# A Correct by Construction Fault Tolerant Voter for Input Selection of a Control System

Arif Ali AP ⊠ •

Department of Computer Science and Engineering, Indian Institute of Technology Palakkad, India

Jasine Babu 

□

Department of Computer Science and Engineering, Indian Institute of Technology Palakkad, India

Deepa Sara John 

□

□

ISRO Inertial Systems Unit, Indian Space Research Organization, Kerala, India

#### Abstract

Safety-critical systems use redundant input units to improve their reliability and fault tolerance. A voting logic is then used to select a reliable input from the redundant sources. A fault detection and isolation rules help in selecting input units that can participate in voting. This work deals with the formal requirement formulation, design, verification and synthesis of a generic voting unit for an N-modular redundant measurement system used for control applications in avionics systems. The work follows a correct-by-construction approach, using the Rocq theorem prover.

2012 ACM Subject Classification Theory of computation -> Logic -> Logic and verification, Theory of computation -> Logic -> Automated reasoning

**Keywords and phrases** Fault Tolerant System Design, Formal Verification, Correct by Construction, Input Selection, Interactive Theorem Proving

Funding A part of this work is funded by Indian Space Research Organisation (ISRO) RESPOND project RES-IISU-2022-013 titled 'Formal Analysis and Verification of Redundancy Management Logic for Navigation Processor used in Man-rated Launch Vehicle'.

Acknowledgements The authors acknowledge Saarang S and Mis Ab VP, former undergraduate students of IIT Palakkad, for their involvement in preliminary experiments with the Coq implementation of the voter, and Sandra S, visiting student at IIT Palakkad, for testing support. The authors also thank Murali Krishnan, Professor, National Institute of Technology Calicut, for his useful feedback on the initial draft.

# 1 Introduction

Redundancy management is used in safety-critical systems to improve their reliability and fault tolerance [9, 10, 6]. Examples of some well known domains of safety critical applications include avionics control, automated vehicles and nuclear reactor control systems [15, 14]. Since the reliability guarantee provided by testing is insufficient, such systems are suitable candidates for formal verification [4]. From the late 1980s, usage of formal verification for the design of fault-tolerant systems [1] was supported by agencies such as NASA, which work in the domain of safety-critical applications [31, 9, 28]. Owre et al. [23] mention the formal verification of fault-tolerant architectures as a major motivation for the development of the PVS theorem prover. Interactive consistency and convergence [19], clock synchronization [29, 21], and redundancy management [9] are some well-studied subproblems of fault-tolerant system design for which formal verification has been attempted. There are also works on the verification of timing specifications of fault-tolerant systems [27, 30, 13, 32]. Most of the works on formal verification of fault tolerant systems [19, 29, 5, 24, 16, 17] discuss only verification of the formal model and not the code synthesis. There is a recent work by Rahli et al. [26] that discusses Rocq based synthesis of a code implementing a Byzantine

Figure 1 In each cycle, the input selection unit has to select an appropriate measured value and feed it to the controller, along with some parameters indicating the reliability of the output. Between consecutive cycles, the change in the physical quantity being measured is assumed to be small. Preserved data stores some information from past cycles and it is used for fault handling and output generation. The output value could be retained from the previous cycle, when the current cycle measurement is not reliable.

fault tolerant state machine replication protocol based on message passing.

Our work deals with the formal modelling, verification and construction of an input selection unit for a synchronous N-modular redundant measurement system used for control applications in avionics systems. Though the redundant input units work synchronously and measure the same physical quantity, there can be differences in the measured value due to the noises in the measurement system (see Section 2.2). Moreover, some of these input units may have transient faults due to factors such as cosmic radiations [22] and vibrations [11], or permanent faults due to hardware failures [7]. Hence, it becomes necessary to apply an input selection logic to the values obtained from the redundant input units and produce a single output value corresponding to the physical quantity. In this work, we follow a correct-by-construction approach for synthesis of the input selection unit which uses a noise-resilient voting algorithm for the selection. Henceforth, we refer to the input selection unit as the *voter unit*.

The interesting application scenarios are when the system is running continuously, where in every cycle of operation, each input unit produces a measurement. In many real scenarios, the incremental change in the physical quantity being measured (such as distance to an object, angular velocity, atmospheric pressure, etc. measured in an avionics control unit) from one cycle to the next is small and smooth. In each cycle, the voter unit has to select one of the input units as a prime unit. The selection should ensure a reliable output value by discarding the measurements having transient errors. In addition, permanently faulty units should be identified and isolated, so that they do not corrupt the feed to the controller. To do this effectively, the input selection of each cycle has to also take into account some information from the past cycles. If the measurement domain is  $\mathcal{D}$ , the number of input units is N and the information based on the past t cycles is used for the calculations, then the voting unit essentially calculates a function  $f: \mathcal{D}^{tN} \mapsto \mathcal{D}$  in every cycle. In a more general situation, some self-reported parameters from the input units are also required for the decision making. It is also desired that, along with the selected measurement value, some reliability parameters of the output are also supplied to the controller. The reader may refer to Figure 1 which depicts an illustration of such a system. The formal verification of such systems becomes challenging due to the large size of the underlying state space [4, Section 1.3.1].

One aspect of transient resilience of the system is to ensure that measurements with transient errors are discarded, without getting fed to the controller. On the other hand, measurements are usually taken from redundant units that are physically separated and there could be minor deviations in the measurements made by them. Hence, it is desirable to avoid frequent switching of the prime unit due to transient failures, to maintain better stability. Although these two aspects of transient resilience may seem conflicting, the assumption that the incremental variations in the measured quantity between consecutive cycles is small, makes it possible to strike a balance. If the prime unit selected for the previous cycle has a transient error in the current cycle, the algorithm can retain the output value of the previous cycle without switching the prime unit, as long as the age of the measurement is within the allowed limits. If an input unit is showing failure for many consecutive cycles beyond a persistence limit, then it has to be permanently isolated. These requirements make the design of an input selection unit for a controller different from a typical voting algorithm.

As explained by Butler [2] and Miller et al. [20], a major effort in the formal verification of a safety critical system goes into unambiguously capturing the requirements of the system using formal logic. A major contribution of this work is the formulation of a consistent set of requirements relevant to the application context presented above. We demonstrate the theoretical limits of achievable soundness and completeness. Our voter unit algorithm is proven to match the theoretical limits.

If the number of input units without a permanent fault falls below a certain threshold, then it is well known that fault identification cannot achieve completeness [6]. However, by fine-tuning the fault identification algorithm, in certain situations it is possible to identify with certainty, a subset of units as non-faulty and another subset of units as faulty. In such cases as well, we derive the theoretical limits of fault identification and our algorithm matches these limits. Such a finer fault identification will help in prolonging the duration of reliable operation of the voter unit, by avoiding mis-classification of a healthy input unit as permanently faulty.

When the algorithm is designed to prolong the duration of operation of the voter unit this way, there could be certain boundary cases wherein the prime unit of the previous cycle got isolated due to a permanent fault and it is not theoretically possible to identify with certainty even one unit as non-faulty to assign as the new prime unit. In such situations, if the number of units without a permanent fault is above the minimum threshold for continuing the operation of the voter unit, then for a short while our algorithm will change the state of the voter unit as un\_id (unidentifiable), in anticipation of a recovery from transient errors, while continuing to provide the output from the previous cycle. The algorithm will take care of bounding the age of the output in such scenarios as well.

To make the system description and its requirements more precise, it is necessary to formally specify the assumed fault model and fault hypotheses [30] indicating the type, number and rate of faults that the system can tolerate while remaining operational [18, 25]. This is done in Section 2. In Section 3, we formulate the functional requirements of the system. In Section 4, we give an overall design of the voter unit. The voter unit is designed as a state transition system with three major functions: fault identification, handling of transient and permanent failures using a multi cycle history of faults, and generation of a reliable output, which will be fed to the controller.

