Comparative study of an explicit state and symbolic model checking using a TCAS system

Final Project Report

Avinash Saikia
Department of Aerospace Engineering

Chandan Kumar
Department of Computer Science

Iowa State University

AERE/COMS 407X/507X

Acknowledgement

We would like to extend our gratitude to Prof. Kristin Y Rozier for her tremendous help and effort in making this project a success. We would also like to thank her for teaching us the course on *Applied Formal Methods* which enabled us to learn various tools and concepts pertaining not only to this project but also the wide range of application in real life.

We would also like to thank our colleagues and classmates for their constructive criticism, encouragement and suggestions.Last but not the least we would like to thank Iowa State University for providing the platform and facilities , which helped greatly in the success of this project.

Contents

1 Intoduction

Testing and Simulation is one way to determine the effectiveness of a model , but there are several limitations to it when it comes to output generation. Many a times these activities are done in an "idle" situation and simulation of a real-life scenario is not possible. In software testing, we cannot come to a conclusion that "if it works for 2 data points, then it will work for every data points". Also, there are various issues related to software testing - there could be bugs that are very difficult to find/notice . Testing cannot solely depend on boundary conditions.

Also, the reliability of a software is questionable. It cannot really give the precise results if the interaction between the software and the hardware is changed, for example a hardware can affect the output of the software. A software can have infinite states whereas a hardware has a clear difference between data and controls- fixed number of bits. To overcome these challenges one might ponder- "How are we going to do it?". This is when Formal Methods and its applications come into play.

1.1 Formal Methods

Formal Methods is a numerically rigorous technique for specific design and verification of software and hardware systems. In Formal Methods both the output and input are user controlled and is based on 2 factors:

a What it does?

b How it behaves?

where "it" refers to a system.
The Formal Methods works on formal language also known as temporal logic formulas-which are based on propositional logic. They are

a Linear Temporal Logic (LTL)

b Computational Tree Logic (CTL)

c CTL*

There are two kinds of Formal Methods viz. Model Checking and Theorem Proving.

a Model Checking:
Clarke & Emerson 1981 et al. "Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for (a given initial state in) that model." Its intuitive definition is :

$$M \vdash \Phi$$

b Theorem Proving:
It describes the system in formal language and satisfies type checking and other proof obligations. One can introduce behaviour properties: axioms, rules and system behaviours.

This report deals with the comparison of 2 kinds of model checking tools- SPIN which is an explicit state model checker and nuXmv which is a symbolic model checker .A performance analysis has been done on these tools using TCAS (Traffic Collision Avoidance System).

1.2 SPIN

SPIN,[2] developed in the year 1980, was one of the very first model checker. It introduced a classical approach in Model Checking for LTL specifications. Model checking could be done on-the-fly. The specifications were written down in Promela and the properties were LTL. The properties compiled in a Büchi automaton. For on-the-fly verifier, SPIN generates a source code in C. The verifier, upon compilation, checks whether the property is being satisfied by the model. A global state transition graph construction is avoided by on-the-fly approach. But, this also means that each transition for a property being verified is recomputed. Hence for the verification of n properties, it is computed n times. Promela is the language used to specify models in SPIN, is inspired from C. It is, hence, an imperative language that has constructs to handle concurrent processes. Like C, it also offers basic data types like char, int, bit and arrays. It has a channel structure over which several processes can read or write. The length of channel depends whether on whether the processes are communicating synchronously or asynchronously. For synchronous communication, the channel length is 0, and for asynchronous it is greater than 0. Also, there is an atomic operator that considers even a compound structure as a single atomic transition. The compound statement works as atomic in all cases except when the statement contains a guard statement or when it blocks a read or write over a channel. In such cases the control is transferred to another process and the atomic construct gets interrupted. In SPIN, transition notion is absent, instead it only considers states. For a LTL formula to hold true for any specification, it must hold true for every possible run of Promela model. A run in Promela is a trace of execution that comprises of the sequence of states that were visited during execution. It can run for infinite states.

