Chapter 7
Arguing Formally About Flight Control Laws Using SLDV and NuSMV

Natasha Jeppu and Yogananda Jeppu

Abstract Software systems have failed in the recent past. This is most often attributed to wrong requirements often caught very late in the program or escapes from the rigorous process leading to failures. There is a necessity to ensure that the requirements are correct up front before the design and verification process start. Formal methods have become popular these days and a lot of impetus is there in the industry to apply these techniques to safety critical projects especially in flight controls. This paper looks at two tools NuSMV, an open source model checker, and Simulink Design Verifier, a commercial model checker. It is seen that these can be practically applied to projects and design. These are very successful in finding defects in design and requirements as demonstrated on a set of mutants.

Keywords Formal methods · NuSMV · Simulink design verifier
Safety critical · Aerospace

7.1 Introduction

Software has failed even as recently as 2015 in aerospace systems. The new bug list of F35 published in 2016 indicates that we have not mastered the art of engineering good software [1]. A preliminary study was reported on the use of Simulink Design Verifier (SLDV) for aerospace application [2]. This paper described mode transition logic and demonstrated how easy it was for engineering students to work with the SLDV software to describe behavior formally. Formal methods are mathematical techniques for specifying, developing, and verifying software. This statement is from the DO 333 aerospace standard supplement to DO 178C, which was

N. Jeppu (✉)
National Institute of Technology Karnataka, Surathkal, Mangalore, India
e-mail: Natasha.Jeppu@gmail.com

Y. Jeppu
Honeywell Technology Solutions, Bangalore, India
e-mail: Yogananda.Jeppu@Honeywell.com

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_7

released in late 2011 [3, 4]. This supplement brings out case studies on the use of formal methods in aerospace software development process. A good study of use of formal methods using DO 333 is also reported in the NASA report [5]. Formal methods have picked up impetus in the last few years with Airbus using it for reducing the test activity in their projects [6, 7]. Other aircraft companies are looking at using formal methods in the certification process. This has however not been very well accepted by the managers nor by the engineers [8].

The barriers to the utilization of formal methods in aerospace projects are brought out in a survey [9]. The major barrier indicated is surprisingly education. The engineer in the industry needs to be educated in the use of formal methods. There is a need for highly trained experts in formal methods as mentors for the engineers. Certification engineers need to be trained and educated in the formal methods process. This also brings out the necessity for practical application papers on the use of formal methods. There is a need for bringing out the ease or complexity of use of formal methods. This paper is an attempt to capture the experiences of using formal methods in form of the use of Simulink Design Verifier—a commercial formal methods tool from MathWorks [10]. An open source tool NuSMV is also used to compare the performance on mode transition logic problem [11]. Several models are available on the MathWorks Web site so that the community can use it to experience the use of the tool [12].

7.2 Simulink Design Verifier

This is a toolbox from MathWorks which work on the Simulink design. It is primarily a formal model checker. Model checking is defined as quote "Model checking is a verification technique that explores all possible system states in a brute-force manner" [13]. The design model is checked against the verification subsystem which is a set of assertions. There is a facility to put assumptions or constraints on the inputs and define an implication as a combination of logic blocks. SLDV supports timed automata. It comes with a set of blocks that can help the user set up time dependency in terms of number of frames. The newer version of SLDV has a feature to take a legacy C code, make it into an S-Function, and use it with the SLDV to find errors in the C code. An S-Function is a Simulink function compiled from a C code and executing in the Simulink environment.

A simple example of an autodestruct system demonstrates the use of SLDV. In an aerospace application, if the pitch angle ||Theta|| is >=25 degrees the autodestruct flag is set to True. This is modeled in Simulink as shown in Fig. 7.1. The system had an error—the absolute was missing in the code [1]. The formal assertions for the system are defined in Fig. 7.2. If the angle (In) >=25 OR angle <=−25 implies that (=>) the autodestruct (in1) input is True. The block (P) indicates to SLDV that this needs to be proved using the inbuilt model checker. SLDV proves that the model without the absolute is wrong and provides counter example with input "in" set to −25.0 to prove it wrong. This is a simple example but all the complicated

