Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 38 (2015) 283 – 288

www.elsevier.com/locate/procedia

The Fourth International Conference on Through-life Engineering Services

FlightGear as a tool for real time fault-injection, detection and self-repair

Alan Purvis[a], Ben Morris[a], Richard McWilliam[a]

[a]*School of Engineering and Computing Sciences, Durham University, South Road, Durham, DH1 3LE, UK*

[*] Corresponding author. Tel.: +44-191-3342437-0000; mob: +44-771-350-5409. *E-mail address:* alan.purvis@durham.ac.uk

Abstract

The development of fault-detection and self-healing methods at both a hardware and software level in modern aircraft is an attractive prospect. However it is expensive to design and test these techniques using real aircraft. This paper appraises the viability of using FlightGear, an open-source Flight Simulator, as a test-bed for these approaches. The paper characterises the realism of various aspects of a model of the Airbus A380. Interfaces are established to abstract critical control system routines from FlightGear. These functions are replicated in both software and hardware environments. The control data can then be subjected to fault-injection and the control modules modified to enable fault-detection and self-healing. By applying cluster analysis techniques to training sets of data, a fault-detection, diagnosis and self-healing model is designed to address these injected faults. FlightGear is found to provide highly realistic simulation of aircraft systems and instrumentation. Hardware-in-the-loop testing shows promise as an area for future work. The proposed fault-detection model is found to provide 96% accuracy.
© 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license
(http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Programme Chair of the Fourth International Conference on Through-life Engineering Services.
Keywords: Self-Healing; FlightGear; Hardware-in-Loop; Fault-Injection; Flight Simulation

1. Main Text

On 1 June 2009 Air France Flight 447, an Airbus A330, entered an aerodynamic stall and crashed into the Atlantic Ocean, killing all 228 people on board [R]. The final report into the accident found that the crash resulted from a succession of events starting with an inconsistency in airspeed readings caused by icing of the aircraft pitot tubes[1].

In this, as in over 50% of all fatal aircraft accidents, pilot error was cited as a major contributing cause [2]. It is clear, however, that these errors were prompted by a lack of accurate information from the aircraft's systems - the pilots were unable to tell which instruments to trust. The icing of the pitot tubes was evidently the root cause of the accident, further, as was pointed out in [1] and [3], while angle of attack data was sent back to the systems computer it was not displayed to the pilots. The display of this and other data could have allowed them to better handle the situation [4]. It can be said that the safe operation of an aircraft depends on the pilot or auto-pilot receiving clear and accurate data relating to its position, orientation, velocity and the forces acting upon it. When the systems providing this information malfunction it is, at best, an inconvenience and in the worst cases can contribute to disaster given time and oversight.

Fault-detection is a field of control engineering concerned with monitoring a system and identifying when a fault has occurred [5]. At the hardware level, a real-time fault-detection approach permits the reliability of critical systems to be analysed on a second-by-second basis. If the system is sufficiently understood it is feasible that the causes of errors could not only be detected but diagnosed and possibly remedied in real-time. Fault-Detection and Self-Repair techniques could provide an extra layer of safety tolerance in aircraft systems which allow safety improvement without the necessity for the introduction of expensive redundancy.

Aircraft manufacturers are particularly sensitive to change. Poorly designed or implemented fault detection routines can result in high rates of false-positives which can be costly and inconvenient for passengers and airliners. Any new system must, therefore, be tested rigorously and its performance in a wide range of scenarios characterised. Access to real aircraft or professional grade simulators, however, is often limited and expensive.

This paper considers whether the open source flight simulator FlightGear could provide a useful test-bed for the development of such approaches and additionally appraises its usefulness in hardware-in-the-loop integration. It also asks whether fault-detection and self-healing techniques can be applied to create an *Autopilot Supervisor*, offering a real-time picture of the health of key system instruments and, where faults are detected, present alternative readings clearly.

