Formal Verification of a Redundancy Management Logic

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

MIS AB V P & Saarang S
(112001026 & 112001035)

Guided by **Dr. Jasine Babu**

COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**Formal Verification of a Redundancy Management Logic**" is a bonafide work of **MIS AB V P & Saarang S (Roll No. 112001026 & 112001035)**, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr. Jasine Babu

Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

We wish to express our sincere gratitude to Dr. Jasine Babu, our project guide, for her constant support and guidance throughout this project. Her insightful feedback and expertise were of great value in the direction of the project. We are highly grateful for her mentorship and for the knowledge she bestowed upon us

We wish to express our deepest appreciation to Dr. Piyush P. Kurur for generously sharing his knowledge and expertise in Coq. His readiness to clarify our doubts and provide guidance whenever we encountered obstacles was invaluable for our progress.

We express our heartfelt thanks to Dr. Arif Ali A P for his active participation in discussions and his unfailing support throughout the project. His presence in every meeting and willingness to assist with any aspect of the work were truly appreciated.

We wish to express our heartfelt appreciation to Deepa Sara John from ISRO for providing us with the detailed project description and always being available to clear our doubts.

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Overview of the Report 3

1.3 System Description of a TMR System 3

1.3.1 Functionality of Components 6

1.4 Review of Prior Works 7

2 Specification of a TMR system **9**

2.1 Component Specification 9

2.2 Specification of States 10

2.3 Specification of the Transition Function 11

2.4 Verified Properties of the System 11

3 Specification of a Generalized Redundancy Management System **13**

3.1 Generalized Specification 13

3.1.1 Modified Specification of States 13

3.1.2 Modified Specification of the Transition Function 15

3.2 Sensor Selection 16

3.2.1 Requirement 16

3.2.2 Single Fault Assumption 16

3.2.3 Sensor State Transition Logic 16

4 Specification and Verification of a Miscomparison Logic **19**

4.1 Single Fault Assumption 20

4.2 Requirements 20

4.3 Miscomparison Logic 21

4.4 Implementation 22

4.4.1 Initial Attempt - Using Decision Trees 22

4.4.2 Final Implementation 23

5 Conclusion and Future Work **27**

References **29**

Chapter 1

Introduction

Formal verification consists of various techniques for mathematical modeling of systems and formally proving the functional correctness and other desirable properties of those systems. Unlike test-case driven verification which involves running the system with different test inputs to detect defects, formal verification provides a higher level of assurance. This makes formal verification particularly relevant in domains where reliability and safety are of the most importance, such as aerospace, healthcare, and financial systems.

A few commonly used classical formal verification methods are,

- **Model Checking**: Model checking is an automatic method used for verification of the correctness of a model of any concurrent finite systems. It is used to verify the given specification of a system holds specific properties in all states reachable from the initial state.
- **Runtime Verification**: The Runtime Verification checks whether a system is satisfying or violating certain properties during runtime.
- **Higher Order Theorem Provers**: Higher order theorem provers use proof tech-

niques to mathematically prove the correctness of a system and prove it meets its specifications under all conditions.

Computer-aided formal verification and synthesis of a verified system using a theorem prover involves 3 major steps.

- **Specifying the system**: Precisely defining the system of software to be verified. This involves creating a formal, mathematical representation of the system's behavior, structure, and properties.
- **Prove the functional correctness and guarantees**: This involves applying formal methods, logic, and proof techniques to ensure that the system specification adheres to the functional requirements.
- **Synthesis of the verified system**: This step is to translate the formally verified system specification into equivalent code in a target language (which could be hardware description languages like Verilog or VHDL or a formally specified programming language).

1.1 Problem Statement

The objective of our project is to build a formally verified redundancy management logic for an avionics system. We are trying to model the system based on an informal specification of the requirements provided by a team from ISRO for a triple modular redundant (TMR) system, which is described in Section 1.3. Our goal is to design a verified redundancy management system for n factor redundancy, for $n \geq 3$. For this project we are using Coq [1] theorem prover for the formal verification of our system.

1.2 Overview of the Report

Section 1.3 of this chapter gives more details of the TMR system. Section 1.4 of this chapter reviews prior work done on a TMR system using Model Checking approach. Chapter 2 describes the specification of a TMR system done in the *Coq* environment. Chapter 3 discusses a more generalized system with n sensors instead of 3. Chapter 4 focuses on the specification and verification of a miscomparison logic used in the voting system. Chapter 5 concludes the project and future work.

1.3 System Description of a TMR System

A single system of three gyroscopes will measure the rate of change of angle in X, Y, and Z axes, whose measurements are converted to the range of discrete finite range. A sensor acquisition electronics associated with such a system is used to gather the rate information from each sensor.

To ensure reliability, 3 identical systems as described above are placed in the avionics deck. An onboard computer (OBC) selects one sensor out of 3 sensors for each axes for the control requirements of the navigation guidance. For each axis, to choose one of the 3 sensors, a sensor voting logic should be applied. The choice is to be made based on i) the health of the communication link ii) acquisition electronics health iii) individual sensor health and iv) a data consistency check done on the measured signals from the three sensors. The OBC has to detect and isolate failures at the system and sensor levels. However, the ground truth, mentioned as *ground_truth* in rest of the report, of the actual rate of change of angle is unknown.

