Formal modeling and analysis of an avionics triplex sensor voter *

Samar Dajani-Brown, Darren Cofer, Gary Hartmann, and Steve Pratt

Honeywell Laboratories, Minneapolis MN
samar.dajani-brown@honeywell.com

Abstract. Digital flight control systems utilize redundant hardware to meet high reliability requirements. In this study we use the SMV model checker to assess the design correctness of a sensor voter algorithm used to manage three redundant sensors. The sensor voter design is captured as a Simulink diagram. The requirements verified include normal operation, transient conditions, and fault handling.
The sensor voter algorithm is a realistic example of flight critical embedded software used to manage redundant air data or inertial reference sensors. We are using it to evaluate different design methods, languages, and tools currently available for formal verification. Key issues are 1) integration of formal verification into existing development processes and tools, and 2) synthesis of the correct environment (world abstraction) needed for analysis of normal and off-normal operating conditions.

1 Redundant sensors in flight control

Early autopilots were allowed a minimum of control authority so they would not cause any serious disturbances if any component failed; they could easily be overpowered by pilot inputs. As the need for aircraft stability and precise flight path following increased, so did flight control authority. This lead to the need for monitoring and the addition of redundant sensors and monitor points to provide the needed information for failure detection (which would then automatically disengage the system). Over time this evolved into a completely redundant channel of sensors and control electronics. At the same time there was a growing demand for increased mission reliability and the need for fail-operative systems arose.

A key part of redundant systems focuses on managing redundant sensors to provide a high integrity measurement for use by down-stream control calculations. Cross-strapping sensors so downstream processing has access to multiple copies of the same variable is an important feature. The advantage of cross-strapping sensors among redundant control channels to improve system availability (or mission success) was recognized in the 1960s but was not exploited due to the limitations of the analog implementations. With the advent of digital

* This paper is based upon work supported in part by NASA under contract NAS1-00079.

flight control it was recognized that fault detection and isolation could be implemented in software. The main advantage of the digital flight control systems which appeared in the mid 1970s compared to earlier analog systems was the ability to handle monitoring, redundancy management, and built-in-test without adding more hardware.

Throughout the 1970s and 1980s many papers appeared describing various algorithms for managing redundant systems and redundant sensors. The NATO Advisory Group for Aerospace Research and Development (AGARD) has sponsored several publications and lecture series dealing with redundancy management in flight control [1],[2],[3]. A recent paper on the subject appears in [4] and textbooks such as [5] now contain chapters on redundancy management in flight control systems.

Many sensors include some internal monitors and provide a logic output to indicate that an internal hardware fault has been identified. These monitors provide on the order of 90 - 95 % failure detection without false alarms. Internal monitors are usually specific to a sensor type. Examples of such internal monitors include checks on power supply voltages, checks on whether ring laser gyros are "lasing", checks on whether vibrating beam accelerometers are at the proper resonant frequency and so forth. If these sensor valid signals are set "false", then the sensor is not used regardless of voter or comparitor algorithm decisions [4]. However, these valid flags are not adequate for detecting all sensor faults; hence the need for real-time software monitors operating at the sampling rate of the sensors.

Sensors exhibit various kinds of deterministic and non-deterministic errors including bias offsets, scale factor errors, and sensitivity to spurious input and environmental factors. The question of what constitutes a "failed" sensor involves certain subtleties. These are reflected in the fact that situations exist, such as when the quantity being measured is zero, in which the behavior of a perfectly functioning instrument is indistinguishable from the behavior of one that is not working. In practice, sensors often fail by small degrees so that the indicated measurement becomes contaminated with nearly unobservable errors. For operational purposes, we can define a sensor as failed when it is contributing measurement errors sufficiently large as to jeopardize the mission. What this imposes on the sensor redundancy management is the requirement that it be able to isolate or tolerate any failure large enough to jeopardize the mission.

Sensor failure detection algorithms ("voters") must detect and isolate a sensor whose output departs by more than a specified amount from the normal error spread. The detailed design of the voter algorithm combined with the downstream control law determines the magnitude of the transient the aircraft may experience as a result of the failed sensor being disconnected. Thus two conflicting requirements emerge:

1. Provide a very low number of nuisance disconnections – if the thresholds are too low a sensor can be disconnected when it is merely at the edge of its tolerances. To further minimize nuisance trips many algorithms may require errors to "persist" for some amount of time before declaring a fault.

2. Provide a minimum transient on disconnecting a failed sensor – if the thresholds are too high when a real failure occurs the magnitude of the resulting transient can be unacceptably large.

