Formal verification of an avionics sensor voter using SCADE *

Samar Dajani-Brown †, Darren Cofer †, and Amar Bouali ‡

†Honeywell Laboratories, Minneapolis, MN, U.S.A
‡Esterel Technologies, Villeneuve-Loubet, France
samar.dajani-brown@honeywell.com

Abstract. Redundancy management is widely utilized in mission critical digital flight control systems. This study focuses on the use of SCADE and its formal verification component, the Design Verifier, to assess the design correctness of a sensor voter algorithm used for management of three redundant sensors. The sensor voter algorithm is representative of embedded software used in many aircraft today. The algorithm is captured as a Simulink diagram; it takes input from three sensors and computes an output signal and a hardware flag indicating correctness of the output. This study is part of an overall effort to compare several model checking tools to the same problem. SCADE is used to analyze the voter's correctness in this part of the study. Since synthesis of a correct environment for analysis of the voter's normal and off-normal behavior is a key factor when applying formal verification tools, this paper is focused on 1) the different approaches used for modeling the voter's environment and 2) the strengths and shortcomings of such approaches when applied to the problem under investigation.

1 Overview of Sensor Voter Problem

With the advent of digital flight control systems in the mid 1970s came the capability to implement monitoring, redundancy management, and built-in-test functions in software without the need for additional hardware components. The sensors used in these flight control systems exhibit various kinds of deterministic and non-deterministic errors and failure modes including bias offsets, scale factor errors, and sensitivity to spurious input and environmental factors. Redundant sensors are used to compensate for these errors and failures. 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. Publications such as [7] and [8] describe redundancy management schemes used in flight control systems.

This paper builds on earlier work in [5] and is part of an overall effort to compare several model checking tools when analyzing the correctness of avionics components. We have used as a test case a typical voter algorithm with many

---

of the features taken from [7]. This class of algorithms 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).

Formal methods techniques, though embryonic to software development, have been used in [5] and [6] to analyze or verify safety critical properties of avionic software. We have found that the most challenging aspect of any formal verification effort is the specification of the environment in which the verified system operates. The environment captures all the assumptions about how the system interacts with the rest of the world, including physical constraints, timing considerations, and fault conditions. It should permit all interesting behavior of the system, while prohibiting any unrealistic behaviors. Much of our work has centered on creating good environment models for the sensor voter.

In this study, we use the SCADE suite and its formal verificatio component, the Design Verifier, to analyze correctness of the voter's algorithm.

1.1 Sensor Voter Algorithm

Simulink [12] is a computer aided design tool widely used in the aerospace industry to design, simulate, and auto-code software for avionics equipment. The voter's algorithm was developed in Simulink. It incorporates the typical attributes of a sensor management algorithm and is intended to illustrate the characteristics of such algorithms. 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. Data flow of the system is shown in Figure 1. A brief description of the voter's design and functionality follows:
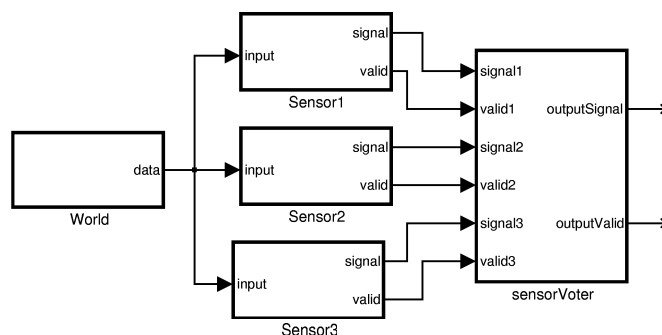

Fig. 1. Voter model and environment.

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 average of the signals of non-faulty sensors. Also output, at the same specified rate, the status of the composite output by setting a *ValidOutput* 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.
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.

A more detailed description appears in earlier work using the symbolic model checker SMV [11] [9] to analyze correctness of the voter's algorithm [5].