1.3 NuXMV

NuXMV [2] inherits all the functionalities of NuSMV. NuSMV is used for formal verification of finite state systems. It is a SMV based model checker. SMV stands for Symbolic Model Verifier. Since the domain of definition was only finite state models, the data types are Boolean, scalar, bit vectors and fixed arrays. Symbolic Model Checking was first implemented using NuSMV. This is a technique in which symbolic notation of a specification is used to check against a property. This comprises of a class of model checkers that verifies temporal properties of a finite state system. Initially SMV was a tool to check only CTL properties of a symbolic model. But SMV has evolved over period of time. Now it can check LTL properties as well. It can now also deal with SAT-based Bounded Model Checking. The language used by NuSMV is SMV which is a description language to specify finite state machines. Specifying a machine means, declaration of modules which may have declarations of variables or/and constraints viz. transition constraints or assignment constraints. NuSMV specifications can be specified as both state as well as event oriented. CTL properties are expressed by State variables. LTL properties are expressed by

state as well as input variables. Invariant specifications gives faster results than CTL or LTL algorithms.

2 Motivation

The Formal Methods world has a variety of tools. Although most function towards a common goal and each model checking tool has its unique way of implementation, the question arises - Which is the best tool for my model? To answer this question, a study has been conducted on a TCAS model and a comparative study has been done over two types of model Checkers- Explicit state (SPIN) and Symbolic (NuXMV).

Past studies have shown that the right choice- Symbolic or Explicit state is quite unclear. Some experiments have argued that symbolic model checking performs better for a synchronous system with hardware like characteristics whereas Explicit State model checking is better for asynchronous systems with a number of communication processes. However, some other experiments report that a symbolic model checker performs better even for asynchronous systems. However, Gerard Holzmann, the inventor of SPIN, has challenged the result saying that the results can be reversed by carefully optimizing the SPIN model. Thus, the correctness of the claims is quite contradicting.

This report will show the performance and limitations of each model checking tools for a TCAS system. One might outperform another but one cannot simply conclude based on just one model. Thus, all the conclusions made in this report will be in regard to this particular TCAS model only.

	SPIN	NUXMV
Type	Explicit State	Symbolic
Temporal Logic	LTL only	LTL & CTL
System Description	High Level Language - Promela	Description of Transition system
Verification	Assertion , LTL , Fairness, Invariants	LTL, CTL , Invariants , Fairness

Table 1. Comparison between SPIN and NUXMV

3 TCAS

3.1 Introduction

TCAS or Traffic Collision Avoidance System is a System that is used to avoid or minimize mid-air collisions of aircrafts. Its main function is to monitor any intruder aircraft in the vicinity

of the designated “safe zone” / airspace. The TCAS consists of an active Transponder, and is independent of Air traffic control (ATC).It is a responsive system that helps tackle any unanticipated pilot response or possible mid-air collision due an undesirable intruder flight.

3.2 History

The research in TCAS is active since the 1950s. International Civil Aviation Organization and Federal Aviation Administration began this research after the Grand Canyon mid-air collision in 1956. Various other Incidents such as the Charkhi Dadri mid-air collision over New Delhi in 1996, Japan Airlines near-miss incident in 2001 and also Überlingen mid-air collision, between a Boeing 757 and a Tupolev Tu-154 in 2002 etc. sparked questions about air transport safety and led to research of mid-air collision. And ever since, various progress have been made and TCAS is one of them. Other Airborne collision avoidance include – next Generation Airborne Collision Avoidance System (ACAS-X).

3.3 Working of TCAS

The TCAS involves a communication between all the aircraft components equipped with a transponder. For example a distance(say minimum safe distance) between 2 aircraft is noted by the system , the TCAS corresponds in the form a warning saying an intruder is in the vicinity of the airspace. The TCAS waits for the pilot to take action, and if the pilot does not respond within a certain period of time it takes maneuvering action. It also nullifies the ATC when the danger is critical. It basically controls al the critical systems and modeling this system needs extreme care and details. TCAS and Formal Methods go hand in hand, Model Checking Formal Method tools are used to check the TCAS model for any counterexamples that the modeller might have missed. Linear Temporal Logic specifications are used to check for the safety property of the system, it is also used to check any liveness or fairness property. And after running the system in a tool, any counterexamples produced plays a vital role in the re-modeling the system.
The TCAS model used here is a rather simple and fictional and is used only for the comparison of two types of model Checkers which is the main focus of this Project. The TCAS model we have used here has very minimal specifications and are such due to the interest of time. The Specifications ate described in the next sub section.