The generic voter used in this study is representative of algorithms in use. Many of its features are taken from [4]. This class of algorithm is applicable to a variety of sensors used in modern avionics, including rate gyros, linear accelerometers, stick force sensors, surface position sensors, and air data sensors (e.g. static and dynamic pressures and temperature). Sensor sample rates are based on the bandwidth of the control loops using the measurements; typical values in flight control applications range from 10 - 100 Hz.

Traditionally, the performance of sensor management algorithms are evaluated using simulated sensor failures together with detailed FMEAs to tune design parameters and establish the correctness of the design. However, this approach cannot guarantee that the worst case combination of sensor inputs and conditions has been indentified and that the design meets its performance and safety requirements under these conditions. In this study we will apply formal methods instead of extensive simulations to establish the correctness of the design.

2 Sensor voter

Simulink [11] is a computer aided design tool widely used in the aerospace industry to design, simulate, and autocode software for avionics equipment. The Simulink diagram representing the sensor management algorithm (Figure 1) incorporates the typical attributes of a sensor management algorithm and is intended to illustrate the characteristics of such algorithms. The design is that of a generic triplex voter utilizing features appearing in the open literature. The voter takes inputs from three redundant sensors and synthesizes a single reliable sensor output. Each of the redundant sensors produces both a measured data value and self-check bit (validity flag) indicating whether or not the sensor considers itself to be operational. The output of a sensor is amplitude limited in hardware by the A/D conversion (+/-20 in the Simulink model). The functionality of the triplex voter is as follows:

1. Sample digitized signals of each sensor measurement at a fixed rate appropriate for the control loop, e.g. 20 Hz. A valid flag supplied by sensor hardware indicating its status is also sampled at the same rate.
2. Use the valid flag and comparison of redundant sensor measurements to detect and isolate failed sensors.
3. Output at a specified sample rate a signal value computed as a composite averaging of the signals of non-faulty sensors, i.e. remove failed sensors from use in computing average. Also, output, at the same specified rate, the status of each sensor by setting a hardware flag "outputValid" flag.
4. Tolerate "false alarms" due to noise, transients, and small differences in sensor measurements. Sensors are not marked failed if they are operating within acceptable tolerances and noise levels.

