Designing and Model Checking a Quad-Redundant Sensor Voter System

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

Master of Technology
in
Computing and Mathematics

by

Vishnuvardhan Prem Paramban
(112002009)

COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the document entitled "**Designing and Model Checking a Quad-Redundant Sensor Voter System**" is a bonafide work of **Vishnuvardhan Prem Paramban (Roll No. 112002009)**, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr.Jasine Babu

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

First and foremost, I am extremely grateful to my mentor, Dr. Jasine Babu, Assistant Professor, Department of Computer Science and Engineering, IIT Palakkad for her invaluable advice,guidance and continuous support. Her vast knowledge and painstaking supervision has been key to my progress.

I would also like to express my sincere gratitude to Mrs. Deepa Sara John, Deputy Division Head at the ISRO Inertial Systems Unit for her constant support and guidance.I am particularly grateful for the tremendous insight she provided on avionic systems. I also thank the ISRO Inertial Systems Unit for the opportunity to collaborate with them on this project.

I would like to thank my Project Coordinator, Dr. Albert Sunny, Departm1ent of Computer Science and Engineering, IIT Palakkad for his kind support. I would also like to express my gratitude to the rest of the faculty of the Mathematics and Computing department at IIT Palakkad for instilling in me the fundamental knowledge I required to complete this project.

Contents

List of Figures

List of Tables

Abstract

Our lives today are extremely reliant on the digital world and as such, it has become crucial to develop techniques to ensure the proper functioning of these digital systems. System verification is used to establish that a product or design under consideration possess certain properties. Of the various methods of system verification, our interests lie primarily in the area of model checking. Through this project, we were hoping to advocate for model checking as a crucial step in the development of both hardware and software systems. With the extremely high costs involved in the testing and simulation of avionic systems, we found it extremely suitable to demonstrate the applications of model checking. For this, we designed a quad redundant voter system and then applied model checking to verify the correctness of the design.

Chapter 1

Introduction

1.1 Model Checking

Given a system model and a formal property, model checking is an automated technique that tests whether the property holds for that model. A model checker takes in a model and a property that the end system must meet. The checker then outputs yes if the given model meets the specifications and no otherwise. If the model doesn't meet the specifications, it is desired that a good model checker returns a counterexample.The counterexample allows us to better understand the model and make the changes as required. Errors that we might have missed during testing and simulation phases may also be revealed using model checking. The general outline of a model checking system is shown in Figure 1.1. With the increasing complexity of information and communication technology systems, model checking has proved to be extremely useful while designing systems. Several catastrophic errors could have been avoided, if model checking had simply been done beforehand. A notable example is that of the launch of the Ariane-5 missile which had a fatal defect in it's control software leading to its crash 36 seconds after launching.[1]

One of the standard methods of representing models for hardware and software systems

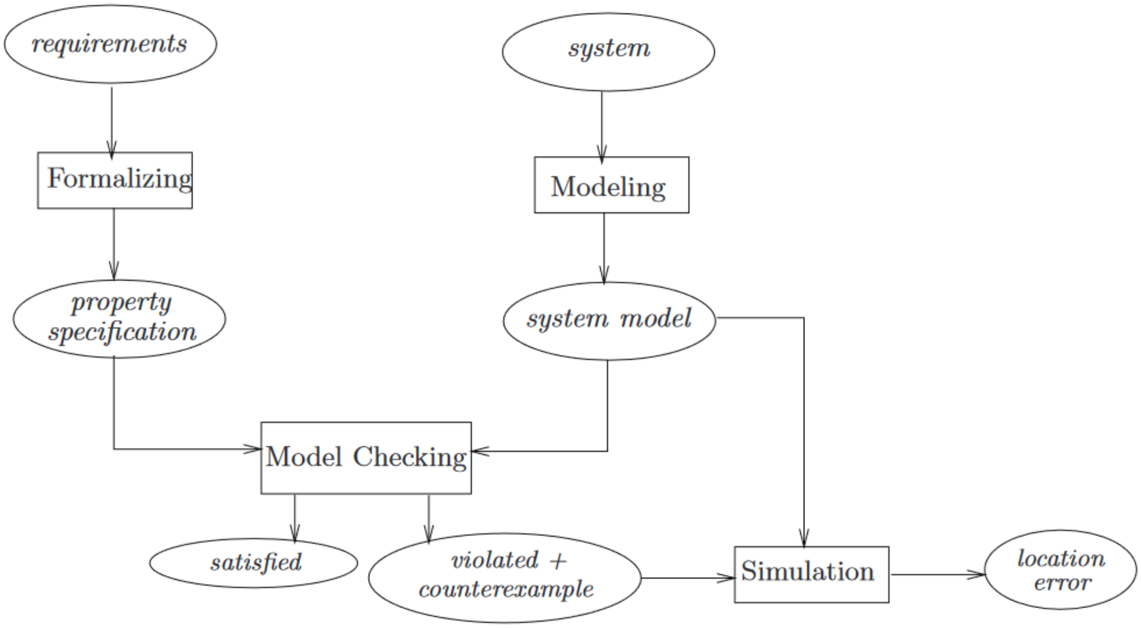

Fig. 1.1 Schematic view of the model-checking approach. [1]

are ***transition systems***. We generally describe the specifications of the systems using some logical formulas written in an appropriate logic such as ***temporal logic***.

1.1.1 Transition Systems

A transition system is a tuple $(S, Act, \rightarrow, I, AP, L)$ where, [1]

- S is a set of states,
- Act is a set of actions ,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation.It is a standard practice to write $s \xrightarrow{\alpha} s^{'}$ instead of $(s, \alpha, s^{'})$.
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labelling function.

We can better understand this with the following transition system of an extremely simplistic design of a vending machine.

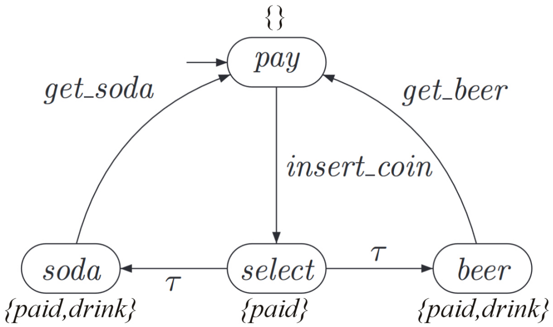

Fig. 1.2 Transition System for a simple beverage vending machine [1]

In this example,

- There are a total of 4 states, $S = \{pay, select, soda, beer\}$
- There are 3 types of actions, $Act = \{get\_soda, get\_beer, \tau\}$
- $pay \xrightarrow{insert\_coin} select$ is an example of a transition.
- There is only one initial state $I = \{pay\}$
- The atomic propositions allows us to better understand each state in the transition system. For example in this case, we only need 2 atomic propositions. $AP = \{paid, drink\}$.
- The labelling function gives us the set of atomic propositions that are satisfied by each states. In this case,$L(pay) = \phi, L(soda) = L(beer) = \{paid, drink\}, L(select) = \{paid\}$.

1.1.2 Executions and Traces

Executions describe the possible behaviour of the transition system. A transition system TS's finite execution fragment π is an alternating series of states and actions that ends with a state.

$$\pi = s_0\alpha_1 s1\alpha_2.... \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i.$$

An infinite execution fragment is an infinite alternating sequence of states and actions. An initial execution fragment begins with an initial state, while a maximal execution fragment is either a finite execution fragment that ends in a terminal state or an infinite execution fragment. An execution of a transition is essentially an initial and maximal execution fragment. For our preceding example of the vending machine, consider the following example of an execution,

$$\rho = pay \xrightarrow{insert\_coin} select \xrightarrow{\tau} soda \xrightarrow{get\_soda} pay \xrightarrow{insert\_coin} select \xrightarrow{\tau} beer.....$$

The ***Trace*** of an execution fragment is an induced finite or infinite word over the alphabet 2^{AP}. The trace of an execution fragment $\pi = s_0\alpha_1 s1\alpha_2....$ is given by,

$$trace(\pi) = L(s_0)L(s_1)....$$

For the above example of an execution, the trace of the execution would be,

$$trace(\rho) = \{\}\{paid\}\{paid, drink\}\{\}\{paid\}\{paid, drink\}...$$

A trace of a state s is the trace of an execution fragment with the first element on the path as s. The traces of a state s is given by,

$$Traces(s) = \{trace(X)|X \text{ is an execution fragment starting from } s\}$$

Traces describe the behaviour of the transition system with respect to atomic propositions. The traces of a transition system is the set of traces of all conceivable executions of the transition system.

$$Traces(TS) = \{Trace(X) | X \text{ is an execution of the transition system.}\}$$

By analysing all traces of a transition system, we can check if the system satisfies different properties.