2212-8271 © 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license
(http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Programme Chair of the Fourth International Conference on Through-life Engineering Services.
doi:10.1016/j.procir.2015.08.040

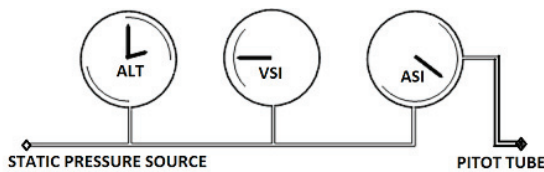

Fig. 1. Diagram of the Pitot-Static system and related instrumentation from [7]

2. FlightGear

At a fundamental level the variables that a pilot is most interested in are those describing the position and orientation of the aircraft and the forces acting upon it. It is when measurements of these variables go wrong that the most serious problems arise.

The fundamental variables describing the relationships between these frames of reference are defined by the flight dynamics model (FDM) used by the simulation. A number of FDMs are availale in FlightGear. The FDM typically used is JSBSim. It is a lightweight, non-linear Six-Degree of Freedom (6DoF) simulation for modelling flight dynamics and control for aircraft [6].

2.1. Interfacing with FlightGear

The Property Tree is described as the "central nervous system" of FlightGear and one of its most powerful assets. It is an interface to low level, run time state variables stored in an intuitive tree-like hierarchy.

2.2. Pitot-Static System

The Pitot-Static system drives several fundamental aircraft instruments; the ASI, Altimeter and VSI, several commercial disasters have been linked to its failure.

Mechanically the pitot-static system consists of a pitot tube, a forward facing tube exposed to the relative wind which measures stagnation pressure, and a static pressure port, a side mounted port which measures static pressure. These two pressure measurements are then used to provide measurements of Indicated Airspeed (IAS), Vertical speed (V/S) and Altitude (Alt). Figure 1 demonstrates the way in which these systems interact.

The pitot-static system is realistically modelled in FlightGear. The FGSource environment module creates an environmental static pressure around the aircraft from altitude lookup tables. The pitot module update function takes the environmental pressure p, aircraft Mach speed (M), angle of attack (α) and side-slip (β). It first calculates a projection factor, to account for orientation errors, as in Equation 1 [8].

$$X = cos(\alpha)|cos(\beta)| \tag{1}$$

The stagnation or pitot pressure p_t is then calculated using Equation 2 [8].

$$p_t = p\left(1 + \left(0.2M^2X^2\right)^{7/2}\right) \tag{2}$$

The static module, accounting for errors due to side-slip and AoA in a similar way, calculates a static port pressure p_s in a similar manner [8].

Considering the ASI: Equation 3 shows how the difference between p_s and p_t provides an impact pressure q_c [8].

$$q_c = p_t - p_s \tag{3}$$

q_c is related mathematically through Equation 4 - where p_0 is standard pressure at sea level and a_0 is the standard speed of sound at 15°C - to a velocity, the IAS [8] [9].

$$IAS = a_0\sqrt{5\left[(\frac{(q_c)}{p_0} + 1)^{2/7} - 1\right]} \tag{4}$$

The IAS is not corrected for variations, with altitude, of dynamic air pressure and hence does not represent the true speed at which the aircraft is travelling through the air (TAS) and will under-read the greater the altitude of the aircraft [10]. However the aerodynamic behaviour of an aircraft is linked directly to the IAS; an aircraft at any altitude will stall at the same IAS. It is therefore critically important that it be measured accurately.

When creating a fault detection model it is important to consider the modes of failure of a system. There are a number of modes of failure of the pitot-static system and the behaviour of the attached instrumentation will depend both on which mode of failure occurs and what phase of flight the aircraft is in. If the pitot tube is blocked the only reading affected will be IAS, which will increase in a constant speed climb - running the risk of aircraft stall, and decrease during descent - running the risk of over-speed beyond critical the critical Mach number, and aircraft breakup [11].

A blocked static port is a more serious problem than pitot failure. Altitude measurements will become stuck, V/S will be frozen at zero and IAS will behave in the opposite manner to a pitot failure - showing falling IAS during a constant speed climb and rising IAS during constant speed descent [11].

The system's behaviour under an additional mode of failure, electronic faults, is less strictly defined. The nature of hardware-based faults means that such instances can cause erratic data behaviour, even occasionally causing erroneous data in some situations but not others. This is explored more using real hardware in Section 3.1.

3. Fault Injection

FlightGear has some rudimentary fault injection capability. Systems such as the Vacuum, Pitot or Static can be turned on and off by toggling Boolean values in the FlightGear property tree. This was insufficient for the purposes of this paper and further programming work was required to create an environment where performance under many different failure modes could be tested.