Valid output should be selected from a single sensor in any cycle. There is a specific priority order in which the sensor needs to be chosen, which is pre-defined. Switching of sensors should be minimized. Sufficient delay has to be provided before permanent isolation of a sensor channel (based on communication, acquisition health, and parity checks) so that

the voter can tolerate false alarms and transients.

Even if a sensor is faulty in the current cycle, the sensor may be used for output generation. A permanently isolated sensor is never used for output generation or in any voting logic. Parity check computes the deviation between the sensors in each axis. The moving average of 5 samples of the deviations is checked against a threshold to detect failure. The concept of threshold is to account for the noise in readings of sensors. It is assumed that two failures cannot occur simultaneously.

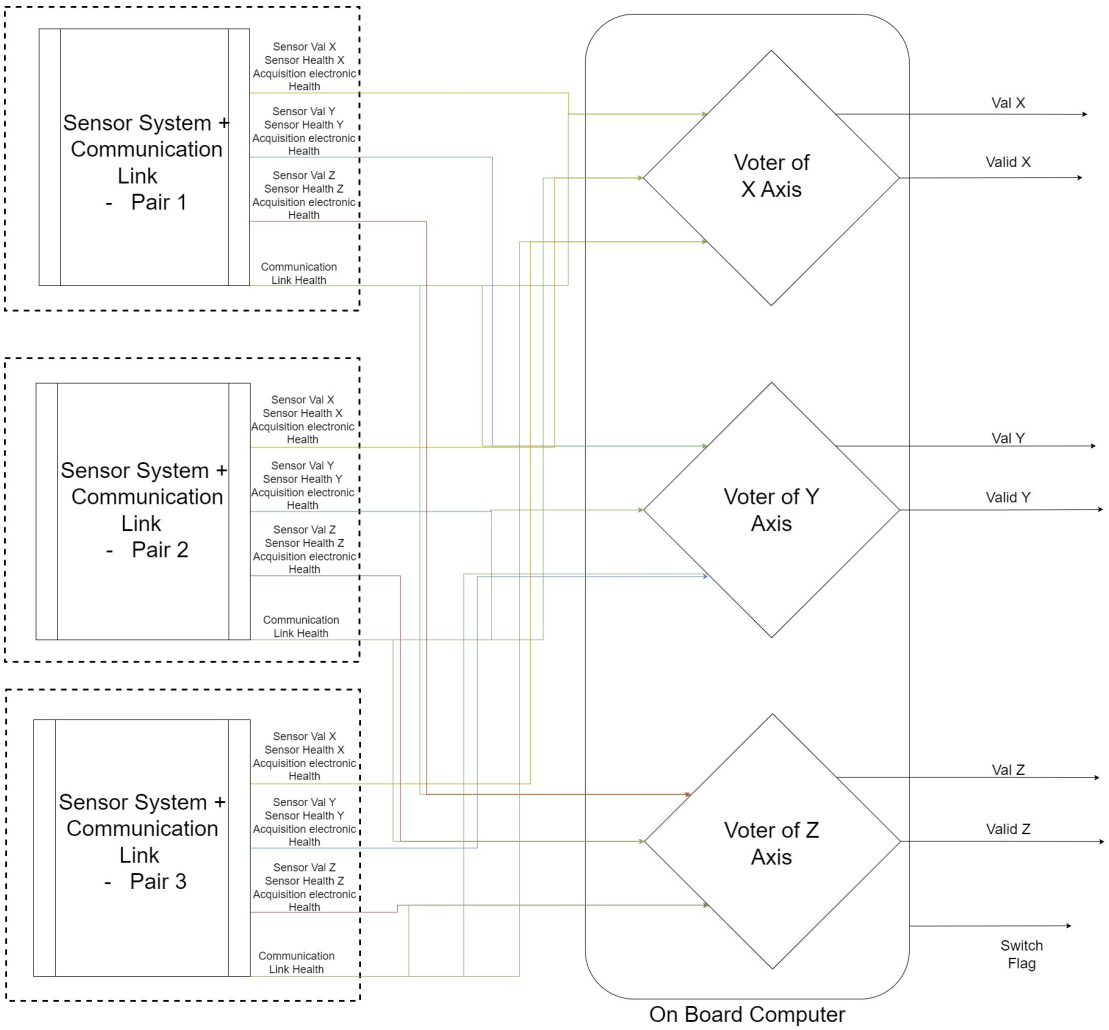

Fig. 1.1 Example of a full system with sensors, acquisition electronics, OBC, and voter when n=3.

