur code source. Seuls les exécutables, les bibliothèques et la documentation (pouvant parfois ne pas être complète) sont disponibles. Il n'est donc pas possible de tester le logiciel avec une approche boîte blanche avec des cas de test sur chacune des fonctions développées en tentant de couvrir tous les chemins dans le code, où l'on vérifie pour chaque cas de test si la sortie de la fonction testée est conforme aux spécifications.

Nous utilisons une approche de test boîte noire, ce qui consiste à utiliser une interface de contrôle pour stimuler le système tout en enregistrant des métriques permettant à des experts dans le domaine d'application de juger le comportement du système. Il est possible de tester chaque composant du système ayant une interface publique accessible ou bien de tester le système en entier. Sélectionner des métriques pour modéliser un composant requiert une bonne connaissance de ce dernier, tandis que l'utilisation des métriques visibles dans le système en entier permet de mettre l'accent sur les entrées et sorties globales, sans considérer les métriques spécifiques (locales) de chaque composant.

Pour effectuer les tests sytèmes dans l'industrie de la simulation, l'équipe de test contracte un pilote chaque fois qu'une nouvelle version d'un composant est intégrée afin de valider que le simulateur fonctionne encore correctement dans son ensemble. Les testeurs doivent également vérifier si les valeurs de certaines métriques enregistrées correspondent à celles des modèles physiques de l'avion simulé. Il en résulte que les testeurs doivent appliquer des tests coûteux en ressources humaines et financières, sans oublier qu'il est parfois difficile d'attendre après le pilote quand les contraintes de temps sur la livraison d'un simulateur sont serrées.

Il s'avère donc intéressant d'automatiser l'analyse des résultats de test dans les simulateurs de vol afin de réduire les coûts inhérents à l'approche manuelle et d'offrir plus de flexibilité aux testeurs lorsqu'il y a des contraintes en temps et en ressources. Afin de générer des données de test, les simulateurs ont pour la plupart des outils de script qui sont utilisés pour automatiser un vol de A à Z, il n'existe par contre pas d'outils pour vérifier automatiquement si le comportement des métriques de vol enregistrées est normal. Une partie de l'évaluation du simulateur repose actuellement sur l'évaluation subjective d'un pilote pour valider que le comportement de la nouvelle version du simulateur est encore conforme à l'ancienne version. Une approche automatisée d'évaluation des résultats de test doit tenir compte d'un modèle comportemental du système ciblé afin de valider si certaines des métriques enregistrées

dévient de leur tangente au cours d'un vol. On émet donc la thèse suivante :

> *L'utilisation d'un modèle comportemental issu de l'apprentissage automatique permet de détecter les versions d'un simulateur de vol qui ont un comportement déviant avec une bonne performance.*

1.2.2 Modèles

Le but d'un modèle est de simuler le comportement stable d'un système à partir de règles reliant ses entrées et ses sorties, c'est-à-dire l'information accessible de ses composants. La figure 1.6 montre un exemple de métrique faisant une transition entre l'"État Stable 1" et l'"État Stable 2" (p. ex., en changeant son altitude, l'avion fait une transition d'un état/altitude stable 1 vers un état/altitude stable 2), un modèle doit donc simuler quelles valeurs en entrée du système il faut pour atteindre chacun des états stables. Nous construisons les règles en utilisant un algorithme d'apprentissage automatique que l'on applique sur un jeu de données d'entraînement extrait du système. En premier lieu, nous utilisons des modèles des états stables, car ces états représentent les situations normalement désirées par des pilotes lorsqu'ils exécutent leurs opérations. Nous pourrions aussi modéliser la durée et la forme de la courbe des transitions des métriques, pour identifier des déviations dans les transitions menant à des problèmes fonctionnels. Nous explorons cette idée vers la fin du mémoire.

Dans le cas du simulateur, nous devons réaliser un vol dans lequel nous enregistrons l'ensemble des métriques nécessaires à la modélisation. La portée d'un modèle tient à la configuration du système dans lequel les métriques sont enregistrées (p. ex., les conditions météorologiques dans un simulateur). Le niveau de raffinement d'un modèle dépend des métriques d'entrée et de sortie utilisées lors de la modélisation, par exemple il serait possible de générer un modèle complet du système de la figure 1.1 en utilisant les entrées I1, I2, I3 et les sortie O3 et O4, tandis que la modélisation du Composant 1 utiliserait les mêmes entrées et les sorties O1, O2, O3.

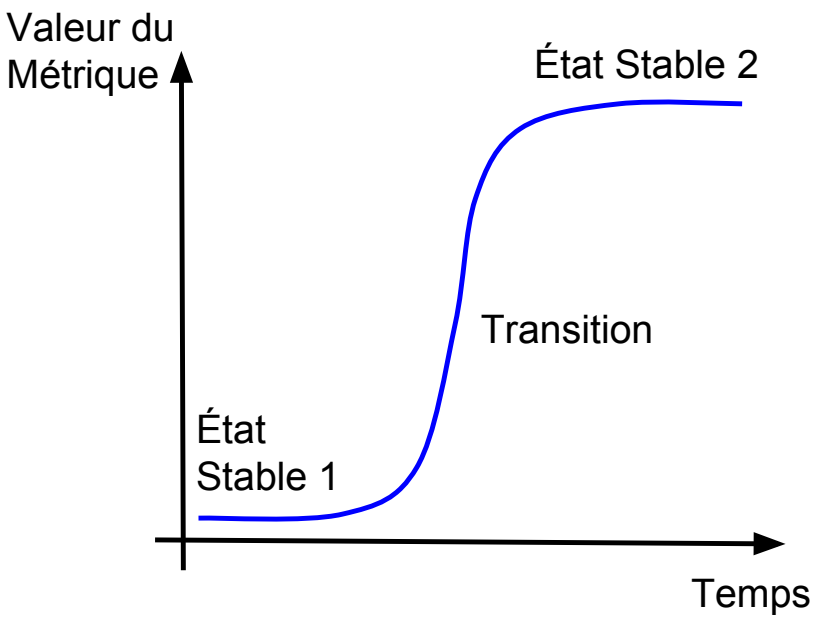


Figure 1.6 Courbe d'exemple des états stables et la transition d'une métrique.

Nous devons choisir, parmi plusieurs algorithmes d'apprentissage automatique, celui qui nous donne un modèle qui explique le mieux le comportement du simulateur et qui donne l'information la plus intelligible aux testeurs pour mieux comprendre le comportement du simulateur modélisé. Les métriques que nous utilisons dans la modélisation doivent être de qualité afin qu'elles génèrent un modèle qui représente le mieux le système, nous devons donc mettre les experts en aéronautique dans la boucle pour nous aider dans le processus de sélection des métriques. Il est également possible de choisir des métriques automatiquement en analysant l'occurrence mutuelle des entrées et sorties (p. ex., cette méthode de sélection est utilisée pour générer les arbres de décision).

1.2.3 Déviation

À partir du modèle de référence du simulateur, nous pouvons détecter s'il y a des déviations dans les métriques enregistrées pendant les tests qui sont faits sur une autre version. Nous considérons une déviation comme étant tout écart de comportement d'un métrique qui est assez grand pour être problématique (c.-à-.d, relié à un problème fonctionnel). Par exemple, on pourrait qualifier une voiture allant à une trop grande vitesse (une métrique) par rapport à notre point de vue comme ayant une vitesse déviante. L'identification d'une déviation peut se faire de manière subjective par un individu qui se base sur son expérience, en utilisant un modèle issu de l'apprentissage automatique (un individu simulé) ou en utilisant une mé-

thodologie statistique qui identifie un écart par rapport à la tendance d'un historique de données.

Puisque dans ce mémoire nous visons l'automatisation de l'analyse des résultats de test, nous utilisons un algorithme bâti avec une première version du système à la place d'un être humain pour juger le bon comportement d'une version ultérieure du système. Un tel algorithme explore l'ensemble des règles expliquant le comportement du système pour vérifier si les métriques enregistrées pour une nouvelle version du simulateur suivent bel et bien les règles du modèle. Par exemple, en nous référant au métrique de la figure 1.6, notre approche détecterait une déviation si le métrique était à l'"État Stable 1" à la place de l'"État Stable 2" (nous aurions dans ce cas une courbe totalement plane).

1.2.4 Scénarios de test

Afin de bâtir un modèle d'un simulateur de vol que l'on utilise pour détecter les potentielles déviations, il est nécessaire de développer une séquence d'opérations afin de stimuler le système. Nous devons donc élaborer conjointement avec des experts un vol de test qui stimule le système en couvrant les comportements d'un vol typique pour évaluer adéquatement la conformité du système. Le vol est constitué d'une séquence d'opérations qui contrôlent les métriques d'entrées. Il est également important de tenir compte de la configuration du système (p. ex., les conditions météorologiques). Afin d'évaluer la performance de notre approche de détection de déviation, nous appliquons notre approche sur différentes versions du simulateur. Cela peut se faire soit en pigeant dans le répertoire du logiciel de gestion de versions ou en modifiant le simulateur en injectant une mutation. Dans la deuxième option, on peut soit modifier le code source du simulateur (s'il est accessible) ou modifier sa configuration.

Dans leur élaboration, les scénarios peuvent être calqués directement des séquences de test manuelles existantes qui sont réalisées dans les vols commerciaux. Un scénario peut également être plus élaboré en recréant un vol complet entre deux villes. Pendant l'exécution d'un scénario, plusieurs métriques sont enregistrées et sont analysées ultérieurement par les experts en test pour vérifier si le simulateur a un comportement adéquat dans le contexte du scénario. Par exemple, un scénario pourrait être une séquence d'opérations pour piloter un avion décollant de Toronto et atterrissant à Montréal.

1.2.5 Évaluation de la performance de notre approche

Pour évaluer la performance de notre approche, nous devons avoir accès à différentes versions du simulateur de vol et nous devons avoir un scénario de test qui stimule le système. Cer-

taines des versions que nous évaluons présentent des modifications engendrant des problèmes fonctionnels et d'autres présentent des modifications gardant le comportement normal du système. Des experts doivent nous indiquer quelles métriques présentent des déviations pour chaque version du simulateur afin de bâtir un oracle de prédiction des déviations que nous utilisons comme référence pour évaluer la performance de notre approche.

1.3 Objectifs de recherche

Le premier objectif de ce mémoire est d'appliquer une technique d'apprentissage automatique pour modéliser le système logiciel d'un simulateur de vol. Le modèle doit donner une représentation visuelle du comportement du système afin d'en améliorer sa compréhension et ainsi aider les développeurs et les testeurs travaillant sur le simulateur.

Le deuxième objectif est d'utiliser un modèle de base du simulateur pour détecter des écarts dans les métriques que l'on obtient en jouant le même vol qu'utilisé lors de la modélisation sur différentes versions du simulateur de vol. En temps normal, les nouvelles versions sont des mise-à-jour du système que l'on veut mettre en production, mais, n'ayant pas accès à de telles versions dans notre cas, nous nous limitons à des mutations artificielles. Ainsi, nous pouvons générer ces nouvelles versions en injectant des mutations dans la configuration et le code source de la version originale du simulateur. Un système muté peut avoir un comportement normal ou il peut parfois avoir un comportement inattendu (un problème fonctionnel). Dans le cas d'un problème fonctionnel, nous nous attendons à ce que notre approche de test soit en mesure de détecter des déviations dans les métriques concernées, et ainsi donner une notification aux testeurs sur le mauvais comportement du système.

Le troisième objectif est de développer une technique permettant d'automatiser la détection des déviations dans les métriques de test collectées. Il est donc nécessaire de réaliser plusieurs fois le même vol sur la version originale du simulateur afin de choisir itérativement un niveau de seuil à partir duquel un écart entre le modèle et la métrique concernée présente une vraie déviation. Le niveau de seuil doit être choisi en faisant un compromis entre la précision et le taux de rappel des déviations.

Afin d'évaluer ces trois objectifs, nous réalisons une étude de cas sur FlightGear dans laquelle nous produisons différentes versions du simulateur de vol en changeant ses conditions météorologiques (p. ex., le vent) ou en injectant des fautes systèmes (p. ex., l'arrêt d'un moteur). Certaines versions comportent des déviations dans les métriques et d'autres n'en ont pas. On répond ainsi aux trois questions de recherche suivantes :

QR1 : Est-ce que nous sommes capable de modéliser un simulateur de vol avec une

bonne performance ? *Il est possible de modéliser le simulateur de vol avec l'utilisation d'un algorithme d'arbre de décision appliqué sur des métriques d'entrée et de sortie soigneusement sélectionnées avec l'aide d'experts.*

QR2 : Est-ce qu'il est possible d'utiliser le modèle pour détecter des écarts significatifs de comportement dans les métriques que nous pouvons utiliser pour identifier des déviations ? *Notre modèle est très sensible aux variations dans le comportement du simulateur, une simple étape de calibration avec des métriques provenant de la version de base du simulateur nous montre que certaines métriques ont des écarts pouvant aller jusqu'à 41% par rapport au modèle de base.*

QR3 : Quelle est la performance en terme de rappel, précision, et de fausses alarmes de notre approche automatisée ? *Il est possible de ne générer aucune fausse alarme pour la mutation avec faible vent, tandis que nous détectons toutes les déviations pour des mutations avec un fort vent avec une précision de 50% et toutes les déviations avec des fautes systèmes, avec seulement dans un cas une précision de moins de 100% (c.-à-.d., 67%.)*

1.4 Plan du mémoire

Nous présentons dans le chapitre 2 une revue de la littérature parcourant les méthodes de test logiciel existantes, les tests d'acceptation dans les simulateurs de vol, les simulateurs de vol disponibles sur le marché et les techniques actuelles de détection des déviations dans les données de vols commerciaux. Dans le chapitre 3, nous présentons la démarche de l'ensemble du travail de recherche que nous utilisons pour détecter automatiquement les déviations dans les métriques des nouvelles versions du simulateur. Nous utilisons cette démarche dans un papier de recherche présenté dans le chapitre 4, dans lequel nous faisons une étude de cas sur FlightGear. Nous développons par la suite une approche pour détecter les déviations dans la transition des métriques dans le chapitre 5. On discute par la suite de l'ensemble de nos résultats dans le chapitre 6 et finissons par présenter une synthèse, les limitations de nos travaux et nos recommandations pour de futures améliorations dans le chapitre 7.

CHAPITRE 2 REVUE DE LITTÉRATURE

L'automatisation des tests de régression logiciel à l'aide d'un modèle qui exprime le comportement normal d'un logiciel est un sujet de recherche couvrant plusieurs sphères allant de l'évaluation des performances du matériel de l'ordinateur qui exécute le logiciel à l'évaluation des transitions d'états dans un logiciel en utilisant des automates finis. Dans chacun de ces cas nous utilisons un algorithme d'apprentissage machine, une analyse statistique ou une méthode manuelle afin de modéliser le comportement normal du logiciel testé. Nous comparons par la suite les métriques recueillies durant les tests au modèle du système afin de déterminer le succès ou non d'un test. Dans les tests sur les simulateurs de vol, nous analysons automatiquement les métriques enregistrées pendant une ou plusieurs séances de pilotage. Un écart au-delà du niveau de seuil (déterminé dans une calibration initiale) de ces métriques par rapport à leur modèle est considéré comme une déviation, signifiant ainsi que le système a un problème fonctionnel.

Nous procédons donc dans la section suivante à une revue de la littérature dans le domaine des tests logiciels et sur ce qui se fait actuellement dans l'industrie aéronautique afin de bâtir notre méthodologie d'automatisation de l'analyse des résultats de test dans les simulateurs de vol.

2.1 L'utilisation des simulateurs de vol

Les simulateurs de vol occupent une partie essentielle de la formation des nouveaux pilotes puisque la simulation permet de réduire les coûts et les risques que comportent les entraînements effectués dans des vrais vols. Il s'avère même trop risqué ou impossible d'utiliser de vrais appareils pour recréer certaines situations critiques (p. ex., une tempête violente) auxquelles les pilotes pourraient faire face durant leur carrière. Même sans les technologies d'aujourd'hui, l'efficacité et l'utilité de l'entraînement avec les simulateurs de vol a été prouvée dans le passé par plusieurs études [1], il y a un avantage significatif à combiner les exercices de simulation avec l'utilisation de vrais appareils lors de la formation. Il est notamment possible avec la simulation de concentrer l'entraînement sur des tâche spécifiques (p. ex., le décollage, la croisière ou l'atterrissage d'un avion), ce qui n'est pas possible de faire dans le cadre de vrais vols. Avec un très haut degré de réalisme, les simulateurs de vols occupent une part très importante de la formation des futurs pilotes et les simulateurs sont maintenant utilisés pour tester la conception de vrais appareils afin de réduire les coûts élevés de prototypage [2]. Il est donc primordial d'assurer la qualité des simulateurs en ayant recours à une batterie

exhaustive de tests sur chacune des nouvelles versions.

2.2 Tests d'acceptation dans les simulateurs de vol

Du fait de leur importance, les simulateurs de vol sont testés comme les vrais avions où un pilote contrôle le simulateur en parcourant une liste de procédures écrites dans un Manuel de Tests d'Acceptation (MTA) qui est préalablement préparé par le manufacturier du simulateur avec l'approbation du client final. Ces tests visent l'acceptation du produit par le client final, d'où le terme tests d'acceptation. Chacune des procédures inclue des opérations (p. ex., tirer un levier et appuyer sur un bouton) et une vérification visuelle (p. ex., vérifier le niveau d'un cadran). Le pilote réalise également des scénarios de vol pendant lesquels il évalue de manière subjective le comportement du simulateur, des métriques de vol sont également enregistrées (p. ex., la vitesse de l'avion ou bien l'enveloppe d'un signal de contrôle). À partir de ces métriques, l'équipe de test évalue les comportements de l'aéronef qui sont reproduits dans le simulateur, ce qui inclut en grande partie : l'exactitude du stimulus électrique dans les contrôles, les caractéristiques temporelles et d'amplitude des réponses des actuateurs, la stabilité des boucles de contrôles et la position des surfaces [3].

Une bonne partie des tests visent à vérifier si le simulateur rencontre les normes mises en place dans le Guide d'Essais de Qualification (GEQ) émis par les autorités d'état afin de valider que les performances et la pilotabilité du simulateur sont conformes aux limites de l'aéronef. Le MTA approuvé par le client final du simulateur vise entre autres à assurer la conformité du simulateur avec le GEQ. Le GEQ est la référence pour tester l'ensemble des simulateurs de vol dans l'industrie indépendamment des requis spécifiques au client. Un maximum de données provenant du simulateur sont enregistrées durant les différentes séances de test (p. ex., la vitesse de vol) afin de vérifier si elles sont conformes aux spécifications de l'aéronef mises en valeur dans le GEQ. Des éléments plus subjectifs, tels que l'appréciation générale du produit, peuvent être évalués durant les tests d'acceptation. Notamment, le GEQ du gouvernement canadien est très explicite sur l'obligation pour les manufacturiers d'exécuter des tests à chaque nouvelle version du simulateur : "Les exploitants doivent prévoir un système de contrôle de la configuration du logiciel et du matériel afin de s'assurer que le simulateur satisfait toujours aux critères de performance de l'évaluation initiale." [4]. Également, on y spécifie clairement qu'une non-conformité avec les caractéristiques de vol de l'avion que l'on simule ou avec une autre spécification mène systématiquement à un arrêt des opérations d'entraînement des pilotes sur le simulateur, ce qui engendre des pertes de revenu et un retard sur l'entraînement des pilotes. C'est pour ces raisons que les tests sur les simulateurs sont un enjeu majeur pour l'industrie et que leur optimisation peut rapporter des réductions

de coût importantes pour la mise en production des simulateurs. Il est possible d'optimiser les tests réalisés sur les logiciels des simulateurs de vol avec l'utilisation de scripts de contrôle et l'analyse automatique des résultats de test.