As discussed in Section 2.2 there are a number of different modes of failure for the pitot-static system, both mechanical and electronic [11]. In order to have the capability to model a variety of different kinds of fault being introduced into the system it was necessary to turn off the native Pitot-Static calculations and replace them with custom, faulty, calculations. Using understanding of the pitot.cxx and airspeed.cxx modules, the mathematics behind the pitot-static system, and the failure

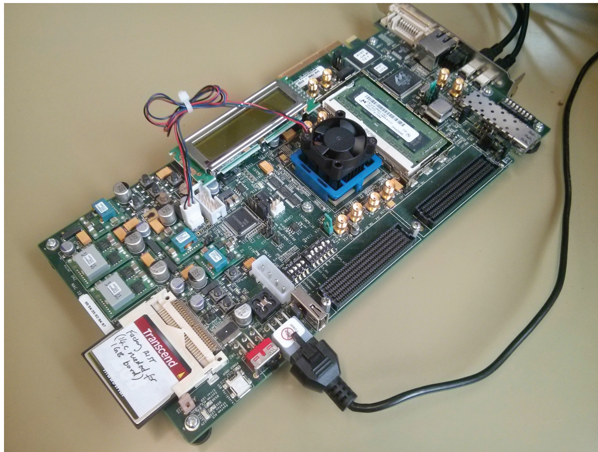

Fig. 2. Image of Xilinx FPGA board

Fig. 3. The effect of hardware pitot fault injection

modes of the pitot static system, code was generated to inject a number of different types of faults. These included stuck, low, high, drifting and noisy values.

3.1. Implementation

A custom UDP protocol, outputting from FightGear the environmental variables used in the Pitot-Static calculations, and taking in the results from those calculations, was created. This permitted Pitot-Static calculations to be carried out by re-implementing the previously developed code onto a simple UDP server written in C. It was then possible to toggle between the (correct) on-board and (incorrect) off-board calculations from within FlightGear by switching a variable in the property tree. In this way it was possible to insert errors into the Pitot-Static system without changing anything within FlightGear.

3.2. Hardware-In-The-Loop

The UDP connection provided further interesting options. The abstraction from the software did not need to be limited to another software implementation. A parallel project by another student was exploring the idea of hardware-based fault-injection and self-repair on a Xilinx FPGA platform [12]. By lifting the pitot-tube calculation from software and implementing it on the FPGA picture in Figure 2 it was possible to experiment with hardware-in-the-loop testing. Faults were injected into the hardware in a manner to simulate bombardment by high-altitude radiation.

The effect of the bit-flips involved in hardware fault-injection saw data taking extreme values, or multipliers being applied to pitot-tube data. Figure 3 shows this phenomenon - switching the serviceable tag from true to false causes the subroutine to be switched from running in the software to running on the Xilinx board. In this case the fault on the board causes an extremely small pitot pressure, correspondingly the measured IAS (on the sliding left-hand axis) becomes very close to zero. The underlined green value is the true airspeed of the aircraft and highlights this discrepancy.

While this helped develop a more robust software-level fault-detection package the importance of this testing was not in the results themselves but in that it demonstrated a useful way of quantifying and observing how hardware faults propagate in a real system. FlightGear can, in this way, act as a useful tool or test-bed for developing and testing hardware-based fault-correction methods, minimising the need to use expensive professional-grade simulators.

4. Fault-Detection, Diagnosis and Self-Healing

Error-correction requires three-phases to be carried out - fault-detection, fault-diagnosis and finally a self-healing phase. To appraise the results of a proposed error-correction model the effectiveness of each of these phases should be quantified.

Faults, in a model-based fault-detection system, are raised when system values deviate significantly from the values expected by a model. Fault-detection is typically judged by two metrics. These are the false positive and the false negative rate, often used interchangeably with the terms Type I and Type II errors respectively. In the context of a flying aircraft, which can go wrong in a matter of seconds, it is also important to consider the response-time - how quickly a fault flag is raised after injection.

Fault-diagnosis is the process of categorising detected faults. The success of a fault-diagnosis method is defined by the percentage of injected faults that are correctly identified.