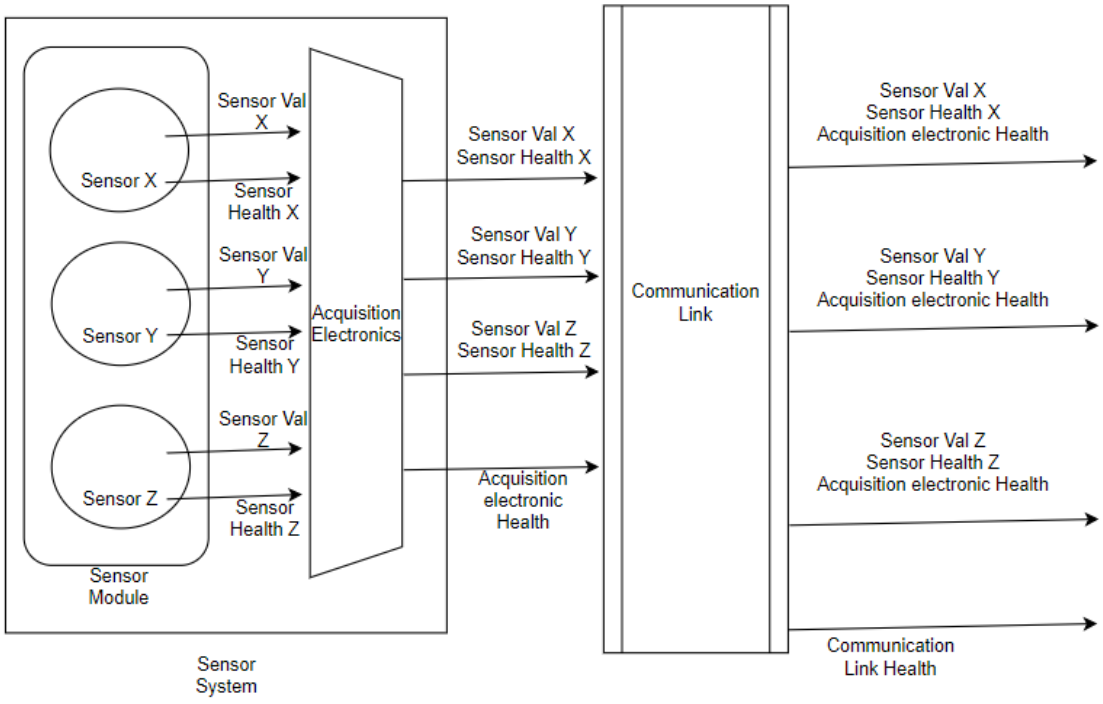

Fig. 1.2 A sensor and acquisition electronics unit.

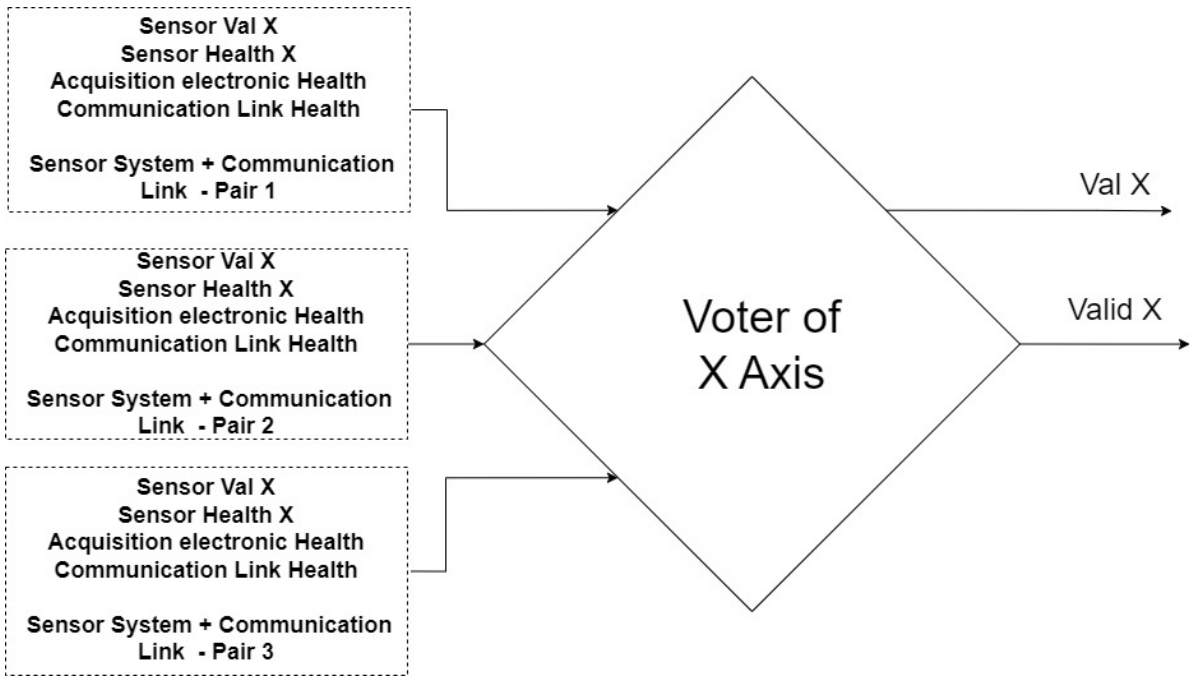

Fig. 1.3 Example of Voter for an axis when n=3.