2.3 Les tests dans les logiciels

2.3.1 Les ordres de grandeur dans les tests

Similaire aux organisations développant des systèmes logiciels grande échelle faits de plusieurs composants, les producteurs de simulateurs séparent généralement les tests logiciels en différentes familles visant à tester le système à plusieurs niveaux de granularité (p. ex., une fonction ou un composant complet) en utilisant des infrastructures et stratégies de test différentes. . Par exemple, Google divise ses tests en trois catégories [5], soient : les tests petits, les tests moyens, et les tests grands. La figure 2.1 montre comment les phases de test s'enchaînent. Les tests petits sont des tests unitaires mis en place par le développeur lui même afin de tester le bon fonctionnement des fonctions et composants qu'il développe, un test unitaire ne prend généralement que quelques secondes à s'exécuter. Les tests moyens couvrent l'intégration de plusieurs composants travaillant de concert dans le logiciel, ces tests peuvent prendre quelques minutes à être exécutés et requièrent l'expertise d'ingénieurs testeurs parce qu'ils couvrent le code de plusieurs développeurs. Les tests grands englobent tout le système logiciel où les ingénieurs de test conçoivent des scénarios de test qui présentent des cas réels d'utilisation du système par les utilisateurs afin de valider si le logiciel est prêt pour la production. Le logiciel en production est utilisé par des millions d'utilisateurs, voir des milliards. Dans le cas où il faut rencontrer les requis du client, il y a également les tests d'acceptation, qui consistent à valider si le logiciel offre la bonne expérience à l'utilisateur ou est conforme aux standards.

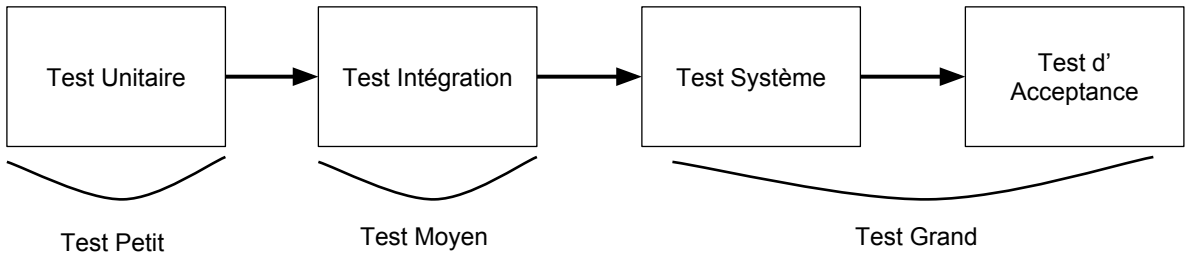


Figure 2.1 Enchaînement des différents niveaux du processus de test avec l'ordre de grandeur de chacune des phases de petit à grand.

Le principal défi de test auquel les entreprises développant les simulateurs de vol font face est au niveau des tests systèmes et des tests d'acceptance. Tel que mentionné précédemment, ces deux niveaux de test visent à assurer que le simulateur est conforme au MTA du manufacturier de l'avion d'origine et doivent suivre les lignes directrices dans le GEQ émis par les autorités d'état. Dans ces procédures de vérification de la conformité du simulateur, il faut s'assurer que le comportement d'une nouvelle version du simulateur est le même que la dernière version (qui est déjà certifiée). Pour l'instant, les tests d'acceptation sont la plupart faits manuellement par des pilotes professionnels qui exécutent une à une les opérations inscrites dans les manuels de test. Ces pilotes font également part de leur appréciation générale du simulateur. Le but fondamental de toutes ces procédures est de valider si le comportement d'une nouvelle version du simulateur est le même que la version précédente. Nous proposons donc de modéliser avec de l'apprentissage automatique une version du simulateur qui est déjà certifiée. Par la suite, il sera possible de comparer les métriques enregistrées lors des tests sur le simulateur avec le modèle de référence, tout écart au-delà d'un niveau de seuil par rapport au modèle est considéré comme une déviation et est signe d'un problème fonctionnel. Outre l'analyse automatique des résultats de test, il est possible d'automatiser les opérations que l'on fait sur le simulateur. En effet, la majorité des simulateurs de vol possèdent des outils d'automatisation des opérations de vol avec un environnement dans lequel il est possible de programmer des séquences d'opérations en utilisant des scripts. Par exemple, il est possible de programmer un vol complet entre deux aéroports. Nous pourrions donc réduire les coûts des tests systèmes et d'acceptation dans les simulateurs de vol en appliquant une stratégie de test complètement automatisée de type boîte noire visant à détecter les déviations.

2.3.2 Les tests de régression

Comme une entreprise de simulateurs de vol est aussi une entreprise logicielle, les tests exécutés sur les logiciels des simulateurs de vol suivent l'état de l'art dans le domaine des tests de régression logiciel. Ces tests de régression ont pour but de valider si le comportement d'une nouvelle version d'un logiciel est normal, attestant ainsi la qualité et la sécurité des fonctions qui existaient avant la mise à jour du système. Ces tests vérifient également le bon fonctionnement des nouvelles fonctionnalités ajoutées au logiciel, ce qui nécessite d'élaborer de nouveaux cas de test. De nos jours, la mise en production des nouvelles versions d'un logiciel est de plus en plus rapide avec la demande croissante du marché pour de nouvelles fonctionnalités, l'automatisation des tests devient donc une question de survie afin de maintenir une structure de coût d'entretien des logiciels raisonnable et compétitive. L'industrie a donc développée des outils d'intégration en continue exécutant une série de cas de test sur chacun des dépôts fait dans les systèmes de gestion de versions. Jenkins [6] est un exemple

d'outil d'intégration en continue avec lequel il est possible de vérifier si le code compile correctement et de vérifier si son fonctionnement est encore adéquat en jouant un jeu de scripts de test codé par des experts en test, il faut bien faire attention à faire évoluer les tests en même temps que les nouvelles fonctionnalités développées. Les tests dans Jenkins sont de type boîte blanche, donc ils sont exécutés sur des composants développés à l'interne dans l'organisation qui développe le logiciel, ce qui ne peut être appliqué dans notre cas parce que nous n'avons pas accès au code source. On peut également développer des infrastructures de test sur mesure afin de tester des aspects particuliers d'un logiciel, p. ex., une infrastructure faisant des tests de charge sur différents services d'un système logiciel. Un test peut être aussi simple qu'une routine vérifiant la validité du retour d'une fonction en injectant des paramètres d'entrée connus. Un test peut également couvrir une gamme de fonctionnalités plus large avec l'exécution d'un scénario d'utilisation du logiciel, ce qui permet de valider son comportement général une fois l'intégration de ses composants complétée.

Les stratégies de test boîte blanche ne sont pas applicables aux simulateurs de vol, il est nécessaire dans ce cas d'utiliser une approche boîte noire qui est plus difficile à automatiser puisqu'il est nécessaire de générer des cas d'utilisation du système en entier à la place de paramétrer une fonction. Au-delà du code source lui même, ces tests doivent couvrir l'ensemble des comportements que l'on attend lors de l'utilisation du système. Dans plusieurs cas, l'industrie utilise encore des testeurs et du personnel en assurance qualité afin de stimulé manuellement le système et d'ainsi voir si le système répond aux attentes. L'industrie met donc actuellement des efforts pour automatiser les tests de type boîte noire [7]. La technique consiste à stimuler le système logiciel tout en enregistrant divers métriques afin d'analyser son comportement et ainsi valider le bon fonctionnement du logiciel. Afin d'être productif, les développeurs mettent au point des outils d'automatisation des tests de type boîte noire qui stimulent un logiciel en utilisant son interface d'entrée et valide par la suite la correctitude des métriques de sortie enregistrées par rapport à un gabarit. Par exemple, Mariani et al. [8] développent l'outil AutoBlackTest d'automatisation de test sur les logiciels avec interface graphique dont le code source n'est pas disponible. L'outil interagit avec l'interface en explorant ses états de la même manière que le ferait un utilisateur. Une étude empirique montre que AutoBlackTest peut exécuter une grande portion du code source de l'application testée, et ce, en interagissant seulement avec l'interface accessible à l'utilisateur. Les auteurs utilisent la couverture du code comme critère d'évaluation de la performance de l'approche, ils assument qu'une grande couverture du code permet de valider la plupart des fonctionnalités. L'utilisation de l'outil permet de stimulé adéquatement un système composé de boîtes noires. Par contre, les auteurs n'utilisent pas de modèle afin d'automatiser complètement le processus d'analyse des résultats de test. Une méthode classique de vérification des résultats

de test est de vérifier si les métriques analysés correspondent bel et bien à une valeur que l'on prédétermine lors de la planification des tests. Cette méthodologie nécessite une compréhension complète de toutes les fonctionnalités du logiciel et ne peut être utilisée pour remplacer les analyses subjectives réalisées par les testeurs (p. ex., vérifier le rendu d'un objet dans un éditeur graphique). Ces analyses subjectives s'appliquent également aux simulateurs de vol puisqu'une partie de l'évaluation du système durant les tests consiste en une évaluation d'appréciation globale du simulateur de la part du pilote utilisant le système, car il est difficile d'analyser de manière quantitative les centaines de métriques du simulateur à chaque fois. En effet, les testeurs n'ont pas accès à toute la documentation nécessaire de la part des fournisseurs des composants de type boîte noire pour évaluer chacune des métriques. Il est possible automatiser l'analyse de quelques métriques en utilisant des gabarits que l'on construit manuellement, mais cela n'est pas suffisant pour valider le bon fonctionnement du simulateur en entier. Si nous voulons automatiser l'analyse des résultats de test, il est donc nécessaire d'avoir une approche en mesure d'évaluer d'elle même s'il y a déviation du comportement des métriques par rapport à leur comportement d'origine.

En ce sens, Chandola et al. ont publié une revue de la littérature sur la détection de déviations dans l'analyse des résultats de test [9]. Les auteurs ont groupé ces techniques d'analyse en six grandes catégories : par classification, par partitionement, par voisin proche, par statistique, par théorie de l'information et par spectre. Chacune de ces techniques cherche à déterminer le comportement normal d'un logiciel où tout écart au-delà d'un certain seuil dans les métriques enregistrées par rapport à leur comportement normal est considéré comme une déviation. La détection des déviations comporte donc deux principaux défis, soient de définir le comportement normal d'un système et de déterminer quel est le niveaux d'écart de comportement qui représente une déviation. Aggarwal et al. propose à cet effet un algorithme d'analyse statistique en temps réel sur le flux de données provenant d'un logiciel afin de détecter des déviations en utilisant un historique de données comme référence. Cette méthode de test a pour avantage de vérifier la santé du système en temps réel, dans le cadre de ce mémoire nous procédons à l'analyse des résultats pour détecter les déviations après l'exécution des tests. Pour déterminer le comportement normal d'un système, les testeurs utilisent dans tous les cas un historique de données collecté lors de précédentes exécutions du système. Cela est l'état de l'art de la détection des déviations dans les logiciels.

2.3.3 La détection des déviations dans la performance des logiciels

La détection des déviations dans les métriques des simulateurs de vol est similaire à ce qui se fait en ce moment avec les tests de régressions de performance [10, 11, 12, 13]. Ces tests

consistent à détecter des déviations dans la performance d'un système pendant l'exécution d'une nouvelle version d'un logiciel. Les métriques enregistrés peuvent par exemple être le taux d'utilisation du processeur, la quantité de mémoire *Random Access Memory (RAM)* utilisée, la quantité d'écritures et de lectures sur le disque, etc. Cette méthodologie de test de régression de performance consiste à enregistrer des métriques de performance pendant une séance d'utilisation du logiciel et de comparer par la suite les métriques enregistrés à un historique de métriques accumulés sous l'utilisation d'une version du logiciel ayant un comportement normal. L'approche de test des auteurs détecte par la suite s'il y régression de performance du logiciel en mesurant l'écart entre les métriques des deux version comparées, tout écart dépassant un certain seuil est alors considéré comme une déviation sur laquelle il faut porter attention.

Foo et al. [10] utilisent des règles associatives et des seuils afin de détecter automatiquement des régressions de performance dans des applications serveurs. Divers scénarios de test sont utilisés afin de générer de nouvelles versions du système et certaines de ces versions ont des métriques de performance déviant. Une règle associative consiste à associer un ensemble de valeurs de métriques qui occurrent fréquemment ensemble. Par exemple, une règle associative de performance normale dans un système serveur pourrait être "Traffic Réseau (**Élevé**) + Taille Paquet (**Lourd**) => Taux Écriture/Lecture (**Elevé**) + Utilisation RAM (**Élevée**)". Une règle déviante pourrait donc être : "Traffic Réseau (**Élevé**) + Taille Paquet (**Lourd**) => Taux Écriture/Lecture (**Faible**) + Utilisation RAM (**Élevée**)". Donc on pourrait présumé dans ce cas que la mémoire tampon sur le disque est moins utilisée pour la même charge sur le réseau, ce qui peut être présage d'un mauvais fonctionnement du logiciel du serveur. Puisqu'une proportion minime de règles de performance déviantes ne pose probablement pas problème, les auteurs choisissent un niveau de seuil de déviation juste au dessus de la proportion de règles qui sont normalement déviantes. Un rapport synthétisant le résultat des tests est produit afin que les testeurs puissent rapidement identifier les métriques qui présentent des déviations.

D'autres travaux extraient les logs textuels (p. ex., "Le client a été authentifié avec succès") provenant des exécutions afin d'identifier des déviations dans le comportement d'une nouvelle version d'un logiciel par rapport à une référence. Jiang et al. [11] présentent une approche statistique qui analyse les logs enregistrés pendant des tests de charge pour détecter des pertes de performance. Syer et al. [14] proposent une approche statistique qui combine les logs d'exécutions et des métriques de performance afin d'identifier les problèmes de mémoire lors des tests de charge. Beschastnikh et al. [15] développent l'outil CSight pour extraire les logs provenant de l'exécution d'un logiciel afin de bâtir un modèle comportemental sous forme d'un automate fini, facilitant ainsi la compréhension du système et l'identification de

déviations. Le travail est une évolution de Daikon [16], un outil de détection des invariants dans les traces systèmes. Toute déviation d'une trace par rapport à un invariant est considérée comme un problème.

Notre travail diffère de ces précédents travaux portés sur les logs et les métriques de performance puisque nous utilisons des métriques spécifiques aux simulateurs de vol au lieu des logiciels Web ou mobiles, ce qui est un contexte différent. Au lieu de tester la performance, nous visons à tester si le comportement de ces simulateurs est adéquat. De plus, nous choisissons de modéliser notre système avec des arbres de décision, ce qui offre des modèles avec abstraction graphique plus intuitifs à comprendre par des testeurs ou ingénieurs [17].

D'autres études sur les tests dans les simulateurs de vol portent sur la prédiction des erreurs de configuration. Une grande partie du fonctionnement d'un simulateur de vol repose sur la manière dont l'environnement virtuel et les différents composants de l'aéronef sont configurés. Par exemple, il est possible de changer les coefficients des différentes équations mathématiques modélisant le système hydraulique. Une erreur de configuration a donc de fortes chances de détériorer le fonctionnement d'un système. Hoseini et al. [18] proposent la technique FELODE (*Feature Location for Debugging*) pour détecter les erreurs de configuration dans les composants d'un simulateur en utilisant ses traces d'exécution et l'opinion d'un expert du système afin d'identifier les composants qui sont source de problèmes. Les auteurs appliquent leur approche sur un simulateur de vol de l'entreprise CAE pour détecter les erreurs de configuration en obtenant une précision moyenne de 50% et un rappel pouvant aller jusqu'à 100%. Cette recherche est celle qui s'approche le plus du domaine d'application de notre mémoire, même si nous mettons l'accent sur la détection de problèmes fonctionnels à la place d'identifier les mauvaises configurations. Comme cette étude, nous visons à faire une recommendation aux testeurs de manière automatique pour leur faire savoir si le simulateur comporte un problème.

2.4 Détection des déviations dans les données de vol

Les transporteurs aériens commerciaux possèdent d'énormes bases de données provenant de l'instrumentation de vol dont leur analyse peut être mise à profit afin d'optimiser les procédures de maintenance, d'assurer une meilleur qualité de service et d'identifier des comportements coûteux en carburant. Certaines données sont même mises à la disposition du publique, c'est le cas des données sur les itinéraires de vol. Certains sites Web comme Flight Radar 24 [1] rassemblent certaines données de vol de l'ensemble des transporteurs aériens dans

1. http ://www.flightradar24.com/data/

le monde afin d'offrir un visuel sur les itinéraires de vol en cours tout autour du globe.

Les données collectées durant les itinéraires de vol sont utilisées dans le processus d'évaluation de la *Flight operations quality assurance (FOQA)* [19], qui consiste, outre la collecte de données brutes, à faire une analyse poussée de ces données en utilisant des méthodes statistiques, des méthodes de visualisation et des méthodes d'apprentissage machine pour faire de la maintenance préventive et pour identifier la cause des problèmes fonctionnels. En déterminant une tendance normale dans les données, il est possible d'identifier des données qui ont un comportement anormal. L'analyse de ces déviations permet de détecter des problèmes fonctionnels de l'aéronef.

Callantine et al. [20] explorent le potentiel d'utiliser une approche basée sur la modélisation pour identifier la cause des déviations détectées dans les données de vols par rapport aux activités de l'équipage dans le cadre de la FOQA, afin d'améliorer la sécurité et la formation de l'équipage. Les auteurs effectuent leur première expérimentation sur des données de vol provenant d'un Boeing 757. Li et al. [21] proposent une méthode de soutient à la FOQA pour détecter automatiquement les anomalies dans les données de vol sans avoir recourt à un seuil de détection que l'on détermine au préalable. La technique consiste à extraire pour chacun des vols un vecteur constitué de tous les paramètres de vol et de former par la suite des clusters de vecteurs représentant des vols nominaux. Un vol est considéré comme anormal lorsque son vecteur n'appartient pas au cluster. Les auteurs sont en mesure de détecter de vraies anomalies dans les données de vol d'un Boeing 777.

Das et al. [22] utilisent un algorithme de filtrage de dynamique symbolique où les séries temporelles provenant de la FOQA sont d'abord transformées en séquences de symboles et par la suite utilisées pour reconstruire un automate fini dans lequel un symbol représente un état de vol. Le rendu final offre un niveau d'abstraction qui permet de comprendre la relation entre les différents états de vol. Il est ainsi possible de détecter de potentielles anomalies dans les données de certains vols en comparant l'automate fini que l'on en extrait avec un automate fini de référence que l'on bâti en utilisant un large historique de données représentant une tendance normale. En comparant leur approche avec l'état de l'art de la détection d'anomalies dans les séries temporelles, les auteurs obtiennent de meilleures performances de détection. Ils découvrent même rétroactivement des vols ayant eu des anomalies uniques. Stolzer et al. [23] proposent d'utiliser l'apprentissage machine sur les données de vol afin d'en extraire un modèle de la consommation de carburant, ce qui permet par la suite d'ajuster le pilotage d'un aéronef afin d'en réduire sa consommation. Les auteurs obtiennent un modèle pouvant expliquer jusqu'à 85% de la consommation d'essence et avec des coefficients de corrélation proche des 99% avec l'utilisation des réseaux de neurones.

Dans ce mémoire, nous cherchons à utiliser le même genre de données qu'obtenues lors de la FOQA et de l'apprentissage machine afin de détecter les problèmes fonctionnels des simulateurs de vol.

2.5 Les simulateurs de vols logiciels

Certains simulateurs de vol offrent des environnements complètement virtuels à des fins de recherche ou pour le divertissement que l'on peut installer sur son ordinateur domestique. Ces simulateurs sont de loin moins coûteux que ceux qui intègrent le vrai habitacle d'un aéronef, mais offrent un degré de réalisme moindre. Parmi les principaux simulateurs de vol commerciaux disponible pour les ordinateurs domestiques, se trouve : Microsoft Flight Simulator et X-Plane 10. Ces simulateurs offrent un rendu visuel très poussé et on peut même accréditer certaines heures de vol sur ces simulateurs pour les petits avions récréatifs (p. ex., un petit Cessna), par contre le code source est fermé, ce qui n'est pas idéal pour réaliser nos expérimentations. Il existe également FlightGear, qui est un logiciel ouvert multiplateforme de simulation de vol qui a débuté en avril 1996 [24]. Le logiciel va au-delà d'un modèle simpliste d'aérodynamique de vol en offrant des options de configuration de l'environnement et un éventail de modèles d'aéronefs développés par la communauté. Le simulateur possède un moteur graphique capable de recréer l'environnement de vol, comme le montre la figure 2.2. Il est possible de piloter des vols tout au tour du monde, de changer les conditions de vol (p. ex., la météo), d'émuler des situations critiques de vol où certains systèmes font défauts. La communauté a modélisé la plupart des aéronefs les plus populaires dans l'industrie, tels que le Boeing 747, l'Airbus A380, le CRJ700 de Bombardier, le Cessna 172, et même le légendaire Spitfire.

Figure 2.2 Le simulateur de vol FlightGear.

