Fault Injection, Detection and Treatment in Simulated Autonomous Vehicles

Daniel Garrido(✉), Leonardo Ferreira, João Jacob, and Daniel Castro Silva

Faculty of Engineering of the University of Porto, Portugal Artificial Intelligence and Computer Science Laboratory (LIACC), Rua Dr. Roberto Frias s/n, 4200-465 Porto, Portugal
{up201403060,up201305980,joajac,dcs}@fe.up.pt

Abstract. In the last few years autonomous vehicles have been on the rise. This increase in popularity lead by new technology advancements and availability to the regular consumer has put them in a position where safety must now be a top priority. With the objective of increasing the reliability and safety of these vehicles, fault detection and treatment modules for autonomous vehicles were developed for an existing multi-agent platform that coordinates them to perform high-level missions. Additionally, a fault injection tool was also developed to facilitate the study of said modules alongside a fault categorization system to help the treatment module select the best course of action. The results obtained show the potential of the developed work, with it being able to detect all the injected faults during the tests in a small enough time frame to be able to adequately treat these faults.

Keywords: Autonomous vehicles · Unmanned aerial vehicles · Fault injection · Fault detection · Fault treatment · Simulation · Safety

1 Introduction

Autonomous vehicles (AVs) have received a lot of attention in the last years thanks to their ability to perform tasks in places humans can't reach or are too dangerous [12]. This increase in popularity drives the need to guarantee that these systems are safe to operate both for operators and surrounding population. To assure safety of operation, AVs must be resilient to failures that create dangerous situations. Since an AV can't rely on the judgement of a human, it must detect and handle faults internally. The simplest way to achieve this is through redundant systems that compare each other's outputs and can take over in case of a failure. However, this approach's disadvantages are exacerbated in small AVs as they can't always accommodate the additional weight and space. The alternative is to analyse the data generated from the vehicle's sensors to detect fault-related patterns and alter its behaviour to handle the fault [2].

Because research with real vehicles can be cumbersome and expensive, the solution to this problem is going to be developed inside a simulation platform capable of coordinating AVs to perform high-level missions, which uses FSX (Flight Simulator

V. V. Krzhizhanovskaya et al. (Eds.): ICCS 2020, LNCS 12137, pp. 471–485, 2020.
https://doi.org/10.1007/978-3-030-50371-0_35

X) as the simulation engine [13]. This research is a continuation of the development of this platform as it currently does not have a fault handling system, which is crucial when dealing with this kind of vehicles. While the platform and the concept of the project can be applied to any AV, it was primarily developed and tested for large fixed-wing UAVs (Unnamed Aerial Vehicles).

The goal of this project is to develop and incorporate a fault diagnosis system to the platform. This system must be easy to use and cover the most common failures in UAVs. In the end, the vehicle should be able to detect and correct fault scenarios on its own, while minimizing computational resources overhead.

To achieve this objective, several new modules were built to integrate in the existing platform. The first is a fault injection tool that allows the user to control fault injections during missions. Then, two modules were added to the vehicle agent: one for fault detection and the other for treatment. In the end, tests to these modules were conducted to assess fault detection rates and times, as well as the quality of the treatment and computational impact on the platform.

The rest of this article is structured as follows. Section 2 quickly reviews the state of the art and previous related work. Section 3 details the implementation process, starting with fault-related tests made to FSX, and the fault injection, detection and treatment modules. In Sect. 4, a description of the performed experiments is presented alongside the results, with their discussion presented in Sect. 5. Finally, Sect. 6 concludes the article and elaborates on future work.

2 State of the Art

In this section a literature review is presented in two parts. First a more general view on fault detection methods is given, before exploring some related work where these methods are applied to AVs.

2.1 Fault Detection Methods

There is a large amount of relevant literature on fault detection, which has been a serious research topic at least since the 1970s. Throughout the years, several surveys have been published which detail the advancements in fault diagnosis.

Usually, these surveys divide fault detection methods in categories to simplify their classification. Different authors propose different but similar classifications. The simplest one was proposed by Gertler, with methods divided in those that make use of a model and those that don't [6]. Miljković used three groups: data methods and signal models; process model-based methods; and knowledge-based methods [10]. The first two groups are identical to Gertler's, with a new group for the recently developed machine learning methods. Isermann's classification is the most complete and detailed, with several groups that relate to each other [9]. The studied classification methods were labeled using Gertler's approach.