1.3.1 Functionality of Components

In this section, we shall briefly discuss the functionality of each of the major components in our system.

A **Sensor** system contains 3 sensors to measure the rate of change of angle in X, Y, and Z direction. The rate measured by the 3 sensors for each axis is passed to the accompanying acquisition electronics as shown in Fig. 1.2. The health status of the 3 sensors is passed to the communication link. A 'Not OK' (or negative) health flag from the sensor denotes that the sensor identifies itself as malfunctioning and its measured values may not be valid.

Acquisition electronics captures rate measurement values from the sensor in each axis. The acquisition electronics also has its health status. It passes the signal from these three sensors to the communication link (as shown in Fig. 1.2) which acts as a signal bus between sensor modules and the onboard computer. A 'Not OK' Health flag value from the acquisition electronics denotes that the acquisition electronics identifies itself as malfunctioning and the values it provides may not be valid.

Communication link relays the information it receives to the OBC (as shown in Fig. 1.1) with an addition of its health flag. Similar to earlier health flags, a 'Not OK' Health flag value indicates that it identifies itself as malfunctioning, and the values it passes to the OBC may not be valid.

The **Voter** for an axis receives reading from 3 sensors along its axis and the health status of all the components, namely sensor, acquisition electronics, and communication link as shown in Fig. 1.3. It uses a voting logic to find the sensor for the current cycle and to detect and isolate failures of sensors.

There are 3 redundant such systems involving sensors, acquisition electronics, and communication links, all connected to the OBC. There are two OBCs in our configuration. Both OBCs acquire the sensor data from the 3 systems. The second OBC has its own communication link to transfer information to it. However, only one will be actively commanding the actuator based on the sensor values. If the controlling computer sees that

there is a communication failure with all 3 input systems, then it will raise the OBC switch flag indicating that the failure is due to the computer. The switch flag will be used by the computer to take charge of the active control of the actuators. An OBC onces switched will not be reintroduced again.

1.4 Review of Prior Works

A sensor voter algorithm of 3 sets of sensors used in flight controlling of spacecraft was done by Brown et al. [2]. They use NuSMV [3] model checker for their implementation. Their NuSMV implementation is not available publicly. Later, Sneha Chakraborty, a former PG student of IIT Palakkad, did an independent implementation of a similar system [4]. This work was done with a single axis in consideration, planning to generalize it to three axes later. Tools including TLA+ [5] and NuSMV were used to model the sensor - voter algorithm, the properties as constraints, and check if the model holds all the properties. The model had three modules I)*realworld*, II) *sensor*, and III)*main*. *Realworld* module produced arbitrary signal values in the range of specified MIN and MAX values. The *sensor* module simulated the function of a real sensor. It took the values from the *realworld* module and passed on the values adding noise to it along with 'fault' values indicating the reliability of the sensor output value like a real sensor. The *main* module acted as the voter which had functions of sensor isolation caused by persistent incorrect or fault values from any sensor, output value based on the voter logic and sensor output values, and output valid flag value indicating the reliability of the output value.

Model checking tools work by exploration of states (states are the unique combination of values of variables in the model) which can be reached from starting states following the possible transitions specified in the model and check if the properties we stated as constraints are being satisfied in each of the states. In case any reachable state does not satisfy a property being checked, the model-checking tools will often provide a trace of such an execution sequence.

A problem with the model-checking approach was the exponential growth of the number of states for a linear increase in the number of variables in a single system. This will lead to a state explosion causing memory overflow in the computation machine for models with a higher number of variables. This was the major reason for encountering scalability issues in the previous work [4], when trying to extend the system to measure 3-dimensional data, instead of 1-dimensional data. To address this issue, we decided to adopt the approach of proof techniques using the *Coq* [1] theorem prover for verifying the system and voting algorithm in our work on *Formal Verification of a Redundancy Management Logic*.

Chapter 2

Specification of a TMR system

In phase I of this project we focused on specifying a triple redundancy management system using Coq. We first specified key components of the fault-tolerant sensor system. Then we defined the states and the transition function of the system. We also proved some sample properties of the system.

2.1 Component Specification

The system has the following components.

- **Sensor:** Represented using the *sensor* type, which indicated the sensor's health status (correct or unhealthy) and its measured rate value.
- **Acquisition Electronics:** Modeled with the *acquisition* type, reflecting its health status and the received sensor data.
- **Communication Link:** Defined by the *communication* type, indicating its health status and the data received from the acquisition electronics.
- **Voter:** Implemented using the *voter* function, which analyzed sensor readings and health information from all components to select a sensor reading.

2.2 Specification of States

To represent the system states, we used the following specifications.