Self-healing is the process of providing alternative readings to instruments that are found to be faulty. The accuracy of a self-healing method can be defined as the difference between the proposed alternative reading for an instrument, and the value of that parameter were it operating normally at that point in time. In FlightGear this can easily be measured by creating and measuring the output of fault-free duplicates of all instruments under scrutiny.

4.1. Modelling and Data Analysis

Time series cluster analysis is used to identify structure in an unlabelled data set by organizing data into homogenous groups [13]. Flight data variables consists of a regularly sampled (10Hz) set of parameters. Many of these parameters are correlated with one another and by analysing this data it is possible to detect unexpected behaviour and discrepencies. In this instance cluster analysis was used, during a training phase, to divide the aircraft's systems and instrumentation time series data into labelled faulty and non-faulty clusters. In this way, it was hoped, a model could be developed to assign live data in real-time to these clusters and identify those samples which represented faulty data.

The initial time series used contained the ten parameters obtained from five different systems (GPS contains two parameters, Altitude and True Ground Speed (TGS)). Some preprocessing of the data was carried out. Equations 5 and 6 show how an estimate for a GPS derived VSI at time k was obtained, using the real-time change in altitude δAlt and sample rate SR in Hz. A fourth-order low-pass Butterworth filter, with coefficients c_0 - c_4 was applied to filter out noise in GPS-Altitude[14]. The filter was chosen because it was stable and performed well under testing.

$$\delta Alt_k = (GPSAlt_k - GPSAlt_{k-1}) \quad (5)$$

$$GPS_{VSI_k} = SR * \frac{(c_0 \delta Alt_k + c_1 \delta Alt_{k-1} + ... + c_4 \delta Alt_{k-4})}{\sum_{i=0}^{4} c_i} \quad (6)$$

Indicated Airspeed can be approximately related to True Airspeed (TAS) by increasing it by 2% for each 1000ft increase in altitude [10]. Equations 6 - 7 show how GPS-TGS was approximated to TAS and then used, with GPS-Alt, to provide a GPS inferred IAS (note this fails to account for wind speed).

$$GPS_{TAS} \approx GPS_{TGS} \quad (7)$$

$$GPS_{IAS} \approx \frac{GPS_{TAS}}{1 + \frac{0.02 GPS_{Alt}}{1000}} \quad (8)$$

4.2. Variable Normalization

It is important that the variables used are of similar order such that they have similar weight in the calculation of cluster centroids. In order to carry out cluster analysis it was useful, therefore, to normalize variables, creating dimensionless numbers representing relationships between multiple variables.

Some of these numbers were obtained heuristically - it is fairly obvious that GPS Altitude would be expected to vary linearly with pressure Altitude. Others were obtained through a combination of dimensionless analysis and examination of the data - by looking at the rate of climb as a function of engine speed (N1) and pitch it was possible to compare the VSI and GPS VSI with the engine and vacuum systems. IAS is shown to vary significantly with GPS IAS, then a quantity representing the ratio between the two is shown to be relatively consistent across the duration of a flight.

Although many combinations of parameters were explored in the end four normalized variables, Equations 9, 10, 11 and 12, representing correlations between seven of the ten parameters initially analysed were selected. Note a multiplier has been added in heuristically in Equation 12 to bring the mean value to approximately 1.0.

$$Var1 = \frac{IAS}{GPSIAS} \quad (9)$$

$$Var2 = \frac{Altitude}{GPSALT} \quad (10)$$

$$Var3 = \frac{VSI}{GPS_{VSI}} \quad (11)$$

$$Var4 = \frac{4 * N1 * pitch}{GPS_{VSI}} \quad (12)$$

4.3. k-Means Clustering

A common partitioning method, the k-means algorithm, aims to partition a sample of data-points into a predefined number of clusters, each with a mean value or centroid [15]. It works by the minimization of an objective function, based on the *within cluster sum of squares (WCSS)*. Each vector in the time series of the sample belongs to one of the clusters. The algorithm starts by creating arbitrarily the desired number of cluster centroids. In the assignment step each observation is assigned to the cluster whose centroid is closest. The cluster centroids are then updated, becoming the centroids of the samples now placed in the new clusters. The assigment phase then recommences and the algorithm repeats until the assignments no longer change [15].