1.2 Sensor voter requirements

Behavioral requirements for the sensor voter fall into two categories:

1. **Computational**, relating to the value of the output signal computed by the voter.
2. **Fault handling**, relating to the mechanisms for detecting and isolating sensor failures. The required fault handling behavior of the voter is shown in Figure 2.

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

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).

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
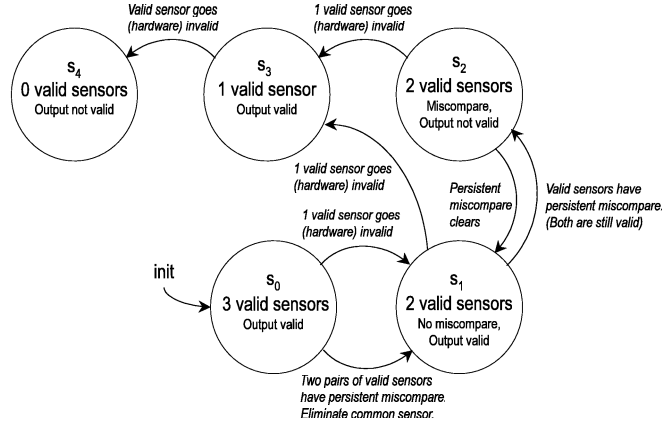

Fig. 2. Fault states of the sensor voter.

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.

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

2 Overview of SCADE

SCADE (Safety Critical Application Development Environment) [1] is a tool suite for the development of real-time embedded software. It provides a programming language called SCADE, a simulation environment, automatic code generation, and formal verification.

2.1 The SCADE design language

SCADE is a graphical deterministic, declarative, and structured data-flow programming language based on the Lustre language [1,2]. SCADE has a synchronous semantics on a cycle-based and reactive computational model. The node is the basic operator or design block of the language and can be either graphical or textual. A control system is modeled in SCADE via nodes connected to one another in a manner similar to how control systems get modeled in Simulink. Some of the advantages of SCADE are that while developing the model, the simulator component can be used to check for syntax and semantic correctness of the nodes. SCADE is a strongly typed language with predefined and user defined types. The language allows for arithmetic operations on real and integer values, logical operation on Boolean variables, control flow operations (if then else, case) and temporal operators to access values from the past.

Figure 3 shows a graphical SCADE node programming a simple counter of the occurrence of an event. The occurrence of an event is given as a Boolean input flow called `Event`. The counter produces an output integer flow called `Count`. The Lustre textual data-flow equations counter-part of the graphical node are listed in the figure as well.

The fby operator is a memory element, whose output is equal to its input value after a fixed number of cycles. For instance, `_L2 = fby(_L1 , 1 , 0)` means `_L2` is equal to `_L1` after 1 cycle. At the initial cycle, `_L2` is equal to 0. The rest of the node behavior is straightforward.

2.2 Formal verification in SCADE

Design Verifier (DV) is the formal verification module of SCADE[2]. DV is a model checker of safety properties. Safety properties are expressed using the SCADE language. A SCADE node implementing a property is called an observer. There is no specific syntax framework as in SMV to express liveness properties in the form of CTL logic [10]. An observer receives as inputs the variables involved in the property and produces an output that should be always true. DV is used to check if a property observer's output is always true. If it is, the property is said to be Valid, otherwise it is said to be Falsifiable, in which case DV generates a counter-example that can be played back in the SCADE simulator for debugging purposes.

