Proving the Correctness of Multicopter Rotor Fault Detection and Identification Software

Ankita Bhaumik*, Airin Dutta†, Fotis Kopsaftopoulos†, Carlos A. Varela*
*Department of Computer Science
†Department of Mechanical, Aerospace and Nuclear Engineering
Rensselaer Polytechnic Institute, Troy, New York
{bhauma, duttaa5, kopsaf}@rpi.edu, cvarela@cs.rpi.edu

***Abstract*—Applications for data-driven systems are expected to be correct implementations of the system specifications, but developers usually test against a few indicative scenarios to verify them. In the absence of exhaustive testing, errors may occur in real time scenarios, especially when dealing with large data streams from moving objects like multicopters, vehicles, etc. Model checking techniques also lack scalability and completeness. We present a novel approach based on some existing tools which enables a developer to write high level code directly as system specifications and simultaneously be able to prove the correctness of the generated code.**

We present a fault detection and identification (FDI) software development approach using declarative programming language: PILOTS. The grammar of PILOTS has been updated to enable easier syntax for threshold validation techniques. The failure detection model is described as high level specifications that the generated code has to adhere to. The complete FDI problem is formally specified using Hoare logic and proven correct using an automated proof assistant: Dafny. A case study of rotor failures in a hexacopter has been used to illustrate the approach and visualize the results.

***Index Terms*—fault detection, formal verification, multicopter, declarative programming, Dafny**

I. Introduction

A fault in any component of a safety-critical data-driven system can be catastrophic to life and property. Therefore, timely and accurate fault detection and identification (FDI) is critical and can be a challenging problem. FDI applications are "correct" if they accurately estimate the state of a system. There are several failure detection and identification techniques available for data-driven systems [1], [2], but it is difficult to formally verify that they are correct. Applications use models for data that can be quite complicated and software systems implementing these models add a layer of complexity. To enable portability and usability, we advocate for high level declarative programming approaches to design, develop, and verify FDI software for data-driven systems. This approach is user-friendly with domain experts enabling them to efficiently implement new failure models in the application.

In this paper, we consider proving the correctness of a streaming application written in our own highly declarative programming language with respect to a data-driven model. PILOTS is a declarative language that has been used for sensor failure detection and state estimation for aircraft data streams. The goal is to prove that a PILOTS program correctly assesses the state of the system using the provided data-driven model. We consider a subset of PILOTS programs that analyze stream data from multiple channels and outputs the mode of the system. We consider a PILOTS program that implements a knowledge based algorithm to detect rotor faults in a multicopter [3] and use Floyd-Hoare style logic reasoning to verify the correctness of the code [4], [5]. System specifications generated using a *PILOTStoDafny* code generator are used to formally prove the program correct with respect to its data-driven model. We use Dafny which automates this verification process using the Z3 SMT solver [6]. The *PILOTStoDafny* code generator produces Dafny code and specifications during compilation of the PILOTS code. Finally, we present the results of the FDI software written in PILOTS for different rotor failures and compare them with the ground truth obtained from the same algorithm to conclude about the correctness of the application considering factors of frequency and availability of the input data streams.

There are two advantages of using PILOTS for this process. First, the high-level specifications in this language are relatively straightforward and can be easily implemented for various data-driven domains. Second, if we can formally verify the compiler's correctness, we can increase the reliability of the system regardless of its applications.

II. Background

A. PILOTS

ProgrammIng Language for spatiO-Temporal data Streaming applications (PILOTS) is a high level declarative programming language for the development of applications for analyzing spatio-temporal data streams [7]–[9]. PILOTS supports interpolation of missing data, error detection and estimation of correct data streams based on analytical redundancy [10]. Output data streams are produced using the mathematical model specified by the application. Erroneous patterns in data are detected using formalized error signatures [11] and modes [12]. It has been demonstrated to successfully detect failures from simulated real time flight data for multiple commercial aviation accidents including the Air France AF447 accident in June 2009 [13].

A PILOTS program has the following sections: 1) `inputs`: specifying the input data streams and the method to interpolate missing data: *closest*, *euclidean*, and *interpolate*; 2) `outputs`: output streams generated by the application with a given frequency; 3) `errors`: residual functions to compute a numerical value from the input data streams over a window of time; 4) `signatures/modes`: constrained functions to capture patterns in the error residuals and estimate the state of the system. The set of error signatures is determined using known failure modes of the system and the corresponding pattern of the error residuals. Fig. 1 shows a simple PILOTS program `Twice`. It takes as input two input streams $a(t)$ and $b(t)$, where b should be twice the value of a. Both a and b are expected to increase by one for a and two for b every second. Thus, the error is zero in the "Normal" mode. If there is unavailability of the signal a, the error keeps on increasing with a slope of 2. Similarly, if there is unavailability of the signal b, the error keeps decreasing with a slope of 2. These patterns can be captured using error signatures: $e = 2t + k$ and $e = -2t + k$, where k is a constant. We calculate $\delta_i(t)$, the distance between the measured error residual $e(t)$ and each error signature S_i by:

$$\delta_i(S_i, t) = \min_{g(t) \in S_i} \int_{t-\omega}^{t} |e(t) - g(t)| dt. \tag{1}$$

where ω is the window size. The smaller the distance $\delta_i(t)$, the closer the raw data is to the theoretical signature S_i. Then, the *mode likelihood vector* is calculated as $L(t) = \langle l_0(t), l_1(t), \ldots, l_n(t) \rangle$ where each $l_i(t)$ is:

$$l_i(t) = \begin{cases} 1, & \text{if } \delta_i(t) = 0 \\ \frac{\min\{\delta_0(t), \ldots, \delta_n(t)\}}{\delta_i(t)}, & \text{otherwise.} \end{cases} \tag{2}$$

Given a significance threshold $\tau \in (0, 1)$ we check for one likely candidate l_j that is sufficiently more likely than its successor l_k by ensuring that $l_k \leq \tau$. Thus, we determine j to be the correct mode by choosing the most likely error signature S_j. If $l_k > \tau$, then regardless of the value of k, "Unknown" error mode (-1) is produced.

```
program Twice;
    inputs
        a (t) using closest(t);
        b (t) using closest(t);
    outputs
        e: (b - 2 * a) at every 1 sec;
end
```

Fig. 1: Program Twice in PILOTS.