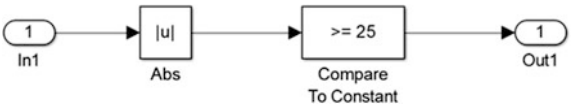

Fig. 7.1 Model of the autodestruct system

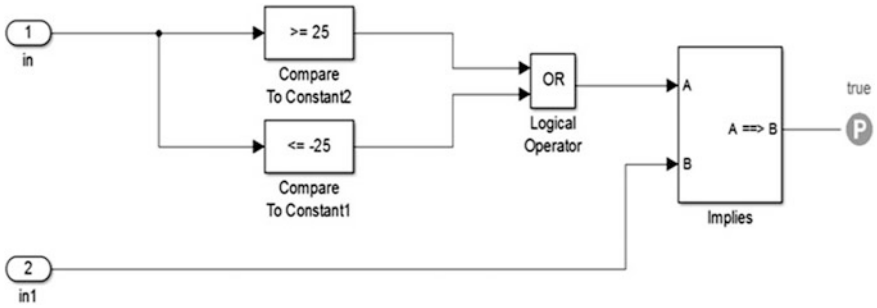

Fig. 7.2 Assertions added to the autodestruct system

examples can be simplified in some form or the other and represented with the logic, comparators, timed blocks in combination with the (P) block to prove a specific assertion.

7.3 NuSMV

NuSMV is a software tool for the formal verification of finite state systems developed jointly by Fondazione Bruno Kessler FBK-IRST and Carnegie Mellon University [11]. It is a reimplementation and a reengineering of the Symbolic Model Verifier (SMV) model checker developed by McMillan at Carnegie Mellon University during his Ph.D. [14]. NuSMV checks finite state machines or systems against specifications in the temporal logic. The language of NuSMV allows the description of finite state systems. It supports both synchronous and completely asynchronous behavior. The purpose of the language is to describe the transition relation of a finite Kripke structure.

The above problem is implemented in the NuSMV language as shown below. LTLSPEC G identifies the assertion as in the SLDV case.

```
MODULE main
VAR
Theta : -30 .. 30;
auto_dest : boolean;
```

```
   ASSIGN

   auto_dest := case
   (Theta) >= 25 : TRUE;
   TRUE : FALSE;
   esac;

   LTLSPEC G (((Theta >= 25) | (Theta <= -25)) ->
   (auto_dest = TRUE));

   -- specification G ((Theta >= 25 | Theta <= -25) ->
auto_dest = TRUE) is false
   -- as demonstrated by the following execution se-
quence
   Trace Description: LTL Counterexample
   Trace Type: Counterexample
    -> State: 1.1 <-
      Theta = -15
      auto_dest = FALSE
    -> State: 1.2 <-
      Theta = -28
    -- Loop starts here
```

NuSMV and SLDV, both behave the same way by generating test cases as counter examples that falsify the assertion if the absolute is missing in the design. This is the power of formal methods that can easily validate control systems, and safety properties in aerospace applications. The rest of the paper brings out examples of the use of NuSMV and SLDV in proving the correctness of the system.

7.4 Autopilot Mode Transition

An autopilot is a system that ensures the aircraft flies a specific course reducing the pilot's workload on long flights. The innermost component of an autopilot is a mode transition logic that changes the autopilot behavior based on pilot inputs or internal software flags. A typical mode transition design and test method were described earlier as an example of using assertions in validation of the design [15]. A simplified model of the mode transition is available on the MathWorks file exchange that models only the vertical modes of an autopilot [16].

Typical vertical modes of an autopilot are the Pitch Attitude Hold (PAH) mode, the Altitude Hold (ALT HOLD) mode, the Speed Hold (SPD HOLD) mode, the Vertical Speed Hold (VS HOLD) mode, and the Altitude Select mode (ALT SEL) as shown in Fig. 7.3. The PAH mode maintains the aircraft's pitch angle fixed