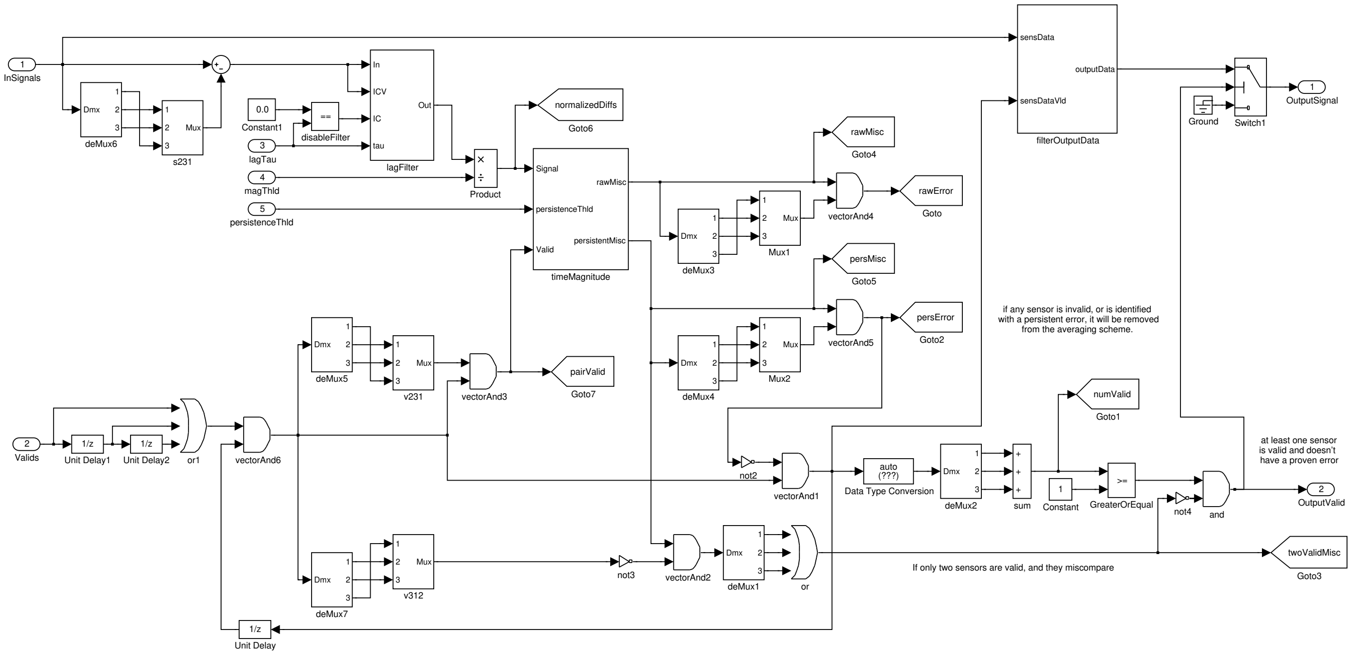


Fig. 1. Simulink diagram of the sensor voter.

5. Maximize the availability of valid output by providing an output whenever possible, even with two failed sensors.
6. The algorithm is not required to deal with simultaneous sensor failures since this is a very low probability event.

Sensor faults can be any combination of sudden bias shifts, ramps, or oscillatory faults up to a maximum amplitude of 20. Oscillatory faults are often specifically defined. For this example, oscillatory sensor failures in the frequency range 3 - 7 Hz with amplitudes equal to or greater than 2 unit (zero to peak) should be detected in 1.5 second or less. In general, the “worst case” failure is unknown.

The operation of the sensor voter algorithm is as follows. All valid sensor signals are combined to produce the voter output. If three sensors are available, a weighted average is used in which an outlying sensor value is given less weight than those that are in closer agreement. If only two sensors are available a simple average is used. If only one sensor is available, it becomes the output.

There are two mechanisms whereby a faulty sensor may be detected and eliminated: comparison of the redundant sensor signals and monitoring of the validity flags produced by the sensors themselves.

The differences between each pair of the three input signals are initially computed; i.e., signal 1 is subtracted from 2, 2 from 3 and 3 from 1. These differences then pass through a limiter and a lag filter to remove unwanted noise. In our current version of the model the limiter and lag filter have been disabled since they have no effect given the range of inputs we are considering. Differences that exceed a given magnitude threshold cause a counter in the persistence threshold block to increment by one. Differences below the threshold cause the counter to decrement, but not below zero. When the counter reaches the persistence threshold for two of the pair differences, a persistent miscompare is detected and the sensor that is common to the two pairs is then eliminated from the output average computation. For example, if sensor 1 and sensor 2 have a persistent miscompare and sensor 1 and sensor 3 have a persistent miscompare, then sensor 1 is flagged with a persistent error and is not used.

If the hardware valid signal produced by a sensor is false for three consecutive samples, that sensor is considered to be faulty. The faulty sensor signal is not used in failure comparisons or in the production of the output signal.

3 Requirements

Behavioral requirements for the sensor voter fall into two categories:

1. Computational, relating the value of the output signal computed by the voter.
2. Fault handling, relating to the mechanisms for detecting and isolating sensor failures.

Each of these categories includes requirements for reliability (correctness under normal operation) and robustness (rejection of false alarms).

3.1 Computational requirements

The main purpose of the sensor voter is to synthesize a reliable output that agrees with the "true" value of the environmental data measured by the redundant sensors. Therefore under normal operation, the output signal should agree with this true value within some small error threshold. In the absence of sensor noise or failures the two values should agree exactly. During the interval between the failure of a sensor and the detection of the failure by the voter, it is expected that the output value will deviate from the true value due to the continued inclusion of the failed sensor in the output average. During this reconfiguration interval the transient error in the output signal must remain within specified bounds, regardless of the type or magnitude of the failure.

The acceptable transient error has bounds in both magnitude and time, and is different for the first and second faults detected. The first failure transient must not exceed a magnitude of 0.1 with a duration of less than 0.15 seconds (corresponding to 3 sample periods at 20Hz). The second fault transient must not exceed 10 units with a duration of less than 0.5 seconds (10 samples). These bounds assume that the sensor inputs are limited to +/- 20 units (based on the A/D scaling).

3.2 Fault handling requirements

The required fault handling behavior for the voter is shown in Figure 2. Initially, all three sensors are assumed to be valid. One of these sensors may be eliminated due to either a false hardware valid signal from the sensor or a miscomparing sensor value, leading to the "2 valid" state. If one of the two remaining sensors sets its hardware valid signal false, it is eliminated leading to the "1 valid" state. If this sensor subsequently sets its valid flag false it is eliminated and the voter output is set to not valid.