- **Axis:** The *axis* type identified the three axes (x, y, z) along which measurements were taken.
- **Current Sensor:** The *currSensor* type stored information about the currently active sensor on each axis, including its ID.
- **Isolation Status:** Some other types tracked the isolation status of different components:
  - **IsolatedSensors:** Indicated whether individual sensors were isolated on each axis.
  - **AllIsolatedSensors:** Represented the isolation status of all sensors across all axes.
  - **AcquisitionIsolated:** Reflected the isolation status of the three acquisition electronics units.
  - **CommIsolated:** Represented the isolation status of the three communication links.
- **Cumulative Health Count:** These types kept track of the number of consecutive health failures for each component:
  - **SensorCumulativeHealthFailure:** Stored the cumulative health failure count for sensors on each axis.
  - **AllSensorCumulativeHealthFailure:** Represented the cumulative health failure counts for all sensors.
  - **CumulativeHealthFailureAcq:** Stored the cumulative health failure count for the acquisition electronics units.

  - **CumulativeHealthFailureComm:** Represented the cumulative health failure count for the communication links.

- **Miscomparison Count:**
  - **MiscomparisonCount:** Tracked the number of miscomparisons for each sensor on each axis.
  - **AllMiscomparisonCount:** Represented the miscomparison counts for all sensors.

- **Switch Flag:** The $OBCFlag$ type indicated the status of the On-Board Computer (OBC) switch flag, which was raised in case of communication failures.

The implementation code is found in `Phase I/state.v` file in the repository [6].

2.3 Specification of the Transition Function

A transition function was defined to update the state based on input values. The transition function had various sub-functions which are described below.

- $SensorIsolation$ function determined the isolation status of sensors based on their cumulative health failures.
- $ManageOBCSwitchFlag$ function raised the OBC switch flag when all three communication links experienced failure.
- $UpdateMisCompCount$ function updated the miscomparison count from the sensor readings.

2.4 Verified Properties of the System

Two properties were proven:

- **Raised OBC Flag Persists:** Once the OBC switch flag was raised, it remained raised indefinitely. This ensures that an abandoned OBC is never used again.
- **Isolation Triggered by Counts:** If cumulative Health failure or miscomparison count for a component exceeds a predefined threshold (5 in our case), the component is isolated.

The implementation code is found in `Phase I/transitionFunctions.v` file in the repository [6].

Chapter 3

Specification of a Generalized Redundancy Management System

As per the inputs from the project review panel for Phase I, in Phase II of the project, we decided to develop a generalized redundancy management system for n sensors rather than for three. We also re-designed our system specifications and miscomparison function for a n sensor system.

3.1 Generalized Specification

We shall briefly go over the modified states and transition functions to accommodate for the n sensor system.

3.1.1 Modified Specification of States

- **Current Sensors:** The *currSensor* type captures information about the currently selected sensor for each axis (x, y, z). It uses a finite set to represent the possible sensor indices, instead of natural numbers.
- **Isolation Status:** This group of types deals with the isolation status of various

components:

  - **IsolatedSensors:** Represents the isolation status of each sensor along a specific axis. It uses a vector of booleans, where each element corresponds to a sensor and its isolation state (true for isolated, false for not isolated).
  - **AllIsolatedSensors:** Combines $IsolatedSensors$ for all three axes (x, y, z) to represent the isolation status of all sensors in the system.
  - **AcquisitionIsolated:** Represents the isolation status of each acquisition electronics unit using a vector of booleans.
  - **CommIsolated:** Represents the isolation status of each communication link using a vector of booleans.

- **Cumulative Health Count:** These types keep track of how many times a component has reported an unhealthy status:

  - **SensorCumulativeHealthFailure:** Stores the cumulative health failure count for each sensor along a specific axis using a vector of natural numbers.
  - **AllSensorCumulativeHealthFailure:** Combines $SensorCumulativeHealthFailure$ for all three axes to represent the health failure counts of all sensors.
  - **CumulativeHealthFailureAcq:** Stores the cumulative health failure counts for the acquisition electronics units using a vector of natural numbers.
  - **CumulativeHealthFailureComm:** Stores the cumulative health failure counts for the communication links using a vector of natural numbers.

- **Miscomparison Count**

  - **MiscomparisonCount:** Tracked the number of miscomparisons for each sensor on each axis. It uses a vector of natural numbers, where each element corresponds to a sensor and its miscomparison count.

– **AllMiscomparisonCount:** Combines MiscomparisonCount for all axes to provide an overall view of miscomparison counts across all the axes.

The implementation code is found in `Phase II/generalized/state_n.v` file in the repository [6].

3.1.2 Modified Specification of the Transition Function

The transition function was also changed to update the modified state based on input values. Many of the sub-functions of the transition function were modified and they are described below.

- $ManageCurSensorAx$ function updates the currently selected sensor for an axis based on the output of the voter function.
- $SingleAxisSensorIsolations$ function updates the isolation status of sensors along a single axis by considering their miscomparison counts, cumulative health failure counts, and current isolation states.
- $SensorIsolation$ function updates the isolation status for all sensors across all axes by utilizing $SingleAxisSensorIsolations$.
- $ManageOBCSwitchFlag$ function manages the OBC switch flag. If all communication links have failed (indicated by $CommIsolated$), it raises the flag. Otherwise, it keeps the flag lowered.

Overall, these functions demonstrate the transition logic for managing sensor selection, isolation, and OBC switching in a system with n sensors.