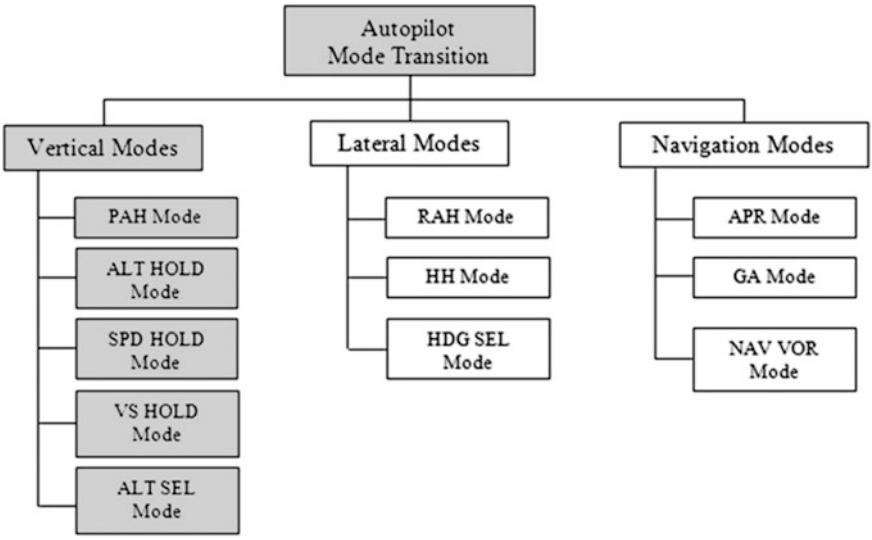

Fig. 7.3 The autopilot modes

constant at the value when the mode is selected. The ALT HOLD mode holds the aircraft's height or altitude at a value when the mode is selected. In this mode, the pitch angle is free to change. The SPD HOLD mode ensures that the aircraft maintains its speed at a value when it is selected but the pitch angle and altitude can change. Similarly, VS mode holds a constant climb rate. The ALT SEL mode is a special mode where the aircraft climbs toward a selected altitude and on reaching it transfers to ALT HOLD mode automatically. All these modes and the transitions are based on pilot selection or internal triggers. These are described in the design as a set of two tables—the mode transition table and the condition table.

Figure 7.4 shows the transition table as given in the example [16]. There are three distinct independent major modes—Vertical, AP, and Alt Sel. Vertical has 6

States	Sl. No.	Modes	Buttons				Software Triggers			
			01	02	03	04	05	06	07	08
			AP	SPD	VS	ALT	ALTS	ALTCAP	ALTCPDN	APFAIL
Vertical	01	DIS(Vertical)	02	00	00	00	00	00	00	00
	02	PAH	01	03	04	05	00	06	00	01
	03	SPD HOLD	01	02	04	05	00	06	00	01
	04	VS	01	03	02	05	00	06	00	01
	05	ALT HOLD	01	03	04	02	00	00	00	01
	06	ALTS CAP	01	00	00	05	00	00	05	01
AP	01	AP ON	02	00	00	00	00	00	00	02
	02	AP OFF	01	00	00	00	00	00	00	00
ALT SEL	01	ALTS OFF	00	00	00	00	02	00	00	00
	02	ALTS ARM	01	00	00	01	01	03	00	01
	03	ALTSEL CAP	01	00	00	01	00	00	01	01

Fig. 7.4 Transition table

States	Sl. No.	Modes	Buttons				Software Triggers			
			01	02	03	04	05	06	07	08
			AP	SPD	VS	ALT	ALTS	ALTCAP	ALTCPDN	APFAIL
Vertical	01	DIS(Vertical)	01	00	00	00	00	00	00	00
	02	PAH	02	03	04	05	00	00	00	02
	03	SPD HOLD	02	00	04	05	00	00	00	02
	04	VS	02	03	00	05	00	00	00	02
	05	ALT HOLD	02	03	04	00	00	00	00	02
	06	ALTS CAP	02	00	00	05	00	00	00	02
AP	01	AP ON	00	00	00	00	00	00	00	00
	02	AP OFF	01	00	00	00	00	00	00	00
ALT SEL	01	ALTS OFF	00	00	00	00	00	00	00	00
	02	ALTS ARM	02	00	00	05	00	00	00	02
	03	ALTSEL CAP	02	00	00	05	00	00	00	02

Fig. 7.5 Condition table

submodes which starts with disconnect (DIS) and the 5 other modes defined earlier. The Vertical major mode can exist in only one submode in a frame. At start (when switched on), the autopilot is in Vertical → DIS, AP → AP OFF, and ALTSEL → ALTSOFF. The (→) indicates the major → submode pair. The autopilot transits from these modes based on the table. The number in the 4th column indicates the modes that it can transit to. There are 4 buttons that the pilot can press—AP, SPD, VS, and ALT. There are 4 software triggers that the system can generate—ALTS (altitude select), ALTCAP (altitude capture), ALTCAPDN (Altitude Capture Done), and APFAIL (Autopilot has failed).