A special situation occurs when there are two valid sensors. If these sensors miscompare, the voter cannot determine which may be faulty. Although there are other possibilities, this voter algorithm continues to keep both sensors in service but it sets its output valid flag false. If the sensors subsequently agree in value, the voter returns to the "2 valid, no miscompare" state and sets its output valid flag to true. Alternatively, if one of the two sensors identifies itself as faulty (via the hardware valid flag) it can be isolated by the voter and the other sensor signal used as the correct output value.

If there are only two valid sensors and they have a persistent miscompare, neither sensor is used and the voter output valid flag is set to false. The output valid flag is also set to false if no sensors are valid. In these cases, the output signal is set arbitrarily to zero.

Robustness requirements apply to both the sensor signal comparisons and the hardware valid flags. No sensor is eliminated until it miscompares with others by a magnitude of at least 0.6 units for at least 0.5 seconds. In addition, any valid flags set to false must persist for three consecutive samples before the sensor is eliminated.

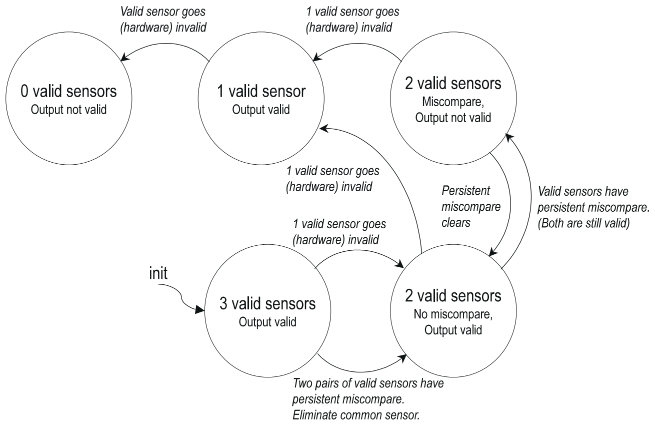
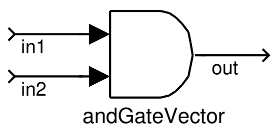


Fig. 2. Fault states of the sensor voter.

4 Modeling the voter and its environment

SMV is a symbolic model checker developed by CMU and was primarily developed with hardware verification problems in mind [6], [8]. While Simulink models are generally implemented in software (manually or using automatic code generation tools), the data flow block diagram representation resembles a hardware design. Therefore, it seems reasonable to apply SMV to this problem.

The SMV model we developed captures the design of the sensor voter by directly translating the Simulink diagram into SMV. Each Simulink block corresponds to an SMV module with the same inputs and outputs (Figure 3). As a result, the SMV representation is easy to understand and trace to the original Simulink representation. Furthermore, it should be possible to automate most of the translation process.




```
MODULE andGateVector(in1, in2)
VAR
out : array 0 .. 2 of boolean;
ASSIGN
out[0] := in1[0] & in2[0];
out[1] := in1[1] & in2[1];
out[2] := in1[2] & in2[2];
```

Fig. 3. Typical Simulink block translated to SMV module.

We are aware of research tools such as sf2smv [9] and Checkmate [10] that have been developed to automatically translate Simulink models into SMV for model checking and representation of counterexamples. However, these tools address only limited portions of the Simulink syntax and were not applicable to our problem. Our concern in this project is not so much the translation process, but the applicability of model checking to problems of this type.

4.1 Environment

The overall SMV model for verification of the sensor voter is shown in Figure 4. The model includes new modules that represent the environment driving the voter. These modules are the sensor modules and the world module.

The world module generates the data that sensors are to measure. In our current model it produces arbitrary data within the valid range, but it could easily be modified to produce data conforming to some frequency or derivative limitations.

The modules sensor1, sensor2, and sensor3 represent physical sensors that generate the measured signal and valid flags that are provided to the voter. These sensor modules are also used to inject faulty behavior to test the ability of the voter to identify and isolate failed sensors.

The sensorVoter module is the only part of the model corresponding to the real system we wish to analyze. It is implemented as two large modules and a number of small modules, following the hierarchical Simulink representation. Its outputSignal value can be compared to the “true” value produced by the world module to evaluate the performance of the voter.

The distinction between the system under study (the voter) and its environment (the world and sensors) is an important one. To produce a convincing argument in support of certification, the voter should be modeled with the highest possible fidelity. Its structure should be traceable to the original design and it should conform as closely as possible to code generated from the design. It should include a minimal number of abstractions. On the other hand, the level of abstraction used in the environment must be carefully optimized. We must ensure that the environment will exercise all possible behaviors of the voter (including fault conditions) without introducing any unrealistic behaviors.