Table 1. Summary of reviewed Fault Detection Methods

	Complexity	Computational cost
Model-free methods		
Limit/trend checking	Very low	Very low
Change detection	Low/Medium	Low
Neural networks/clustering	Medium/High	Medium
Model-based methods		
Parity equations	High	Medium/High
Parameter estimation	Very high	Very High
State observer	High	High
Output observer	High	High

Model-free methods, also called data-driven methods, use the input and output data from the system under diagnosis to search for fault patterns. These are usually less accurate than model-based methods but use less computational resources as they don't need to make model-related calculations. On the other hand, model-based methods use a model of the system in conjunction with a combination of inputs and outputs, depending on the method, resulting in more accurate detection, but with a computing performance penalty [8].

For more information on the other studied methods, refer to the previous work [5]. A collection and comparison of these methods regarding group, complexity and computational cost can be seen in Table 1.

Since the developed work focused on creating the whole system, the simplest method was used for fault detection, the limit and trend check methods. These are similar methods that monitor the values of specific variables while comparing them to predetermined upper and lowers bounds. When that variable is out of these bounds, a fault trigger can be activated. In limit checking, only the current value is taken into consideration, while in trend checking the rate of change of said variable is used [9].

2.2 Fault Detection in Autonomous Vehicles

In existing literature detailing the implementation of fault detection methods in autonomous vehicles, these can be either real or simulated, with some using both. In this literature review only those that study UAVs and present significant results are discussed.

Cork et al. applied the data collected from nominal flights to train Neural Networks to predict the output of a specific sensor and compare it with the measured values [3]. When a high difference between the two was detected, the system knew something was not right. For a data-driven system it obtained good results and could even train while being used.

Table 2. Summary of most relevant fault detection literature in AVs

Work	System	FD method	Results
[3]	Angular rate sensors	Neural networks	• Avg. detection rate: 84% • False-positives rate: 10% • Avg. detection time: 36 s
[7]	Positioning Sensors	Model based observer	• Avg. detection time: 0.55 s • #False-negatives: 6 •#False-positives: 9
[4]	Aileron actuators	Model-based observer and change detection with Z-test	• Model Based TIC avg.: 0.143 • Change Detection avg. detection time: 0.8 s
[11]	Pitot-static systems	Clustering (K-means and EM)	• Detection rate: 96% • False-positive rate: 1.5% • Detection time: "Almost Instant"

While not as popular as fixed wing UAVs, single rotor UAVs also exist. One of this kind of UAVs was used as a platform to create a model-based observer system to detect faults in positioning sensors. This work concluded that detection was possible but was more difficult in the case of additive and multiplicative faults, when compared to faults that made the sensors reading freeze [7].

Freeman et al. monitored the aileron actuators of a light UAV by two distinct approaches: change detection (data-driven) and observers (model-based) [4]. Both systems were tested with real flight data. It was found that the model-based approach was better at detecting faults, but it was also noted that the process of modelling the UAV was time-consuming. Meanwhile, the data-driven method was easier to implement and could also detect most of the faults.

As for a fault detection system that utilizes a game/simulation engine like the one used in this project, only one such case was found. Purvis et al. used the open-source flight simulator FlightGear to create a system that could inject, detect and treat faults related to the pitot-static system of a simulated commercial airliner. Their solution used clustering methods to label the flight data as faulty and not faulty, with very good results [11].

Table 2 summarizes the results of this small literature review, showing for each work the type of faulty system, fault detection method and experimental results. As expected, model-based methods worked better than data-driven ones, with Clustering being the better method when no model is used.

3 Implementation

The implementation process was divided in three parts. First a classification system for UAV faults was created. Next, a fault injection system was implemented in the multi-agent platform; and lastly, the agent responsible for controlling the vehicles was extended to include both a fault detection and treatment modules.

3.1 Fault Classification System

A classification system for UAV faults was created to categorize faults by severity according to the affected system and extent of the fault, while also providing recommended actions that the UAV should take in case of a fault. This system will prove helpful when there is a need to assess the impact of a detected fault to the UAV and what actions it can take to handle the situation. Table 3 presents a summary of the system and Table 4 explains the severity scale used [1]. The same failures were divided in several entries to accommodate different extents that progressively increase in severity. The failure influence on the aircraft was also included to help classify faults that are not included but cause similar problems.

