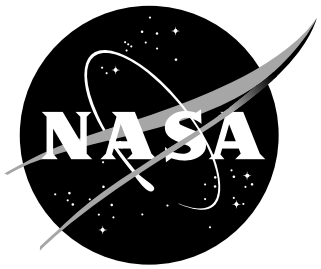
NASA/TM–2019–220374



Evaluation of the FRET and CoCoSim tools on the Ten Lockheed Martin Cyber-Physical Challenge Problems

Anastasia Mavridou
SGT, NASA Ames Research Center, Moffett Field, CA 94035, USA

Hamza Bourbouh
SGT, NASA Ames Research Center, Moffett Field, CA 94035, USA

Pierre-Loic Garoche
Onera, FR and SGT, NASA Ames Research Center, Moffett Field, CA 94035, USA

Mohammad Hejase
NASA Ames Research Center, Moffett Field, CA 94035, USA

October 2019

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

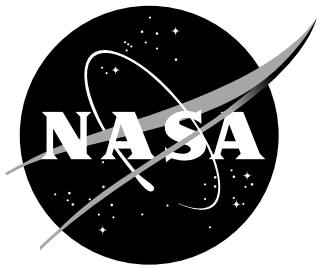
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at ***http://www.sti.nasa.gov***
- E-mail your question to help@sti.nasa.gov
- Phone the NASA STI Help Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681–2199

NASA/TM–2019–220374



Evaluation of the FRET and CoCoSim tools on the Ten Lockheed Martin Cyber-Physical Challenge Problems

Anastasia Mavridou
SGT, NASA Ames Research Center, Moffett Field, CA 94035, USA

Hamza Bourbouh
SGT, NASA Ames Research Center, Moffett Field, CA 94035, USA

Pierre-Loic Garoche
Onera, FR and SGT, NASA Ames Research Center, Moffett Field, CA 94035, USA

Mohammad Hejase
NASA Ames Research Center, Moffett Field, CA 94035, USA

National Aeronautics and
Space Administration

Ames Research Center, Moffett Field, CA 94035
Onera, The French Aerospace Lab, FR

October 2019

Acknowledgments

The authors would like to thank Chris Elliot and the Lockheed Martin company for sharing these challenge problems. This work also benefited from the commitment of interns at NASA, who worked on use cases, Julian Rhein and Tanja de Jong. Last, we have had numerous and fruitful interactions regarding this work with Cesare Tinelli and Daniel Larraz, from University of Iowa, as well as Mohammad Hejase and Temesghen Kahsai from NASA. This work was supported by the NASA Aeronautics Mission Directorate System-Wide Safety program's SAAFE project and the French ANR JCJC project FEANICSES.

This report is available in electronic form at
http://URL

Executive Summary

We connect the Formal Requirements Elicitation Tool (FRET) with the CoCoSim verification tool to enable the automated analysis of hierarchical dataflow models against requirements written in a restricted English language. Our framework supports 1) automatic extraction of Simulink model information and association of high-level requirements with target model signals and components; 2) translation of temporal logic formulas into synchronous dataflow CoCoSpec specifications as well as Simulink monitors; and 3) interpretation of counterexamples produced by the analysis both at the requirement and model level. We report on the lessons learned from the application of our approach to the Lockheed Martin Cyber-Physical, aerospace-inspired challenge problems. For the analysis, we used the Kind2, Zustre, and Simulink Design Verifier (SLDV) tools.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.0.1 Machine performance and tools versions

All the experimentation has been carried out on a MacBook Pro machine with 3.1 GHz (intel Core i7) processor and 16 GB Memory. The Malab/Simulink version used is 2019b, and Kind2 version v1.1.0.

Chapter 2

Summary of Block Types in LMCPS

Table 2.1 summarizes the number of blocks used in each challenge problem, as well as the block types used. Some of the blocks make verification difficult due to:

- Transcendental Functions: Such as the trigonometric functions. Challenge 7 (AP) uses *cos*, *sin*, *atan2*, *asin*. Challenge 9 (EUL) uses *sin* and *cos*.
- Nonlinearities and Discontinuous Math Such as: *Abs*, *MinMax*, *Saturation*, *Switch*. Inverse of Matrix (3 by 3 and 5 by 5 Matrices) are used in Challenge 6 (EB) and 7 (AP).
- Vectors and Matrices: All LMCPS challenge problems manipulate signals with multi-dimensions. CoCoSim inlines these signals into scalars, which increases the number of variables and may make verification harder depending on the operations applied on these signals. Challenges 6 (EB) and 7(AP) use the inverse of matrices, which is abstracted in Lustre. Additionally, challenge 7 (AP) manipulates Quaternions with some advanced *Quaternion* operations (e.g. *Quaternion Modulus*, *Quaternion Norm* and *Quaternion Normalize*).
- States: Blocks such as *Delay* and *Unit Delay* are used in the majority of LMCPS. They are used to access memories of signals up to n steps back (n=1 for UnitDelay).

Table 2.1: Summary of Block Types and number of blocks used in LMCPS. Blocks in bold are the ones that mainly affect the complexity of a model.
Additional information for each block can be found in `https://www.mathworks.com/help/simulink/block-libraries.html`

Model Name	# Blocks	Block Types used
0_triplex	479	'**Abs**', 'Action Port', 'Constant', '**Delay**', 'Demux', 'From', 'Goto' '**If**', 'Inport', '**Logic**', '**Merge**', 'Mux', 'Outport', '**Product**', '**Relational Operator**', '**Selector**', 'Signal Conversion', 'Subsystem', '**Sum**', '**Switch**', 'Terminator'
1_fsm	279	'Action Port', 'Constant', 'Demux', 'From', 'Goto', '**If**', 'Inport', '**Logic**', '**Merge**', 'Mux', 'Outport', '**Relational Operator**', '**Signal Conversion**', 'Subsystem', '**Switch**', '**Unit Delay**'
2_tustin	45	'DataType Duplicate', 'Data Type Propagation', 'From', 'Gain', 'Goto', 'Inport', 'Outport', '**Product**', '**Relational Operator**', '**Saturation Dynamic**', 'Subsystem', '**Sum**', '**Switch**', '**Unit Delay**'
3_regulators	271	'**BusCreator**', '**BusSelector**', 'Constant', 'From', 'Gain', 'Goto', 'Inport', '**Lookup_nD**', '**Math**', '**Memory**', 'Outport', '**Product**' '**Relational Operator**', '**Saturate**', '**Saturation Dynamic**', '**Signal Conversion**', 'SubSystem', 'Sum', '**Switch**', 'Terminator', '**UnitDelay**'
4_nlguide	355	'ActionPort', 'Constant', 'Demux', 'Display', '**DotProduct**', 'From', '**Gain**', 'Goto', '**If**' , 'Inport', 'InportShadow', '**Logic**', '**Math**', '**Merge**', 'Mux', 'Outport', '**Product**', '**Relational Operator**', '**Selector**', '**Sqrt**', 'SubSystem', '**Sum**', 'Terminator'
5_nn	699	'ActionPort', 'Constant', 'Demux', '**Gain**', '**If**', 'Inport', '**Merge**', 'Mux', 'Outport', '**Product**', '**Saturate**', 'SubSystem', '**Sum**'
6_eb	75	'Constant', 'Display', 'Inport', '**Math**', 'Outport', '**Product**', '**Relational Operator**', '**Reshape**', '**Selector**', 'SubSystem', '**Sum**', '**Switch**'
7_autopilot	1357	'**Abs**', '**BusCreator**', '**BusSelector**', '**Concatenate**', 'Constant', '**Data Type Conversion**', 'Demux', 'Display', '**DotProduct**', '**Fcn**', 'From', '**Gain**', 'Goto', 'Ground', 'Inport', 'InportShadow', '**Logic**', '**Lookup_nD**', '**Math**', '**MinMax**', 'Mux', 'Outport', '**Product**', '**RateLimiter**', '**Relational Operator**', '**Reshape**', '**Rounding**', '**Saturate**', 'Scope', '**Selector**', '**Signum**', '**Sqrt**', 'SubSystem', 'Sum', '**Switch**', 'Terminator', '**Trigonometry**', '**UnitDelay**', '**CMBlock**', '**Create 3x3 Matrix**', '**Passive**', '**Quaternion Modulus**', '**Quaternion Norm**', '**Quaternion Normalize**', 'Rate Limiter Dynamic'
8_swim	141	'ActionPort', 'Constant', 'Display', '**Gain**', '**If**', 'Inport', '**Logic**', '**Merge**', 'Outport', '**Relational Operator**', '**Sqrt**', 'SubSystem', '**Sum**', '**UnitDelay**'
9_euler	97	'**Concatenate**', '**Fcn**', 'Inport', 'Mux', 'Outport', '**Product**', '**Reshape**', 'SubSystem', '**Trigonometry**', '**Create 3x3 Matrix**'

Chapter 3

Triplex Signal Monitor (TSM)

The TSM challenge problem involves the verification of a redundancy management system using quantum simulation techniques. The purpose of this system is to prevent errors from propagating past the input portion of an airborne application. The assumed platform configuration is a set of three computers which execute identical software applications, and which take the same set of input values from sensors on the aircraft. For each set of triplex inputs, the TSM must monitor failures (or imminent failures), report the failure status of the set to the necessary functions, and choose an acceptable signal for computational use. Tables 3.1 and 3.2 list the inputs and outputs of the TSM component, respectively.

Table 3.1: TSM Inputs

Input Scope	**Name**	**Type**	**Description**
Global	ia	Double	Signal to monitor on branch A
Global	ib	Double	Signal to monitor on branch B
Global	ic	Double	Signal to monitor on branch C
Global	Tlevel	Double	Threshold level leading to a miscompare between 2 signals
Global	PClimit	Integer	Persistence Count limit governing duration of miscompare prior to latched fault

Table 3.2: TSM Outputs

Output Scope	**Name**	**Type**	**Description**
Global	PC	Integer	Current Persistence Count in # of frames
Global	TC	Integer	Current Totalizer Count in # of frames
Global	FC	Integer	Latched Fault Code
Global	sel_val	Double	Selected value between the triplex signals

Each input is triply redundant and has its own failure state that is generally independent from the failure states of all other input sets. There are three failure states: 1) no-fail state, 2) single-fail state, and 3) dual-fail state. In the no-fail and single-fail cases, an error will manifest itself as a difference in the values of the signals of an input set. The difference is referred to as a *miscompare* until it is latched and reported to redundancy management as a *failure*. In particular, a miscompare is characterized by one branch differing with the other two branches by a unique trip level. The trip level defines the total allowable

discrepancy between the compared signals. Whenever the difference between compared signals is greater than the designated trip level (`Tlevel`), a miscompare shall be latched. The purpose of input failure persistence is to prevent false failure declarations of an input that may be temporarily mistracking for some unknown transitory problem. Input failure persistence or persistence count (`PC`) is therefore defined as the total number of iterations that signals are permitted to miscompare outside of their designated trip level before a failure is declared.

We define a `miscompare` in CoCoSpec as follows:

```
var C1:bool = abs(ia - ib) > Tlevel;
var C2:bool = abs(ib - ic) > Tlevel;
var C3:bool = abs(ia - ic) > Tlevel;
var miscompare : bool = (not C1 and C2 and C3)  or (C1 and not C2
   and C3) or (C1 and C2 and not C3);
```

We define in CoCoSpec a function that returns the absolute value of `x`, as follows:

```
node abs(x:real) returns (y:real);
let
  y = if x >= 0.0 then x else - x;
tel
```

If the signals miscompare for a single frame longer than the allocated persistence (`PCLimit`), an input failure is declared. The term *pending failure* or *failure in progress* is used to describe the period of time that a signal is out of tolerance but has not reached the persistence count limit. When the persistence limit has been reached, the monitor begins to operate without the failed input and the failure is called a *latched failure*.

We define a `failure in progress` in CoCoSpec as follows:

```
var  failure_in_progress : bool = miscompare  and prePC <= PCLimit
   and PC > 0;
var prePC: int = 0 -> pre PC;
```

We define a `latched failure` in CoCoSpec as follows:

```
var  failure_must_be_latched : bool = miscompare  and prePC  >
   PCLimit;
```

We know that `FC` is an Integer that represents the Latched Fault Code, as shown in Table 3.2. However, we did not have information regarding the values of `FC` during no-fail and single fail states. For this reason, we looked at the provided Simulink model. `FC` equals 0 at the no-fail state. Additionally, `FC` equals 1 when there is a single failure reported in branch `ic`, `FC` equals 2 when there is a single failure reported in branch `ib`, and finally, `FC` equals 4 when there is a single failure reported in branch `ia`.

```
var no_fail : bool = (FC =0);
var single_fail_reported : bool = (FC=1) or (FC=2) or (FC=4);
```

To select a value, the monitor either uses 1) the mid-value selection (MVS) algorithm for the no-fail case or 2) the good channel average for the single fail case. The mid-value is defined as the signal value bound by the remaining two signals. We define the `mid-value` in CoCoSpec as follows:

```
var mid_value : real =  if ((ia <= ib and ib <= ic)
                            or (ic <= ib and ib <= ia))
```

Table 3.3: FRET to Model Variables mapping for TSM

FRET name	**Model path**
ia	triplex_12B_PP/ia
ib	triplex_12B_PP/ib
ic	triplex_12B_PP/ic
Tlevel	triplex_12B_PP/Tlevel
PCLimit	triplex_12B_PP/PCLimit
PC	triplex_12B_PP/PC
TC	triplex_12B_PP/TC
FC	triplex_12B_PP/FC
set_val	triplex_12B_PP/sel_val

Table 3.4: TSM Verification Results with Kind2 (abbr. by K) and SLDV (abbr. by S).

Requirement	**K Result**	**K Time**	**S Result**	**S Time**
[TSM-001]	Valid	0.716 sec	Valid	26 sec
[TSM-002]	Valid	0.716 sec	Valid	26 sec
[TSM-003a]	Valid	0.716 sec	Valid	26 sec
[TSM-003b]	Valid	0.716 sec	Valid	26 sec
[TSM-003c]	Valid	0.716 sec	Valid	26 sec
[TSM-004]	Invalid	0.427 sec	Invalid	26 sec
Total running time	CoCoSim: 37.65s		SLDV: 26s	

```
        then ib
    else if ((ib <= ia and ia <= ic)
        or (ic <= ia and ia <= ib))
        then ia
    else ic;
```

Next, we provide 1) the TSM natural language requirements, 2) their FRETish form, and 3) the CoCoSpec equivalent code. The mapping between the FRET variables and the model Inputs and Outputs is shown in Table 3.3. The verification results are summarized in Table 3.4.

[TSM-001]

- **Natural language**:
  In the no-fail state, a miscompare, which shall be characterized by one branch differing with the other two branches by a unique trip level that lasts for more than the persistence limit, shall be reported to failure management as a failure.
- **FRET input**:
  TriplexSignalMonitor shall always satisfy `((pre_no_fail & failure_must_be_latched) => single_fail_reported)`,
- **CoCoSpec**:

```
guarantee "TSM-001" (pre_no_fail and failure_must_be_latched)
   => single_fail_reported;
```

  where

```
var pre_no_fail : bool = (true -> pre no_fail);
```

- **Verification result**: Valid by Kind2 and SLDV.

[**TSM-002**]

- **Natural language**: In the no-fail state, the mid-value shall be the selected value. Note: a first failure in progress will not affect the method for determining the selected value.
- **FRET input**: TriplexSignalMonitor shall always satisfy (`no_fail => (set_val = mid_value)`),
- **CoCoSpec**:

```
guarantee "TSM-002" no_fail => set_val = mid_value;
```

- **Verification result**: Valid by Kind2 and SLDV.

[**TSM-003**]

- **Natural language**: In the single fail state, a good channel average of the remaining two good branches shall be used to determine the selected value.
- **FRET input**: We decomposed this requirement into three requirements:
  - TriplexSignalMonitor shall always satisfy `FC = 1 => (set_val = 0.5 * (ia +ib))`
  - TriplexSignalMonitor shall always satisfy `FC = 2 => (set_val = 0.5 * (ia +ic))`
  - TriplexSignalMonitor shall always satisfy `FC = 4 => (set_val = 0.5 * (ib +ic))`
- **CoCoSpec formalizations**:

```
guarantee "TSM-003a" FC = 1 => set_val = 0.5 * ( ia + ib );
guarantee "TSM-003b" FC = 2 => set_val = 0.5 * ( ia + ic );
guarantee "TSM-003c" FC = 4 => set_val = 0.5 * ( ib + ic );
```

- **Verification result**: Valid, Valid, Valid by Kind2 and SLDV.

[**TSM-004**]

- **Natural language**: If a second failure is in progress, the selected value shall remain unchanged from the previous selected value.
- **FRET input**: TriplexSignalMonitor shall always satisfy (`single_fail_reported & failure_in_progress => set_val = pre_set_val`),
- **CoCoSpec formalization**:

```
guarantee "TSM-004" single_fail_reported and failure_in_progress
    => set_val = pre_set_val;
```

where

```
var pre_set_val : real = (0.0 -> pre set_val);
```

- **Verification result**: Invalid by Kind2.

  Table 3.5 illustrates a counterexample generated by Kind2, which violates requirement `TSM-004`. At the first step of the execution (`T=0`) there is a miscompare between `ia`, `ib` and `ic`. `PC` is incremented by one, as the error has not been latched yet (`FC=0`) and the selected value is equal to the mid value of (`ia`, `ib`, `ic`) which in this case is `ib` since `ia < ib < ic`. At the second step, a second miscompare happens and `PC` is again incremented by one. The value of `FC` is based on the previous value of `PC`. Since `PC` is not greater than `PCLimit`, TSM remains in the no-fail state (`FC=0`) and the selected value is equal to the mid-value of (`ia`, `ib`, `ic`). At the third execution step (`T=2`), the previous value of PC exceeds `PCLimit`. Thus, a single-fail state is latched (`FC=1`). In this counterexample `abs(ic-ia) > TLevel` and `abs(ic-ib)> Tlevel`, therefore `ic` is considered as the faulty branch and the selected value is equal to the average of the remaining signals `ia` and `ib`. In the final step (`T=3`), another miscompare happens between the remaining signals `ia` and `ib`. Thus, `PC` is incremented by one and the selected value becomes equal to the average of (`ia` and `ib`). This is exactly the case when a second failure is in progress, but the selected value did not remain unchanged which violates the requirement.

Table 3.5: Counter example of requirement [TSM-004]

Inputs	T = 0	T = 1	T = 2	T = 3
ia	-4730	-2	-18043	-19126
ib	-1	40337	-17467	-16387
ic	5616	-17754	-12223	-14919
Tlevel	5616	17752	5243	2738
PCLimit	1	1	1	1
Outputs				
PC	1	2	0	1
FC	0	0	1	1
selected_value	-1	-2	-17755	-17756

Fig. 3.1 shows the part of the TSM Simulink model that computes the selected value (`sel_val`) output. The computation of `sel_val` can be summarized as follows:

$$sel\_val = \begin{cases} mid\_value(ia, ib, ic) & \text{if } FC = 0 \\ average(ia, ib) & \text{if } FC = 1 \\ average(ia, ic) & \text{if } FC = 2 \\ average(ib, ic) & \text{if } FC = 4 \end{cases} \tag{3.1}$$

The model looks incomplete; it does not consider the *second failure is in progress* condition for the generation of `sel_val`. In the case of a single failure (`FC=1 or FC=2 or FC=4`), unless the average of the remaining correct inputs remains unchanged between two consecutive execution steps, the selected value will not remain unchanged.