1.1.3 Properties

Let AP be a set of atomic propositions. Let AP_{INF} be the set of all infinite words over the alphabet, 2^{AP}.A property P is any subset of AP_{INF}.

Let P be a property over AP and $TS = (S, Act, \to, I, AP, L)$ be a transition system. We say that TS satisfies P, denoted by $TS \models P$, if and only if $Traces(TS) \subseteq P$. A state $s \in S$ satisfies P , denoted by $s \models P$, whenever $Traces(s) \subseteq P$.

Let T be a transition system over the set of atomic propositions $AP = \{crit_1, crit_2\}$, used to represent a mutual exclusion system where $crit_1$ represents that the first process is in it's critical section and $crit_2$ represents that the second process is in it's critical section.Now let ,

$$P = \{w = A_0A_1A_2...; \forall i \{crit_1, crit_2\} \not\subseteq A_i\}$$

This property represents the mutual exclusion property and for the system to satisfy mutual exclusion, T must satisfy P.

1.1.4 Linear Temporal Logic

Temporal logic is an extension of propositional logic that allows to refer to a system's infinite behaviour. Linear Temporal Logic(LTL) describes properties of individual executions and it's semantics are defined as a set of executions.

LTL formulae are composed of atomic propositions, logical operators $\vee, \wedge$ and $\neg$ and temporal operators which describe properties of a path such as,

Xp	(Next p)	p holds at the next point in time
Gp	(Globally p)	p holds at every point in time
Fp	(Future p)	p holds at some future point in time
pUq	(p Until q)	p holds until q holds

Table 1.1 Basic temporal operators[4]

LTL is comprised of ***state formulas*** and ***path formulas***. The syntax of state formulas is given by[4],

$$f ::= \top \mid \bot \mid p \mid \neg f \mid f_1 \vee f_2 \mid f_1 \wedge f_2$$

State formulas are evaluated for the state in question, whereas path formulas are evaluated for a certain path. A path formula is composed of the following elements[4],

$$g ::== f \mid \neg g \mid g_1 \vee g_2 \mid g_1 \wedge g_2 \mid \mathbf{X}g \mid \mathbf{F}g \mid \mathbf{G}g \mid g_1\mathbf{U}g_2 \mid g_1\mathbf{R}g_2$$

Suppose f is an LTL formula over a set of atomic propositions AP, we can define a property P_f as

$$P_f = \{w \in AP_{INF}; w \text{ satisfies } f\}$$

Let us now define the semantics of LTL[5]. First, let us define the binary satisfaction, denoted by $\models$, for LTL formulae. This satisfaction is with respect to a pair, $< T, \pi >$, a transition system and an execution.

- $T, \pi \models \top$, true is always satisfied.
- $T, \pi \not\models \bot$, false is never satisfied.
- $(T, \pi \models p)$ if and only if $(p \in L(\pi_0))$, atomic propositions are satisfied when they are members of the execution's first element's labels.
- $(T, \pi \models \neg\phi)$ if and only if $(T, \pi \not\models \phi)$.
- $(T, \pi \models \phi_1 \wedge \phi_2)$ if and only if $[(T, \pi \models \phi_1) \wedge (T, \pi \models \phi_2)]$.

- $(T, \pi \models \phi_1 \vee \phi_2)$ if and only if $[(T, \pi \models \phi_1) \vee (T, \pi \models \phi_2)]$.
- $(T, \pi \models X\phi)$ if and only if $(T, \pi^1 \models \phi)$.
- $(T, \pi \models F\phi)$ if and only if ($\exists i$ such that $T, \pi^i \models \phi$).
- $(T, \pi \models G\phi)$ if and only if ($\forall i$ such that $T, \pi^i \models \phi$).
- $(T, \pi \models \phi_1 U \phi_2)$ if and only if $[\exists i$ such that $(\forall j < i(T, \pi^j \models \phi_1)) \wedge (T, \pi^i \models \phi_2)]$.
- $(T \models_T \phi)$ if and only if $\forall \pi$ such that $\pi_0 \in I$, $(T, \pi \models \phi)$, A model or transition system, satisfies an LTL formula when all it's executions do.

1.1.5 Computation Tree Logic

The attributes of a computation tree are described by Computation Tree Logic. CTL's semantics are built on a branching notion of time and are defined in terms of an infinite, directed tree of states rather than an endless sequence. The tree itself represents all conceivable paths because each traversal represents a single path. The tree rooted at a state s would represent all possible computations in the system which start at that state.

In CTL, as well as the temporal operators U, F, G and X of LTL, we also have the following path quantifiers[6],

$$A : \text{For all paths}$$

$$E : \text{There exists a path}$$

Let p represent an atomic formula, then the language of CTL is generated by the following grammar,

$$f ::= \top \mid \bot \mid p \mid \neg f \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid AX\ f \mid EX\ f$$
$$\mid AF\ f \mid EF\ f \mid AG\ f \mid EG\ f \mid A[f_1 U f_2] \mid E[f_1 U f_2]$$

Some examples of CTL formulas are below,

$AG(EF\ p)$	It is always possible to reach a state where p holds
$EF(p_1 \wedge p_2)$	It is possible to get to a state where p_1 and p_2 holds
$AF(AG\ p)$	p eventually holds across all paths
$AF(AF\ p)$	p is infinitely true on all paths

Table 1.2 Basic temporal operators[4]

Suppose F is a CTL formula over a set of atomic propositions AP, we can define a property P_F as

$$P_F = \{w \in AP_{INF}; w \text{ satisfies } F\}$$

Let us now define the semantics of CTL [5]. First, let us define the binary satisfaction, denoted by $\models$, for CTL formulae. This satisfaction is with respect to a pair, $< T, s >$, a transition system and a state.

- $T, s \models \top$, true is always satisfied.
- $T, s \not\models \bot$, false is never satisfied.
- $(T, s \models p)$ if and only if $(p \in L(s))$, atomic propositions are satisfied when they are members of the state's labels.
- $(T, s \models \neg\phi)$ if and only if $(T, s \not\models \phi)$.
- $(T, s \models \phi_1 \wedge \phi_2)$ if and only if $[(T, s \models \phi_1) \wedge (T, s \models \phi_2)]$.
- $(T, s \models \phi_1 \vee \phi_2)$ if and only if $[(T, s \models \phi_1) \vee (T, s \models \phi_2)]$.
- $(T, s \models AX\phi)$ if and only if $(\forall \pi$ such that $\pi_o = s, T, \pi^1 \models \phi)$.
- $(T, s \models AF\phi)$ if and only if $(\forall \pi$ such that $\pi_0 = s, \exists i$ such that $T, \pi^i \models \phi)$.
- $(T, s \models AG\phi)$ if and only if $(\forall \pi$ such that $\pi_0 = s, \forall i\ T, \pi^i \models \phi)$ for all paths starting at s,always ϕ.

- $(T, s \models \phi_1 AU \phi_2)$ if and only if $(\forall \pi$ such that $\pi_0 = s, \exists i$ such that $(\forall j < i(T, \pi^j \models \phi_1)) \wedge (T, \pi^i \models \phi_2))$ for all paths starting at s, ϕ_1 until ϕ_2.

- $(T, s \models EX\phi)$ if and only if $(\exists \pi$ such that $\pi_o = s, T, \pi^1 \models \phi)$.

- $(T, s \models EF\phi)$ if and only if $(\exists \pi$ such that $\pi_0 = s, \exists i$ such that $T, \pi^i \models \phi)$.

- $(T, s \models EG\phi)$ if and only if $(\exists \pi$ such that $\pi_0 = s, \forall i\ T, \pi^i \models \phi)$ for all paths starting at s,always ϕ.

- $(T, s \models \phi_1 EU \phi_2)$ if and only if $(\exists \pi$ such that $\pi_0 = s, \exists i$ such that $(\forall j < i(T, \pi^j \models \phi_1)) \wedge (T, \pi^i \models \phi_2))$ for all paths starting at s, ϕ_1 until ϕ_2.

- $(T \models_T \phi)$ if and only if $\forall s \in I$, $(T, s \models \phi)$, A model or transition system, satisfies a CTL formula when all it's states do.

1.2 NuSMV

NuSMV is an open source automatic model checker that can check finite state systems against temporal logic specifications such as Linear Temporal Logic(LTL) and Computation Tree Logic(CTL). ITC-IRST and Carnegie Mellon University collaborated for the development of NuSMV.