The implementation code is found in `Phase II/generalized/transitionFunctions_n.v` folder in the repository [6].

3.2 Sensor Selection

3.2.1 Requirement

If the number of isolated sensors is less than q, then the sensor selection logic should identify a non-isolated sensor whose reading is within a guaranteed threshold from the $ground\_truth$. Here q is a value that we propose based on our voting logic and the threshold depends on the maximum anticipated noise level.

3.2.2 Single Fault Assumption

The *Single Fault Assumption* is that in any given cycle, among the non-isolated and healthy sensors, there is atmost one sensor that is more than δ away from the $ground\_truth$.

3.2.3 Sensor State Transition Logic

Each sensor in the system can be in one of the following three states:

- **Normal:** The sensor is functioning correctly and its reading is considered correct.
- **Transient:** The sensor has a fault (either a "Not OK" health flag or disagreement with other non-isolated sensors) in the current cycle, but it is not yet considered permanently faulty. This state allows for tolerating glitches or false alarms.
- **Isolated:** The sensor is considered permanently faulty and is excluded from the voting logic and agreement checks.

State Transitions

- **Initial State:** All sensors start in the Normal state.
- **Normal to Transient:** If a sensor in the **Normal** state exhibits a fault in the current cycle, it changed to the **Transient** state.

- **Transient to Isolated:** If a sensor in the **Transient** state continues to exhibit faults for iso_thresh consecutive cycles, it transitions to the **Isolated** state.
- **Transient to Normal:** If a sensor in the **Transient** state does not exhibit any fault in the current cycle, it transitions back to the **Normal** state.
- **Isolated to Isolated:** Once a sensor is in the **Isolated** state, it remains in that state permanently and is not considered for voting or agreement checks.

Sensor Selection Logic

Sensor selection is critical for selecting a reliable reading from the n redundant sensors. The sensor selection algorithm considers the sensor's health, isolation status, and reading to find the most reliable reading. The algorithm uses a miscomparison logic to detect any erroneous values from non-isolated healthy sensors. Our algorithm for sensor selection is described below.

Let num_iso be the number of isolated sensors in a cycle and q as defined in the specification. num_iso is updated every cycle before applying the voting logic.

- If $num\_iso \leq q$: Any one of the following changes can happen to the state of a sensor.
  - normal to transient: If the current sensor is in the transient state, then the value selected is the previous valid value of the current sensor, else select the current value of the current sensor.
  - transient to transient: If the current sensor is in the transient state, then the value selected is the previous valid value of the current sensor, else select the current value of the current sensor.
  - transient to isolated: If the current sensor is isolated, select the current value of the next sensor in priority order and set it as the current sensor. Else select the current value of the current sensor.

- transient to normal: Select the current value of the current sensor.
- normal to normal (all other cases): Select the current value of the current sensor.

Note: In any of these cases the validity flag is set to **valid**.

- If $num\_iso > q$: As we have not made any claims, we can return any value. We shall select the value of the current sensor with validity flag as **invalid**.

Chapter 4

Specification and Verification of a Miscomparison Logic

This chapter describes our miscoparison logic and its proof of correctness done in *Coq*. A miscomparing sensor is one whose reading differs more than a threshold away from (the unknown) *ground_truth*. A basic assumption is that there is atmost one such sensor. Since the *ground_truth* is unknown, we must have at least three values to find the miscomparing one. So we assume that there are at least three values for applying the miscomparison logic. One of the key functionalities of a voter is to find the miscomparing sensor. This allows the voter to determine if the selected sensor is valid or if the current sensor should be switched.

Miscomparison logic tries to find a sensor that is deviating more than δ away from the *ground_truth* if any exist, where δ represents the maximum anticipated noise level. When a sensor is said to be miscomparing with another sensor, it means their respective reading differ by at least a predefined threshold (*miscomp_thresh*). In our algorithm, we choose *miscomp_thresh* to be 2δ. Our logic ensures that if our algorithm identifies a sensor as faulty its value is definitely δ away from the *ground_truth* and if there is any sensor whose value is 3δ away from the *ground_truth* then our algorithm will correctly identify it. Our

miscomparison logic uses only $O(n)$ comparisons where n is the number of sensors being compared.

4.1 Single Fault Assumption

The *Single Fault Assumption* is that in any given cycle, among the non-isolated and healthy sensors, there is atmost one sensor that is more than δ away from the $ground\_truth$.

4.2 Requirements

R1. If a miscomparing sensor is found, it deviates more than δ from the *ground truth*.

R2. If a miscomparing sensor is found, no other sensor deviates more than δ from the $ground\_truth$

R3. If a sensor deviates more than 3δ from the $ground\_truth$, then that sensor should be found as the miscomparing sensor.

Selection of Miscomparison Threshold All the sensor readings which fall in the interval $[ground\_truth - \delta, ground\_truth + \delta]$ are considered correct, because we know δ is the maximum anticipated error due to noise. So the difference between any two acceptable sensor readings can be atmost 2δ (when one is $ground\_truth - \delta$ and the other is $ground\_truth + \delta$). So if the difference between two sensor readings is more than 2δ then we can confirm at least one of the sensors is deviating more than δ from the $ground\_truth$. This was the reason for the selection of $miscomp\_thresh$ to be 2δ.