Table 3. UAV Fault Classification Table

Failure	Influence	Severity	Reaction
Engine (partial)	Reduced lift and speed	Medium	Return to airport and emergency landing
Engine (complete)	Complete loss of lift	High/Extreme	Emergency landing/crash where possible
Communications	Loss of comms with ATC and potential flyaway	Medium	Return to airport and emergency landing with visual indication of communications fault
Control surfaces (single, free float)	Extra effort and care in controlling aircraft required	Medium	Return to airport and emergency landing
Control surfaces (single, stuck)	Difficulty in controlling aircraft	High	Return to airport and emergency landing
Control surfaces (multiple)	Total loss of control	Extreme	Imminent crash
Sensors (single)	None, remaining sensors should be able to compensate faulty one	Low	Procced mission
Sensors (multiple)	Loss of spatial awareness	Medium/High	Return to airport if possible, emergency landing/crash where possible otherwise
Sensors (complete)	Complete loss of spatial awareness	Extreme	Imminent crash
Electrical	Complete loss of sensors, control surfaces and electrical propulsion	Extreme	Imminent crash
Landing gear	Harsh landing	Low	Nothing
Brakes	Prolonged landing distance	Low	Abort landing and retry in longest runway, using all of the runway

3.2 Injecting Faults in Flight Simulator X

Working with FSX as a simulation platform is facilitated by using the integrated SimConnect SDK[1] which allows an external program to read and modify simulation variables through a client-server interface. Additionally, FSX includes fault injection in aircraft natively, but when the development began it was found that this was mainly supported for the player aircraft, and support for the AI-controlled aircraft that the platform uses was very limited. In spite of these limitations, some faults were able to be reliably injected in the platform's vehicles, including engines, brakes and communications.

Table 4. Fault severity scale

Severity	Flight control impact
Low	No or very subtle alterations in control; could easily reach landing site and have no problems touching down in the designated area
Medium	Significant alterations in control; can reach landing site but might have difficulty landing in the designated area
High	Very compromised control; difficulty in reaching landing site
Extreme	Very limited or no control at all

Before a fault can be injected, it first must be described. The user can create several faults that can affect any number of aircraft at any given time or during a number of special conditions. The fault itself is defined by a number of variables that determine when it should be triggered, when it ends, how strong the fault effect should be and what behaviour it should follow. Each fault contains a list of vehicles it can affect and a list of faults that can be injected to these vehicles. Different vehicles can be injected with different faults. The user can also define to great detail what conditions will trigger the fault, which can be based on the aircraft speed, altitude or location, elapsed time, weather conditions, ground surface type, etc. The value of the fault determines how severe the impact of the fault is or, in the case of control surfaces, the position at which they should be kept for the duration of the fault. The user can also choose the time behaviour that governs the fault injection, which can be set to permanent, intermittent, transient or noise. To simulate drift-like faults a ramping variable was also added that specifies how much time the fault should take to reach the desired strength. To facilitate the creation and modification of faults to be injected in a mission, a graphical interface was created to intuitively and quickly allow a user to specify changes. Figure 1 shows an example of this interface during use.

Engine faults can be injected to individual engines or to all engines. Due to the limitations of FSX, only the "all engines" fault can make use of the strength

[1] More information available online at https://docs.microsoft.com/en-us/previous-versions/microsoft-esp/cc526983(v=msdn.10).

value, with the single engine faults being restricted to being toggled, setting the engine on or off. Brakes fault is another toggle-type fault that affects the aircraft when it is trying to slow down after landing. The communications fault was handled entirely through the platform messaging system and effectively blocks all messages from reaching or leaving the affected vehicle.

3.3 Fault Detection and Treatment

Since no aircraft model was accessible from FSX, model-based methods could not be used. Instead, data-driven methods were used to detect faults in the three systems mentioned above. Due to FSX limitations on AI-controlled aircraft, not much data was available to use in the detectors, which limited the available methods to the simpler ones that don't require much data to be effective.