Since theorem proving techniques are known to be more scalable compared to other approaches to formal verification [4] for systems with a large state space, this work is done using the Rocq interactive theorem prover [34]. Rocq theorem prover is based on the calculus of inductive construction and satisfies the de Bruijn criterion [3]. In addition, Rocq supports higher-order logics and dependent types, which makes it easy to express complex specifications succinctly. We have attempted to follow Rocq's design principle of defining functions carrying proof of its properties embedded with it, rather than completing a code

#### 4 A Correct by Construction Fault Tolerant Voter for Input Selection of a Control System

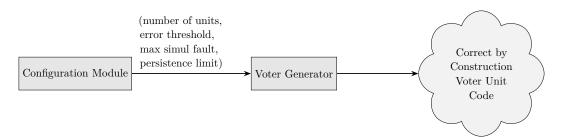


Figure 2 To adjust the reliability levels, some parameters are kept as designer configurable, with a permissible range of values for each. A voter unit code with the specified set of parameter values can be generated.

first and then proving the properties. Some of the requirements are shown to be satisfied as invariants of the states, while the remaining requirements, which depend on the multi cycle behaviour of the system, are captured as invariants maintained during every state transition. To allow the designer to fine-tune the reliability parameters to match the application specific requirements, we maintain the definition of some basic reliability parameters in a separate configuration module, and the system implementation is parameterized by them. The code of the voter unit is synthesized with the selected values of these parameters (see Figure 2).

# 2 System Description and Fault Model

In this section, we formally define the input and output formats of the voter unit and the fault model and fault hypothesis.

# 2.1 Input Format of the Voter Unit

In each cycle of operation, the voter receives input from N redundant input units that work synchronously and measure the same physical quantity that continuously varies over time. The value of N can be decided by the user. The voter unit maintains a unique identification number uid for each input unit based on the port of the voter unit interface to which the input unit is connected. From each input unit, the voter receives a reading, which is a 2-tuple consisting of a) the measured value val, and b) the self-identified health status of the unit hw\_hlth. This kind of data format where a measurement is available with a health status is commonly used by sensors and measurement units used for critical applications [8, 12].

The measurement from a properly functioning input unit may be noisy, but within an acceptable threshold  $\delta$  from the ground truth. An input unit can malfunction due to a transient or permanent fault.

# 2.2 Fault Model

As described in the input format, in each cycle, each input unit has a self-identified health status: good or bad. Another type of fault that is not self-identifying is a deviation greater than a fixed noise threshold  $\delta$  from the ground truth of the physical quantity being measured. We call it a *deviation fault*. A faulty behaviour of an input unit is defined as an occurrence of any one of these two faults. A faulty behaviour is considered *transient* if it persists less than a consecutive number of cycles of operation, denoted by persistence\_lmt. Otherwise, it is considered a *permanent fault*. For transient fault resilience, persistence\_lmt should be greater than 1. It is assumed that the faulty behaviour is limited to the input units

alone, and the hardware interface of the voter unit receiving readings from the input units is assumed to be non-faulty. Byzantine faults [18] are not considered in our formulation. We need to make a *simultaneous fault hypothesis* about the maximum number of simultaneous new faults that are anticipated in a cycle of operation. This assumption is stated below.

▶ Hypothesis 1 (Simultaneous Fault Hypothesis). In any cycle, among the input units that do not have a permanent fault before the cycle begins, at most max\_simul\_fault units can show faulty behaviour.

The correctness of the voter unit's behaviour can be guaranteed only if a certain minimum number of input units without a permanent fault are present. Formally, we define a parameter min\_required and say that the system satisfies maximum permanent fault assumption when there are at least min\_required number of input units without a permanent fault. By simultaneous fault hypothesis, it is evident that min\_required should be at least max\_simul\_fault + 1. The values of parameters  $\delta$ , persistence\_lmt and max\_simul\_fault can be decided by the user.

# 2.3 Output Format of the Voter Unit

The format of the output produced by the voter unit is a 4-tuple consisting of voter\_output, output\_age, validity\_status and presrvd\_data. The first three are for feeding to the controller, while presrvd\_data is a feedback to the voter unit itself for decision making during state transitions.

In each cycle, the voter should select one of the input units as the *prime unit*. The voter\_output consists of the uid of the prime unit and a reading from the prime unit. The voter\_output is a current cycle measurement of the prime unit if the prime unit has no identified fault in the current cycle. Otherwise, the previous cycle voter\_output is retained. The output\_age is used to indicate the number of cycles since when the voter output measurement has not been updated. The validity\_status is an indication of the reliability of the voter\_output and can be valid, un\_id (unidentifiable) or not\_valid. The presrvd\_data consists of the uid, reading and accumulated fault history of the prime input corresponding to the voter\_output.

### 3 Formulation of Functional Requirements

The functional requirements of the generic data selection and voting unit is described in this section. The requirements are broadly classified into three categories: (i) related to fault identification, (ii) related to isolation of permanently faulty units and (iii) related to output generation. We provide an analysis of the theoretical limits achievable for fault identification and show that the requirements formulated are tight with respect to these limits. The proof of correctness of fault identification rules presented in a conventional way in this section are machine verified in our Rocq implementation. The consistency of the set of requirements is verified in Rocq, and the code of the voter unit produced is proven to be correct by construction satisfying all the requirements.

#### 3.1 Fault Identification

In each cycle, the voter unit should analyze the input received from the input units and record the information of faults identified by the fault identification algorithm. As a method of recording the faults, the voter unit will maintain a risky\_count corresponding to each

input unit, which indicates the number of consecutive cycles of operation (including the current cycle) in which the measurement from the unit is not identified as non-faulty. Before explaining the details of requirements related to fault identification, we need to first give the definitions of soundness and completeness of fault identification.

#### 3.1.1 Soundness and completeness

As the actual value (ground truth) of the physical quantity being measured is unknown, units with deviation faults have to be identified using mutual deviation checks. As per the fault model, a non-faulty measurement value can have a deviation of  $\delta$  from the ground truth, in a positive or negative direction. Hence, a deviation of  $2\delta$  between two inputs is allowed and cannot be identified as a fault.

▶ **Definition 1** (miscomparison). A deviation of more than  $2\delta$  between measurements in a cycle from a pair of input units is a miscomparison.

A miscomparison indicates that at least one of the two measurements involved in the comparison is faulty. Since it is theoretically impossible to identify all deviation faults in certain scenarios [6], we want to classify the units to have their miscomparison\_status to be one among miscomparing, not\_miscomparing and maybe\_miscomparing.

We start by analysing a scenario of fault identification that will justify the definitions of soundness and completeness of fault identification that follow.

 $\triangleright$  Claim 2. There are situations where the measurement of an input unit u deviates  $3\delta$  from the ground truth and it is still impossible to detect the unit as faulty.

**Proof.** Suppose that unit u gives a measurement  $x+3\delta$ , while other units give a measurement value  $x+\delta$ . Now, consider two scenarios (i) the (unknown) ground truth is x and (ii) the ground truth is  $x+2\delta$ .

In both scenarios, the mutual deviation check will not show any miscomparison. In the first case, the measurement of u deviates  $3\delta$  from the ground truth and in the second case, its deviation is within the noise threshold  $\delta$  from the ground truth. Since the ground truth is unknown, it is impossible to distinguish between these two scenarios.

The above claim justifies the definitions of soundness and completeness of the fault identification stated below. We will discuss the acheivability of these in Section 3.1.2.

- ▶ **Definition 3** (Soundness and Completeness of Deviation Fault Identification).
- $\blacksquare$  (Soundness a) If the miscomparison\_status of a unit is miscomparing, its measurement deviates more than  $\delta$  from the (unknown) ground truth.
- $\blacksquare$  (Soundness b) If the miscomparison\_status of a unit is not\_miscomparing, then the deviation of the measurement of that unit from the ground truth is at most  $3\delta$ .
- $\blacksquare$  (Completeness a) If the measurement of a unit deviates more than  $3\delta$  from the ground truth, then its miscomparison\_status should be miscomparing.
- $\blacksquare$  (Completeness b) If the deviation of the measurement of a unit from the ground truth is at most  $\delta$ , its miscomparison\_status should be not\_miscomparing.

# 3.1.2 Requirements of soundness and completeness of fault identification

The requirement regarding the handling of self-identifying faults is straightforward. Input units with self-identifying bad health must be identified as faulty. The following basic correctness requirement for fault identification algorithm is the following.

- **R1.** For an input unit which is not yet marked as permanently faulty,
- **risky\_count** is either zero or one more than its risky\_count in the previous cycle and
- risky\_count is incremented in a cycle if and only if its self identifying health status is bad or its miscomparison\_status is miscomparing or maybe\_miscomparing.