The k-means algorithm was applied to various time series of these normalized variables generated from test flights with and without faults injected. These clusters were then labelled, using knowledge of the data set from which they were derived, as "fault-free", "pitot-high", "pitot low", "static blocked" etc., and integrated into a fault-detection model. This created an initial set of k vectors of the co-ordinates of the centroids or means of the clusters, $m_1,...,m_k$.

Once the system is "trained" it is used to classify new data points. At time t, the model takes a new vector of the n normalized system parameters, v_t, and calculates its squared Euclidean distance from each of the cluster centroids, as in Equation 13, which shows the squared euclidean distance from the ith centroid.

$$d^2(v_t, m_i) = (v_{t_1} - m_{i_1})^2 + (v_{t_2} - m_{i_2})^2 + ... + (v_{t_n} - m_{i_n})^2 \quad (13)$$

The vector is then assigned to the cluster to which this distance is shortest. If the latest vector has been assigned to the "fault-free" cluster then no action will be taken, if it has been assigned to one of the faulty clusters then an error detected flag will be raised with an associated diagnostic message and the self-repair method will be called.

4.4. Expectation-Maximization Clustering

One problem with k-means clustering is that it forms hard, spherical, cluster boundaries, using the Euclidean distance, that do not account for any variation in the shape or size of the clusters [16]. While this is useful for some applications, in the case where distributions do not fit these pre-defined patterns it can provide unsatisfactory results. Figure 4, showing the distribution of labelled data for Var 1 and Var 2 when a static fault is

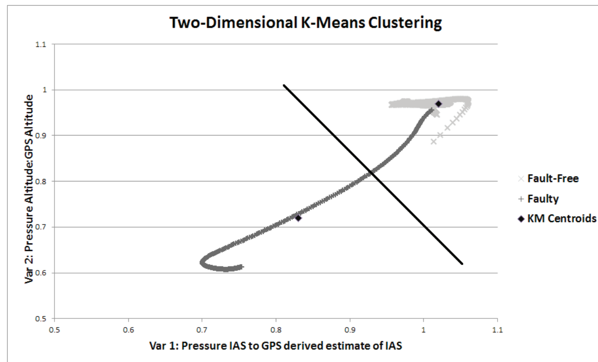

Fig. 4. Graph demonstrating two-dimensional K-Means Cluster Analysis

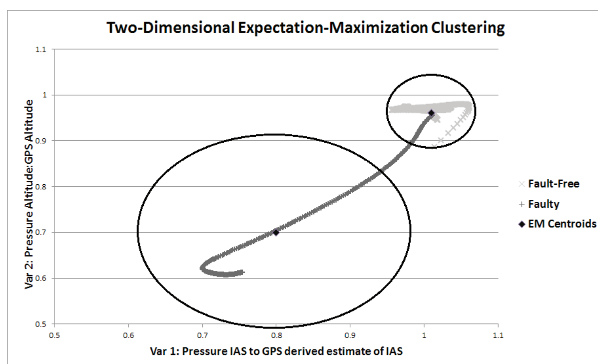

Fig. 5. Graph showing two-dimensional EM Cluster Analysis

introduced into the system, demonstrates this problem. While the fault-free data is very tightly clustered, the labelled faulty data set is much more broadly spread. As a result the k-means method, despite introducing cluster centroids in sensible locations, will mis-classify a number of faulty points as being fault-free. This results in lower detection rates and less satisfactory performance.

An alternative to the k-means method is the Expectation-Maximization (EM) algorithm. EM assumes samples are members of a number of Gaussian distributions. Similar to k-means it provides estimates of the locations of cluster centroids, representing the Gaussian means. However, in addition to the coordinates of the mean, each cluster centre has an associated vector of variances, representing the Gaussian width.

When the model is created it seeds randomly both the cluster mean locations *and* a matrix of variances for each cluster. The EM algorithm iteration then alternates between performing two steps. The Expectation step calculates, based on the current estimate of the mean and variance for each cluster, the expected value a log-likelihood function that describes the likelihood that the current position of the Gaussian distributions is a correct representation of the observed data. The Maximization step then chooses the values for means and variances that maximize this log-likelihood function. This log-likelihood function of the new distributions is then calculated and the cycle continues until the a maximum value is reached.