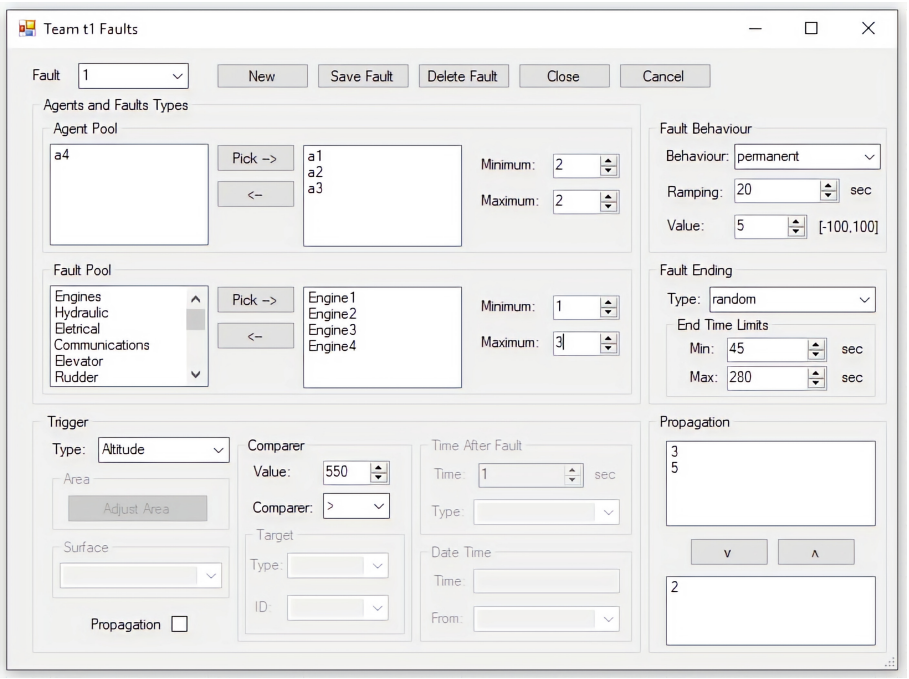

Fig. 1. Fault configuration window

The engine fault detector uses a combination of limit and trend checkers on the available engine variable: the propeller speed. The trend checker constantly analyses the propeller speed rate of change and triggers when this value is higher than a predefined value. For situations when the engine thrust descended slowly over a long period of time, also called ramping, a limit checker was also implemented that simply verifies if the engine RPM is too low (300 RPM in this case).

These two methods only trigger if the aircraft's current altitude is lower than the desired one, to prevent falsely detecting a fault when the aircraft is descending.

Faults related to brakes are detected with another trend checker. When the aircraft touches down to land, it immediately starts analysing the rate at which the aircraft slows. If this rate stays low for too long (above $-2\,\mathrm{m/s^2}$ for over 5 s in this case), a brake failure is detected.

The communications fault detector uses a very simple method to verify if the communications are working. Every 10 s the vehicle pings the closest ATC (Air Traffic Controller), who replies with an acknowledge. If the vehicle doesn't receive a response after 10 s of sending the ping, it knows the communications are not working properly. This means that in a best case scenario a fault can be detected in just 10 s, but in the worst it will take up to 20 s. The waiting time between messages could be reduced, but this could present problems when an ATC is responsible for several aircraft and can't handle all messages in a timely fashion.

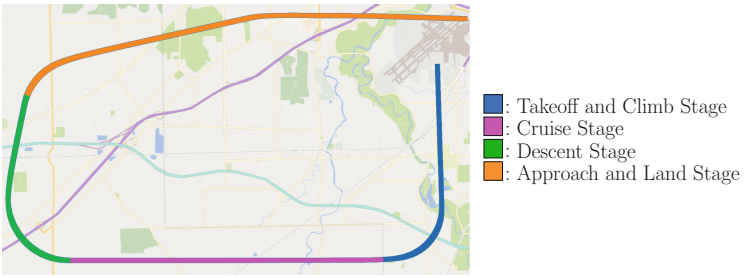

Fig. 2. Test flight scenario (note the airport on the top-right corner)

Once a fault is detected, the fault treatment module gives it a classification and follows the recommended action. In cases where several faults have been detected it will perform the action associated with the fault with the highest severity. In extreme cases, such as full engine failures, this module will track the aircraft return course to the airport and deduce if the aircraft has enough altitude to reach it. If this is not the case, a new landing site that the aircraft knows not to be populated is chosen to prevent crashing into a building or humans.