FlightGear utilise les bibliothèques de SimGear pour construire son moteur de simulation. Ces bibliothèques ont pour but d'offrir une solution versatile de simulation pouvant aller au-delà de la simulation de vol, même si ce projet a démarré pour supporter le développement de FlightGear. SimGear offre une interface permettant de concevoir les panneaux d'instrumentation et de contrôle, un calculateur de position, une interface de communication réseau, une interface pour interagir avec les périphérique (p. ex., le son et les contrôles), et un environnement graphique [25]. Cette bibliothèque est utilisée pour développer divers modèles d'aéronefs [26, 27, 28].

Certains projets utilisent conjointement SimGear et des bibliothèques spécialisées dans la dynamique de vol, telles que JSBsim. Cette bibliothèque ouverte est développée par des institutions en aéronautique tels que la NASA afin de simuler des modèles de vol dans le cadre de travaux de recherche de pointe. La communauté de ses institutions fait des mise-à-jours fréquentes. Ce qui distingue vraiment FlightGear de JSBsim est son rendu graphique. Les modèles d'aéronef de FlightGear sont constitués de plusieurs composants qui encapsulent principalement :

- la logique de contrôle ;
- les éléments du cockpit ;
- les équations mathématiques de la dynamique de vol ;
- Le groupe propulseur ;

- le système électrique ;
- le système hydraulique ;
- l'instrumentation ;
- les systèmes d'assistance au pilotage (p. ex., le pilote automatique) ;
- les système de communication.

Tous les composants des modèles d'aéronef et de la plateforme de simulation communiquent par le biais d'une base de données partagée qui rend accessible publiquement à la fois les données se transigeant entre composants et celles exposées aux interfaces publiques (p. ex., les contrôles dans le cockpit et les coordonnées de position de l'aéronef). Tout ce qui est contenu dans la base de données peut être enregistré dans un format de type *Comma-Separated Values (CSV)* sous forme de séries temporelles en utilisant une fréquence d'échantillonnage au choix. Chaque ligne dans le fichier correspond à une capture à un moment spécifique de la valeur de chacune des métriques de la base de données que nous avons choisies d'enregistrer. Il est même possible de transmettre les données en utilisant différents protocoles de communication tel que TCP ou UDP.

Les données sont accessibles en empruntant une arborescence de chemins similaire à celle des systèmes de fichier dans les systèmes d'exploitation, on part ainsi d'une étiquette générique à la racine identifiant un système particulier jusqu'à la feuille qui est une étiquette identifiant chaque variable unique. Nous appelons cette arborescence l'arbre de propriétés. La figure 2.3 montre un exemple de variables contenues dans l'arbre de propriétés. Par exemple, /position/ est un répertoire contenant les propriétés de longitude, latitude et d'altitude. Les technologies SimGear, FlightGear et JSBsim utilisent tous le format *Extensible Markup Language (XML)* afin de configurer initialement les modèles d'aéronef, les paramètres de simulation et les différents panneaux du cockpit. L'arborescence du fichier XML est calquée sur celle de la base de données afin de simplifier le chargement et l'échange des données. La figure 2.4 montre un exemple de configuration XML utilisée pour initialiser l'arbre de propriétés où la valeur d'altitude est contenue dans l'arborescence /position/ -> altitude-ft, ce qui calque le chemin <position> -> <altitude-ft> montré à la figure 2.3.

```
/sim/aircraft = A333
/position/
/position/longitude-deg = “-122.3576677” (double)
/position/latitude-deg = “37.61372424” (double)
/position/altitude-ft = “28.24418581” (double)
/position/altitude-agl-ft = “22.47049626” (double)
/controls/seat/eject/initiate
/controls/electric/APU-generator
```

Figure 2.3 Extrait de l’arbre de propriétés de FlightGear.

```
<position>
<altitude-ft>10000</altitude-ft>
<latitude-deg>-117.00</latitude-deg>
<longitude-deg>35</longitude-deg>
</position>
```

Figure 2.4 Fichier XML de configuration de l’arbre de propriétés de FlightGear.

2.6 Sommaire de la revue de la littérature

Lors de notre revue de la littérature, nous avons tout d’abord expliqué l’utilité des simulateurs et pourquoi il est primordiale d’assurer leur qualité tout au long de leur développement et utilisation. L’industrie de la simulation de vol et les autorités d’état ont développé des normes rigoureuses qui doivent être suivies à la lettre lors de la mise en opération du simulateur, tout écart par rapport aux normes empêche sa mise en opération. Nous sommes donc intéressé à automatiser l’analyse des résultats de test de la partie logicielle d’un simulateur de vol afin d’accélérer le processus d’assurance de conformité de son fonctionnement. Une telle approche permettrait à l’industrie de réduire ses coûts reliés au processus de test.

Nous procédons donc par la suite à une exploration des méthodologies existantes dans les tests de régression logiciel durant laquelle nous soulignons qu’il y a différents niveaux de test. Chacun de ces niveaux couvre une différente granularité du logiciel en allant d’une fonction simple au système en entier. Afin de choisir la meilleure stratégie de test applicable aux simulateurs de vol, nous avons vérifié l’état de l’art des tests qui se font dans les logiciels qui sont composés de composants de type boîte noire.

Des études dans le domaine des tests logiciels au niveau système montrent qu’il est possible

d'automatiser la détection de problèmes fonctionnels d'un logiciel en analysant des métriques de performance qui sont enregistrées au cours de l'exécution d'un scénario d'utilisation. Des métriques présentant des déviations par rapport à leur valeur normale peuvent causer un problème fonctionnel. Les métriques peuvent être reliées à la performance du matériel utilisé pour exécuter un logiciel où à l'information que l'on tire des logs d'exécution du logiciel. L'utilisation de métriques provenant des données d'un logiciel semble un choix intuitif pour automatiser l'analyse des résultats de test du système logiciel d'un simulateur de vol. Nous explorons par la suite les techniques de la FOQA qui sont actuellement utilisées par les transporteurs aériens commerciaux sur leurs données de vol. Nous réalisons que l'analyse des données enregistrées lors de la FOQA utilise presque la même méthodologie que pour l'automatisation des analyses des résultats de test dans les logiciels.

Pour réaliser une étude des cas, il existe différents logiciels de simulation de vol que l'on peut utiliser sur notre ordinateur personnel. FlightGear est le plus adéquat pour faire nos expérimentations puisque le projet est très populaire et son code source est ouvert. Il est également possible d'enregistrer toutes les données de vol contenues dans l'arbre de propriétés du simulateur. Nous pouvons donc appliquer dans FlightGear l'état de l'art de l'analyse des résultats de test logiciel.

Les travaux de recherche à ce jour dans le domaine de l'aérospatiale se sont concentrés sur l'analyse des données de la FOQA pour la maintenance des aéronefs [23, 20, 21, 22, 23]. Par contre, peu de travaux concernent la détection automatiquement de problèmes fonctionnels dans les nouvelles versions des simulateurs de vol en utilisant une méthodologie de détection de déviation des métriques contenus dans le système.

CHAPITRE 3 DEMARCHE DE L'ENSEMBLE DU TRAVAIL DE RECHERCHE

Dans le cadre de ce mémoire, nous avons produit et soumis un article de journal s'intitulant : *Signature-based Detection of Behavioural Deviations in Flight Simulators – Case Study on FlightGear*. Dans cet article, nous proposons une approche détectant automatiquement les déviations dans les métriques enregistrées d'une version de référence (N) d'un simulateur de vol et d'une nouvelle version (N + 1) du simulateur.

Puisque nous voulons avoir un modèle de qualité, nous sélectionnons d'abord avec l'aide d'experts des métriques d'entrée et de sortie parmi les 148 métriques que nous pouvons enregistrer dans le simulateur. Cette étape de sélection doit se faire une fois, conjointement avec des experts du domaine afin de s'assurer que nos métriques produisent le meilleur modèle comportemental du système. Comme première exploration du sujet de recherche, nous préférons modéliser le système en entier à la place de modéliser chacun de ses composants puisque cela requiert moins d'expertise sur un composant en particulier. En effet, quand l'on veut modéliser un composant en particulier, il faut choisir des métriques d'entrée et de sortie qui expliquent bien le composant en question, ce qui requiert un rare niveau d'expertise plus approfondi. Il est possible d'utiliser des algorithmes de sélection automatique des métriques pour former un modèle, mais le modèle est moins représentatif du simulateur dans la plupart des cas. Nous nous fions donc aux experts pour bâtir le modèle du simulateur.

Comme technique d'apprentissage automatique, nous utilisons un arbre de décision pour chacune des observations (c.-à.d., métriques de sortie) que nous voulons expliquer en utilisant un ensemble de données en entrée (c.-à.d., métriques d'entrée). L'arbre est composé de noeuds de décision correspondant chacun à une métrique d'entrée et de feuilles qui contiennent la catégorie finale de sortie (qui englobe un interval de valeurs). Les noeuds et les feuilles sont interconnectés avec des branches qui correspondent à la valeur du métrique dans le noeud d'origine. L'arbre de décision possède une abstraction très visuelle qui permet de bien comprendre le système modélisé [17].

Nous générons les arbres de décisions à partir des données que nous enregistrons en pilotant un vol. Pour une version d'un système donnée, l'exécution du même vol ne donne pas exactement les mêmes données, nous devons donc analyser l'écart naturel des données provenant de la version de référence. Une fois que nous avons une idée de la plage d'écart naturel des métriques de sortie par rapport à leur modèle, nous choisissons un niveau de seuil en haut de la plage d'écart pour seulement détecter les écarts au-delà de la normalité comme étant des déviations,

nous voulons ainsi éviter de générer trop de fausses alarmes. Nous avons d'abord choisi un niveau de seuil commun à toutes les métriques de sorties, mais nous discutons par la suite de la pertinence de spécialiser le niveau de seuil pour chacune des versions. Cela donne donc une flexibilité d'ajustement du niveau de seuil avenant le cas où l'échantillonnage initial n'est pas suffisant.

L'ensemble des données utilisées pour simuler la dynamique de vol et les divers équipements d'un aéronef sont facilement accessibles par le biais d'un arbre de propriétés. Dans l'application de notre approche sur le simulateur FlightGear, nous avons d'abord élaboré un plan d'un vol que nous exécutons une première fois afin de générer un arbre de décision pour chacune des métriques de sortie. Nous avons consulté des experts de la communauté FlightGear afin de savoir quelles étaient les métriques de sortie et d'entrée pertinentes à choisir pour avoir un modèle représentatif du simulateur. Nous exécutons par la suite plusieurs fois le même vol afin de déterminer le niveau de seuil qui discrimine entre un écart normal et une déviation. À ce point nous pouvons automatiquement détecter les déviations dans les autres versions du simulateur.

Afin d'évaluer la performance de notre approche pour détecter les déviations, nous produisons artificiellement cinq nouvelles versions de FlightGear à partir d'une version de référence. Pour ce faire, soit nous changeons la configuration de l'environnement du simulateur, soit nous utilisons les outils existant d'injection de fautes systèmes pour reproduire des situations de vol critiques. Il y a deux configurations changeant l'environnement et trois injections de fautes dans le système de l'aéronef simulé. Quatre de ces modifications contiennent des déviations et l'autre modification qui a trait à l'environnement ne doit pas causer de déviations.

Suite aux expérimentations de cet article, nous répondons à nos trois questions de recherche. Pour QR1, nous sommes en mesure de reproduire un modèle fiable du simulateur en sélectionnant avec l'aide d'experts des métriques d'entrée et de sortie. Pour QR2, notre modèle est sensible aux variations dans le comportement du simulateur, par exemple les métriques de la version modélisée ont des écarts allant jusqu'à 41% par rapport au modèle de base. Pour QR3, notre approche détecte automatiquement les fautes reliées à l'environnement sans fausses alarmes pour le faible vent et détecte toutes les déviations avec une précision de 50% pour le fort vent. Pour ce qui est des fautes systèmes, nous sommes en mesure de détecter la majorité des déviations avec une précision de 100%, nous avons un seul cas à 67%. Nous n'avons pas comparer cette évaluation à un classificateur aléatoire parce que notre premier critère est d'avoir un rappel le plus haut possible, donc un classificateur aléatoire viable identifierait toutes les métriques comme ayant des déviations. Puisque nous avons un rappel de 100% dans toutes les versions, notre approche peut soit égaliser ou battre la classificateur

aléatoire.

Cet article couvre les trois objectifs de notre mémoire : soit de modéliser un simulateur en utilisant de l'apprentissage automatique, de détecter des écarts dans les métriques de différentes versions d'un simulateur de vol et de détecter automatiquement les déviations lors de l'analyse des résultats de test sur d'autres versions du simulateur.

CHAPITRE 4 SIGNATURE-BASED DETECTION OF BEHAVIOURAL DEVIATIONS IN FLIGHT SIMULATORS – CASE STUDY ON FLIGHTGEAR

Authors

Vincent Boisselle
École Polytechnique de Montréal
vincent.boisselle@polymtl.ca

Giuseppe Destefanis
CRIM, Montreal
giuseppe.destefanis@crim.ca

Bram Adams
Polytechnique Montreal
bram.adams@polymtl.ca

Submitted to
Journal of Systems and Software, August 27th, 2015

Abstract

Flight simulators are systems composed of numerous off-the-shelf components that allow pilots and maintenance crew to prepare for common and emergency flight procedures for a given aircraft model. A simulator must follow severe safety specifications to guarantee correct behaviour and requires an extensive series of prolonged manual tests to identify bugs or safety issues. In order to reduce the time required to test a new simulator version, this paper presents rule-based models able to automatically identify unexpected behaviour (deviations). The models represent signature trends in the behaviour of a successful simulator version that are compared to the behaviour of a new simulator version. Empirical analysis on five types of injected faults in the popular FlightGear open source simulator shows that our approach can obtain an acceptable precision of 50% and higher, without missing any deviation.

4.1 Introduction

Aircraft simulators [29] reproduce common scenarios such as “take-off", “automatic cruising" and “emergency descent" in a virtual environment [30]. The trainees then need to act according to the appropriate flight procedures and react in a timely manner to any event generated by the simulator [31]. Since a simulator is the most realistic ground-based training experience that these trainees can obtain, aircraft companies all invest significant amounts of money into simulator training hours.

To ensure that a simulator is realistic, it must undergo onerous qualification procedures before being deployed. These qualification procedures [32] [33] need to demonstrate that the simulator behaviour represents an aircraft with high fidelity, including an accurate representation of performance and sound. There are four levels of training device qualification, from A to D, as described by the Federal Aviation Administration (FAA) with one hour of flight training in a level D training device being recognized as one hour of flight training in a real aircraft. Since re-qualification is mostly manual (requiring to book actual pilots), the total process can last from a week to several months.

The high cost of training device qualification derives partly from the fact that a simulator training device is composed of sophisticated off-the-shelf components from 3rd party providers (aircraft manufacturers for instance) that need to be treated as black-boxes during the integration process. Each provider of a COTS can use its own format to describe the interface of the COTS, yet the producer of the simulator needs to integrate all components in a coherent way without access to the components' source code. Furthermore, during the development and lifecycle of a simulator, a component's provider may update her component at any time, for example by updating hardware or software. Since the avionics domain lacks robust tool support [34] for automating tasks such as analysis of COTS interactions, test selection and test prioritization, even a small modification to a COTS still requires (manual) re-qualification of the whole system [35] [36] [37] [38].

What further hampers the automation of these tests is the large amount of data gathered during the qualification tests. Despite the safety-critical nature of these tests, the test data needs to be analyzed manually which is not only tedious but also risky, since any missed anomaly potentially can be life-threatening as incorrect behaviour of the simulator could incorrectly train pilots and crew members and cause tragic accidents [39]. Since the analysis of large data to identify deviating trends compared to previous versions of a software system are common to other domains as well, such as the analysis of software performance test results, this paper adapts a state-of-the-art technique for software performance analysis [10] to automatically

detect whether a new simulator version exhibits behaviour that is out of the ordinary. Instead of building a signature of normal system performance based on performance-related metrics, then comparing the signature to the metrics of a new test run to find performance deviations, our signatures are rules that describe the normal functional behaviour of a simulated aircraft observed in successful versions of a flight simulator, with deviations of these rules indicating deviating behaviour of an aircraft. Instead of collecting software performance metrics such as CPU utilization, we need to collect specific flight metrics that are related to the aircraft's functional behaviour.

For example, if the correct behaviour of a particular flight scenario on a successful version of a simulator exhibits high speed when the engines run at maximal capacity, there would be a signature rule **"max. engine capacity => high speed"**. A deviation would then be a case where, for a new version of the simulator, the aircraft suddenly would have a low speed while the engines still run at the maximal capacity. Of course, our approach needs to be robust against noise, where only during a short period a rule would be violated.

After calibration of the approach, we perform an empirical study on the FlightGear open source simulator, in which we introduce 2 faults related to the flight environment and 3 faults related to flight behaviour to address the following research questions :

RQ1 : What is the performance of the model for faults that change the flight environment ? *It is possible to have no false alarms for the Weak Wind fault, whereas all deviations for the Strong Wind fault are flagged, obtaining a precision of 50%.*

RQ2 : What is the performance of the model for functional faults ? *We find all injected deviations, and only in one case have a precision less than 100% (i.e., 67%).*

4.2 Background and Related work

Flight simulators are tested like as real aircraft. The pilot performing those simulator tests takes the pilot seat and goes through a list of procedures, specified in the Acceptance Test Manual, by performing different actions with the cockpit control inputs. After each procedure, the tester manually checks the cockpit indicators and verifies if their values correspond to the expected outcomes. If further investigations are needed, testers are able to monitor metrics within the flight simulator to collect information not available from the cockpit.

Although our paper does not aim to replace such time-consuming manual tests, we aim to complement them with automatic detection of behavioural deviations in between real pilot sessions. As such, for each component change the simulator can be steered automatically (instead of manually) through the required flight use cases, followed by automatic identification

of deviations. Use cases with detected deviations would then require more thorough manual analysis by a human pilot, while use cases without deviations might need less thorough human intervention. Hence, we do not aim to replace manual testing, but rather improve prioritization of test procedures.

The detection of anomalies and behavioural deviations are problems that have been studied within different areas and domains. Hodge et al. [40] surveyed techniques for outlier detection developed in statistical domains and machine learning, comparing their advantages and disadvantages in a comparative review. The authors considered five categories, namely outlier detection, novelty detection, anomaly detection, noise detection, and deviation detection. Their study is based on the definition of outlier by Barnett and Lewis as “An observation that appears to be inconsistent with the remainder of that set of data" [41]. Our work is closest to outlier detection, which Hodge et al. [40] define as “detection of abnormal running conditions".

Similarly, Chandola et al. [9] published a survey of research on anomaly detection. The authors grouped the reviewed techniques into 6 categories based on the approach adopted (i.e., Classification Based, Clustering Based, Nearest Neighbor Based, Statistical, Information Theoretic, Spectral). The authors provide also a discussion on the computational complexity of the selected techniques.

Aggarwal [42] presented an abnormality detection algorithm for supervised abnormality detection from multi-dimensional data streams. The algorithm performs statistical analysis on multiple dimensions of the data stream and is able to detect abnormalities with any amount of historical data. However, the accuracy improves with progression of the stream (i.e., more data received). Our work focuses on offline analysis, however future work could consider stream algorithms.

The problem of deviation detection in flight simulators is similar to that of regression detection on performance test results [10] [11] [12] [13]. Performance regression detection typically requires a system to run for a long time, after which recorded logs or metrics need to be analyzed to find deviations from the recorded logs and metrics of a previous release (baseline). Such deviations correspond to performance regressions in a system. Whereas such analysis for a long time had to be done manually by human experts, several studies have used data mining approaches on the recorded data to identify important deviations and rank them according to importance, after which the human expert could then look at the most important deviations.

In particular, Foo et al. [10] used association rules to automatically detect potential regressions in a performance regression test. Their approach compares the results of a new test run

to a set of rules built on successful test runs (extracted from a performance regression testing repository). If the new test run violates too many rules, it is tagged as a performance regression. The result of the approach is a regression report ranking potential problematic metrics that violate the extracted performance signatures. In other work [43], the authors extended the previous approach to take into account tests performed in a heterogeneous environment, i.e., tests executed with different hardware and/or different software.

Nguyen et al. [13] proposed to match the behaviour of previous tests, where a regression has occurred, to the behaviour of a new version. After data filtering, regression detection and regression cause identification are performed using machine learning techniques. The authors evaluated the approach on a commercial system and on an open-source system. The results showed that the machine learning approach can identify the cause of performance regression with up to 80% accuracy using a small number of historical test runs.