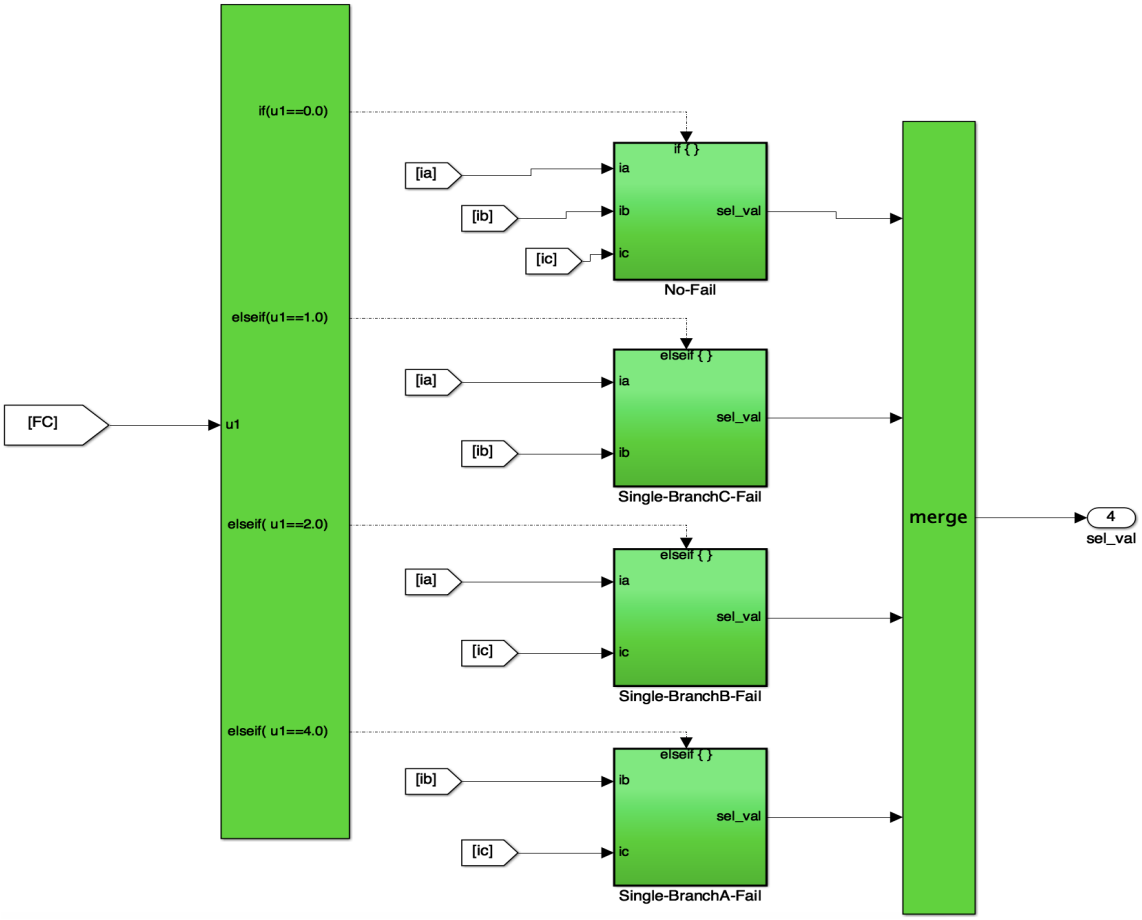


Figure 3.1: Simulink model that describes the computation of the selected value in TSM

Chapter 4

Finite State Machine (FSM)

This challenge problem represents an abstraction of detailed design requirements for an Advanced Autopilot System interacting with an independent sensor platform. The Finite State Machine (FSM) represents a Cyber-Physical system with two independent components executing in real time for the purpose of ensuring a safe automatic operation in the vicinity of hazardous obstacles. The autopilot system, tightly integrated with the vehicle flight control computer, is responsible for commanding a safety *maneuver* in the event of a hazard. The sensor is the reporting agent to the autopilot with observability of imminent danger. When the conditions are met to allow an autopilot operation (*supported* and *no failures*) and the pilot is not manually controlling the vehicle (*standby* is false), the autopilot is on until a maneuver is required in the event of a hazard (sensor is not *good*).

Tables 4.1 and 4.2 list the inputs and outputs of the FSM component, respectively. As shown in Table 4.1, the state of the Autopilot is represented by the `state` input of `Double` type. There are four different Autopilot states: `TRANSITION`, `NOMINAL`, `MANEUVER`, and `STANDBY`, which correspond to `state = 0.0`, `state = 1.0`, `state = 2.0`, and `state = 3.0`, respectively. Similarly, the state of the Sensor is represented by the `senstate` input of `Double` type. There are three different Sensor states: `NOMINAL`, `TRANSITION`, and `FAULT`, which correspond to `senstate = 0.0`, `senstate = 1.0`, and `senstate = 2.0`, respectively.

Next, we provide 1) the FSM natural language requirements, 2) their FRETish form, and 3) the CoCoSpec equivalent code. For this challenge problem, we created three CoCoSpec contracts, which we deployed in different hierarchical levels of the Simulink model. The mapping between the FRET variables and the model inputs and outputs is shown in Table 4.3.

The verification results are summarized in Table 4.4. Since in this challenge we have requirements attached to different components of the model, we use the monolithic setting of Kind2 (option –compositional set to false). This allows us to prove top level requirements against the implementation instead of the abstraction of the components by their requirements.

In the requirements, we use the following auxiliary variable that returns the previous value of the `limits` input:

```
var pre_limits : bool = false -> pre limits;
```

The following constants are used:

```
--autopilot states
const ap_transition_state : real = 0.0;
```

Table 4.1: FSM Inputs

Input Scope	Name	Type	Description
Global	standby	Boolean	True when the pilot is in control of the vehicle
Global	apfail	Boolean	Failure indication from an external source
Global	supported	Boolean	Indication of general health of system, must be True to enable operation
Global	limits	Boolean	External indication to sensor on faulty condition impeding safe operation
Autopilot	standby	Boolean	True when the pilot is in control of the vehicle
Autopilot	apfail	Boolean	Failure indication from an external source
Autopilot	supported	Boolean	Indication of general health of system, must be True to enable operation
Autopilot	state	Double	Autopilot State variable, Indication of prior value
Sensor	mode	Boolean	Indication from autopilot on current mode of operation
Sensor	request	Boolean	Secondary indication from autopilot on current mode of operation
Sensor	limits	Boolean	External indication to sensor on faulty condition impeding safe operation
Sensor	senstate	Double	Sensor State variable, Indication of prior value

Table 4.2: FSM Outputs

Output Scope	Name	Type	Description
Global	pullup	Boolean	True when the autopilot is in the MANEUVER state
Autopilot	MODE	Boolean	Autopilot internal moding discrete indicating not in transition state
Autopilot	REQUEST	Boolean	Autopilot internal moding discrete indicating transition to a nominal operation
Autopilot	PULL	Boolean	True when the autopilot is in the MANEUVER state
Autopilot	STATE	Integer	Autopilot State
Sensor	good	Boolean	False indicates an unsafe situation where a hazard may exist
Sensor	SENSTATE	Integer	Sensor State

```
const ap_nominal_state : real = 1.0;
const ap_maneuver_state : real = 2.0;
const ap_standby_state : real = 3.0;
--sensor states
const sen_nominal_state : real = 0.0;
const sen_transition_state : real = 1.0;
const sen_fault_state : real = 2.0;
```

[FSM-001]

- **Natural language**: Exceeding sensor limits shall latch an autopilot pullup when the pilot is not in control (not standby) and the system is supported without failures (not

Table 4.3: FRET to Model Variables mapping for FSM

FRET name	**Scope**	**Model path**
standby	Global	fsm_12B/standby
apfail	Global	fsm_12B/apfail
supported	Global	fsm_12B/supported
limits	Global	fsm_12B/limits
pullup	Global	fsm_12B/pullup
standby	Autopilot	fsm_12B/FiniteStateMachine/Manager/standby
apfail	Autopilot	fsm_12B/FiniteStateMachine/Manager/apfail
supported	Autopilot	fsm_12B/FiniteStateMachine/Manager/supported
state	Autopilot	fsm_12B/FiniteStateMachine/Manager/state
MODE	Autopilot	fsm_12B/FiniteStateMachine/Manager/MODE
REQUEST	Autopilot	fsm_12B/FiniteStateMachine/Manager/REQUEST
PULL	Autopilot	fsm_12B/FiniteStateMachine/Manager/PULL
STATE	Autopilot	fsm_12B/FiniteStateMachine/Manager/STATE
mode	Sensor	fsm_12B/FiniteStateMachine/Sen/mode
request	Sensor	fsm_12B/FiniteStateMachine/Sen/request
limits	Sensor	fsm_12B/FiniteStateMachine/Sen/limits
senstate	Sensor	fsm_12B/FiniteStateMachine/Sen/senstate
good	Sensor	fsm_12B/FiniteStateMachine/Sen/good
SENSTATE	Sensor	fsm_12BFiniteStateMachine/Sen/SENSTATE

apfail).

- **FRET input**: FSM shall always satisfy `(limits & ! standby & ! apfail & supported => pullup)`
- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "FSM-001" (limits and not standby and not apfail and
   supported) => pullup;
```

- **Verification result**: Invalid by both Kind2 and SLDV. We are not sure whether the invalid result is due to an incorrect natural language specification or due to an incorrect model. While analyzing requirement `[FSM-001]` we got the counterexample shown in Table 4.5.

  After revisiting the requirement, we thought that potentially there is a time step difference between `limits = true` and the activation of `pullup`. Thus, we updated the requirement as follows:
  if autopilot & pre_autopilot & pre_limits FSM shall immediately satisfy pullup
  FRET generated the following CoCoSpec code:

```
var autopilot: bool = (not standby) and supported and (not
   apfail);
var pre_autopilot: bool = false -> pre autopilot;
var pre_limits: bool = = false -> pre limits;
```

Table 4.4: FSM Verification Results with Kind2 (abbr. by K) and SLDV (abbr. by S)

Requirement	**K Result**	**K Time**	**S Result**	**S Time**
[FSM-001v1]	Invalid	0.254 sec	Invalid	25 sec
[FSM-001v2]	Invalid	0.465 sec	Invalid	38 sec
[FSM-001v3]	true up to 28 steps	timeout (2h)	Valid	94 sec
[FSM-002]	Valid	0.252 sec	Valid	94 sec
[FSM-003]	Invalid	0.191 sec	Invalid	21 sec
[FSM-003v2]	Valid	0.252 sec	Valid	94 sec
[FSM-004]	Invalid	0.191 sec	Invalid	32 sec
[FSM-004v2]	Valid	0.252 sec	Valid	94 sec
[FSM-005]	Valid	0.252 sec	Valid	94 sec
[FSM-006]	Valid	0.252 sec	Valid	94 sec
[FSM-007]	Invalid	0.135 sec	Invalid	35 sec
[FSM-007v2]	Valid	0.252 sec	Valid	94 sec
[FSM-008v1]	Invalid	0.165 sec	Invalid	28 sec
[FSM-008v2]	Valid	0.252 sec	Valid	94 sec
[FSM-009]	Valid	0.252 sec	Valid	94 sec
[FSM-010]	Valid	0.132 sec	Valid	94 sec
[FSM-011v1]	Invalid	0.110 sec	Invalid	18 sec
[FSM-011v2]	Valid	0.132 sec	Valid	95 sec
[FSM-012]	Valid	0.132 sec	Valid	12 sec
[FSM-013]	Valid	0.132 sec	Valid	95 sec
Total running time	CoCoSim: 141.09sec		SLDV: 96sec	

Table 4.5: Counter example of requirement [FSM-001]

Inputs	T = 0
standby	false
apfail	false
supported	true
limits	true
Outputs	
pullup	false

```
guarantee "FSM-001v2" ((((autopilot and pre_autopilot and
   pre_limits) and (false -> pre ( not (autopilot and
   pre_autopilot and pre_limits)))) or ((autopilot and
   pre_autopilot and pre_limits) and FTP)) => (pullup));
```

`[FSM-001v2]` was again shown to be invalid by Kind2. In particular, Kind2 returned the counterexample shown in Table 4.6, from which we can see that even though `pullup` was latched the first time `limits = true`, it was not latched the second time `limits = true`.

To understand better the behavior of the model, we simulated the model based on the counterexample (Table 4.6). Figure 4.1 illustrates a scenario when `limits` were equal to true multiple times during the autopilot operation, during which conditions `supported and not standby and not apfail` must be true. We found that `pullup` is latched only when `limits` is true in the previous step and has not been true for

Table 4.6: Counter example of requirement [FSM-001v2]

Inputs	T = 0	T = 1	T = 2	T = 3
standby	false	false	false	false
apfail	false	false	false	false
supported	true	true	true	true
limits	true	false	true	false
Outputs				
pullup	false	true	false	false

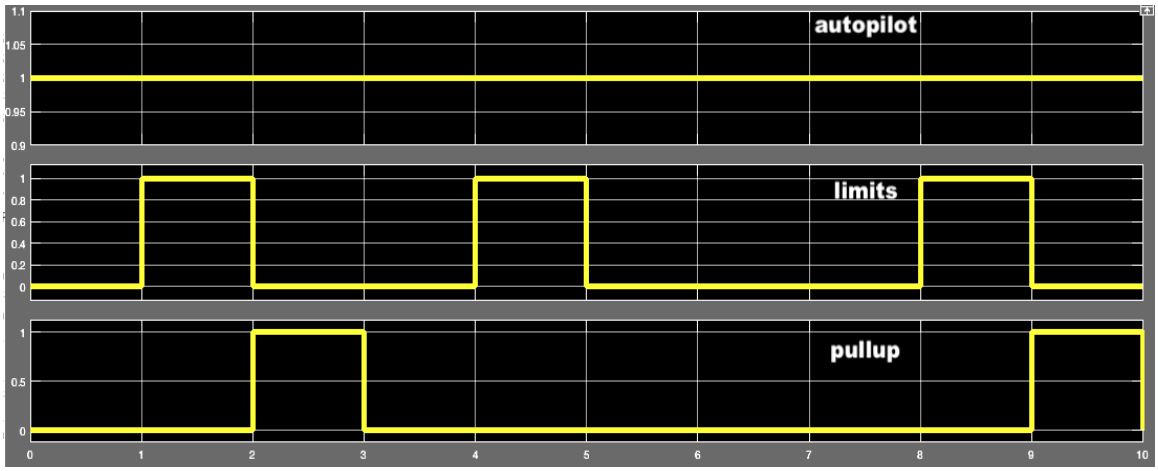


Figure 4.1: Simulation of requirement [FSM-001v2]

at least three steps before that (See example in 4.1 at t=9). Thus, we rewrote the requirement in FRET as follows:

if htlore3_autopilot & htlore3_notpreprelimits & pre_limits FSM shall immediately satisfy pullup
FRET generated the following the following CoCoSpec code:

```
var not_pre_pre_limits : bool = (false -> pre (false -> not
   pre_limits));
var htlore3__notpreprelimits =  HTlore(3, not_pre_pre_limits);
var htlore3_autopilot = HTlore(3, autopilot);
guarantee "FSM-001v3" ((((htlore3_autopilot and
   htlore3_notpreprelimits and pre_limits) and (false -> pre (
   not (htlore3_autopilot and htlore3_notpreprelimits and
   pre_limits)))) or ((htlore3_autopilot and
   htlore3_notpreprelimits and pre_limits) and FTP)) => (pullup)
   );
```

which is weaker than the requirement `[FSM-001v2]` since it additionally restricts `limits` to be false for at least three steps since the last time `limits = true`. Requirement `FSM-001v3` was shown to be valid by Kind2 and SLDV. We can clearly see that the initial natural language requirement is very different than `FSM-001v3`.

Table 4.7: Counter example of requirement [FSM-003]

Inputs	T = 0
standby	true
apfail	false
supported	true
good	true
state	0
Outputs	
STATE	3

[**FSM-002**]

- **Natural language**: The autopilot shall change states from TRANSITION to STANDBY when the pilot is in control (standby).
- **FRET input**: FSM shall always satisfy `(standby & state = ap_transition_state) => STATE = ap_standby_state`
- **Scope**: Autopilot
- **CoCoSpec formalization**:

```
guarantee "FSM-002" (standby and state = ap_transition_state) =>
    STATE = ap_standby_state;
```

- **Verification result**: Valid by Kind2 and SLDV.

[**FSM-003**]

- **Natural language**: The autopilot shall change states from TRANSITION to NOMINAL when the system is supported and sensor data is good.
- **FRET input**: FSM shall always satisfy `state = ap_transition_state & good & supported => STATE = ap_nominal_state`
- **Scope**: Autopilot
- **CoCoSpec formalization**:

```
guarantee "FSM-003" (state = ap_transition_state and good and
   supported) => STATE = ap_nominal_state;
```

- **Verification result**: Invalid due to incomplete natural language specification. The generated counterexample is shown in Table 4.7.

  We can see from the counterexample that, even though the conditions are satisfied, Autopilot changes state from `TRANSITION` to `STANDBY` (instead of `NOMINAL`). The conditions for the `TRANSITION` to `STANDBY` state change were defined previously in requirement `[FSM-002]`. From this, we can conclude that the requirements are not mutually exclusive. If we additionally constraint requirement `[FSM-003]` with condition `not standby`, we get the following FRET input and CoCoSpec formalization, which is proven valid.
  FSM shall always satisfy `state = ap_transition_state & good & supported & ! standby => STATE = ap_nominal_state`

Table 4.8: Counter example of requirement [FSM-004]

Inputs	T = 0
standby	true
apfail	false
supported	false
good	false
state	1
Outputs	
STATE	3

```
guarantee "FSM-003v2" (state = ap_transition_state and good and
    supported and not standby) => STATE = ap_nominal_state;
```

[FSM-004]

- **Natural language**: The autopilot shall change states from NOMINAL to MANEUVER when the sensor data is not good.
- **FRET input**: FSM shall always satisfy `(! good & state = ap_nominal_state) => STATE = ap_maneuver_state`
- **Scope**: Autopilot
- **CoCoSpec formalization**:

```
guarantee "FSM-004" (not good and state = ap_nominal_state) =>
    STATE = ap_maneuver_state;
```

- **Verification result**: Invalid due to incomplete natural language specification. The generated-by-Kind2 counterexample is shown in Table 4.8.

  We can see from the counterexample that, even though the conditions are satisfied, Autopilot changes state from `NOMINAL` to `STANDBY` (instead of `MANUEVER`). Similarly to requirement `[FSM-002]`, if we additionally constraint requirement `[FSM-004]` with condition `not standby`, we get the following FRET input and CoCoSpec formalization, which is proven valid.

  FSM shall always satisfy `state = ap_nominal_state & ! good & ! standby => STATE = ap_maneuver_state`