Now, we will identify some conditions under which the completeness property of fault identification cannot be achieved, so that the requirements on the completeness can be fine tuned. One such scenario is when among the input units that do not have a permanent fault, there are not enough units without self-identifying faults. We will make this statement more precise shortly. Among input units without a permanent fault, if there are k units with self-identifying bad health in a cycle, then by  $simultaneous\ fault\ hypothesis$ , there can be at most max\_simul\_fault -k units with non-self-identifying deviation faults. Let mis flt lmt = max simul fault -k.

Description Claim 4. There are situations in which the number of input units that do not have a permanent fault and having good health is 2 \* mis\_flt\_lmt and yet it is not possible to achieve completeness of deviation fault identification.

**Proof.** Consider a situation where mis\_flt\_lmt is 2 and there are 4 units  $u_1, u_2, u_3, u_4$  without a permanent fault yet and with good health. Suppose  $u_1$  and  $u_2$  have the same measurement x and  $u_3$  and  $u_4$  have the same measurement y, which is more than  $3\delta$  away from x. There are two possible scenarios: (i) both  $u_1$  and  $u_2$  are faulty, while y is within  $\delta$  of the ground truth.

Here, since  $|x-y| > 3\delta$ , it is clear that either x or y deviates more than  $\delta$  from the ground truth. However, since the ground truth is unknown, it is impossible to distinguish which among these actually occurred. Hence, even when the number of input units that do not have a permanent fault and having good health is  $2*mis_flt_lmt$ , completeness cannot be achieved.

▶ Definition 5 (minimum surviving units assumption). The system is said to satisfy the minimum surviving units assumption when among input units that do not have a permanent fault in a cycle, at least 2 \* mis\_flt\_lmt + 1 units have good health status.

By Claim 4, the following requirement regarding completeness of fault identification is tight with respect to the theoretical limit.

**R2.** When the system satisfies the minimum surviving units assumption, the miscomparison\_status of each unit should be either miscomparing or not\_miscomparing and this classification should satisfy the completeness requirements given in Definition 3.

It can be shown that this requirement is always satisfiable. We will address this point later. The most important requirement of our deviation fault identification algorithm is the following.

**R3.** The soundness conditions specified in Definition 3 must always be satisfied by the fault identification algorithm.

### 3.1.3 Avoiding mis-classifications to extend operational life

If the minimum surviving units assumption is not satisfied, our main objective is to satisfy the soundness requirements given in Definition 3. For this, it suffices to trivially declare all units as maybe\_miscomparing when the *minimum surviving units assumption* is not satisfied. However, this may lead to an early declaration of the input units as permanently faulty and

may cause the system to fail to satisfy the maximum permanent fault assumption. Hence, let us explore some more scenarios in which partial fault identification can be achieved.

If the measurement value of an input unit u miscompares with at least mis flt lmt + 1 other input units without a permanent fault yet and with good health, then the measurement of u deviates more than  $\delta$  from the ground truth. Further, there are situations, where a unit is miscomparing with mis\_flt\_lmt other units and still the unit cannot be identified as faulty.

**Proof.** Among the mis flt lmt + 1 other units with which unit u miscompares, by the simultaneous fault hypothesis, there must be at least one non-faulty unit u'. Since measurement of u' is within  $\delta$  from ground truth and the measurements of u and u' deviate more than  $2\delta$ , it follows that the measurement of u definitely deviates more than  $\delta$  from the ground truth. Hence, u is faulty. To see the second part of the claim, it suffices to consider the scenario described in the proof of Claim 4.

Now, let us consider another interesting scenario where a partial fault identification is possible. To simplify our presentation, we make the following definitions.

- ▶ Definition 7 (miscomparing\_list, rem\_mis\_flt\_lmt). miscomparing\_list in a cycle is the list of input units without a permanent fault yet and with good health and whose measurement miscompares with at least mis\_flt\_lmt + 1 other such units. We define rem mis flt lmt = mis flt lmt - |miscomparing list|.
- ▶ **Definition 8** (identified fault). A unit is said to have an identified fault in a cycle if its health status is bad or it is in the miscomparing list.
- $\triangleright$  Claim 9. Let u be a not yet permanently faulty input unit without an identified fault. If the deviation of the measurement of u is within  $2\delta$  from rem\_mis\_flt\_lmt other such units, then the measurement of u is within  $3\delta$  of the ground truth. Further, there are situations where the measurement of such a unit u is within  $2\delta$  from exactly rem\_mis\_flt\_lmt -1other units and still it is impossible to assign a miscomparison status miscomparing or  $not_miscomparing$  to u satisfying the soundness requirements.

**Proof.** By simultaneous fault hypothesis, rem\_mis\_flt\_lmt is an upper bound on the remaining number of units with faults among units with health status good, after applying the rule in Claim 6. Hence, if the deviation of the measurement of a unit u which does not have an *identified* fault is within  $2\delta$  from rem\_mis\_flt\_lmt other such units, then we can conclude that either u or at least one among them is non-faulty. In both these cases, we can conclude that the unit u has a measurement within  $3\delta$  from the ground truth.

To see the second part of this claim, it is sufficient to analyze a similar situation as given in the proof of Claim 4. Suppose mis\_flt\_lmt is 2 and there are 4 units  $u_1, u_2, u_3, u_4$ without a permanent fault yet and with good health. Suppose  $u_1$  and  $u_2$  have the same measurement x and  $u_3$  and  $u_4$  have the same measurement y, which is more than  $3\delta$  away from x. Consider two possible scenarios: (i) the ground truth is x (ii) the ground truth is y. Here, observe that rem\_mis\_flt\_lmt is the same as mis\_flt\_lmt, which is 2. To satisfy the soundness condition, in the first scenario, miscomparison status of neither  $u_1$  nor  $u_2$  can be assigned as miscomparing. In the second scenario, their miscomparison status cannot be assigned as not\_miscomparing. The situation of  $u_3$  and  $u_4$  is symmetric.

Since it is impossible to distinguish between the two scenarios without knowing the ground truth, the miscomparison status of none of the units can be set to miscomparing or not\_miscomparing ensuring the soundness requirements.

Claim 6 and Claim 9 show that the two requirements stated below are tight with respect to the theoretical limits.

- **R4.** If the measurement value of an input unit miscompares with at least mis\_flt\_lmt+1 other input units without a permanent fault yet and with good health, then it should have miscomparison\_status as miscomparing.
- **R5.** Among the not permanently faulty input units which do not have an identified fault, if the measurement of a unit is within  $2\delta$  from at least rem\_mis\_flt\_lmt others, then its miscomparison\_status should be not\_miscomparing.

Now, the following is an interesting observation.

▶ Proposition 10. When the system satisfies the minimum surviving units assumption, requirements R4 and R5 are sufficient to guarantee requirements R2 and R3.

**Proof.** Suppose that the system satisfies the minimum surviving units assumption. Then  $2*mis\_flt_lmt + 1$  input units are available among units without a permanent fault yet and with good health. Consider such an arbitrary unit u. Among the other  $2*mis_flt_lmt$  units other than u, if there are at least  $mis_flt_lmt + 1$  units with which u miscompares, then by Claim 6, measurement of u deviates more than u from the ground truth. Otherwise, there are at most  $mis_flt_lmt$  other units with which u miscompares and hence there are at least  $mis_flt_lmt \ge rem_mis_flt$  units whose measurement is within u from that of u. In this case, by Claim 9, the measurement of u is within u deviation from the ground truth. In the first case, by R4 the miscomparison status of u will be u miscomparing and the second case, by R5 u will have miscomparison status u not\_miscomparing. In both cases, the soundness and completeness properties are satisfied. Since this can be done for an arbitrary u, R2 will be satisfied. Since soundness is ensured, R3 also holds.

▶ Note 11. Requirements R2 and R3 together means that under minimum surviving units assumption, all units are assigned miscomparison status miscomparing or not\_miscomparing and the classification is sound and complete. By Proposition 10, this can be ensured by satisfying requirements R4 and R5. Moreover, we can algorithmically ensure R4 and R5 by applying the straightforward conditions in Claim 6 and Claim 9.

#### 3.2 Isolation of Permanently Faulty Units

As discussed in the previous section, in each cycle the voter unit uses its fault identification algorithm to identify the fault status of each input unit. The voter unit maintains some information to keep track of the accumulated fault status of each unit. This information will be used to distinguish between transient faults and permanent faults in order to avoid corruption of the feed to the controller in future from a permanently faulty unit.