1.2.1 NuSMV Architecture

NuSMV was created to be an open system that could be easily changed, extended or modified. The entire system architecture is structured and organized as modules. The architecture of NuSMV can be seen below

It includes the following modules:[2]

Kernel: All the low level functionalities such as data structure manipulation and dynamic memory allocation is handled by the NuSMV kernel. It also provides all the basic binary

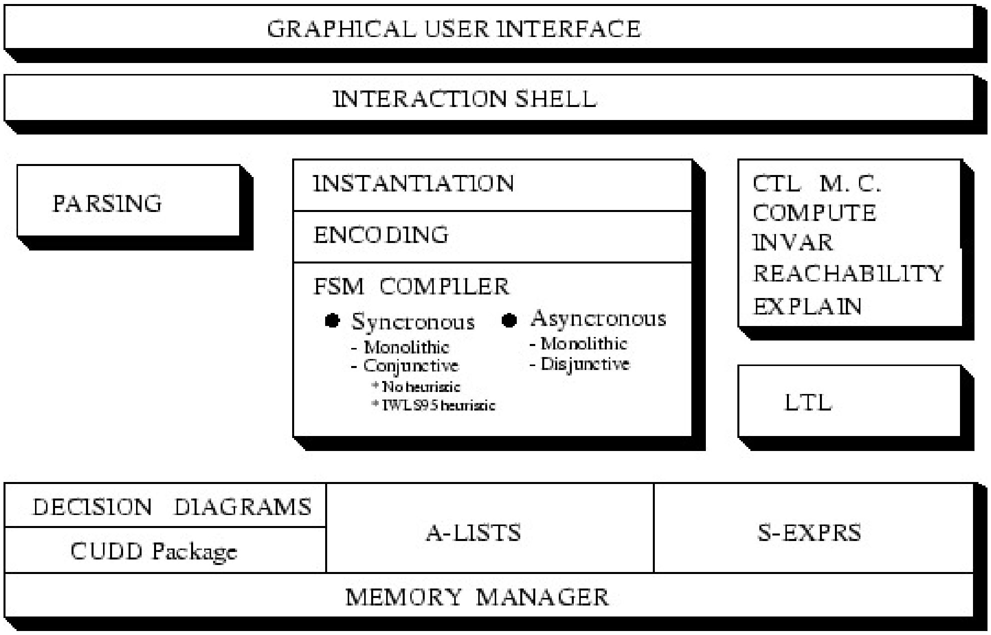

Fig. 1.3 Architecture of NuSMV [2]

decision diagram primitives.

Parser: As typical parsers, this modules is responsible for implementing the routines to a process a file, verify it's syntax and then build a parse tree to represent the structure of the input file.

Compiler: The compiler is responsible for compilation of the parsed model into Binary Decision Diagrams. The Instantiation submodule processes the parse tree from the parser to describe a finite state machine representing the model. The Encoding submodule performs the encoding of data types and finite ranges into boolean domains. The FSM Compiler submodule provides the routines for constructing and manipulating finite state machines.

Model Checking: The model checking module as the name suggests provides all the functionalities to perform reachability checking, CTL model checking and invariant checking. It is also responsible for counterexample generation and inspection.

LTL: LTL model checking is performed in NuSMV by first converting the LTL formula into a tableau. Then an algorithm is applied to perform model checking. The LTL module is responsible for calling an external program which will translate the LTL formula into the tableau format for NuSMV.

Interactive Shell and GUI: The user is allowed access to the full functionality of NuSMV via the interactive shell. The Graphical User Interface is built over the interactive shell and in addition to all the functionalities, allows the user to inspect and set the values of the environment variables.

We write programs for NuSMV in the SMV language. It allows for the description of finite state systems that range from completely synchronous to completely asynchronous. Further along, we shall look at some examples to better understand how models are represented in NuSMV and how we can specify properties to be checked for the designed system.

1.2.2 Structure of a NuSMV program

A NuSMV program is comprised of modules, which can be reused. There must be one module with name *main* and no formal parameters. The main module is the one evaluated by the interpreter. A module has primarily 3 parts,

- **VAR**: Identifies a portion of code where variables are defined.

- **ASSIGN**: Identifies a portion of a code where variables are initialised and evolution is described.

- **SPEC**: Here we define properties that are to be verified. We can also check for properties from the interactive shell while running the program as well.

The following code of a simple model shows the general structure of a NuSMV program.

```
MODULE main
VAR
    location:l1,l2;
    x:0..100;
ASSIGN
    init(location):=l1;
    init(x):=0;
    next(location):=case
                    (location=l1) &(x<10):l2;
                    location=l2:l1;
                    TRUE:location;
                    esac;
    next(x):= case
                (location=l1):x;
                (location=l2)&(x<100):x+1;
                TRUE:x;
                esac;
LTLSPEC G(request=false)
```

We will look at some systems and how they are modelled and checked for properties using NuSMV in the next chapter.

Chapter 2

Some sample models

In this chapter, we shall be looking at some systems we modelled as part of our learning. We will also look into how they are then modelled in NuSMV and also explain how we can check for different properties using NuSMV.

2.1 Modelling a hardware circuit

Consider the following sequential circuit,

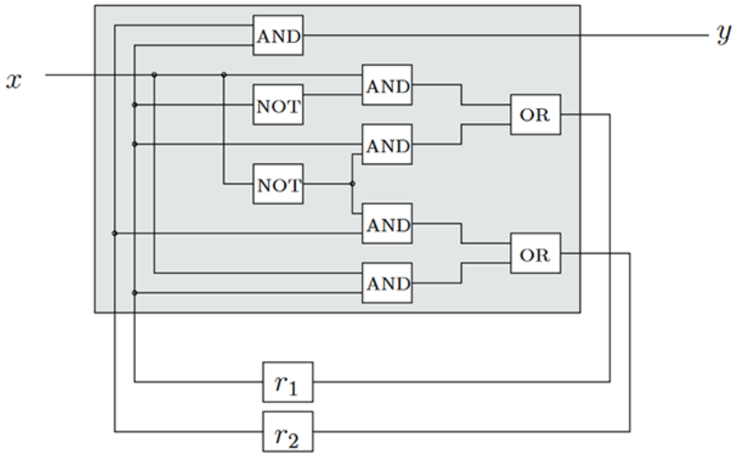

Fig. 2.1 A sequential hardware circuit[1]

To model the hardware circuit, we must look at the possible values of x, r_1, r_2 and y. as in our earlier circuit. Let us now create the truth table for this,

x	r_1	r_2	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 2.1 Truth table of the harware circuit 2.1

There are 8 possible states. The transition system we drew for this circuit is given below.

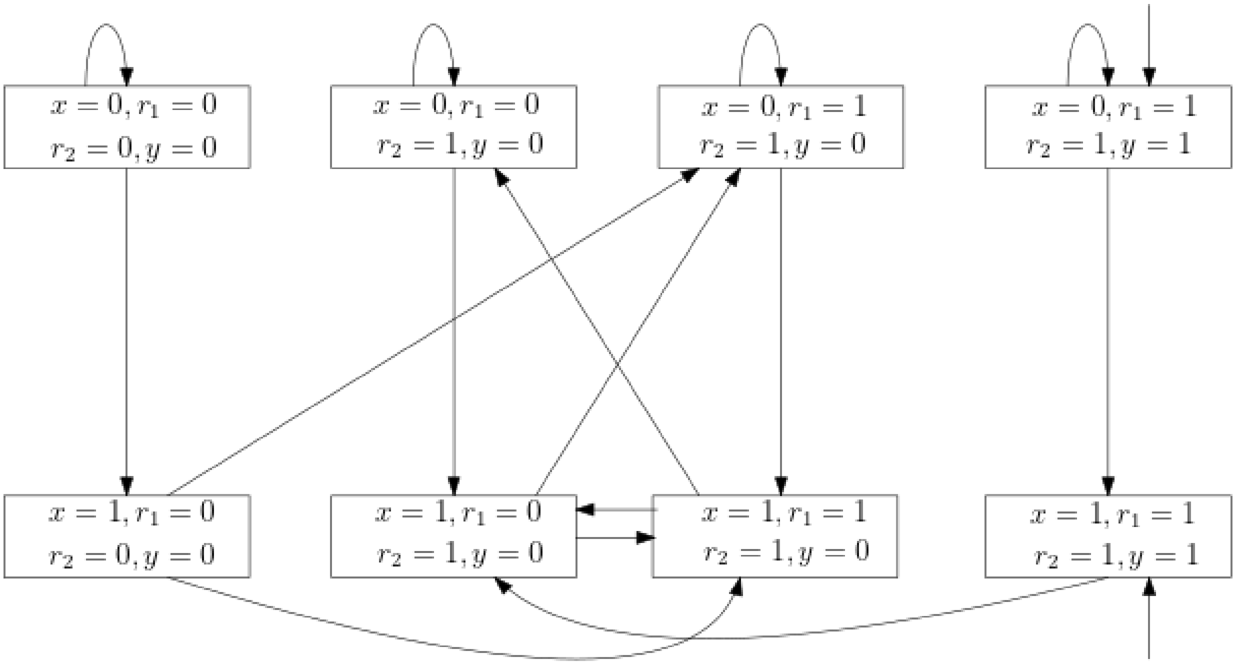
$x = 0, r_1 = 0$ $r_2 = 0, y = 0$
$x = 0, r_1 = 0$ $r_2 = 1, y = 0$
$x = 0, r_1 = 1$ $r_2 = 1, y = 0$
$x = 0, r_1 = 1$ $r_2 = 1, y = 1$
$x = 1, r_1 = 0$ $r_2 = 0, y = 0$
$x = 1, r_1 = 0$ $r_2 = 1, y = 0$
$x = 1, r_1 = 1$ $r_2 = 1, y = 0$
$x = 1, r_1 = 1$ $r_2 = 1, y = 1$