Justification for choosing 3δ in R3 If a sensor is reading $ground\_truth + 3\delta$ and all the other sensors readings are equal to $ground\_truth + \delta$ then none of the sensors miscompares with another. In such cases, we cannot detect the miscomparing sensor, because the ground

truth is unknown. This is the reason why we can only guarantee to detect sensors reading more than 3δ away from $ground\_truth$.

4.3 Miscomparison Logic

Under the assumption that at most one sensor exhibits either miscomparison or a 'Not OK' flag within a given cycle, the following logic is used to identify the faulty sensor.

A sensor is considered to be miscomparing with another sensor if the absolute difference between their measured values exceeds a predefined threshold of $2 * \delta$, where δ represents the maximum anticipated noise level.

The absolute difference between any two sensors is the maximum for the minimum value and maximum valued sensors. Since the minimum or the maximum valued sensor deviates the most from the ground truth, using $single\_fault$ assumption, we can say that if a miscomparing sensor exists then it is either the maximum valued or the minimum valued sensor. From $single\_fault$ assumption we can also see that if a sensor miscompares with more than two sensors then its reading is more than δ away from the $ground\_truth$.

Our claims are formally defined as properties given below.

P1. If a miscomparing sensor is found, it should be either the maximum or minimum valued sensor.

P2. If a miscomparing sensor is found, it miscompares with at least 2 sensors.

The pseudo-code for the logic is given in Listing 4.1.

Listing 4.1 Logic to Find miscomparing sensor

```
# We are only using non-isolated sensors for any
   computation in this algorithm


1. if the minimum value sensor is miscomparing with more
   than 2 sensors, then
```

```
        return index of the minimum valued sensor
2. else if the maximum value sensor is miscomparing with
   more than 2 sensors, then
        return index of the maximum valued sensor
3. else
        return None
```

4.4 Implementation

4.4.1 Initial Attempt - Using Decision Trees

To formally verify the correctness of the miscomparison logic using the Coq theorem prover, we adopted a decision tree-based strategy and proved its correctness. As a preliminary exercise, we decided to develop an algorithm to compute the decision tree for a well-known problem of finding the maximum element from a list. This serves as a practical example of establishing the correctness of both the decision and the algorithm that uses the decision tree.

Find a max element of a list

Tree: The leaves of the binary tree are an index of the list. The internal nodes contain two indices that denote the indices that are to be compared. The Left subtree contains the decision tree corresponding to the comparisons to be made if the element corresponding to the first index (of the two) is greater than the element corresponding to the second. The Right subtree contains the decision tree for the comparisons to be made otherwise.

Valid Tree: A valid tree is a tree that always finds the maximum element from the list when we execute the decision tree. In our algorithm, we used a decision tree that

implements the algorithm of finding the max element by finding the max of the first two elements and comparing it with the third and so on.

A valid decision tree for a 4-element list will look like as shown below 4.1.

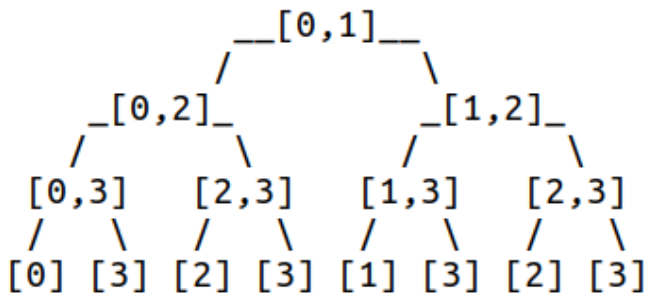

Fig. 4.1 The decision tree for 4 element list

Computing Max: A compute function will take any list of size n and a decision tree (need not be valid) with valid indices, and executes its decisions to find the index. If the passed decision tree is valid then the found index will be the max element of the list.

The code for the implementation can be found in `Phase II/tree_approach` folder in the repository [6].

Proving Correctness

While the generation of the decision tree for the miscomparison logic is complete, we weren't able to complete the formal proof of its correctness as we abandoned it for another approach as the writing of the proof became tedious.

4.4.2 Final Implementation

Our algorithm to find the miscomparing sensor given in Listing 4.1 requires computing maximum and minimum values from a list of values. In the actual implementation, our miscomparison logic requires only $O(n)$ comparisons, where n is the length of the list.

Finding minimum and maximum

We wrote functions to compute the maximum and minimum values from a list of natural numbers. The functions are written in such a way that the proof of correctness is embedded in the type definition of the function itself. For instance, the function that computes the maximum value from the list also gives proof that the computed value is indeed from the list and its value is greater than or equal to all the values in the list.