B. Dafny

Dafny is an imperative, sequential language that can be used to statically verify code based on provided specifications. It is a satisfiability-modulo-theories (SMT) based program verifier which requires minimum interaction of the user with the solver. Dafny takes as input a program along with programmer specified constraints and verifies whether the code satisfies the specifications for all possible inputs. The specifications may be written as preconditions and postconditions, loop invariants and termination metrics. Preconditions verify the inputs to the program and postconditions specify statements that should hold true after the program snippet completes execution. Loop invariants are conditions that should hold true before and after every iteration of a loop. The verifier translates a given program into the intermediate verification language Boogie 2 [14], [15]. The Boogie tool is then used to generate first-order verification conditions that are passed to an SMT solver [16].

A method is the smallest piece of executable code in Dafny. Methods have input parameters and return statements like any other imperative programming language. Dafny is more advanced because it has the power to add specifications to these methods and ascertain whether these constraints are satisfied for all possible inputs and execution paths. Keywords `requires`, `ensures`, `invariant` and `assert` are the standard ways to express preconditions, postconditions, loop invariants and inline assertions respectively. Fig. 2 illustrates a method written and verified in Dafny that will return the maximum element in an array. The `ensures` clause ascertains the program always returns the intended output. The loop invariants and `decreases` clause is used to arrive at the postcondition and prove termination of the program. The `requires` clause specifies preconditions that restrict the inputs to the program.

```
method max(arr: array?<int>) returns (z: int)
    requires arr!=null && arr.Length > 0
    ensures forall k:: 0 <= k < arr.Length ==> arr[k] <= z
    {
        var i := 1;
        z := arr[0];
        while (i < arr.Length)
        decreases arr.Length-i
        invariant i <= arr.Length
        invariant forall k:: 0 <= k < i ==> arr[k] <= z
        {
            if(arr[i]>z)
            {
                z := arr[i];
            }
            i := i+1;
        }
    }
```

Fig. 2: A verified method in Dafny to find the maximum element in an array.

C. Hexacopter model and data generation

In this study, data was generated using a rigid body flight simulation model of a regular hexacopter (Fig. 3) using a summation of forces and moments to calculate aircraft accelerations. This model allows for the simulation of abrupt rotor failure by ignoring the failed rotor inflow states and setting the output rotor forces and moments to zero. A PID-based feedback controller is implemented on the nonlinear model to stabilize the aircraft altitude and attitudes, as well as track desired trajectories written in terms of the aircraft

velocities [17]. This control design has been demonstrated to perform well even in the event of rotor 1, 2 or 6 failure, with no adaptation in the control laws themselves. A continuous Dryden wind turbulence model is implemented to simulate realistic flight conditions [18].

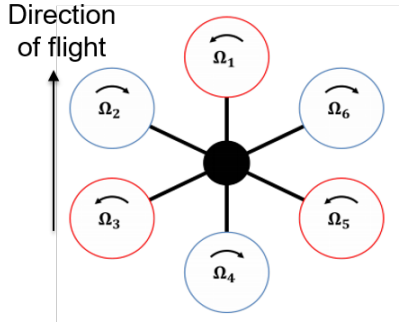


Fig. 3: Schematic diagram of a hexacopter.

This model is used as the main source of simulated data for 5 m/s forward speed under light, moderate, and severe levels of turbulence. Figure 4 shows indicative attitude signals for the cases of healthy flight and front rotor (rotor 1), left-side rotor (rotor 2), and right-side rotor (rotor 6) failures. For the simulation results presented, rotor failure occurs at $t = 10$ s, indicated by the vertical dashed line.

From Figure 4, it may be observed that front rotor (1) failure results in a larger deviation in the pitch attitude than in the case of side rotor (2/6) failure. In the case of front rotor failure, the hexacopter pitches down without any substantial change in roll angle, as the loss of rotor 1 thrust does not significantly affect the aircraft roll equilibrium. However, in the case of side rotor failure both the pitch and roll attitudes change and the roll attitude compensation is observed to be underdamped. The deviation of roll attitude in rotor 2 failure is reversed for rotor 6 failure, because of unbalanced roll moment in rightward and leftward directions, respectively. The heading of the aircraft is observed to deviate in different directions with the failure of the front rotor compared to the side rotor. This is due to the different rotor spin directions of front and side rotors, and consequently the direction of the hub torque generated by each rotor. It should be noted here that the signals show an instantaneous transient response immediately after rotor failure followed by a fault-compensated steady-state response.

In the knowledge-based fault detection and identification algorithm, the above knowledge of configuration of the hexacopter at hand along with principles of force and moment equilibrium has been applied to ascertain how the aircraft roll, pitch, and yaw attitudes will behave under instantaneous loss of thrust and consequent moment imbalance in an event of rotor failure. Therefore, the decision-making follows how the signals initially deviate from their 99% confidence intervals under abrupt rotor 1, 2, and 6 failure, as shown in the enlarged view in Fig. 5. The details of the accuracy and performance of this method can be found in Ref. 3.

III. Fault detection and identification using PILOTS

A. Mode estimation

The motivation to use PILOTS to detect and identify rotor failures in a multicopter is twofold: 1) PILOTS is a high level declarative programming language, making it easier to learn and adapt to, and 2) code and proof generation from PILOTS programs would enable building verified FDI software for safety critical systems like a multicopter.

From release v0.4 of PILOTS [8], [12], a mode estimation method has been introduced which allows us to define multiple error functions and use their relationships to detect erroneous patterns in data streams. Conditions specified in modes are evaluated one by one from the top, and as soon as a condition is satisfied, its associated mode is chosen. If none of the conditions are satisfied, "Unknown"(-1) mode is chosen. Unlike the signature-based mode estimation, the mode estimation using "modes" is purely based on boolean expressions. To adjust the difference between the two mode estimation methods, the threshold values have to be expanded to incorporate the threshold τ.