Fig. 2.2 Transition system for the hardware circuit 2.1

We modelled the circuit in NuSMV as shown below.

```
MODULE main
VAR
    x:boolean;
    r1:nand2(x,r1.out);
    r2:nand3(x,r2.out,r1.out);
DEFINE
    fout:=r1.out & r2.out;
MODULE nand2(in1,in2)
VAR
    out:boolean;
ASSIGN
    init(out):=TRUE;
    next(out):=(in1 & !in2) | (!in1 & in2);
MODULE nand3(in1,in2,in3)
VAR
    out:boolean;
ASSIGN
    init(out):=TRUE;
    next(out):=(!(in1) & in2) | (in1 & in3);
```

Checking for reachable states using the *print_reachable_states* command in NuSMV shows that there are 6 reachable states in the circuit and that the system diameter is 3.

2.2 Modelling Synchronous and Asynchronous systems

Consider the following system of two traffic lights.

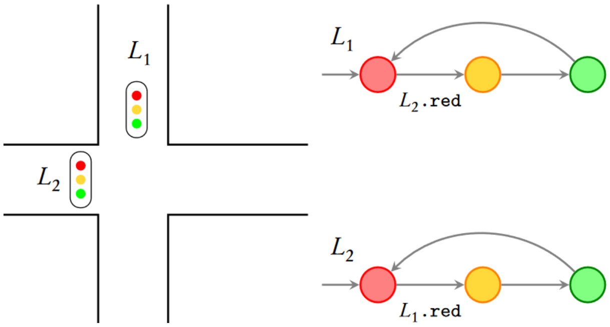

Fig. 2.3 L_1 represents the transition system of the first light and L_2 represents the transition system of the second light.Either traffic light can move from the red state to the yellow state only when the other has become red. [7]

We first built a synchronous model of the system despite it being impractical. The system was modelled in NuSMV as follows,

```
MODULE light(other)
VAR
    state:{red,yellow,green};
ASSIGN
    init(state):=red;
    next(state):=case
                    state=red & other=red:{red,yellow};
                    state=yellow:green;
                    state=green:{green,red};
                    TRUE:state; esac;
MODULE main
VAR
    trafficlight1:light(trafficlight.state);
    trafficlight2:light(trafficlight1.state);
```

Printing the reachable states in NuSMV returned a state where both lights are simultaneously green which is an undesirable outcome.

We then built an asynchronous version of the same system in the hopes of avoiding this outcome. In NuSMV, we use the ***PROCESS*** keyword to make a system asynchronous.The code snippet for the asynchronous version of the same system is given below,

```
MODULE light(other)
VAR
    state:{red,yellow,green};
ASSIGN
    init(state):=red;
    next(state):=case
                    state=red & other=red:{red,yellow};
                    state=yellow:green;
                    state=green:{green,red};
                    TRUE:state;
                esac;
MODULE main
VAR
    trafficlight1:process light(trafficlight2.state);
    trafficlight2:process light(trafficlight1.state);
```

Checking for reachable states in NuSMV showed us that there are no reachable states that had both lights at green simultaneously. Clearly, this is a better implementation of the system.

2.3 Modelling Mutual Exclusion algorithms

Pnueli's Mutual Exclusion Algorithm

Consider the following mutual exclusion algorithm by Pnueli for two processes p_0 and p_1, [8]

```
I0:loop forever do
   begin
   I1: Noncritical section
   I2: (y[i],s):=(1,i);
   I3: wait until ((y[1-i=0) or (s!=i));
   I4: Critical section
   I5: y[i]:=0
end.
```

Here, s is a shared variable which can only take the values, 0 and 1. In addition to the shared variable, each process has a local boolean variable y that is initially 0. The statement $(y_i, s) = (1, i)$ is a multiple assignment in which $y_i = 1$ and $s = i$ is a single atomic step.

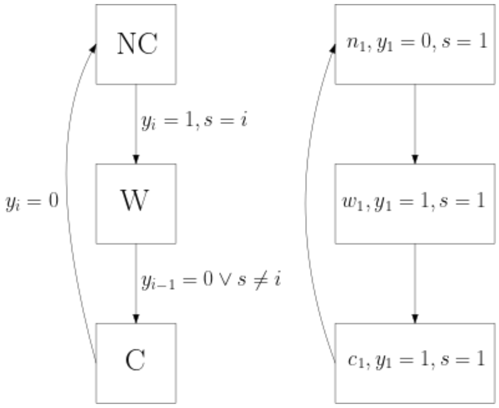

Fig. 2.4 Program graph for a process i and transition system for process p_1

Before drawing the transition system for the process,we first draw a program graph for it to better understand how the algorithm works as in figure. We then build the transition system from the program graph. This can be seen in figure 2.3.

We also drew up a parallel composition of the two processes to figure out if it was possible for the processes to violate mutual exclusion. On tracing, it becomes clear that mutual exclusion is guaranteed as it is not possible to move from the states WC or CW to the state CC as the condition on s won't allow it. The composition we came up with is below,

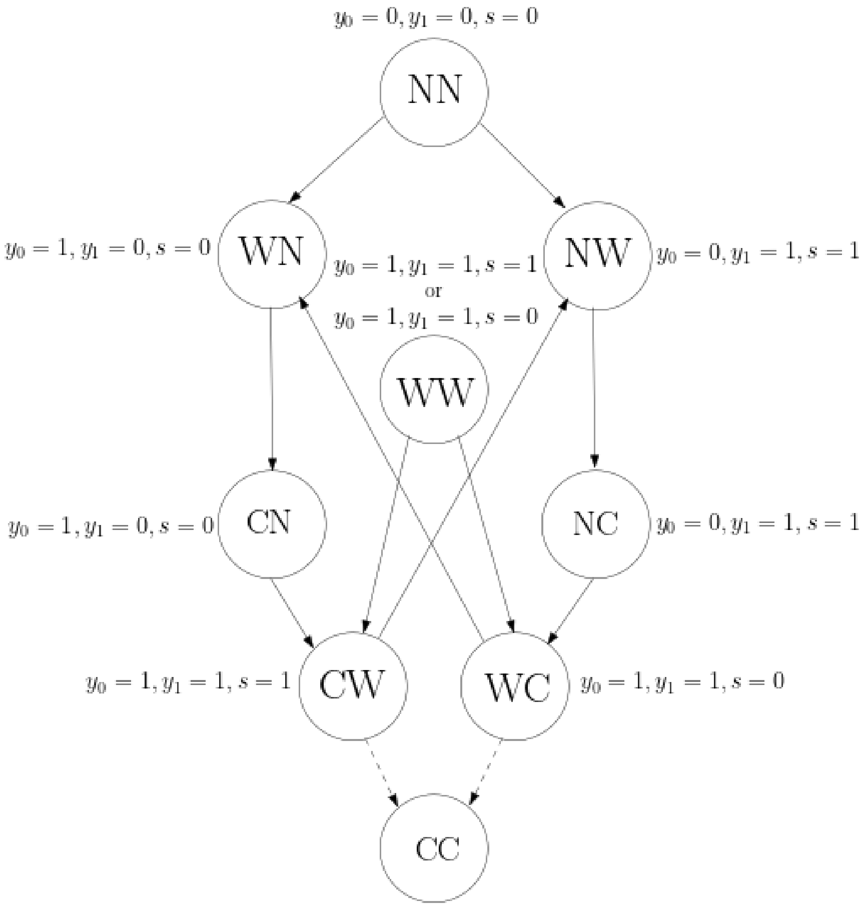
$y_0 = 0, y_1 = 0, s = 0$
NN
$y_0 = 1, y_1 = 0, s = 0$
WN
$y_0 = 1, y_1 = 1, s = 1$
or
$y_0 = 1, y_1 = 1, s = 0$
NW
$y_0 = 0, y_1 = 1, s = 1$
WW
$y_0 = 1, y_1 = 0, s = 0$
CN
NC
$y_0 = 0, y_1 = 1, s = 1$
$y_0 = 1, y_1 = 1, s = 1$
CW
WC
$y_0 = 1, y_1 = 1, s = 0$
CC