One such information maintained is the risky\_count of an input unit, which needs to be updated as mentioned in requirement R1. An input unit with a non-zero risky\_count which is less than persistence\_lmt is regarded to have a transient fault. If the transient fault clears before it reaches persistence\_lmt, then the risky\_count is reset to zero. A sequence of identified faulty behaviour in consecutive cycles, leading to the risky\_count becoming equal to persistence\_lmt is regarded as a permanent fault.

For each input unit, the voter unit also maintains an isolation status. The following are the requirements related to the isolation of a unit.

**R6.** A unit is isolated if and only if the risky\_count of the unit reaches the predefined threshold indicated by persistence\_lmt.

**R7.** If an input unit is isolated in a cycle of operation, it continues to remain isolated for all future cycles.

# 3.3 Output Generation Related Requirements

Using the current cycle input data from each input unit and based on the updated status information of each input unit, the voter generates an output, which is used as feed to the controller. As mentioned in the output format, the feed to the controller consists of voter\_output, output\_age and validity\_status.

# 3.3.1 Voter output and prime unit switching

A unit\_output consists of the uid of a unit and a reading received from it. To explain the requirements regarding the output, the following definition will be useful.

▶ Definition 12 (healthy data). A healthy\_data is a unit\_output with good health such that, in the cycle of its measurement the input unit with the corresponding uid is not isolated and its miscomparison\_status is not\_miscomparing.

The voter\_output in a cycle is the selected unit\_output and the uid present in it is the prime unit of the cycle. In case of a transient fault in the prime unit, the voter\_output in the previous cycle will remain the voter\_output for current cycle, until the fault disappears or becomes permanent. One basic requirement is that if a fault is identified, it should be contained within the faulty unit and should not be allowed to corrupt the input to the controller. This requirement is stated below.

**R8.** The voter always outputs a healthy\_data.

If the prime unit of the previous cycle gets isolated in the current cycle and another healthy unit is available, then the voter should select another input unit to provide the feed for the controller. However, as mentioned in the introduction, a requirement in this context is that frequent change of the prime unit used for output generation must be avoided. This requirement is formulated below.

**R9.** Switching of the prime unit occurs in a cycle only if the prime unit of the previous cycle got isolated.

The following proposition shows that if the requirements defined so far are satisfied, switching of prime unit will not happen frequently.

▶ Proposition 13. By satisfying the requirements R1, R6, R8 and R9, it is ensured that the input unit selected as the prime unit shall not change for at least persistence\_lmt number of cycles.

**Proof.** Consider the time when an input unit u is just selected as the prime unit. By R8, u has health status good and miscomparison status not\_miscomparing. From R9, a new prime unit is selected only when u gets isolated. However, by requirements R1 and R6, it must take at least persistence\_lmt cycles for this to happen.

#### 3.3.2 Validity status related requirements

The correctness of the voter behaviour (fault identification, isolation and output generation) is guaranteed only if the maximum permanent fault assumption is satisfied. The voter maintains

a validity\_status to indicate the reliability of the output generated. The validity status can be valid, un\_id(unidentifiable) or not\_valid.

The validity status not\_valid indicates that the voter output is not reliable because the maximum permanent fault assumption does not hold and hence, no meaningful fault identification is possible. As mentioned in the introduction and in section 3.1.3, our voter unit design does a finer fault identification, even if it is incomplete, to delay the situation of the system failing to satisfy the maximum permanent fault assumption. As a consequence, there could be situations wherein the prime unit of the previous cycle is isolated in the current cycle and there is no input unit with healthy\_data to be used as the new prime unit, satisfying R8. This condition occurs when the fault identification logic is failing to identify even one input unit with healthy\_data because the condition mentioned in R5 is not satisfied. In such scenarios, if maximum permanent fault assumption still holds, there is a possibility that the faulty condition is transient and the voter may be able to recover from this condition in a few cycles. Hence, the voter unit can choose to retain the voter\_output of the previous cycle. The un\_id validity status is used to capture this situation. The validity status valid represents a higher level of reliability than un\_id, as we will explain shortly. The requirements regarding validity status are captured below.

- **R10.** The validity status of the voter is not\_valid if and only if the system does not satisfy the maximum permanent fault assumption.
- **R11.** The validity status of the voter is un\_id if and only if the maximum permanent fault assumption is satisfied and the prime unit of the previous cycle is isolated and no input unit is identified to be providing a healthy\_data.
- **R12.** If the system satisfies the maximum permanent fault assumption and the minimum surviving units assumption (see Definition 5), then the validity status of the voter must be valid.
- **R13.** If the *maximum permanent fault assumption* is satisfied, and if at least one unit is identified to be providing a healthy\_data, then the validity status of the voter must be valid.

#### 3.3.3 Output age related requirements

The output\_age is used by the voter to indicate the number of cycles since when the voter output measurement has not been updated. A basic correctness requirement related to output\_age updation is the following:

R14. Unless the validity\_status is not\_valid, output\_age is either zero or one more than the previous cycle output\_age. Further, if validity status is valid, then the output\_age is equal to the risky\_count of the prime unit.

An upper bound on output\_age is required to ensure that the feed to the controller is a measurement with a bounded lag. When the validity status is valid, our requirements for the output\_age are as follows.

- R15. The output\_age is zero if and only if the validity status is valid and voter output is a current cycle measurement.
- **R16.** If the validity status is valid, then output\_age  $\leq$  persistence\_lmt 1.

It is interesting to note that from the requirements stated so far, an upper bound of  $2*(persistence\_lmt-1)$  is guaranteed for output\_age when the validity status is un\_id. To prove this, we will use a relation that exists between the risky\_count of non isolated

input units and the output\_age when the voter validity is other than not\_valid. The relation is stated below.

 $\triangleright$  Claim 14. Suppose all the requirements are satisfied. Then, unless the validity status is not\_valid, for each input unit u which is not isolated, output\_age - risky\_count of u < persistence lmt.

**Proof.** We will show that this invariant is satisfied in each cycle if all the requirements (specifically, R1, R6, R10, R11, R14, and R16) are satisfied. By requirements R6 and R10, if the validity status becomes  $not_valid$ , it must remain so in the future. Initially, all units have risky\_count zero and the output\_age is also zero. Since persistence\_lmt > 0, the relation holds initially for all input units. Now, consider a cycle in which the validity status is different from  $not_valid$  after the cycle's fault identification. If the validity status is valid, then by requirement R16, output\_age < persistence\_lmt and hence the claim holds. Otherwise, the validity status must be un\_id. In this case, consider an arbitrary input unit u which remains non-isolated after the cycle. It suffices to show that if the relation holds for u before the cycle, it holds after the cycle as well.

By R11, u cannot be providing a healthy\_data in the current cycle. Therefore, by requirement R1, the risky\_count of u must be exactly one more than its value in the previous cycle. Moreover, by R14, output\_age can increase by at most one. Hence, as required, if the relation holds for u before the cycle, it holds after the cycle as well.

The following proposition is now easy to prove.

▶ Proposition 15. If the system satisfies requirements R1 to R16, then the output\_age is at most  $2 * (persistence_lmt - 1)$  as long as the maximum permanent fault assumption holds.

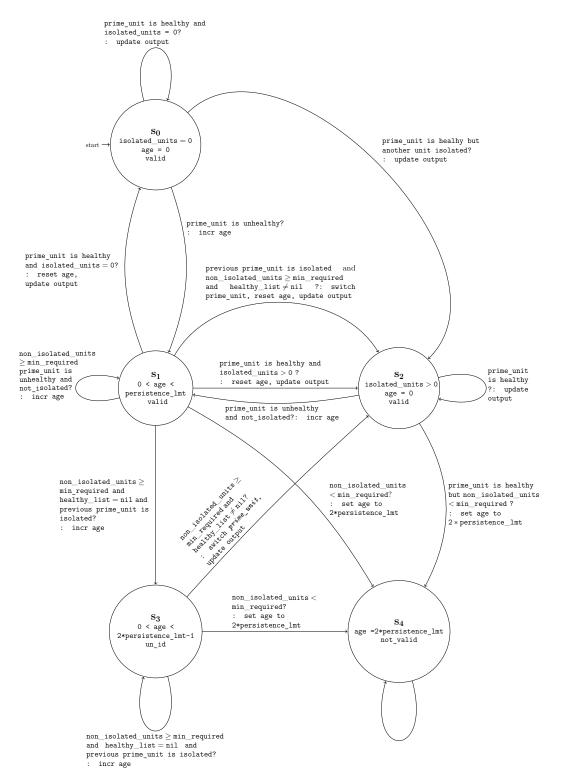
**Proof.** By requirement R10, unless the validity status is not\_valid, the maximum permanent fault hypothesis holds and so, there are at least min\_required > 0 non-isolated units. Consider one such non-isolated unit u. By Claim 14, output\_age - risky\_count of  $u \le persistence_lmt-1$ . Further, by R6, the risky\_count of u is at most persistence\_lmt-1. Hence, output\_age  $< 2 * (persistence_lmt-1)$ .