---

[1] SCADE is distributed by Esterel Technologies (www.esterel-technologies.com)
[2] DV is based on Prover Technology proof engines (www.prover.com)

```
node CountEvent(Event: bool) returns (Count: int)
var
_L1: int;_L2: int;
let equa eq_CountEvent [ , ]
    _L1 =  if Event then (1 + _L2) else (_L2)
    _L2 = fby(_L1 , 1 , 0)
    Count = _L2
tel
```
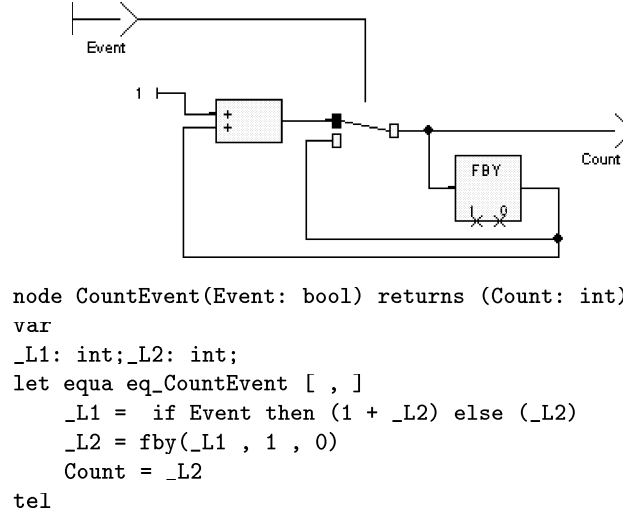
Fig. 3. A simple SCADE node

DV is able to verify properties mixing Boolean control logic, data-value trans-
formations, and temporal behavior. DV core algorithms are based on Stalmarck's
SAT-solving algorithm for dealing with Boolean formulae, surrounded by in-
duction schemes to deal with temporal behavior and state space search [3].
These algorithms are coupled with constraint solving and decision procedures to
deal with the data-path. SAT-based model-checking has shown interesting per-
formances compared to BDD-based model-checking [4], in particular for many
real-world applications with enormous formulas that could not be handled by
current BDD packages [3].

DV, similar to SMV, allows for restriction of the state space while verifying
a property through a notion of assertions. A SCADE assertion on an input or
output variable is similar to an SMV invariant and prunes the search space to
only those instances where the assertion holds true. Therefore, in using SCADE
and the DV, our work let us use and compare DV and SMV as applied to the
same problem.

3   Modeling And Analysis Using SCADE

The overall SCADE model of the system under consideration, as shown in Fig-
ure 4, consists of the voter model and the environment model. The voter's model
corresponds to the "real" system that we wish to analyze. It is composed of a
hierarchal representation of graphical and textual SCADE nodes. The environ-
ment model consists of three sensors models and a world model and is intended
to be a realistic description of the environment driving the voter's functionality.
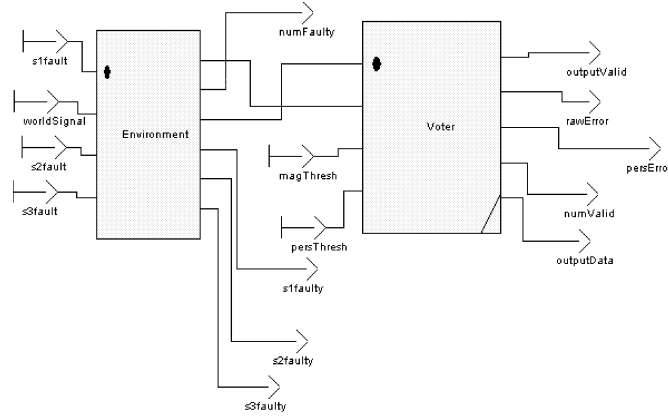
Fig. 4. Top Level SCADE Model of Voter and Environment System

The distinction between the system under study (voter model) and its environment (the sensors and world models) is an important one. The voter's model 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. The level of abstraction used in modeling the environment should be carefully optimized; for we must ensure that the environment will exercise all possible behaviors of the voter (including faulty conditions) without introducing any unrealistic behavior [5].

The design of a realistic environment is crucial when verifying "real" system properties using model checking techniques. We can consider the environment as the outside world that interacts with the system under study and the systems' requirements as collection of statements about the outside world that we want the system to help make true. The design of a good environment must model the outside world requirements accurately without introducing erroneous behavior. The environment model must capture all general, including faulty, expected behavior of the outside world that interfaces with the system to be verified.