The table is read as follows. These are the system behavior requirements.

The autopilot can transit from Vertical → DIS (01) (Sl No 1) to 02 (read in column under AP, same row) if AP button is triggered.

The autopilot can transit from AP → AP OFF to AP → AP ON (01) if AP button is triggered.

The autopilot can transit from Vertical → SPD HOLD (03) to Vertical → VS (04) if the button VS is triggered.

Normally, the transition happens if certain conditions are true. This is captured in the condition table (Fig. 7.5). Thus, the transition table is read in conjunction with the condition table. The requirements are actually modified in conjunction with the condition table as

The autopilot transits from Vertical → DIS (01) (Sl No 1) to 02 (read in column under AP) if AP button is triggered AND condition number 01 is true.

The autopilot transits from AP → AP OFF to AP → AP ON (01) if AP button is triggered AND condition number 01 is true.

The autopilot transits from Vertical → SPD HOLD (03) to Vertical → VS (04) if the button VS is triggered AND condition number 01 is true.

The condition number 01 could be a set of parameters that set it to True indicating that the autopilot can come on. A typical set of condition would be C01 = (100 kts < Speed < 300 kts) AND (−5 deg < Pitch Angle Theta < 15 deg) AND (−20 deg < Roll Angle < 20 deg). Condition 04 could be a bound on the current vertical speed.

7.5 Automated Validation

A set of Matlab scripts are available on the MathWorks file exchange Web site. These help in validating a mode transition design given in the above form [16, 17]. This method has been tried with different mode transitions like a Radar application or the famous Therac-25 modes and found to work well. One of the scripts generates an English text output that defines the modes. A typical output generated automatically given the design looks like below. This can be read and verified for correctness.

(1) If in State DIS (Vertical) AND Trigger AP occurs THEN transition to PAH if condition C1 Is TRUE
(2) If in State PAH AND Trigger AP occurs THEN transition to DIS (Vertical) if condition C2 Is TRUE

Two additional scripts generate a Matlab code and a NuSMV code from the same two tables. These can be used for the validation of the mode transition using assertions in SLDV or in NuSMV, respectively. The assertions are defined based on the behavioral truth about the system. Four assertions are identified.

1. If Vertical mode = ALTCAP then ALTSEL mode = ALTSEL CAP
2. If Vertical mode = ALT HOLD then ALTSEL mode = ALTS OFF
3. If Vertical mode = DIS then AP = AP OFF
4. If Vertical mode = DIS then ALTSEL = ALTS OFF

The SLDV assertion of (2) and (3) above is shown in Fig. 7.6. The first input "in" is the Vertical mode. If this is equal to 5, i.e., ALT HOLD, it "implies that" the third input "in2", i.e., ALT SEL mode is equal to 1 (ALTS OFF). Similarly, if the second input "in1" AP is equal to 2, i.e., AP OFF "implies that" Vertical mode equals 1 (DIS). The ease of putting in assertions is very clear from the figure.

A similar assertion in NuSMV would look like this

$$\text{LTLSPEC G } ((\text{Vm} = \text{ALT\_HOLD}) \rightarrow (\text{ASm} = \text{ALTSOFF}))$$
$$\text{LTLSPEC G } ((\text{Vm} = \text{DIS}) \rightarrow (\text{APm} = \text{APOFF}))$$

Executing the SLDV and the NuSMV indicates that the assertions are true and therefore we are confident that the design is good. This does not show the capability of NuSMV or SLDV to find design errors. A set of 9 errors or mutation were

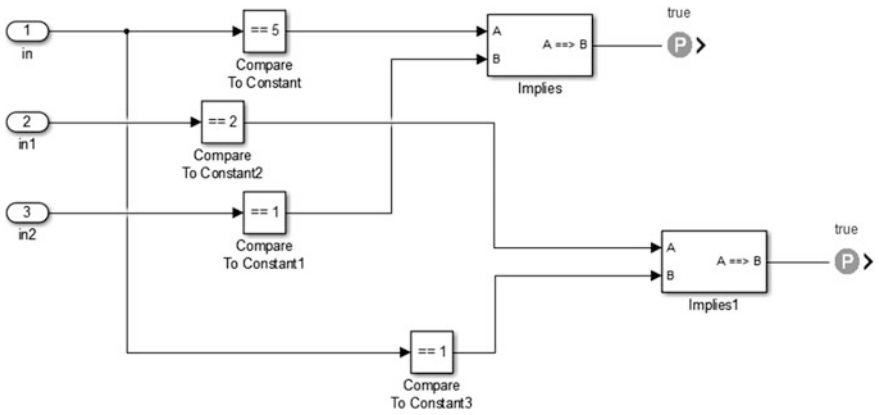

Fig. 7.6 SLDV assertions for (*2*) and (*3*)

