Signature-based detection of behavioural deviations in flight simulators - Experiments on FlightGear and JSBSim

Vincent Boisselle ¹, Giuseppe Destefanis Corresp. ², Agostino De Marco ³, Bram Adams ¹

¹ Computer Science, Polytechnique Montréal, Montreal, Quebec, Canada
² Computer Science, CRIM Montreal - Brunel University, London, United Kingdom
³ Dipartimento di Ingegneria industriale, University of Naples Federico II, Napoli, Italy

Corresponding Author: Giuseppe Destefanis
Email address: giuseppe.destefanis@brunel.ac.uk

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 nine types of injected faults in the popular FlightGear and JSBSim open source simulators shows that our approach does not miss any deviating behaviour considering faults which change the flight environment, and that we are able to find all the injected deviations in 4 out 7 functional faults and 75% of the deviations in 2 other faults.
Signature-based detection of behavioural deviations in flight simulators - Experiments on FlightGear and JSBSim

Vincent Boisselle
Polytechnique Montreal
vincent.boisselle@polymtl.ca

Giuseppe Destefanis
CRIM, Montreal - Brunel University London
giuseppe.destefanis@crim.ca

Agostino De Marco
University of Naples Federico II
agostino.demarco@unina.it

Bram Adams
Polytechnique Montreal
bram.adams@polymtl.ca

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 nine types of injected faults in the popular FlightGear and JSBSim open source simulators shows that our approach does not miss any deviating behaviour considering faults which
change the flight environment, and that we are able to find all the injected deviations in 4 out of 7 functional faults and 75% of the deviations in 2 other faults.

*Keywords:* behavioural deviation, performance regression testing, flight simulator

1. Introduction

Aircraft simulators [3] reproduce common scenarios such as “take-off”, “automatic cruising” and “emergency descent” in a virtual environment [44]. 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 [25]. 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 [1, 24, 39] 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 [35, 53]. Upgrades of a single component (engine, hydraulic component, electrical system) could lead to a complete flight simulator re-qualification process.

Re-qualification is an important problem to tackle since the process is very expensive and it is one of the main sources of expenses for flight simulator companies. Upgrades, new versions, and new functionality are continuously...
delivered from COTS providers to flight simulator companies. The changes these components can introduce can vary from different output values (different thresholds, different unit of measures) to completely new interfaces. The main challenge is to avoid a complete re-qualification of the whole flight simulator, by focusing the attention only to a single or a specific set of components. Since the avionics domain lacks robust tool support for automating tasks such as analysis of COTS interactions, change propagation [27], test selection and test prioritization, even a small modification to a COTS still requires (manual) re-qualification of the whole system [18] [31] [34] [50].

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 [46]. 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 [22] 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 and JSBSim [6] open source simulators at different granularity
levels, in which we introduce 2 faults related to the flight environment and 7 faults related to flight behaviour to address the following research questions:

**RQ1:** What is the performance of the approach for faults that change the flight environment? It is possible to have no false alarms for one environmental fault, whereas flagging all deviations for a second environmental fault is not possible without having 50% false alarms.

**RQ2:** What is the performance of the approach for functional faults? We are able to find all injected deviations in 4 out of 7 faults, and 75% of the deviations in 2 other faults, yet especially for JSBSim the precision using the initial thresholds is low (i.e., less than 44%).

2. Background and Related work

Flight simulators are tested like 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.

Several studies analyzed the possibility of automatising tests for flight simulators. Braun et al. [9] reported about automated fidelity test system which compares flight test results and manual execution of flight tests in simulators. Durak et al. [19] presented a metamodel for objective flight simulator evaluation using model based testing approach. The authors developed an Experimental Frame Ontology (EFO) adopting experimental frames from Discrete Event System Specification (DEVS), and as an upper ontology to specify a formal structure for simulation test. Domain specific meta-test definitions were captured in Objective Fidelity Evaluation Ontology. In an other paper, Durak et al. [20] used Model-Based Testing approach for flight simulator objective fidelity evaluation of engineering and research simulators. Similarly, Wang et al. [48, 49] implemented an automated test system for evaluating flight simulator fidelity (determined by direct comparison to flight test). The system was designed to automatically test the performance of flight simulators.
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. [29] 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” [1]. Our work is closest to outlier detection, which Hodge et al. [29] define as “detection of abnormal running conditions”.