Design of good environment models is gaining recognition as an important research problem. For example, in [16], automatically generated environment models are used to verify the source code of the Windows kernel. The environment to be modeled is the application programming interface (API). The environment model is generated via a "training" process by taking several programs that use a common API and applying model checking to create abstractions of the API procedures. These abstractions are then reused on subsequent verification runs to model-check different programs utilizing the same API.

In our study, we use knowledge gained from counter examples generated from using an environment model to modify/refine newer environment models that accurately capture the requirements. We expect our system to handle a range of sensor signals some of which are faulty signals but we must not have more than one sensor exhibiting faulty behavior at any given time; it is true about

the outside world that no two sensors can become faulty simultaneously as the probability of such an event is very close to zero. As discussed in later sections, we capture this property in two different ways through different environment models. The environment models are developed manually and results of three such models are reported in later sections.

Our SCADE model of the voter was developed based on the voter's original Simulink design. SCADE provides a gateway for automatic translation of Simulink models, but we have not evaluated this capability in our current study.

The sections below list the three approaches used to model the voter's environment. Whenever possible, we compare our modeling approach in SCADE versus our earlier work using SMV. Results to follow were computed using a Pentium III PC with 512 MBytes of RAM running Windows 2000.

3.1   Modeling Assumptions and Simplifications

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

Fault Injection
A Boolean flag is read from the outside world by the sensor model; such a flag injects non-deterministic faulty behavior in the sensor. The sensor model is designed such that when this flag is set then the sensor is faulty in both hardware and signal. To test correctness of the voter behavior when signal or hardware faults occur, we use different values for the persistence value constant; by the voter's design, this constant is the number of cycles that a sensor is allowed to differ in signal prior to being eliminated by the voter. We define a "signal fault" as the fault detected by the voter when a sensor differs in signal from the good expected signal value. To test for a signal fault, we set persistent threshold equal to two. This assumption guarantees that signal failure must be detected and isolated by the voter in two cycles; i.e. in one cycle after the cycle where the fault occurs. We also define a sensor's "hardware fault" as the fault detected by the voter due to hardware failure in the sensor; by the voter's design, the hardware fault should be detected and the faulty sensor should be isolated in three cycles. To test for a hardware fault, we set persistent threshold equal to four. Thus the isolation and detection of hardware fault which requires three cycles to complete; i.e. in two cycles after the cycle where the fault occurs, will occur prior to the detection and isolation of the signal fault which in this case requires four cycles to complete.

Time model
The SCADE models do 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 that was adjusted so that the model is independent of time. A detailed justification for modeling the voter's algorithm independent of time is described in [5].

No Simultaneous Sensor Failures
The algorithm assumes that two sensors cannot fail 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 approached this single fault hypothesis in two ways: 1) In the first environment model, referred to in later sections as "Environment Model I", we used a similar approach to our earlier work in SMV. Given that it is the case the number of valid sensors plus the number of sensors declared faulty is bounded between three and four inclusive, we used a SCADE assertion to satisfy this bound.

Note that the number of faulty sensors (numFaulty) is computed within the environment as sensors become faulty but the number of valid sensors (numValid) is computed within the voter and its correctness must be verified independently. With this in mind, we developed different environment model, referred to as "Environment Model II" and "Environment Modell III", that handled the single fault hypothesis by asserting that the number of faulty sensors (a value computed within the environment) is always less than or equal to one.

Noise Free Signals
The models developed did not deal with signal noise. In all of our analysis, we assume that the sensor signals read by the voter are accurate and are noise free.

4 Analysis

We will use the following notation to express our properties throughout this section.

$p \overset{i\ tick}{\to} q$	True iff q is true i cycles after the cycle where p is true.
VS	The number of valid sensors.
F_{h1}, F_{h2}, F_{h3}	True if there is a hardware failure of one, two, and three sensors respectively.
F_{s1}, F_{s2}	True if there is a signal failure of one and two sensors respectively.