introduced into the Simulink model and the NuSMV to identify if the tools bring out the error. Eight mutants are caught by the formal methods. One mutant is a "dud" as the later blocks correct this error, and therefore the model behaves correctly even with the mutation present. These mutants are available in file exchange [16].

7.6 Formal Method Versus Random Tests

There is always an argument when one discusses formal methods—how does it compare against testing. This aspect is addressed here. NuSMV model cannot be tested with a test vector whereas the Simulink models can be tested. Simulink has both a simulation platform and a formal methods platform. The Autopilot mode transition was modeled in Simulink and the assertions added as above. Very specific errors were injected into the model to create 9 mutant files. The mutant files are the actual model copy with a very specific error introduced deliberately into it. Random test case vectors are injected into the test harness and the test stops as soon as the assertion is falsified. The same exercise is repeated with the SLDV and the time takes for falsification of the assertion is captured. The test and SLDV proof are repeated 5 times, and a mean and standard deviation are computed from the trials and tabulated in Table 7.1.

The overall mean times for SLDV and random tests are similar at about 20 s. The mean times for individual mutants are, however, very different. The overall minimum time for SLDV is 12.2 s, and for random this is 1.31 s. The overall maximum time of SLDV is 23.6 s, and for random 91.05 s. There is a large disparity in the time to falsify the assertion in the random tests. This is also seen in the mean standard deviation across all tests being small for SLDV at 1.3 s, and the

Table 7.1 Summary of mutants and test timing

S. No.	Mutant	SLDV mean (s)	SLDV std (s)	Random mean (s)	Random std (s)
1	DIS replaced by code 2 instead of 1	16.8	0.45	1.31	0.79
2	VS_HOLD replaced by code 1 instead of 4	21.4	0.55	64.61	15.17
3	APOFF made APON in AP mode	21.4	0.55	4.56	2.74
4	ALTS trigger connected directly without checking for ALTHOLD	23.6	2.30	7.24	6.52
5	Condition C2 for ALTCAP trigger removed	12.2	0.45	5.17	5.46
6	ATLCAP connected directly without checking for ATLSARM state	18.4	2.41	25.74	12.11
7	Condition C2 changed to APOFF true instead of APON true	21.6	5.18	1.35	1.08
8	ALTCPDN connected directly without checking for ALTCAP state	–	–	–	–
9	Switching transition conditions for ALTSARM and ALTSCAP in ALTSEL mode	19.2	0.45	91.05	94.76

random tests have 17.3 s. The large standard deviation for the last mutant at 94.7 s indicates that random tests can take a very long time before they find an error. This is also seen in an earlier study on test cases for control system models [18]. SLDV finds the deliberately injected errors in the models very easily. Can it find errors in safety critical flight code and models?

7.7 Up Down Counter

Several examples of SLDV finding bugs found during testing in flight control models are available [1]. These errors were found during the extensive software test activity. SLDV finds all these errors very easily. A very recent error found in a flight code for an Up/Down counter is described here as an example. Up/Down counters are blocks that count up with a fixed amount if there is an error in the signal, and count down at a slower rate if the error does not exist. Thus, these blocks can penalize a more frequent error than an intermittent fault that could be attributed to noise. The schematic of an Up/Down counter is shown in Fig. 7.7.