Fig. 2.5 Parallel composition of the two processes

We also modelled the same in NuSMV to verify our findings. The code for the same follows,

```
MODULE program1(y0,y1,s,p)

VAR
    location:{nc,w,c,exit};

ASSIGN
    init(location):=nc;
    next(location):=case
                    (location=nc):w;
                    ((location=w) & ((y1=0) | (s!=p))): c;
                    (location=c):nc;
                    TRUE:location;
                    esac;
    next(y0):=case
                (location=nc):1;
                (location=c):0;
                TRUE:y0;
              esac;
    next(y1):=case
                TRUE:y1;
              esac;

    next(s):=case
                (location=nc):p;
                TRUE:s;
             esac;
```

```
 FAIRNESS running

MODULE main

VAR

    y0:0..1;

    y1:0..1;

    s:0..1;

    thread1:process program1(y0,y1,s,0);

    thread2:process program1(y1,y0,s,1);

ASSIGN

    init(y0):=0;

    init(y1):=0;

    init(s):=1;
```

check_ltlspec is used to check LTL properties in NuSMV, we can either specify it within the code itself which will return the output once the code is run or check for it via the interactive shell in NuSMV. Also, *FAIRNESS RUNNING* is used to ensure that the scheduler for the processes is a fair one. We verified the following property to be true in NuSMV which indicates that mutual exclusion is guaranteed by the system.

```
G(!thread1.location=c & thread2.location=c)
```

Dijkstra's Mutual Exclusion Algorithm

We also attempted to verify the following mutual exclusion algorithm [9] using model checking, which is a simplification of Dijkstra's mutual exclusion algorithm when there are just two processes.

```
1 Boolean array b(0;1) integer k,i,j,

2 comment This is the program for computer i, which may be,
either 0 or 1, computer j=/=i is the other one,1 or 0
C0: b(i):false;
C1: if k!=i then begin
C2: if not b(j) then go to C2;
    else k:=1i; go to C1 end;
    else critical section;
    b(i):=true;
    remainder of program;
    go to C0;
    end
```

We came up with the following program graph for the process running on computer 0. The graph for the process running on computer 1 has not been shown here as it is more or less similar to this.

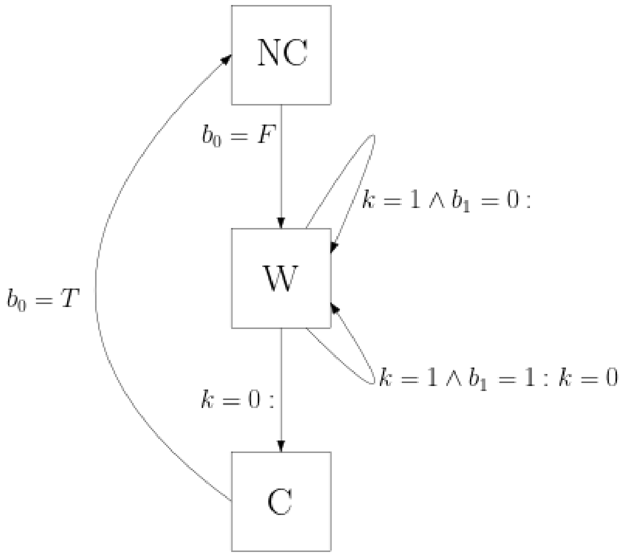

Fig. 2.6 Program graph for the process p_0 in the algorithm

From the program graph above, we developed a model of the algorithm was in NuSMV as follows,

```
MODULE program1(k,b,p)


VAR

    location:nc,w,c;

ASSIGN

    init(location):=nc;

    next(location):=case

                        location=nc:w;

                        location=w & k=p:c;

                        location=c:nc;

                        TRUE:location;

                    esac;


    next(b[p]):=case

                    (location=nc):0;

                    (location=c):1;

                    TRUE:b[p];

                esac;

    next(k):=case

                (location=w)&(k=1-p & b[1-p]=1):1-p;

                TRUE:k;

            esac;

FAIRNESS running

MODULE main

VAR
```

```
        k:0..1;
        b:array 0..1 of 0,1;
        process1:process program1(k,b,0);
        process2:process program1(k,b,1);
    ASSIGN
        init(b[0]):=1;
        init(b[1]):=1;
        init(k):=1;
```

The following specification returned true which confirms that the algorithm guarantees mutual exclusion.

```
G!(process1.location=c & process2.location=c)
```

Chapter 3

Modelling and analyzing an avionics triplex sensor voter

A core principle in the design of flights and avionics systems is that reliability is increased through redundancy. As such, for mission critical systems, it is customary to have multiple redundant components. Digital flight control systems make use of various sensors to measure different parameters for their navigation systems, guidance systems and control loops. When sensor redundancy is introduced, we must use a voter algorithm which would allow us to provide an output for as long as possible without jeopardizing the mission and also allow for fault detection and isolation if and when a failure is detected in a sensor. Our aim was to develop a model of a generic voting algorithm and verify it's properties to demonstrate how model checking can applicable to real world situations.

3.1 The System

The system we are currently taking a look at has triple modular redundancy, that is, there are 2 redundant sensors for every sensor. Many of the features of the system are taken from [10]. A Simulink diagram of the sensor voter module was available in [3]. As the internal workings of some of the modules were unknown to us, we proceeded with our abstraction of the same system.

Sensors are prone to various kinds of errors such as bias offsets, scale factor errors and sensitivity to various environmental factors. But we classify a sensor as failed, when it's output strays from the actual output by more than a predefined threshold. The threshold should be carefully selected, as too low a threshold would lead to a large number of sensor failures despite it being usable and too high a threshold would greatly increase the risk of using invalid inputs from a failed sensor. Sensors would also have a hardware flag that would identify that an internal hardware fault has occurred. To manage the sensor redundancy, we must use a voting algorithm. Sensor failure detection algorithms or voters must detect and isolate the failed sensors and provide an output for as long as the mission is not jeopardized.

3.1.1 The Voter Algorithm

The voter algorithm in question is a generic algorithm that is representative of the various algorithms in use today in real world systems. The functionality of the voter is as follows:

1. It is responsible for sampling the signals as well as the hardware flag of each sensor at a fixed rate.

2. It is up to the voter to detect faulty sensors and isolate them. This is done by by keeping a track of persistent hardware failure and comparing the redundant sensor values.

3. Provide the output at a constant rate by taking a weighted average of the sensor values and determine it's validity.

4. Sensors are considered valid if they are operating withing acceptable tolerances. The voter should tolerate any false alarms that may arise due to noise, transients and small differences in sensor measurements.

5. The voter is responsible for providing valid and accurate output for as long as possible.

3.1.2 Detecting Faults in the sensors

There are two methods by which a fault maybe detected in a sensor: by comparing the redundant sensor values and by monitoring the hardware flags of the sensors.

1. The differences between the values of the different pairs of sensors are computed. When the difference is past a specified threshold, a persistence counter is incremented by one. When it is within the threshold, it is decremented by one. When the counter itself reaches a certain threshold for two of the pair differences, the sensor that is common to both pairs is regarded as faulty.

2. The second method to detect a fault in a sensor is as earlier mentioned by checking the hardware flag of the signal. If it is returned false for three consecutive samples, then the sensor is regarded to be faulty.

3.1.3 Fault Handling

Initially, we have 3 valid sensors. We assume that there are no simultaneous sensor failures and that once a sensor is regarded faulty, it can no longer return to a non faulty state. Any one of these sensors may become faulty due to a hardware flag or a persistent miscompare. We deem the output to be invalid when all 3 pairs of sensors simultaneously miscompare. However, an output is still returned and it is up to the system user to decide whether to use the output or not.