4 Experimental Setup and Results

This section is organized in two main parts: first, an explanation of the tests is given, followed by the presentation and analysis of the results.

4.1 Test Configuration and Scenarios

The tests to the developed work were conducted in the proximity of an airport previously modelled in detail in the platform. It was chosen because it has an

interesting layout of two long and one short runway. The model of the aircraft used in the simulation was the Beechcraft Baron 58. It was picked for its relatively small size and engine configuration as it is the smallest and lightest aircraft with a twin prop engine. The small size makes it comparable to the bigger UAVs like the United States Air Force Predator, in terms of wingspan and weight, while the dual engine configuration allows for more flexibility when testing.

For every test the aircraft was given a simple mission to perform, as seen in Fig. 2, which includes taking off, making a right bank turn while ascending, holding altitude for a few miles, performing another right bank turn while now descending, approaching the smallest runway at the airport and finally landing. The different colours represent the different flight phases. The tests were all conducted with FSX running at a simulation rate of 4x to reduce test times.

The tests were separated in two phases: in the first phase only the fault detectors are active and in the second phase both the fault detection and treatment modules are operational. This way a benchmark of the outcomes of the faults can first be recorded to then compare to the outcomes when the same test is run with the fault treatment module enabled. Table 5 shows a summary of the test with all settings used.

Test #0 is a control test, with no faults active. It serves as a baseline to compare to the behaviour of the actual tests when faults are injected. Since tests #1 and #2 are not dependant of the flight stage, ramping and fault value, one test is enough to test if the module can correctly detect these faults. Test #2 is run with intermittent time behaviour to effectively allow the test to run several times to make sure the detection times don't surpass the theoretical maximum of 20 s. This is the only scenario where having an intermittent fault type is advantageous, as this type of time behaviour uses random injection times which are not ideal when the behaviour of the plane is being tested. This can result in the fault being injected for too little time to be detected or even have a meaningful effect on the aircraft.

Table 5. Tests to be performed to the fault detection module.

Test	Fault	Fault value	Stage	Ramping	Time behaviour	Duration
#0	–	–	–	–	–	–
#1	Total Brakes	–	–	–	Permanent	Unspecified
#2	Communications	–	Cruise	–	Intermittent	180 s
#3	Engine 1	–	Cruise	–	Permanent	Unspecified
#4	Engine 1	–	Climb	–	Permanent	Unspecified
#5	Engine 1	–	Descent	–	Permanent	Unspecified
#6	Engines	0	Cruise	–	Permanent	Unspecified
#7	Engines	0	Climb	–	Permanent	Unspecified
#8	Engines	0	Descent	–	Permanent	Unspecified
#9, #10	Engines	0	Cruise	30, 60	Permanent	Unspecified
#11, #12	Engines	0	Climb	30, 60	Permanent	Unspecified
#13, #14	Engines	0	Descent	30, 60	Permanent	Unspecified

Tests #3, #4 and #5 cover single engine full failure in all 3 flight phases, while tests #6, #7 and #8 do the same but with 2 failing engines. Finally, tests #9 to #14 test the effects of different ramping values in the different flight stages. This effect will only be tested with engine faults since this is the only one that supports continuous analog injection.

Finally, the fault treatment module is enabled, and tests #1, #2, #4 and #6 are ran again to test the ability to treat the faults in the expected way and comparing the outcome of the tests with the previous non-treated tests.

4.2 Fault Detection Test Results

In test #0 the fault detectors did not pick up any fault and as expected the aircraft performed the complete mission without problems.

For test #1, the brake fault detector successfully activated after the aircraft failed to slow down after landing, taking 12 s after touchdown to do so. However, when comparing the aircraft speed over time during the landing it is revealed that the aircraft only starts to slow down after 7 s in the control test, as can be seen in Fig. 3. This means that the actual fault detection time for this test was around 5 s. Because of the brake failure the aircraft ran out of asphalt and only stopped in the grassy area surrounding the runways.