4.1 Fault Handling Requirements Properties

The properties extracted from the fault handling requirements of Figure 2 that we want to formally verify can be grouped by the number of sensor failures.

One Sensor Failure Here are the properties when one sensor is detected as faulty.

- **Hardware fault:**
$(VS = 3 \land ValidOutput \land F_{h1}) \overset{2\ tick}{\to} (VS = 2 \land ValidOutput)$

This property means that if the number of valid sensors is 3, and the Voter's output is valid, and there is one hardware faulty sensor, then after 2 cycles the number of valid sensors becomes 2 and the Voter's output is valid.

– **Signal fault:**
$(VS = 3 \wedge ValidOutput \wedge F_{s1}) \overset{1 \ tick}{\rightsquigarrow} (VS = 2 \wedge ValidOutput)$
This property means that if the number of valid sensors is 3, and the Voter's output is valid, and there is one software faulty sensor, then after 1 cycle the number of valid sensors becomes 2 and the Voter's output is valid.

Two Sensor Failures Here are the same properties when a second sensor is detected as faulty.

– **Hardware fault:**
$(VS = 2 \wedge ValidOutput \wedge F_{h2}) \overset{2 \ tick}{\rightsquigarrow} (VS = 1 \wedge ValidOutput)$.
– **Signal fault:**
$(VS = 2 \wedge ValidOutput \wedge F_{s2}) \overset{1 \ tick}{\rightsquigarrow} (VS = 2 \wedge \neg ValidOutput)$.
This property means that if 2 valid sensors miscompare in signal, then after 1 cycle the output of the voter is declared invalid. Both sensors remain valid since the voter algorithm is not designed to determine the faulty sensor for this case. However, the faulty sensor will exhibit a faulty behavior in the following cycle and will get eliminated by the voter. This last special case is discussed in more detail in a later section.

Three sensor failures Here are the same properties when a third sensor is detected as faulty.

– **Hardware fault:**
$(VS = 1 \wedge ValidOutput \wedge F_{h3}) \overset{2 \ tick}{\rightsquigarrow} (VS = 0 \wedge \neg ValidOutput)$.
– **Signal fault:**
If there is only one valid sensor in the system, then the voter can only eliminate this sensor based on a hardware fault, i.e. we cannot test for a signal fault since there is no other valid sensor to compare to.

Sensor elimination Property When a sensor is eliminated by the Voter it will never be considered in any future computation. We express it as: $AG((VS = k) \rightarrow \neg(VS > k)), k \in [0, 2]$. This property means that if the number of valid sensors becomes k then this value will never become a value greater than k.

4.2 Environment Model I

Sensor Model
The sensor model captured as a SCADE graphical node takes as input a non-deterministic Boolean flag from the environment. A signal of one and a hardware valid flag is produced by the sensor if the Boolean flag is false (i.e. sensor not

faulty); a signal of five and a hardware invalid flag is broadcasted when the Boolean input flag is true. In addition, logic combination of a single cycle delay and an or-gate are used so that once a sensor becomes faulty (i.e. Boolean input flag is true) is stays faulty and does not recover. This last quantity is also sent as output from the sensor and is used in calculating the numFaulty value. The assumption that a faulty sensor does not recover differs from our sensor model in SMV where we allowed a faulty sensor to recover its faulty behavior. Furthermore, we use only two input signal values {1,5} to indicate the faulty/non-faulty behavior of a sensor which is different than our analysis in SMV where we used a signal range of {1,2,3}. Assuming two input signal values is justifiable since in our analysis of the voter algorithm we only care about the difference between two signals.

Environment Node
This SCADE node is composed of three sensor nodes, a multiplexer node that groups and outputs the hardware valid flags of the sensors into an array of three Boolean values, a multiplexer that groups and outputs the signals broadcast by the three sensors into an array of three integers and a numFaulty node. The numFaulty node sums the Boolean flags sent by each sensor indicating its faulty/non-faulty status and outputs that sum as numFaulty. We restrict the sum of faulty and valid sensors to be bounded between three and four inclusive. This restriction guarantees our single fault hypothesis because it allows one sensor to become faulty and a second sensor cannot become faulty until the first faulty sensor is detected and eliminated by the voter.