Using PILOTS, the knowledge based fault detection algorithm can be written in a very straightforward way. A PILOTS program called `Rotor_check_threshold` takes the pitch, roll and yaw attitudes as input streams of data and outputs the same along with the estimated mode of the system every 0.1 seconds. These three attitudes are used as error functions that capture patterns in the input streams. The upper and lower bounds of healthy input streams have been calculated and defined as constants using the 99% confidence intervals for each signal. We use the mode estimation method as there are multiple error functions to examine for fault detection. The error modes define the different modes of the hexacopter due to a fault in rotor 1, 2, or 6. The different modes of the output data stream have been outlined in Table I. The modes include an "Unknown" mode where none of the defined modes correspond to the error signal pattern. In case of multiple matching modes, the first match is the output mode of the system. To change the underlying fault detection model, the error modes have to be calibrated accordingly.

TABLE I: Modes in Pilots state estimation code

State	Mode
Healthy	0
Rotor 1 failure	1
Rotor 2 failure	2
Rotor 6 failure	3
Unknown	-1

B. Threshold check

As studied across various failure detection models (hexacopter rotor failures, pitot tube sensor failures and GPS failures [12]), errors in analytically redundant data streams are often captured using thresholds on the input values. This

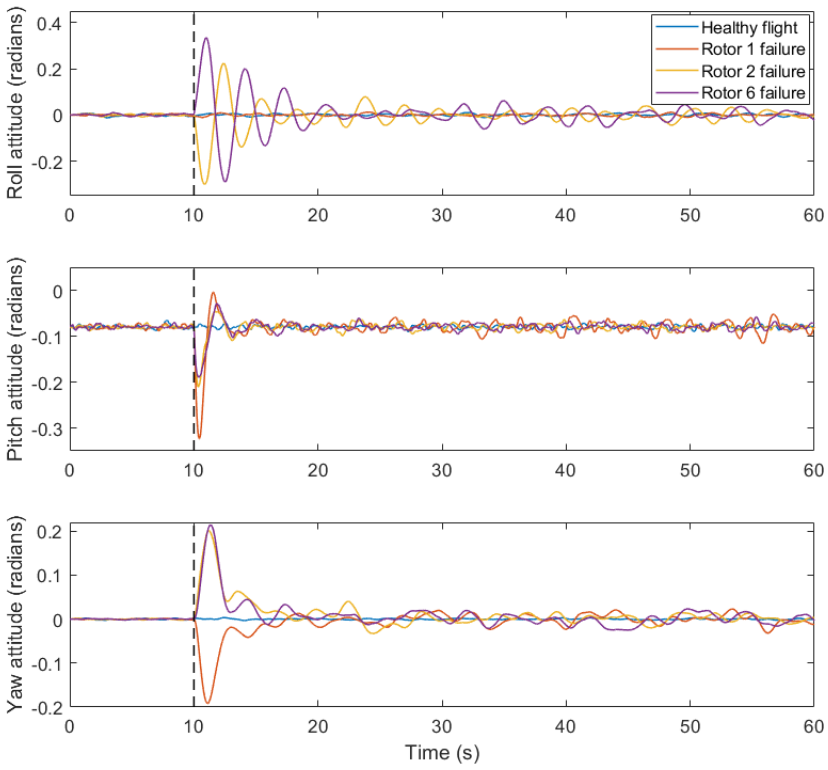


Fig. 4: Roll, pitch, and yaw attitude signals time history for healthy and faulty states of a hexacopter.Reproduced from Ref. [3]

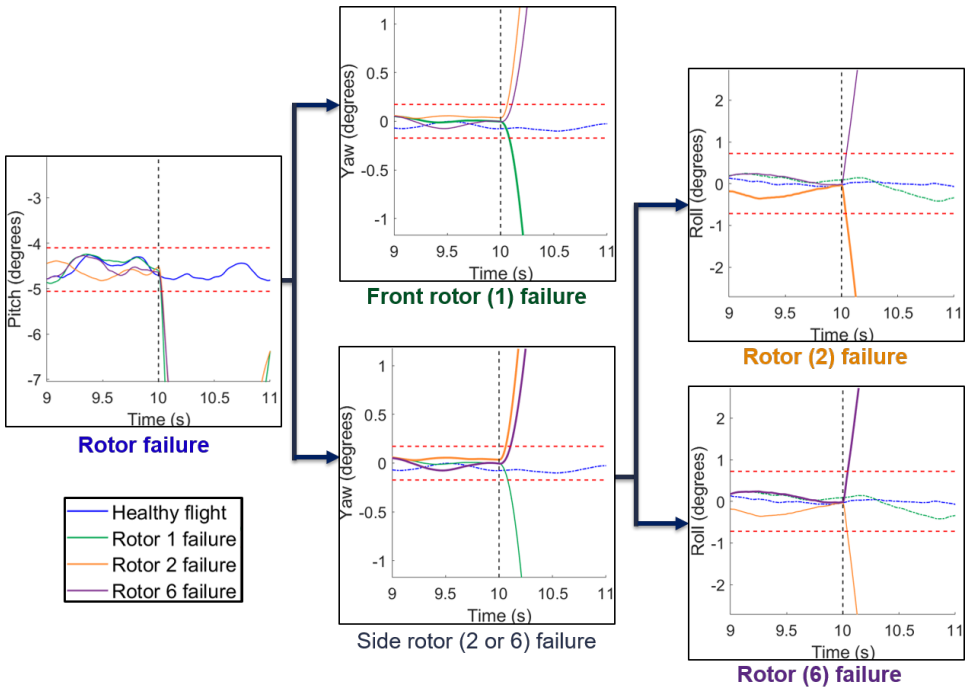


Fig. 5: Indicative results for fault detection and identification by knowledge-based method. The black dashed vertical line indicates the instant of rotor failure and the red dashed horizontal lines indicates the 99% confidence intervals. Reproduced from Ref. [3]

motivated us to extend the PILOTS grammar with a shorthand syntax for checking thresholds. Brackets can be used to limit the thresholds for a particular error function during mode estimation. [ ] and ( ) can be used to denote inclusive and exclusive thresholds respectively on error functions. For example, the PILOTS syntax `e1 in [A_LOW .. A_HIGH)` would translate to the condition `e1 >= A_LOW and e1 < A_HIGH`. We illustrate a trivial example of a state estimation program from two input data streams in Fig. 6.

```
program Threshold_check;
  inputs
      a, b(t) using closest(t);
  constants
    A_LOW = -1;
    A_HIGH = 1;
  outputs
    a,b,mode at every 1 sec;
  errors
    e1: b - a;
  modes
    m0: e1 > A_LOW and e1 < A_HIGH "Normal";
    m1: e1 <= A_LOW "Failure";
end;
```

Fig. 6: Mode estimation using thresholds.