```
guarantee "FSM-004v2" (state = ap_nominal_state  and not good
    and not standby) => STATE = ap_maneuver_state;
```

[FSM-005]

- **Natural language**: The autopilot shall change states from NOMINAL to STANDBY when the pilot is in control (standby).
- **FRET input**: FSM shall always satisfy `(state = ap_nominal_state & standby) => STATE = ap_standby_state`
- **Scope**: Autopilot

Table 4.9: Counter example of requirement [FSM-007]

Inputs	T = 0
standby	true
apfail	false
supported	true
good	true
state	2
Outputs	
STATE	3
PULL	false

- **CoCoSpec formalization**:

```
guarantee "FSM-005" (state = ap_nominal_state and standby) =>
    STATE = ap_standby_state;
```

- **Verification result**: Valid by Kind2 and SLDV.

[**FSM-006**]

- **Natural language**: The autopilot shall change states from MANEUVER to STANDBY when the pilot is in control (standby) and sensor data is good.
- **FRET input**: FSM shall always satisfy `(state = ap_maneuver_state & standby & good) => STATE = ap_standby_state`
- **Scope**: Autopilot
- **CoCoSpec formalization**:

```
guarantee "FSM-006" (state = ap_maneuver_state and standby and
    good) => STATE = ap_standby_state;
```

- **Verification result**: Valid by Kind2 and SLDV.

[**FSM-007**]

- **Natural language**: The autopilot shall change states from PULLUP to TRANSITION when the system is supported and sensor data is good.
- **FRET input**: FSM shall always satisfy `(state = ap_maneuver_state & supported & good) => STATE = ap_transition_state`
- **Scope**: Autopilot
- **CoCoSpec formalization**:

```
guarantee "FSM-007" (state = ap_maneuver_state and supported and
     good) => STATE = ap_transition_state;
```

Table 4.10: Counter example of requirement [FSM-008]

Inputs	T = 0
standby	false
apfail	true
supported	true
good	true
state	3
Outputs	
STATE	2

- **Verification result**: Invalid due to incomplete natural language specification. The generated-by-Kind2 counterexample is shown in Table 4.9.

  We can see from the counterexample that, even though the conditions are satisfied, Autopilot changes state from `PULLUP` (same as `MANEUVER`) to `STANDBY` (instead of `TRANSITION`). This means that requirements `[FSM-006]` and `[FSM-007]` are not mutually exclusive. If we additionally constraint requirement `[FSM-007]` with condition `not standby`, we get the following FRET input and CoCoSpec formalization, which is proven valid.

  FSM shall always satisfy `(state = ap_maneuver_state & supported & good & ! standby) => STATE = ap_transition_state`

```
guarantee "FSM-007v2" (state = ap_maneuver_state and supported
   and good and not standby) =>  STATE = ap_transition_state;
```

[**FSM-008**]

- **Natural language**: The autopilot shall change states from STANDBY to TRANSITION when the pilot is not in control (not standby).
- **FRET input**: FSM shall always satisfy `satisfy (state = ap_standby_state & ! standby) => STATE = ap_transition_state`
- **Scope**: Autopilot
- **CoCoSpec formalization**:

```
guarantee "FSM-008" (state = ap_standby_state and not standby)
   =>  STATE = ap_transition_state;
```

- **Verification result**: Invalid due to incomplete natural language specification. The generated-by-Kind2 counterexample is shown in Table 4.10.

  We can see from the counterexample that, even though the conditions are satisfied, Autopilot changes state from `STANDBY` to `MANEUVER` (instead of `TRANSITION`). The transition from state `STANDBY` to `MANEUVER` is defined in the next requirement `[FSM-009]`. This means that requirements `[FSM-008]` and `[FSM-009]` are not mutually exclusive. If we additionally constrain requirement `[FSM-008]` with condition `not apfail`, we get the following FRET input and CoCoSpec formalization, which is proven valid.

  FSM shall always satisfy `(state = ap_standby_state & ! standby & ! apfail) => STATE = ap_transition_state`

```
guarantee "FSM-008v2" (state = ap_standby_state and not standby
   and not apfail) =>  STATE = ap_transition_state);
```

[**FSM-009**]

- **Natural language**: The autopilot shall change states from STANDBY to MANEUVER when a failure occurs (apfail).
- **FRET input**: FSM shall always satisfy `(state = ap_standby_state & apfail ) => STATE = ap_maneuver_state`
- **Scope**: Autopilot
- **CoCoSpec formalization**:

```
guarantee "FSM-009" (state = ap_standby_state and apfail ) =>
   STATE = ap_maneuver_state ;
```

- **Verification result**: Valid by Kind2 and SLDV.

[**FSM-010**]

- **Natural language**: The sensor shall change states from NOMINAL to FAULT when limits are exceeded.
- **FRET input**: FSM shall always satisfy `(senstate = sen_nominal_state & limits) => SENSTATE = sen_fault_state`
- **Scope**: Sensor
- **CoCoSpec formalization**:

```
guarantee "FSM-010" (senstate = sen_nominal_state and limits) =>
    SENSTATE = sen_fault_state;
```

- **Verification result**: Valid by Kind2 and SLDV.

[**FSM-011**]

- **Natural language**: The sensor shall change states from NOMINAL to TRANSITION when the autopilot is not requesting support (not request).
- **FRET input**: FSM shall always satisfy `( senstate = sen_nominal_state & ! request) => SENSTATE = sen_transition_state`
- **Scope**: Sensor
- **CoCoSpec formalization**:

```
guarantee "FSM-011" (senstate = sen_nominal_state and not
   request) => SENSTATE = sen_transition_state;
```

Table 4.11: Counter example of requirement [FSM-011]

Inputs	T = 0
mode	false
request	false
limits	true
senstate	0
Outputs	
SENSTATE	2

- **Verification result**: Invalid due to incorrect specification. The generated-by-Kind2 counterexample is shown in Table 4.11.

  We can see from the counterexample that, even though the conditions are satisfied, Sensor changes state from `NOMINAL` to `FAULT` (instead of `TRANSITION`). The transition from state `STANDBY` to `MANEUVER` is defined in requirement `[FSM-010]`. This means that requirements `[FSM-010]` and `[FSM-011]` are not mutually exclusive. If we additionally constraint requirement `[FSM-011]` with condition `not limits`, we get the following FRET input and CoCoSpec formalization, which is proven valid by both Kind2 and SLDV.

  FSM shall always satisfy `( senstate = sen_nominal_state & ! request & ! limits) => SENSTATE = sen_transition_state`

```
guarantee "FSM-011v2" (senstate = sen_nominal_state and not
    request and not limits) => SENSTATE = sen_transition_state;
```

[**FSM-012**]

- **Natural language**: The sensor shall change states from FAULT to TRANSITION when the autopilot is not requesting support (not request) and limits are not exceeded (not limits).
- **FRET input**: FSM shall always satisfy `(senstate = sen_fault_state & ! request & ! limits) => SENSTATE = sen_transition_state`
- **Scope**: Sensor
- **CoCoSpec formalization**:

```
guarantee "FSM-012" (senstate = sen_fault_state and not request
    and not limits) => SENSTATE = sen_transition_state;
```

- **Verification result**: Valid by Kind2 and SLDV.

[**FSM-013**]

- **Natural language**: The sensor shall change states from TRANSITION to NOMINAL when the autopilot is requesting support (request) and the autopilot reports the correct active mode (mode).
- **FRET input**: FSM shall always satisfy (senstate = sen_transition_state & request & MODE) ⇒ SENSTATE = sen_nominal_state

- **Scope**: Sensor
- **CoCoSpec formalization**:

```
guarantee "FSM-013" (senstate = sen_transition_state and request
    and MODE) => SENSTATE = sen_nominal_state;
```

- **Verification result**: Valid by Kind2 and SLDV.

Chapter 5

Tustin Integrator

This challenge problem represents a common flight control utility for computing the Tustin Integration of a signal $yout = T/2*(xin+xinpv)+ypv$, where $xinpv$ and ypv are the previous values of xin and $yout$, respectively. The algorithm bounds the allowable integration range with a position limiter, with TL as the Top Limit, and BL as the Bottom Limit. Other inputs are the signal to be integrated (xin), the time step (dt), a Boolean reset flag (reset), and the initial condition upon a reset condition (ic). A provision has been implemented for the limiter functionality in the algorithm. If the user plumbs a TL value that is less than BL, the algorithm swaps these numerical values to correctly bound the signal. The integrator is in Normal operation when is not in reset mode, and the output is within the specified limits (TL and BL). Tables 5.1 and 5.2 list the inputs and outputs of the Tustin Integrator component, respectively.

Next, we provide 1) the Tustin Integrator natural language requirements, 2) their FRETish form, 3) the CoCoSpec equivalent code and 4) the verification results. The verification results are summarized in Table 5.3.

Assumptions:

We added the following assumption that assumes always bottom limit (BL) is less than or equal to top limit (TL).

```
assume BL <=  TL;
```

[**TUI-001**]

- **Natural language**: When Reset is True and the Initial Condition (ic) is bounded by the provided Top and Bottom Limits (BL $\leq$ ic $\leq$ TL), the Output (yout) shall equal the Initial Condition (ic).
- **FRET input**: TUI shall always satisfy (BL $\leq$ ic & ic $\leq$ TL & reset) $\Rightarrow$ yout = ic
- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "TUI-001" (BL <= ic and ic <= TL and reset) => yout =
   ic;
```

- **Verification result**: Valid (see Table. 5.3).

Table 5.1: Tustin Integrator Inputs

Input Scope	Name	Type	Description
Global	xin	Double	Input Signal to be integrated with the Tustin method
Global	T	Double	Time step quantified by rate of execution
Global	TL	Double	Top Limit bounding the output, yout.
Global	BL	Double	Bottom Limit bounding the output, yout.
Global	reset	Boolean	Reset control, initializes output to ic value.
Global	ic	Double	Initial Condition for yout when in reset mode.

Table 5.2: Tustin Integrator Outputs

Output Scope	Name	Type	Description
Global	yout	Double	Output Signal result of Tustin integration method on xin

Table 5.3: Tustin Integrator Verification Results with Kind2 (abbr. by K) and SLDV (abbr. by S). Properties were verified individually for both Kind2 and SLDV.

Requirement	K Result	K Time	S Result	S Time
[TUI-001]	valid	0.077 sec	valid	7 sec
[TUI-002]	valid	0.062 sec	valid	7 sec
[TUI-003]	Invalid	0.092 sec	Invalid	12 sec
[TUI-003v2]	Invalid	0.072 sec	Invalid	11 sec
[TUI-003v3]	Valid	0.135 sec	Invalid*	19 sec
	Lustre generation time: 19.01sec			

*[TUI-003v3]: SLDV used additional analysis to reduce instances of rational approximation which resulted in Counterexample because of comparing two double signal without using an epsilon.

[TUI-002]

- **Natural language**: The Output (yout) shall be bounded by the provided Top and Bottom limits (TL and BL).
- **FRET input**: TUI shall always satisfy yout $\leq$ TL & yout $\geq$ BL
- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "TUI-002" yout <= TL and yout >= BL;
```

- **Verification result**: Valid (see Table. 5.3).

[TUI-003]

- **Natural language**: When in normal operation, the output shall be the result of the equation, $yout = T * 0.5 * (xin + xinpv) + ypv$

Table 5.4: Counter example of requirement [TUI-003v1]

Inputs	T = 0
xin	0.0
reset	false
T	0.0
ic	0.0
TL	-1.0
BL	-1.0
Outputs	
yout	-1.0

- **FRET input**: TUI shall always satisfy normal $\Rightarrow yout = T*0.5*(xin+xinpv)+ypv$

- **Scope**: Global

- **CoCoSpec formalization**:

```
var xinpv: real = 0.0 -> pre xin;
var ypv: real = 0.0 -> pre yout;
var normal: bool = not reset and yout <= TL and yout >= BL;
var normal_yout : real = T * 0.5 * ( xin + xinpv ) + ypv;
guarantee "TUI-003" normal => (yout = normal_yout);
```

- **Verification result**: Invalid by Kind2 (see counterexample in Table. 5.4). When changing the definition of normal variable to be defined only by 'not reset', Kind2 can find a counterexample. The following version 'TUI-003v2' is stronger thant 'TUI-003'.

```
var normal: bool = not reset; -- we removed yout <= TL and yout
   >= BL
var normal_yout : real = T * 0.5 * ( xin + xinpv ) + ypv;
guarantee "TUI-003v2" normal => (yout = normal_yout);
```

Kind2 produces the same counterexample shown in Table 5.4 that falsifies the requirement 'TUI-003v2', the expected value of *yout* is 0.0 since *normal_yout* is zero but *yout* took the value of the upper bound TL since $0.0 > TL = -1.0$ from requirement 'TUI-002'. Therefore, the conjunction of requirement 'TUI-002' must be added (additional assumption that the computed value *normal_yout* is bounded between BL and TL).

```
var normal: bool = not reset and normal_yout <= TL and
   normal_yout >= BL;
var normal_yout : real = T * 0.5 * ( xin + xinpv ) + ypv;
guarantee "TUI-003v3"  normal => (yout = normal_yout);
```

The requirement "TUI-003v3" was proven valid (by Kind2) since we added the condition `normal_yout <= TL and normal_yout >= BL`.

SLDV produced a counterexample for this requirement, because it used additional analysis to reduce instances of rational approximation which resulted in Counterexample because of comparing two double signal without adding an epsilon (`yout = normal_yout` instead of `abs(yout - normal_yout) < eps`).

Once we set the following parameter to off:

`ReduceRationalApprox - ['off' | {'on'}] causes Simulink Design Verifier to try reducing the instances of rational approximation from analysis results.`

SLDV returns "Undecided due to nonlinearities" answer to the requirement.

Also, once we use `abs(yout - normal_yout) < 0.001`, SLDV returns "Undecided due to nonlinearities".

[**TUI-004**]

- **Natural language**: a. After 10 seconds of Computation at an execution frequency of 10 hz, the Output should equal 10 within a +/- 0.1 tolerance, for a Constant Input (xin = 1.0), and the sample delta time T = 0.1 seconds when in normal mode of operation.
- **FRET input**: Cannot be formalized in FRET since it contains condition with duration.
- **Scope**: Global

Chapter 6

Control Loop Regulators

This challenge problem demonstrates a simplified regulators inner loop architecture used in many feedback control applications. The model includes two subsystems. The first is the input subsystem, which is used solely for signal routing with variable renaming and bus creation. The second subsystem is the primary system under test, the regulators algorithm. The regulators algorithm consists of 5 classical controllers for establishing the desired dynamics of a vehicle for the roll, pitch, yaw, axial, and height channels. Tables 6.1 and 6.2 list the inputs and outputs of the Regulators component, respectively.

Next, we provide 1) the Regulators natural language requirements, 2) their FRETish form, 3) the CoCoSpec equivalent code and 4) the verification results. The verification results are summarized in Table 6.3.

Note on requirements 1 to 5

The first 5 requirements are of the same type: '*the command should not exceed a given threshold for more than 100 frames*'. In order to prove this, we need to prove that for any consecutive 100 steps, the command can not keep a value exceeding the threshold. A way to formalize this is to count how many times the command exceeds the threshold. The counter reset to zero anytime the threshold is not exceeded:

```
var threshold: real = 50.0; --example of a threshold
var count: int = 0 -> if (command > threshold) then pre count + 1
   else 0;
guarantee count <= 100; -- we never exceed 100 frames
```

From Table 6.3 we can see that this type of requirements are hard to prove since, in addition to non-linear arithmetic in this model, the solver must prove that the property holds for 100 steps from any state of the system.

Note on requirements 6 to 10

Requirements 6 to 10 follow the same template '*The transient changes of an acceleration (roll/putch/yaw) shall never exceed a threshold.*' This can be formalized by computing the transient change, which is `0.0 -> (command - pre command) / dt`, where dt is the sample time of the execution of the model. We want to check that the transient change never exceeds the threshold.

Requirements 6 to 10 were falsified by Kind2 because the model is an open loop system and no constraints are made on the inputs. The commands computed by the regulators

Table 6.1: Regulators Inputs

Input Scope	Name	Type	Description
Global	beta_adc_deg	Double	Angle of sideslip [deg]
Global	vtas_adc_kts	Double	True airspeed [knots]
Global	lcv_cmd_fcs_dps	Double	Roll Control Variable Command for Flight Control System [deg/sec]
Global	hdg_des_deg	Double	Desired Heading [deg]
Global	mcv_cmd_fcs_dps	Double	Pitch Control Variable Command for Flight Control System [deg/sec]
Global	alt_des_ft	Double	Desired Altitude [feet]
Global	ncv_cmd_fcs_dps	Double	Yaw Control Variable Command for Flight Control System [deg/sec]
Global	xcv_cmd_fcs_fps	Double	Axial Control Variable Command for Flight Control System [ft/sec]
Global	airspeed_des_fps	Double	Desired Airspeed [feet/sec]
Global	hcv_cmd_fcs_fps	Double	Height Control Variable Command for Flight Control System [ft/sec]
Global	lcv_fps_dps	Double	Roll Control Variable for Flight Control System [deg/sec]
Global	mcv_fcs_dps	Double	Pitch Control Variable for Flight Control System [deg/sec]
Global	ncv_fcs_dps	Double	Yaw Control Variable for Flight Control System [deg/sec]
Global	dcv_fcs_fps	Double	Axial Control Variable for Flight Control System [ft/sec]
Global	zcv_fcs_fps	Double	Height Control Variable for Flight Control System [ft/sec]
Global	betadot	Double	Angle of sideslip rate [deg/sec]

Table 6.2: Regulators Outputs

Output Scope	Name	Type	Description
Global	lcvdt_cmd_fcs_dps2	Double	Roll Output Command [deg/sec2]
Global	mcvdt_cmd_fcs_dps2	Double	Pitch Output Command [deg/sec2]
Global	ncvdt_cmd_fcs_dps2	Double	Yaw Output Command [deg/sec2]
Global	xcvdt_cmd_fcs_fps2	Double	Axial Output Command [ft/sec2]
Global	hcvdt_cmd_fcs_fps2	Double	Height Output Command [ft/sec2]

depend on the response of the controlled system. As a result, the counterexamples given by the model checker may make no sense physically.

[REG-001]

- **Natural language**: The Inner Loop Roll Regulator Shall not command angular roll accelerations greater than the capability of the system ($50 deg/sec2$) for durations exceeding 100 frames (1 second @ 100 hz).
- **FRET input**: REG shall always satisfy count_roll_output_exceeding_50 $\leq$ 100

Table 6.3: Regulators Verification Results with Kind2 (abbr. by K) and SLDV (abbr. by S). Timeout was set to 2 hours

Requirement	K Result	K Time	S Result	S Time
[REG-001]	Undecided	Timeout	Undecided	Timeout
[REG-002]	Undecided	Timeout	Undecided	Timeout
[REG-003]	Valid	10.046 sec	Valid	12 sec
[REG-004]	Undecided	Timeout	Undecided	Timeout
[REG-005]	Undecided	Timeout	Undecided	Timeout
[REG-006]	Invalid	5.998 sec	Undecided	Timeout
[REG-007]	Invalid	5.998 sec	Undecided	1321 sec
[REG-008]	Invalid	5.998 sec	Undecided	2163 sec
[REG-009]	Invalid	5.998 sec	Undecided	Timeout
[REG-010]	Invalid	5.998 sec	Undecided	Timeout
Total running time	CoCoSim: Timeout		SLDV: Timeout	

- **Scope**: Global
- **CoCoSpec formalization**:

```
var count_roll_output_exceeding_50: int = 0 -> if (
   lcvdt_cmd_fcs_dps2 > 50.0) then pre
   count_roll_output_exceeding_50 + 1 else 0;
guarantee "REG001" count_roll_output_exceeding_50 <= 100 ;
```

- **Verification result**: Undecided, see Table. 6.3.

[**REG-002**]

- **Natural language**: The Inner Loop Pitch Regulator Shall not command angular pitch accelerations greater than the capability of the system ($50deg/sec2$) for durations exceeding 100 frames (1 second @ 100 hz).
- **FRET input**: REG shall always satisfy count_pitch_output_exceeding_50 $\leq$ 100
- **Scope**: Global
- **CoCoSpec formalization**:

```
var count_pitch_output_exceeding_50: int = 0 -> if (
   mcvdt_cmd_fcs_dps2 > 50.0) then pre
   count_pitch_output_exceeding_50 + 1 else 0;
guarantee "REG002" count_pitch_output_exceeding_50 <= 100;
```

- **Verification result**: Undecided, see Table 6.3.

[**REG-003**]

- **Natural language**: The Inner Loop Yaw Regulator Shall not command angular yaw accelerations greater than the capability of the system ($50deg/sec2$) for durations exceeding 100 frames (1 second @ 100 hz).
- **FRET input**: REG shall always satisfy count_yaw_output_exceeding_50 $\leq$ 100

- **Scope**: Global
- **CoCoSpec formalization**:

```
var count_yaw_output_exceeding_50: int = 0 -> if (
   ncvdt_cmd_fcs_dps2 > 50.0) then pre
   count_yaw_output_exceeding_50 + 1 else 0;
guarantee "REG003" count_yaw_output_exceeding_50 <= 100;
```

- **Verification result**: Valid, see Table 6.3.

[**REG-004**]

- **Natural language**: The Inner Loop Airspeed Regulator Shall not command translational axial accelerations greater than the capability of the system ($32 ft/sec2$) for durations exceeding 100 frames (1 second @ 100 hz).
- **FRET input**: REG shall always satisfy count_airspeed_output_exceeding_32 $\leq$ 100
- **Scope**: Global
- **CoCoSpec formalization**:

```
var count_airspeed_output_exceeding_32: int = 0 -> if (
   xcvdt_cmd_fcs_fps2 > 32.0) then pre
   count_airspeed_output_exceeding_32 + 1 else 0;
guarantee "REG004" count_airspeed_output_exceeding_32 <= 100;
```

- **Verification result**: Undecided, see Table. 6.3.

[**REG-005**]

- **Natural language**: The Inner Loop Height Regulator Shall not command translational height accelerations greater than the capability of the system ($32 ft/sec2$) for durations exceeding 100 frames (1 second @ 100 hz).
- **FRET input**: REG shall always satisfy count_height_output_exceeding_32 $\leq$ 100
- **Scope**: Global
- **CoCoSpec formalization**:

```
var count_height_output_exceeding_32: int = 0 -> if (
   hcvdt_cmd_fcs_fps2 > 32.0) then pre
   count_height_output_exceeding_32 + 1 else 0;
guarantee "REG005" count_height_output_exceeding_32 <= 100 ;
```

- **Verification result**: Undecided, see Table 6.3.

Table 6.4: Counter example of requirement [REG-006]

Inputs	T = 0	T = 0.01
lcv_cmd_fcs_dps	0.0	0.8
all other inputs	0.0	0.0
Outputs		
lcvdt_cmd_fcs_dps2	0.0	2.0

[REG-006]

- **Natural language**: The Inner Loop Roll Regulator Shall not command transient changes in angular roll acceleration greater than $50 deg/sec2/sec$.
- **FRET input**: REG shall always satisfy roll_command_acceleration ≤ 50
- **Scope**: Global
- **CoCoSpec formalization**:

```
var roll_command_acceleration: real = 0.0 ->  (
   lcvdt_cmd_fcs_dps2 - pre lcvdt_cmd_fcs_dps2) * 100.0;
guarantee "REG006" roll_command_acceleration <= 50.0;
```

- **Verification result**: Invalid using Kind2. See counterexample in Table 6.4. Roll Output command changed from $0.0 deg/sec2$ to $2.0 deg/sec2$ in one step of 0.01 which exceeds the maximum transient change allowed $50 deg/sec2/sec$.

[REG-007]

- **Natural language**: The Inner Loop Pitch Regulator Shall not command transient changes in angular pitch acceleration greater than $50 deg/sec2/sec$.
- **FRET input**: REG shall always satisfy pitch_command_acceleration ≤ 50
- **Scope**: Global
- **CoCoSpec formalization**:

```
var pitch_command_acceleration: real = 0.0 ->  (
   mcvdt_cmd_fcs_dps2 - pre mcvdt_cmd_fcs_dps2) * 100.0;
guarantee "REG007" pitch_command_acceleration <= 50.0;
```

- **Verification result**: Invalid for Kind2. See counterexample in Table 6.5. Pitch Output command changed from $-477.0 deg/sec2$ to $110063.46 deg/sec2$ in one step of 0.01 which exceeds the maximum transient change allowed $50 deg/sec2/sec$.

[REG-008]

- **Natural language**: 1. The Inner Loop Yaw Regulator Shall not command transient changes in angular yaw acceleration greater than $50 deg/sec2/sec$.
- **FRET input**: REG shall always satisfy yaw_command_acceleration ≤ 50
- **Scope**: Global

Table 6.5: Counter example of requirement [REG-007]

Inputs	T = 0	T = 0.01
vtas_adc_kts	-75.0	0.0
mcv_cmd_fcs_dps	1.0	19160.0
mcv_fcs_dps	19081.0	0.0
all other inputs	0.0	0.0
Outputs		
mcvdt_cmd_fcs_dps2	-477.0	110063.46

- **CoCoSpec formalization**:

```
var yaw_command_acceleration: real = 0.0 ->  (ncvdt_cmd_fcs_dps2
    - pre ncvdt_cmd_fcs_dps2) * 100.0;
guarantee "REG008" yaw_command_acceleration <= 50.0;
```

- **Verification result**: Invalid using Kind2. See counterexample in Table 6.6. Yaw Output command changed from $-3.0 deg/sec2$ to $0.0 deg/sec2$ in one step of 0.01 which exceeds the maximum transient change allowed $50 deg/sec2/sec$.

Table 6.6: Counter example of requirement [REG-008]

Inputs	T = 0	T = 0.01
ncv_cmd_fcs_dps	3.0	0.0
all other inputs	0.0	0.0
Outputs		
ncvdt_cmd_fcs_dps2	-3.0	0.0

[REG-009]

- **Natural language**: The Inner Loop Airspeed Regulator Shall not command transient changes in translational axial acceleration greater than $32 ft/sec2/sec$.
- **FRET input**: REG shall always satisfy airspeed_command_acceleration $\leq$ 32
- **Scope**: Global
- **CoCoSpec formalization**:

```
var airspeed_command_acceleration: real = 0.0 -> (
   xcvdt_cmd_fcs_fps2 -  pre xcvdt_cmd_fcs_fps2) * 100.0;
guarantee "REG009" airspeed_command_acceleration <= 32.0;
```

- **Verification result**: Invalid using Kind2. See counterexample in Table 6.7. Airspeed command changed from $-2.0 ft/sec2$ to $0.0 ft/sec2$ in one step of 0.01 which exceeds the maximum transient change allowed $32 ft/sec2/sec$.

[REG-010]

- **Natural language**: The Inner Loop Height Regulator Shall not command transient changes in translational height acceleration greater than $32 ft/sec2/sec$.
- **FRET input**: REG shall always satisfy height_command_acceleration $\leq$ 32

Table 6.7: Counter example of requirement [REG-009]

Inputs	T = 0	T = 0.01
xcv_cmd_fcs_fps	-4.0	0.0
all other inputs	0.0	0.0
Outputs		
xcvdt_cmd_fcs_fps2	-2.0	0.0

Table 6.8: Counter example of requirement [REG-010]

Inputs	T = 0	T = 0.01
zcv_cmd_fcs_fps	0.0	1.0
all other inputs	0.0	0.0
Outputs		
hcvdt_cmd_fcs_fps2	0.0	1.001

- **Scope**: Global
- **CoCoSpec formalization**:

```
var height_command_acceleration: real = 0.0 ->  (
   hcvdt_cmd_fcs_fps2 - pre hcvdt_cmd_fcs_fps2) * 100.0;
guarantee "REG010" height_command_acceleration <= 32.0;
```

- **Verification result**: Invalid using Kind2. See counterexample in Table 6.8. Height command changed from $0.0 ft/sec2$ to $1.001 ft/sec2$ in one step of 0.01 which exceeds the maximum transient change allowed $32 ft/sec2/sec$.

Chapter 7

Nonlinear Guidance Algorithm

This example is a nonlinear algorithm for generating a guidance command for an air vehicle. The vector method determines an intercept location in Euclidean space for a target (static) referred to as aim points. The intercept location is a function of the relative position between the vehicle and target, where a minimum relative position to maintain is specified as a desired standoff distance (See Fig. 7.1).

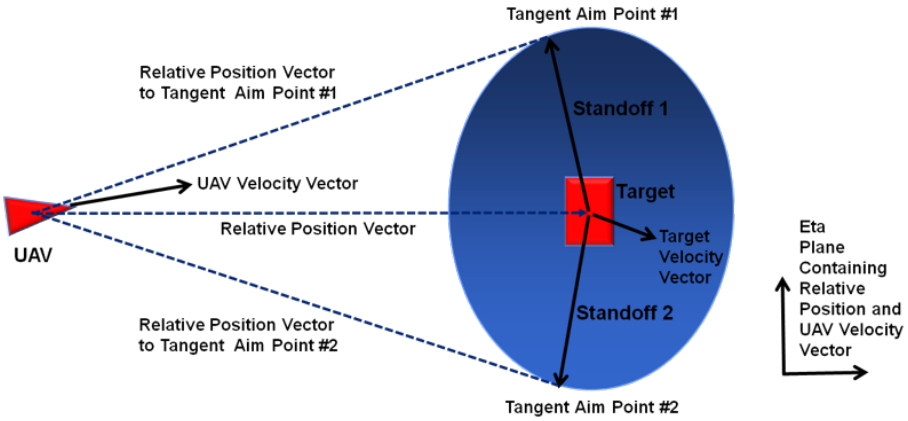


Figure 7.1: Nonlinear Guidance illustration.

Tables 7.1 and 7.2 list the inputs and outputs of the Nonlinear Guidance component, respectively. Table 7.3 lists additional variables of the model, without specifying their scope. The requirements of this problem are high-level and to understand how to formalize them and connect them with model variables we discussed multiple times with an expert. Additional refinement of these requirements can lead to a more direct formalism. Next, we provide 1) the Nonlinear Guidance component natural language requirements and 2) their FRETish form.

[NLG-001]

- **Natural language**: NLGuidance shall always maintain the target on the port-side of the vehicle.
- **Interpretation**: We interpret this requirement to that the dot product of velocity of the vehicle (V_v) and the computed Aim Point position ($yout$) should be greater than zero ($V_v \cdot yout > 0$).

Table 7.1: Nonlinear Guidance Inputs

Input Scope	Name	Type	Description
Global	Xtarg	Double	Inertial Position 3x1 Vector of the Target [m]
Global	Xv	Double	Inertial Position 3x1 Vector of the Vehicle [m]
Global	Vv	Double	Inertial Velocity 3x1 Vector of the Vehicle [m/sec]
Global	r	Double	Minimum Standoff Radius [m]
Global	Vt	Double	Inertial Velocity 3x1 Vector of the Target [m/sec]

Table 7.2: Nonlinear Guidance Outputs

Output Scope	Name	Type	Description
Global	yout	Double	Aim Point Inertial Position 3x1 Vector for CCW Orbit Acquisition [m]

Table 7.3: Additional Model Variables. No scope is specified.

Name	Variable Name	Definition
Relative Position Vector	Xr	Inertial Position Vector describing position of Vehicle with respect to the Target
Aim Point #1	Xap1	Inertial Position Vector of Aim Point#1
Aim Point #2	Xap2	Inertial Position Vector of Aim Point#2
Relative Point Vector to Tangent Aim Point#1	r1	Inertial Position Vector describing position of Vehicle with respect to the Xap1
Relative Point Vector to Tangent Aim Point#2	r2	Inertial Position Vector describing position of Vehicle with respect to the Xap2
Unit Vector of Aim Point #1 from Target	u1	Inertial Position Vector describing Aim Point #1 with respect to Target
Unit Vector of Aim Point #2 from Target	u2	Inertial Position Vector describing Aim Point #2 with respect to Target

Table 7.4: NLG Verification Results with Kind2 (abbr. by K) and SLDV (abbr. by S). Timeout was set to 2 hours.

Requirement	K Result	K Time	S Result	S Time
[NLG-001]	Undecided	Timeout	Undecided (Due to nonlinearities)	15 sec
[NLG-002]	Undecided	Timeout	Undecided (Due to nonlinearities)	15 sec
[NLG-003]	Undecided	Timeout	Undecided (Due to nonlinearities)	15 sec
[NLG-004]	Undecided	Timeout	Undecided (Due to nonlinearities)	15 sec
[NLG-005]	Undecided	Timeout	Undecided (Due to nonlinearities)	15 sec
[NLG-006]	Undecided	Timeout	Undecided (Due to stubbing)	15 sec
[NLG-007]	Undecided	Timeout	Undecided (Due to nonlinearities)	15 sec
Total running time	CoCoSim: Timeout		SLDV: 15sec	

- **FRET input**: NLGuidance shall always satisfy $V_v \cdot yout > 0$.
- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "NLG001" dot_product(Vc, yout) > 0;
```

where `dot_product` is the dot product defined by:

```
node  dot_product(x : real^3; y : real^3) returns (z : real;);
let
  z =  x[0] * y[0]  +  x[1] * y[1]  +  x[2] * y[2] ;
tel
```

- **Verification result**: Undecided due to nonlinearities, see Table 7.4.

[**NLG-002, NLG-003**]

- **Natural language NLG-002**: NLGuidance shall compute the inertial position vector for aim point 1, defining the location at Standoff 1 with an offset from the target position, and oriented on a vector perpendicular to the tangent relative position vector from the vehicle to the corresponding aim point.

- **Natural language NLG-003**: NLGuidance shall compute the inertial position vector for aim point 2, defining the location at Standoff 2 with an offset from the target position, and oriented on a vector perpendicular to the tangent relative position vector from the vehicle to the corresponding aim point.

- **Interpretation**: Both requirements can be checked at the same time. Output *yout* defines Aim Point Inertial Position, in fact *yout* is referring to X_{ap1} or X_{ap2} based on what is the chosen Aim point. Both requirement are verified if $mag(yout - X_{targ}) \geq r$ & $(yout - X_{targ}) \cdot (yout - X_v) = 0$. where mag is the magnitude function and the second product is a dot product.

- **FRET input**: NLGuidance shall always satisfy $mag(yout - X_{targ}) \geq r$ & $dot\_product((yout - X_{targ}), (yout - X_v)) = 0$, where mag is the magnitude function.

- **Scope**: Global

- **CoCoSpec formalization**:

```
guarantee "NLG002&3" mag(yout - Xtarg) >= r and dot_product((
   yout - Xtarg),(yout - Xv)) = 0;
```

where `dot_product` is defined in "NLG001" and `mag` function is represented in Lustre as follows:

```
node  mag(x : real^3) returns (y : real;);
let
  y = sqrt( x[0] * x[0]  +  x[1] * x[1]  +  x[2] * x[2] );
tel
```

- **Verification result**: Undecided due to nonlinearities, see Table 7.4.

[**NLG-004**]

- **Natural language**: NLGuidance shall always select an inertial position vector of aim point #1 or #2 which shall result in a counter clockwise loiter for the UAV. For example, the picture above with vehicle position, UAV, would return Tangent Aim Point #2.

- **FRET input**: NLGuidance shall always satisfy $yout \cdot Xr > 0$ where yout is the selected aim point by the guidance algorithm. item **Scope**: Global

- **CoCoSpec formalization**:

```
guarantee "NLG004" dot_product(yout, Xr) > 0;
```

  where

```
var Xr: real^3 = Xtarg - Xv;
```

- **Verification result**: Undecided due to nonlinearities, see Table. 7.4.

[**NLG-005**]

- **Natural language**: When the UAV relative position to the target is less than the minimum standoff distance, NLGuidance shall command the nearest inertial position in order to reestablish the minimum standoff distance while maintaining the target on the port-side of the vehicle.

- **FRET input**:We decomposed NLG-005 into the following two requirements NLG-005a and NLG-005b:
  If $\text{mag}(Xr) < r$ & $\text{mag}(r1) < \text{mag}(r2)$ NLGuidance shall always satisfy $yout = Xap1$
  If $\text{mag}(Xr) < r$ & $\text{mag}(r1) \geq \text{mag}(r2)$ NLGuidance shall always satisfy $yout = Xap2$,

  where $r1$ (resp. $r2$) is the inertial position vector describing position of vehicle with respect to $Xap1$ (resp. $Xap2$).

- **Scope**: Global

- **CoCoSpec formalization**:

```
guarantee "NLG005A" H(not (mag(Xr) < r and mag(r1) < mag (r2)))
  ) or S(((yout = Xap1) and ((mag(Xr) < r and mag(r1) < mag(r2)
  ) and ((pre (not (mag(Xr) < r and mag(r1) < mag(r2)))) or FTP
  ))), (yout = Xap1));
```

```
guarantee "NLG005B" H(not (mag(Xr) < r and mag(r1) >= mag (r2))
  )) or S(((yout = Xap1) and ((mag(Xr) < r and mag(r1) >= mag(
  r2)) and ((pre (not (mag(Xr) < r and mag(r1) >= mag(r2)))) or
   FTP))), (yout = Xap2));
```

- **Verification result**: Undecided due to nonlinearities, see Table 7.4.