Verification of fault handling requirements
Table 1 summarizes the results we obtain when checking the properties. The first column recalls the name of the property. The second column gives the CPU time in seconds spent by DV to check the property. The third column gives the result of verification, that either Valid to mean that the property holds, or Falsifiable in the opposite case, and some additional comments when necessary.

Recall that in our sensor model, hardware and signal faults occur simultaneously, therefore, we expect that the signal fault is detected by the voter in a number of cycles equal to the persistence threshold value (i.e. two) where the hardware fault should be detected in three cycles. Therefore, by setting persistence threshold equal to two, not only can we check that a second faulty sensor with a signal fault leads to a state where the voter's output is declared invalid and the number of valid sensors is still equal to two; for we can also check weather this same sensor demonstrates a hardware failure in three cycles; i.e. in one cycle after it reached the state of 2 valid sensors and invalid input, the voter should be in the state of one valid sensor and valid output. We were able to verify that $(VS = 2 \land \neg ValidOutput) \overset{1 \; tick}{\to} (VS = 1 \land ValidOutput)$.

Table 1. Verification Results

Property	Time(secs)	Result
One sensor failure		
Hardware fault	134	Valid: a faulty sensor on a hardware fault is detected by the voter
Signal fault	3	Valid
Sensor elimination is final	0.13	Valid with persistent threshold = 2 (software fault)
Sensor elimination is final	0.14	Valid with persistent threshold = 4 (hardware fault)
Two sensor failures		
Hardware fault	137	Valid: a second faulty sensor on a hardware fault is detected by the voter
Signal fault	81	Valid
Sensor elimination is final	0.11	Valid with persistent threshold = 2 (software fault)
Sensor elimination is final	0.13	Valid with persistent threshold = 4 (hardware fault)
Three sensor failures		
Hardware fault	137	Valid: a second faulty sensor on a hardware fault is detected by the voter
Signal fault		Not relevant, see section 4.1
Sensor elimination is final	0.14	Valid

Drawbacks to Environment Model I

1. One drawback to this sensor model is that a failed sensor remains faulty and continues to produce an invalid hardware flag and a bad signal. This means that we cannot test the voter's ability to transition from the state where the number of valid sensors is two and output is not valid to the state where the number of valid sensors is two and the output is valid as shown in Figure 2. A different sensor model is required to investigate this capability of the voter

2. The assertion that number of valid sensors plus the number of faulty sensors is between three and four inclusive, coupled with the sensor design used allows the second faulty sensor on a software fault to manifest its fault as a hardware fault, thus transitioning to the state of 1 valid sensor and valid output. This design eliminates the possibility that the third healthy sensor manifests a faulty hardware behavior before the second fault is detected by the voter.

4.3 Environment model II

Environment Node
This SCADE node differs from the environment in the previous model in that the numFaulty node sums the Boolean flag read by each sensor from the outside world indicating its faulty/non-faulty status and outputs that sum as numFaulty

as opposed to summing up the Boolean flag sent by each sensor indicating its faulty/non-faulty status. Such change is necessary in order to use the assertion that the number of faulty sensors is less than or equal to one (i.e. numFaulty $<= 1$). This assertion does not make use of the variable numValid which is computed by the voter. The assertion guarantees the single fault hypothesis since it allows only one sensor to be faulty at any given time. The results obtained from this environment model are described below.

Verification of fault handling requirements

Table 2 summarizes the results we obtain when checking the properties. For one sensor failure the verification attempt resulted in a counter-example where sensor1 is faulty for one cycle (i.e. signal = 5), sensor3 is faulty for the next cycle (i.e. signal = 5) so sensor1 being faulty agrees with sensor3 being faulty in the next cycle instead of being eliminated. Using the assertion that the number of valid and faulty sensors is between three and four inclusive prevented a situation where the second sensor becomes faulty before the first faulty sensor is eliminated by the voter. However, the assertion made use of numValid which is an output of the voter itself. When we avoid using numValid, and instead use the numFaulty $<= 1$ assertion, we permit more random behavior of the system and receive the counter example above. The problem is that we have not allowed enough time between faults for the voter to eliminate the sensor.