The modes section of the `Threshold_check` program could be rewritten as follows:

```
m0: e1 in (A_LOW .. A_HIGH) "Normal";
m1: e1 in ( .. A_LOW] "Failure";
```

This syntax will not only simplify the process of specifying models with thresholds in error residuals from data streams, it is also an important step to be able to extract postconditions to be proven correct about the mode estimation code. Appendix A lists the updated PILOTS grammar for version 0.7.

IV. Proving the correctness of PILOTS code

We aim to prove the correctness of the PILOTS generated state estimation code with respect to the underlying model used for the development of the logic. The code must exactly replicate the fault detection model to be proven correct. We have developed a *PILOTStoDafny* code generator which produces Dafny code and specifications during compilation of the PILOTS program. For our proof of correctness, we assume the sliding window size ω to be 1. So, the PILOTS program determines the state of the system by looking at the latest stream values only. The verification approach is depicted in Fig. 7. The following sections explain the method to formalize the FDI software in Dafny. We focus on proving the correctness of the mode estimation method of PILOTS generated code.

A. Formalizing the system

In a data streaming application, we have to ascertain that our computing resources are enough to process the frequency of the input data streams. State estimation will be correct only if we consider every data point in the input data streams. In case of limited resources, the application may skip data points

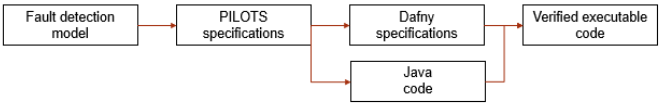


Fig. 7: FDI software verification approach.

potentially compromising correctness. In PILOTS, we specify the frequency of state estimation and output streams in the code. In our proof of correctness we have to establish a set of frequencies T that the PILOTS code can process with the available computing resources. The denotational semantics of a PILOTS program can be represented as a set of three equations as shown in (3). At discretized time t, i represents a function of input data streams, e is an error function that maps input data streams to error streams and f selects a mode m from the error streams.

$$\begin{aligned} i(t) &= \vec{x} \\ e(\vec{x}) &= \vec{e} \\ f(\vec{e}) &= m \end{aligned} \tag{3}$$

The FDI model g is represented as a partial function where an input stream value corresponds to mode m.

$$g(\vec{x}) = m \tag{4}$$

The correctness of the PILOTS code is formally specified in Eq. 5. If $\vec{x}$ is in the model, the output of the program f must match the mode of g, else f returns -1(unknown).

$$\begin{aligned} \forall t \in T.i(t) = \vec{x} \implies & (\vec{x} \in dom(g) \wedge f(e(\vec{x})) = g(\vec{x})) \\ & \vee (\vec{x} \notin dom(g) \wedge f(e(\vec{x})) = -1) \end{aligned} \tag{5}$$

B. Defining the system in Dafny

Data types are created to generalize the input data streams and the error functions in the system. This makes it easier to use variables of these types to formalize and prove the correctness of the program. We continue with the example code from Fig. 6. The input streams and error functions in the program are encoded in Dafny as follows:

```
datatype Data = DataVal (a: real, b: real)
datatype Error = ErrorVal (e1: real)
```

C. State estimation

Dafny has to prove that the estimated state of the system is consistent across the model (g) and the PILOTS program (f). To prove this we have a *modeAnalyzer()* method generated from the PILOTS program. It is replicated from the Java code generated by the PILOTS compiler. A snippet of the Java code generated from (Fig. 6) has been depicted in Fig. 8. The method is implemented in Dafny as illustrated in Fig. 9. The *modeAnalyzer()* method takes as input the value of the error function at a point of time and outputs the mode of the system

(function f). The postcondition of this method ascertains that the method output is consistent with the model. This method and postcondition has been generalized over a window of time to prove the correctness of the complete program.

```
// Errors computation
data.put("e1", data.get("b")-data.get("a"));
LOGGER.fine("Errors: " + "e1=" + data.get("e1"));

// Error detection
int mode = -1;
if ((data.get("e1")>A_LOW) && data.get("e1")<A_HIGH) {
    mode = 0;   // "Normal"
} else if (data.get("e1")<=A_LOW) {
    mode = 1;   // "Failure"
}
LOGGER.fine("Detected: mode=" + mode);

// Data transfer
Date now = getTime();
try {
    sendData(0, data.get("a"), data.get("b"), mode);
    LOGGER.info("Outputs: " + now + " " + "a=" + data.get("a") +
        " " + "b=" + data.get("b") + " " + "mode=" + mode + " ");
} catch (Exception ex) {
    ex.printStackTrace();
}

time += interval;
progressTime(interval);
```

Fig. 8: Java code generated from the threshold check program.

```
datatype Error = ErrorVal (e1: real) //Error functions

//Predicate to check if the mode of a particular input is consistent

predicate modeCorrect(e: Error, A_LOW: real, A_HIGH: real , m: int) {

/*If condition*/
(e.e1>A_LOW && e.e1<A_HIGH ==> m == 0)

    /* Else if condition reached when earlier conditions are not satisfied*/
    && (!(e.e1>A_LOW && e.e1<A_HIGH) && e.e1<=A_LOW ==> m == 1)

        /* Default condition for Unknown mode*/
        && (!(e.e1>A_LOW && e.e1<A_HIGH) && !(e.e1<=A_LOW) ==> m == -1 )
}

/*Method to calculate the mode from a given input value
Input: Value of the error function and thresholds
Output: Mode of the system at that point */
method ModeAnalyzer(e: Error, A_LOW: real, A_HIGH: real ) returns (mode: int)
ensures modeCorrect(e, A_LOW, A_HIGH,mode)
{
    mode := -1;
    if ((e.e1>A_LOW) && e.e1<A_HIGH) {
        mode := 0;  // "Normal"
    }
    else if (e.e1<=A_LOW) {
        mode := 1;  // "Failure"
    }
}
```

Fig. 9: Dafny mode analyzer method generated from the threshold check program.