▶ Note 16. It is not obvious that the requirements R1 to R16 are consistent. The consistency is established by machine assisted proof created using Rocq. En route, we also obtained machine generated proofs of the first part of Claim 6, the first part of Claim 9, Proposition 10, Proposition 13 and Claim 14.

# 4 Design of the Voter Unit

As described in the voter output format, the voter unit provides a voter\_output for the controller along with an indication of the validity\_status and output\_age. The design of the voter unit has three key aspects: fault identification, handling of transient and permanent faults and computation of the voter output. The voter unit is designed as a state transition system. The configurable parameters of the voter unit are: num\_units to indicate the count of input units,  $\delta$  to indicate the noise threshold, persistence\_lmt used for isolation of permanent faulty units and max\_simul\_fault to indicate the maximum number of simultaneous faults that are anticipated in a cycle of operation.

In every transition, the voter unit will update the fault status (unit\_status) of each input unit by performing fault identification on the current cycle reading. The unit\_status includes



**Figure 3** State transition diagram of the voter unit. To avoid cluttering, the update in the isolation status of units and the number of isolated units during the transitions are not shown. The update in validity status is not explicitly shown during the transitions, but can be understood from the states.

the isolation\_status, miscomparison\_status and the risky\_count of the corresponding input unit. The logic used in the fault identification algorithm is an exact translation of the conditions specified in requirements R4 and R5. If both conditions do not apply, then the unit will be classified as maybe\_miscomparing. This method will satisfy requirements R2 - R5, as explained in Section 3.1.2 and 3.1.3. After the fault identification, the update in risky\_count is done as described in R1. Using this, the isolation of permanently faulty units is done as mentioned in requirements R6 and R7.

We explain the voter unit behaviour using five abstract states, as shown in the state transition diagram in Figure 3. The description of the states are as follows.

- S<sub>0</sub>: This is the initial state where there are no isolated inputs, the output\_age is zero and the validity\_status is valid. The output value will be the current cycle measurement of the prime unit.
- $\blacksquare$   $S_1$ : This state occurs when the prime unit used for voter\_output generation is having a transient fault. The previous cycle voter\_output is retained to satisfy R8, and output\_age is incremented on transition to this state.
- S<sub>2</sub>: This state occurs when the number of isolated units is non-zero, but non-isolated units are at least min\_required. There is at least one unit providing healthy\_data and one such unit is selected as the prime unit. Further, the output\_age is zero, indicating that the output value is a current cycle measurement from prime unit, and validity\_status is valid.
- S<sub>3</sub>: This state occurs when the input unit used for voter\_output generation in the previous cycle got isolated and there are no input units providing healthy\_data, even though the number of non-isolated units is at least min\_required. This state satisfies the premise of the requirement R11 and hence validity\_status is un\_id. The previous cycle voter\_output is retained to satisfy R8, and output\_age is incremented on transition to this state.
- S<sub>4</sub>: This state occurs only when the number of non-isolated units is below min\_required. In this state, the validity status is not\_valid as per requirement R10.
- ▶ Note 17. It can be verified from the description of states that the requirements R6, R8, R10, R11, R13, R15, R16 and the second part of requirement R14 are invariants for all states.

The state transitions are summarized below.

- From the initial state  $S_0$ , there are three transitions. These are a) to state  $S_1$ , when the prime unit used for voter\_output generation is having a transient fault and b) to state  $S_2$ , when the prime unit is healthy and one of the other units is isolated. The system remains in state  $S_0$  as long as the initially selected prime unit is healthy and none of the other units are isolated.
- From states  $S_1, S_2$  and  $S_3$ , a transition to  $S_4$  happens if and only if the maximum permanent fault assumption is not satisfied due to isolation of input units.
- From state  $S_1$ , there are five more transitions. These transitions are a) to state  $S_1$  when the prime unit has a transient fault b) to state  $S_2$  when the prime unit of the previous cycle is isolated and there is at least one healthy unit to be identified as the prime unit c) to state  $S_2$  when the prime unit of the previous cycle recovers from a transient fault and at least one input unit is isolated d) to state  $S_0$  when prime unit of previous cycle recovers from a transient fault and none of the input units are isolated e) to state  $S_3$  when prime unit of previous cycle is isolated and there are no healthy units to be identified as a prime unit.

- From state  $S_2$ , there is a transition to the same state when the prime unit is healthy. There is also a transition to state  $S_1$  when the selected prime unit has a transient fault.
- From state  $S_3$ , there are two more transitions. These are a) to state  $S_3$  when there are no healthy units and b) to state  $S_2$  when one of the input units recovers from a transient fault and becomes the prime unit.
- Once the system reaches the state  $S_4$ , it should remain there, in order to satisfy the requirements R7 and R10.
- ▶ Note 18. It can be seen that the requirements R1 R5, R7, R9, R12, and the first part of requirement R14 are invariants of the state transition rules.

# 5 Implementation in Rocq

We have used the design given in Section 4 for our Rocq implementation. We have used Rocq's support for higher-order logics and dependent types extensively (see Appendix A) while formally expressing the requirements mentioned in Section 3. Our implementation follows a correct-by-construction approach by defining the voter state and the state transition rules in such a way that they carry with them the proof terms of the invariants they satisfy. This makes sure that when these definitions are completed and are verified by Rocq, the invariants will certainly hold by the implementation. Since Rocq is based on the Calculus of Construction, all proofs are fully constructable, rather than being existential. The total time taken for verification in Rocq for properties R1 to R16 is approximately 23 seconds on a laptop with Debian 12 OS, 32GB RAM, 64-Bit Processor (Intel® Core<sup>TM</sup> Ultra 5 125U  $\times$  14).

The verified code obtained in Rocq is extractable to OCaml/Haskell/Scheme using the provisions in Rocq [33]. The number of lines of the extracted OCaml code is 362. The execution time of the extracted OCaml code for a single cycle state transition is around 3 milliseconds.

To get an overview of our implementation using Rocq, the reader may refer to Appendix A. The definition of the voter unit is given as the record type voter\_state. It may be noticed that the definition of voter\_state carries embedded proof terms of properties corresponding to requirements R8, R10, R11, R13, R15, R16 and the second part of requirement R14. The requirement R6 is an invariant of unit\_status, which is a field inside the voter\_state. The the state transition function is named as voter\_state\_transition. It carries proof terms of properties corresponding to requirements R1 - R5, R7, R9, R12, and the first part of requirement R14. The entire source code of our implementation is available in GitHub<sup>1</sup>.

#### 6 Discussion and Future Directions

Our work gives the design and synthesized code of a voter unit that feeds a control system. The synthesized code is correct by construction. The design is parameterized by the number of units and some reliability parameters. The verified code for the parameterized system is produced in Rocq. The values of the parameters can be appropriately fixed by the user during extraction of the code from Rocq to OCaml/Haskell/Scheme. The support available in Rocq for higher order logics and dependent types helped in presenting the formal specification of our requirements in a clear and succinct way.

In fault tolerant systems with a multi-level hierarchical design, redundancy management is required in each level of hierarchy. An extension in this direction is interesting. Further,

https://github.com/arifali-ap/Fault\_tolerant\_voter.git

one may have to deal with fault handling of asynchronous redundant units in such systems. For example, in a distributed fault-tolerant system, the control unit may receive feed from multiple asynchronous voting units, each working with independent logical clocks with small deviations. Each of these voting units may be receiving data from redundant input units. Synthesis of verified code for identifying and handling faults in such asynchronous systems is another direction for future work.