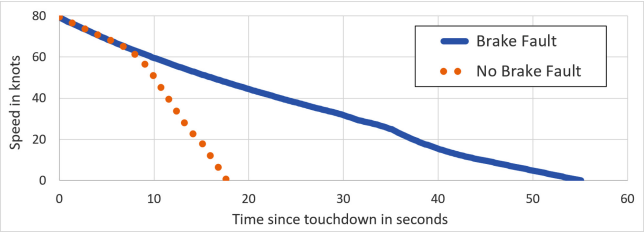

Fig. 3. Speed comparison after touchdown with and without brake fault

Since test #2 ran in an intermittent configuration where the fault was being toggled on and off repeatedly for 3 min, the detector had to correctly determine when the communications were off 3 times, as each on/off cycle takes about a minute. The results of this test can be consulted in Table 6. It achieved an average detection time of 15 s, with all detection times below the 20 s mark, as expected. In this case the aircraft completed the mission normally since communications don't affect the physical behaviour of the aircraft.

The results of the engine faults can be seen in Table 7. All failures were detected and no false positives were recorded. In general, failures that occurred during takeoff were the fastest ones to be detected, followed by the ones during cruising, the descending ones being the slowest overall. Regarding the outcome, the only tests where the aircraft was able to complete the test flight were the

Table 6. Results of intermittent communications fault

Injection timestamp (s)	Pause timestamp (s)	Injection delta (s)	Detection timestamp (s)	Detection delta (s)
96	125	29	114	18
152	187	35	165	13
235	270	35	250	12

ones with single engine failure. In the others the aircraft slowly descended until it hit the ground, without first deploying the landing gear.

While conducting the tests a strange behaviour was detected in the engine faults with ramping. It seemed that the thrust of the engines was not reducing at the expected rate, only starting to decrease after the ramping time was past the half point. This was then confirmed in the collected data when analysing the propeller speed after injecting the fault in tests #11 and #12, as seen in Fig. 4. As can be seen, the fault only starts taking effect after 2/3 of the ramping time and from there it linearly decreases to zero. This is another limitation of FSX that other tests confirm is only present in the AI-controlled vehicles and not in the user-controlled one. This means that the detection times recorded for tests that incorporate ramping are not accurate and the real detection times were included between parentheses for these tests in Table 7.

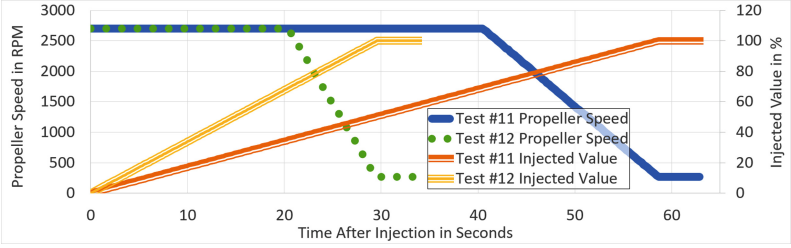

Fig. 4. Propeller speed after ramping fault injection in tests #11 and #12

4.3 Fault Treatment Test Results

With the treatment module enabled, the outcomes of the tests should vary to accommodate the injected faults. Starting with test #0, no changes were detected to mission execution and again no faults were detected.

In test #1 the brake failure was correctly identified once more on landing, but this time the aircraft aborts it, again taking off and making the necessary manoeuvres to approach the longest runway in the airport and land, as suggested in the categorization system. Even with the brake failure, the aircraft was able to stop within the length of the runway. The influence of the treatment module in this test can be seen in Fig. 5.

Table 7. Results of the various engine faults

Test	Injection timestamp (s)	Detection timestamp (s)	Detection delta (s)	Outcome
#3	35061.672	35160.782	99.11	Aircraft able to complete test flight
#4	36666.778	36667.445	0.667	Aircraft able to complete test flight
#5	38242.778	38361.663	118.885	Aircraft able to complete test flight
#6	42241.875	42252.986	11.111	Aircraft crashed
#7	41275.433	41276.099	0.666	Aircraft crashed
#8	40097.436	40148.547	51.111	Aircraft crashed
#9	45742.754	45774.976	32.222 (12.222)	Aircraft crashed
#10	46874.307	46929.862	55.555 (15.555)	Aircraft crashed
#11	50975.629	50996.296	20.667 (0.667)	Aircraft crashed
#12	49585.188	49625.855	40.667 (0.667)	Aircraft crashed
#13	54417.398	54468.509	51.111 (31.111)	Aircraft crashed
#14	56887.613	56944.058	56.445 (16.445)	Aircraft crashed

For test #2 the fault was detected the first time it was triggered, just like in the first test, and immediately the aircraft started changing its course to perform the recommended action of flying over the desired runway, as shown in Fig. 6. This maneuver is intended to inform the ATC that the aircraft has encountered an emergency situation and cannot communicate, so the ATC should clear the runways and airspace for the vehicle to land.