Similarly, Chandola et al. [11] 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 also provide a discussion on the computational complexity of the selected techniques.

Aggarwal [2] 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 [12, 22, 32, 33, 36, 37, 52]. 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. [22] 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 [23], 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. [37] 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 [13] 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 [14], 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. [32], presented a statistical approach that analyzes the execution logs of a load test for detecting performance problems. Syer et al. [47] proposed an automated statistical approach combining execution logs and performance counters to diagnose memory issues in load tests.

Beschastnikh et al. [8] developed a tool called CSight to mine logs from a systems 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 [21], 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. [30] 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.

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 [22]. Fig. 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.

3.1. Simulator Versions and Use Case

A baseline version of a simulator is used as reference to test the behaviour of other versions of the system against. Usually, the latter versions are relatively 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, a release with a specific bug 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 the baseline version changed in a specific way.

Having chosen two versions, we also need to identify representative use cases (i.e., execution scenarios) 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.

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 (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 (instead of metric data) is outside the scope of this paper.

The metrics recorded during execution of a flight simulator 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. Classifying metrics as input or output cannot be done automatically because it requires basic knowledge about the

Figure 1: Overview of our approach.
system behaviour. Internal metrics, i.e., metrics that are neither input nor output are ignored in our study, since they are not observable externally.

In our running example (Table 1), we know that 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.

### 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 (floating point numbers) into a limited number of discrete values such as “low”, “medium”, 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 substantially faster than 25 miles per hour and blazingly faster than 3 miles per hour. For this reason, considering we wish to discretize into three categories,
the discretization step could convert the first two values to the same discrete value ("high"), while the second value could be mapped to "medium" and third one to "low".

To discretize numeric data, we used the popular equal frequency bin discretization algorithm [51], which clusters data in K bins, each of which ideally containing 1/K of the data. Basically, this algorithm first sorts the data from small to large values, then splits it as much as possible into K equally sized bins. In our running example, the airspeed output of the baseline version takes on the values “[0, 18, 36, 54, 72, 90, 108, 126, 144, 162, 180, 180, 180, 180, 180]”, which the equal frequency algorithm set with K=3 bins would split into bins “[−inf, 81]”, “[81, 171]” and “[171, inf]” that can be labeled as “low”, “medium” and “high”. [Table 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 the Throttle value “1.0” in [Table 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 [51].

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 [51]. In addition, decision trees can easily be transformed into a “rule” form that is straightforward to check automatically on new data.

[Fig. 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 Low Throttle corresponds to Low Airspeed, Medium Throttle corresponds to Medium Airspeed and a High Throttle corresponds to High Airspeed. We can summarize this tree into the three rules “Low Throttle ⇒ Low Airspeed”, “Medium Throttle ⇒ Medium Airspeed”, and
“High Throttle ⇒ High Airspeed”. Trees can have an arbitrary number of
test nodes, in which case the rules become more complex, for example “High
A and Medium 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 would not generalize to other cases
(simulator versions), rendering them ineffective [51]. This means that in
practice 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
or too many bins are used to discretize the data, accidentally putting data
points into the wrong bin. Table 1 shows that while transitioning from 0.0
Throttle to 1.0 (blue cells), our vertical speed is still considered as medium
in the baseline version (despite high throttle) due to the limitations of the
discretization used.

### 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 calculate for each signature the proportion
of incorrect classifications, yielding the Version Signature Deviation (VSD)
metric in [Formula 1]

\[
VSD = 1 - \frac{\# \text{Correctly classified snapshots}}{\# \text{Classified snapshots}}
\]  

(1)

In this formula, a “snapshot” corresponds to the metric values recorded
at a specific timestamp (i.e., one row in Table 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:

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

For our running example, Table 1 highlights in blue or red 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 $\Rightarrow$ airspeed=medium” rather than “throttle=high $\Rightarrow$ airspeed=high” and the deviating version snapshots also observe the rule “throttle=medium $\Rightarrow$ airspeed=low” rather than “throttle=medium $\Rightarrow$ airspeed=medium”. For the baseline version, there are 13 rows that matched the signature and 2 rows that did not match it, hence we obtained a VSD of $1 - \frac{13}{15} = 13\%$, which corresponds to the $\text{VSD}_{\text{Baseline Version}}$ variable for Table 1. We applied the same technique for the deviating version, which has 9 rows that matched the signature out of a total of 15, yielding a VSD of $1 - \frac{9}{15} = 40\%$. By using Formula 2, we obtain an RSD of $40\% - 13\% = 27\%$ for the airspeed output metric.