#### References -

- A. Aviziens. Fault-tolerant systems. IEEE Transactions on Computers, C-25(12):1304-1312, 1976. doi:10.1109/TC.1976.1674598.
- Ricky Butler and Sally Johnson. Formal Methods For Life-Critical Software, pages 319–329. Aerospace Research Central, October 1993. doi:10.2514/6.1993-4516.
- Adam Chlipala. Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. The MIT Press, 2013.
- Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. Handbook of Model Checking. Springer, 2018. doi:10.1007/978-3-319-10575-8.
- Samar Dajani-Brown, Darren Cofer, Gary Hartmann, and Steve Pratt. Formal modeling and analysis of an avionics triplex sensor voter. In Model Checking Software, pages 34-48, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- Daniel Davies and John F. Wakerly. Synchronization and matching in redundant systems. IEEE Transactions on Computers, C-27:531-539, 1978. URL: https://api.semanticscholar. org/CorpusID:15222328.
- J Gloria Davis. An analysis of redundancy management algorithms for asynchronous fault tolerant control systems. Technical report, Ames Research Center, California, NASA, 1987.
- Analog Devices. ADIS1675 Datasheet. Accessed: 2025-07-10. URL: https://www.analog. com/en/products/adis16575.html.
- Ben L DiVito, Ricky W Butler, and James L Caldwell. Formal design and verification of a reliable computing platform for real-time control. Phase 1: Results. Technical report, NASA Langley Research Center, 1990.
- 10 Jack Goldberg, Milton W. Green, William H. Kautz Karl N. Levitt, P. Michael Melliar-Smith, Richard L. Schwartz, and Charles B. Weinstoc. Development and analysis of the software implemented fault-tolerance (sift) computer. Technical report, SRI International, Menlo Park,
- E.F. Hitt and D. Mulcare. Fault-tolerant avionics. In C.R. Spitzer and Spitzer, editors, Digital Avionics Handbook (1st ed.), chapter 28, pages 1-28. CRC Press, 2006. doi:https: //doi.org/10.1201/9781420036879.
- 12 GG1320 User manual. Accessed: Honeywell. 2025-07-10. URL: https: //aerospace.honeywell.com/content/dam/aerobt/en/documents/learn/products/ sensors/user-manuals/GG1320-usermanual.pdf.
- Benjamin F. Jones and Lee Pike. Modular model-checking of a byzantine fault-tolerant protocol. In NASA Formal Methods, pages 163-177, Cham, 2017. Springer International Publishing.
- M. A. Kassab, H. S. Taha, S. A. Shedied, and A. Maher. A novel voting algorithm for redundant aircraft sensors. In Proceeding of the 11th World Congress on Intelligent Control and Automation, pages 3741-3746, 2014. doi:10.1109/WCICA.2014.7053339.
- J.C. Knight. Safety critical systems: challenges and directions. In Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, pages 547–550, 2002.
- Ranjani Krishnan, Ashutosh Gupta, Nitin Chandrachoodan, and V R Lalithambika. Formal verification of clock synchronization algorithms for aerospace systems. In 2024 8th International Conference on Electronics, Communication and Aerospace Technology (ICECA), pages 1603– 1608, 2024. doi:10.1109/ICECA63461.2024.10800832.

- 17 Ranjani Krishnan, Ashutosh Gupta, Nitin Chandrachoodan, and VR Lalithambika. Formal verification of voting algorithms for safety critical systems using two approaches. In 2024 IEEE Space, Aerospace and Defence Conference (SPACE), pages 211–215. IEEE, 2024.
- 18 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. ACM Trans. Program. Lang. Syst., 4(3):382-401, July 1982. doi:10.1145/357172.357176.
- 19 Patrick Lincoln and John Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In *Computer Aided Verification*, pages 292–304, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.
- 20 Steven P. Miller, Alan C. Tribble, and Mats P. E. Heimdahl. Proving the shalls. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, pages 75–93, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- 21 Paul S Miner. Verification of fault-tolerant clock synchronization systems. Technical report, NASA Langley Research Center, 1993.
- M. Nicolaidis. Soft Errors In Modern Electronic Systems. Springer New York, NY, 01 2011. doi:10.1007/978-1-4419-6993-4.
- S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995. doi:10.1109/32.345827.
- 24 L. Pike. A note on inconsistent axioms in rushby's "systematic formal verification for fault-tolerant time-triggered algorithms". *IEEE Transactions on Software Engineering*, 32(5):347–348, 2006. doi:10.1109/TSE.2006.41.
- D. Powell. Failure mode assumptions and assumption coverage. In [1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing, pages 386–395, 1992. doi:10.1109/FTCS.1992.243562.
- Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. Velisarios: Byzantine fault-tolerant protocols powered by Coq. In *Programming Languages and Systems*, Cham, 2018. Springer International Publishing.
- 27 J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Transactions on Software Engineering, 25(5):651-660, 1999. doi:10.1109/32.815324.
- J.M. Rushby and F. von Henke. Formal verification of algorithms for critical systems. IEEE Transactions on Software Engineering, 19(1):13–23, 1993. doi:10.1109/32.210304.
- 29 John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '94, page 304–313, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/197917.198115.
- 30 John Rushby. An overview of formal verification for the time-triggered architecture. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 83–105, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
- 31 John Rushby and Frieder VonHenke. Formal verification of a fault tolerant clock synchronization algorithm. Technical report, NASA Langley Research Center, 1989.
- 32 Indranil Saha, Suman Roy, and S. Ramesh. Formal verification of fault-tolerant startup algorithms for time-triggered architectures: A survey. *Proceedings of the IEEE*, 104:1–19, 03 2016. doi:10.1109/JPROC.2016.2519247.
- Rocq Core Team. Program extraction. Accessed: 2025-07-08. URL: https://rocq-prover.org/doc/master/refman/addendum/extraction.html.
- 34 Rocq Core Team. Rocq prover. Accessed: 2025-07-08. URL: https://rocq-prover.org/.

# A Implementation Details

In this section we describe the details of our implementation of the voter's operational logic as per the design in the Section 4.

# A.1 Basic Data Types and their Invariants

As mentioned in Section 2, each input unit is assumed to have a unit\_id. This is defined as below.

```
Inductive unit_id :=
uid_con : nat -> unit_id.
```

The signal\_health type given below is used to represent the self-declared health status of input unit (good or bad).

```
Inductive signal_health :=
    | bad : signal_health
    | good : signal_health.
```

The signal type given below is used to represent measurement from an input unit (val  $\in \mathbb{N}$ ) along with a health status (hw\_hlth).

```
Record signal := signal_build {
   val : nat;
   hw_hlth : signal_health;
}.
```

A reading of type signal and uid of type unit\_id together constitute a unit\_output data type. As mentioned in Section 2, this represents the information received by the voter from each input unit.

```
Record unit_output := unit_output_build {
    reading : signal;
    uid : unit_id;
}.
```

As explained in the functional requirement, an accumulated status of the fault identification and isolation of each input unit is maintained by the voter. This is represented using the data type unit\_status defined below.

The proof term pf\_risky\_count is included as a field in unit\_status so as to satisfy the requirement R6 as an invariant.

The voter sees the data type unit\_data from each input unit. This is a record of unit\_output and unit\_status as defined below.

The proof term pf\_healthy carries the invariant that the risky\_count is zero if and only if the data of the input unit is a healthy\_data.

#### A.2 Voter State and its Invariants

As mentioned in Section 2, the voter receives input from a predefined number of input units. In our implementation, this is represented using a parameter  $num\_units \in \mathbb{N}$ . To indicate that the voter sees inputs from units with unique ids, we define a data type  $uid\_list$  which is a list of  $unit\_id$ , with an embedded proof that there is no duplication in the list.

A list named u\_ids of type uid\_list is the concrete list of unit\_id defined for the system in which the unit ids correspond to the sequence of numbers from 1 to num\_units.