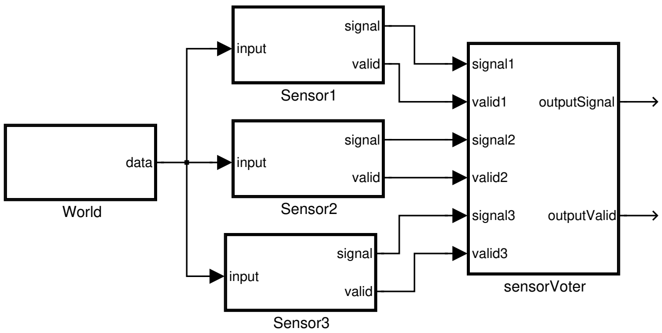


Fig. 4. SMV modules for the voter and environment.

4.2 Assumptions and simplifications

The following assumptions and simplifications have been made in modeling the sensor voter.

No Simultaneous Sensor Failures The algorithm assumes that two sensors cannot fail exactly at the same time. In particular, the first sensor failure must be detected and isolated by the voter before it is able to respond to a second failure. This fault hypothesis is reasonable if sensor failures are independent so that the probability of simultaneous failures is sufficiently low.

We note that under this assumption the number of valid sensors plus the number of sensors declared faulty is more or less constant. That is, when a sensor becomes faulty this sum will temporarily increase by one (the number faulty increments) and when the fault is detected it will decrease by one (the number valid decrements).

This assumption was handled in the SMV model by using the INVAR declaration. The INVAR declaration specifies an invariant that every state in the transition system must satisfy, thus restricting the transition relation to only those states. The invariant used in our model is that the number of valid sensors plus the number of faulty sensors cannot be less than 3 nor greater than 4.

```
INVAR
3 ≤ sensorVoter.numValid + sensorVoter.numFaulty
INVAR
sensorVoter.numValid + sensorVoter.numFaulty ≤ 4
```

NumFaulty is computed within the environment as sensors become faulty. However, numvalid is computed within the voter and its correctness must be verified independently.

Time model The SMV model does not explicitly model time. Each execution step in the model corresponds to one sample in the Simulink design, independent of the actual sample rate. The only place where time enters into the model is a gain constant used in the "Time Magnitude" module. This constant serves to convert the number of time steps that a sensor miscompare persists into a time for comparison with the persistenceThreshold time. We have let this value be 1 and adjusted the persistenceThreshold accordingly so that all values will be integers.

Sensor Signals The sensor signals used in the voter design are floating point numbers in the range from -20 .. 20. The sensor signals in our initial SMV model are restricted to be integers. We have also limited the range to reduce the state space of the model during implementation and debugging.

4.3 Translation of Simulink to SMV

The SMV model contains three main modules: realWorld(), sensor(), and voting3Signals().

Module realWorld() This module takes no arguments and non-deterministically generates integer values in a fixed range $\{a..b\}$, currently $\{1, 2, 3\}$. We have used larger ranges of input values but with significant increase in verification time.

Module sensor(world) This module takes the output of realWorld() for an argument and contains 3 internal variables, signal, valid, and fault. The variable fault is initialized to 0 in the first cycle or sample. For the next cycle, fault is assigned nondeterministically any value between 0 and 2. Once fault is assigned a value of 1 or 2, then that value remains unchanged throughout the remaining cycles.

A fault value of 1 means, in our model, that the sensor's signal is faulty, at which point the sensor broadcasts an arbitrary value as signal. Similarly, a fault value of 2 implies that the hardware flag for the sensor became invalid, at which point the hardware flag fluctuates between true and false. Our model assumes that once a sensor is faulty, whether in its signal or hardware flag, then it remains faulty for the rest of the computation. A non-faulty sensor, on the other hand, broadcasts the signal passed from the world as well as a hardware flag of true. In our model, we are not taking into account the realistic situation that a sensor may have hardware and signal failure simultaneously. However, we chose to model the sensor in such a manner as an initial investigation of the problem.

Module voting3Signals() This module takes 4 arguments as input: an array of signals of all 3 sensors, array of hardware flags for all 3 sensors, and 2 constants defining fault thresholds. This module has several variables, most of which are instances of other modules. The variables' names preserve the name of the logic blocks in the Simulink Diagram, Figure 1.