### 3.6. Detecting Behavioural Deviations

Finally, all that remains is to decide which RSD values are high enough to indicate a behavioural deviation instead of just measurement noise. Hence, our approach uses thresholds to flag behavioural deviations. 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 each output metric’s RSD value to its corresponding threshold. If the RSD of an output metric exceeds its threshold, this means that that simulator version shows behavioural deviations for at least that output metric.

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. Experimental Setup

This section explains the organization of the experiments that we performed to evaluate how well the approach presented in the previous section is able to identify behavioural deviations in a flight simulator. We provide all relevant information as required by the guidelines of Runeson et al. [43].

4.1. Objective

The objective of our experiments is to evaluate the degree to which our approach is able to identify behavioural deviations in real flight simulators. Furthermore, we aim to perform this evaluation at two different levels: system-level and component-level. System-level evaluation considers the behaviour of the complete simulator, observing the behaviour (i.e., metric data) of all its components, from cockpit, over simulation engine to specific subsystems like the hydraulic components. Component-level evaluation focuses on one specific component, considering only execution data from that component.

Through this system- and component-level evaluation, we aim to address the following research questions:

- RQ1: What is the performance of the approach for faults that change the flight environment?
- RQ2: What is the performance of the approach for functional faults?

4.2. Subject Systems and Use Case Studied

Since we did not have access to commercial flight simulator data, we searched for realistic open source flight simulators. One of the most popular ones is FlightGear, which we studied at the system-level as well as at component-level (focusing on one of its internal simulation engines, i.e., JSBSim). Here, we discuss both FlightGear (system-level) and JSBSim (component-level).

4.2.1. FlightGear

The FlightGear flight simulator [38] 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 default simulation engine and it is used both for
research purposes and as a stand-alone flight sim [45, 41, 40]. FlightGear
exposes the internal state of the SimGear simulation through a property
database that dynamically maps a value (such as speed or latitude) onto 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 maximum 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 con-
figuration, which uses a Boeing 700-300 aircraft and no extra environmental
conditions. We opted for this use case as landing (just like take-off) is one
of the most critical moments of a flight, 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.

4.2.2. JSBSim

JSBSim is an open source flight simulator engine conceived in 1996 as
a batch application that aims at modeling flight dynamics and control for
aircrafts. It has already been used as a simulation tool by major industry
organizations (e.g., NASA) and academia [7, 15, 16, 17] (e.g., University of
Naples Federico II, Italy, and RWTH Aachen University, Germany). An
end user can build her own simulation components without touching the
binary code (from a black box perspective) by editing XML files that describe
aircrafts, engines, flight scripts, etc. It is possible to store performance and
dynamics of a flight in data files (usually in CSV format) that can be used
for further analysis and studies. JSBSim is also integrated in other flight
simulators adding new features such as full 3D rendering of a flight and
various sound effects (e.g., FlightGear, Outerra[2] OpenEagles[3]).

JSBSim can be scripted to automatically execute flight runs using an

XML file that specifies navigation events where each event can be triggered on a certain condition (e.g., triggering an event if altitude reach 10,000 ft). When triggered, an event executes a series of defined actions that set properties of the simulator using a lookup table, a static value or a function. It also prints defined notifications giving feedback to the end user about performed actions. Properties set by an event can be any independent simulation property such as environmental properties (e.g., wind), flight dynamic coefficients, control interfaces (e.g., a switch), and so on. JSBSim properties can be set/read using a single interface via the Property Manager system allowing interaction with the system using various communication protocols (e.g., UDP), script files, specification files, and the command line. Properties are categorized using a tree-like structure similar to the Unix file system composed of a root node, sub-nodes (like directories) and end nodes (properties). Properties can be static or a function applying algebra on other static properties to generate an output value. We used JSBSim 0.1.2 in our experiments.

4.3. Experimental Method: Injecting Faults

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. [28] 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.