The voter is implemented as a state transition system. States are represented using the voter\_state data type which is defined below.

```
Record voter_state := voter_state_build {
    u data 1st
                : list unit data;
    (* On unit switching, the next unit is selected as per the
       order in u_data_lst *)
    pf_ud_lst
                  : get_u_ids_of_snsr_data u_data_lst = u_ids.(1);
    voter_output : unit_output;
    voter_validity: validity_status;
    output_age
                 : nat;
    presrvd_data : unit_data;
   pf_v_output
                  : In (uid voter_output)
                            (get_u_ids_of_snsr_data u_data_lst);
    pf_presrvd_data : presrvd_data.(u_output) = voter_output
                            /\ healthy_data presrvd_data;
   pf_age
                  : output_age = 0 <->
                                (voter_validity = valid
                                /\ In presrvd_data u_data_lst);
    pf_validity : pf_validity_prop
                          voter_output voter_validity u_data_lst;
   pf_out_not_isolated : voter_validity = valid ->
                            In (uid voter_output)
                                (get_u_ids_of_snsr_data
```

```
non_isolated_list u_data_lst));
   pf_risky_cnt
                   : risky_cnt_age_prop pf_ud_lst
                                          pf_v_output
                                           output_age
                                           voter_validity;
    pf_age_bound
                    : voter_validity <> not_valid ->
                    forall x,
                      In x u data 1st
                      -> x.(u_status).(iso_status) = not_isolated
                      -> output_age - x.(u_status).(risky_count)
                              < persistence_lmt;</pre>
   pf_age_validity : (output_age < persistence_lmt</pre>
                               <-> voter validity = valid)
                       /\ (output_age >= 2*persistence_lmt
                              <-> voter_validity = not_valid)
}.
```

The important fields of voter\_state are as follows.

- u\_data\_lst, which is a list of unit\_data, with one element corresponding to each input unit, to keep the readings and status of the corresponding unit. The proof term pf\_ud\_lst indicates that the list of unit ids of the input units in the u\_data\_lst is the list u\_ids.
- voter\_output of type unit\_output that is assumed to be fed to the controller. The proof term pf\_v\_output states that the voter\_output of the voter is a measurement from one of the input units whose unit id is in the u\_data\_lst.
- woter\_validity of type validity\_status that is assumed to be fed to the controller
- **output\_age**  $\in \mathbb{N}$  that is assumed to be fed to the controller.
- **presrvd\_data**, which is the unit\_data of the prime unit selected for voter output generation. This information is required by the voter to decide the state transition.

The voter\_state definition also has the following proof terms specifying invariants to satisfy some of the functional requirements described in Section 3. The requirement R6 is an invariant of unit\_status, which is defined in Section A.1. The requirements R8, R10, R11, R13,R15, R16 and the second part of requirement R14 are embedded as invariants using the proof terms in the voter\_state definition.

- The requirement R8 is captured by the proof term pf presrvd.
- pf\_validity is a direct translation of the requirements R10, R11 and R13 relating the fields voter\_output, voter\_validity and the u\_data\_lst.

The translation of R10 is done as follows.

The translation of R13 is as follows.

■ The second part of requirement R14 is captured by the proof term pf\_risky\_cnt.

The translation of the second part of requirement R14 is done as follows.

```
voter_validity = valid ->
  let voter_out_data := find_data_of_a_given_unit pf_ud_lst pf_in in
  output_age = ( risky_count (u_status (proj1_sig voter_out_data)))
```

- The requirement R15 is captured by the proof term pf\_age in combination with the proof term pf\_presrvd.
- The requirement R16 and Proposition 15 are captured by the proof term pf\_age\_validity.
- pf\_age\_bound is a direct translation of Claim 14 and it is used for proving Proposition 15.
- pf\_out\_not\_isolated satisfies the invariant that if the voter\_validity is valid, then the prime unit is not isolated. This proof term helps in proving that R11 is maintained during a state transition.

# A.3 Voter State Update Rules and its Properties

Among the requirements, those related to a single voter\_state are embedded in the voter state definition as explained in the previous section. The remaining requirements are about the invariants to be satisfied during each state transition.

The state transition in each cycle is done by using the voter\_state\_transition function, which takes the previous cycle voter state vs and the current cycle unit\_output of each input unit and generates an updated voter state new\_vs. The function is designed in such a way that its output is the new voter state along with a proof term voter\_state\_transition\_prop which encodes the remaining requirements. The voter\_state\_transition function has two main steps.

- 1. The build\_updated\_u\_data\_lst function takes as input the u\_data\_lst (which is a list of unit\_data) of the previous cycle voter state vs and the current cycle unit\_output of all input units. It creates a new list of unit\_data denoted by new\_p\_ud\_lst in which the unit\_status of each input unit is updated as per the fault identification and isolation algorithm.
- 2. The unit\_status of input units in new\_p\_ud\_lst is used as the u\_data\_lst of the new voter state new\_vs. Based on the updated isolation status of input units available in new\_p\_ud\_lst, the assessment of whether the system satisfies maximum permanent fault assumption is done. Based on this, the remaining fields of the new voter state new\_vs are generated.

We will now describe the details of the steps given above.

#### A.3.0.1 Creating the updated unit\_data list new\_p\_ud\_lst.

To begin with, fault identification has to be done. For this, a filtered list of unit\_output, namely

all\_good\_non\_iso\_lst, is created based on the previous cycle voter state vs and the list of current cycle unit\_output.

This is done so as to filter out units previously isolated as per vs and units with self identifying health status bad in the current cycle, so that they are not used for identifying deviation faults. From all\_good\_non\_iso\_lst, two sub-lists of unit ids are computed. The list of unit ids which satisfy the premise of requirement R4 are identified as dev\_uid\_lst.

The list of unit ids which are not included in dev\_uid\_lst and which do not satisfy the premise of requirement R5 are identified as maybe\_uid\_lst.

The next step is to use the lists computed above to build the updated list of unit\_data, called new\_p\_ud\_lst. The current cycle unit\_output available as input to build\_updated\_u\_data\_lst is copied to the unit\_output of the corresponding element of new\_p\_ud\_lst. Now, the unit\_status of each input unit of new\_p\_ud\_lst needs to be computed from the unit\_status of the corresponding input unit of the u\_data\_lst of the previous cycle voter state vs. This is done as follows.

If the isolation status in the u\_data\_lst of vs is isolated, the isolation status in new\_p\_ud\_lst is isolated. The other fields in unit\_status of new\_p\_ud\_lst remain as in u\_data\_lst of vs. If the isolation status in u\_data\_lst of vs is not\_isolated, then the following rules are applied.

- If the self identified health status in current cycle unit\_output is bad, then risky\_count in new\_p\_ud\_lst is one more than that in u\_data\_lst of the voter state vs.
- If the self identified health status in current cycle unit\_output is good and if the uid is included in dev\_uid\_lst (resp. in maybe\_uid\_lst), the miscomparison status in new\_p\_ud\_lst is set to miscomparing (resp. is set to maybe\_miscomparing) and the risky\_count in new\_p\_ud\_lst is one more than that in u\_data\_lst of vs. This is required to meet the second part of the requirement R1.
- If the self identified health status in current cycle unit\_output is good and if the uid is not included in dev\_uid\_lst and in maybe\_uid\_lst, the miscomparison status in new\_p\_ud\_lst is not\_miscomparing and the risky\_count in new\_p\_ud\_lst is set to zero.
- When the risky\_count after updation reaches persistence\_lmt, the isolation status of that unit in new\_p\_ud\_lst is set to isolated.

It is proved that the above rules of creation of new\_p\_ud\_lst guarantee that requirements R1 to R5 and R7 are satisfied by the fault identification and isolation algorithm. Since new\_p\_ud\_lst is used as the u\_data\_lst of new\_vs, these requirements will also be satisfied by the voter state transition.

#### A.3.0.2 Voter Output Generation using new\_p\_ud\_lst.

The new voter state new\_vs uses new\_p\_ud\_lst created above as its u\_data\_lst. The other fields of new\_vs, such as validity\_status, voter\_output and output\_age are defined based on the updated information available in new\_p\_ud\_lst.

Using the count of units with isolation status as not\_isolated in new\_p\_ud\_lst, the decision whether the system satisfies the maximum permanent fault assumption is taken. If this count is less than min\_required, then as per requirement R10, new\_vs has validity\_status as not\_valid. In this case, the output\_age is set as 2\*persistence\_lmt and the remaining fields are unchanged from those of previous voter\_state vs. If the count is greater than or equal to min\_required, the rules used are the following.

As per new\_p\_ud\_lst, if the new isolation status of the unit used in the voter\_output of the previous state vs is isolated, then a healthy data list is computed from new\_p\_ud\_lst, which is the list of its elements with isolation status as not\_isolated, self identified health status good and miscomparison status as not\_miscomparing.