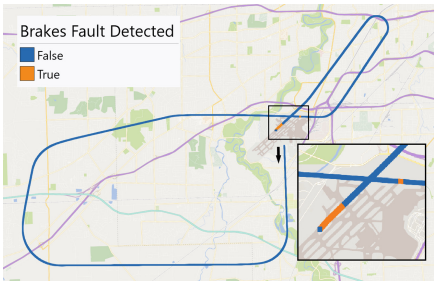

Fig. 5. Test #1 fault treatment path

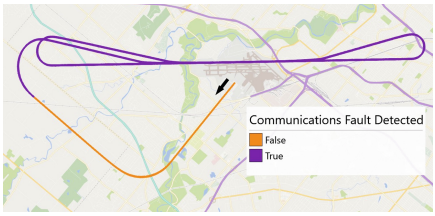

Fig. 6. Test #2 fault treatment path

The fault injected in test #4 was also detected just like in the first test. The aircraft started the emergency landing protocol immediately by redirecting to the closest runway available to land as depicted in Fig. 7. Compared to the first test, where the aircraft was able to finish the mission in a safe manner, diverting

to the airport immediately decreases the chances of an accident in case the fault propagates to the other engine.

Finally, in test #6 the fault was correctly identified, and the same emergency landing protocol was activated as in test #4. However, this time with both engines producing no thrust, the aircraft had no way of making it back to the airport. This was quickly detected and as a consequence the aircraft landed in a close field it knew was uninhabited, as can be seen in Fig. 8. In a real-world scenario this behaviour has the potential to decrease the number of accidents involving bystanders and decrease the probability of losing the aircraft in a crash.

Performance benchmarks were also conducted to test the impact of the new modules on the platform. The test measured the CPU (Central Processing Unit) load, memory allocated and CPU time for the platform in three scenarios: Just the Control Panel open; The Control Panel and Vehicle Agent running without the detection module; and all the modules active. Table 8 displays the results.

Since Flight Simulator is the one that controls the autonomous vehicles, a change in the performance of the platform is not detected from test #1 to test #2. Contrarily, when the detection module is being used, a small increase in CPU load and CPU time is detected but is very small to be significant to affect the overall performance of the platform.

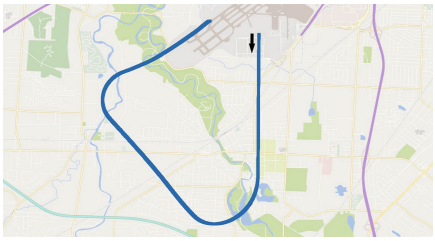

Fig. 7. Test #4 fault treatment path

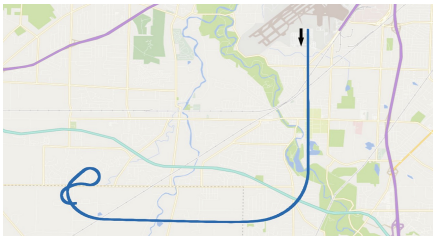

Fig. 8. Test #6 fault treatment path

Table 8. Resources used by the platform with different active modules (test were performed on a Laptop with an Intel Core i7-4710HQ processor @3.30 GHz)

Active modules	Max CPU load (%)	Max. memory (MB)	CPU time per minute (s)
Control Panel (CP)	0.4	35.5	~ 0
CP + Vehicle Agent (VA)	0.4	45.6	~ 0
CP + VA + Detection Module	0.9	46.1	~ 0.7

5 Discussion

The achieved results are promising, with all faults being detected, and no false positives. This shows that the current implementation is robust, accurate and resilient to false triggers. On the other hand, detection times were overall good but not great. This was to be expected since simple fault detection methods were used, while other authors use more advanced ones. This could be improved by using more advanced methods, such as those used in the literature mentioned in Sect. 2. Despite the slow reaction time, it was fast enough to allow the treatment module to intervene in a positive way in otherwise dangerous scenarios.