Cohen et al. applied probabilistic models [44] to identify metrics and threshold values that correlate with high-level performance states. The approach automatically analyzes data using Tree-Augmented Bayesian Networks to forecast failure conditions. In other work [45], the authors presented a method based on supervised machine learning techniques for extracting signatures able to identify when a system state is similar to a previously-observed state. The technique allows operators to quantify the frequency of recurrent issues.

Several works focused on mining execution logs (instead of metrics) to flag anomalies from normal behaviour. Jiang et al. [11], presented a statistical approach that analyzes the execution logs of a load test for detecting performance problems. Syer et al. [14] proposed an automated statistical approach combining execution logs and performance counters to diagnose memory issues in load tests.

Beschastnikh et al. [15] developed a tool called CSight to mine logs from a system's execution in order to infer a behavioural model in the form of communicating finite state machines, from which engineers can then detect anomalies and better understand their implementation. This work is a further evolution of Daikon [16], a tool to detect relevant program invariants from system traces. Program invariants are behavioural characteristics of a system that have to persist across versions of a system. It can be used in testing to look whether a system shows a regression. Our work differs from both lines of work in terms of the input data : instead of detailed system traces, we work on metrics that are sampled from observable component inputs and outputs. Furthermore, we build decision tree models instead of state machines.

Other studies have used data mining approaches for issue prediction in flight simulators. Hoseini et al. [18] proposed FELODE (Feature Location for Debugging) to detect configuration errors, using a semi-automated approach that combines a single execution trace and user

feedback to locate the connections among modules that are the most relevant to an observed failure. The authors have applied the approach to the CAE flight simulator system (CAE is one of the largest commercial flight simulator producers), achieving on average a precision of 50% and a recall of up to 100%. This paper is the closest to our work, however we (1) focus on behavioural deviations instead of configuration errors, (2) we do not consider user feedback, but aim for a fully automated approach, and (3) we use specialized metrics picked using domain knowledge to infer models.

4.3 Approach

This section presents our approach to automatically detect behavioural deviations in a new version of a flight simulator. As mentioned earlier, the approach is adapted from state-of-the-art machine learning approaches for detecting regressions in performance tests [10]. Fig. 4.1 shows an overview of our approach. The rest of this section explains the different steps of our approach, using a running example comprising a baseline (correct) version of a simulator and a deviating version.

4.3.1 Simulator Versions and Use Case

A baseline version of a simulator is used as a baseline to test other versions of the system against. Usually, the latter versions are very similar to the baseline, sharing a large proportion of the source code and configuration files. For example, a baseline version could be the previous release or a release before a large restructuring. In our running example, the Baseline Version is a correctly functioning version, while the Deviating Version corresponds to a change made to the Baseline Version. Having chosen two versions, we also need to identify representative use cases based on which we will compare the behaviour of both simulator versions. These use cases can correspond to the typical use cases of a simulator like take-off, cruising and landing, or to more specific use cases prescribed by an Acceptance Testing Manual. The running example in this section uses a take-off use case.

4.3.2 Flight Data

During their operation, flight simulators generate vast amounts of metric and log data at a configurable frequency, containing a mixture of numeric, discrete, and boolean data. This data logs a pilot's actions as well as the simulator's reactions, and is used for certification and for legal purposes. To access this data, one typically can tap into the system's public interface using an external logger, or use the system's built-in tools if available. Metric data

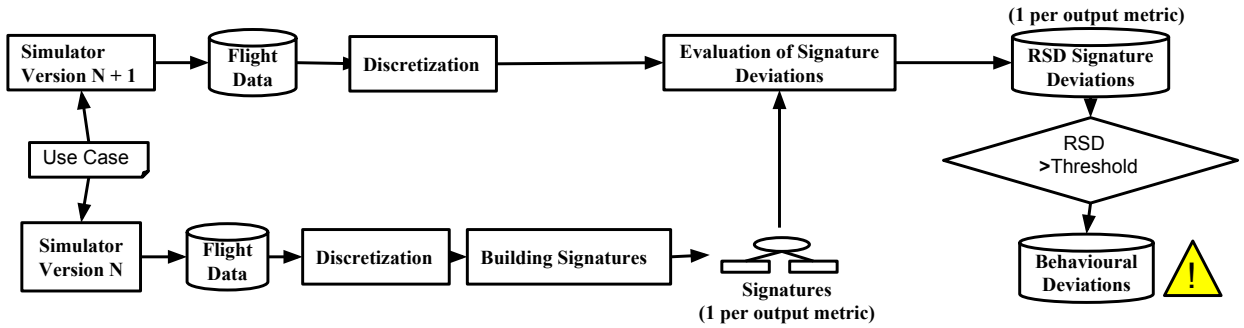


Figure 4.1 Overview of our approach.

(where we focus on) is typically stored in CSV files with one column per metric and one row per snapshot of metric values (i.e., the logger makes a snapshot of all metric values at a given frequency). Studying textual log data is outside the scope of this paper.

The metrics of the system can be divided in two groups : input metrics measure characteristics of the cockpit instruments and controllers used by the pilot, while output metrics observe the resulting behaviour of the simulator. Assignment of metrics into inputs and outputs cannot be done automatically because it requires basic knowledge about the system behaviour. For our analysis, we ignore internal metrics, i.e., all the metrics not readable from the cockpit. In our running example (Table 4.1), we know that the Throttle is an input metric because Throttle drives the engine power, while the Airspeed metric is an output that assesses the behaviour of the simulator. Note that we record the same set of input metrics and output metrics for both analyzed versions of the system.

4.3.3 Discretization

Once we have flight data collected from possibly multiple use case runs of both versions of the system, we need to prepare it for the machine learning techniques used in later steps. In particular, we should discretize the continuous metric data into a limited number of discrete values such as “low” and “high”. The idea is that a speed of 50 miles per hour is not that different from 51 miles per hour, while it is blazingly faster than 3 miles per hour. For this reason, the discretization step would convert the first two values to the same discrete value (“high”), while the third value would be mapped to “low”.

For this, we used the equal frequency bin discretization algorithm, which is a popular algorithm [17] that clusters data in K bins, each of which containing 1/K of the data. Basically, this algorithm first sorts the data, then splits it as much as possible into K equally sized

Table 4.1 Running example's flight data for the Baseline and Deviating version. Data that violates the signatures is highlighted in blue and red, respectively.

Input Metric		**Baseline Version**		**Deviating Version**	
Throttle	Discr. Throttle	Airspeed	Discr. Airspeed	Airspeed	Discr. Airspeed
0	low	0	low	0	low
0	low	0	low	0	low
0.8	high	10	low	10	low
0.8	high	60	low	40	low
0.8	high	130	low	90	low
0.8	high	180	high	130	low
0.8	high	180	high	150	low
0.8	high	180	high	180	high
0.8	high	180	high	180	high
0.8	high	180	high	180	high
0.8	high	180	high	180	high
0.8	high	180	high	180	high
0.8	high	180	high	180	high
0.8	high	180	high	180	high
0.8	high	180	high	180	high

bins. In our running example, the airspeed output of the Baseline Version takes the values of "[0,0,10,60,130,180,180,180,180,180,180,180,180, 180, 180]", which the equal frequency algorithm set with K=2 bins would split into bins [-inf, 180[and [180, inf] that we label as "low" and "high". Table 4.1 shows the resulting discretized values. Note that the algorithm puts all instances with the same value into the same discretized bin, even if that results in a bin having more than 1/K of the data, which is what happens for "180" in Table 4.1. Popular values of K are K=3 (low/medium/high) and K=5 (very low/low/medium/high/very high), since those are easy to interpret by humans [17].

4.3.4 Building Signatures

The core of our approach is the learning of models ("signatures") that represent the essence of the behaviour of a basic simulator version for a particular use case and output metric. In principle, we can use any kind of machine learning technique for this, by using the input metrics as independent variables and one of the output metrics as dependent variable. We have a preference for decision tree models because they provide a graphical abstraction that is easy to interpret, even for people without machine learning background [17]. In addition, decision trees can easily be transformed into a "rule" form that is straightforward to check automatically on new data.

Fig. 4.2 illustrates this on a decision tree signature for our running example, with throttle as input and airspeed as output. Each non-leaf node (ellipse) of the tree corresponds to a test in terms of one of the input metrics (here there is only one input metric), while the leaf nodes (rectangles) correspond to the value of the output metric suggested by the model based on the tests. Here, the model states that a High Throttle corresponds to High Airspeed, while Low Throttle correspond to Low Airspeed. We can summarize this tree into the two rules "High Throttle => High Airspeed" and "Low Throttle => Low Airspeed". Trees can have an arbitrary number of test nodes, in which case the rules become more complex (e.g., "High A and Low B and High C => High D").

Standard algorithms for building decision trees are C4.5 and C5.0. They typically try to reduce the depth of a tree, since more complex models tend to overfit a given data set and hence are less effective [17]. Note that the produced models for a given data set never will be 100% accurate. Indeed, if the algorithm sees that 95% of the time x=4 yields y=5, it would probably generate the rule “x=4 => y=5”, even though 5% of the data would be classified incorrectly. Such rule violations can also occur when too few bins are used to discretize the data (two in our running example), as one loses granularity in the data. Table 4.1 shows that while transitioning from 0 Throttle to 0.8 (cells highlighted in blue), our vertical speed is still considered as low in the Baseline Version (despite high throttle) due to the limitations of the discretization used.

4.3.5 Evaluation of Signature Deviations

In order to determine whether a new simulator version contains behavioural deviations for a given use case, we need to measure the degree to which the new version’s metrics deviate from the Baseline Version’s signatures (for each output metric). For this, we use the proportion of correct classifications, for a given signature, on the data of a given use case, which yields the Version Signature Deviation (VSD) metric in Formula 4.1.

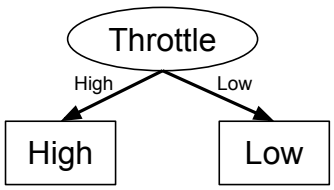


Figure 4.2 Signature decision tree for our running example.

$$\text{VSD} = (1 - \frac{\text{\# Correctly classified snapshots}}{\text{\# Classified snapshots}}) \tag{4.1}$$

In this formula, a "snapshot" corresponds to the metric values recorded at a specific timestamp (i.e., one row in Table 4.1). The more snapshots are classified correctly, the lower the VSD value. However, we cannot directly use the VSD to determine behavioural deviations, since, as explained in the previous section, signatures are never 100% accurate (to avoid overfitting). This means that even the baseline version of the simulator has a VSD higher than zero. For this reason, our approach uses the following relative signature deviation (RSD) metric :

$$\text{RSD} = \text{VSD}_{\text{New Version}} - \text{VSD}_{\text{Baseline Version}} \tag{4.2}$$

For our running example, Table 4.1 highlights in blue or pink the airspeed snapshots that do not match the signature in the Basic and Deviating versions of the simulator. In both versions, the coloured snapshots observe the rule "throttle=high => airspeed=low" rather than "throttle=high => airspeed=high". For the Baseline Version, there are 12 rows that matched the signature and 3 rows that did not match it, hence we obtained a VSD of 1 - 12/15=20%, which corresponds to the $\text{VSD}_{\text{Baseline Version}}$ variable for Table 4.1. We applied the same technique for the Deviating Version, which has 10 rows that matched the signature out of a total of 15, yielding a VSD of 1-10/15=33%. By using Formula 4.2, we obtain an RSD of 33% -20% = 13% for the airspeed output metric.

4.3.6 Detecting Behavioural Deviations

Knowing the RSD is not enough, since one still needs to decide which values of RSD are high enough to indicate a behavioural deviation (instead of noise in the measurements). Hence, our approach uses thresholds to flag deviations related to functional problems. If the RSD for a given output metric is larger than the threshold, we consider the deviation too large and hence indicative of a behavioural deviation for that metric. Since each output metric has its own signature, we need to compute and compare multiple RSDs, one per output metric. This means that a particular simulator version could show behavioural deviations for multiple output metrics.

In our running example, if we use a deviation detection threshold of 10%, the airspeed output

in the Deviating Version would be flagged as a deviation because the RSD of 13% is above that threshold.

4.4 Case Study Setup

To address the research questions mentioned in the introduction, we evaluated our approach on the FlightGear flight simulator, in which we inject faults to obtain different versions to compare to a baseline version using the proposed approach. This section explains how we implemented each step of the approach and it also explains how we determined the RSD threshold to determine whether or not a given signature deviation is normal.

4.4.1 Simulator Versions and Use Case

The FlightGear flight simulator [24] is a sophisticated open source framework for flight simulation. The project was started in April 1996 and its first multi-platform release was developed in July 1997. Since then, it has expanded beyond flight aerodynamics and by now it offers a broad range of configurable features to recreate flight environments similar to reality. It is possible to take off from several airports located all around the world, change the weather condition by generating storms, and much more. This allows us to use realistic flight scenarios.

SimGear is FlightGear's simulation engine and it is used both for research purposes and as a stand-alone flight-sim [26, 27, 28]. FlightGear exposes the internal state of the SimGear simulation through a property database that dynamically maps a value (such as speed or latitude) into an object with getter and setter methods. In our study, we used FlightGear version 3.2 on a recent computer with high performance video cards (CPU : Intel i5-2500K, RAM 8GB, OS : Windows 7) to perform smooth virtual flights at a maximal frame rate.

For our evaluation, we focused on one use case in which we manually piloted and recorded flight data with version 3.2 of Flight Gear. This use case consists of a normal landing maneuver using the default FlightGear configuration, which uses a Boeing 700-300 aircraft and no extra environmental conditions. We opted for this use case, as the landing (just like take-off) is one of the most critical moments of a flight [cite], while the Boeing 700-300 is one of the most popular flight models (and is also the default flight model used in FlightGear). To pilot the aircraft, we used a FlightGear tutorial [1] and we also received advice from experts of a major flight simulator company.

1. http ://www.flightgear.org/Docs/getstart/getstartch6.html

4.4.2 Flight Data

Similar to existing deviation detection approaches, an organization wanting to adopt our approach needs to have a set of simulator versions that are known to be "correct" (i.e., passed qualification testing) to calibrate and build models on. For example, Herzig et al. [46] used a historic set of recorded test executions mapped to different versions of the analyzed software version to collect data to simulate the behaviour of their test selection strategy.

However, with FlightGear, since we did not have knowledge about which versions of FlightGear had known behavioural deviations, hence we had to artificially create such versions by injecting faults in the simulator [47]. To determine which types of faults to inject, it is important to understand how FlightGear works. At run-time, FlightGear loads an aircraft model of the flight behaviour, a configuration for the flight environment (e.g., wind turbulence), and an input file containing the pilot's cockpit operations in case of an automatic flight. Hence, we decided to play with these elements to generate new simulator versions with known faults.

Since we are performing the same use case on all versions of the simulator, the input file needs to be identical for all versions, modulo random noise to simulate harmless timing differences between the pilot's operations. These differences make the flight data more realistic, as human test runs for the same qualification test will never be identical. However, a new version can feature changes to the flight environment (e.g., different default settings for wind) or to the aircraft model flight behaviour (e.g., a coding error in the flight model). For this reason, we inject faults that either manipulate the flight's environment or behaviour. These faults represent safety-critical situations that can occur in flight simulators, and are known to have caused important accidents in real flights [39]. Table 4.2 summarizes these faults, and below we provide more details about them.

Since this paper performs a first experimental evaluation of our approach, we focus on five common types of faults that we identified through consultation with experts from a major commercial flight simulator vendor. Note that although some of these faults seem easy to spot manually on small data sets if one knows what behavioural deviation to look for, the challenge addressed by our approach is to spot those deviations on large data sets, without prior knowledge about what deviations are present.

Weak/Strong Wind

Weather conditions, such as a strong wind, add stress to the aircraft during the flight and can sometimes cause accidents in critical flight phases like the landing phase. Therefore, we injected two wind faults with different magnitudes, one introducing a weak tail wind of 5

Tableau 4.2 Injected Faults

Fault Name	Type	Description	Behaviour
Weak Wind	Weather	Wind at 5 knots 25 deg.	Normal
Strong Wind	Weather	Wind at 25 knots 25 deg.	Deviations
Flaps Fault	Functional	Flaps disabled	Deviations
Auto Brake Fault	Functional	Auto brake disabled	Deviations
Right Engine Fault	Functional	Failure of right engine	Deviations

knots (10 km/h) at 25 degrees clockwise, and one introducing strong tail wind of 25 knots (50 km/h) at the same angle. A strong tailwind is well recognized to be dangerous while landing or taking off, since it reduces the lift and increases speed [2]. On the other hand, the weak tail wind should be considered as normal, because it does not cause major deviations. Hence, the weak wind fault is a false positive that we added to our set of faults to evaluate false alarms generated by our approach.

Flaps Fault

Flaps are an essential aircraft wing component that, when extended, increase the camber of the wing, and raise the maximum lift coefficient. This coefficient has to be increased to perform a take-off operation, and decreased for landing. Many accidents are caused by flaps not deployed during take-off or landing phases [39]. The Flaps Fault has all flaps disabled, i.e., when a pilot activates the flaps, nothing happens.

Auto Brake Fault

Auto brakes are automatic wheel-based hydraulic brake systems used during the take-off and landing phases. There are known cases were these brakes have caused accidents (especially during the landing phase) [3]. We injected an Auto Brake fault in which the auto brake does not work during the landing phase.

2. https ://www.tc.gc.ca/eng/civilaviation/publications/tp185-1-05-614-2901.htm
3. http ://www.fss.aero/accident-reports/look.php ?reports_key=488

Right Engine Fault

Engines are the propulsion systems of the aircraft and there are multiple known cases where a problem with the engines caused accidents [48]. We injected a fault where the right engine does not work during the landing phase.

4.4.3 Discretization

Out of all metrics available in the simulator, we retained all metrics linked to manipulations of cockpit controllers as input metrics, while metrics linked to cockpit-observable data are used as output metrics. The input and output metrics are presented in Table 4.3 and Table 4.4.

We selected them based on the importance that they have from a pilot's perspective for assessing the behaviour of the aircraft and controlling it. Input metrics indicate the data of all controls available from the cockpit such as throttle, elevators, ailerons, landing gears, auto brake, flaps, autopilot controls, aircraft control switches, etc. Output metrics indicate the data produced by cockpit instruments like the airspeed indicator, altitude indicator, altimeter, turn coordinator, directional gyro, and vertical speed indicator. We first selected output metrics that are displayed by these instruments, such as Airspeed, Roll, Pitch, Altitude, Heading, and Vertical Speed, then selected additional output metrics, not necessarily visible from the cockpit, that often are used by analysts to interpret a flight's behaviour.

We generated the metric data for each faulty version by running the version according to the landing use case, then using FlightGear's built-in data collection tool configured with a sampling rate of one sample per second, which gives enough data without taking too much disk space. We discretized data into a maximal number of K=3 bins (labeled "low", "medium", and "high"), which is a number of bins that can be easily interpreted by a human.

4.4.4 Generation of Signatures and Deviation Detection

We used the approach of section III.D to build signatures for each output metric (see Table 4.3) of the baseline (correct) FlightGear version, then used III.E and III.F to determine possible behavioural deviations for each faulty version and output metric. We use the Weka data mining tool [49] for the signatures and R [50] scripts for the deviations.