Since no databases exist with execution data of the two subject systems, and hence no oracles, we instead inject faults into the systems for which we know which metrics should show a behavioural deviation [4]. We then use our approach to compare the version with fault (deviating version) to the version without fault (baseline version). Given that FlightGear is a full flight simulator system, while JSBSim only is a simulation engine (e.g., no graphical user interface), we injected different faults in each. This section explains the faults injected.

4.3.1. FlightGear

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
Table 2: Injected Faults for FlightGear.

<table>
<thead>
<tr>
<th>Fault Name</th>
<th>Type</th>
<th>Description</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak Wind</td>
<td>Weather</td>
<td>Wind at 5 knots 25 deg.</td>
<td>Normal</td>
</tr>
<tr>
<td>Strong Wind</td>
<td>Weather</td>
<td>Wind at 25 knots 25 deg.</td>
<td>Deviations</td>
</tr>
<tr>
<td>Flaps Fault</td>
<td>Functional</td>
<td>Flaps disabled</td>
<td>Deviations</td>
</tr>
<tr>
<td>Auto Brake Fault</td>
<td>Functional</td>
<td>Auto brake disabled</td>
<td>Deviations</td>
</tr>
<tr>
<td>Right Engine Fault</td>
<td>Functional</td>
<td>Failure of right engine</td>
<td>Deviations</td>
</tr>
</tbody>
</table>

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 realistic, but 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. As such, for our experiments, we can make changes either 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), obtaining a version of the simulator for which we expect clearly different behaviour than the baseline version. This more liberal definition of “fault” allows us to experiment not only with incorrect behaviour, but with different behaviour in general. The resulting faults in our experiments were obtained through discussions with experts from a commercial flight simulator company, with which the authors are collaborating in the context of the FRQNT ACACIA project.

Table 2 summarizes the five faults that we injected into FlightGear.

Note that although some of these faults seem easy to spot manually on small data sets if one knows what behavioural deviations 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. We now discuss these five faults.
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 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. 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 [46]. 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) [5]. We injected an Auto Brake fault in which the auto brake does not work during the landing phase.

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 [10]. We injected a fault where the right engine does not work during the landing phase.

4.3.2. JSBSim

Given that JSBSim is a flight simulator component instead of a full simulator system, we could not inject environment-related faults (RQ1). Instead, we only injected faults in the aircraft model, which models the aircraft’s behaviour. Hence, we developed different mutated versions of the c310.xml aircraft model (\(<\text{JSBSim root}>/\text{aircraft/c310/c310.xml}\) on


\[5\]http://www.fss.aero/accident-reports/look.php?reports_key=488
which we performed multiple executions of the flight scenario encoded in c3104.xml (<JSBSim root>/script/c3104.xml) to generate flight data. The c310.xml aircraft model represents the Cessna C310, which is a small recre-ative/military aircraft developed by Cessna in the 1960’s (see the specifications in Table 3).

Table 3: C310 specifications (short form)

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. Cylinders</td>
<td>6</td>
</tr>
<tr>
<td>HP</td>
<td>260</td>
</tr>
<tr>
<td>Fuel Capacity</td>
<td>102 gallons</td>
</tr>
<tr>
<td>Max Takeoff Weight</td>
<td>4,830 lbs.</td>
</tr>
<tr>
<td>Max Landing Weight</td>
<td>4,600 lbs.</td>
</tr>
<tr>
<td>Standard Empty Weight</td>
<td>3,125 lbs.</td>
</tr>
<tr>
<td>Max. Useful Load</td>
<td>1,750 lbs.</td>
</tr>
</tbody>
</table>

The c310.xml flight script specifies a small round-trip scenario where the c310 plane takes off from Ellington airport in Texas, cruises around until reaching a given waypoint, then lands again on Ellington airport.

To inject the faults in the Cessna aircraft model, we used the help of the third author, who is one of the original developers of JSBSim [15]. This resulted in three different faults, which we applied individually and all together, yielding four different fault configurations. We discuss the three individual faults below.

**Inertia Fault.** The moment of inertia measures the inertia of a rigid body when its rotation speed changes and determines the torque needed for a desired angular acceleration about a certain rotational axis. It depends on the body’s mass distribution and the axis chosen, with larger moments requiring more torque to change the body’s rotation rate[6]. For an aircraft, considering the pitch axis for instance, the larger the moment of inertia with respect to the pitching axis $y$, usually named $I_{yy}$, the stronger the inertial couple counteracting pitch accelerations $\dot{q}$. We injected a typical uncertainty of the aircraft moments of inertia, e.g. $I_{yy}$ and also added a nonzero $I_{xz}$ (product of

inertia). The latter is often set to zero when a mass layout is not accurately known, assuming that the aircraft reference axes are central axes of inertia.