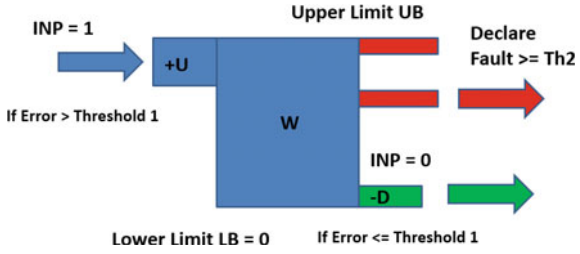

Fig. 7.7 Schematic of up/down counter

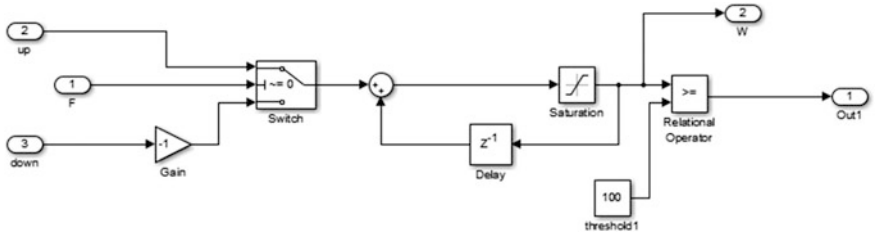

Fig. 7.8 Model of the up/down counter

The input to the counter is INP. This is set to true or 1 when the error (what we are monitoring) is greater than a threshold "Threshold 1". INP equal to adds the up count +U to the counter W. W is initially 0. INP equals 0 adds "−D" the down count to the count W. The lower limit for W is LB which is normally 0. The upper limit for W is UB. The value of W is clipped at UB. If W is greater than or equal to a trigger threshold "Th2", the output of the Up/Down counter is set to True.

In an aircraft flight control law, this was modeled in Simulink and coded in C language for implementation on board. The variable W was defined as an unsigned integer. During testing, a very specific combination led to W becoming negative which caused a wraparound, and it triggered a fault even though the error was less than threshold. This was tested using SLDV to see if it could catch the error. In the Simulink model, the data type for the adder block internal variable was defined as UINT16. The model is shown in Fig. 7.8.

There are two behavior properties that define the truth about the system or are the assertion

1. If the output is True previously and the input is True in the current frame implies the output is True in the current frame
2. If the output is False previously and the Input is False in the current frame implies the output is False in the current frame

These are modeled in SLDV as shown in Fig. 7.9. If Not first frame AND previous output (out1) is NOT True AND Input (F2) is NOT True AND U > D Implies Output is Not True. Similar assertions are made for the other properties. SLDV brings out a counter example to prove the assertion wrong replicating the condition observed during the test activity.

SLDV has new feature in the newer version of Matlab where a C code can be compiled into the Simulink environment. This compiled code is made compatible for SLDV analysis using options provided in the software. The actual code can be tested against the assertion. This is done for the Up/Down counter C code, and this error is easily caught by the formal methods analysis. The ease with which an assertion using the blocks can find errors in the C code is a very useful workflow. Legacy code from earlier projects, code libraries can now be validated in the Simulink environment.