4.4.5 Evaluation of Approach

To evaluate how well the approach works for the two research questions, we need to know each fault's oracle, i.e., the metrics that really are deviating in those scenarios and hence should

Table 4.3 List of output metrics

Flight Metric	**Category**
latitude-deg	Position
longitude-deg	Position
altitude-ft	Position
roll-deg	Orientation
pitch-deg	Orientation
heading-deg	Orientation
side-slip-rad	Orientation
airspeed-kt	Velocity
speed-down-fps	Velocity
speed-east-fps	Velocity
speed-north-fps	Velocity
uBody-fps	Velocity
vBody-fps	Velocity
wBody-fps	Velocity
vertical-speed-fps	Velocity
ned-down-accel-fps-sec	Acceleration
ned-east-accel-fps-sec	Acceleration
ned-north-accel-fps-sec	Acceleration
pilot-x-accel-fps-sec	Acceleration
pilot-y-accel-fps-sec	Acceleration
pilot-z-accel-fps-sec	Acceleration
elevator-pos-norm	Surface Position
flap-pos-norm	Surface Position
left-aileron-pos-norm	Surface Position
right-aileron-pos-norm	Surface Position
rudder-pos-norm	Surface Position

Table 4.4 List of Input Metrics

slats	speedbrake	throttle
throttle1	starter	starter1
fuel-pump	fuel-pump1	cutoff
cutoff1	mixture	mixture1
propeller-pitch	propeller-pitch1	magnetos
magnetos1	ignition	ignition1
brake-left	brake-right	brake-parking
steering	gear-down	gear-position
gear-position1	gear-position2	gear-position3
gear-position4	engine-pump	engine-pump1
electric-pump	electric-pump1	battery-switch
external-power	APU-generator	engage
heading-select	altitude-select	bank-angle-select
rudder	rudder-trim	flaps
aileron-trim	elevator	elevator-trim
vertical-speed-select	speed-select	

be flagged by our approach. To obtain our oracle, we interviewed 6 aerospace practitioners from 2 major aerospace companies. Since we got only few experts relative to the number of flight metrics to analyze, we lifted up the output metrics mentioned by experts to the level of metric categories, as shown in the second column of Table 4.3, which guaranteed a better convergence of expert opinions. Hence, for each faulty version we asked them which of the output metric categories would show deviations in their recorded value (see Table 4.5). We consider our approach to detect a metric category deviation as soon as it detects one metric of that category.

Based on the oracle, we then calculate precision, recall and false alarm rate of our approach. **Precision** corresponds to the percentage of correctly flagged deviations (i.e., absence of false alarms), as shown in Formula 4.3. **Recall** is the percentage of all deviations in the oracle that

Table 4.5 Oracles for each fault

Fault	**Deviating Flight Metric Categories**
Weak Wind	Nothing
Strong Wind	Velocity and Position
Flaps Fault	Velocity and Orientation
Auto Brake Fault	Acceleration, Position, Orientation and Velocity
Right Engine Fault	Acceleration, Velocity and Orientation

the approach is able to identify, cf. Formula 4.4. Finally, for faulty versions without deviations (like the Weak Wind fault), we use the False Alarm Rate in Formula 4.5 to measure how well the approach avoids false alarms.

$$\text{Precision} = \frac{\text{\# Correctly identified deviation categories}}{\text{\#Identified deviation categories}} \tag{4.3}$$

$$\text{Recall} = \frac{\text{\# Correctly identified deviation categories}}{\text{\#Deviation categories}} \tag{4.4}$$

$$\text{False Alarm Rate} = \frac{\text{\#Incorrectly identified deviation categories}}{\text{\#Non-deviation categories}} \tag{4.5}$$

In the running example, the Airspeed metric belongs to the Speed category. If this metric's category would be part of the oracle for the running example, our approach would have 100% recall/precision because it flags this output metric.

4.4.6 Calibration

Ideally, we aim to have both high precision and recall, while for false alarm rate lower values are better. In practice, each performance metric goes at the expense of the other, hence a trade-off is necessary. Given the safety-critical nature of flight simulators, we favour recall over precision and false alarm rate. The most direct way to control this trade-off is via the RSD threshold. The lower this threshold, the more output metrics will be flagged as deviating, which will increase recall and likely reduce precision.

To determine the threshold value to use, we use two approaches, resulting in an "initial threshold" and "optimal threshold". The initial threshold is suitable when one starts to adopt the proposed approach (since all one needs are one or more correct simulator versions), or when the new simulator version is too different from the previous one. The optimal threshold on the other hand exploits results of previous runs of the approach to find the best possible threshold based on historical results. section 4.6 discusses in more detail how we derived the optimal threshold, while this subsection explains how we determine the initial threshold.

The initial threshold is robust enough to tolerate slight variations in the output metrics of the baseline version of the simulator (i.e., the version with known correct behaviour) without generating any false alarm. Such variations could be due to the random noise caused by timing differences between the operations of a pilot or by non-deterministic scheduling of the

simulator process by the operating system.

To perform the calibration of the initial threshold, we executed a baseline run of FlightGear, then executed 4 more runs of the baseline version. As explained before, each run, even though following the same sequence of operations as the initial baseline run, has random differences in timing. RSD values across the four baseline runs. We then determine a threshold for RSD (Formula 4.2) that is able to tolerate these timing differences by not flagging false deviations.

Fig. 4.3 (top section) shows, for each output metric category, a boxplot of the RSD values across the four basic runs. Such a boxplot shows the minimum (lowest whisker), 25th percentile, median (line inside the box), 75th percentile and maximum (highest whisker).

To find the lowest threshold that avoids flagging false deviations, we use the false alarm rate of Formula 4.5, whose results are shown in Table 4.6.

There are 2 distinct clusters of flight metrics, which means that some output metrics contain much more noise than others. By looking at the trend of the RSD medians, two clusters can be distinguished. The first cluster (Cluster 1) contains the majority of flight metrics except for “left-aileron-pos-norm" and “right-aileron-pos-norm" (i.e., the two blue boxplots), both of which compose the second cluster (Cluster 2).

The signature deviation of the baseline metrics has a range of 41%. Fig. 4.3 shows that, across the output metric RSDs, the lowest box plot whisker has a value of -4% and the highest one has a value of 37%, providing a range of 41%.

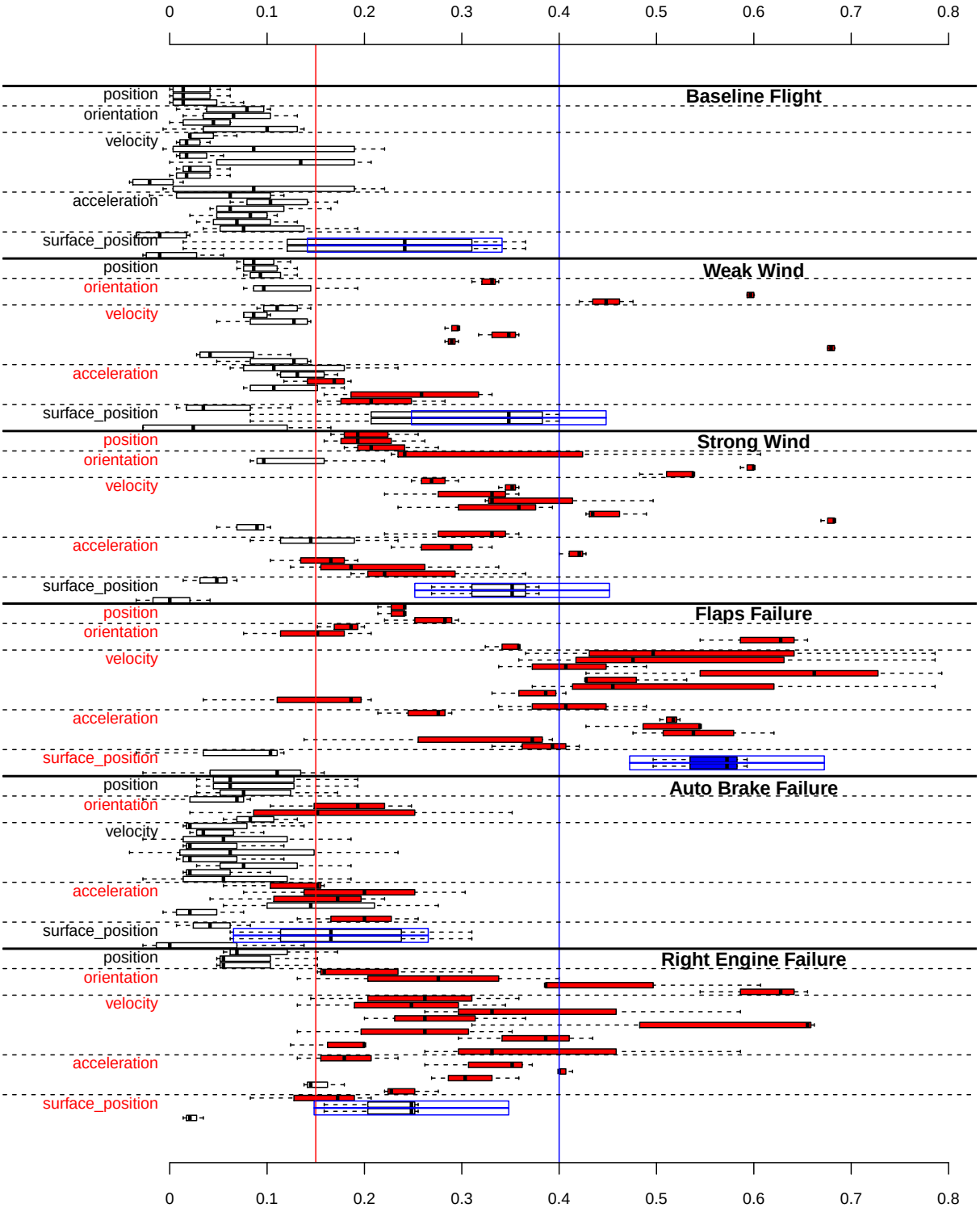


Figure 4.3 Flight metric RSDs by categories, for all scenarios. Red categories contain at least one flagged metric represented by a red (Cluster 1) or a blue boxplot (Cluster 2).

A negative value means that the output metric matches the signature better than the metric in the original baseline flight. Noise is responsible for the different variations of the baseline flight, with some variables being more susceptible to it than others. Hence, a good threshold choice is necessary to deal with these variations and avoid incorrectly flagged deviations.

If we pick initial thresholds of 15% and 40% respectively for clusters 1 and 2, we obtain a false alarm rate of 0%. Assuming that the only data available for determining RSD thresholds is the data obtained by running the baseline version of FlightGear (which is the case for any organization adopting the approach for the first time), we can pick an initial threshold of 15% (the red line in Fig. 4.3) for Cluster 1 and a threshold of 40% (the blue line) for Cluster 2 to obtain a false alarm rate of 0%. These thresholds correspond to the clusters' highest median value plus a safe margin, in order not to flag deviations incorrectly.

As mentioned, for our research questions, we will also use an optimal threshold, which can only be determined once more flight data is available, but obtains the best possible performance, which on the calibration data is identical to that of the initial thresholds (i.e., 0% false alarm rate).

4.5 Case Study Results

RQ1 : What is the performance of the model for faults that change the flight environment ?

Motivation

Different weather circumstances can render a flight very different from the original flight. Whereas the introduction of a weak wind only slightly modifies the baseline flight, the introduction of a strong wind condition heavily alters the Baseline Flight's behaviour.

Approach

We collected data for 3 runs of the Weak Wind FlightGear version and 3 runs of the Strong Wind FlightGear version (see related RSD values on Fig. 4.3). Using the initial and optimal thresholds, we obtain the precision, recall, and false alarm rate values of Tables 4.6 (last row) and 4.7 (first row).

Results

Some output metrics have RSDs that are much larger than for other metrics. Especially for Strong Wind we can see that the median RSD values have doubled or even tripled compared to the calibration runs of section 4.6. Furthermore, across all flight metrics and both flight environment faults, there is a range of 51% between the lowest RSD median

and the highest one. There are 2 particularly large deviations : metrics “heading-deg" and “vertical-speed-fps". These are common to both faults, hence these metrics seem to have a strong relation to wind. This makes sense, since the wind for both faults increases the speed of the aircraft and the wind angle changes the aircraft's direction.

We achieve a perfect recall for the Strong Wind configuration, but only 50% precision, while the false alarm rate for the Weak Wind is 60%. For the Weak Wind fault, we wrongly flag output metrics related to orientation, velocity, and acceleration categories because our Cluster 1 calibration threshold is too low. We have better results with the higher optimal threshold (false alarm rate of 0%), whereas it is not possible to get better results for the Strong Wind fault. This is because whenever we increase the Cluster 1 threshold to narrow our selection of metric categories, we first lose the metrics predicted by our oracle before incorrectly flagged metrics as the former show less deviation than the latter. These results show that, while we do not miss any deviating behaviour (perfect recall), human analysts would still need to consider 50% false alarms (low precision).

It is possible to have no false alarms for the Weak Wind version, whereas all deviations for the Strong Wind version are flagged, resulting in a precision of only 50%.

RQ2 : What is the performance of the model for functional faults ?

Motivation

System faults can render a new version's behaviour very different from the original version and are the cause of many accidents, often impacting people's lives. For this reason, these failures are the most important injected faults on which we evaluate our approach.

Approach

We collected data for 3 runs of each of the 3 faulty FlightGear versions (see related RSD values on Fig. 4.3). We evaluated our approach using the same methodology as RQ1. Table 4.7 shows the performance results for each of the configurations, for both thresholds.

Results

Most of the system failures have deviations across all metric categories. Both the Flaps Failure and Right Engine Failure injected faults have large deviations (i.e., high RSDs values) by flagging almost all metric categories, except for the Position category for the Right Engine Failure. Auto Brake Failure is the fault with the least deviation compared to Baseline, having low RSD values not higher than 25%.

We do not miss any deviation (100% recall), yet for one fault (Flaps Fault) we obtain a precision less than 100%. Furthermore, we note that the optimal threshold performs better in terms of precision and/or recall than the initial threshold for all 3 faults. For flaps, our approach might not be precise (40%) when using our initial threshold, however it has successfully warned testers about system's misbehaviour by covering at least all predicted metric categories (100% recall). Increasing the threshold by 40% provides a better precision of 67%. For the Right Engine Failure, our approach with initial threshold was close to a perfect precision (75%) with a perfect recall (100%), where it has only triggered a false alarm on the Surface Position category. Increasing the threshold by a few percent (+5%) achieves a perfect detection.

We noticed a special case for Auto Brake Fault in which the recall using the initial thresholds is lower than 100%. Fig. 4.3 shows that this likely is due to accidental variation of the metrics related to velocity and position categories, as the reported metrics for this fault injection are close to the thresholds, hence the deviations were narrowly missed.

We are able to find all injected deviations, and only in one case have a precision less than 100% (i.e., 67%).

4.6 Discussion

4.6.1 Optimal thresholds

In our case study, we have guided our initial threshold choice based on the characteristics of the Baseline Flight, trying to minimize the false alarm rate. However, after running our approach on multiple versions, possibly including behavioural deviations, more data becomes available that can be used to tune the thresholds, in particular to try to optimize the recall for finding deviations. This section analyzes how sensitive our approach is to the choice of thresholds, and how much we give up on precision when optimizing for recall (and vice versa).

To do the analysis, we evaluated our approach's recall/precision for different thresholds in a range from 0 to 80% (since in practice higher values would lead to no deviation being found),

Table 4.6 False alarm rates for threshold calibration and RQ1

Fault	Initial False Alarm Rate	Optimal False Alarm Rate
Baseline Flight	0%	0%
Weak Wind	60%	0%

Table 4.7 Precision and recall for RQ1 and RQ2

Fault	Initial Recall	Initial Precision	Optimal Recall	Optimal Precision
Strong Wind	100%	50%	100%	50%
Flaps Fault	100%	40%	100%	67%
Auto Brake Fault	50%	100%	100%	100%
Right Engine Fault	100%	75%	100%	100%

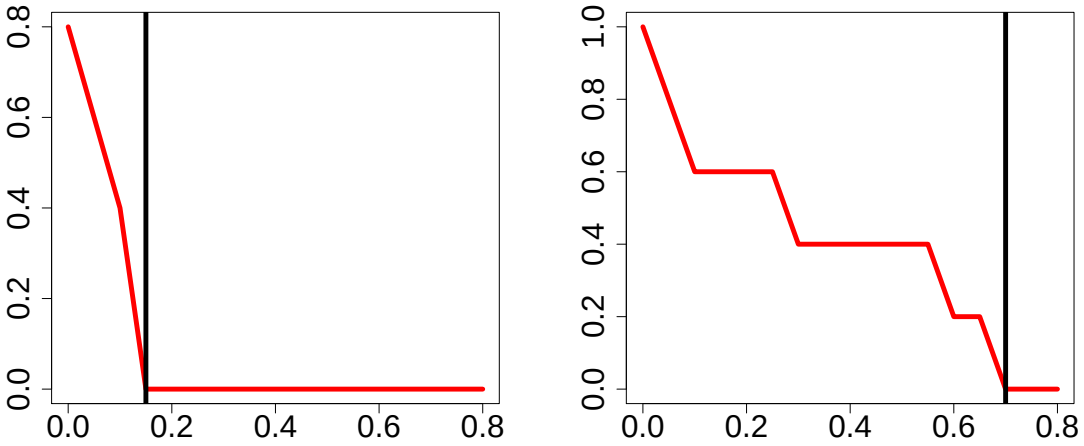
then plot the recall and precision for each threshold in order to find a sweet spot where both values are high (Fig. 4.7). For the versions without deviations (Baseline Flight and Weak Wind), we plot the false alarm rate instead, which we aim to keep low (Fig. 4).

For instance, for the Strong Wind version (Fig. 4.7), the optimal threshold is 20%, since we have the highest recall of 100% and the best precision of 50%. One could also pick the 15% threshold (which offers the same recall/precision), but a higher threshold might be more conservative, which is better from the false positive point of view. The black lines in Fig. 4 and 5 are the optimal thresholds used in Section V.

Compared to the initial thresholds, we see that a lower threshold would have been interesting to get a perfect precision/recall for the Auto Brake Fault version (Fig. 4.7). For the Strong Wind version, we see that a higher precision of 100% is possible, albeit with a lower recall of 50%. The Weak Wind and Flaps Fault versions (Fig. 4.5 and Fig. 4.7) need high thresholds to reach a lower false alarm rate, respectively higher precision, however for the Flaps Fault we cannot get a perfect precision (maximum of 67%). The Right Engine Fault version (Fig. 4.7) requires a slightly higher threshold (5%) than the initial one to have perfect precision/recall.

As is clear from these observations, if one wants to obtain an optimal precision (with perfect recall), one would need to pick a different threshold for each new simulator version. Since in practice one does not know whether a new version contains a deviation (and if so, which one), as this is the whole point of our approach, using version-specific thresholds is impractical. Hence in practice, the high precision values of RQ1 and RQ2 for the optimal thresholds cannot be achieved at the same time. Yet, considering that current practices require 100% human intervention, this is still an important improvement.

However, if one would pick the optimal (recall) threshold of the Auto Brake Fault version for all faults, one would still achieve a precision between 40% and 80%. Although human testers might need to spend some effort analyzing falsely flagged metrics, at least this is less than having to analyze all data manually (as is currently the case).



Baseline Flight Weak Wind

Figure 4.5 False alarm rate across multiple thresholds. The black lines represent the optimal thresholds.

Although the optimal thresholds are better than the initial thresholds, the initial thresholds derived from a baseline version still obtain a relatively good overall performance. Experiments with more metrics and versions are needed to further refine our findings.

4.6.2 Applicability of our approach

At the basis of our approach for detecting behaviour deviations, we need to collect data from multiple system runs of (at least one) correct version. This suffices to build a model with decent (initial) RSD threshold. The model becomes better if one also has a set of incorrect versions, i.e., versions that did not pass qualification tests, as this allows to have a more accurate model and threshold.

Any flight simulator vendor or user should be able to collect at least a set of correct simulator versions, either consisting of older official releases or (even better) via the simulator's version control system, which has each developer version. As long as these versions allow to collect flight metric data, our approach can be applied. Once data have been collected from multiple runs of a system under test to calibrate our approach, only a few minutes are needed to analyze this data. Depending on the system, collecting data is the most time-consuming task in the test analysis process.