Table 2. Verification Results

Property	Time(secs)	Result
One sensor failure		
Hardware fault	88	Valid: a faulty sensor on a hardware fault is detected by the voter
Signal fault		Falsifiable
Two sensor failures		
Hardware fault	116	Valid: a second faulty sensor on a hardware fault is detected by the voter
Signal fault		Falsifiable

For two sensor failures the verification resulted in a counter-example where sensor1 is faulty for two cycles and is eliminated by the voter, after which sensor3 is faulty for one cycle (i.e. signal equals 5), then sensor2 is faulty for the next immediate cycle (signal = 5 also). Therefore, sensor3 and sensor2, though faulty, agree on signal and we do not get to the state where the number of valid sensors is 2 but the output is not valid. The fact that we are not allowing enough time between faults causes this behavior to occur. This problem is addressed in Environment model III described below.

4.4 Environment Model III

Modifications to Sensor Model
Similar to previous section, this environment model uses the assertion that num-Faulty $<= 1$ and also assumes that a time delay that exceeds the thresholds for detecting a hardware or signal fault exists between sensor faults. The sensor model, Figure 5, is modified such that the fault Boolean flag for faulty sensor remains true for five cycles after which it is set to false thus allowing a second sensor to fail under the assumption that numFaulty $<= 1$. Recall that a hardware fault must be detected in three cycles and we are using a persistent threshold of two and four for the detection of signal failure, thus the five cycle delay is justified. The sensor model is further modified to receive a signal range between one and five from the outside world; a non-faulty sensor broadcasts the signal received whereas a faulty sensor broadcasts the signal received plus one. Hence a faulty sensor always exhibits faulty behavior in signal. This assumption is valid because we are interested in the voter behavior when the sensors differ in signal.
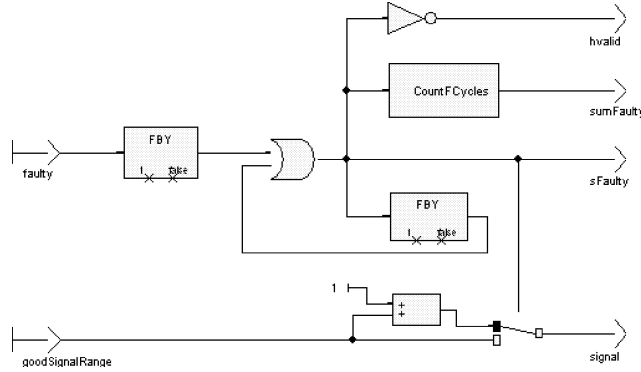

Fig. 5. SCADE Model of Sensor used in Environment III

In all the analysis below we use an assertion that numFaulty $<= 1$ and that the sensor signal range is between one and five inclusive.

Verification of fault handling requirements
Table 3 summarizes the results we obtain when checking the fault-handling properties. Table 1 and Table 3 are verification results for the same properties with the exception that the single fault hypothesis used in table 3 is independent of any values computed by the voter.