5 Analysis

The requirements for the sensor voter have been translated to computation tree logic (CTL) specifications and analyzed using SMV. CTL is a temporal logic having connectives that refer to the future [7]. An initial set of specifications derived mostly from the fault handling requirements are discussed in this section, along with the analysis results. These specifications have been derived from the fault handling state machine in Figure 2 and specify what conditions can and cannot occur at any time. We also consider some specifications dealing with the validity of the computed output. Note that the variable names in the detailed specifications match the variable names in the Simulink diagram in Figure 1.

5.1 Fault handling

We need to verify that faulty sensor detection and elimination is final. The following SMV SPECS verify this desired property.

Three valid sensors Given 3 valid sensors, once a sensor is detected and isolated based either on a hardware fault of persistent miscompare then this isolation is final.

SPEC

`AG(sensorVoter.numValid` $= 2 \rightarrow \neg$ `EFsensorVoter.numValid` $= 3$`)`

This SPEC is true. It states that it is globally true that if the number of valid sensors is 2 then there is no execution path in which that number may become 3 again, i.e. once a sensor is eliminated, then it stays eliminated.

Two valid sensors A similar SPEC portrays that the same argument holds when only 2 sensors are valid.

SPEC

`AG(sensorVoter.numValid` $= 1 \rightarrow \neg$ `EFsensorVoter.numValid` $= 2$`)`

This SPEC is true and states that it is globally true that there is no execution path where the number of valid sensors changes from 1 to 2.

One valid sensor Given only one valid sensor, we need to verify that should this sensor fail then it cannot recover.

SPEC

`AG(sensorVoter.numValid` $= 0 \rightarrow \neg$ `EF sensorVoter.numValid` $= 1$`)`

This specification is true and states that it is globally true that there is no execution path where the number of valid sensors changes from 0 to 1.

The above specifications do not test the situation where 2 valid sensors deviate in signal. The sensor management algorithm is designed in such a way that the number of valid sensors changes from 2 to 1 only if one of the 2 valid sensors has a hardware invalid flag. If, on the other hand, the 2 valid sensors deviate in signal, then their output is declared invalid, but the two sensors remain in service allowing for the situation that the two sensors may conform and produce the same signal at a later point in time or that the faulty sensor may identify itself via the hardware valid flag. This desired behavior is explored in the 2 specifications described below.

Two miscomparing sensors The first specification requires that if the 2 valid sensors deviate in signal then the output is declared invalid but the sensors remain in service. The second specification explores the scenario where 1 of the remaining sensors sets its hardware invalid flag. At that point, the sensor with the invalid hardware flag is isolated and the sensor with the valid hardware flag is trusted as the only valid sensor and its signal is considered as the valid output signal.

SPEC
EF(A[(sensorVoter.numValid $= 2\wedge$ sensorVoter.outputValid)U
(sensorVoter.numValid $= 2 \wedge \neg$ sensorVoter.outputValid)]) $\wedge$
EF(A[(sensorVoter.numValid $= 2 \wedge \neg$ sensorVoter.outputValid)U
(sensorVoter.numValid $= 2\wedge$ sensorVoter.outputValid)])

SPEC
EF(A[(sensorVoter.numValid $= 2 \wedge \neg$ sensorVoter.outputValid)U
(sensorVoter.numValid $= 1\wedge$ sensorVoter.outputValid)])

The above set of specifications for the fault handling behavior of the voter was verified to be true using SMV. However, all the specifications described so far do not take into consideration the specific error conditions that trigger the particular fault states. A general specification pattern that would cover each of the state transitions in Figure 2 could be of the form:

SPEC
AG((state$_i$ $\wedge$ errorCondition) $\rightarrow$ AF(state$_j$))

That is, in state$_i$ the occurrence of errorCondition must eventually lead to state$_j$. For example, when there are three valid sensors the occurrence of an invalid sensor flag must cause the voter to eliminate the sensor and go to the two valid sensors state.

Let $\mathcal{V} = (\mathcal{S}, \delta, \mathcal{F})$ be the sensor voter model, where $\mathcal{S}$ is the set of states the sensor voter can be in as based on the fault-handling FSM in Figure 2 , δ is a transition relation, and $\mathcal{F}$ is a set of the different sensor faults possible, i.e. $\mathcal{F} = \{f_0, f_1, f_2, f_3\}$ where f_0 is defined to be no sensor failure, f_1 is defined to be the first sensor failure, f_2 is the second sensor failure and f_3 is the third sensor failure. We know that $\mathcal{S} = \{s_0, s_1, s_2, s_3, s_4\}$, where s_0 is the state of 3 valid sensors, valid output and no sensor failure, s_1 is the state of 2 valid sensors, valid output and one sensor failure detection and isolation, s_2 is the state of 2 valid sensors, invalid output, one sensor failure detection and isolation and one sensor failure detection, s_3 is the state of one valid sensor, valid output and 2 sensor failure detection and isolation, and finally s_4 is the state of invalid output and no valid sensors. The transition relation $\delta : \mathcal{S} \times \mathcal{F} \rightarrow \mathcal{S}$ follows Figure 2.