[**NLG-006**]

- **Natural language**: NLGuidance shall output consistent aim point with a static target without appreciable transient behavior in the command generation other than aim point switching where a transient is required to maintain a counter clockwise loiter (ref requirement 3). Appreciable transient behavior is defined as erratic changes in the aim point command, beyond the following specific tolerance:
  - The change in the magnitude of the output over one frame of execution with T sample period shall not exceed the quantity of the combined velocity of the target plus the velocity of the vehicle multiplied by T.
- **FRET input**: NLGuidance shall always satisfy mag(yout - pre_yout) $\leq$ T * mag(Vt + Vv), where T is the sample period of the nonlinear guidance algorithm.
- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "NLG006" (0.0 -> mag(yout - pre yout)) <= T * mag(Vt
  + Vv);
```

- **Verification result**: Undecided due to nonlinearities, see Table 7.4.

[**NLG-007**]

- **Natural language**: NLGuidance shall output the equivalent altitude of the vehicle for in-plane navigation. In-plane navigation is defined where the target and the vehicle altitude (3rd component in the input inertial position vectors) are equal.
- **FRET input**: NLGuidance shall always satisfy (yout[2] - Xtarg[2]) =0 .
- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "NLG007" (yout[2] - Xtarg[2])=0;
```

- **Verification result**: Undecided due to nonlinearities, see Table 7.4.

Chapter 8

Feedforward Cascade Connectivity Neural Network

This example is a two-input single-output two hidden layer feed forward nonlinear 2x10x10x1 neural network. Neural networks of this form are common utilities in modeling and simulation for capturing complex numerical dependencies. In this example, a single dependent variable, z, is computed based on two independent parameters, x and y. The truth data for this example is provided in a ".mat" matlab file, in the matrix original_data $\in \mathbb{R}^{200x3}$. xt is the 1st column, yt is the 2nd column, and zt is the 3rd column of data for 200 indices. Tables 8.1 and 8.2 list the inputs and outputs of the Neural Network component, respectively.

Next, we provide 1) the Neural Network component natural language requirements, 2) their FRETish form, 3) the CoCoSpec equivalent code and 4) the verification results. The verification results are summarized in Table 8.3.

[**NN-001**]

- **Natural language**: The maximum value of the NN output, z, shall always be less than or equal to 1.1, regardless of the input values.
- **FRET input**: NN shall always satisfy z $\leq$ 1.1
- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "NN001" z <= 1.1;
```

- **Verification result**: Undecided, see Table 8.3.

[**NN-002**]

- **Natural language**: The minimum value of the NN output, z, shall always be greater than or equal to -0.2, regardless of the input values.
- **FRET input**: NN shall always satisfy z $\geq$ -0.2
- **Scope**: Global

Table 8.1: Neural Network Inputs

Input Scope	Name	Type	Description
Global	x	Double	First Input Signal
Global	y	Double	Second Input Signal

Table 8.2: Neural Network Outputs

Output Scope	Name	Type	Description
Global	z	Double	Dependent Variable Output Signal

Table 8.3: Neural Network component Verification Results with Kind2 (abbr. by K) and SLDV (abbr. by S). Timeout was set at 2 hours.

Req.	K Result	K Time	S Result	S Time
[NN-001]	Undecided	Timeout	Undecided	Timeout
[NN-002]	Undecided	Timeout	Undecided	Timeout
[NN-003]	Undecided	Timeout	Undecided	Timeout
[NN-004]	Undecided	Timeout	Undecided	Timeout
Total running time	CoCoSim: Timeout		SLDV: Timeout	

- **CoCoSpec formalization**:

```
guarantee "NN002" z >= -0.2;
```

- **Verification result**: Undecided, see Table 8.3.

[NN-003]

- **Natural language**: Using a first order finite backward difference equation, the spatial derivatives of $\delta z/\delta xt = (z(n,1) - z(n-1))/(xt(n,1) - xt(n-1,1)$ and $\delta z/\delta yt = (z(n,1) - z(n-1))/(yt(n,1) - yt(n-1,1)$ shall never exceed a top bound of +10 or bottom bound of -35 (e.g. $-35 <= \delta z/\delta(xt, yt) <= 10$), where n denotes an index to the current values and n-1 denotes the prior values in the included truth data for xt and yt.
- **FRET input**:
  - **NN-003a** NN shall for 200 secs satisfy DeltaZDividedByXt $\leq$ 10 & DeltaZDividedByXt $\geq$ -35
  - **NN-003b** NN shall for 200 secs satisfy DeltaZDividedByYt $\leq$ 10 & DeltaZDividedByYt $\geq$ -35
- **Scope**: Global
- **CoCoSpec formalization**:

```
var DeltaZDividedByXt: real = 0.0 -> (z- pre z)/(xt -  pre xt);
var DeltaZDividedByYt: real = 0.0 -> (z- pre z)/(yt -  pre yt);
guarantee "NN003a" OT(200,0,FTP) => (DeltaZDividedByXt <= 10.0
   and DeltaZDividedByXt >= -35.0);
guarantee "NN003b" OT(200,0,FTP) => (DeltaZDividedByYt <= 10.0
   and DeltaZDividedByYt >= -35.0);
```

- **Verification result**: Undecided, see Table 8.3.

[**NN-004**]

- **Natural language**: The absolute error between the zt truth data and the output z shall never exceed a tolerance of 0.01, for the equivalent input of (xt, yt).
- **FRET input**: NN shall for 200 sec satisfy (x = xt & y = yt => AbsoluteErrorZtMinusZ $\leq$ 0.01)
- **Scope**: Global
- **CoCoSpec formalization**: We check the requirement only for the first 200 steps since the provided data xt, yt and zt has only a reference to 200 values.

```
var AbsoluteErrorZtMinusZ: real = if (zt-z) > 0.0 then zt - z
   else z - zt;
guarantee "NN004" OT(200,0,FTP) => ( x = xt and y = yt =>
   AbsoluteErrorZtMinusZ <= 0.01 );
```

- **Verification result**: Undecided, see Table 8.3.

Chapter 9

Abstraction of a Control Allocator (Effector Blender)

This challenge problem provides a subset of an algorithm commonly referred to as the control allocation method, which enables the calculation of the optimal effector (surface) configuration for a vehicle, given a problem type (typically desired acceleration error, or desired control minimization effort, or a combination of both). In this specific case, the problem type is control minimization of the form:

$$\text{Minimize } J(\delta u) = (\delta u - \delta up)^T * W_p * (\delta u - \delta up). \text{ Subject to } B * \delta u = \delta d$$

where J represents the cost of the control effort, u is the control solution, up is the preferred control solution, Wp is a Weighting matrix, B is the linearized control effectivity matrix, and d is the desired acceleration error . In this specific case for the over-determined solution, when there are more surface effectors than commanded axes ($n < m$) where n is the length of the d vector, and m is the length of the u vector, an analytic solution exists [1].

$$\delta u = P * \delta d$$

, where

$$P = (W_p^T)^{-1} * B^T * (B * (W_p^T)^{-1} * B^T)^{-1}$$

Tables 9.1 and 9.2 list the inputs and outputs of the Effector Blender component, respectively.

Next, we provide 1) the Effector Blender component natural language requirements, 2) their FRETish form, 3) the CoCoSpec equivalent code and 4) the verification results. The verification results are summarized in Table 8.3.

[EB-001]

- **Natural language**: When the determinant of $B(inv(Wp)B$ is $<= 1e{-}12$ as indicated by *ridge_on* set to True, the inversion of the $B(inv(Wp)B$ matrix should be accurate to 6 digits precision in that each element in the check output matrix is within a $1e-6$ tolerance with respect to the elements of a 3x3 identity matrix.

[1] Bordignon, Ken, and John Bessolo. "Control Allocation for the X-35B." 2002 Biennial International Powered Lift Conference and Exhibit. 2002.

Table 9.1: Effector Blender Inputs

Input Scope	Name	Type	Description
Global (Specified)	Wp	Double	The control weighting preference matrix of size 5x5
Global	B	Double	The control effectiveness matrix of size 3x5
Global (Specified)	d	Double	The desired acceleration error 3x1
Global (Specified)	up	Double	The preferred control solution 5x1

Table 9.2: Effector Blender Outputs

Output Scope	Name	Type	Description
Global	check	Double	Inversion Check Matrix of size 3x3
Global	yinv	Double	Inverse of $(B(inv(Wp)B$ of size 3x3
Global	P	Double	Transformation Gain Matrix Solution of size 5x3
Global	u	Double	The control effector solution vector of size 5x1
Global	Buminusd	Double	Error vector of constraint of size 3x1
Global	J	Double	Total Cost of solution of size 1x1
Global	ridge_on	Double	Ridge Regression Diagonalization Term Active

Table 9.3: Effector Blender component Verification Results with Kind2. Timeout was set at 2 hours. SLDV terminated due to nonlinearities.

Requirements	Kind2 Result	Kind2 Time	SLDV Result	SLDV Time
[EB-001]	Undecided	Timeout	Undecided	131 sec
[EB-002]	Undecided	Timeout	Undecided	131 sec
[EB-003]	Unformalized		Unformalized	
[EB-004]	Undecided	Timeout	Undecided	128 sec
[EB-005]	Unformalized		Unformalized	
Total running time	CoCoSim: Timeout		SLDV: 131 sec	

- **FRET input**: EB shall always satisfy (det_B_BT $\leq$ eps12) $\Rightarrow$ (ridge_on = true) & abs(check_1_1 -1.0) < eps6 & abs(check_1_2) < eps6 & abs(check_1_3) < eps6 & abs(check_2_1) < eps6 & abs(check_2_2 - 1.0) < eps6 & abs(check_2_3) < eps6 & abs(check_3_1) < eps6 & abs(check_3_2) < eps6 & abs(check_3_3 - 1.0) < eps6

- **Scope**: Global

- **CoCoSpec formalization**: In this example, W_p is provided as Identity I_{3x3}, therefore $inv(Wp)$ is I_{3x3} and $B(inv(Wp)B = BB$. We compute BB using the variables `B_B_T_i_j`, where `B_B_T_i_j` refers to $BB(i, j)$. The determinant of BB is then calculated using the node `det_3x3` defined in Section 12.

```
var B_B_T_1_1 : real = B_1_1*B_1_1 + B_1_2*B_1_2 + B_1_3*B_1_3 +
    B_1_4*B_1_4 + B_1_5*B_1_5;
var B_B_T_1_2 : real = B_1_1*B_2_1 + B_1_2*B_2_2 + B_1_3*B_2_3 +
    B_1_4*B_2_4 + B_1_5*B_2_5;
var B_B_T_1_3 : real = B_1_1*B_3_1 + B_1_2*B_3_2 + B_1_3*B_3_3 +
    B_1_4*B_3_4 + B_1_5*B_3_5;

var B_B_T_2_1 : real = B_B_T_1_2;
var B_B_T_2_2 : real = B_2_1*B_2_1 + B_2_2*B_2_2 + B_2_3*B_2_3 +
```

```
    B_2_4*B_2_4 + B_2_5*B_2_5;
var B_B_T_2_3 : real = B_2_1*B_3_1 + B_2_2*B_3_2 + B_2_3*B_3_3 +
    B_2_4*B_3_4 + B_2_5*B_3_5;

var B_B_T_3_1 : real = B_B_T_1_3;
var B_B_T_3_2 : real = B_B_T_2_3;
var B_B_T_3_3 : real = B_3_1*B_3_1 + B_3_2*B_3_2 + B_3_3*B_3_3 +
    B_3_4*B_3_4 + B_3_5*B_3_5;

var det_B_BT : real = det_3x3(B_B_T_1_1, B_B_T_2_1, B_B_T_3_1,
   B_B_T_1_2, B_B_T_2_2, B_B_T_3_2, B_B_T_1_3, B_B_T_2_3,
   B_B_T_3_3);

var eps12 :real = 0.000000000001;
var eps6 : real = 0.000001;

guarantee "EB-001" (det_B_BT <=  eps12) => (ridge_on = true)
and  abs(check_1_1 -1.0)  < eps6
and  abs(check_1_2)       < eps6
and  abs(check_1_3)       < eps6
and  abs(check_2_1)       < eps6
and  abs(check_2_2 - 1.0) < eps6
and  abs(check_2_3)       < eps6
and  abs(check_3_1)       < eps6
and  abs(check_3_2)       < eps6
and  abs(check_3_3 - 1.0) < eps6;
```

- **Verification result**: Undecided, see Table 9.3.

[EB-002]

- **Natural language**: When the determinant of $B(inv(Wp)B$ is $> 1e-12$ as indicated by *ridge_on* set to False, the inversion of the $B(inv(Wp)B$ matrix should be accurate to 12 digits precision in that each element in the check output matrix is within a $1e-12$ tolerance with respect to the elements of a 3x3 identity matrix.
- **FRET input**: Similar to [EB-001]
- **Scope**: Global
- **CoCoSpec formalization**:

  Using the same variables defined in [EB-001], we have the following property:

```
guarantee "EB-002" (det_B_BT <=  eps12) => (ridge_on = true)
and  abs(check_1_1 -1.0)  < eps12
and  abs(check_1_2)       < eps12
and  abs(check_1_3)       < eps12
and  abs(check_2_1)       < eps12
and  abs(check_2_2 - 1.0) < eps12
and  abs(check_2_3)       < eps12
and  abs(check_3_1)       < eps12
and  abs(check_3_2)       < eps12
and  abs(check_3_3 - 1.0) < eps12;
```

- **Verification result**: Undecided, see Table 9.3.

[**EB-003**]

- **Natural language**: The output u vector should be a 5x1 vector.
- **Formalization**: This requirement is trivial as the dimension of a Variable can be checked in Simulink. The dimension of u depends on the dimension of B and W_p. In the provided example, B is a 3x5 matrix and W_p is 5x5, therefore using the mathematical definition of u provided in the introduction of this case study, u is 5x1 vector.

[**EB-004**]

- **Natural language**: The 2-norm of the output Buminusd should be less than 0.01.
- **FRET input**: EB shall always satisfy Buminusd_norm $<$ 0.0001
- **Scope**: Global
- **CoCoSpec formalization**:

```
var Buminusd_norm : real = Buminusd_1*Buminusd_1 + Buminusd_2*
   Buminusd_2 + Buminusd_3*Buminusd_3;
--to avoid using sqrt, sqrt(Buminusd_norm) < 0.01 =>
   Buminusd_norm < 0.0001
guarantee "EB004" Buminusd_norm < 0.0001;
```

- **Verification result**: Undecided, see Table 9.3.

[**EB-005**]

- **Natural language**: The output cost J shall be the minimum possible value given the set of input conditions.
- **Formalization**: Unclear requirement, we could not formalize it.

Chapter 10

6DOF with DeHavilland Beaver Autopilot

This challenge problem includes a full six degree of freedom simulation of the DeHavilland Beaver airplane with autopilot. This system represents a realistic model environment where formal methods analysis could prove to be extremely beneficial to help the designer prove aspects of the closed loop system without exhaustive Monte-Carlo Simulation.

The model consists of 5 main components:

- Autopilot: This component is responsible of computing the Aileron command as well as Elevator and Rudder command.
- Signal Conditioning: This contains a signal mapping from the autopilot generated commands to the aircraft input commands.
- Environment: This component contains a wind model.
- Aircraft Dynamics: Six degree of freedom aircraft model.
- Sensors: Contains sensor models for generating output measurements.

This challenge problem has no global inputs. Inputs were instead replaced by specific constants that are provided in a data file. The requirements are generic and some of them can not be analyzed with the provided constant inputs. For instance, we have two modes of operation: 1) the heading mode and 2) the hold mode, which both appear in the provided requirements. However, due to the constant inputs, heading mode is always true, so hold mode will never be true. Therefore, we changed the model graphically to create one global component that contains the whole model without the constant inputs; the main model still feeds the constants inputs to the global component. As contract scope is local, we can express our properties in the global scope where inputs are not constants. In this manner, properties can be proved regardless of a specific scenario.

Moreover, to be able to perform compositional verification, it is important to ensure that controllers are coded and constructed in a hierarchical and modular manner based on the functional decomposition of the system. For example, it is a good practice to place high-level logic on the upper hierarchy level and place low-level controllers and system dynamics further down in the hierarchy. When analyzing complex models, such as the Autopilot, this is especially important since one might be able to prove properties locally but not at the global level. However, this is not always respected by the given Autopilot model. To

Table 10.1: Autopilot Inputs

Input Scope	Name	Type	Description
Global & Autopilot	APeng	Boolean	True when the autopilot is engaged
Global & Autopilot	HDGmode	Boolean	True when Heading mode is active
Global & Autopilot	ALTMode	Boolean	True when Altitude Hold mode is active
Global & Autopilot	HDGref	Double	Heading reference when Heading mode is active
Global & Autopilot	TurnKnob	Double	Turn Knob command
Global & Autopilot	trim_altref	Double	Trim value for altitude reference
Global & Autopilot	trim_pitchwheel	Double	Trim value for pitch wheel
Global & Signal Conditioning	trim_flap	Double	Trim value for flap
Global & Signal Conditioning	trim_throttle	Double	Trim value for throttle
Global & Signal Conditioning	trim_rudder	Double	Trim value for rudder
Autopilot	AD	Bus	Bus containing: altRate, alpha, beta, airspeed and altitude
Autopilot	ID	Bus	Bus containing: phi, theta, psi p, q, r
Signal Conditioning	Aileron	Double	Aileron command
Signal Conditioning	Elevator	Double	Elevator command
Signal Conditioning	Rudder	Double	Rudder command
Aircraft Dynamics	EnvirBus	Bus	Environment variables: g, rho and wind speed
Aircraft Dynamics	Pilot	Bus	Controls signals: uaero and uprop
Sensors	AC Bus	Bus	Bus containing : Hdot, Ve, X2, xyh, phi, theta, psi, DCM, Vbody, p, q, r

Table 10.2: Autopilot Outputs

Output Scope	Name	Type	Description
Global	Altitude	Double	Altitude [ft]
Autopilot	Aileron Cmd	Double	Aileron command
Autopilot	Elevator Cmd	Double	Elevator command
Autopilot	Rudder Cmd	Double	Rudder command
Signal Conditioning	Controls	Bus	Controls signals: uaero and uprop
Aircraft Dynamics	AC Bus	Bus	Bus containing : Hdot, Ve, X2, xyh, phi, theta, psi, DCM, Vbody, p, q, r
Sensors	Altitude	Double	Altitude [ft]

be able to perform meaningful verification of requirements that could be checked locally at the component level, we added extra outputs logging the value of local-to-the-component signals. These extra outputs are ignored and not used elsewhere hence they do not modify the semantics of the model.

Interestingly, through analysis we noticed that the model of this challenge problem is a closed loop system that contains an *algebraic loop* involving all top level components. An algebraic loop occurs when there is a circular dependency of block outputs and inputs in

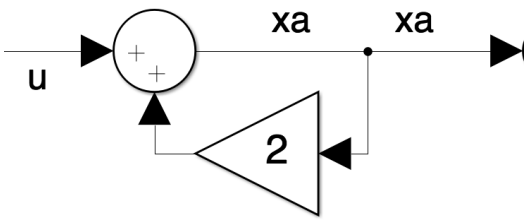
Table 10.3: FRET to Model Variables mapping for Autopilot (abbr. ap_12BAdapted/GlobalScope by global).

FRET name	**Scope**	**Model path**
altitude_hold	Global	global/ALTMode
autopilot_engaged	Roll Autopilot	global/Autopilot/Roll_Autopilot/APEng
roll_actuator_command	Roll Autopilot	global/Autopilot/Roll_Autopilot/AilCmd
roll_cmd	Roll Autopilot	global/Autopilot/Roll_Autopilot/phiCmd
roll_angle	Roll Autopilot	global/Autopilot/Roll_Autopilot/Phi
roll_hold_reference	Roll Autopilot	global/Autopilot/Roll_Autopilot/PhiRef_cmd
hdg_hold_mode_cmd	Roll Autopilot	global/Autopilot/Roll_Autopilot/HdgMode_cmd
aileron_cmd	Roll Autopilot	global/Autopilot/Roll_Autopilot/Ail Cmd

Table 10.4: Autopilot Verification Results with Kind2 (abbr. by K) and SLDV (abbr. by S)

Reqs	**Scope**	**K Result**	**K Time**	**S Result**	**S Time**
[AP-000]	Global	Unsupported (Algebraic-Loop)		Undecided due to stubbing	90 sec
[AP-001]	Roll Autopilot	Valid	< 1 sec	Valid	8 sec
[AP-002]	Roll Autopilot	Valid	< 1 sec	Valid	17 sec
[AP-003a]	Roll Autopilot	Invalid	< 1 sec	Invalid	22 sec
[AP-003b]	Roll Autopilot	Invalid	< 1 sec	Invalid	27 sec
[AP-003c]	Roll Autopilot	Invalid	< 1 sec	Invalid	17 sec
[AP-003d]	Roll Autopilot	Valid	< 1 sec	Valid	28 sec
[AP-004]	Global	Unsupported (Algebraic-Loop)		Undecided	Timeout
[AP-005]	Global	Unsupported (Algebraic-Loop)		Undecided due to nonlinearities	80sec
[AP-006]	Global	Unsupported (Algebraic-Loop)		Undecided due to stubbing	20 sec
[AP-007]	Roll Autopilot	Valid	< 1 sec	Valid	8 sec
[AP-008]	Roll Autopilot	Valid	< 1 sec	Valid	17 sec
[AP-009]	Out of scope				
[AP-010]	Global	Unsupported (Algebraic-Loop)		Undecided due to nonlinearities	20 sec
Total running time		CoCoSim: 40.589s		SLDV: 32s	

the same time-step. Figure 10.1 illustrates a simple example of an algebraic loop that is accepted by Simulink. The Sum block is an algebraic variable x_a that is constrained to be equal to the first input u plus $2 * x_a$ (i.e. $x_a = u + 2 * x_a$ or $x_a = -u$).



(a) Example of an algebraic loop accepted by Simulink.