Property for the Output Signal Value It is expected that there will be a transient condition in which the voter output data may differ from the signal

Table 3. Verification Results

Property	Time(secs)	Result
One sensor failure		
Hardware fault	204	Valid: a faulty sensor on a hardware fault is detected by the voter
Signal fault	101.6	Valid
Sensor elimination is final	0.31	Valid with persistent threshold = 2 (software fault)
Sensor elimination is final	0.23	Valid with persistent threshold = 4 (hardware fault)
Two sensor failures		
Hardware fault	225	Valid: a second faulty sensor on a hardware fault is detected by the voter
Signal fault	122.67	Valid
Sensor elimination is final	0.19	Valid with persistent threshold = 2 (software fault)
Sensor elimination is final	0.22	Valid with persistent threshold = 4 (hardware fault)
Three sensor failures		
Hardware fault	0.22	Valid: a second faulty sensor on a hardware fault is detected by the voter
Signal fault		Not relevant (see section 4.1)
Sensor elimination is final	0.23	Valid

received from the world when a sensor becomes faulty. However, after certain threshold, the voter output data must agree with the signal received from the world. Using a delay of 2 cycles, a persistent threshold of 4, and asserting that numValid is greater than 0, we verified that:

$$(worldSignal \neq voterdata) \overset{2\ tick}{\rightarrow} (worldSignal = voterdata)$$

The verification completed in 10.4 seconds and produced a valid result.

5 Conclusion

In this paper we have used the formal verification capabilities of the SCADE Design Verifier to analyze an embedded avionics software design. Model checking is used to verify the correctness of the design with respect to its high-level fault-handling requirements.

The main contribution of this work is to demonstrate the significance of the environment model in verifying the design requirements of an algorithm. The process of capturing a system's design requirements for algorithm development and subsequent software implementation is never an easy task. The design of a suitable environment model is centered around developing a model that drives the system to be implemented and tests the captured design requirements of the

16

system. Our work shows that capturing and developing the correct environment model is a key issue and can be as hard as verifying correctness of the system itself.

Acknowledgment

We extend our gratitude to Gary Hartmann and Steve Pratt at Honeywell Laboratories for designing the voter algorithm and providing insight on its behavioral properties and functional requirements.

References

1. P. Caspi, A. Curic, A Maignan, C. Sofonois, S. Tripakis,P Niebert, and T. Le Sergent, "From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications", (Submitted to CASES'02), Esterel Technologies.
2. N.Halbwachs, P Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language Lustre", Proceeding of the IEEE, September, 1991.
3. Mary Sheeran and Gunnar Stalmarck, "A tutorial on Stalmarck's proof procedure for propositional logic", Prover Technology AB and Chalmers University of Technology, 1998, Sweden.
4. Per Bjesse and Koen Claessen, "SAT-based Verification without State Space Traversal", Formal Methods in Computer-Aided Design, 2000.
5. S. Dajani-Brown, D. Cofer, G Hartmann, and S. Pratt, "Formal Modeling and Analysis of an Avionics Triplex Sensor Voter", Model Checking Software, 10th International SPIN Workshop, , Springer-Verlag, May 2003.
6. G. Berry, A. Bouali, X. Fornari, E. Ledinot, E. Nassor, R. de Simone, "ESTEREL: a formal method applied to avionic software development", Science of Computer Programming V 36 No 1, January 2000.
7. S. Osder, "Practical View of Redundancy Management Application and Theory", Journal of Guidance and Control, Vol. 22 No. 1 , Jan-Feb 1999.
8. R.P.G. Collinson, Introduction to Avionics, Chapman & Hall, London, 1998.
9. K. McMillan, Symbolic Model Checking , Kluwer Academic Publishers, Boston, Dordrecht, London, 1993.
10. Micheal R A Huth and Mark D Ryan, Logic in Computer Science Modelling and reasoning about systems, University Press, Cambridge, United Kingdom, 2000.
11. SMV web page: http://www-2.cs.cmu.edu/ modelcheck
12. Simulink weg page: http://www.mathworks.com/products/simulink
13. SCADE 4.1.1, Training Manual, Esterel Technologies, Montreal, Canada.
14. SCADE 4.2, Design Verifier User Manual, Esterel Technologies, Montreal, Canada.
15. Gerard Berry, "The effectiveness of Synchronous Languages for the Development of Safety-Critical Systems", white paper, Esterel Technologies 2003
16. T. Ball, V. Levin, and F. Xei, "Automatic Creation of Environment Models via Training", TACAS 2004, Barcelona, Spain