When there are only 2 sensors, we come across a problem. When there is a miscompare, we cannot determine which sensor is faulty. To overcome this problem, we consider the output to be invalid until the miscompare clears. If one of the sensors register a hardware flag error, we disregard the sensor and proceed with just one sensor.

In the case with only one valid sensor, the output is quite simply the output of the last remaining sensor. When the sensor fails, we can no longer proceed with the mission and the output of the system is considered to be invalid. The different states in the fault handling mechanism can be seen in Figure. 3.1.3.

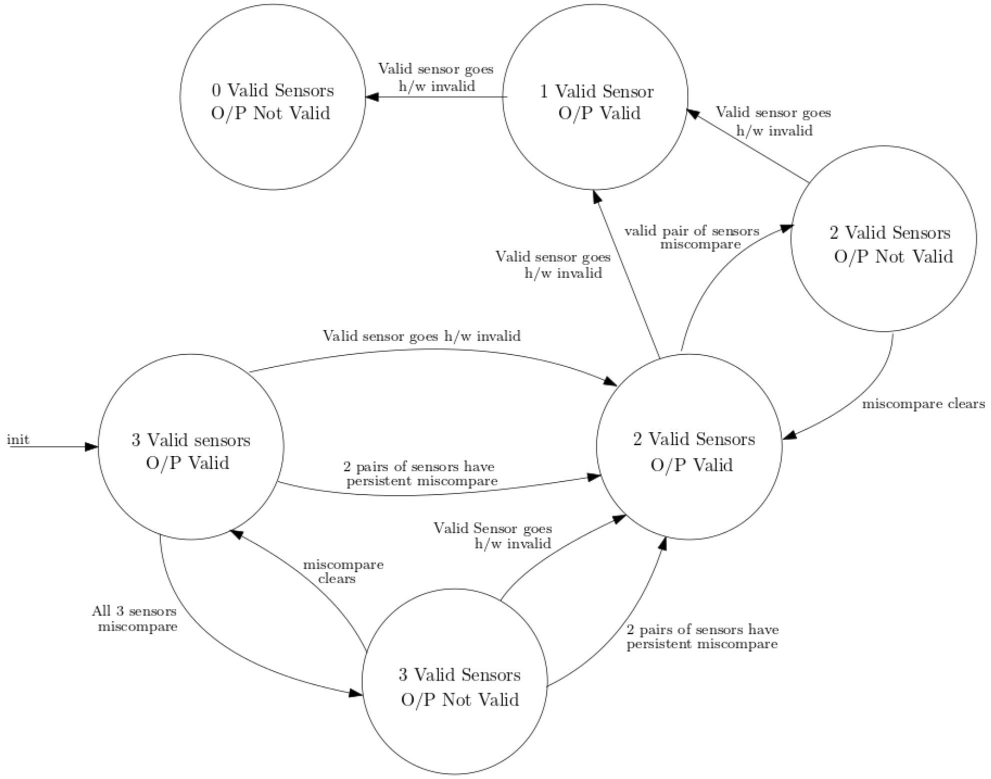

Fig. 3.1 Fault states of our system

3.1.4 Components of the system

The system can be divided into 3 major components, the world module, the sensor modules and the sensor voter module as shown in Figure 3.1.4.

The world module is responsible for generating the signals that are to be measured by the sensors. In our system, we intend it to only generate integer values in a limited range. Increasing the range of the values would lead to a larger state space as expected. We restrict ourselves to integers for the world signals and the sensor noise as floating point values are not available in NuSMV.

The sensor modules would take the signals from the world module as output and the module will allow to non-deterministically introduce noise to the input. The noise that can be introduced is also limited to a range of integers. The sensor module outputs the new

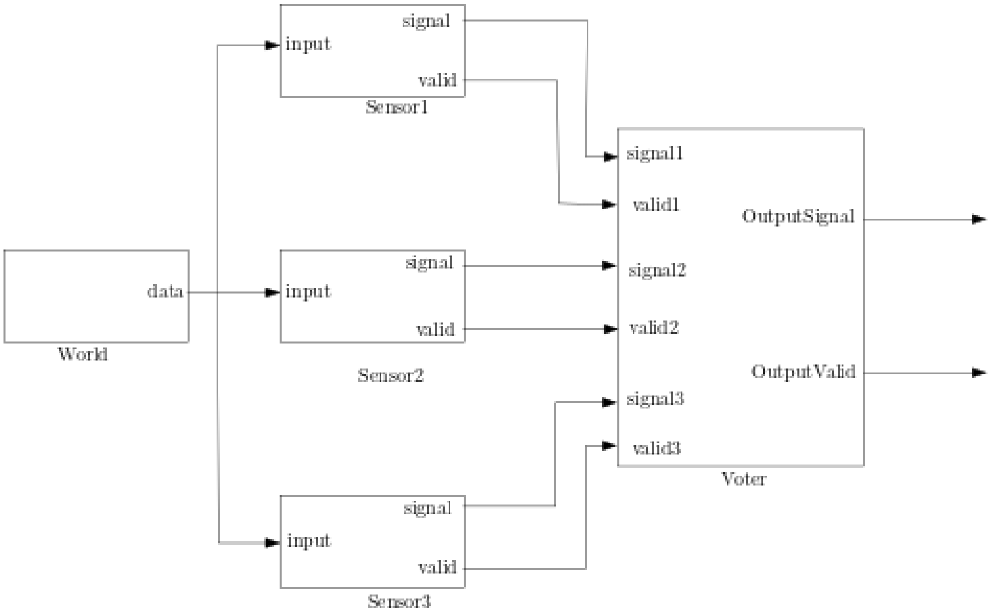

Fig. 3.2 SMV Modules of the system [3]

signal along with a hardware flag indicating the hardware status of the sensor.

The sensor voter module takes the inputs from the sensor and provides two outputs, an output signal and an output valid flag indicating whether the output is valid or not. The sensor module will have two counters, a persistent miscompare counter and a persistent hardware counter. The persistent miscompare counter is used to keep track of the number of times each pair of sensors miscompare and the persistent hardware counter is used to check for how many consecutive cycles a sensor has it's hardware flag set.

3.2 Modelling the system in NuSMV

As mentioned earlier, modelling the system in NuSMV involves primarily 3 modules: the world or main module, the voter module and the sensor module. There are also two types of variables involved: defined symbols and state variables. State variables as the name suggests contribute to the state space of the model while defined symbols do not. Defined symbols also only allow for forward definitions and cannot be assigned a value non-deterministically. The source code for our model can be found at this repo.

3.2.1 The Main Module

The main module is responsible for generating the world readings for the sensors as well as for initialising the 3 sensor modules and the voter module. As earlier mentioned, due to the limitations of NuSMV, we are restricted to using only integer values for the world readings. Being restricted to integer values also limit the size of our state space. We have restricted the world readings to a small range for our testing as a larger state space cannot be visualized in NuSMV.

3.2.2 The Sensor Module

The main module initialises the 3 sensors and each sensor non deterministically introduces noise to the world reading to produce the sensor's reading. The sensor also has another variable, the hardware flag for the sensor, which indicates whether the sensor is experiencing any hardware error at each state. Both the sensor reading and the hardware flag is then passed to the sensor voter module.

3.2.3 The Voter Module

The voter module is responsible for providing the output value as well as establishing it's validity. The validity of the output is determined as mentioned earlier.

The output is obtained as follows,

1. In the case of 3 valid sensors, when there is a miscompare in two pairs of sensors, the output is a weighted average of the readings from the 3 sensors, with less weight given to the sensor that is common to both pairs. When there is no miscompare or if all 3 pairs miscompare, then the output is a simple average of the readings.

2. When there are 2 valid sensors, the output is the simple average of the readings of the 2 sensors.

3. When there is only one valid sensor, the output is simply the reading from the lone sensor.

The voter module is also responsible for determining if a sensor is faulty or not. It makes use of the persistent miscompare counter and the persistent hardware counter to determine if the sensor is valid or not.

3.3 Verifying some properties

Once we built our model, we verified some properties of the system in NuSMV. We specified the properties that we checked using CTL. Some of the properties that we had checked are given below,

1. In the case of 3 valid sensors, once a sensor has failed either due to a persistent hardware fault or due to a persistent miscompare,then there is no possible means of recovery for the failed sensor.

   ```
   AG(votermodule.num_sensor_valid=2 -> !(EF(votermodule.num_sensor_valid=3)))
   ```

2. Similarly, in the case of 2 valid sensors, there is no possibility of returning to a state with 2 valid sensors once a sensor has failed.