3.4 Specifications

1. The domain includes only small 2-4 seater planes (like Cessna or Dornier aircrafts).
2. The planes can fly only within the 4 layers of atmosphere (Layer_1, Layer_2, Layer_3 and Layer_4) , beyond and below these range, it is regarded as unsafe region- included region lesser than Layer_1 (below the ground).
3. There are 3 zones of distance that separate 2 planes :
   Distance : L600, L1000, L1600

The region beyond L1600 is for visualization ,and if the pilot don't react when the intruder plane is at the threshold of L1600, Auto Resolver takes command.

4. When there is a conflict. If the planes travels :
   a from east to west , the plane climbs down
   b From west to east , the plane climbs up
   c From north to south, the plane climbs down
   d From south to north , the plane climbs up.
5. All planes are cruising at layer_3. The planes transitions from Layer_3 to either Layer_2(climb down) or Layer_1 (climb up)

Fig. 1 Demonstrates a graphical representation of our model

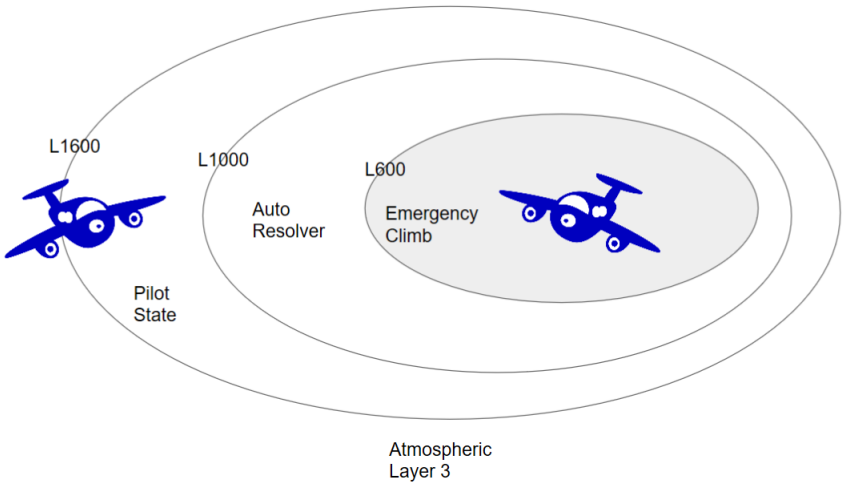

Fig. 1 Layout

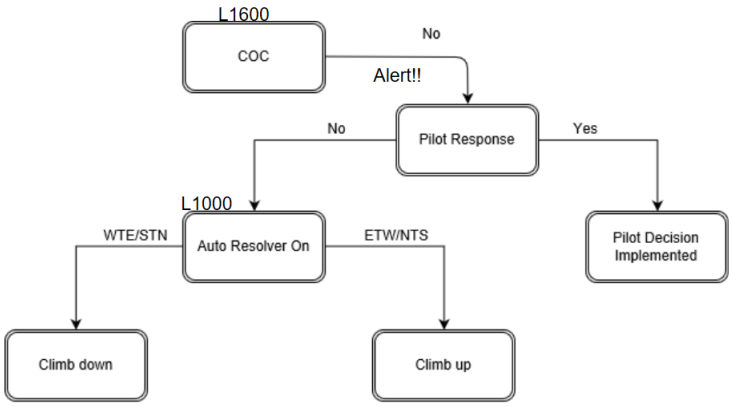

Fig. 2 Working Principle of the TCAS system

4 Results and Discussion

4.1 Model Validation

The main motive of this study was fulfilled and the results obtained was quite satisfactory.
To validate the model the initial step was to check if the safety property satisfies. Initially many counter examples were given by both SPIN and NuXMV, after closely identifying the errors and rectifying each model, no counter examples were provided. Since the plane has varying start state like moving from east to west, north to south etc. and also has many actions according to move direction , the LTL specifications also varied for each condition. The LTL Specifications are as follows. The first LTL Specification is to check if there is no clear of conflict at all. The Specification was