```
xa = u + 2*xa;
```

(b) The generated Lustre that will be rejected because of the circular dependency.

Figure 10.1: A simple example of an algebraic loop.

In Simulink models, algebraic loops are algebraic constraints. Models with algebraic loops define a system of differential algebraic equations. Simulink solves these algebraic equations numerically for x_a at each step of simulation using the ODE (Ordinary Differential Equation) solver. On the other hand, Lustre forbids such constructs and no cyclic dependency is allowed.

In this challenge problem, Simulink does not identify the closed loop (between autopilot, signal conditioning, aircraft dynamics and sensors) as an algebraic loop because all involved subsystems are virtual (non-atomic) subsystems. In virtual subsystems, Simulink ignores component boundaries when determining block execution order.

CoCoSim compiler is modular and preserves the hierarchy of the model, therefore it treats all subsystems as atomic (black box approach). Each subsystem is translated as a Lustre node. Preserving the hierarchy helps analyzing the model compositionaly. But once a subsystem is causing an algebraic loop when treated as atomic, CoCoSim automatically detects it and tries to inline its content This inlining solution works only with subsystems but not with atomic blocks (e.g. Sum, Gain, Math Functions). Strangely, Kind2 was not able to detect the algebraic loop. We contacted a Kind2 developer and confirmed that there is a bug on the algebraic loop detection algorithm.

All requirements in this case study that are expressed on the top level, where the algebraic loop exists, can not be analyzed using Lustre-based model checkers. At the global scope, we have 5 out of 10 requirements. The other 5 requirements, which do not have a global scope, can be analyzed (there is no algebraic loop at this level).

[AP-000]

- **Natural language**: The altitude hold autopilot shall maintain altitude within 35 feet of the initial condition.
- **FRET input**: Autopilot shall always satisfy altitude_hold ⇒ absOf_alt_minus_IC ≤ 35.0
- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "AP-000" altitude_hold => absOf_alt_minus_altIC <=
    35.0;
```

where `absOf_alt_minus_altIC` is an internal variable defined as follows:

```
var absOf_alt_minus_altIC: real = abs(altitude - altitude_IC);
```

`abs` returns the absolute value of a real number, Additionally, we know from the data file that the initial condition is equal to 7225

```
const trim_altref: real = 7225.0; -- given in data file
var altitude_IC : real = trim_altref;
```

- **Verification result**: Unsupported by Kind2, Undecided by SLDV, see table 10.4.

[**AP-001**]

- **Natural language**: Roll Autopilot shall engage when the pilot selects the autopilot engage switch in the cockpit and disengage when the switch is deselected. When not engaged, the command to the roll actuator shall be zero.
- **FRET input**: `AP-001:` RollAutopilot shall always satisfy ! autopilot_engaged ⇒ roll_actuator_command = 0.0
- **Scope**: Roll Autopilot
- **CoCoSpec formalization**:

```
guarantee "AP-001" not autopilot_engaged =>
    roll_actuator_command = 0.0;
```

where `autopilot_engaged` and `roll_actuator_command` correspond to `AP Eng` and `Aileron Cmd` model signals, respectively.

- **Verification result**: Valid, see table 10.4.

[**AP-002**]

- **Natural language**: Roll hold mode shall be the active mode whenever the autopilot is engaged and no other lateral mode is active.
- **FRET input**:
  in roll_hold mode RollAutopilot shall always satisfy `autopilot_engaged & no_other_lateral_mode`
  in roll_hold mode RollAutopilot shall always satisfy `roll_cmd = roll_hold_reference`
- **Scope**: Roll Autopilot
- **CoCoSpec formalization**:

```
var no_other_lateral_mode: bool = not HDGmode;
mode roll_hold_mode (
    require autopilot_engaged and no_other_lateral_mode;
    ensure roll_cmd = roll_hold_reference;
);
```

- **Verification result**: Valid, see table 10.4.

[**AP-003**]

- **Natural language**:
  AP-003a: When roll hold mode becomes the active mode the roll hold reference shall be set to the actual roll attitude of the aircraft, **except under the following conditions**:
  - **AP-003b:** The roll hold reference shall be set to zero if the actual roll angle is less than 6 degrees, in either direction, at the time of roll hold engagement.
  - **AP-003c:** The roll hold reference shall be set to 30 degrees in the same direction as the actual roll angle if the actual roll angle is greater than 30 degrees at the time of roll hold engagement.
  - **AP-003d:** The roll reference shall be set to the cockpit turn knob command, up to a 30 degree limit, if the turn knob is commanding 3 degrees or more in either direction.

- **FRET input**: Let us write first requirements `AP-003b`, `AP-003c`, and `AP-003d`, since requirement `AP-003a` depends on them.
  `AP-003b`: RollHoldReference shall always satisfy Cb $\implies$ roll_hold_reference = 0.0, where
  Cb = roll_angle < 6.0 & roll_angle > -6.0 & at_time_of_roll_hold_engagement.

  Alternatively, we can specify `AP-003b` as follows:
  in roll_hold mode RollHoldReference shall immediately satisfy (roll_angle < 6.0 & roll_angle > -6.0) $\implies$ roll_hold_reference = 0.0. Notice that we express the *at the time of roll hold engagement* constraint with the `immediately` FRET timing constraint.

  `AP-003c`: in roll_hold mode RollHoldReference shall always satisfy Cc $\implies$ roll_hold_reference = 30.0*Sign(roll_angle), where Cc = abs_real(roll_angle) $\geq$ 30.0 & at_time_of_roll_hold_engagement, where `Sign` returns the sign of a real number, defined in CoCoSpec as follows:

```
node  Sign(x : real;) returns(y : real;);
let y = if (x >= 0.0) then 1.0  else (- 1.0); tel
```

  Alternatively, we can specify `AP-003c` as follows:
  in roll_hold mode RollHoldReference shall immediately satisfy abs_real(roll_angle) $\geq$ 30.0 $\implies$ roll_hold_reference = 30.0*Sign(roll_angle)

  `AP-003d`: RollHoldReference shall always satisfy Cd $\implies$ roll_hold_reference = TurnKnob, where `Cd =` (TurnKnob $\geq$ 3.0 | TurnKnob $\leq$ -3.0) & (TurnKnob $\leq$ 30.0 | TurnKnob $\geq$ -30.0))

  `AP-003a`: RollHoldReference shall always satisfy (Ca) $\implies$ roll_hold_reference = roll_angle, where Ca = !(Cb | Cc1 | Cc2 | Cd) & at_time_of_roll_hold_engagement
  Alternatively, we can specify `AP-003a` as follows:
  in roll_hold mode RollHoldReference shall immediately satisfy !(Cb* | Cc1* | Cc2* | Cd) $\implies$ roll_hold_reference = roll_angle

- **Scope**: Roll Hold Reference

- **CoCoSpec formalization**:

```
var at_time_of_roll_hold_engagement : bool = false ->  not pre
   autopilot_engaged and  autopilot_engaged;
var Cb : bool = (roll_angle > -6.0)  and  (roll_angle < 6.0) and
    at_time_of_roll_hold_engagement;
var Cc: bool = (abs_real(roll_angle) >= 30.0) and
   at_time_of_roll_hold_engagement;
var Cd : bool = (TurnKnob >= 3.0 or TurnKnob <= -3.0) and (
   TurnKnob <= 30.0 or TurnKnob >= -30.0);
var Ca : bool = not ( Cb or Cc1 or Cc2 or Cd) and
   at_time_of_roll_hold_engagement;

mode roll_hold_mode (
    -- require condition is given by [AP-002]
    ensure "AP-003a" Ca => roll_hold_reference = roll_angle;
    ensure "AP-003b" Cb => roll_hold_reference = 0.0;
    ensure "AP-003c" Cc => roll_hold_reference = 30.0*Sign(
   roll_angle);
    ensure "AP-003d" Cd => roll_hold_reference = TurnKnob;
);
```

Alternatively, if we use `immediately` instead of `always`, we do not need to implicitly define the `at_time_of_roll_hold_engagement` temporal constraint. The generated code for properties `AP-003b` and `AP-003d` looks as follows:

```
var Cb* : bool = (roll_angle > -6.0)  and  (roll_angle < 6.0);
var Cd* : bool = (TurnKnob >= 3.0 or TurnKnob <= -3.0) and (
   TurnKnob <= 30.0 or TurnKnob >= -30.0);

guarantee "AP-003b" (H(((roll_hold_mode and (pre (  not
   roll_hold_mode ))) and ( not FTP)) => (pre (S( (((( not
   roll_hold_mode) and (FTP or (pre ( roll_hold_mode )))) => (
   not Cb* => roll_hold_reference = 0.0)) and (( not
   roll_hold_mode) and (FTP or (pre ( roll_hold_mode ))))), (((
   not roll_hold_mode) and (FTP or (pre ( roll_hold_mode )))) =>
    ( not Cb* => roll_hold_reference = 0.0)) )))) and ((S( ((
   not (roll_hold_mode and (pre (  not roll_hold_mode )))) and
   (( not roll_hold_mode) and (FTP or (pre ( roll_hold_mode ))))
   ), ( not (roll_hold_mode and (pre (  not roll_hold_mode ))))
   )) => (S( ((( not roll_hold_mode) and (FTP or (pre (
   roll_hold_mode )))) => ( not Cb* => roll_hold_reference =
   0.0)) and (( not roll_hold_mode) and (FTP or (pre (
   roll_hold_mode ))))), ((( not roll_hold_mode) and (FTP or (
   pre ( roll_hold_mode )))) => ( not Cb* => roll_hold_reference
    = 0.0)) )));

guarantee "AP-003d" Cd => roll_hold_reference = TurnKnob;
```

- **Verification result**:
  - **AP-003a:** Invalid. Kind2 returned the counter-example shown in Table 10.5. From the counterexample we see that at the time of roll hold engagement (T=0.025, autopilot_engaged is true and HDGmode is false) the inputs satisfy none of the

Table 10.5: Counter example of requirement `[AP-003a]`

Inputs	T = 0	T = 0.025
roll_angle	-23.0	18.0
autopilot_engaged	false	true
HDGmode	true	false
TurnKnob	-3.0	0.0
Outputs		
roll_hold_reference	-3.0	-23.0

Table 10.6: Counter example of requirement [AP-003aV2]

Inputs	T = 0	T = 0.025
roll_angle	-1.0	-6.0
autopilot_engaged	false	true
HDGmode	false	false
TurnKnob	3.0	-1.0
Outputs		
roll_hold_reference	3.0	0.0

other three cases: `roll_angle` is equal to 6 therefore first and second conditions (AP-003b and AP-003c) are not met, also the `TurnKnob` (turn knob command) is less than 3 degrees, therefore condition AP-003d is not met either. Thus, output `roll_hold_reference` should be equal to the actual roll attitude `roll_angle`. However, the `roll_hold_reference` is equal to the previous value of `roll_angle` .

We updated the requirement to check if this is always the case, i.e., that `roll_hold_reference` is equal to the previous value of `roll_angle` at the time of roll hold engagement. The second version of the requirement is as follows:

`AP-003aV2`: RollHoldReference shall always satisfy `(Ca) => roll_hold_reference = pre_roll_angle`, where `Ca = not(Cb or Cc1 or Cc2 or Cd) and at_time_of_roll_hold_e` and `pre_roll_angle = 0.0 -> pre roll_angle;`

```
var pre_roll_angle : real = 0.0 -> pre roll_angle;
mode roll_hold_mode(
    require no_other_lateral_mode and autopilot_engaged;
    ensure "AP-003aV2" Ca => roll_hold_reference =
   pre_roll_angle;
)
```

We analyzed the second version and got the counterexample shown in Table 10.6. `roll_hold_reference` is equal to zero but the previous value of `roll_angle` is not equal to zero neither `turnKnob`. We noticed that `roll_angle` is equal to -1 in $T = 0$, in which case condition Cb (`abs_real(roll_angle) < 6.0`) is satisfied. Thus, we modified the previous requirement in order to hold condition Ca in the previous step as well.

```
var pre_roll_angle : real = 0.0 -> pre roll_angle;
var pre_Ca : bool = false -> pre Ca;
mode roll_hold_mode(
    require no_other_lateral_mode and autopilot_engaged;
```

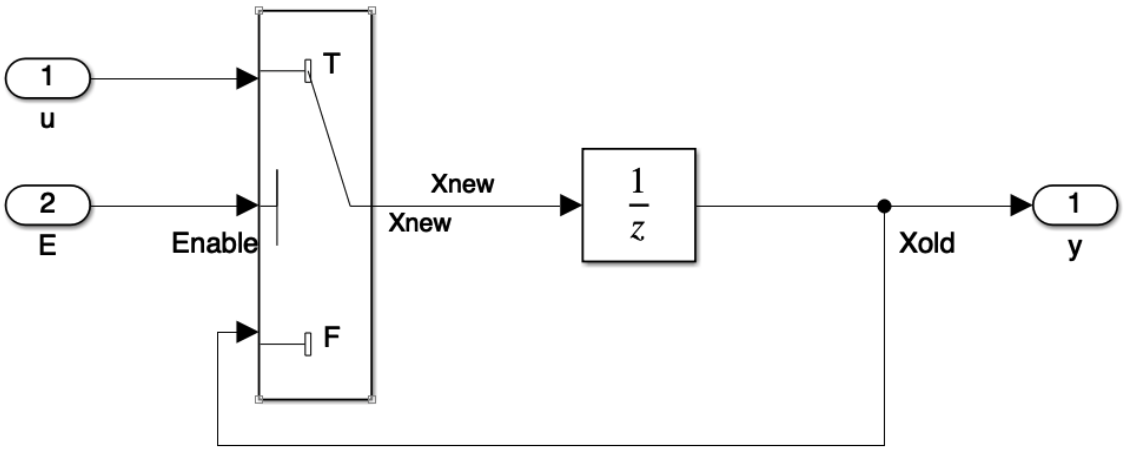


Figure 10.2: Simulink code responsible for holding the last roll angle before engagement on roll hold mode.

Table 10.7: Counter example of requirement [AP-003b]

Inputs	T = 0	T = 0.025
roll_angle	0.0	0.0
autopilot_engaged	false	true
HDGmode	true	false
TurnKnob	3.0	22.0
Outputs		
roll_hold_reference	3.0	22.0

```
    ensure "AP-003aV3" Ca and pre_Ca => roll_hold_reference
   = pre_roll_angle;
)
```

This last property [AP-003V3] is proven **valid**, see Table 10.4. In order to understand why the output is based on the previous value of roll angle, we looked at the model and we found that indeed the reference roll command of a previous timestep is held rather than the command at the time of engagement. Figure 10.2 illustrates the Simulink code responsible for holding the `roll_angle`, where u is the roll angle and E is the condition 'not engaged in hold mode'. If E is true, output y is equal to the previous value of roll angle. Once the roll hold mode becomes active (E is false), the value of output y is equal to `pre y`; y holds this value while roll hold mode is active. Thus, the component holds the value of the roll angle just before the activation of roll hold mode and not 'at the time of activation'. We believe that the requirement is not satisfied due to an erroneous model.

– **AP-003b:** Invalid: From Counterexample in Table. 10.7 we noticed that `roll_hold_reference` is equal to `TurnKnob` even if the condition `(abs_real(Phi) < 6.0) and at_time_of_roll_hold_e` is met. The model confirms that requirement [AP-003d] is stronger than [AP-003b], i.e., if the condition of requirement [AP-003d] is met (`abs_real(TurnKnob ) >= 3.0`) the output is equal to `TurnKnob`. We updated requirement [AP-003b] by adding condition "`abs_real(TurnKnob) < 3.0`". This version was falsified

Table 10.8: Counter example of requirement [AP-003bv2]

Inputs	T = 0	T = 0.025
roll_angle	-22.0	0.0
autopilot_engaged	false	true
HDGmode	true	false
TurnKnob	-3.0	0.0
Outputs		
roll_hold_reference	-3.0	-22.0

Table 10.9: Counter example of requirement [AP-003c]

Inputs	T = 0	T = 0.025
roll_angle	0.0	-90.0
autopilot_engaged	false	true
HDGmode	true	false
TurnKnob	3.0	0.0
Outputs		
roll_hold_reference	3.0	0.0

again (see Table 10.8) (same reason as in [AP-0003a]: the output value is the roll angle from the previous step).