*Lift Fault.* The lift coefficient $C_L$ is a dimensionless coefficient that relates the lift generated by a lifting body to the fluid density around the body, the fluid velocity and an associated reference area[7]. We modified the lift coefficient dependency on angle of attack (AoA), adding an offset to the function “aero/coefficient/CL0”. This can be seen as an uncertainty on the aerodynamic lift coefficient at zero AoA.

*Pitch Fault.* An aircraft develops an aerodynamic pitching moment with respect to the center of gravity[8]. The dependency of the aerodynamic pitching moment on the AoA, and consequently on the lift coefficient, is of great importance for the vehicle’s rotational equilibrium (ability to be “trimmed” in pitch) and stability (ability to return to equilibrium flight conditions after external perturbations). We have modified the pitching moment coefficient curve versus AoA. The original dependence is linear in AoA, given by the sum:

$$C_m = C_{m0} + C_{ma} \alpha$$

This is a typical linear expression where $\alpha$ is the AoA, and $C_{ma}$ is called “longitudinal static stability”. The above function is acceptable when

$$C_L \approx C_{Lo} + C_{Lalpha} \alpha$$

where $C_{Lo}$ is known as the “lift curve slope”. The linear dependency of $C_L$ on $\alpha$ is accurate for angles of attack smaller than the so called “alpha-star”, i.e. a certain value $\alpha^*$ for a given aircraft. For $\alpha > \alpha^*$ the $C_L$ varies nonlinearly and reaches a maximum at the so called “$\alpha$ of stall” ($\alpha_{stall} > \alpha^*$). This behaviour is fairly well modelled in JSBSim in terms of lift but not in terms of pitching moment. We have introduced a nonlinear variation of $C_m$ that typically occurs for $\alpha > \alpha^*$ and matches the nonlinear behaviour of $C_L$.

4.4. Experimental Method

This section explains how we implemented each step of our approach for our experiments. The next section then explains how we determined the
RSD threshold to determine whether or not a given signature deviation is normal.

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. We considered 25 input metrics and 26 output metrics for FlightGear (a summary is presented in Table 4).

Table 4: FlightGear Output metrics summary.

<table>
<thead>
<tr>
<th>output</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Position</td>
</tr>
<tr>
<td>4</td>
<td>Orientation</td>
</tr>
<tr>
<td>8</td>
<td>Velocity</td>
</tr>
<tr>
<td>6</td>
<td>Acceleration</td>
</tr>
<tr>
<td>5</td>
<td>Surface Position</td>
</tr>
</tbody>
</table>

We selected these metrics 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 and aircraft control switches. 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 use case at hand, using FlightGear’s or JSBSim’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. We used the Weka data mining tool [26] to generate our signatures and R scripts [42] for the analysis of deviations.

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 be flagged by our approach. Since we
injected the faults ourselves, with the help from experts of a commercial
flight simulator company as well as the third author, we could obtain the
FlightGear and JSBSim oracles of Table 5 and Table 6, respectively.

Table 5: FlightGear oracles for each fault.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Deviating Flight Metric Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak Wind</td>
<td>Nothing</td>
</tr>
<tr>
<td>Strong Wind</td>
<td>Velocity and Position</td>
</tr>
<tr>
<td>Flaps Fault</td>
<td>Velocity and Orientation</td>
</tr>
<tr>
<td>Auto Brake</td>
<td>Acceleration, Position, Orientation and Velocity</td>
</tr>
<tr>
<td>Right Engine</td>
<td>Acceleration, Velocity and Orientation</td>
</tr>
</tbody>
</table>

Table 6: JSBSim oracle for metric deviations in the output metrics for each injected fault, where “X” means that there is a deviation. The category to which each output metric belongs is shown in the Category column, where “P” means Position, “O” Orientation, “V” Velocity, and “A” Acceleration.