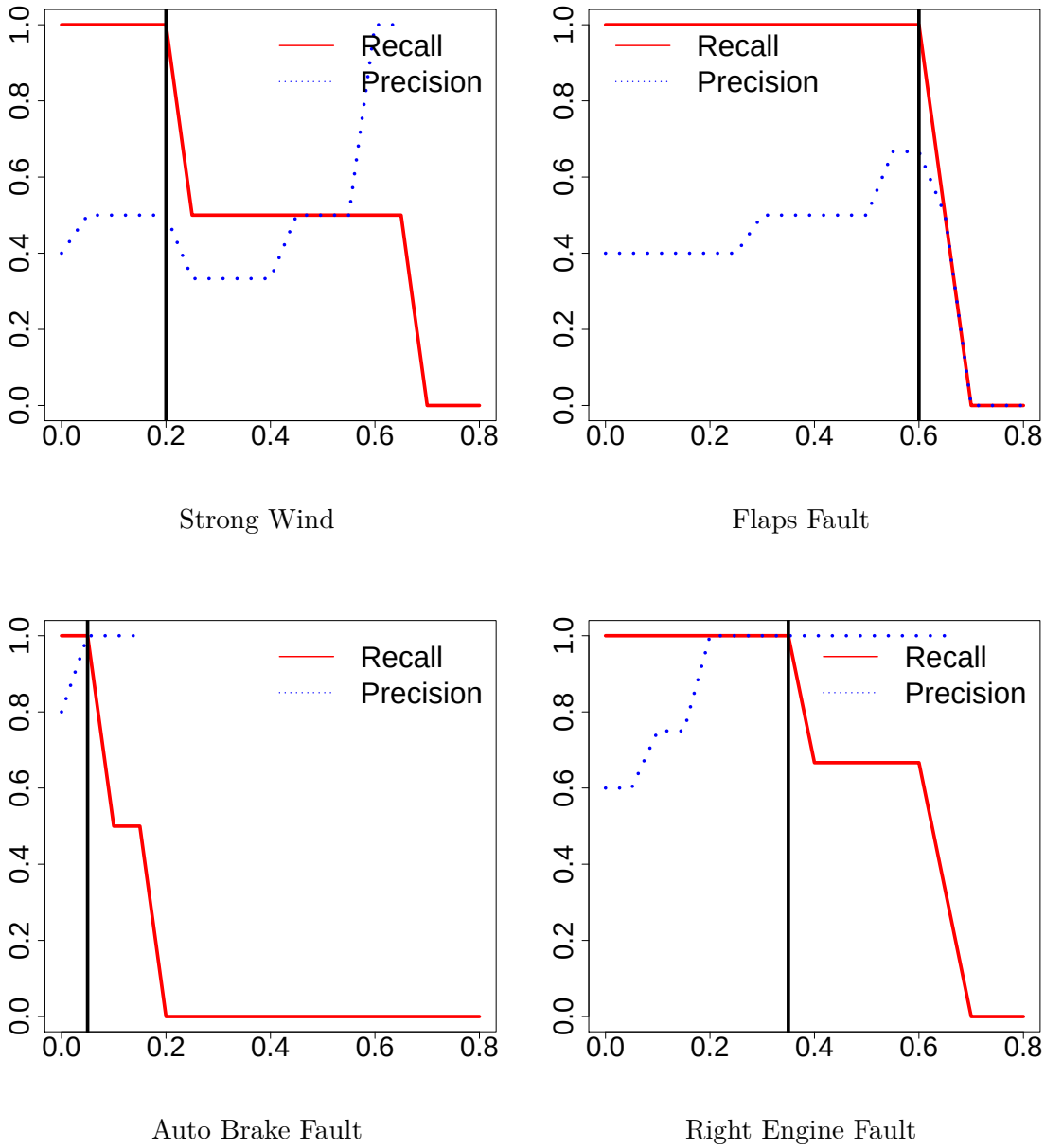


Figure 4.7 Recall and precision values across multiple thresholds for injected faults. The black lines represent the optimal thresholds of a version with an injected fault. The plot for Auto Brake Fault does not show its complete precision curve, since from a given threshold on, no true or false deviations are found.

4.6.3 Type of Deviations

Our approach naively detects deviations by using the proportion of samples in a given period that violate baseline rules. Since the approach essentially ignores the chronological nature of snapshots (instead of a sequence of snapshots, we use a bag of snapshots), it does not cover all possible behavioural deviations. In time series theory, there are two major types of behavioural deviations, i.e., those that impact the amplitude of a signal (i.e., the values of metrics across time) and those that impact the transients. The latter correspond to the periods in which a system transitions from one value of a metric to another. For example, if a plane is cruising at a speed of 200 knots, and the pilot directs the plane to accelerate to 260 knots, the plane will not immediately have the speed of 260 knots, but will transition according to same trend (e.g., linearly) from the initial speed to the desired one.

Our current machine learning approach only detects deviations in amplitude, whereas transient deviations are missed because transients typically are too short and hence not picked up by the decision tree algorithms. For example, if we analyze the altitude of an aircraft during a full flight, the take off and landing phases would be insignificant compared to the rest because the aircraft is cruising most of the time (i.e., altitude would look constant). However, a deviation during these small transition phases between ground and cruise altitude can be dangerous, e.g., an aircraft landing in a few seconds rather than a few minutes represents a danger for passengers. Hence, in future work we will adapt our approach to also support transients, initial results to that end seem promising.

4.6.4 Fault Injection vs Bugs

Because we did not have access to versions of FlightGear with known behavioural deviations, we artificially injected faults by changing the flight environment or flight behaviour of an aircraft model for one FlightGear version. In reality, our approach would work for any case where a change (e.g., bug or new feature) introduced in the source by a developer could impact the behaviour of the system, as long as the same use case can be run on both versions and the same flight metrics can be extracted. We expect the most typical scenarios to be between a fixed version and the buggy version, but also between an initial version of a simulator and a version with an updated COTS component, or between an initial version and a heavily refactored one.

4.7 Threats to Validity

Construct validity threats concern the relation between theory and observation. To evaluate the precision and recall of our approach, our oracle did not point out individual metrics, but rather categories of flight metrics. The reason for this is that the number of metrics was much larger than the number of available experts, and that the experts contacted for our oracle are domain experts that are sufficiently familiar with categories of flight metrics (Speed, Acceleration, Velocity, an Position), but not with the flight simulator used and its specific flight metrics. For example, we consider the "airspeed-kt" and the "vertical-speed-fps" to be in the same Speed category. Evaluation of the approach at the individual flight metric-level, as well as subsequent root cause analysis to pinpoint the component responsible for a behavioural deviation, is left for future work.

Furthermore, some parameters of our evaluation, such as the value of K=3 for discretization and the number of runs considered could impact our findings. Since the generation of data for multiple runs of a use case is time-consuming, we limited our study to [how many ?] runs. We plan to automatically generate runs using an automated pilot system for FlightGear.

Threats to external validity are related to the selection of one specific version of the FlightGear open source flight simulator for our study. Although Flight Gear is a well known system widely used for research purposes, it may not be representative of the whole flight simulator's domain. Further studies on commercial flight simulators are needed to test the generality of our approach.

4.8 Conclusion

In this paper, we explored a signature-based data mining approach to flag behavioural deviations at the system level of safety-critical systems such as a flight simulator. Evaluation of the approach on FlightGear (an open source flight simulator) shows that we are able to automatically detect all injected deviations related to flight environment and flight behaviour with a perfect recall of 100% and a precision of at least 50%. However, in a setting where only limited data is available for tuning the approach, it can still obtain a recall/precision of 40% to 100%. This is promising, since it enables organizations who want to adopt this approach to obtain a decent performance out-of-the-box.

CHAPITRE 5 DÉTECTION DES DÉVIATIONS DANS LES TRANSITIONS DES MÉTRIQUES

Nos résultats à la section 4.5 montrent la capacité de notre approche pour détecter les déviations dans l'amplitude générale des métriques. Par contre, nous ne détectons pas les déviations des métriques dans leur régime transitoire, ce qui est un autre aspect à évaluer quand on test les simulateurs. En effet, comme stipulé dans le QEG, les tests d'acceptation incluent une évaluation du régime transitoire de certaines métriques du simulateur. Une déviation dans le régime transitoire peut soit être un changement dans la forme de la courbe de transition ou bien être une période de transition d'une longueur différente. Dans chacun des cas, une déviation dans la transition de métrique indique que le simulateur est atteint d'un problème fonctionnel.

Les deux courbes du haut de la figure 5.2 montrent un exemple de déviation dans le régime transitoire de la métrique "Angle du Roulis". La courbe verte enregistrée dans la nouvelle version du simulateur évolue beaucoup plus lentement par rapport à la courbe noire enregistrée dans la version de référence du simulateur. On y constate également que la courbe déviante possède une forme plus arrondie que la courbe de base qui possède une transition abrupte. Dans les deux cas, une métrique part d'un premier état pour atteindre un autre état. Ces états sont similaires à travers les deux versions, par contre la déviation est dans la transition.

Dans la plupart des cas, les périodes de transition dans une métrique sont minimes par rapport aux périodes dans lesquels le métrique est à l'état stable. Par exemple, l'altitude d'un aéronef reste stable la majorité du temps entre les phases d'atterrissage et de décollage. Cela implique que notre première méthodologie de détection des déviations ignore les valeurs du métrique pendant le régime transitoire et essaie d'associer chacune des valeurs de sortie du modèle avec l'une des valeurs dans les états stables. Une partie des observations du métrique qui n'étaient pas bien expliquées (modélisées) par nos modèles, correspond aux périodes transitoires, ce qui nous a poussé à explorer une variante de notre méthodologie de base qui permet de cibler le régime transitoire des métriques afin d'y détecter des déviations.

Nous réalisons une étude de cas préliminaire sur le simulateur de vol JSBsim (c.-à-.d, le moteur de simulation de FlightGear). Les résultats montrent que nous n'en somme pas encore à une approche tout à fait au point à cause de ses performances par rapport à un classificateur aléatoire, mais continuer à explorer cette méthodologie pourrait donner des résultats prometteurs et compléter notre approche général de détection des problèmes fonctionnels dans les

simulateurs de vol.

5.1 Adaptation de la méthodologie existante

Utilisation du modèle du simulateur pour détecter les déviations

Pour détecter les déviations dans le régime transitoire des métriques, il faut d'abord détecter les périodes transitoires d'un vol. Pour cela, nous utilisons une particularité du modèle du simulateur que l'on utilise dans la méthodologie de la Section 4.3. En effet, les erreurs de prédiction de l'arbre de décision qui modélise le simulateur se concentrent dans les zones où il y a transition du métrique entre deux états stables. Puisque les transitions sont plus courtes que les états stables et ne durent parfois que quelques secondes, il faut s'assurer que la fréquence d'échantillonnage des métriques est assez élevée pour capter les transitions. Nous avons donc choisi une fréquence de cinquante échantillons par seconde, ce qui est suffisant pour capter les transitions dans les métriques.

Le tableau 5.1 montre un exemple où l'on explique la vitesse d'un avion avec le niveau de l'accélérateur (normalisé), nous avons dans ce cas les règles "Accélérateur (**élevé**) => Vitesse (**élevée**)" et "Accélérateur (**bas**) => Vitesse (**basse**)". Nous pouvons remarquer que durant la transition de la vitesse de "basse" vers "élevée", la vitesse reste basse même si l'accélérateur est "élevé", cette transition viole donc les règles établies. Le fait que la vitesse reste basse malgré sa valeur supérieure à 0 est due au nombre de catégories dans les métriques couvertes par le modèle. Dans le cas de la version déviante, nous avons une période de transition plus longue dans laquelle la vitesse reste basse.

La figure 5.1 montre les zones correspondantes aux périodes de transition où les règles sont violées dans chacune des versions. Ces violations s'expliquent par le fait que l'algorithme d'entraînement de l'arbre de décision ne tient compte que des états stables qui représentent la tendance générale d'un métrique pour expliquer sont comportement, ainsi l'algorithme ne tient pas compte du régime transitoire. Une période de transition se délimite par une subséquence de métriques qui violent les règles du modèle, ce qui correspond respectivement à la zone bleue et à la zone rouge dans le tableau 5.1. La zone rouge est plus large que la zone bleue, faisant en sorte que la période de transition de la version déviante est plus longue que celle dans la version de base. Il est donc possible de comparer ces périodes de transition entre deux versions du simulateur afin d'obtenir un le niveau de similarité entre ces périodes de 0% (pas du tout pareil) à 100% (parfaitement similaire). Par contre, nous ne pouvons pas évaluer la différence dans la forme des courbes de transition en utilisant une telle approche.

Tableau 5.1 Exemple de transition dans un métrique entre une version saine et une version ayant un problème fonctionnel. Les valeurs du métrique violant les règles sont surlignées en bleu (normal) et en rouge (déviant).

Métrique d'Entrée		**Version de Base**		**Version Déviante**	
Accélérateur	Accélérateur Disc.	Vitesse	Vitesse Disc.	Vitesse	Vitesse Disc.
0	bas	0	bas	0	bas
0	bas	0	bas	0	bas
0.8	haut	10	bas	10	bas
0.8	haut	60	bas	40	bas
0.8	haut	130	bas	90	bas
0.8	haut	180	haut	130	bas
0.8	haut	180	haut	150	bas
0.8	haut	180	haut	180	haut
0.8	haut	180	haut	180	haut
0.8	haut	180	haut	180	haut
0.8	haut	180	haut	180	haut
0.8	haut	180	haut	180	haut
0.8	haut	180	haut	180	haut
0.8	haut	180	haut	180	haut
0.8	haut	180	haut	180	haut

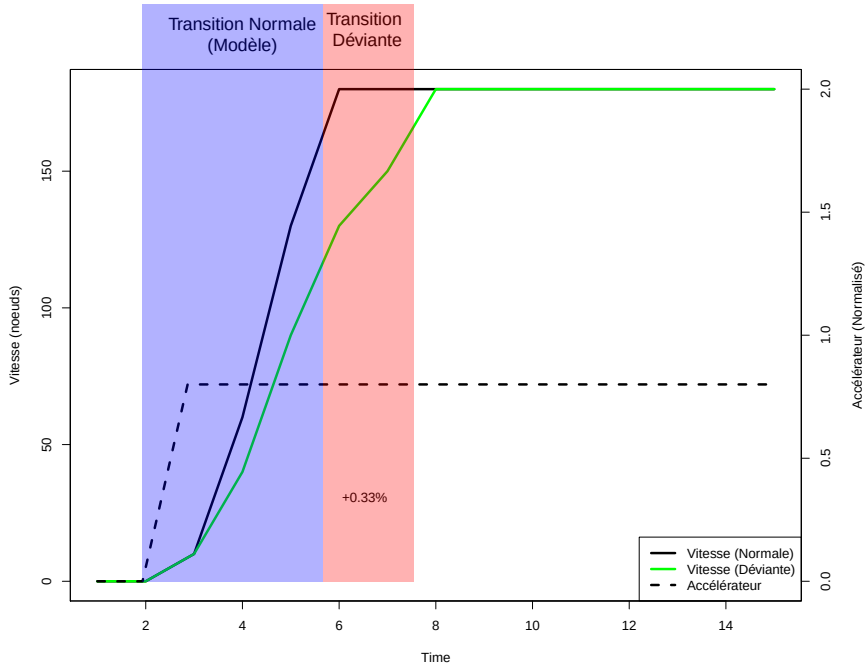


Figure 5.1 Déviation dans la transition du métrique de la Vitesse.

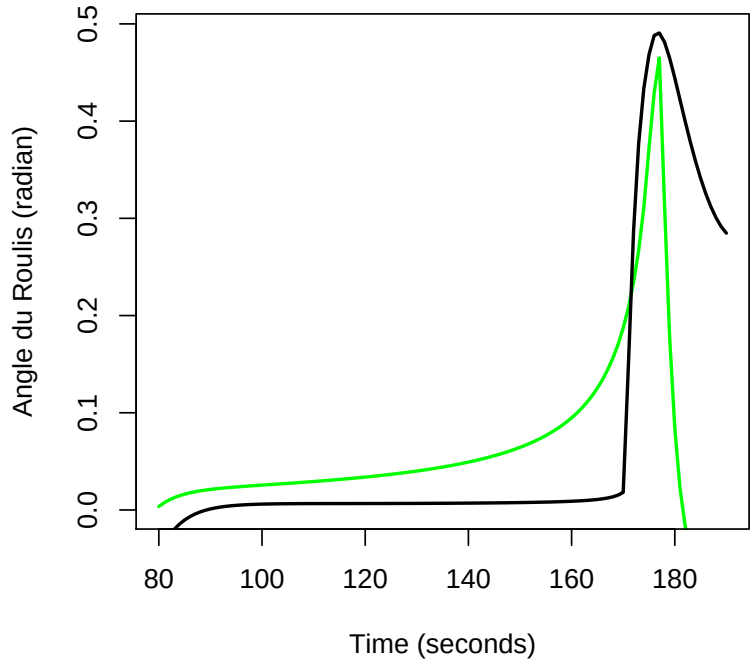
Similarité des périodes de déviation par rapport au modèle de base pour un métrique donné

Nous comparons les périodes de transition détectées dans deux versions en évaluant la proportion du temps que les périodes s'entrelacent. Les deux zones du bas dans la figure 5.2 montrent le chevauchement des périodes de transition (zone en mauve) pour la métrique "Angle du Roulis" entre une version du simulateur (zone en bleu) et une autre version mutante (zone en rouge). Nous pouvons observer que la zone rouge de la nouvelle version est plus large que la zone bleue et que, par conséquent, la zone mauve est plus petite que la zone bleue. Nous ne considérons que les zones de transition du modèle comme référence pour comparer les périodes de transition dans les deux versions. Nous ignorons donc les périodes de transition qui ne chevauche pas de transition dans la version de référence pendant au moins une unité de temps de l'échantillonnage (une seconde dans le cas de notre exemple). Dans le cas où il y a une nouvelle période de transition dans la nouvelle version, il est possible d'utiliser notre approche normale de détection des déviations dans l'état stable des métriques.

Nous calculons la similarité des périodes de transition avec l'équation 5.1 où PC est la période de chevauchement (zone mauve), PNT est la période normale de transition (zone bleue) et PTV est la période de transition dans la nouvelle version (zone rouge). Cette équation calcule la proportion de temps qu'il y a chevauchement des deux périodes (zones) par rapport au temps total occupé par les deux périodes. Dans notre exemple concernant le métrique "Angle du Roulis", nous avons une PC de 57 secondes, une PNT de 64 secondes et une PTV de 122 secondes. Nous obtenons alors une similarité de la période de 44%.

$$\text{Similarité de la Periode} = |\frac{PC}{PNT - PC + PTV}| \tag{5.1}$$

Une trop faible similarité entre les périodes de transition peut être indicateur d'une déviation dans le régime transitoire que l'on obtient dans une nouvelle version du simulateur. De nouveau, nous avons besoin d'un seuil, tel qu'utiliser dans notre approche précédente à la section 4.3.6, pour déterminer quel niveau de similarité discrimine une transition normale d'une transition déviante.



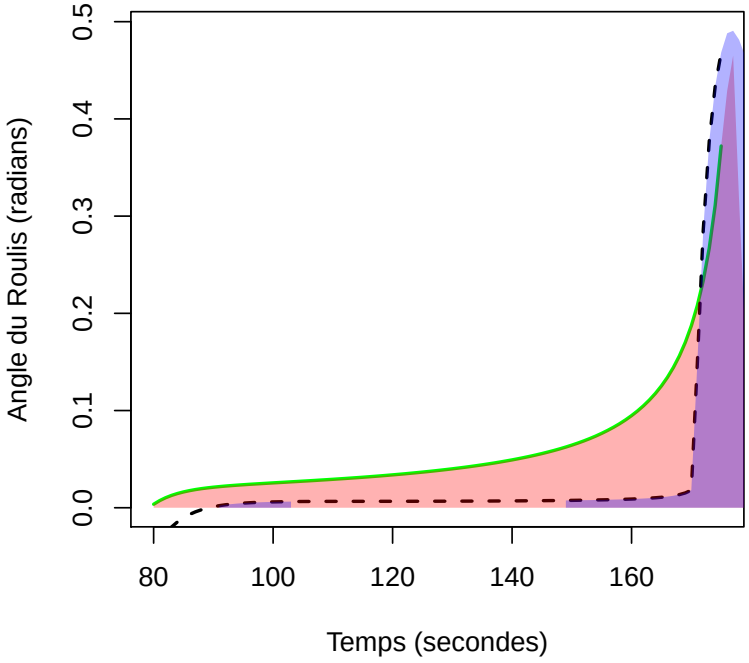


Figure 5.2 Déviation dans la transition du métrique de l'Angle du Roulis. La courbe noire (période de transition bleue) présente la transition normale de la métrique d'altitude et la courbe rouge présente la transition déviante (période de transition rouge). La zone mauve présente le chevauchement entre les deux transitions (rouge par-dessus bleu).