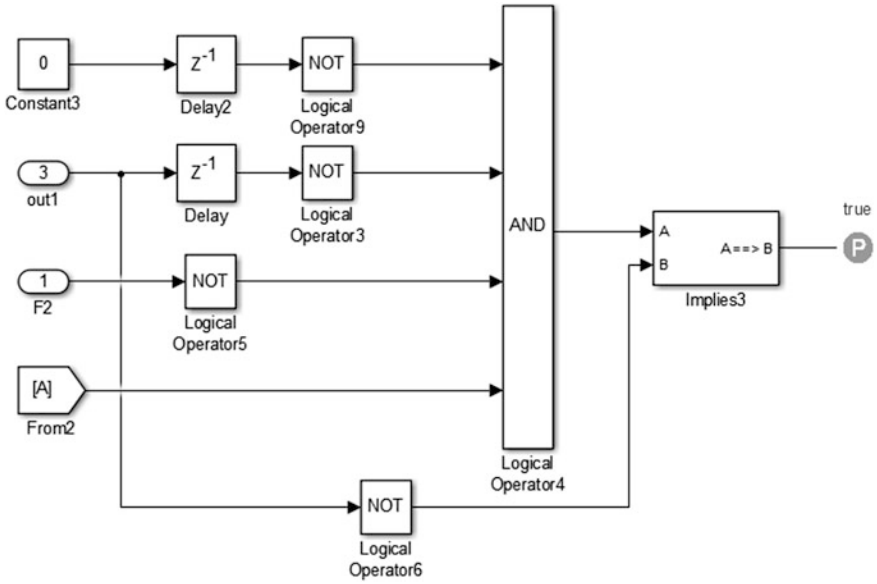

Fig. 7.9 Assertions of the up/down counter

7.8 Conclusion

SLDV and NuSMV, two model checking tools can be easily deployed in project and find errors in design very easily. The only constraint is that the requirements should be clear. It is very important that the system engineers clearly define the intent of their design as requirements rather than the design itself. It is very important that the requirements specify the "what" rather than "how". The use of formal methods drives home this discipline in defining requirements. The assertions are easily implemented by novice engineers with help from systems experts specifying the correct intent. Automation is brought in very easily to write NuSMV code from specification. It would be worthwhile, a exercise to specify requirements and automatically convert them into NuSMV or SLDV assertions. This will reduce the burden of the engineers trying to learn a new language or modeling paradigm.

The title uses the term "argue". The use of formal methods in a project brings out the argument between the engineers that is very important. We need to argue technically on the correctness of the design. The message and argument "is this what you want?" going all the way to the system designers by using these methods will lead to a better system design and safer world.

References

1. Jeppu Y (2016) Testing of safety critical control systems. http://in.mathworks.com/matlabcentral/fileexchange/39047-testing-of-safety-critical-control-systems. Accessed 15 July 2016
2. Jeppu N, Jeppu Y, Murthy N (2015) Arguing formally about flight control laws. Paper presented at international conference on industrial instrumentation and control (ICIC), pp 378–383, 28–30 May 2015. doi:10.1109/IIC.2015.7150771
3. RTCA, Inc (2011) Formal methods supplement to DO-178C and DO-278A. RTCA DO-333, USA
4. RTCA, Inc (2011) Software considerations in airborne systems and equipment certification. RTCA DO-178C, USA
5. Darren C, Steven PM (2014) Formal methods case studies for DO-333. NASA/CR–2014-218244
6. Laurent O (2010) Using formal methods and testability concepts in the avionics systems validation and verification (V&V) process. In: Third international conference on software testing, verification and validation, Paris, pp 1–10. doi:10.1109/ICST.2010.38
7. Moy Y, Ledinot E, Delseny H, Wiels V, Monate B (2013) Testing or formal verification: DO-178C alternatives and industrial experience. In: IEEE Software, vol 30, no 3, pp 50–57, May–June 2013. doi:10.1109/MS.2013.43
8. Stidolph DC, Whitehead J (2003) Managerial issues for the consideration and use of formal methods. In: Araki K, Gnesi S, Mandrioli D (eds) FME 2003: formal methods: international symposium of formal methods Europe, pp 170–186. Springer, Berlin, Heidelberg. doi:10.1007/978-3-540-45236-2_11
9. Davis JA et al (2013) Study on the barriers to the industrial adoption of formal methods. In: Pecheur C, Dierkes M (eds) Formal methods for industrial critical systems. In: 18th international workshop, FMICS 2013. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-41010-9_5
10. MathWorks. Simulink design verifier. http://in.mathworks.com/products/sldesignverifier/. Accessed 15 July 2016
11. NuSMV. http://nusmv.fbk.eu. Accessed 15 July 2016
12. Jeppu N (2014) Exploring design verifier. http://in.mathworks.com/matlabcentral/fileexchange/48858-exploring-design-verifier. Accessed 15 July 2016
13. Baier C, Katoen J-P (2008) Principles of model checking. MIT Press
14. McMillan KL (1992) Symbol model checking an approach to the state explosion problem. Dissertation, Carnegie Mellon University. http://www.kenmcmil.com/pubs/thesis.pdf
15. Rao M et al (2015) A methodology to design a validated mode transition logic. In: Vijay V et al (eds) Systems thinking approach for social problems. Lecture notes in electrical engineering, vol 327. doi:10.1007/978-81-322-2141-8_24
16. Jeppu N (2014) Exploring design verifier—2. http://in.mathworks.com/matlabcentral/fileexchange/51567-exploring-simulink-design-verifier-2. Accessed 15 July 2016
17. Jeppu N (2014) Exploring design verifier—3. http://in.mathworks.com/matlabcentral/fileexchange/54945-exploring-simulink-design-verifier-03. Accessed 15 July 2016
18. Jeppu Y et al (2014) Generating test cases with 100-percent requirements coverage using design of experiments. J Aerosp Inf Syst (Special Section on Software Challenges in Aerospace) 11:632–648. doi:10.2514/1.I010159