The type of the *find_min* and *find_max* are as follows,

Listing 4.2 Type of *find_max*

```
Fixpoint find_max (x: nat) (l : list nat):
{n:nat | In n (x::l) /\ Forall (fun y => y <= n) (x::l)}.
```

Listing 4.3 Type of *find_min*

```
Fixpoint find_min (x: nat) (l : list nat):
{n:nat | In n (x::l) /\ Forall (fun y => n <= y) (x::l)}.
```

The implementation code is found in `Phase II/final_implementation/min_max.v` file in the repository [6].

Finding miscomparing sensor

The *faultySensor* function computes the index of the miscomparing value if any exists. The list contains the readings from non-isolated sensors in a cycle. The pseudo-code of the function is defined in Listing 4.1. Along with the index of miscomparing value, it also returns the proof of satisfaction of properties P1 and P2 given in Section 4.3.

The type of the *faultySensor* function is

Listing 4.4 Type of faultySensor

```
Definition faultySensor (x : nat) (l : list nat) : option
   {n : nat | minindx_or_maxindx x l n} .
```

where *minindx_or_maxindx* is

Listing 4.5 Definition of minindx_or_maxindx

```
Definition minindx_or_maxindx (x : nat) (l : list nat) (
   index : nat) : Prop :=
((index = find_index_from_in (proj1 ( proj2_sig (find_min
   x l)))) /\
      (1 < length (filter (fun y => thresh <? y - proj1_sig
          (find_min x l)) (x::l)))) \/
((index = find_index_from_in (proj1 ( proj2_sig (find_max
   x l)))) /\
     (1 < length (filter (fun y => thresh <?  proj1_sig (
        find_max x l) - y ) (x::l)))).
```

Verification of Properties

We proved that our *faultySensor* function satisfies the requirements R1 and R2 in Section 4.2. The formal definition of the lemma in Coq is as shown below.

- R1.

```
Lemma faulty_sensor_deviates_gt_delta {mis} :
    faultySensor x l=Some(mis) ->
    adiff  ground_truth (nth (proj1_sig mis) (x::l
       ) 0) > delta.
```

- R2.

```
Lemma no_other_sensor_is_faulty {v}{mis}{in_pf :
   In v (x::l) } : (faultySensor x l = Some (mis))
    -> (nth (proj1_sig(mis)) (x::l) 0) <> v ->
   adiff ground_truth v  <= delta.
```

We are yet to prove 1 more important claim, that is,

- R3.

```
Lemma finds_value_deviate_3delta {v} {mis} (pf_in
   : (In v (x::l))) :
    adiff ground_truth v  > 3*delta ->
    (faultySensor x l = Some (mis)) /\
        (nth (proj1_sig(mis)) (x::l) 0) = v.
```

The implementation code is found in `Phase II/final_implementation/miscomp_logic.v` file in the repository [6].

Chapter 5

Conclusion and Future Work

In this project, our main goal was to define and verify the correctness of a redundancy management system in the Coq environment. This report has presented the ongoing efforts toward formally verifying a redundancy management logic for an avionics system with n redundant sensors. We have reviewed the prior work on a simpler version of the system with three sensors. Using the Coq theorem prover, we first specified a Triple Modular Redundant (TMR) system based on informal requirements obtained from ISRO. The TMR design modeled the sensors, acquisition electronics, communication links, voter logic, and states and transition function for proper management of the three redundant systems.

We then generalized this to formally design an n-redundant system capable of handling any redundancy level of $n \geq 3$ sensors. We modified the states and the transition function to accommodate for the n redundant sensors.

We then designed a miscomparison logic for n sensors. We initially took a tree-based approach to design and prove the correctness of the miscomparison logic, while the generation of the decision tree was complete, we couldn't prove its correctness as it became tedious. We then tried a different approach, using which we defined and verified the miscomparison logic which requires only $O(n)$ comparisons. We proved two properties (P1 and P2 from Section 4.3) and two other requirements (R1 and R2 from Section 4.2) of the

miscomparison logic.

Future Work:

The proof of the requirement R3 from Section 4.2, concerning the detection of sensors deviating more than 3δ from ground truth, remains pending. We were not able to complete it due to time constraints. The voting logic discussed in Chapter 3 is yet to be formally defined and verified in Coq. The overall system's correctness and invariance need to be proved as well. Translating the formally verified Coq specifications into a hardware description language for real-world implementation presents a valuable future step.

References

[1] Coq Development Team. (2024) Coq - formal proof management system. [Online]. Available: https://coq.inria.fr/

[2] S. Dajani-Brown, D. Cofer, G. Hartmann, and S. Pratt, "Formal modeling and analysis of an avionics triplex sensor voter," in *Model Checking Software*, T. Ball and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 34–48.

[3] NuSMV Development Team. (2024) Nusmv: a new symbolic model checker. [Online]. Available: https://nusmv.fbk.eu/overview.html

[4] S. Chakraborty, "Model checking of a triple redundancy management system," M-Tech Project Report of IIT Palakkad, 2023.

[5] TLA+ Development Team. (2024) Tla+ home page. [Online]. Available: https://lamport.azurewebsites.net/tla/tla.html

[6] Saarang, MIS AB. (2024) btp_final_code. [Online]. Available: https://github.com/saarangs2002/btp_report_code