<table>
<thead>
<tr>
<th>Output Metric</th>
<th>Total</th>
<th>Inertia</th>
<th>Lift</th>
<th>Pitch</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdm/jsbsim/position/h-sl-ft</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>P</td>
</tr>
<tr>
<td>fdm/jsbsim/position/lat-geod-deg</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>P</td>
</tr>
<tr>
<td>fdm/jsbsim/altitude/pitch-rad</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>fdm/jsbsim/altitude/roll-rad</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>fdm/jsbsim/attitude/theta-rad</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>fdm/jsbsim/velocities/u-fps</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>V</td>
</tr>
<tr>
<td>fdm/jsbsim/velocities/v-fps</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>V</td>
</tr>
<tr>
<td>fdm/jsbsim/velocities/w-fps</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>fdm/jsbsim/velocities/p-dot-rad-sec</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>fdm/jsbsim/velocities/q-dot-rad-sec</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>fdm/jsbsim/velocities/r-dot-rad-sec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>fdm/jsbsim/accelerations/p-dot-rad-sec2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>fdm/jsbsim/accelerations/q-dot-rad-sec2</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>fdm/jsbsim/accelerations/r-dot-rad-sec2</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>fdm/jsbsim/accelerations/v-dot-ft-sec2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>A</td>
</tr>
</tbody>
</table>
For each output metric, we also mentioned the metric's general category, since many metrics are measuring related concepts and experts tended to think in terms of metric category deviations rather than individual output metric deviations. Based on these oracle, we then calculate precision, recall and false alarm rate of our approach in terms of metric categories (not individual metrics). **Precision** corresponds to the percentage of correctly flagged deviations (i.e., absence of false alarms), as shown in [Formula 3]. **Recall** is the percentage of all deviations in the oracle that the approach is able to identify, cf. [Formula 4]. Finally, for faulty versions without deviations (like the Weak Wind fault), we use the False Alarm Rate in [Formula 5] to measure how well the approach avoids false alarms.

Precision = \[
\frac{\text{# Correctly identified deviation categories}}{\text{#Identified deviation categories}}
\] (3)

Recall = \[
\frac{\text{# Correctly identified deviation categories}}{\text{#Deviation categories}}
\] (4)

False Alarm Rate = \[
\frac{\text{#Incorrectly identified deviation categories}}{\text{#Non-deviation categories}}
\] (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 category.

4.5. **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 6 illustrates in more detail how one can derive the optimal threshold for FlightGear, while this subsection explains how we determine the initial threshold for FlightGear and JSBSim.

The initial threshold should be 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 by the operating system.

To perform the calibration of the initial threshold, we executed different baseline runs of FlightGear and JSBSim in which we gradually add extra weight or light amounts of wind, both of which staying within the aircraft’s tolerances, we then determine a threshold for RSD \[ \text{Formula 2} \] that is able to tolerate there non-harmful differences by not flagging false deviations. We played with two ideas to determine the threshold:

1. using the lowest threshold that avoids flagging false deviations (via the false alarm rate of \[ \text{Formula 3} \])
2. using the median RSD across the baseline runs

We found that for FlightGear, the first approach worked well (in terms of precision/recall for RQ1/RQ2), while for JSBSim, the second (simpler) approach worked well (RQ2).

For FlightGear, Fig. 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 \[ \text{Formula 3} \] whose results are shown in Table 7.

There are 2 distinct clusters of flight metrics, which means that some output metrics contain much more noise than others. 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).
Table 7: False alarm rates for threshold calibration and RQ1 (FlightGear).

<table>
<thead>
<tr>
<th>Fault</th>
<th>Initial False Alarm Rate</th>
<th>Optimal False Alarm Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline Flight</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Weak Wind</td>
<td>60%</td>
<td>0%</td>
</tr>
</tbody>
</table>

If we pick initial thresholds of 15% and 40% respectively for clusters 1 and 2 of FlightGear, 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. 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).

For JSBSim, we use the median value of each output metric in the baseline runs as threshold. For space purposes, we did not include the baseline runs in Fig. 4. Further research is necessary to determine why the median approach worked better as initial threshold for JSBSim, compared to FlightGear.

5. Results

This section presents the answers to the research questions based on the results of our experiments with FlightGear (RQ1/RQ2) and JSBSim (RQ2).