With some detection times below one second, this simple approach managed to match the detection times in other works that used model-based approaches such as Freeman et al. [4] and Heredia et al. [7], but can't keep up in more demanding scenarios. On another note, this solution managed to achieve an average detection time similar to that of Cork et al. [3]. The work of Purvis et al. [11] is the most similar to this one due to also using a flight simulator as a testbed and using a data-driven method. The use of clustering methods allowed for better results in reaction time with similar detection performance.

6 Conclusion and Future Work

A fault injection tool was successfully implemented in an existing simulation platform, alongside a fault categorization system. Both these components proved useful in the development of a simple but capable fault detection and treatment system for the aircraft controller. The fault detection module managed to perform above expectations, with good detection performance during testing, with comparable results to the works mentioned above, while using much simpler detection methods. The fault detection times were generally good, with time-sensitive faults like brakes and engines being detected quickly enough for the fault treatment module to act. This module also proved to perform well, being able to determine the best action to take when a fault occurred and maintaining the safety of bystanders always in first place by taking into consideration the surroundings of the vehicle. All of this was achieved while keeping the CPU and memory loads very minimal.

The developed work sets a solid base to continue fault-related research in this platform. The fault injection tool in particular is very useful for this kind of research as it helps create detailed fault scenarios for the detection and treatment algorithms that while being tested only with one aircraft, can handle concurrent fault injection in teams of multiple vehicles. The implementation of all the stages of a fault diagnosis system with a modular architecture also facilitates future development of new algorithms without having to redesign the system.

While the results were satisfactory, they could be improved in the future by increasing the number of failures to detect, and using different and/or more sophisticated data-driven methods that analyse more data. To do this it would likely be necessary to base the platform in another similar but more advanced

simulator that can offer more data for AI-controlled vehicles and supports more fault injection options than FSX. Detection times could also be improved by using the mission details to know what should be normal and abnormal behaviour for the aircraft at a certain location or time.

References

1. Belcastro, C.M., et al.: Preliminary risk assessment for small unmanned aircraft. In: Proceedings of the 17th AIAA Aviation Technology, Integration, and Operations Conference, June 2017, Denver, Colorado, USA (2017)
2. Chen, J., Patton, R.J.: Robust Model-Based Fault Diagnosis for Dynamic Systems, 1st edn. Springer, New York (1999). https://doi.org/10.1007/978-1-4615-5149-2
3. Cork, L.R., Walker, R., Dunn, S.: Fault detection, identification and accommodation techniques for unmanned airborne vehicle. In: Proceedings of the 11th Australian International Aerospace Congress (AIAC 2005), 14–17 March 2005, Melbourne, Australia (2005)
4. Freeman, P., Pandita, R., Srivastava, N., Balas, G.J.: Model-based and data-driven fault detection performance for a small UAV. IEEE/ASME Trans. Mechatron. **18**(4), 1300–1309 (2013)
5. Garrido, D.: Fault injection, detection and handling in autonomous vehicles. Mathesis, Faculty of Engineering of the University of Porto (2019)
6. Gertler, J.J.: Survey of model-based failure detection and isolation in complex plants. IEEE Control Syst. Mag. **8**(6), 3–11 (1988)
7. Heredia, G., Ollero, A., Bejar, M., Mahtani, R.: Sensor and actuator fault detection in small autonomous helicopters. Mechatronics **18**(2), 90–99 (2008)
8. Isermann, R.: Model-based fault-detection and diagnosis - status and applications. Ann. Rev. Control **29**(1), 71–85 (2005)
9. Isermann, R.: Fault-Diagnosis Systems. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-30368-5
10. Miljković, D.: Fault detection methods: a literature survey. In: Proceedings of the 34th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2011), 23–27 May 2011, Opatija, Croatia, pp. 750–755 (2011)
11. Purvis, A., Morris, B., McWilliam, R.: FlightGear as a tool for real time fault-injection, detection self-repair. Proc. CIRP **38**, 283–288 (2015)
12. Schoenwald, D.A.: AUVs: in space, air, water, and on the ground. IEEE Control Syst. Mag. **20**(6), 15–18 (2000)
13. Silva, D.C.: Cooperative multi-robot missions: development of a platform and a specification language. Ph.D. thesis, Faculty of Engineering, University of Porto (2011)