- **AP-003c:** Invalid: From Counterexample in Table 10.9 we notice requirement gets falsified for the same reason as for [AP-0003a]: the output value is the roll angle from the previous step.
- **AP-003d:** Valid.

[AP-004]

- **Natural language**:
  - **AP-004a:** Steady state roll commands shall be tracked within 1 degree in calm air.
  - **AP-004b:** Response to roll step commands shall not exceed 10% overshoot in calm air.
  - **AP-004c:** Small signal (<3 degree) roll bandwidth shall be at least 0.5 rad/sec.
- **FRET input**:
  `AP-004a`: when in roll_hold mode, when steady_state Autopilot shall always satisfy abs_roll_err $\leq$ 1.0
  `AP-004b`: when in roll_hold mode Autopilot shall always satisfy overshoot $\leq$ 0.1
  `AP-004c`: We could not formalize it.
- **Scope**: Global
- **CoCoSpec formalization**: This requirement requires the definition of what is a steady state and an overshoot. After discussing with a domain expert, we formalized overshoot such that it is measured with respect to the issued roll step command. This means that the requirement is valid for the Roll Hold mode when a step signal is issued for the aircraft to track. The step signal is defined as the difference between

the roll command and roll angle at the time of issuing the roll hold command. $OS = (measurement - Step)/Step$. In order to ensure that the requirement is satisfied, it is enough to ensure that the roll measurement never exceeds the 10% overshoot level for the currently issued command (when a new one is issued, the percent overshoot expression will be regenerated based on the new step command).

Additionally, after discussing with a domain expert, we defined steady state as follows: `roll_measured - roll_command <= epsilon`.

```
assume calm_air; --this assumption is used in the Simulink model
   , the wind speed is zero.
var roll_err: real = roll_angle - roll_cmd;
var epsilon: real = 0.0001;
var steady_state: bool =false -> abs(roll_err - pre roll_err) <=
    epsilon;
var initial_roll_cmd: real = roll_cmd -> pre initial_roll_cmd;
var overshoot : real = (roll_angle - step)/step;
var step: real = roll_cmd - rollAtZero;
var rollHoldMode : bool = autopilot_engaged and not HDGmode;
var rollAtZero: real = roll_angle -> if (rollHoldMode and not
   pre rollHoldMode) then roll_angle else pre rollAtZero;
mode roll_hold_mode(
    ensure "AP-004a" steady_state => abs_real(roll_err) <= 1.0;
    ensure "AP-004b" overshoot <= 0.1;
);
```

- **Verification result**: Unsupported by Kind2, Undecided by SLDV, see table 10.4.

[**AP-005**]

- **Natural language**: The maximum roll rate for large commands shall be 6 deg/sec +/-10% in calm air.
- **FRET input**: Autopilot shall always satisfy abs_roll_rate < 6.6
- **Scope**: Global
- **CoCoSpec formalization**:

```
var abs_roll_rate: real = abs_real(roll_rate);
guarantee "AP-005" abs_roll_rate < 6.6 and FTP;
```

- **Verification result**: Unsupported by Kind2, Undecided by SLDV, see table 10.4.

[**AP-006**]

- **Natural language**: The maximum roll angle allowed shall be 30 deg +/-10% in calm air.
- **FRET input**: Autopilot shall always satisfy abs_roll_angle < 33.0
- **Scope**: Global
- **CoCoSpec formalization**:

```
var abs_roll_angle: real = abs_real(roll_angle);
guarantee "AP-006" abs_roll_angle < 33.0;
```

Verification result: Unsupported by Kind2, Undecided by SLDV, see table 10.4.

[**AP-007**]

- **Natural language**: The maximum aileron command allowed shall be 15 deg.
- **FRET input**: RollAutopilot shall always satisfy abs_aileron_cmd $\leq$ 15.0
- **Scope**: RollAutopilot
- **CoCoSpec formalization**:

```
var abs_aileron_cmd: real = abs_real(aileron_cmd);
guarantee "AP-007" abs_aileron_cmd <= 15.0;
```

Verification result: Valid, see table 10.4.

[**AP-008**]

- **Natural language**: Heading Hold shall become the active mode when the pilot selects the heading switch in the cockpit and deactivate when the switch is deselected.
- **FRET input**:
  in hdg_hold mode RollAutopilot shall always satisfy hdg_mode_is_active
  in hdg_hold mode RollAutopilot shall always satisfy roll_cmd = hdg_hold_mode_cmd
- **Scope**: Roll Autopilot
- **CoCoSpec formalization**:

```
var hdg_mode_is_active: bool = HDGmode;
mode hdg_mode (
    require hdg_mode_is_active;
    ensure roll_cmd = hdg_hold_mode_cmd;
);
```

- **Verification result**: Valid, see table 10.4.

[**AP-009**]

- **Natural language**: When heading hold mode becomes the active mode the heading hold reference shall be set by the pilot via a cockpit control.
- **Comment**: This requirement is out of scope of the given Simulink model.

[**AP-010**]

- **Natural language**:
  - **AP-010a:** Steady state heading commands shall be tracked within 1 degree in calm air.
  - **AP-010b:** Response to heading step commands shall not exceed 10% overshoot in calm air.
- **FRET input**:
  `AP-0010a`: when in hdg mode, when hdg_steady_state Autopilot shall always satisfy abs_hdg_err $\leq$ 1.0
  `AP-0010b`: when in hdg mode Autopilot shall always satisfy overshoot $\leq$ 0.1
- **Scope**: Global
- **CoCoSpec formalization**:

```
var hdg_err: real = Psi - HDGmodeCmd;
var hdg_epsilon: real = 0.0001;
var hdg_steady_state: bool =false -> abs_real(hdg_err - pre
   hdg_err) <= hdg_epsilon;
var initial_hdg_cmd: real = HDGmodeCmd -> pre initial_hdg_cmd;
var hdg_overshoot : real = (Psi - hdg_step)/hdg_step;
var hdg_step: real = HDGmodeCmd - psiAtZero;
var psiAtZero: real = Psi -> if (HDGmode and not pre HDGmode)
   then Psi else pre psiAtZero;
mode hdg_mode (
    require HDGmode;
    ensure "AP-010a" hdg_steady_state => abs_real(hdg_err) <=
   1.0;
    ensure "AP-010b"  hdg_overshoot <= 0.1;
);
```

- **Verification result**: Unsupported by Kind2, Undecided by SLDV, see table 10.4.

Chapter 11

System Wide Integrity Monitor (SWIM)

This challenge problem describes a safety algorithm for monitoring airspeed in the SWIM (System Wide Integrity Monitor) suite in order to provide warning to an operator when the vehicle speed is approaching a boundary where an evasive flyup maneuver cannot be achieved.

[t] Next, we provide 1) the SWIM natural language requirements, 2) their FRETish form, 3) the CoCoSpec equivalent code and 4) the verification results. The verification results are summarized in Table 11.4. The model has Sqrt block that is not supported by Simulink Design Verifier. They were abstracted during the analysis. This can lead Simulink Design Verifier to produce only partial results for parts of the model that depend on the output values of these blocks.

[SWIM-001]

- **Natural language**: The SWIM Airspeed algorithm shall output the minimum AGCAS airspeed required to perform a 2g flyup as follows:
  IF CAT I, Auto GCAS Minimum Vcas (knots) = 1.25921 * SQRT(Gross Weight) + 10.0
  IF CAT III, Auto GCAS Minimum Vcas (knots) = 1.33694 * SQRT(Gross Weight) + 10.0

- **FRET input**:

  - `SWIM-001a`: SWIM shall always satisfy (storeCat = CAT1) ⇒ (CalAirspeedMin = 1.25921 * sqrtOfWeight + 10.0)
  - `SWIM-001b`: SWIM shall always satisfy (storeCat = CAT3) ⇒ (CalAirspeedMin = 1.33694 * sqrtOfWeight + 10.0)

- **Scope**: Global

- **CoCoSpec formalization**:

```
var CAT1: int = 0;
var CAT3: int = 1;
var sqrtOfWeight: real = sqrt(weight);
```

Table 11.1: SWIM Inputs

Input Scope	Name	Type	Description
Global	muxGet_T_Mux_AircraftGrossWeight_lbs	Double	Aircraft gross weight
Global	diGet_S_CatSwitchPosition	Integer	Store Category of vehicle (I or III)
Global	E_AI_IMPACT_PRESSURE	Double	Air Data Impact Pressure
Global	diGet_S_LandingGearAltFlap	Boolean	Powered Approach Discrete
Global	muxGet_T_Mux_AirspeedMonitorEnable	Boolean	Indicates the aircraft is at an altitude when the airspeed monitor can be enabled
Global	cvGet_V_AgcasLowSpeedWarn	Boolean	Low speed warning indication from the automated ground collision avoidance system
Global	olcGet_AgcasInterlocks	Boolean	Indicates state of automated ground collision avoidance system (Notfailed or Failed)

```
guarantee "SWIM-001A" ( storeCat = CAT1 ) ⇒ ( CalAirspeedMin =
   1.25921 * sqrtOfWeight + 10.0 );
guarantee "SWIM-001B" ( storeCat = CAT3 ) ⇒ ( CalAirspeedMin =
   1.33694 * sqrtOfWeight + 10.0 );
```

We assume the weight is always positive.

```
assume weight > 0.0;
```

- **Verification result**: In this requirement, *Sqrt* function is used. Lustre does not provide built-in support for mathematical functions such as sqrt, and thus, an abstraction must be provided. CoCoSim provides a library of Mathematical functions that can be used as an abstraction in Lustre. The following is an abstraction used by CoCoSim for *sqrt* function:

```
node sqrt (x: real) returns (y: real) ;
let
    assert (x >= 0.0);
    assert (y >= 0.0);
    assert (x > 1.0 ) = (x > y);
    assert (x < 1.0 ) = (y > x);
    assert (x = 1.0 ) = (y = 1.0);
    -- Use one of these abstractions based on the complexity of
   the model
    -- Abstraction 1: using Lookup table of some values of SQRT
    assert x <= 10000.0 => y = sqrt_lookup_0_10000(x);
    assert x >  10000.0 => y > 100.0;
    -- Abstraction 2: using the mathematical definition of SQRT
    assert ((y * y) = x);
```

Table 11.2: SWIM Outputs

Output Scope	Name	Type	Description
Global	swimGet_AgcasLowSpeedWarn	Boolean	Low speed warning detection
Global	swimGet_QcMinDisable_lbspft2	Double	Impact pressure value corresponding to 20 knots slower than warning airspeed threshold in order to disable ongoing warnings.
Global	SWIM_Qcmin_lbspft2	Double	Warning trigger of minimum impact pressure in which a safe AGCAS evasive maneuver can be accomplished
Global	SWIM_CalAirspeedmin_kts	Double	Warning trigger corresponding minimum airspeed in which a safe AGCAS evasive maneuver can be accomplished
Global	SWIM_ASWarningAllowed	Boolean	Low speed warning allowed

Table 11.3: FRET to Model Variables mapping for SWIM

FRET name	Scope	Simulink name
weight	Global	muxGet_T_Mux_AircraftGrossWeight_lbs
storeCat	Global	diGet_S_CatSwitchPosition
vehAirPress	Global	E_AI_IMPACT_PRESSURE
CalAirspeedMin	Global	SWIM_CalAirspeedmin_kts
lowSpeedWarningAllowed	Global	SWIM_ASWarningAllowed
swimGet_AgcasLowSpeedWarn	Global	swimGet_AgcasLowSpeedWarn
warningTrigForMinPress	Global	SWIM_Qcmin_lbspft2

Table 11.4: SWIM Verification Results with Kind2 (abbr. by K) and SLDV (abbr. by S)

Requirement	K Result	K Time	S Result	S Time
[SWIM-001A]	Valid	2.482 sec	Undecided (Due to Stubbing)	14 sec
[SWIM-001B]	Valid	2.482 sec	Undecided (Due to Stubbing)	17 sec
[SWIM-002]	Invalid	2.193 sec	Invalid	17 sec
	CoCoSim total time: 24.95 sec		SLDV total time: 18 sec	

```
tel
```

When we use less precise abstraction (Abstraction 1) we got counterexample where

the `sqrt(3864) = 67` and not 62.1611. So both requirements were falsified because of less precise abstraction of sqrt (the default one used by CoCoSim). We changed the abstraction to use more precise abstraction of sqrt (Abstraction 2) and Kind2 was able to prove both 'SWIM-001A' and 'SWIM-001B' valid (see Table 11.4).

[**SWIM-002**]

- **Natural language**: When a low speed warning is allowed, as computed by the SWIM Airspeed algorithm, a low speed warning shall be true when the vehicle air data impact pressure is less than the warning trigger for minimum impact pressure in which a safe AGCAS evasive maneuver can be accomplished where:

  The warning trigger for minimum impact pressure in which a safe AGCAS evasive maneuver can be accomplished is computed as
  SWIM_Qcmin_lbspft2 = [(-2.0906 + 0.020306*Auto GCAS Minimum Vcas) + 0.1] * (70.7184 (lbspft2/in Hg))

- **FRET input**: SWIM shall always satisfy (lowSpeedWarningAllowed = 1.0 & vehAirPress < warningTrigForMinPress) ⇒ lowSpeedWarningTrue

- **Scope**: Global

- **CoCoSpec formalization**:

```
var E_SWIM_WARNING_ON:real = 1.0;
var lowSpeedWarningTrue:bool = swimGet_AgcasLowSpeedWarn =
   E_SWIM_WARNING_ON;
guarantee "SWIM-002" ( lowSpeedWarningAllowed = 1.0 and
   vehAirPress < warningTrigForMinPress ) => lowSpeedWarningTrue
   ;
```

- **Verification result**: Invalid, counterexample is described in table 11.5. The output `swimGet_AgcasLowSpeedWarn` is not activated at second time step even if `lowSpeedWarningAllowed` is active and `vehAirPress` is less than `warningTrigForMinPress`. When checking the model it appears that the condition of activating `lowSpeedWarningTrue` used in the Simulink model only holds when the following condition is true

```
    (muxGet_T_Mux_AirspeedMonitorEnable == 1.0) and (
   cvGet_V_AgcasLowSpeedWarn == 1.0) and (olcGet_AgcasInterlocks
    == 0.0)  and lowSpeedWarningTrue
```

  This means that the aircraft is 1) at an altitude when the airspeed monitor can be enabled, 2) low speed warning from the automated ground collision avoidance system is active, 3) the automated ground collision avoidance system does not fail, and 4) low speed warning is allowed. These conditions are clearly missing from the given requirement text.

Table 11.5: Counter example for requirement [SWIM-002]

Inputs	First Step	Second Step
muxGet_T_Mux_AircraftGrossWeight_lbs	0.25	1
diGet_S_CatSwitchPosition	-1073741824	-1073741824
E_AI_IMPACT_PRESSURE	-116	-126.0
diGet_S_LandingGearAltFlap	1	1
muxGet_T_Mux_AirspeedMonitorEnable	0	0
cvGet_V_AgcasLowSpeedWarn	0	0
olcGet_AgcasInterlocks	0	0
Outputs		
swimGet_AgcasLowSpeedWarn	0.0	0.0
SWIM_Qcmin_lbspft2	-125.452	-124.4921
SWIM_ASWarningAllowed	1.0	1.0

Chapter 12

Euler Transformation

This component creates a Rotation Matrix describing a 321 rotation about the z-axis, y-axis, and finally x-axis of an Inertial frame in Euclidean space, and, given an Input Vector, outputs the representation of the Input Vector in the new rotated frame. Note the y-axis and x-axis subsequent rotations are the second and third transform operations, identified in the 321 operation by the 2 and 3 label description, respectively. The latter two operations are rotations about the intermediate frames. The computed Rotation Matrix Shall Equal a Special Orthogonal(3) $3x3$ with the following properties: (i) the rows and columns of the Output Matrix shall be orthonormal; (ii) the Output Matrix multiplied with the Transpose of the Output Matrix shall be the Identity ($3x3$) Matrix; and (iii) the determinant of the Output Matrix shall be equal to 1. Tables 12.1 and 12.2 list the inputs and outputs of the Euler Transformation component, respectively.

Definitions Euler3 Roll rotation:

$$R_3(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & sin(\phi) \\ 0 & -sin(\phi) & cos(\phi) \end{bmatrix}$$

Euler3 Pitch rotation:

$$R_2(\theta) = \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{bmatrix}$$

Euler3 Heading rotation:

$$R_1(\psi) = \begin{bmatrix} cos(\psi) & sin(\psi) & 0 \\ -sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Abstracted blocks

The model has trigonometric function (cos and sin) blocks that are not supported by both SLDV and Kind2. Thus, they were abstracted during the analysis. However, because of the abstraction both SLDV and Kind2 produced non-sensical counterexamples.

`cos` is abstracted by restricting the output to the [-1, 1] interval:

```
node imported cos (x: real) returns (y: real) ;
(*@contract
```

Table 12.1: Euler Transformation Inputs

Input Scope	Name	Type	Description
Global	phi	Double	Roll angle [rad]
Global	theta	Double	Pitch angle [rad]
Global	psi	Double	Yaw angle [rad]
Global	Vi	Double	3x1 Inertial Vector

Table 12.2: Euler Transformation Outputs

Output Scope	Name	Type	Description
Global	DCM321	Double	3x3 Rotation Matrix
Global	Vb	Double	3x1 Body Vector

```
      guarantee (y >= -1.0 and y <= 1.0);
*)
```

More precise abstraction can be provided, but requirements that compare two values involving such abstracted nodes could still get falsified.
Because of this abstraction, all the requirements of this challenge are either falsified or undecided due to the complexity of the model. For the invalid requirements, we can not judge their validity as the counterexample is non-sensical (e.g. $cos(0.0) = 0.0$).

Note on Inlining Matrices:

CoCoSim inlines matrices into scalars therefore some of the requirements must be decomposed into several sub-requirements. `DCM321_1_1` stands for $DCM321(1,1)$, `DCM321_2_1` stands for $DCM321(2,1)$ and so on.

Next, we provide 1) the Euler Transformation natural language requirements, 2) their FRETish form, 3) the CoCoSpec equivalent code and 4) the verification results. The verification results are summarized in Table 12.3.

[EUL-001]

- **Natural language**: The Rotation Matrix Output, DCM321, of this Function Shall Equal a 3x3 Matrix Product of a 3x3 Euler 3 (Roll) Rotation Matrix times a 3x3 Euler 2 (Pitch) Rotation Matrix times a 3x3 Euler 1 (Heading) Rotation Matrix.