■ If the healthy data list is non-empty, then to be consistent with the requirement R13, the validity\_status of new\_vs is set to valid. The unit\_output and unit\_data from the first element in the list are respectively used as voter\_output and presrvd\_data of new\_vs. The output\_age is set as zero.

■ If the healthy data list is empty, then to be consistent with the requirement R11, the validity\_status of new\_vs is set to un\_id. The output\_age is one more than that of vs. The remaining fields are unchanged from those of previous voter\_state vs.

As per new\_p\_ud\_lst, if the new isolation status of the unit used in the voter\_output of the previous state voter state vs is not\_isolated, then the validity\_status of new\_vs is set as valid, to be consistent with the requirement R13.

- If as per new\_p\_ud\_lst, that unit is providing a healthy data, then the voter\_output and presrvd\_data of new\_vs are build using the unit\_data of that unit, available in new\_p\_ud\_lst and the output\_age is set to zero.
- Otherwise, the voter\_output and presrvd\_data of new\_vs are same as they were in the previous voter state vs. The output\_age is one more than that of the previous voter state vs.

#### A.3.0.3 Invariants Maintained during State Transitions.

As mentioned earlier, the state transition function is voter\_state\_transition. It takes the previous cycle voter state vs and the current cycle unit\_output of each input unit and generates a new voter state new\_vs, which satisfies a proof term voter\_state\_transition\_prop. The property voter\_state\_transition\_prop is an encoding of requirements R1 - R5, R7, R9 and R12 and first part of requirement R14.

We define a property simul\_fault\_prop, which is the translation of the *Simultaneous Fault Hypothesis* of the system. The property means the following:

The number of non-isolated units with bad health + the number of non-isolated units with good health and having at least  $\delta$  deviation from ground truth is  $\leq$  simul\_max\_fault. The requirement R1 is translated as below.

The first part of requirement R2 is translated as given below.

```
-> forall x, In x (u_data_lst vs)
            -> forall y, In y (u_data_lst new_vs)
           -> (uid (u_output x) = uid (u_output y))
           -> iso_status (u_status x) = not_isolated
           -> y.(u_status).(miscomp_status) <> maybe_miscomparing )
The completeness part of requirement R2 is translated as below.
( simul_fault_prop (pf_l u_ids) pf_all_unit_outputs (pf_ud_lst vs)
   -> min_required <= count_of_non_isolated_units vs.(u_data_lst)
   -> forall x, In x (u_data_lst vs)
   -> forall y, In y (u_data_lst new_vs)
   -> forall z, In z all_unit_outputs
   -> (uid (u_output x) = uid (u_output y))
   -> (uid (u_output x) = uid z)
   -> iso_status (u_status x) = not_isolated
   ->( let mis_flt_lmt := flt_lmt_among_good
   u_ids.(pf_l) pf_all_unit_outputs (pf_ud_lst vs) in
   let 1 := proj1_sig(all_good_non_iso_lst
   u_ids.(pf_l) pf_all_unit_outputs (pf_ud_lst vs) ) in
    length 1 > 2*mis_flt_lmt
    -> z.(reading).(hw_hlth) = good
    -> adiff ground_truth z.(reading).(val) > 3*delta
    -> y.(u_status).(miscomp_status) = miscomparing )
   /\( let mis_flt_lmt := flt_lmt_among_good
    u_ids.(pf_l) pf_all_unit_outputs (pf_ud_lst vs) in
   let 1 := proj1_sig (all_good_non_iso_lst
    u_ids.(pf_l) pf_all_unit_outputs (pf_ud_lst vs) ) in
   length 1 > 2*mis flt lmt
   -> In z 1
    -> adiff ground_truth z.(reading).(val) <= delta
   -> y.(u_status).(miscomp_status) = not_miscomparing)
The requirement R3 is translated as below.
( simul_fault_prop (pf_l u_ids) pf_all_unit_outputs (pf_ud_lst vs)
   -> min_required <= count_of_non_isolated_units vs.(u_data_lst))
   -> forall x, In x (u_data_lst vs)
   -> forall y, In y (u_data_lst new_vs)
   -> forall z, In z all_unit_outputs
   -> (uid (u_output x) = uid (u_output y))
   -> (uid (u_output x) = uid z)
   -> iso_status (u_status x) = not_isolated
```

-> ( (\* soundness a prop \*)

y.(u\_status).(miscomp\_status) = miscomparing

```
-> adiff ground_truth z.(reading).(val) > delta )
     (* soundness B prop *)
 /\ ( let l := proj1_sig (all_good_non_iso_lst (pf_l u_ids)
     pf_all_unit_outputs (pf_ud_lst vs)) in
 In z l -> y.(u_status).(miscomp_status) = not_miscomparing
 -> adiff ground_truth z.(reading).(val) <= 3*delta )
The requirement R4 is translated as below.
 In x (u_data_lst vs)
 -> forall y, In y (u_data_lst new_vs)
 -> forall z, In z all_unit_outputs
 -> (uid (u_output x) = uid (u_output y))
 -> (uid (u_output x) = uid z)
 -> iso_status (u_status x) = not_isolated
 ->
( let mis_flt_lmt := flt_lmt_among_good
 u_ids.(pf_l) pf_all_unit_outputs (pf_ud_lst vs) in
 let 1 := proj1_sig (all_good_non_iso_lst
 u_ids.(pf_l) pf_all_unit_outputs (pf_ud_lst vs) ) in
 In z l
 -> miscomparing_many_check l mis_flt_lmt z = true
 -> y.(u_status).(miscomp_status) = miscomparing )
The requirement R5 is translated as below.
 In x (u_data_lst vs)
 -> forall y, In y (u_data_lst new_vs)
 -> forall z, In z all_unit_outputs
 -> (uid (u_output x) = uid (u_output y))
 -> (uid (u_output x) = uid z)
 -> iso_status (u_status x) = not_isolated
 -> (let mis lst :=
              proj1_sig(miscomparing_lst
              (pf_l u_ids) pf_all_unit_outputs(pf_ud_lst vs) ) in
     let mis_flt_lmt := flt_lmt_among_good (pf_l u_ids)
                          pf_all_unit_outputs(pf_ud_lst vs) in
     let rem_mis_flt_lmt := mis_flt_lmt - length ( mis_lst ) in
     let gd_non_iso_lst := proj1_sig (all_good_non_iso_lst
                  (pf_l u_ids)pf_all_unit_outputs(pf_ud_lst vs) )in
     let negb_mis_lst
                          := filter(fun y =>
     {\tt negb \ (miscomparing\_many\_check}
              gd_non_iso_lst mis_flt_lmt y ) )gd_non_iso_lst in
     In z negb_mis_lst
      -> agreeing_many_check negb_mis_lst rem_mis_flt_lmt z = true
      -> y.(u_status).(miscomp_status) = not_miscomparing )
).
```

The requirement R7 is translated as below.

```
( forall x, In x (get_u_ids_of_unit_data
              (isolated_list (u_data_lst vs))) ->
              In x (get_u_ids_of_unit_data
                         (isolated_list (u_data_lst new_vs))))
The requirement R9 is described here.
 ( (uid (voter_output vs)) <> (uid (voter_output new_vs))
    -> In (uid(voter_output vs))
    (get_u_ids_of_unit_data (isolated_list (u_data_lst new_vs))))
The requirement R12 is translated as below.
  (simul_fault_prop (pf_l u_ids) pf_all_unit_outputs (pf_ud_lst vs)
     -> count_of_non_isolated_units (u_data_lst new_vs) >= min_required
     -> let mis_flt_lmt := flt_lmt_among_good
     u_ids.(pf_l) pf_all_unit_outputs (pf_ud_lst vs) in
     let 1 := proj1_sig (all_good_non_iso_lst
            u_ids.(pf_l) pf_all_unit_outputs (pf_ud_lst vs) ) in
     length 1 > 2*mis_flt_lmt
     -> voter_validity new_vs = valid )
The first part of requirement R14 is proved using the proof term described.
  ( voter_validity new_vs <> not_valid
   -> ( output_age new_vs =
      S (output_age vs) \/ output_age new_vs = 0 ))
```

It may be recalled that the other requirements are maintained as invariants of voter\_state, as mentioned in Section A.2.