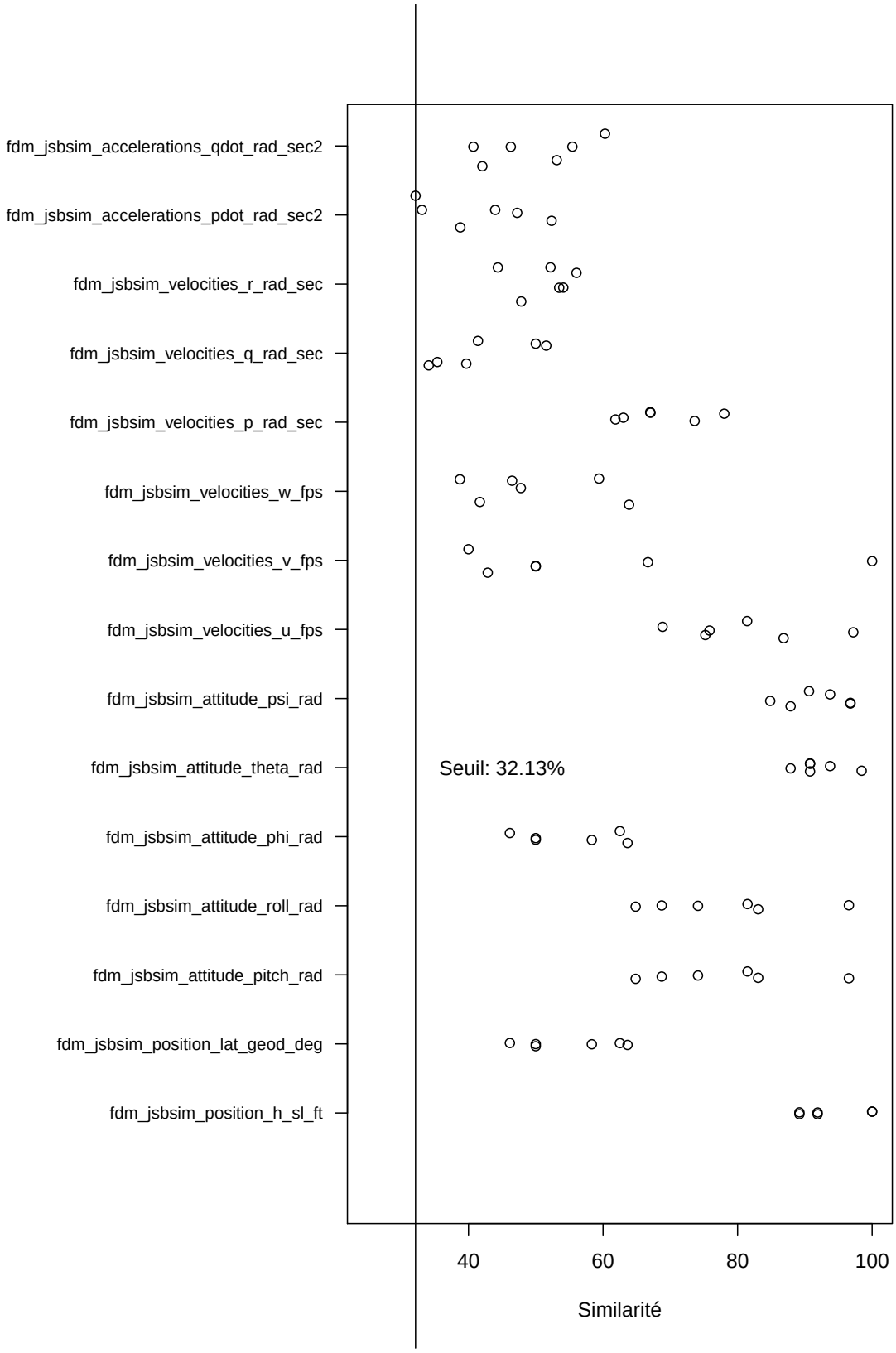


Figure 5.3 Variation normale de l'écart dans les périodes de transition de chacune des métriques enregistrées dans des versions du simulateur ne présentant pas de déviations.

5.2 Étude de cas sur JSBsim

La configuration flexible qu'offre JSBsim est importante, car pour isoler l'évaluation de notre approche de détection des périodes de transition de celle de détection des déviations entre les états stables, il est nécessaire d'utiliser des scénarios de vol où les états stables n'ont pas de déviation. En d'autres mots, il faut des variations de scénario de vol où seulement les périodes de transition peuvent être déviées. JSBsim nous permet de produire des mutations qui ont seulement un impact sur le régime transitoire de certaines métriques. Quoique JSBsim est plus simple que FlightGear [1] au niveau de son environnement de simulation et que les modèles d'aéronef dans JSBsim sont plus simples, JSBsim possède tout de même un moteur de dynamique de vol complet qui nous donne assez de flexibilité pour effectuer notre étude de cas.

Les métriques de vol

Comme FlightGear, JSBsim utilise un arbre de propriétés afin de contenir les données de simulation à partir duquel nous sélections des métriques d'entrée et de sortie pour produire un modèle comportemental du simulateur. Les métriques que l'on peut sélectionner dans JSBsim sont très similaires à ce que l'on peut trouver dans FlightGear. Par exemple le métrique "fdm/jsbsim/position/h-sl-ft" de JSBsim correspond à la métrique "altitude-ft" dans FlightGear (voir tableau 4.3).

Les tableaux 5.2 et 5.3 montrent spécifiquement les métriques d'entrée et de sortie que nous sélectionnons pour bâtir le modèle de JSBsim. Les noms *fcs* (*Flight Control System*) et *fdm* (*Flight Dynamic Model*) que l'on trouve à la racine de l'arborescence des métriques, séparent bien les entrées reliées au contrôle des métriques des sorties reliées à la dynamique de vol.

1. En fait, FlightGear utilise JSBsim comme engin de vol à l'interne, comme discuté à la section 2.5

Tableau 5.2 Métriques d'entrée de JSBsim.

Métriques d'Entrée		
fcs/pitch-trim-cmd-norm	fcs/roll-trim-cmd-norm	fcs/yaw-trim-cmd-norm
fcs/left-aileron-pos-rad	fcs/left-aileron-pos-deg	fcs/left-aileron-pos-norm
fcs/right-aileron-pos-rad	fcs/right-aileron-pos-deg	fcs/right-aileron-pos-norm
fcs/elevator-pos-rad	fcs/elevator-pos-deg	fcs/elevator-pos-norm
fcs/rudder-pos-rad	fcs/rudder-pos-deg	fcs/rudder-pos-norm
fcs/flap-pos-rad	fcs/flap-pos-deg	fcs/flap-pos-norm
fcs/flap-cmd-norm	fcs/speedbrake-cmd-norm	fcs/spoiler-cmd-norm
fcs/aileron-cmd-norm	fcs/elevator-cmd-norm	fcs/rudder-cmd-norm
fcs/steer-pos-deg		

Scénarios de mutation de la version de référence de JSBsim

Afin de générer différentes versions de JSBsim, nous utilisons des scénarios de mutation comme fait à la section 4.4.2. La différence est que l'on crée des scénarios dans lesquels le régime transitoire des métriques de sortie change. Nous élaborons un scénario de léger surpoids (0) dans lequel nous ajoutons du poids dans l'appareil sans aller au-delà des spécifications (nous allons l'utiliser pour calibrer notre approche). Nous élaborons également un autre scénario de surpoids (1) qui est cette fois très élevé, dans lequel nous élevons le poids de l'avion au-delà des spécifications. Finalement, nous élaborons trois scénarios de modification de l'environnement (2-4) comme fait précédemment. Dans ces modifications, nous créons des vecteurs de vent qui ont des angles précis afin de changer la transition de certains métriques. Nous élaborons alors un scénario de fort vent au-dessus de l'avion, un scénario de fort vent croisé et un scénario de fort vent de face. Chacun de ces scénarios a un impact sur la transition de certaines métriques de sortie. Nous faisons plusieurs exécutions de ces scénarios avec de légères différences à chaque fois afin d'évaluer la capacité de notre approche à détecter des déviations dans les transitions.

Évaluation de l'approche et oracle

Pour évaluer notre approche de détection des déviations de transition dans JSBsim, nous calculons la précision, le rappel et l'aire sous la courbe en comparant nos résultats avec un oracle. Nous calculons la précision et le rappel en utilisant les équations 4.3 et 4.4 et en utilisant des variables de ces deux équations nous dessinons la courbe ROC, à partir de laquelle nous évaluons l'aire sous la courbe. Nous élaborons l'oracle à l'aide d'un expert dans le domaine de la simulation qui nous aide à identifier quelles métriques devant présenter une déviation dans leur transition pour chacune des différentes versions de JSBsim que nous avons générée avec nos scénarios de mutation. Contrairement à l'oracle du tableau 4.3, notre oracle prédit chacun des métriques déviant à la place d'englober une famille de métriques parce que le contexte de nos mutations fait dévier certains métriques précis. Il est donc plus aisé d'identifier quelles métriques comportent des déviations dans leur transition pour une version donnée du simulateur.

Calibration du niveau de seuil de détection des déviations

En utilisant une méthode similaire à ce qui a été fait à la section 4.4.6, nous analysons les écarts normals dans la similarité des périodes de transition par rapport au modèle de base pour déterminer un niveau de seuil minimal avec lequel nous ne générons pas de fausses alarmes. Toute valeur en dessous de ce seuil est considérée comme une déviation. À cet effet, nous avons utilisé différentes variantes de notre scénario de léger surpoids en changeant le poids légèrement à chaque fois tout en restant sous la spécification maximale afin d'enregistrer

Tableau 5.3 Oracle qui prédit les déviations transitoires dans les métriques de chacune des versions du simulateur que nous avons générées. Vrai signifie qu'il y a déviation et faux signifie qu'il n'y pas de déviation.

Métriques de Sortie	**Surpoids**	**Vent de Dessus**	**Vent Croisé**	**Vent de Face**
fdm/jsbsim/position/h-sl-ft	vrai	vrai	faux	vrai
fdm/jsbsim/position/lat-geod-deg	faux	faux	vrai	faux
fdm/jsbsim/attitude/pitch-rad	faux	faux	faux	vrai
fdm/jsbsim/attitude/roll-rad	faux	faux	vrai	faux
fdm/jsbsim/attitude/phi-rad	faux	faux	vrai	faux
fdm/jsbsim/attitude/theta-rad	faux	faux	faux	vrai
fdm/jsbsim/attitude/psi-rad	faux	faux	faux	faux
fdm/jsbsim/velocities/u-fps	vrai	vrai	vrai	vrai
fdm/jsbsim/velocities/v-fps	vrai	vrai	vrai	vrai
fdm/jsbsim/velocities/w-fps	vrai	vrai	faux	faux
fdm/jsbsim/velocities/p-rad-sec	vrai	vrai	vrai	faux
fdm/jsbsim/velocities/q-rad-sec	vrai	faux	faux	vrai
fdm/jsbsim/velocities/r-rad-sec	vrai	faux	faux	faux
fdm/jsbsim/accelerations/pdot-rad-sec2	vrai	vrai	vrai	faux
fdm/jsbsim/accelerations/qdot-rad-sec2	vrai	faux	faux	vrai
fdm/jsbsim/accelerations/rdot-rad-sec2	vrai	faux	faux	faux
fdm/jsbsim/accelerations/vdot-ft-sec2	vrai	vrai	vrai	vrai

un ensemble de niveaux de similarité des périodes. Nous utilisons un échantillon basé sur quelques vols pour déterminer le niveau de seuil. Avenant le cas que l'échantillon ne convient pas pour trouver un niveau de seuil optimal, il est possible de d'ajuster le niveau de seuil en conséquence.

La figure 5.3 montre l'ensemble des niveaux de similarité de chacun des métriques de sortie que nous obtenons en analysant chacun des vols exécutés sur des variantes du scénario de léger surpoids. Nous y constatons que la similarité est généralement faible dans l'ensemble des métriques, excepté pour "fdm/jsbsim/position/h-sl-ft" (altitude), "fdm/jsbsim/attitude/theta-rad" (angle de l'axe de tangage) et "fdm/jsbsim/attitude/psi-rad" (angle de l'axe de lacet) qui présentent des niveaux de similarité dans leur transition de 84% et plus. Nous choisissons tout de même de mettre un niveau de seuil de 32.13% qui ne génère aucune fausse alarme dans ces versions du simulateur que nous considérons comme saines. Toute métrique ayant une similarité de sa période de transition sous 32.13% est ainsi considérée comme une déviation.

5.3 Résultats de l'étude de cas

En utilisant le niveau de seuil de 32.13% déterminé par notre calibration, nous évaluons la précisions et le rappel pour chacune des nouvelles versions. Le tableau 5.4 montre les résultats que nous obtenons. En moyenne, nous obtenons un rappel des déviations maximal de 75% avec une précision maximale de 58%.

Tableau 5.4 Évaluation de l'approche de détection de déviation dans la transition des métriques pour chacun des scénarios de mutation.

Version	**Nb. Exécutions**	**Rappel**	**Précision**	**AUC**
Surpoids	5	0.69	0.58	0.62
Vent de Dessus	6	0.75	0.30	0.55
Vent Croisé	10	0.59	0.45	0.51
Vent de Face	6	0.29	0.33	0.50

Pour voir si notre approche fait mieux qu'un classificateur aléatoire des déviations, nous avons analysé l'aire sous la courbe en parcourant différents niveaux de seuil de 0% à 100% avec des pas de 5%. La figure 5.5 présente la courbe Receiver Operating Characteristic (ROC) mettant en relation le taux de faux positifs et celui de vrais positifs lors de la détection des déviations dans les différentes versions du simulateur, l'aire sous la courbe y est affichée. Une valeur au-dessus de 0.5 (c.-à-d., une courbe généralement au dessus de la ligne grise) signifie que notre approche de détection est meilleure qu'un classificateur aléatoire.

Nous pouvons constater que notre approche a la meilleure aire sous la courbe dans le cas du scénario de surpoids avec une aire sous la courbe de 0.62. Pour le reste des versions, notre approche a une aire sous la courbe qui varie de 0.50 à 0.55 (c.-à-.d, près du classificateur aléatoire). En d'autres mots, l'approche semble ne pas bien marcher pour les scénarios où nous avons changé les conditions de vent puisque la classification des déviations de notre approche s'approche de celle d'un classificateur aléatoire.

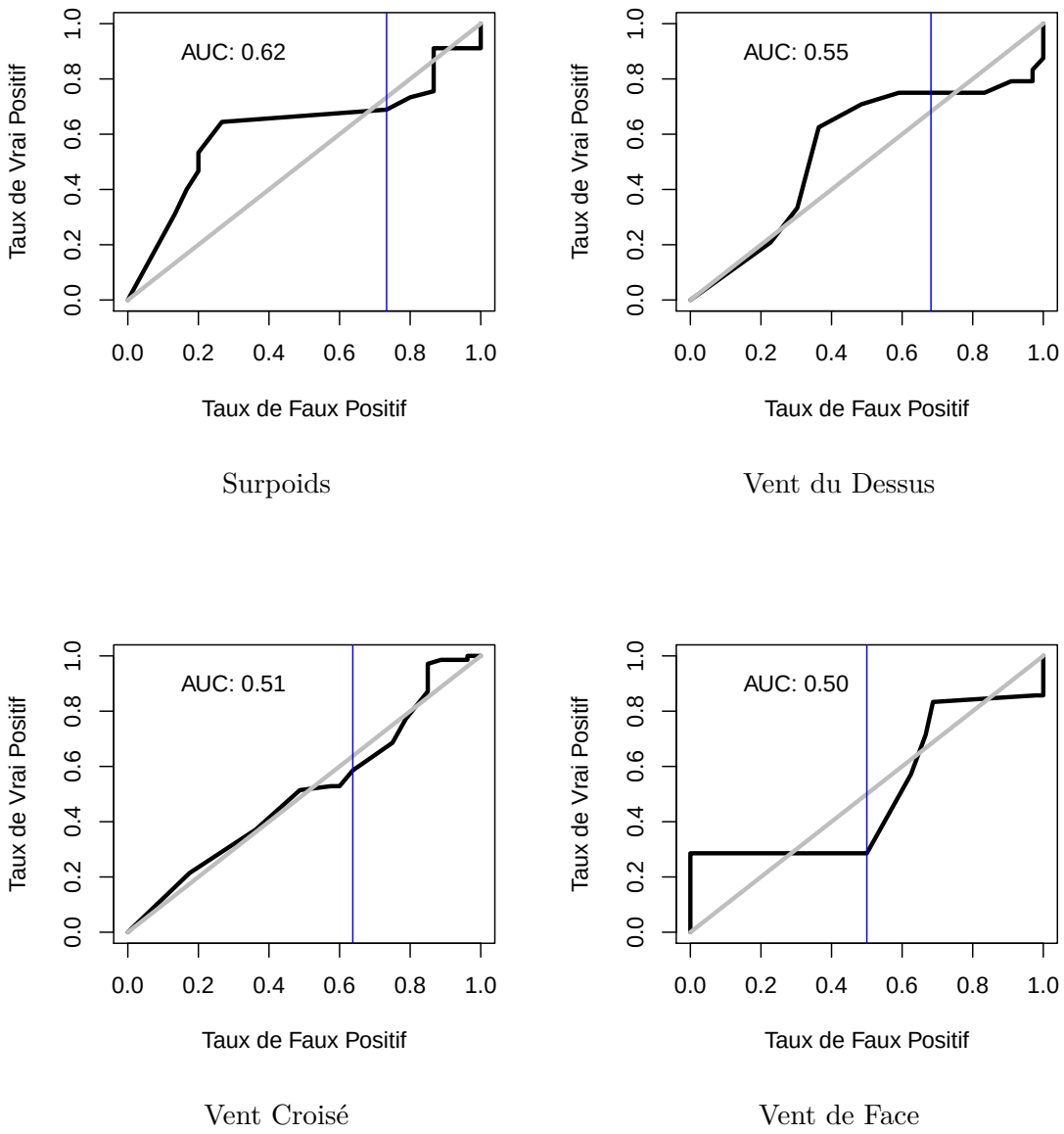


Figure 5.5 Courbe ROC pour chacun des scénarios. Les lignes bleues verticales correspondent au niveau de seuil choisi.

5.4 Discussion et améliorations futures

Bien que l'on obtient une précision maximale de 58% et un rappel maximal de 75%, notre approche détecte les déviations dans les versions du simulateur qui changent le vent avec

une performance similaire à un classificateur aléatoire. Nous avons souligné lors de notre précédente discussion portant sur l'optimisation de la détection des déviations (section 4.6.1) qu'il est possible d'ajuster le niveau de seuil de détection pour chacune des versions pour obtenir une meilleure performance. Nous pourrions donc appliquer la même méthodologie pour optimiser la détection des déviations transitoires.

Nous avons fait cette première expérimentation en utilisant une idée simple de détection des déviations dans les périodes de transition en utilisant le modèle d'un simulateur. Nous devrions dans de futurs travaux explorer des techniques plus complexes permettant de mieux cibler les transitions pour détecter les déviations. Il sera également nécessaire d'avoir plus de variétés dans les versions testées de JSBsim ou d'un autre simulateur. Nous nous sommes limités dans cette première étude de cas à des mutations sur le poids de l'aéronef et des conditions de vent dans l'environment.

CHAPITRE 6 DISCUSSION GENERALE

Dans ce mémoire nous visons à automatiser l'identification des problèmes fonctionnels dans les simulateurs de vol en répondant à trois objectifs qui sont : de modéliser le simulateur en utilisant l'apprentissage automatique, de détecter les écarts dans les métriques que l'on enregistre dans des nouvelles versions du simulateur et d'être en mesure de détecter automatiquement les déviations dans les cas où une nouvelle version du simulateur comporte des problèmes fonctionnels. Nous discutons dans cette section de la méthodologie et des résultats que nous avons obtenus pour rencontrer ces trois objectifs tout en nous référant à la littérature existante.

6.1 Objectifs 1 et 2 : modélisation du simulateur et détection des écarts dans les métriques

Dans la littérature, on utilise plusieurs techniques d'apprentissage automatique afin de modéliser un système logiciel à partir de données enregistrées. Parmi ces techniques, il y a la régression logistique, les réseaux de neurones, les modèles de Bayes, les règles associatives et les arbres de décision. Puisque nous visons à travailler avec des systèmes composés de boîtes noires, il est intéressant de choisir un modèle permettant un visuel graphique du comportement du système. Nous optons donc pour des arbres de décisions qui sont des modèles qui offrent une abstraction visuelle en utilisant une représentation d'arbre composé de noeuds, de feuilles et de branches. D'autres travaux optent pour une modélisation utilisant des règles associatives qui mettent en lien des observations sur les métriques sans les diviser entre entrées et sorties. Puisque nous cherchons à expliquer l'impact d'un stimulus (c.-à-.d, des entrées) sur le comportement (c.-à-.d, les sorties) du simulateur, nous sélectionnons avec l'aide d'experts un ensemble de métriques de sortie et d'entrée, ce qui nous permet d'avoir un modèle représentant le système et de remplir notre premier objectif.