   ```
   AG(votermodule.num_sensor_valid=1 -> !(EF(votermodule.num_sensor_valid=2)))
   ```

3. And in the case of a single valid sensor as well, once all sensors have failed, there is no possibility of recovery.

   ```
   AG(votermodule.num_sensor_valid=0 -> !(EF(votermodule.num_sensor_valid=1)))
   ```

4. This specification is used to check whether the output is declared invalid when there is a miscompare in the case of 2 valid sensors and if the sensors remain in service.

```
EF(A[(votermodule.num_sensor_valid=2 & votermodule.outputvalid=TRUE)U
(votermodule.num_sensor_valid=2 & votermodule.outputvalid=FALSE)]) &
EF(A[(votermodule.num_sensor_valid=2 & votermodule.outputvalid=FALSE)U
(votermodule.num_sensor_valid=2 & votermodule.outputvalid=TRUE)])
```

5. The next property deals with the case of 2 miscomparing sensors, with one of the sensors having a hardware fault. The expected behaviour is for the faulty sensor to get isolated and the output of the remaining sensor be considered as the valid output.

   ```
   EF(A[(votermodule.num_sensor_valid=2 & votermodule.outputvalid=FALSE) U
   (votermodule.num_sensor_valid=1 & votermodule.outputvalid=TRUE)])
   ```

6. This specification checks whether the output is produced correctly by the voter module. When the output signal is different from the original world reading, it checks whether the output will eventually stabilize by removing a faulty sensor or not return a valid output.

   ```
   AG(!(votermodule.op_value=worldreading)->
   AF(A[!(votermodule.op_value=worldreading) U
   ((votermodule.op_value=worldreading) &
   votermodule.outputvalid)] |
   !(votermodule.op_value=worldreading)))
   ```

7. This specification is used to check whether the output is within the specified error threshold whenver the output is valid.

   ```
   AG(votermodule.outputvalid ->
   abs((worldreading)-(votermodule.op_value)) <= errorthreshold)
   ```

Our current system, as stated earlier, is an abstraction of the system in [3], as there is no official documentation on how the original system functions. We also attempted to modify the system slightly to make some improvements to the original design. However, the core functionalities and operations of the voter remain the same in our system. The developed model of the system also meets the requirements of the system as verified in NuSMV. In the next chapter, we will look at how we further developed this model to design and analyze a quad redundant sensor voter system.

Chapter 4

Modelling and Analyzing a Quad-Redundant Sensor Voter

We developed a model of a quad-redundant sensor in NuSMV by iteratively building upon our model of a triplex redundant voter system. Verifying various properties of the system after building our model allowed us to identify problems and intricacies in our design and after completion, ensure the system satisfied all it's requirements. Through this project, we aimed to advocate for model checking in the design and analysis of avionic systems and components. The following sections will walk through how we modelled the system and also discuss a few properties that we had verified in NuSMV.

4.1 Modelling the system

Our design of the system and it's requirements were mostly similar to the the triplex voter system[3]. However, we had to accommodate a fourth sensor and modify the voting algorithm to make a decision in the case of a sensor failing when there are 4 valid sensors. Once we gathered our requirements and came up with an abstraction for the design of a 4 senor voter system, we proceeded to model the system.

The system can be divided into 3 major components as in the case of the triplex sensor voter system, the world module, the sensor modules and the sensor voter module as shown in Figure 4.1 .

The world module and the sensor modules are similar their counterparts in the triplex sensor voter system, the primary difference being the introduction of an extra sensor.

The sensor voter module takes the inputs from the sensor and provides two outputs, an output signal and an output valid flag indicating whether the output is valid or not. The sensor module will now have three counters, a persistent miscompare counter and a persistent hardware counter as in the case of our triplex voter and also a new persistent median deviation counter. The persistent median deviation counter as the name suggests will keep track of how many cycles a sensor deviates from the median.

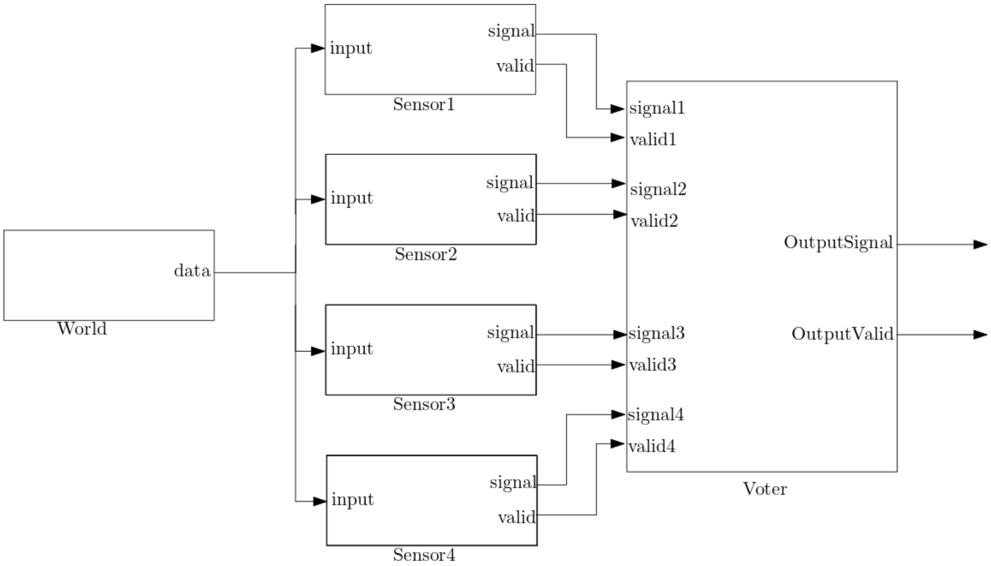

Fig. 4.1 SMV Modules of the system[3]

4.2 Modifications in Fault Detection for the 4 Sensor Model

We used a median based approach to accommodate the four sensor model. In the case of 4 valid sensors, we first calculate the median of the output of the sensor values. A persistence counter is incremented for any sensor that deviates from the median by a certain threshold. Once the persistence counter for a sensor reaches a predefined threshold, it is deemed to be faulty and is not considered for computing the output any further. In the case of all sensors deviating from the median, we consider the output to be invalid. The output is obtained by taking a weighted average of the sensor values, with lower weights assigned to sensors that deviate from the median as in the case of 3 valid sensors. The fault states for the quad redundant sensor model can be seen below in figure 4.2