D. Specification

The constraints of the state estimation model can be formally specified using a predicate in Dafny. A predicate is a method that always returns a boolean value. A predicate *modeCorrect()* specifies each constraint in the model that the state estimation program should adhere to. This predicate inputs a value of the error function and a mode to check whether the mode is consistent with the constraints. The specifications are extracted following the semantics of the $modes$ section of a PILOTS code. For every $mode := Var : Exps\ String$; the $Exps$ should imply the mode variable being set to Var. If neither of the $Exps$ are satisfied, an "Unknown" mode is generated as output. Therefore, the conjunction of all the negated expressions should imply the mode -1. The specification ϕ that we want to prove about the program states that *modeCorrect()* holds true for all input values in the time window start to end. The preconditions state that the end of the time window should be greater than the start and the stream of input data ($dataStream$) should not be null. Each value in the output ($modeStream$) should be consistent with constraints defined in *modeCorrect()*. The specification ϕ is stated by the *SystemCheck()* predicate as follows:

```
forall i :: start <= i < end ==>
 modeCorrect(dataStream[i], LOW, HIGH, modeStream[i])
```

E. Proof

The approach to prove the correctness of the program is inductive. Dafny enables automated inductive theorem proving by iterations or recursion. The invariants defined in the program should be maintained while on entry and after every iteration of the loop. This ascertains that if a property holds at time t, it holds true at time $t+1$. The invariants not only ascertain correctness of the *modeAnalyzer()* method but also guarantees termination of the state estimation loop using the `decreases` keyword.

The postcondition of the *inductiveProof()* method states the correctness property we aim to prove using the `ensures` clause (Fig. 10). It takes as input the stream of input values, the start and end of the window and the thresholds for mode estimation. The goal of the proof is to ascertain the correct mode is estimated for the sequence of input values in the provided time window. The base case of the inductive proof verifies whether the specifications hold true and the system is in "normal" mode when no data is received from the system. The inductive hypothesis states that the specifications hold true at time t i.e. ϕ holds true from $start$ to time t. The loop invariant in the *inductiveProof()* method proves that ϕ holds true from $start$ to time $t+1$. We have a few assumptions for the proofs as stated by the `requires` clause of the method:

- The start and end of the time window should be valid.
- The input data stream should have values until the end of the time window.

In future work, we can weaken these assumptions using the idea of `closest` [11] when PILOTS can even estimate the mode of a system if all data points are not available in the specified time window.

V. Results and Discussion

Based on the data set described in Section II-C, we have used PILOTS to detect failures in any one of the front rotors of a hexacopter. The roll, pitch, and yaw signals for healthy flight under severe turbulence follow a normal distribution [3].

```
/*Method to calculate the mode of every incoming value and
    prove it consistent with existing specifications */

method inductiveProof(start: int, end: int, dataStream: array<Error>,
                      A_LOW: real, A_HIGH: real)
                          returns (modeStream: array<int>)
requires end >= start && start >= 0
requires dataStream.Length > 0 ==> dataStream.Length - 1  >= end
requires dataStream.Length == 0 ==> dataStream.Length >= end
ensures modeStream.Length == dataStream.Length
ensures SystemCheck(start, end, dataStream, modeStream, A_LOW, A_HIGH)
{
modeStream := new int[dataStream.Length];
 if(dataStream.Length == 0) //When do data is available
    {
        assert SystemCheck(0, 0, dataStream, modeStream, A_LOW, A_HIGH);
    }
 else
    {
        var idx := start;
        while(idx < end)
        invariant idx <= end //property is consistent over all values in the time window
        invariant SystemCheck(start, idx, dataStream, modeStream,A_LOW, A_HIGH)
        decreases end - idx
        {
            var m := ModeAnalyzer(dataStream[idx], A_LOW, A_HIGH);
            modeStream[idx] := m;
            idx := idx + 1;
        }
    }
 }
```

Fig. 10: Proving the program correct using induction.

```
program Rotor_check_threshold;
   inputs
    roll,pitch,yaw (t) using closest(t);
   constants
    R_LOW = -0.0153;
    R_HIGH = 0.006;
    P_LOW = -0.0884;
    P_HIGH = -0.0703;
    Y_LOW = -0.0040;
    Y_HIGH = 0.0022;
   outputs
    roll,pitch,yaw,mode at every 0.1 sec;
   errors
    e1: roll;
    e2: pitch;
    e3: yaw;
   modes
    m0: e2 in [P_LOW .. P_HIGH]
        or (e2 in ( .. P_LOW) and e3 in [Y_LOW .. Y_HIGH])
      or (e2 in ( .. P_LOW) and e3 in (Y_HIGH .. )
      and e1 in [R_LOW .. R_HIGH]) "Normal";
    m1: e2 in ( .. P_LOW)
        and e3 in ( .. Y_LOW) "Rotor 1 failure";
    m2: e1 in ( .. R_LOW) and e2 in ( .. P_LOW)
        and e3 in ( Y_HIGH .. ) "Rotor 2 failure";
    m3: e1 in (R_HIGH .. ) and e2 in ( .. P_LOW)
        and e3 in (Y_HIGH .. ) "Rotor 6 failure";
end;
```

Fig. 11: PILOTS program for rotor failure detection using thresholds.