Les autres techniques d'apprentissage automatique, malgré une représentation plus fidèle du sytème modélisé dans certains cas, n'offrent pas une représentation graphique du comportement du système. Même si nous cherchons à avoir un modèle représentant parfaitement notre système, il faut faire attention au sur-ajustement du modèle qui n'est pas préférable puisque l'on veut être tolérant aux variations normales dans les métriques enregistrées sous un même stimulus (la figure 4.3 montre ce phénomène). Nous voulons ainsi éviter de générer trop de fausses alarmes, ce qui rendrait l'adoption de notre approche de test inutile. En effet, un système considérant toute métrique comme étant déviante aurait un taux de rappel parfait

avec une précision très faible, ce qui alourdirait la tâche des testeurs qui devraient vérifier le système en entier. La calibration à la section 4.4.6 montre que l'écart des métriques par rapport au modèle peut aller jusqu'à 41%. Cela confirme notre intuition qu'il faut adopter une approach de détection flexible et tolérante aux écarts tout en ne manquant pas les déviations lorsqu'elles sont présentes.

6.2 Objectif 3 : automatisation de la détection avec un niveau de seuil

Nous proposons une approche qui détecte automatiquement les déviations dans des métriques que nous enregistrons pendant le pilotage d'un simulateur de vol, permettant ainsi d'identifier ses problèmes fonctionnels. L'avantage de détecter les déviations au niveau des métriques est mis en valeur dans la littérature, permettant ainsi de mieux identifier la nature d'un problème. Les résultats de l'étape de calibration à la section 4.4.6 montrent que les écarts des métriques par rapport à leurs signatures peuvent être très disparates. Nous sélectionnons donc un premier niveau de seuil de détection juste au dessus de la médiane d'écart la plus haute afin d'éviter les fausses alarmes avec les versions ayant un comportement normal tout en maximisant le rappel des déviations dans les versions avec problème fonctionnel.

Un niveau de seuil optimisé pour un meilleur rappel réduit la précision de notre approche, nous obtenons donc initialement un taux de fausses alarmes de 60% et une précision aussi basse que 40%. Nous améliorons notre approche dans la discussion de la section 4.6.1 en proposant des niveaux de seuil sur les métriques adaptés à chacune des versions, obtenant ainsi un taux de fausses alarmes parfait pour les versions n'ayant pas de problèmes fonctionnels et, pour les versions ayant des problèmes fonctionnels, une précision minimale de 50% tout en ne manquant aucune déviation avec un taux de rappel de 100%.

La fluctuation de la précision et du rappel de notre approche en fonction des modifications apportées au simulateur se manifeste aussi dans l'approche développée par Foo et al. [10]. Les auteurs obtiennent une précision fluctuant entre 75% et 100% et un rappel fluctuant entre 54% et 67% en fonction de l'erreur qu'ils injectent dans le système testé. Dans notre cas, avec l'utilisation des seuils optimaux, nous obtenons un taux de rappel de 100% et une précision variant de 50% à 100%. En comparant les résultats, nous remarquons que Foo et al. mettent plus l'accent sur la précision de leur approche, tandis que nous préférons le rappel dans notre cas à cause du niveau critique du système que l'on test. L'orientation des résultats est donc en fonction du contexte dans lequel nous détectons les problèmes fonctionnels. Le dilemme suivant se pose donc, veut-on assurer la sécurité du système ou veut-on simplement mieux assister les testeurs en leurs donnant des diagnostics précis ?

CHAPITRE 7 CONCLUSION ET RECOMMENDATIONS

7.1 Synthèse des travaux

Pour atteindre le premier objectif de modéliser un simulateur de vol, nous avons modélisé le simulateur de vol FlightGear en utilisant une approche de modélisation boîte noire avec laquelle il est possible de modéliser le comportement modèle du simulateur sans avoir accès à son code source. Pour générer le modèle de FlightGear, nous avons manuellement sélectionné, avec des experts en simulation, des métriques contenues dans l'arbre de propriétés du simulateur afin de concevoir un ensemble d'entraînement séparé en métriques d'entrée et de sortie. Nous aurions pu utiliser des algorithmes automatiques de sélection des métriques pour éviter d'avoir recours à des experts, mais la sélection manuelle nous a permise d'avoir un modèle de bonne qualité. Nous avons exécuté un vol sur une version normale de FlightGear afin d'enregistrer un ensemble de métriques d'entraînement, à partir desquels nous avons bâti un modèle composé d'arbres de décision qui expliquent la relation entre les métriques d'entrée et de sortie. L'arbre de décision nous a offert une abstraction graphique du comportement du système modélisé.

Pour atteindre le deuxième objectif de détecter des écarts dans les métriques, nous avons comparé, pour chacune des nouvelles versions de FlightGear, les métriques enregistrées avec chacune des signatures composant notre modèle pour ainsi obtenir le niveau d'écart de chacune des métriques de sortie. Les résultats que nous avons obtenus lors de nos expérimentations sur différentes versions de FlightGear (voir figure 4.3) montrent qu'il y a des écarts dans les métriques par rapport au modèle. Ces écarts peuvent être très volatiles, ce qui nécessite de cumuler des données provenant de plusieurs répétitions du même vol avant de choisir un niveau de seuil pour automatiser notre approche.

Pour atteindre le troisième objectif de détecter automatiquement les déviations dans les métriques, nous avons utilisé le niveau d'écart dans les métriques de sortie (voir figure 4.3) afin de détecter automatiquement les déviations introduites dans les nouvelles versions de FlightGear comportant des problèmes fonctionnels. Basé sur un historique de données, nous avons sélectionné un niveau de seuil qui détermine quel écart dans une métrique doit être considéré comme une déviation. Pour évaluer la performance de notre approche automatisée, nous avons demandé l'avis d'experts en simulation afin d'identifier les métriques ayant des déviations pour chacune des versions de FlightGear. Ces avis ont constitué notre oracle de référence pour faire l'évaluation de notre approche. Nous avons évalué la précision, le rappel et le taux de fausses alarmes de notre approche. Les résultats de cette évaluation montrent

que nous avons été capable d'identifier toutes les déviations avec un rappel de 100% et avec une précision d'au moins 50%. En ce qui a trait aux versions de FlightGear n'ayant pas de problèmes fonctionnels, notre approche ne génère pas de fausses alarmes.

7.2 Limitations de la solution proposée

Accès à l'expertise pour bâtir les modèles

Une fois mise sur pied, notre approche détecte les déviations dans les versions problématiques de FlightGear de manière complètement autonome. Puisque nous voulons un modèle de qualité, nous avons consulté des experts en simulation de vol au départ pour sélectionner les métriques d'entrée et de sortie qui sont utilisées pour bâtir ce modèle. Par contre, la modélisation d'un composant précis du simulateur (p. ex., un composant hydraulique) peut nécessiter une expertise de pointe qui n'est parfois disponible que chez le fournisseur qui a développé le composant (dont nous avons pas accès au code source), ce qui rend l'accès à cet expertise plus difficile pour le manufacturier du simulateur de vol. Nous pensons, que dans un premier effort de modélisation, qu'une connaissance générique du composant est suffisante.

Applicabilité de notre approche sur d'autres systèmes

Nous n'avons réalisé qu'une seule étude de cas de notre approche sur FlightGear, ce qui ne couvre pas les autres simulateurs de vol ou les autres systèmes du même genre. En effet, dans d'autres systèmes composés de boîtes noires, l'architecture globale peut être différente et la technologie pour enregistrer les métriques qualifiant le comportement du système peut également être différente. Notre approche repose entièrement sur l'analyse des résultats de test, elle ne dépend donc pas d'une plateforme en particulier, il nous faut seulement des métriques. Par exemple, il nous serait possible d'utiliser notre approche pour générer le modèle d'un vrai avion en utilisant ses données collectées dans le processus de la FOQA. Le principal défi d'appliquer notre approche sur d'autres systèmes serait au niveau de l'enregistrement et la sélection des métriques pour modéliser un système. Nous sommes confiant que l'application de notre approche sur d'autres études de cas, une fois les métriques obtenues, donnera des résultats dans la même direction que ceux que l'on a obtenu avec FlightGear.

Nous avons réalisé ce mémoire dans le cadre d'un projet de recherche industriel en collaboration avec la compagnie de simulateurs de vol CAE en collaboration avec le CRIM et avec le financement du CRSNG qui nous a octroyé une bourse collaborative en recherche et développement. Nous visons donc à long terme à appliquer l'approche que nous avons développée dans le cadre de ce mémoire sur des simulateurs de vol plus complexes que FlightGear, tels que ceux développés par la compagnie CAE.

Sensibilité de notre approche dans la détection des déviations

Les nouvelles versions problématiques de FlightGear que nous avons créées avaient des problèmes fonctionnels bien délimités. En effet, nous avons injecté des changements importants, tels qu'un fort vent ou la perte d'un moteur, ce qui peut avoir facilité la tâche à notre approche lors de la détection des déviations. Le problème fonctionnel d'un système ou de l'un de ses composants pourrait provenir de légères modifications au simulateur. Par exemple, un coéfficient de contrôle qui change légèrement pourrait avoir un impact dans le comportement du simulateur. C'est pour cela que nous proposons initialement de sélectionner un niveau de seuil sensible (légèrement au dessus du minimum) afin d'amoindrir les chances de manquer une déviation. Un testeur qui utilise notre approche doit faire un choix entre la précision et le rappel lorsqu'il détermine le niveau de seuil de détection des déviations. Il est donc essentiel de passer par des étapes de calibration afin de déterminer quels niveaux de seuils sont les plus efficace en fonction du contexte dans lequel notre approche est utilisée.

7.3 Améliorations futures

7.3.1 Isoler la source du problème

Nous avons apporté dans ce mémoire une solution afin de détecter automatiquement la présence de déviations dans le comportement d'une nouvelle version d'un simulateur de vol en analysant l'écart des métriques enregistrés lors d'une séance de vol. Il serait intéressant de mettre ces informations à profit afin de mieux cibler la source d'un problème, facilitant ainsi le processus de déverminage d'un système. Par exemple, il serait possible de descendre la granularité des modèles au niveau des composants et ainsi identifier précisément quels composants ont des déviations dans leur comportement.

7.3.2 Détecter les déviations transitoires

Nous mettons en évidence dans nos résultats complémentaires au chapitre 5, que nous devons également considérer les déviations dans le régime transitoire des métriques qui peuvent soit affecter la période de transition ou la forme même de la transition. Nous avons donc exploré comment nous pourrions utiliser le modèle de notre approche initiale pour détecter les déviations dans les périodes de transition. Cette première exploration nous montre que nous somme capable d'isoler les transitions, mais qu'il est difficile de détecter les déviations dans les périodes de transition de manière fiable. Il est par contre pertinent de faire de futures recherches sur ce sujet afin de compléter notre approche initiale de détection des déviations. Outre la période de transition elle même, il y a la forme de la transition. À cet égard, plusieurs

travaux dans le domaine industriel et de l'aéronautique se sont attaqués à la détection des anomalies de transition [51, 52, 53, 54]. Les auteurs utilisent des signatures comportementales et des signaux de référence afin de détecter des déviations dans les transitions qui sont reliées à des problèmes fonctionnels. Il serait possible d'utiliser une partie de la méthodologie mise en valeur dans ses précédents travaux afin de détecter les déviations transitoires dans les simulateurs de vol et autres systèmes du genre.

7.4 Articles réalisés dans le cadre de ce mémoire

Dans le cadre de ce mémoire, nous avons réalisé un total de trois articles de conférence et de journal. Nous avons soumis l'article inclus dans ce mémoire dans le *Journal of Systems and Software* : *Signature-based Detection of Behavioural Deviations in Flight Simulators – Case Study on Flight Gear*. Nous avons eu un résumé accepté qui nous permet d'écrire un papier pour la conférence AIAA (*American Institute of Aeronautics and Astronautics*) sur la détections des déviations de nature transitoire dans les simulateurs de vol (*Automatic Detection of Behavioural Deviations in Flight Simulators*). Nous avons également eu un papier complémentaire à ce mémoire accepté et présenté à la *International Working Conference on Source Code Analysis and Manipulation* sur l'analyse de l'impact de la duplication des rapports de bogue à travers différentes distributions de Linux : *The Impact of Cross-Distribution Bug Duplicates, Empirical Study on Debian and Ubuntu.*

RÉFÉRENCES

[1] R. T. Hays, J. W. Jacobs, C. Prince, and E. Salas, "Flight simulator training effectiveness : A meta-analysis." *Military Psychology*, vol. 4, no. 2, p. 63, 1992.

[2] D. W. Babka, "Flight testing in a simulation based environment," 2011.

[3] K. S. J.M. Rolfe, *Flight Simulation*. Cambridge University Press, 1986.

[4] G. du Canada, *Flight Testing in a Simulation Based Environment*, 1998.

[5] J. A. Whittaker, J. Arbon, and J. Carollo, *How Google tests software*. Addison-Wesley, 2012.

[6] J. Smart, *Jenkins : The Definitive Guide*. O'Reilly, 2011.

[7] B. Beizer and J. Wiley, "Black box testing : Techniques for functional testing of software and systems," *IEEE Software*, vol. 13, no. 5, p. 98, 1996.

[8] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "Autoblacktest : Automatic black-box testing of interactive applications," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 81–90.

[9] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection : A survey," *ACM Computing Surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[10] K. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 2010, pp. 32–41.

[11] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 125–134.

[12] J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test application frequency," *Software Testing, Verification and Reliability*, vol. 15, no. 4, pp. 257–279, 2005.

[13] T. H. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "An industrial case study of automatically identifying performance regression-causes," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 232–241.

[14] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "Leveraging performance counters and execution logs to diagnose memory-related performance issues," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 110–119.

[15] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 468–479.

[16] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[17] I. H. Witten and E. Frank, *Data Mining : Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[18] S. Hoseini, A. Hamou-Lhadj, P. Desrosiers, and M. Tapp, "Software feature location in practice : debugging aircraft simulation systems." in *ICSE Companion*, 2014, pp. 225–234.

[19] A. SAFETY, "Efforts to implement flight operational quality assurance programs," *AVIATION SAFETY*, 1997.

[20] T. J. Callantine, "Analysis of flight operational quality assurance data using model-based activity tracking," SAE Technical Paper, Tech. Rep., 2001.

[21] L. Li, M. Gariel, R. J. Hansman, and R. Palacios, "Anomaly detection in onboard-recorded flight data using cluster analysis," in *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*. IEEE, 2011, pp. 4A4–1.

[22] S. Das, S. Sarkar, A. Ray, A. Srivastava, and D. L. Simon, "Anomaly detection in flight recorder data : A dynamic data-driven approach," in *American Control Conference (ACC), 2013*. IEEE, 2013, pp. 2668–2673.

[23] A. J. Stolzer and C. Halford, "Data mining methods applied to flight operations quality assurance data : a comparison to standard statistical methods," *Journal of Air Transportation*, vol. 12, no. 1, p. 6, 2007.

[24] A. R. Perry, "The flightgear flight simulator," in *2004 USENIX Annual Technical Conference, Boston, MA*, 2004.

[25] R. S. Burns, M. M. Duquette, J. B. Howerton, and R. J. Simko, *Development of a low-cost simulator for demonstration and engineer training*. Defense Technical Information Center, 2003.

[26] E. F. Sorton and S. Hammaker, "Simulated flight testing of an autonomous unmanned aerial vehicle using flightgear," *American Institute of Aeronautics and Astronautics, AIAA 2005*, vol. 7083, 2005.

[27] Z. Qi, D. Hu, A. Liu, and X. Hu, "Airborne imu simulation based on simulink and flight gear," *Journal of Chinese Inertial Technology*, vol. 16, no. 4, pp. 400–403, 2008.

[28] J. Qi, J. Liu, B. Zhao, S. Mei, J. Han, and H. Shang, "Visual simulation system design of soft-wing uav based on flightgear," in *Mechatronics and Automation (ICMA), 2014 IEEE International Conference on*. IEEE, 2014, pp. 1188–1192.

[29] D. Allerton, "The impact of flight simulation in aerospace," *Aeronautical Journal*, vol. 114, no. 1162, pp. 747–756, 2010.

[30] M. Scheer, F. M. Nieuwenhuizen, H. H. Bülthoff, and L. L. Chuang, "The influence of visualization on control performance in a flight simulator," in *Engineering Psychology and Cognitive Ergonomics*. Springer, 2014, pp. 202–211.

[31] P. Gontar, H.-J. Hoermann, J. Deischl, and A. Haslbeck, "How pilots assess their non-technical performance–a flight simulator study," *Advances in Human Aspects of Transportation : Part I*, vol. 7, p. 119, 2014.

[32] N. Gandhi, N. Richards, and A. Bateman, "Simulator evaluation of an in-cockpit cueing system for upset recovery," in *AIAA Guidance, Navigation, and Control Conference*, 2014.

[33] T.-A. Pham, "Validation and verification of aircraft control software for control improvement," Ph.D. dissertation, Citeseer, 2007.

[34] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 2004, pp. 284–293.

[35] E. L. Duke, "V&v of flight and mission-critical software," *Software, IEEE*, vol. 6, no. 3, pp. 39–45, 1989.

[36] S. A. Jacklin, M. R. Lowry, J. M. Schumann, P. P. Gupta, J. T. Bosworth, E. Zavala, J. W. Kelly, K. J. Hayhurst, C. M. Belcastro, and C. Belcastro, "Verification, validation, and certification challenges for adaptive flight-critical control system software," in *American Institute of Aeronautics and Astronautics (AIAA) Guidance, Navigation, and Control Conference and Exhibit*, 2004, pp. 16–19.

[37] G. Köksal, İ. Batmaz, and M. C. Testik, "A review of data mining applications for quality improvement in manufacturing industry," *Expert systems with Applications*, vol. 38, no. 10, pp. 13 448–13 467, 2011.

[38] N. Wilde and M. C. Scully, "Software reconnaissance : mapping program features to code," *Journal of Software Maintenance : Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.

[39] M. S. Stephens and W. I. Ukpere, "An empirical analysis of the causes of air crashes from a transport management perspective," *Mediterranean Journal of Social Sciences*, vol. 5, no. 2, p. 699, 2014.

[40] V. J. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, no. 2, pp. 85–126, 2004.

[41] V. Barnett and T. Lewis, *Outliers in statistical data*. Wiley New York, 1994, vol. 3.

[42] C. C. Aggarwal, "On abnormality detection in spuriously populated data streams." in *SDM*. SIAM, 2005, pp. 80–91.

[43] K. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "An industrial case study on the automated detection of performance regressions in heterogeneous environments," in *Software Engineering In Practice (SEIP) track at the 37th International Conference on Software Engineering, ICSE (Florence, Italy)*. IEEE, 2015, p. to appear.

[44] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating instrumentation data to system states : A building block for automated diagnosis and control." in *OSDI*, vol. 4, 2004, pp. 16–16.

[45] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 105–118.

[46] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *International Conference on Software Engineering*, 2015, pp. 483–493.

[47] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments ?[software testing]," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 402–411.

[48] G. Campbell and R. Lahey, "A survey of serious aircraft accidents involving fatigue fracture," *International Journal of Fatigue*, vol. 6, no. 1, pp. 25–30, 1984.

[49] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software : an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[50] R Core Team, *R : A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2014. [Online]. Available : http://www.R-project.org/

[51] M. Timusk, M. Lipsett, and C. K. Mechefske, "Fault detection using transient machine signals," *Mechanical Systems and Signal Processing*, vol. 22, no. 7, pp. 1724–1749, 2008.

[52] M. J. Roemer and G. J. Kacprzynski, "Advanced diagnostics and prognostics for gas turbine engine risk assessment," in *Aerospace Conference Proceedings, 2000 IEEE*, vol. 6. IEEE, 2000, pp. 345–353.

[53] R. Martin, M. Schwabacher, N. Oza, and A. Srivastava, "Comparison of unsupervised anomaly detection methods for systems health management using space shuttle," in *Main Engine Data," Proceedings of the Joint Army Navy NASA Air Force Conference on Propulsion, 2007.* Citeseer, 2007.

[54] D. L. Simon and A. W. Rinehart, "A model-based anomaly detection approach for analyzing streaming aircraft engine measurement data," in *ASME Turbo Expo 2014 : Turbine Technical Conference and Exposition.* American Society of Mechanical Engineers, 2014, pp. V006T06A032–V006T06A032.