$$\square\neg COC$$

NuXMV returned a comment saying the LTL spec is true
Now a detailed LTL Specification is checked again:

$$\square(\neg COC \wedge \neg pilot\_response \wedge (direction = ETW \vee NTS) \wedge (atm\_layer = layer\_3)$$

$$\Rightarrow auto\_resolver \wedge climb\_up \wedge (atm\_layer = layer\_4)$$

This LTL specification is for the case when the plane is moving east to west or north to south, whenever there is conflict , if there is no pilot response , then auto resolver will be turned on and the plane will climb up to atmospheric layer 4, thus avoiding any possible collision.

$$\square(\neg COC \wedge \neg pilot\_response \wedge (direction = WTE \vee STN) \wedge (atm\_layer = layer\_3)$$

$$\Rightarrow auto\_resolver \wedge climb\_down \wedge (atm\_layer = layer\_2)$$

This LTL specification is for the case when the plane is moving west to east or south to north, whenever there is conflict , if there is no pilot response , then auto resolver will be turned on and the plane will climb down to atmospheric layer 2, thus avoiding any possible collision.

All of these specifications returned :

```
-- specification is true
```

This concludes our model verification experiment

4.2 Performance Analysis

While modelling the TCAS, we tried to keep the variables same for each tool. While there was no error in the NuXMV model , however in the SPIN model there was a case of **State Explosion**. There was a time-out after 345279 steps and a screenshot of the time-out is displayed below.

```
345271: proc  1 (fly:1) promela_e2.pml:21 (state 25)
[atm_layer[2] = 1]
345272: proc  1 (fly:1) promela_e2.pml:21 (state 26)
[climb[2] = 1]
345273: proc  1 (fly:1) promela_e2.pml:21 (state 27)
[COC = 1]
345274: proc  1 (fly:1) promela_e2.pml:21 (state 28)
[auto_resolver = 0]
345275: proc  1 (fly:1) promela_e2.pml:21 (state 29)
[pilot_response = 0]
                pilot_response = 0
345276: proc  1 (fly:1) promela_e2.pml:38 (state 70)
[.(goto)]
345277: proc  1 (fly:1) promela_e2.pml:20 (state 19)
[atm_layer[2] = 1]
345278: proc  1 (fly:1) promela_e2.pml:20 (state 20)
[climb[2] = 1]
345279: proc  1 (fly:1) /var/www/courses/sefm/spi
Command exited with error code 124, which usually
means a timeout.
```

fig 3. State Explosion in SPIN

This state-explosion demonstration shows the limitations of SPIN. Using continuous values of distance will lead to a state-explosion. This problem can be resolved by using "zones" as shown in figure 1. Zones can be Boolean and it simplifies the problem to a great extent. However use of zones instead of numerical values of distance also causes limitations to the TCAS model in general. The SPIN models can be found in the Appendix of this report.

4.3 Counterexamples

As we see from previous result that the model is correct and the LTL specifications are satisfied . But how can one validate a model? The negation of the LTL specification will simulate a bad state. The counter examples provided by each tool is fairly the same . The counter example

```
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
horizontal_dist = 0
move_direction = ETW
pilot_response = TRUE
atm_layer = layer_3
climb = no
next_atm_layer = layer_1
climb_status = up
auto_resolver = FALSE
COC = FALSE
```

From this counterexample we can see that there is a conflict despite there is a response from the pilot, it assumes a state when 2 planes start at the same point hence the horizontal distance = 0. Although this is practically impossible, this counterexample we can see how detailed and important model Checking is to trace out possible conflicts which a human would miss or ignore because it is not and unreal for a practical situation. Such counterexamples helps a modeler to find faults and help rectify minute errors that could possible lead to fatal situations.