The statistical properties, namely μ and σ^2, the expected value (mean) and variance, respectively of the signals are extracted. Next, the 99% confidence intervals; the upper confidence limit (UCL) and lower confidence limit (LCL) for each signal,$y[t]$ are calculated in radians by Eq. (6):

$$\begin{aligned} &\text{Upper confidence limit} = \mu + 2.576\sigma \qquad \mu = \mathbb{E}\{y[t]\} \\ &\text{Lower confidence limit} = \mu - 2.576\sigma \qquad \sigma^2 = \mathbb{E}\{(y[t]-\mu)^2\} \end{aligned} \tag{6}$$

where, $\mathbb{E}\{\cdot\}$ denotes the statistical expectation.

Error modes have been defined in PILOTS to replicate the above thresholds for healthy data. The threshold syntax introduced in this paper has been used to write the program for rotor failure detection in Fig. 11. The program outputs the three attitudes along with the mode every 0.1 seconds. The window size ω has been set to 1. The frequency of the output data stream is 10Hz. The PILOTS visualizations of the state estimation have been shown in Fig. 12. PILOTS correctly replicates the behavior of the knowledge based FDI model described in Section II-C.

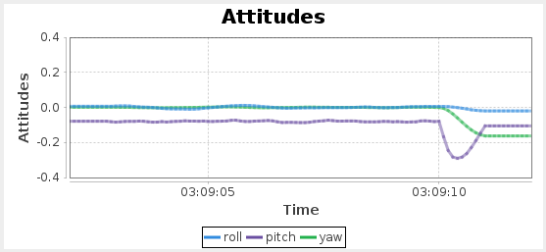
The correctness of the mode estimation model of the `Rotor_check_threshold` program in PILOTS has been modeled in Dafny using approach described in Section IV. The input streams and error functions in the program are encoded in Dafny as follows:

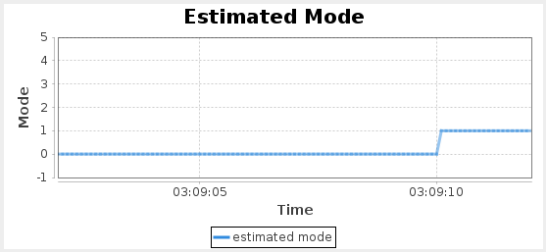
```
datatype Data = DataVal(e1: real, e2: real, e3: real)
datatype Error = ErrorVal(e1: real, e2: real, e3: real)
```

The *modeAnalyzer()* method replicates the conditional logic embedded in the PILOTS mode estimation code. It takes as input the set of error functions and produces an output mode of the system. The *modeCorrect()* predicate contains the set of specifications that the *modeAnalyzer()* method should adhere to. It states the system invariants that must hold true at every point of time during the mode estimation time window. The specifications are extracted following the semantics of the `modes` section of a PILOTS program. In PILOTS, the conditions are evaluated from the top (i.e. $m0$), and the first condition that is satisfied is the output mode at that point. The final specification states the conjunction of all the negated conditions, which implies that if none of the conditions are satisfied, the output mode is Unknown (i.e. -1). This set of specifications in *modeCorrect()* is iteratively invoked in the *inductiveProof()* method which further verifies that the conditions hold true during the complete time window. The specifications are satisfied by the PILOTS code only when the frequency of the input data stream is in the set of frequencies that the computing resources support and complete data is available for the time frame. The Dafny *modeCorrect()* method generated from the `Rotor_check_threshold` program is shown in Fig. 13. Table II summarizes the verification results for various time windows and data availability from the hexacopter.

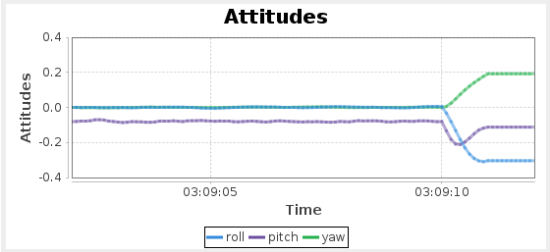
TABLE II: Verification results for rotor FDI.

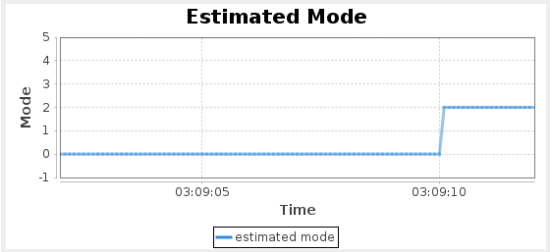
Start (s)	End (s)	Frequency (Hz)	Data points	Verification
0	5	1	5	Verified
0	5	10	50	Verified
0	10	1	10	Verified
0	10	10	100	Verified



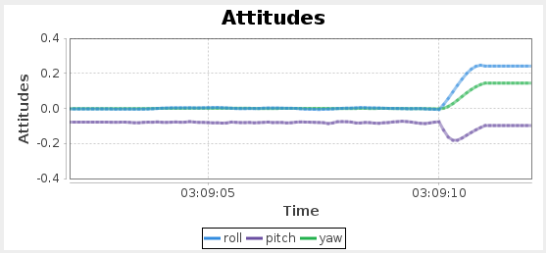


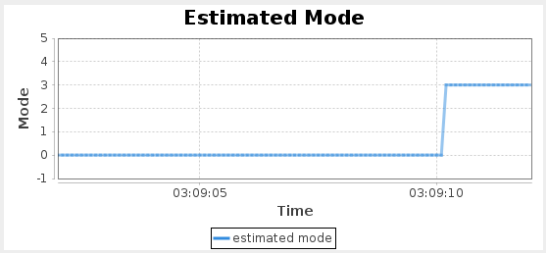
(a) Rotor 1 failure and normal mode.



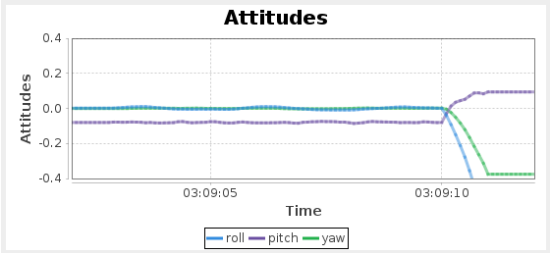


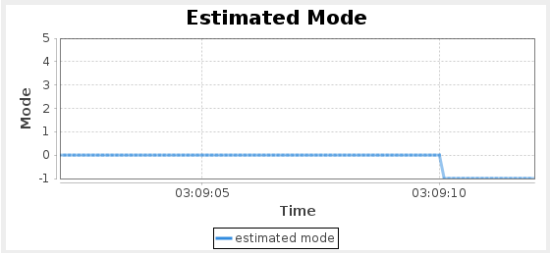
(b) Rotor 2 failure.