Rather than being hard-assigned to a cluster, the probability

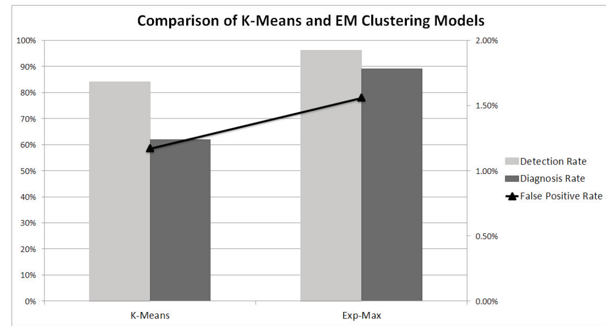

Fig. 6. Comparison of K-Means and EM clustering models over 20 flights

of a new sample belonging to each system cluster is calculated, and the sample is then assigned to the cluster to which it is most likely to belong. This means that the acceptance rate of clusters with relatively tight groupings is much lower than in k-means clustering and should provide a lower rate of false-negatives. Figure 5 demonstrates this improvement by overlaying the results from EM clustering on the same data as before.

4.5. Model Testing

The data from the training sets were used to assemble nine labelled clusters in the four normalized variables. One cluster represented the data behaving normally and the other eight clusters represented different types of labelled fault.

Both models, one based on k-means and one based on E-M clustering, were tested on 20 sets of flight data. All flights followed the same pattern, taking off from runway 27L at London Heathrow (EGLL) airport and following a standard instrument departure (SID) path before setting on a course for New York JFK (KJFK). Once the airspeed had settled at 210kts and an altitude of 5000ft had been reached data recording began. The fault-injection window lasted for approximately 5 minutes and contained two faults that fell into six broad categories, Pitot-Stuck, Pitot-High, Pitot-Low, Static-Stuck Static-High and Static-Low. These faults were injected at random times within the window for between 15-30 seconds each.

The results of this testing are shown in Figure 6. The model based on the EM clustering method, as expected, performed significantly better at fault detection, with an average detection rate of 96% compared to 84% for the k-means method. This difference in performance comes about from the flawed assumption of the k-means model that clusters in the data have identical widths, and led to misassignment of faulty data as being fault-free. This same property lead to the k-means method performing slightly better in terms of throwing false-positives. It is expected that the implementation of a sensibly calibrated low-pass filter could allow the EM model to find an acceptable balance between false-positive rate and response-time, which was almost instant when analysing data on a sample-by-sample basis.

Of faults that were detected the EM model correctly identified the faulty system (pitot/static) in 95% cases and the precise nature of the fault (high, stuck etc) in 89%. The k-means method performed poorly in this area. Once faults had fully manifested it provided the correct categorisation, but in the in-

terim period it failed to deliver good results, on average delivering 80% system categorisation and classifying the nature of the fault 62% of the time.

The self-healing model was similar for both methods. When faults were isolated the system used *Var*1 - *Var*4 to provide alternative instrument readings for those instruments deemed to be faulty. Both methods performed extremely well here, delivering alternative IAS, Altimeter readings with 98% of the correct value and and VSI readings within 90% of the true value. While these results seem very positive the GPS system in FlightGear is overly accurate and this could be going some way towards providing these excellent results.

5. Conclusions and Further Discussion

This paper finds that FlightGear provides detailed simulation of many areas of aircraft avionics. In particular the fundamental flight instruments are modelled accurately and in a detailed manner. It concludes that in these areas FlightGear can be a useful tool for testing and visualising methods of real-time fault detection and diagnosis.

FlightGear does not have sufficiently realistic modelling of engine thrust control to carry out fault-analysis in this area. An XML or Nasal-based implementation of a control system similar to a simple FADEC, monitoring engine and environmental parameters while actuating fuel-flow, would introduce the extra layer of abstraction necessary to fault-test electronic engine faults and could provide interesting results.

5.1. Fault-Detection and Self-Healing

The goal of the Fault-Detection and Self-Healing experimentation was to explore the possibility of creating an *Autopilot Supervisor* that provides a layer of software-redundancy in the case that physical or non-recoverable electrical fault causes a system to become unreliable. The suggested model, based on an Expectation-Maximization clustering approach, detects 96% of faults and has a false-positive rate of 1.5%. The self-repair system implemented provides very accurate results. While this is partly due to an over accurate GPS implementation it demonstrates that the methods are sound. However, due to the limited testing (only 20 flights worth of data was tested). While this model is an improvement on no error-detection at all a complete solution needs further study. Significantly more fault-types and flight situations must be explored before this model could be described as a reliable *Autopilot Supervisor*.