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

Motivation

Different weather circumstances can render a flight very different from the original flight. Although technically not a fault, the behaviour is expected to be substantially different in case of strong wind from a baseline run, hence in the context of our experiments we consider strong wind to be a fault. On the other hand, the introduction of a weak wind only slightly modifies the
Figure 3: Flight metric RSDs for FlightGear 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).
Figure 4: Flight metric RSDs for JSBSim 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).
baseline flight, and should not be identified as different from the baseline behaviour. In other words, the weak wind case is a sanity check for our approach, to check false positives.

**Approach**

We collected data for 3 runs of the Weak Wind FlightGear version and 3 runs of the Strong Wind FlightGear version (see the corresponding RSD values on [Fig. 3](#)). Using the initial and optimal thresholds, we obtain the precision, recall, and false alarm rate values of Table 7 (last row) and Table 8 (first row). Note that we only use FlightGear in this RQ, since it represents a full flight simulator system, whereas JSBSim is only one component and hence environment faults do not apply.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Initial Recall</th>
<th>Initial Precision</th>
<th>Optimal Recall</th>
<th>Optimal Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Wind</td>
<td>100%</td>
<td>50%</td>
<td>100%</td>
<td>50%</td>
</tr>
<tr>
<td>Flaps Fault</td>
<td>100%</td>
<td>40%</td>
<td>100%</td>
<td>67%</td>
</tr>
<tr>
<td>Auto Brake Fault</td>
<td>50%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Right Engine Fault</td>
<td>100%</td>
<td>75%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Inertia Fault</td>
<td>100%</td>
<td>44%</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Lift Fault</td>
<td>75%</td>
<td>25%</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Pitch Fault</td>
<td>24%</td>
<td>13%</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Total Fault</td>
<td>75%</td>
<td>25%</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Results**

Some FlightGear 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 [subsection 4.5](#). 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 correctly flagged metrics predicted by our oracle before incorrectly flagged metrics start to disappear (RSD below increasing threshold). 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 flagging all deviations for the Strong Wind version is not possible without having 50% false alarms.

**RQ2: What is the performance of the approach 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 injected faults are the most important ones 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. 3), as well as 5 of the 4 faulty JSBSim versions (Fig. 4). We evaluated our approach using the same methodology as RQ1. Table 8 shows the performance results for each of the configurations, for both thresholds of FlightGear and the initial threshold of JSBSim.

**Results**

In both flight simulators, most of the faults have deviations across all metric categories. For FlightGear, both the Flaps and Right
Engine 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 Fault. Auto Brake Fault is the fault with the least deviation compared to the baseline run, having low RSD values not higher than 25%.

For JSBSim, the Total and Lift faults have the largest RSD values, spread across all metric categories, although the deviations for different metrics are widely varying. Some metrics have a very low RSD value, which is still flagged as deviation, while others have higher RSD values that are not being flagged. This is due to our choice for the median of each baseline metric as threshold for deviations.

**Except for Pitch fault, we obtain recall values of at least 75%, with four faults not missing any deviation (100% recall).** For FlightGear, the Auto Brake Fault has a recall lower than 100% using the initial thresholds. Fig. 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 injected fault are close to the thresholds, the deviations were narrowly missed.

For JSBSim, the Pitch fault has a recall of only 24%. This is due to the low RSDs obtained for this fault, i.e., the proportion of deviating snapshots is too similar to that of the baseline runs, causing the RSD to fall below the threshold. The better the JSBSim thresholds are calibrated, the less sensitive the approach to fluctuations in the baseline runs and hence the higher the corresponding recall for Pitch fault will be. In other words, given its sensitivity, we could use the Pitch fault data to better calibrate our thresholds.

**Only three faults obtain a precision of at least 67%, while the remaining faults obtain a precision between 13% and 50%**

In the case of FlightGear, the Flaps fault has a low precision of 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.

In JSBSim, on the other hand, the best precision is 44% for the Inertia Fault, while Pitch fault obtains an abysmal precision of only 13%. The same explanation holds as for the low recall value for Pitch fault, i.e., better
calibration could improve this sensitive fault’s results. Although random guessing would obtain a better precision (50%), the high recall values for all but Pitch fault makes our approach still worth the additional work of checking false positives.

The optimal threshold performs markedly better in terms of precision and/or recall than the initial threshold for the Flight-Gear faults. This suggests that similar improvements could resolve the low precision values for JSBSim. Future research should analyze this in detail.