(c) Rotor 6 failure.





(d) Normal and Unknown mode.

Fig. 12: Rotor FDI output visualized using PILOTS.

VI. Related work

Previous work in formalizing streaming applications and FDI software have introduced several languages and proof systems for verification. Proving the correctness of streaming applications has been explored in [19] and [20]. The authors have presented a proof system that can verify computations performed on outsourced streaming data. Bozzano et al. [21], [22] have introduced a pattern based language built on the semantics of Temporal Epistemic Logic [23] to formally specify and verify alarms in FDI. Our work focuses on replicating FDI models using high level declarative syntax which would require minimum technical expertise to use. The approach used in this paper to detect and identify rotor failures works only with sharp transients in the input signals. There are other sophisticated algorithms which can detect rotor failures from steady signals too [3]. We focus on proving the correctness of the language instead of the FDI model using established logic systems and proof assistants. The power of automated proof assistants like Dafny has been explored in work like [24] and [25], where complex distributed systems have been modeled and proven correct using Dafny predicates and lemmas.

VII. Conclusion and Future work

This paper is an initial step for proving the correctness of the PILOTS compiler, which can be deployed in various domains like aerospace, healthcare, etc. The verification procedure has been carried out with assumptions and considerations of a small kernel of the language. The correctness of the mode estimation method in PILOTS is proven. Both Java and Dafny code and specifications are generated by the code/proof generators at compile time. The PILOTS mode estimation semantics has been modeled using Hoare logic and proven to be correct using Dafny specifications. Furthermore, the PILOTS syntax has been updated to ensure easier implementation of threshold checks which are a popular requirement in FDI systems.

```
datatype Error = ErrorVal (e1: real, e2: real, e3:real) //Error functions

//Predicate to check if the mode of a particular input is consistent
predicate modeCorrect(e: Error, R_LOW: real, R_HIGH: real, P_LOW: real,
        P_HIGH: real, Y_LOW: real, Y_HIGH: real , m: int) {

/*If condition*/
((e.e2>=P_LOW && e.e2<=P_HIGH) ||
    (e.e2<P_LOW && e.e3>=Y_LOW && e.e3<=Y_HIGH) ||
        (e.e2<P_LOW && e.e3>Y_HIGH && e.e1>=R_LOW && e.e1<=R_HIGH) ==> m == 0)

    /* Else if condition reached when earlier conditions are not satisfied*/
     && (!((e.e2>=P_LOW && e.e2<=P_HIGH) ||
        (e.e2<P_LOW && e.e3>=Y_LOW && e.e3<=Y_HIGH) ||
            (e.e2<P_LOW && e.e3>Y_HIGH && e.e1>=R_LOW && e.e1<=R_HIGH))
                && e.e2<P_LOW && e.e3<Y_LOW ==> m == 1)

        /* Else if condition reached when earlier conditions are not satisfied*/
         && (!((e.e2>=P_LOW && e.e2<=P_HIGH) ||
                (e.e2<P_LOW && e.e3>=Y_LOW && e.e3<=Y_HIGH) ||
                    (e.e2<P_LOW && e.e3>Y_HIGH && e.e1>=R_LOW && e.e1<=R_HIGH))
                        && !(e.e2<P_LOW && e.e3<Y_LOW) && e.e2<P_LOW && e.e3>Y_HIGH
                            && e.e1<R_LOW ==> m == 2)

            /* Else if condition reached when earlier conditions are not satisfied*/
             && (!((e.e2>=P_LOW && e.e2<=P_HIGH) ||
                (e.e2<P_LOW && e.e3>=Y_LOW && e.e3<=Y_HIGH) ||
                    (e.e2<P_LOW && e.e3>Y_HIGH && e.e1>=R_LOW && e.e1<=R_HIGH))
                        && !(e.e2<P_LOW && e.e3<Y_LOW)
                            && !(e.e2<P_LOW && e.e3>Y_HIGH && e.e1<R_LOW)
                                && e.e2<P_LOW && e.e3>Y_HIGH && e.e1>R_HIGH ==> m == 3)

                /* Default condition for Unknown mode*/
                 && (!((e.e2>=P_LOW && e.e2<=P_HIGH) ||
                    (e.e2<P_LOW && e.e3>=Y_LOW && e.e3<=Y_HIGH) ||
                        (e.e2<P_LOW && e.e3>Y_HIGH && e.e1>=R_LOW && e.e1<=R_HIGH))
                            && !(e.e2<P_LOW && e.e3<Y_LOW)
                                && !(e.e2<P_LOW && e.e3>Y_HIGH && e.e1<R_LOW)
                                    && !(e.e2<P_LOW && e.e3>Y_HIGH && e.e1>R_HIGH) ==> m == -1 )
}
```

Fig. 13: Dafny code generated from rotor state estimation program.

PILOTS has been used to detect and identify rotor failures in a hexacopter with input data streams at a frequency of 10 Hz. The application will correctly output the state of the system provided the computing resources are sufficient to process data with the required frequency of the incoming data streams.

Future work will include proving the correctness of PILOTS code when the window size ω which is evaluated during mode estimation is greater than 1 and the input data frequency is higher than what can be analyzed with the existing computing resources. The next step would be to formalize error signatures in a PILOTS program and model the same in Hoare logic to prove their functional correctness. This would enable us to produce more generalized proofs of verification of PILOTS code and create verified state estimation software for a wider range of applications.