In terms of improving the model a number of the false-positives detected were a result of system noise causing data entries to cross cluster boundaries. Many engine and inertial (accelerometers) system variables showed promising correlations with various system instrumentation. By reconciling more instruments into normalized variables the diversity of the solution space will be increased more system cross-checks provided. Ultimately this will create a more complete picture of the system and should allow for faults to be characterised more accurately and reliably.

5.2. Hardware-in-the-Loop

FlightGear shows enormous promise as a test-bed for hardware fault-injection. Rather than extracting only the pitot-tube subroutine, it could be possible to extract multiple subroutines to FPGAs, or even use real aircraft components in the loop, providing a rigorous and exciting testing environment. Further testing of system performance under hardware fault-injection should be carried out and the performance at software level of hardware self-healing methods characterised.

If combined with a top-down software based approach the hardware route could help pave the way to a complete suite of fault-detection and self-healing methods. Providing a hybrid top-down and bottom-up approach to fault-detection and self-healing that could lead to improved component resilience and ultimately lower costs and higher levels of safety in aircraft.

Acknowledgements

This work was carried out with the support of the EPSRC Centre for Innovative Manufacturing in Through-life Engineering Services. [EP/IO33246/1]. AP is also grateful for the support and welcome he received whilst a visitor at the University of Sydney Australian Centre for Field Robotics.

References

[1] BEA, “Final report on the accident on 1st june 2009 to the airbus 330-203 registered f-gzcp operated by air france flight af 447 rio de janeiro - paris,” http://www.bea.aero/docspa/2009/f-cp090601.en/pdf/f-cp090601.en.pdf, 2012, (Accessed April 2015).
[2] “Plane crash info,” http://www.planecrashinfo.com/cause.htm, (Accessed April 2015).
[3] N. Ross, “Air france flight 447 - damn it we’re going to crash,” http://www.telegraph.co.uk/technology/9231855/Air-France-Flight-447-Damn-it-were-going-to-crash.html, 2012, (Accessed April 2015).
[4] G. Salvendy. and M. J. Smith., Eds., *Human Interface and the Management of Information*. Springer Verlag ISBN 978-3-642-21669-5, 2011, (Pilot Information Presentation on the Flight Deck, Machiarella et al. p500).
[5] I. Hwang, “A survey of fault detection, isolation and reconfiguration methods,” *IEEE Control Systems Society*, vol. 18, 2010.
[6] “Jsbsim,” http://jsbsim.sourceforge.net/, (Accessed December 2014).
[7] D. Parry, “Pitot-static instruments,” http://www.langleyflyingschool.com-/Pages/CPGS/(Accessed April 2015).
[8] “Flightgear pitot module,” flightgear-3.0.0/src/, (Accurate as of version 3.0.0).
[9] “True, equivalent and calibrated airspeeds,” http://www.mathpages.com-/home/kmath282/kmath282.htm, (Accessed January 2015).
[10] CSG, “True airspeed calculator,” http://www.csgnetwork.com-/tasinfocalc.html, (Accessed February 2015).
[11] L. Monteiro, “The dramatic effects of pitot-static system blockages and failures,” *LuizMonteiro.com*, vol. 1, pp. 1–14, 2013.
[12] T. Carney, *Modelling Electronic Circuit Failures using a Xilinx FPGA system*, preprint, April 2015.
[13] T. Liao, “Clustering of time series data - a survey,” *The Journal of The Pattern Recognition Society*, vol. 38, 2005.
[14] C. Bond, “Butterworth filters,” *Polynomials, Poles and Circuit Elements*, 2003, http://www.crbond.com/papers/btf2.pdf.
[15] D. MacKay, *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003, (Chapter 20: An Example Inference Task: Clustering).
[16] N. Alldrin, “Clustering with em and k-means,” http://cseweb.ucsd.edu/ atsmith/project1_253.pdf, (Accessed March 2015).