5 Conclusion

From the results and discussions above one can conclude that Model Checking tools has their own limitations. And from this study we have found that SPIN is not the right choice for this particular TCAS system. Although NuXMV outperformed and showed minimal limitations, we cannot come to a conclusion about NuXMV being a better tool, however for this particular model with the limited specifications, NuXMV did perform well.
From this study we can state that assigning continuous values for a variable is not feasible for SPIN however using "zones" instead of distance can resolve the problem to an extent but practically , it will cause limitations for the TCAS model.
A better tool is totally dependent on the modeler and there is hard and fast rule about one tool being better over the other. The advantage of one particular model over another also solely depends on type of the model- some work well for synchronous systems, some of asynchronous whereas some work well for both synchronous and asynchronous systems.

6 Appendix

6.1 Appendix A

NuXMV model

```
MODULE main

VAR
horizontal_dist:0 .. 1600;
move_direction:{ETW ,WTE , NTS, STN};
pilot_response : boolean;

DEFINE
COC:= ((horizontal_dist>1600) & (auto_resolver=FALSE) & (atm_layer=layer_3)) ;
auto_resolver:= ((horizontal_dist<=1000) & (pilot_response=FALSE));

ASSIGN
init(pilot_response):=TRUE;



VAR
atm_layer : {layer_1,layer_2,layer_3,layer_4};

ASSIGN
init(atm_layer):=layer_3;

next(atm_layer):=
case
(atm_layer=layer_3) & (COC=FALSE) & (pilot_response=FALSE ) & (horizontal_dist
    <=1000) & (move_direction=ETW)&(auto_resolver=TRUE) :layer_4;
(atm_layer=layer_3) & (COC=FALSE) & (pilot_response=FALSE ) & (horizontal_dist
    <=1000) & (move_direction=WTE)&(auto_resolver=TRUE):layer_2;
(atm_layer=layer_3) & (COC=FALSE) & (pilot_response=FALSE ) & (horizontal_dist
    <=1000) & (move_direction=NTS)&(auto_resolver=TRUE):layer_2;
(atm_layer=layer_3) & (COC=FALSE) & (pilot_response=FALSE ) & (horizontal_dist
    <=1000) & (move_direction=STN)&(auto_resolver=TRUE):layer_4;
TRUE:atm_layer;
esac;


VAR
climb: {up, down,no};

ASSIGN
init(climb):=no;
next(climb):=
case
(atm_layer=layer_3) & (COC=FALSE) & (pilot_response=FALSE ) & (horizontal_dist
    <=1000) & (move_direction=ETW)&(auto_resolver=TRUE) :up;
(atm_layer=layer_3) & (COC=FALSE) & (pilot_response=FALSE ) & (horizontal_dist
    <=1000) & (move_direction=WTE)&(auto_resolver=TRUE) :down;
(atm_layer=layer_3) & (COC=FALSE) & (pilot_response=FALSE ) & (horizontal_dist
    <=1000) & (move_direction=NTS)&(auto_resolver=TRUE) :down;
(atm_layer=layer_3) & (COC=FALSE) & (pilot_response=FALSE ) & (horizontal_dist
    <=1000) & (move_direction=STN)&(auto_resolver=TRUE) :up;
```

```
TRUE:climb;
esac;

VAR
next_atm_layer : {layer_1,layer_2,layer_3,layer_4};
ASSIGN
next(next_atm_layer):=
case
(atm_layer=layer_4) & (COC=TRUE) & (pilot_response=FALSE ) & (horizontal_dist
    >1000) & (move_direction=ETW)&(auto_resolver=TRUE) :layer_3;
(atm_layer=layer_2) & (COC=TRUE) & (pilot_response=FALSE ) & (horizontal_dist
    >1000) & (move_direction=WTE)&(auto_resolver=TRUE) :layer_3;
(atm_layer=layer_2) & (COC=TRUE) & (pilot_response=FALSE ) & (horizontal_dist
    >1000) & (move_direction=NTS)&(auto_resolver=TRUE) :layer_3;
(atm_layer=layer_4) & (COC=TRUE) & (pilot_response=FALSE ) & (horizontal_dist
    >1000) & (move_direction=STN)&(auto_resolver=TRUE) :layer_3;
TRUE:next_atm_layer;
esac;
```