To be more specific, we would like to able to show that

1. AG(($s_0 \wedge f_1$) $\rightarrow$ AF s_1),where $s_0, s_1 \in \mathcal{S}$,and $f_1 \in \mathcal{F}$
2. AG(($s_1 \wedge f_2$) $\rightarrow$ AF ($s_2 \vee s_3$)),where $s_1, s_2 \in \mathcal{S}$,and $f_2 \in \mathcal{F}$
3. AG(($s_2 \wedge f_1$) $\rightarrow$ EF s_1),where $s_2, s_1 \in \mathcal{S}$,and $f_1 \in \mathcal{F}$
4. AG(($s_2 \wedge f_2$) $\rightarrow$ EF s_3),where $s_2, s_3 \in \mathcal{S}$,and $f_2 \in \mathcal{F}$
5. AG(($s_3 \wedge f_3$) $\rightarrow$ AF s_4),where $s_3, s_4 \in \mathcal{S}$,and $f_3 \in \mathcal{F}$

We tested AG(($s_0 \wedge f_1$) $\rightarrow$ AF s_1), where $s_0, s_1 \in \mathcal{S}$,and $f_1 \in \mathcal{F}$ as follows:

SPEC
`AG( (sensorVoter.numValid` $= 3\wedge$ `sensorVoter.outputValid` $\wedge f_1$`)` $\rightarrow$
`AF (sensorVoter.numValid = 2` $\wedge$ `sensorVoter.outputValid))` where
f_1`=(sensor1.fault` $\neq$ `0` $\wedge$ `sensor2.fault` $\neq$ `0` $\wedge$ `sensor3.fault =0)` $\vee$
`(sensor1.fault=0` $\wedge$ `sensor2.fault` $\neq$ `0` $\wedge$ `sensor3.fault=0)` $\vee$
`(sensor1.fault=0` $\wedge$ `sensor2.fault=0` $\wedge$ `sensor3.fault` $\neq$ `0).`

SMV produced a counter example corresponding to the scenario in which a sensor enters the faulty state but never manifests faulty behavior.

The choices we made in modeling of the environment driving the sensor voter allow for the counter example produced. Given the critical importance of a sensor voter in avionics, we are interested in proving that faulty sensors, once detected, will get eliminated. We therefore eliminated from our environment the behavior that allows a faulty sensor not to manifest faulty behavior. The above SPEC is true after such an elimination.

Similarly the specification AG(($s_1 \wedge f_2$) $\rightarrow$ AF ($s_2 \vee s_3$)),where $s_1, s_2 \in \mathcal{S}$,and $f_2 \in \mathcal{F}$ produces the counter example where a faulty sensor does not manifest faulty behavior. However, the specification is true after eliminating from the environment such sensor behavior.

Testing AG(($s_1 \wedge f_2$) $\rightarrow$ AF (s_3)) yields false; the counter example, as expected, is $\delta(s_1, f_2) = s_2$.

We verified that AG(($s_2 \wedge f_1$) $\rightarrow$ EF s_1) is correct. The detailed specification is below:

SPEC
`AG( (sensorVoter.numValid = 2` $\wedge$ $\neg$ `sensorVoter.outputValid)` $\rightarrow$
`EF (sensorVoter.numValid = 2` $\wedge$ `sensorVoter.outputValid))`
The choice of AG(($s_2 \wedge f_1$) $\rightarrow$ EF s_1) and not AG(($s_2 \wedge f_1$) $\rightarrow$ AF s_1) is because we want to verify that s_1 is reachable from s_2 in certain execution paths and not all execution paths.

Similarly, in AG(($s_2 \wedge f_2$)$\rightarrow$ EF s_3, we want to verify that s_3 is reachable from s_2 in certain execution paths and not all execution paths. However, an interesting result shows up while testing AG(($s_2 \wedge f_2$) $\rightarrow$ EF s_3) because our second invariant does not allow us to transition from s_2 to s_3.

The following specification produces a counter example.