Acknowledgements

This research was partially supported by the National Science Foundation (NSF), Grant No. – CNS-1816307 and the Air Force Office of Scientific Research (AFOSR), DDDAS Grant No. – FA9550-19-1-0054. The authors would like to acknowledge the authors of previous PILOTS papers: Shigeru Imai, Erik Blasch, Richard Klockowski, Alessandro Galli, Frederick Lee, and Colin Rice.

References

[1] Afef Fekih. Fault diagnosis and fault tolerant control design for aerospace systems: A bibliographical review. In *2014 American Control Conference*, pages 1286–1291, 2014.

[2] Xin Qi, Didier Theillol, Jutong Qi, Youmin Zhang, and Jianda Han. A Literature Review on Fault Diagnosis Methods for Manned and Unmanned Helicopters. In *2013 International Conference on Unmanned Aircraft Systems*, May 2013.

[3] Airin Dutta, Michael E McKay, Fotis Kopsaftopoulos, and Farhan Gandhi. Fault Detection and Identification for Multirotor Aircraft by Data-Driven and Statistical Learning Methods. In *2019 AIAA/IEEE Electric Aircraft Technologies Symposium (EATS)*, pages 1–18. IEEE, 2019.

[4] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[5] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[7] Shigeru Imai, Erik Blasch, Alessandro Galli, Wennan Zhu, Frederick Lee, and Carlos A Varela. Airplane flight safety using error-tolerant data stream processing. *IEEE Aerospace and Electronic Systems Magazine*, 32(4):4–17, 2017.

[8] Shigeru Imai, Sida Chen, Wennan Zhu, and Carlos A Varela. Dynamic data-driven learning for self-healing avionics. *Cluster Computing*, 22(1):2187–2210, 2019.

[9] Shigeru Imai and Carlos A Varela. A programming model for spatio-temporal data streaming applications. *Procedia Computer Science*, 9:1139–1148, 2012.

[10] E. Chow and A. Willsky. Analytical redundancy and the design of robust failure detection systems. *IEEE Transactions on Automatic Control*, 29(7):603–614, 1984.

[11] Shigeru Imai, Richard Klockowski, and Carlos A Varela. Self-healing spatio-temporal data streams using error signatures. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 957–964. IEEE, 2013.

[12] Shigeru Imai, Frederick Hole, and Carlos A Varela. Self-healing data streams using multiple models of analytical redundancy. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE, 2019.

[13] Bureau d'Enquêtes et d'Analyses et al. Final report on the accident on 1st June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro–Paris. *Paris: BEA*, 2012.

[14] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[15] K Rustan M Leino. This is Boogie 2. *Manuscript KRML*, 178(131):9, 2008.

[16] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[17] Michael McKay, Robert Niemiec, and Farhan Gandhi. Post-Rotor-Failure-Performance of a Feedback Controller for a Hexacopter. In *American Helicopter Society 74th Annual Forum, Phoenix, AZ*. AHS, May 2018.

[18] Teuku Mohd Ichwanul Hakim and Ony Arifianto. Implementation of Dryden Continuous Turbulence Model into Simulink for LSA-02 Flight Test Simulation. In *Journal of Physics: Conference Series 1005(2018) 012017*, August 2018.

[19] Graham Cormode, Justin Thaler, and Ke Yi. Verifying computations with streaming interactive proofs. *arXiv preprint arXiv:1109.6882*, 2011.

[20] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 90–112, 2012.

[21] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal Specification and Synthesis of FDI through an Example. In *Workshop on Principles of Diagnosis (DX'13)*, pages 174–179, 2013.

[22] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal design of fault detection and identification components using temporal epistemic logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 326–340. Springer, 2014.

[23] Joseph Y Halpern and Moshe Y Vardi. The complexity of reasoning about knowledge and time. I. Lower bounds. *Journal of Computer and System Sciences*, 38(1):195–237, 1989.

[24] K Rustan M Leino. Modeling Concurrency in Dafny. In *School on Engineering Trustworthy Software Systems*, pages 115–142. Springer, 2017.

[25] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.

APPENDIX A
PILOTS GRAMMAR DESIGN

The grammar of the PILOTS programming language v0.7 has been updated to support inclusive and exclusive threshold checks as shown in Figure 14.

```
Program      ::= program Var;
                   constants Constants
                   inputs Inputs
                   outputs Outputs
                   [errors Errors]
                   [signatures Signatures]
                   [modes Modes]
                 end
Constants    ::= [(Constant;)* Constant];
Constant     ::= Var = Exp;
Inputs       ::= [(Input;)* Input];
Input        ::= Vars: Dim using Methods;
Outputs      ::= [(Output;)* Output];
Output       ::= Vars: Exps at every Time
                   [when (Var | Exp) [Integer times]];
Errors       ::= [(Error;)* Error];
Error        ::= Vars: Exps;
Signatures   ::= [(Signature;)* Signature];
Signature    ::= Var [Const]: Var = Exps String [Estimate];
Estimate     ::= Estimate Var = Exp;
Modes        ::= [(Mode;)* Mode];
Mode         ::= Var: Exps String [Estimate];
Dim          ::= '(t)' | '(x,t)' | '(x,y,t)' | '(x,y,z,t)'
Methods      ::= Method | Method, Methods
Method       ::= (closest | euclidean | interpolate | model)
                   '(' Exps ')'
Time         ::= Number (msec | sec | min | hour)
Exps         ::= Exp | Exp, Exps
Exp          ::= Func (Exps) | Exp Func Exp | IntervalExp |
                   '(' Exp ')' | Value
IntervalExp  ::= Value in ( '[' | '(' ) [Exp] .. [Exp] ( ']' | ')' )
Func         =   { +, -, *, /, >, <, != , ==, and, or, xor, sqrt, sin, cos, abs, ... }
Value        ::= Number | Var
Number       ::= Sign Digits | Sign Digits'.'Digits
Sign         ::= '+' | '-' | ''
Integer      ::= Sign Digits
Digits       ::= Digit | Digits Digit
Digit        =   { 0, 1, 2, ...,9 }
Vars         ::= Var | Var, Vars
Var          =   { a, b, c, ... }
String       =   { "a", "b", "c", ... }
```

Fig. 14: PILOTS v0.7 grammar with threshold syntax incorporated.