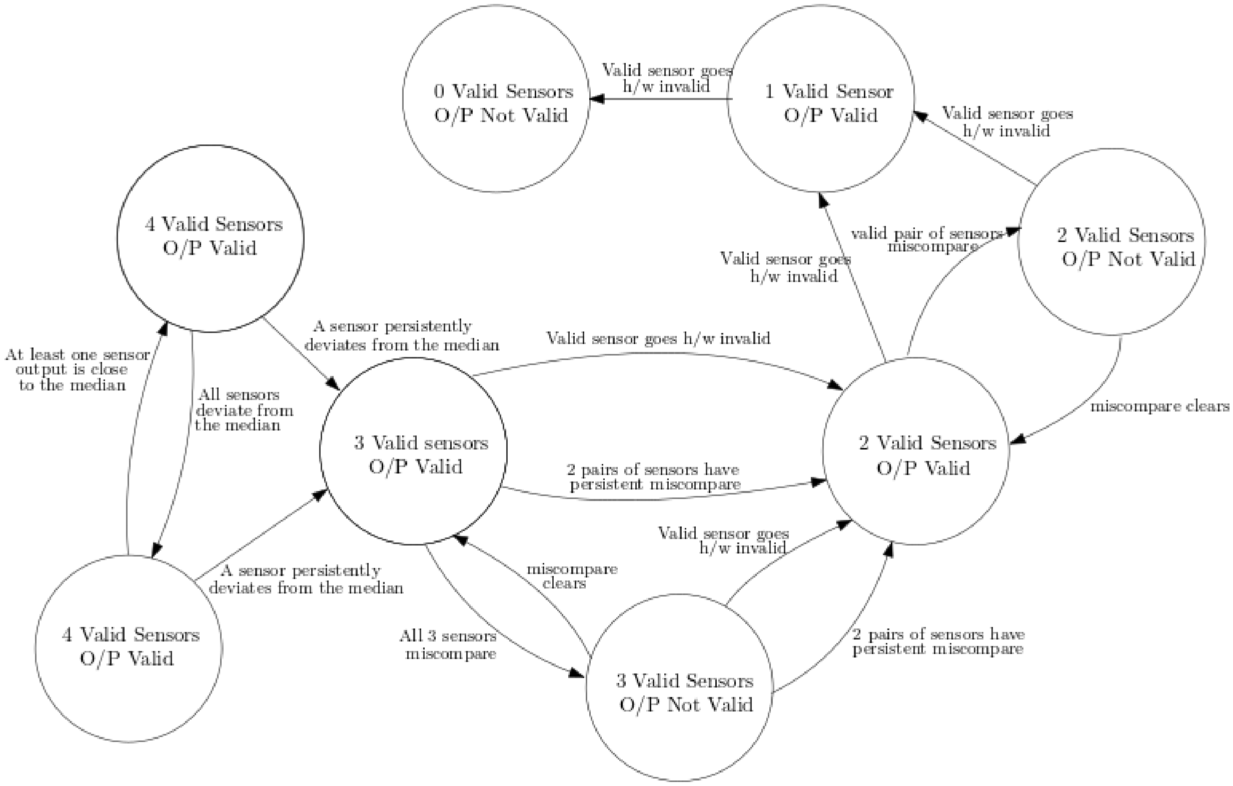

Fig. 4.2 Fault States of the quad sensor model

4.3 Modelling the system in NuSMV

We developed our model of a quad redundant sensor voter system by using our previously developed model for a triplex voter system as a foundation. The source code for our model can be found at this repo. As in the triplex voter system, our model for the quad redundant sensor voter system is comprised of 3 modules: the world or main modules, the sensor modules and the sensor voter module. The main module and the sensor modules remain largely similar to their counterparts in the triplex voter system. However, significant changes have been made to the voter module.

4.3.1 The Main Module

The main module is responsible for generating the world readings for the sensors as well as for initialising the 4 sensor modules and the voter module. As earlier mentioned, due to the limitations of NuSMV, we are restricted to using only integer values for the world readings.

4.3.2 The Sensor Module

The main module initialises the 4 sensors and each sensor non deterministically introduces noise to the world reading to produce the sensor's reading. The sensor also has another variable, the hardware flag for the sensor, which indicates whether the sensor is experiencing any hardware error at each state. Both the sensor reading and the hardware flag is then passed to the sensor voter module.

4.3.3 The Voter Module

The voter module is responsible for providing the output value as well as establishing it's validity. The validity of the output is determined as mentioned earlier.

The output is obtained as follows,

1. In the case of 4 valid sensors, if any of the sensors deviate from the median beyond a certain threshold, the output is a weighted average of the readings from the sensors, with more weight being given to the sensors within the threshold. If no sensors deviate from the median, the output is a simple average of the readings.

2. In the case of 3 valid sensors, when there is a miscompare in two pairs of sensors, the output is a weighted average of the readings from the 3 sensors, with less weight

given to the sensor that is common to both pairs. When there is no miscompare or if all 3 pairs miscompare, then the output is a simple average of the readings.

3. When there are 2 valid sensors, the output is the simple average of the readings of the 2 sensors.

4. When there is only one valid sensor, the output is simply the reading from the lone sensor.

The voter module is also responsible for determining if a sensor is faulty or not. It makes use of the persistent miscompare counter and the persistent hardware counter to determine if the sensor is valid or not.

4.4 Verifying some Properties

With the initial model complete, we can verify it's properties using NuSMV. By verifying various properties, we can identify flaws in the system and modify the design as we see fit. This is a much more cost efficient approach when compared to first coming up with a prototype and running extensive simulations on it. We specified the properties checked using CTL. Some of the properties we verified to be true are below. We once again verified the specifications we tested in the triplex sensor voter system and they were verified to be true as well.

1. In the case of 4 valid sensors, once a sensor has failed either due to a persistent median deviation or due to a hardware fault, then there is no possibility of returning to a state with 4 sensors.

   ```
   AG(votermodule.num_sensor_valid=3 -> !(EF(votermodule.num_sensor_valid=4)))
   ```

2. In the case of 3 valid sensors, once a sensor has failed either due to a persistent hardware fault or due to a persistent miscompare,then there is no possible means of

recovery for the failed sensor.

```
AG(votermodule.num_sensor_valid=2 -> !(EF(votermodule.num_sensor_valid=3)))
```

3. Similarly, in the case of 2 valid sensors, there is no possibility of returning to a state with 2 valid sensors once a sensor has failed.

```
AG(votermodule.num_sensor_valid=1 -> !(EF(votermodule.num_sensor_valid=2)))
```

4. And in the case of a single valid sensor as well, once all sensors have failed, there is no possibility of recovery.

```
AG(votermodule.num_sensor_valid=0 -> !(EF(votermodule.num_sensor_valid=1)))
```

5. In the case of 4 valid sensors, when the output is declared invalid, there is a possibility to return to a state where the output is valid without eliminating any sensor.

```
AG((votermodule.num_sensor_valid=4 & votermodule.outputvalid=FALSE)->
    EF(votermodule.num_sensor_valid=4 & votermodule.outputvalid= TRUE))
```

6. This specification is used to check whether the output is declared invalid when there is a miscompare in the case of 2 valid sensors and if the sensors remain in service.

```
EF(A[(votermodule.num_sensor_valid=2 & votermodule.outputvalid=TRUE)U
(votermodule.num_sensor_valid=2 & votermodule.outputvalid=FALSE)]) &
EF(A[(votermodule.num_sensor_valid=2 & votermodule.outputvalid=FALSE)U
(votermodule.num_sensor_valid=2 & votermodule.outputvalid=TRUE)])
```

7. The next property deals with the case of 2 miscomparing sensors, with one of the sensors having a hardware fault. The expected behaviour is for the faulty sensor to get isolated and the output of the remaining sensor be considered as the valid output.

```
EF(A[(votermodule.num_sensor_valid=2 & votermodule.outputvalid=FALSE) U
(votermodule.num_sensor_valid=1 & votermodule.outputvalid=TRUE)])
```

8. This specification checks whether the output is produced correctly by the voter module. When the output signal is different from the original world reading, it checks whether the output will eventually stabilize by removing a faulty sensor or not return a valid output.

```
AG(!(votermodule.op_value=worldreading)->
AF(A[!(votermodule.op_value=worldreading) U
((votermodule.op_value=worldreading) &
votermodule.outputvalid)] |
!(votermodule.op_value=worldreading)))
```

9. This specification is used to check whether the output is within the specified error threshold whenver the output is valid.

```
AG(votermodule.outputvalid ->
abs((worldreading)-(votermodule.op_value)) <= errorthreshold)
```

Chapter 5

Conclusion and Future Work

This report provided an overview of how model checking can be implemented practically in the design and analysis of avionic systems. We successfully designed a quad-redundant voter system and subsequently verified it's properties in NuSMV. Integrating model checking into the design process of avionic systems could lead to a substantial decrement in the costs involved. This report also provided an overview of my learning of model checking, detailing on the theoretical side of things as well as its practical implementation using NuSMV. The code for all my NuSMV implementations can be found at my github repository, here.

In the future, we are considering to expand our system design a bit further and design a 3 axis version of the same system and explore the various limitations of our design using model checking. We are also considering exploring other methods of fault detection and isolation in the 4 sensor model and eventually compare the different models to determine which is the most efficient. I also hope to further study the area and in general, develop a more thorough understanding of the intricacies involved in model checking.

References

[1] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[2] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new Symbolic Model Verifier," in *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds., no. 1633. Trento, Italy: Springer, July 1999, pp. 495–499.

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

[4] M. Fredrikson, "Automated program verification and testing 15414," 2016. [Online]. Available: https://www.cs.cmu.edu/~mfredrik/15414/lectures/

[5] R. B. Krug, "Ctl vs. ltl," https://www.cs.utexas.edu/users/moore/acl2/seminar/2010.05-19-krug/slides.pdf, May 2010.

[6] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. USA: Cambridge University Press, 2004.

[7] B. Srivathsan, "Model checking and system verification," 2015. [Online]. Available: https://www.cmi.ac.in/~sri/Courses/MC2015/

[8] W.-P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhneche, M. Poel, and J. Zwiers, *Concurrency Verification: Introduction to Compositional and Noncompo-*

sitional Methods, ser. Cambridge Tracts in Theoretical Computer Science. United Kingdom: Cambridge University Press, 2001.

[9] H. Hyman, "Comments on a problem in concurrent programming control," *Commun. ACM*, vol. 9, no. 1, p. 45, Jan. 1966. [Online]. Available: https://doi.org/10.1145/365153.365167

[10] S. Osder, "Practical view of redundancy management application and theory," *Journal of Guidance Control and Dynamics*, vol. 22, pp. 12–21, 1999.