SPEC
`AG( (sensorVoter.numValid = 2` $\wedge$ $\neg$ `sensorVoter.outputValid)` $\rightarrow$
`EF (sensorVoter.numValid = 1` $\wedge$ `sensorVoter.outputValid))`

The error produced is the following: Sensor1 gets a faulty signal, miscompares with sensors 2 and 3 and is eliminated (3 $\leq$ numFaulty + numValid $\leq$ 4; numValid = 2). Sensor2 then gets a faulty signal and miscompares with sensor3 causing outputValid to become false while numValid stays 2. At this point, numValid + numFaulty = 4. To transition from the current state, we need one of the sensors to produce a hardware invalid flag. However, this would violate our INVAR assumption. Further refinements are required to accurately capture the fault hypothesis.

5.2 Computation of output

Our work so far has concentrated on verification of the fault-handling requirements of the voter. We are currently working on verification of the requirements for the voter output signal. The following discussion covers some of the relevant specifications that we are analyzing.

Output signal The algorithm is required to correctly compute the output signal. The specification below attempts, though weakly, to capture the correctness of this requirement.

SPEC
```
AG(sensorVoter.outputSignal.out ≠ world.out →
AF (A[sensorVoter.outputSignal.out ≠ world.out U
(sensorVoter.outputSignal.out = world.out ∧
sensorVoter.outputValid)] ∨
(sensorVoter.outputSignal.out ≠ world.out)))
```

This specification states that when the output signal of the sensors is different than the environment's signal, then either the sensors will eventually stabilize by voting the faulty sensor or it may be the case that the output will never stabilize.

Output signal bounds It should always be the case that when the voter outputValid is true, then the voter outputSignal should be within a set error tolerance of the true world value. This tolerance will be a function of the noise introduced by the sensor model and any fault conditions introduced. With no noise and no faults they should agree exactly. The desired specification will be of the form:

SPEC
```
AG(sensorVoter.outputValid →
| world.data - sensorVoter.outputSignal | < errorThresh)
```

Output transient bounds The requirements bounding the difference between the voter output and the true signal during a transient are based on establishing a time window around the transient.

The first and second transients should satisfy the following specifications:

SPEC `--first transient`
```
AG(sensorVoter.numValid = 3 → transient.timer < 3)
```

SPEC `--second transient`
`AG(sensorVoter.numValid = 2 → transient.timer < 10)`

6 Conclusion

Our future plans in this study include the following work:

1. Increasing the signal range and checking the specifications using a magnitude threshold greater than 1. We will assess the relationship between the signal range and the performance of the model checker (state space, analysis time).
2. Analysis of additional specifications of the voter behavior and refinement of the environment model.
3. We are also researching automation of the translation process to generate an SMV model from a Simulink model. Wide spread and practical use of model checking in this domain will require automation of this sort.
4. In addition, we plan to use other model checking tools to examine the correctness of the sensor voter algorithm.

References

1. P. Kurzhals, et al, "Integrity in Electronic Flight Control Systems", AGARDograph No. 224, April 1977. (available through National Technical Information Service, Springfield, VA)
2. T. Cunningham, et al, "Fault Tolerance Design and Redundancy Management Techniques", NATO AGARD Lecture Series No. 109. Sept 1980. (available through National Technical Information Service, Springfield, VA) see especially Chapter 3: Computer Based In-flight Monitoring; Chapter 7: Failure Management for Saab Viggen JA-37 Aircraft; Chapter 8: Flight Experience with Flight Control Redundancy Management
3. G. Belcher, D. McIver and K. Szalai, "Validation of Flight Critical Control Systems", AGARD Advisory Report No. 274, Dec 1991. (available through National Technical Information Service, Springfield, VA)
4. S. Osder, "Practical View of Redundancy Management Application and Theory", Journal of Guidance and Control, Vol. 22 No. 1 , Jan-Feb 1999.
5. R.P.G. Collinson, Introduction to Avionics, Chapman & Hall, London, 1998.
6. K. McMillan, Symbolic Model Checking , Kluwer Academic Publishers, Boston, Dordrecht, London, 1993.
7. Micheal R A Huth and Mark D Ryan, Logic in Computer Science Modelling and reasoning about systems, University Press, Cambridge, United Kingdom, 2000.
8. SMV web page: `http://www-2.cs.cmu.edu/ modelcheck`
9. sf2smv web page: `http://www.ece.cmu.edu/ webk/sf2smv`
10. Checkmate web page: `http://www.ece.cmu.edu/ webk/checkmate`
11. Simulink weg page: `http://www.mathworks.com/products/simulink`