We are able to find all injected deviations in 4 out of 7 faults, and 75% of the deviations in 2 other faults, yet especially for JSBSim the precision using the initial thresholds is low (i.e., less than 44%).

6. Discussion

6.1. Optimal thresholds

In our experiments, we have guided our initial threshold choice based on the characteristics of different runs of the baseline flights done with different weights or light wind within the aircraft’s tolerances. We either tried to minimize the false alarm rate (FlightGear) or to use the median RSD values of the baseline metrics (JSBSim). 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). We performed the analysis for FlightGear, but a similar approach can be followed for JSBSim.

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), then plot the recall and precision for each threshold in order to find a sweet spot where both values are high (Fig. 6).

For the versions without deviations (Baseline Flight and Weak Wind), we plot the false alarm rate instead, which we aim to keep low (Fig. 5).

For instance, for the Strong Wind version (Fig. 6), 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. 5 and Fig. 6 are the optimal thresholds used in section 5.
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. 6). 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. 5 and Fig. 6) 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. 6) requires a slightly higher threshold (5%) than the initial one to have perfect precision/recall.

![Graphs](https://example.com/graphs.png)

(a) Baseline Flight  
(b) Weak Wind

Figure 5: False alarm rate across multiple thresholds. The black lines represent the optimal thresholds.

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
Figure 6: 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.

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).
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, except for precision in the case of JSBSim. Experiments with more metrics and versions are needed to further refine our findings.

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 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.

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 sec-
onds rather than a few minutes represents a danger for passengers. Hence,
in future work we will adapt our approach to also support transients, which
requires substantial changes to our approach.

6.4. Fault Injection vs Bugs

Because we did not have access to versions of FlightGear and JSBSim
with known behavioural deviations, we artificially injected faults by changing
the flight environment or flight behaviour of an aircraft model for a simulator
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.

7. Threats to Validity

Construct validity threats concern the relation between theory and obser-
vation. 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 20 runs. We plan to automatically
generate runs using an automated pilot system for FlightGear and JSBSim.

Threats to external validity are related to the selection of one specific
version of the FlightGear and JSBSim open source flight simulators for our
study. Although both simulators are well-known and are widely used for
research purposes, they may not be representative of the whole flight simu-
lator’s domain. Further studies on commercial flight simulators are needed
to test the generality of our approach.

8. Conclusion

In this paper, we explored a signature-based data mining approach to
flag behavioural deviations at two different levels, system and component, of
safety-critical systems such as a flight simulator. We evaluated the approach
on FlightGear and JSBSim open source flight simulators introducing faults
related to both flight environment and flight behaviour. Results show that
while we did not miss any deviating behaviour (maximum recall) related to
flight environment, human analyst would still consider 50% false alarms (low
precision). Considering functional deviating behaviour, we were able to find
all injected deviations in 4 out of 7 faults and 75% of the deviations in 2
other faults. This is promising, since it enables organizations who want to
adopt this approach to obtain a decent performance out-of-the-box.

References

International Civil Aviation Organization, Quebec, Canada, 2009.


appropriate tool for testing experiments?[software testing]. In Software
Engineering, 2005. ICSE 2005. Proceedings. 27th International Confer-


[14] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving


[23] King Foo, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Ying Zou, and Parminder 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), page to

of an in-cockpit cueing system for upset recovery. In AIAA Guidance,

Haslbeck. How pilots assess their non-technical performance—a flight
simulator study. Advances in Human Aspects of Transportation: Part

[26] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter
Reutemann, and Ian H Witten. The weka data mining software: an

[27] Ahmed E Hassan and Richard C Holt. Predicting change propagation

[28] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy.
The art of testing less without sacrificing quality. In International Con-

[29] Victoria J Hodge and Jim Austin. A survey of outlier detection method-

[30] Salman Hoseini, Abdelwahab Hamou-Lhadj, Patrick Desrosiers, and
Martin Tapp. Software feature location in practice: debugging aircraft

[31] Stephen A Jacklin, Michael R Lowry, Johann M Schumann, Pramod P
Gupta, John T Bosworth, Eddie Zavala, John W Kelly, Kelly J Hay-
hurst, Celeste M Belcastro, and Christine Belcastro. Verification, val-
idation, 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,