6.2 Appendix B

SPIN Model (with continuous values of distance)

```
int H;
bool COC, auto_resolver, pilot_response;
byte atm_layer[4];
/*atm_layer[0]=Layer1, atm_layer[1]=Layer2, atm_layer[2]=Layer3, atm_layer[3]=
    Layer4*/
byte move_direction[2];
/*move_direction[0]=1 -> east_to_west
move_direction[0]=0 -> west_to_east
move_direction[1]=1 -> north_to_south
move_direction[1]=0 -> south_to_north */
byte climb[3];
/*climb[0]=up, climb[1]=down, climb[2]=none*/

proctype fly(){
do
::atomic{atm_layer[2]==1 && climb[2]==1 && COC==0 && auto_resolver==0 &&
    pilot_response==0}
::atomic{atm_layer[2]==1 && climb[2]==1 && COC==0 && auto_resolver==0 &&
    pilot_response==1}
::atomic{atm_layer[2]=1; climb[2]==1; COC==0; auto_resolver==1; pilot_response
    ==0}
::atomic{atm_layer[2]==1; climb[2]==1; COC==0; auto_resolver==1; pilot_response
    ==1}
::atomic{atm_layer[2]==1; climb[2]==1; COC==1; auto_resolver==0; pilot_response
    ==0}
if
::(atm_layer[2]==1 && COC==0 && pilot_response==0 && move_direction[0]==0) -> (
    climb[0]==1 && atm_layer[1]==1)
::(atm_layer[2]==1 && COC==0 && pilot_response==0 && move_direction[0]==1) -> (
    climb[0]==1 && atm_layer[3]==1)
::(atm_layer[2]==1 && COC==0 && pilot_response==0 && move_direction[1]==0)-> (
    climb[1]==1 && atm_layer[1]==1)
::(atm_layer[2]==1 && COC==0 && pilot_response==0 && move_direction[1]==1) -> (
    climb[1]==1 && atm_layer[3]==1)
::(atm_layer[2]==1 && COC==1)-> (climb[2]==1);
::(atm_layer[0]==1) -> (atm_layer[1]==0 && atm_layer[2]==0 && atm_layer[3]==0);
::(atm_layer[1]==1) -> (atm_layer[0]==0 && atm_layer[2]==0 && atm_layer[3]==0);
::(atm_layer[2]==1) -> (atm_layer[0]==0 && atm_layer[1]==0 && atm_layer[3]==0);
::(atm_layer[3]==1) -> (atm_layer[0]==0 && atm_layer[1]==0 && atm_layer[2]==0);
::(COC==1) -> (auto_resolver==0 && pilot_response==1);
fi
::(H==0)-> break;
od
}

proctype horizontal(){
do
::atomic{atm_layer[2]==1 && climb[2]==1}
//::atomic{COC=1 -> H>0}
if
::(atm_layer[2]==1 && climb[2]==1 && H>1600) -> (COC==1 && auto_resolver==0 &&
    pilot_response==0)
::(atm_layer[2]==1 && climb[2]==1 && 1600>=H>1000) -> (COC==0 && auto_resolver
    ==0 && pilot_response==1)
```

```
::(atm_layer[2]==1 && climb[2]==1 && 1000>H) -> (COC==0 && auto_resolver==1 &&
    pilot_response==0 && climb[1]==1)
fi
//::(H==0)-> break;
od
}


init{
run fly();
run horizontal();
}
```

6.3 Appendix C

SPIN Model (with Boolean Values of distance not omitting continuous values of distance)

```
byte H_Zone[3];
/*H_Zone[0]=when H>1600, H_Zone[1]=when 1600>=H>1000, H_Zone[2]=1000>=H*/
bool COC, auto_resolver, pilot_response;
byte atm_layer[4];
/*atm_layer[0]=Layer1, atm_layer[1]=Layer2, atm_layer[2]=Layer3, atm_layer[3]=
    Layer4*/
byte move_direction[2];
/*move_direction[0]=1 -> east_to_west
move_direction[0]=0 -> west_to_east
move_direction[1]=1 -> north_to_south
move_direction[1]=0 -> south_to_north */
byte climb[3];
/*climb[0]=up, climb[1]=down, climb[2]=none*/

proctype fly(){
do
::atomic{atm_layer[2]==1 && climb[2]==1 && COC==0 && auto_resolver==0 &&
    pilot_response==0}
::atomic{atm_layer[2]==1 && climb[2]==1 && COC==0 && auto_resolver==0 &&
    pilot_response==1}
::atomic{atm_layer[2]=1; climb[2]==1; COC==0; auto_resolver==1; pilot_response
    ==0}
::atomic{atm_layer[2]==1; climb[2]==1; COC==0; auto_resolver==1; pilot_response
    ==1}
::atomic{atm_layer[2]==1; climb[2]==1; COC==1; auto_resolver==0; pilot_response
    ==0}
//::atomic{atm_layer[2]=1; climb[2]=1; COC=1; auto_resolver=0; pilot_response=1}
//::atomic{atm_layer[2]=1; climb[2]=1; COC=1; auto_resolver=1; pilot_response=0}
//::atomic{atm_layer[2]=1; climb[2]=1; COC=1; auto_resolver=1; pilot_response=1}
if
::(atm_layer[2]==1 && COC==0 && pilot_response==0 && move_direction[0]==0) -> (
    climb[0]==1 && atm_layer[1]==1)
::(atm_layer[2]==1 && COC==0 && pilot_response==0 && move_direction[0]==1) -> (
    climb[0]==1 && atm_layer[3]==1)
::(atm_layer[2]==1 && COC==0 && pilot_response==0 && move_direction[1]==0)-> (
    climb[1]==1 && atm_layer[1]==1)
::(atm_layer[2]==1 && COC==0 && pilot_response==0 && move_direction[1]==1) -> (
    climb[1]==1 && atm_layer[3]==1)
::(atm_layer[2]==1 && COC==1)-> (climb[2]==1);
::(atm_layer[0]==1) -> (atm_layer[1]==0 && atm_layer[2]==0 && atm_layer[3]==0);
::(atm_layer[1]==1) -> (atm_layer[0]==0 && atm_layer[2]==0 && atm_layer[3]==0);
::(atm_layer[2]==1) -> (atm_layer[0]==0 && atm_layer[1]==0 && atm_layer[3]==0);
::(atm_layer[3]==1) -> (atm_layer[0]==0 && atm_layer[1]==0 && atm_layer[2]==0);
::(COC==1) -> (auto_resolver==0 && pilot_response==1);
fi
od
}

proctype horizontal(){
do
::atomic{atm_layer[2]==1 && climb[2]==1}
if
::(atm_layer[2]==1 && climb[2]==1 && H_Zone[0]==1) -> (COC==1 && auto_resolver
    ==0 && pilot_response==0)
```

```
::(atm_layer[2]==1 && climb[2]==1 && H_Zone[1]==1) -> (COC==0 && auto_resolver
    ==0 && pilot_response==1)
::(atm_layer[2]==1 && climb[2]==1 && H_Zone[2]==1) -> (COC==0 && auto_resolver
    ==1 && pilot_response==0 && climb[1]==1)
fi
od
}


init{
run fly();
run horizontal();
}
```

References

[1] Choi, Y. (2007). From nusmv to spin: Experiences with model checking flight guidance system. *Formal Method in System Design 30*, 199–216.

[2] Eisner, C. and P. D. (2002). Comparing symbolic and explicit model checking of a software system. In *In: Bošnački D., Leue S. (eds) Model Checking Software. SPIN 2002. Lecture Notes in Computer Science*, Volume 2318, pp. 230–239. Berlin, Heidelberg: Springer.

[3] Holzmann, G. (2003). *Spin Model Checker, the: Primer and Reference Manual* (First ed.). Addison-Wesley Professional.

[4] Rozier, K. Y. (2010). Linear temporal logic symbolic model checking. *Computer Science Review 5*(2), 163–203.

[5] Ruys, T. C. (2002). Spin beginners' tutorial.

[6] Ruys, T. C. and H. Gerard (2004). Advanced spin tutorial.