- **FRET input**:
  EUL shall always satisfy $DCM321\_1\_1 = cos(theta) * cos(psi)$
  EUL shall always satisfy $DCM321\_2\_1 = -cos(phi) * sin(psi) + sin(phi) * sin(theta) * cos(psi)$
  EUL shall always satisfy $DCM321\_3\_1 = sin(phi) * sin(psi) + cos(phi) * sin(theta) * cos(psi)$
  EUL shall always satisfy $DCM321\_1\_2 = cos(theta) * sin(psi)$
  EUL shall always satisfy $DCM321\_2\_2 = cos(phi) * cos(psi) + sin(phi) * sin(theta) * sin(psi)$
  EUL shall always satisfy $DCM321\_3\_2 = -sin(phi) * cos(psi) + cos(phi) * sin(theta) * sin(psi)$

Table 12.3: Euler Transformation Verification Results with Kind2 (abbr. by K) and SLDV (abbr. by S). In this use case, requirements were analyzed individually. When verified together, Kind2 gives Undecided with a timeout 2 hours.

Req.	K Result	K Time	S Result	S Time
[EUL-001]	Invalid (due to abstraction of cosine and sine)	0.874 sec	Undecided due to nonlinearities	30 sec
[EUL-002]	Valid	3.068 sec	Valid	19 sec
[EUL-003]	Invalid (due to abstraction of cosine and sine)	0.140 sec	Undecided due to nonlinearities	8 sec
[EUL-004]	Invalid (due to abstraction of cosine and sine)	0.185 sec	Undecided due to nonlinearities	22 sec
[EUL-005]	Unformalized			
[EUL-006]	Invalid (due to abstraction of cosine and sine)	2.443 sec	Undecided due to nonlinearities	20 sec
[EUL-007]	Invalid (due to abstraction of cosine and sine)	0.191 sec	Undecided due to nonlinearities	17 sec
[EUL-008]	Invalid (due to abstraction of cosine and sine)	0.152 sec	Undecided due to nonlinearities	26 sec
	Lustre generation time: 39.37sec			

EUL shall always satisfy $DCM321\_1\_3 = -sin(theta)$
EUL shall always satisfy $DCM321\_2\_3 = sin(phi) * cos(theta)$
EUL shall always satisfy $DCM321\_3\_3 = cos(phi) * cos(theta)$

- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "EUL-001.1" S( DCM321_1_1 = cos(theta)*cos(psi) and
   FTP, DCM321_1_1 = cos(theta)*cos(psi) );
guarantee "EUL-001.2" S( DCM321_2_1 = -cos(phi)*sin(psi) + sin(
   phi)*sin(theta)*cos(psi) and FTP, DCM321_2_1 = -cos(phi)*sin(
   psi) + sin(phi)*sin(theta)*cos(psi) );
guarantee "EUL-001.3" S( DCM321_3_1 = sin(phi)*sin(psi) + cos(
   phi)*sin(theta)*cos(psi) and FTP, DCM321_3_1 = sin(phi)*sin(
   psi) + cos(phi)*sin(theta)*cos(psi) );
guarantee "EUL-001.4" S( DCM321_1_2 = cos(theta)*sin(psi) and
   FTP, DCM321_1_2 = cos(theta)*sin(psi) );
guarantee "EUL-001.5" S( DCM321_2_2 = cos(phi)*cos(psi) + sin(
   phi)*sin(theta)*sin(psi) and FTP, DCM321_2_2 = cos(phi)*cos(
   psi) + sin(phi)*sin(theta)*sin(psi) );
guarantee "EUL-001.6" S( DCM321_3_2 = -sin(phi)*cos(psi) + cos(
   phi)*sin(theta)*sin(psi) and FTP, DCM321_3_2 = -sin(phi)*cos(
   psi) + cos(phi)*sin(theta)*sin(psi) );
guarantee "EUL-001.7" S( DCM321_1_3 = -sin(theta) and FTP,
   DCM321_1_3 = -sin(theta) );
guarantee "EUL-001.8" S( DCM321_2_3 = sin(phi)*cos(theta) and
   FTP, DCM321_2_3 = sin(phi)*cos(theta) );
guarantee "EUL-001.9" S( DCM321_3_3 = cos(phi)*cos(theta) and
   FTP, DCM321_3_3 = cos(phi)*cos(theta) );
```

- **Verification result**: Invalid using Kind2 because of abstraction of cosine and sine functions. `cos` and `sin` are abstracted by restricting the output to the [-1, 1] interval. The counterexample is non-sensical (e.g. $cos(0.0) = 0.0$)

[**EUL-002**]

- **Natural language**: The Body Vector Output, Vb, of this Function Shall Equal a 3x1 Vector Product of the 3x3 Rotation Matrix Output, DCM321, times the Input Inertial Vector, Vi.
- **FRET input**:
  EUL shall always satisfy $Vb\_1 = DCM321\_1\_1 * Vi\_1 + DCM321\_1\_2 * Vi\_2 + DCM321\_1\_3 * Vi\_3$
  EUL shall always satisfy $Vb\_2 = DCM321\_2\_1 * Vi\_1 + DCM321\_2\_2 * Vi\_2 + DCM321\_2\_3 * Vi\_3$
  EUL shall always satisfy $Vb\_3 = DCM321\_3\_1 * Vi\_1 + DCM321\_3\_2 * Vi\_2 + DCM321\_3\_3 * Vi\_3$
- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "EUL-002.1" S( (Vb_1 = DCM321_1_1 * Vi_1 + DCM321_1_2
   * Vi_2 + DCM321_1_3 * Vi_3) and FTP, (Vb_1 = DCM321_1_1 *
   Vi_1 + DCM321_1_2 * Vi_2 + DCM321_1_3 * Vi_3));
guarantee "EUL-002.2" S( (Vb_2 = DCM321_2_1 * Vi_1 + DCM321_2_2
   * Vi_2 + DCM321_2_3 * Vi_3) and FTP, (Vb_2 = DCM321_2_1 *
   Vi_1 + DCM321_2_2 * Vi_2 + DCM321_2_3 * Vi_3));
guarantee "EUL-002.3" S( (Vb_3 = DCM321_3_1 * Vi_1 + DCM321_3_2
   * Vi_2 + DCM321_3_3 * Vi_3) and FTP, (Vb_3 = DCM321_3_1 *
   Vi_1 + DCM321_3_2 * Vi_2 + DCM321_3_3 * Vi_3));
```

- **Verification result**: "EUL-002.1", "EUL-002.2" and "EUL-002.3" are proved valid by Kind2.

[**EUL-003**]

- **Natural language**: The magnitude of the Body Vector Output, Vb, shall equal the magnitude of the Input Inertial Vector, Vi.
- **FRET input**:
  EUL shall always satisfy $Vb\_1 * Vb\_1 + Vb\_2 * Vb\_2 + Vb\_3 * Vb\_3 = Vi\_1 * Vi\_1 + Vi\_2 * Vi\_2 + Vi\_3 * Vi\_3$
- **Scope**: Global
- **CoCoSpec formalization**:

```
guarantee "EUL-003" S( (Vb_1*Vb_1 + Vb_2*Vb_2 + Vb_3*Vb_3 = Vi_1
   *Vi_1 + Vi_2*Vi_2 + Vi_3*Vi_3) and FTP, (Vb_1*Vb_1 + Vb_2*
   Vb_2 + Vb_3*Vb_3 = Vi_1*Vi_1 + Vi_2*Vi_2 + Vi_3*Vi_3));
```

- **Verification result**: Invalid using Kind2 because of abstraction of cosine and sine functions.

[**EUL-004**]

- **Natural language**: The Rotation Matrix, DCM321, shall be invertible with the exception of the case where $theta = +/- pi/2$ radians.
- **FRET input**: EUL shall always satisfy $abs(theta)! = pi/2.0 => DCM\_det! = 0.0$
- **Scope**: Global
- **CoCoSpec formalization**:

```
var pi:real = 3.1415926536;
var DCM_det :real = det_3x3(DCM321_1_1, DCM321_2_1, DCM321_3_1,
   DCM321_1_2, DCM321_2_2, DCM321_3_2, DCM321_1_3, DCM321_2_3,
   DCM321_3_3);
guarantee "EUL-004" S( (abs(theta) <> pi/2.0 =>  DCM_det <> 0.0)
    and FTP, (abs(theta) <> pi/2.0 =>  DCM_det <> 0.0));
```

  where *abs* is the absolute value node and *det_3x3* is the determinant of a 3x3 matrix:

```
node  det_3x3(a11 : real;
 a21 : real;
 a31 : real;
 a12 : real;
 a22 : real;
 a32 : real;
 a13 : real;
 a23 : real;
 a33 : real;)
returns(det : real;);
var
 adj11 : real;
 adj21 : real;
 adj31 : real;
let
 det = (((a11 * adj11) + (a12 * adj21)) + (a13 * adj31));
 adj11 = ((a22 * a33) - (a23 * a32));
 adj21 = ((a23 * a31) - (a21 * a33));
 adj31 = ((a21 * a32) - (a31 * a22));
tel
```

  This requirement requires an exact value of *pi*, which we give an approximation up to 10 digits. This requirement will get falsified not because of the precision of *pi* but because of the abstraction of *cos* and *sin* as explained above.

- **Verification result**: Invalid using Kind2 because of abstraction of cosine and sine functions.

[**EUL-005**]

- **Natural language**: The Rotation Matrix, DCM321, shall provide a distinct mapping from the input vector, Vi, to the output vector, Vb, for each pitch angle, theta. Note: the DCM321 is not distinct for all phi and psi inputs.
- **Formalization**: We could not formalize it.
- **Verification result**:

[**EUL-006**]

- **Natural language**: The rows and columns of the Rotation Matrix, DCM321, shall be orthonormal. For instance, denoting r1 as row 1 and r2 as row 2 of DCM321, $< r1, r2 >= r1 * r2' = 0$ and $< r1, r1 >= r1 * r1' = 1$. Likewise, with c1 as column 1 and c2 as column 2 of DCM321, $< c1, c2 >= c1 * c2' = 0$ and $< c1, c1 >= c1 * c1' = 1$.

- **FRET input**:
  EUL shall always satisfy $r1xr1Transpose = 1.0$ & $r2xr2Transpose = 1.0$ & $r3xr3Transpose = 1.0$
  EUL shall always satisfy $r1xr2Transpose = 0.0$ & $r1xr3Transpose = 0.0$ & $r2xr3Transpose = 0.0$
  EUL shall always satisfy $c1xc1Transpose = 1.0$ & $c2xc2Transpose = 1.0$ & $c3xc3Transpose = 1.0$
  EUL shall always satisfy $c1xc2Transpose = 0.0$ & $c1xc3Transpose = 0.0$ & $c2xc3Transpose = 0.0$

- **Scope**: Global

- **CoCoSpec formalization**:

```
var r1xr1Transpose : real = DCM321_1_1 * DCM321_1_1 + DCM321_1_2
    * DCM321_1_2 + DCM321_1_3 * DCM321_1_3;
var r2xr2Transpose : real = DCM321_2_1 * DCM321_2_1 + DCM321_2_2
    * DCM321_2_2 + DCM321_2_3 * DCM321_2_3;
var r3xr3Transpose : real = DCM321_3_1 * DCM321_3_1 + DCM321_3_2
    * DCM321_3_2 + DCM321_3_3 * DCM321_3_3;

var r1xr2Transpose : real = DCM321_1_1 * DCM321_2_1 + DCM321_1_2
    * DCM321_2_2 + DCM321_1_3 * DCM321_2_3;
var r1xr3Transpose : real = DCM321_1_1 * DCM321_3_1 + DCM321_1_2
    * DCM321_3_2 + DCM321_1_3 * DCM321_3_3;
var r2xr3Transpose : real = DCM321_2_1 * DCM321_3_1 + DCM321_2_2
    * DCM321_3_2 + DCM321_2_3 * DCM321_3_3;

var c1xc1Transpose : real = DCM321_1_1 * DCM321_1_1 + DCM321_2_1
    * DCM321_2_1 + DCM321_3_1 * DCM321_3_1;
var c2xc2Transpose : real = DCM321_1_2 * DCM321_1_2 + DCM321_2_2
    * DCM321_2_2 + DCM321_3_2 * DCM321_3_2;
var c3xc3Transpose : real = DCM321_1_3 * DCM321_1_3 + DCM321_2_3
    * DCM321_2_3 + DCM321_3_3 * DCM321_3_3;

var c1xc2Transpose : real = DCM321_1_1 * DCM321_1_2 + DCM321_2_1
    * DCM321_2_2 + DCM321_3_1 * DCM321_3_2;
var c1xc3Transpose : real = DCM321_1_1 * DCM321_1_3 + DCM321_2_1
    * DCM321_2_3 + DCM321_3_1 * DCM321_3_3;
var c2xc3Transpose : real = DCM321_1_2 * DCM321_1_3 + DCM321_2_2
    * DCM321_2_3 + DCM321_3_2 * DCM321_3_3;


guarantee "EUL-006.1" S( (r1xr1Transpose = 1.0 and
   r2xr2Transpose = 1.0 and r3xr3Transpose = 1.0) and FTP , (
   r1xr1Transpose = 1.0 and r2xr2Transpose = 1.0 and
   r3xr3Transpose = 1.0));
```

```
guarantee "EUL-006.2" S( (r1xr2Transpose = 0.0 and
    r1xr3Transpose = 0.0 and r2xr3Transpose = 0.0) and FTP, (
    r1xr2Transpose = 0.0 and r1xr3Transpose = 0.0 and
    r2xr3Transpose = 0.0));
guarantee "EUL-006.3" S( (c1xc1Transpose = 1.0 and
    c2xc2Transpose = 1.0 and c3xc3Transpose = 1.0) and FTP, (
    c1xc1Transpose = 1.0 and c2xc2Transpose = 1.0 and
    c3xc3Transpose = 1.0));
guarantee "EUL-006.4" S( (c1xc2Transpose = 0.0 and
    c1xc3Transpose = 0.0 and c2xc3Transpose = 0.0) and FTP, (
    c1xc2Transpose = 0.0 and c1xc3Transpose = 0.0 and
    c2xc3Transpose = 0.0));
```

- **Verification result**: Invalid using Kind2 because of abstraction of cosine and sine functions.

[**EUL-007**]

- **Natural language**: The Rotation Matrix, DCM321, multiplied by the transpose of the Rotation Matrix, DCM321T shall be the Identity (3x3) Matrix.
- **FRET input**: EUL shall always satisfy $DCMxDCMTranspose\_1\_1 = 1.0$ & $DCMxDCMTranspose\_1\_2 = 0.0$ & $DCMxDCMTranspose\_1\_3 = 0.0$ & $DCMxDCMTranspose\_2$ 0.0 & $DCMxDCMTranspose\_2\_2 = 1.0$ & $DCMxDCMTranspose\_2\_3 = 0.0$ & $DCMxDCMTranspose\_3\_1 = 0.0$ & $DCMxDCMTranspose\_3\_2 = 0.0$ & $DCMxDCMTranspose\_3\_3 = 1.0$
- **Scope**: Global
- **CoCoSpec formalization**:

```
var r1xr1Transpose : real = DCM321_1_1 * DCM321_1_1 + DCM321_1_2
    * DCM321_1_2 + DCM321_1_3 * DCM321_1_3;
var r1xr2Transpose : real = DCM321_1_1 * DCM321_2_1 + DCM321_1_2
    * DCM321_2_2 + DCM321_1_3 * DCM321_2_3;
var r1xr3Transpose : real = DCM321_1_1 * DCM321_3_1 + DCM321_1_2
    * DCM321_3_2 + DCM321_1_3 * DCM321_3_3;

var r2xr1Transpose : real = DCM321_2_1 * DCM321_1_1 + DCM321_2_2
    * DCM321_1_2 + DCM321_2_3 * DCM321_1_3;
var r2xr2Transpose : real = DCM321_2_1 * DCM321_2_1 + DCM321_2_2
    * DCM321_2_2 + DCM321_2_3 * DCM321_2_3;
var r2xr3Transpose : real = DCM321_2_1 * DCM321_3_1 + DCM321_2_2
    * DCM321_3_2 + DCM321_2_3 * DCM321_3_3;

var r3xr1Transpose : real = DCM321_3_1 * DCM321_1_1 + DCM321_3_2
    * DCM321_1_2 + DCM321_3_3 * DCM321_1_3;
var r3xr2Transpose : real = DCM321_3_1 * DCM321_2_1 + DCM321_3_2
    * DCM321_2_2 + DCM321_3_3 * DCM321_2_3;
var r3xr3Transpose : real = DCM321_3_1 * DCM321_3_1 + DCM321_3_2
    * DCM321_3_2 + DCM321_3_3 * DCM321_3_3;
```

```
var DCMxDCMTranspose_1_1 : real = r1xr1Transpose;
var DCMxDCMTranspose_1_2 : real = r1xr2Transpose;
var DCMxDCMTranspose_1_3 : real = r1xr3Transpose;
var DCMxDCMTranspose_2_1 : real = r2xr1Transpose;
var DCMxDCMTranspose_2_2 : real = r2xr2Transpose;
var DCMxDCMTranspose_2_3 : real = r2xr3Transpose;
var DCMxDCMTranspose_3_1 : real = r3xr1Transpose;
var DCMxDCMTranspose_3_2 : real = r3xr2Transpose;
var DCMxDCMTranspose_3_3 : real = r3xr3Transpose;

guarantee "EUL-007"
S(
DCMxDCMTranspose_1_1 = 1.0 and
DCMxDCMTranspose_1_2 = 0.0 and
DCMxDCMTranspose_1_3 = 0.0 and
DCMxDCMTranspose_2_1 = 0.0 and
DCMxDCMTranspose_2_2 = 1.0 and
DCMxDCMTranspose_2_3 = 0.0 and
DCMxDCMTranspose_3_1 = 0.0 and
DCMxDCMTranspose_3_2 = 0.0 and
DCMxDCMTranspose_3_3 = 1.0 and FTP,
DCMxDCMTranspose_1_1 = 1.0 and
DCMxDCMTranspose_1_2 = 0.0 and
DCMxDCMTranspose_1_3 = 0.0 and
DCMxDCMTranspose_2_1 = 0.0 and
DCMxDCMTranspose_2_2 = 1.0 and
DCMxDCMTranspose_2_3 = 0.0 and
DCMxDCMTranspose_3_1 = 0.0 and
DCMxDCMTranspose_3_2 = 0.0 and
DCMxDCMTranspose_3_3 = 1.0 and);
```

- **Verification result**: Invalid using Kind2 because of abstraction of cosine and sine functions.

[EUL-008]

- **Natural language**: The determinant of the Rotation Matrix, $\|DCM321\|$, shall be equal to 1.0.
- **FRET input**: EUL shall always satisfy $DCM\_det = 1.0$
- **Scope**: Global
- **CoCoSpec formalization**:

```
--DCM_det is defined above in "EUL-004"
guarantee "EUL-008"  S(DCM_det = 1.0 and FTP, DCM_det = 1.0);
```

- **Verification result**: Invalid using Kind2 because of abstraction of cosine and sine functions.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD-MM-YYYY)*: 01-10-2019

2. REPORT TYPE: Technical Memorandum

3. DATES COVERED *(From - To)*:

4. TITLE AND SUBTITLE

Evaluation of the FRET and CoCoSim tools on the Ten Lockheed Martin Cyber-Physical Challenge Problems

5a. CONTRACT NUMBER: NNA14AA60C

5b. GRANT NUMBER:

5c. PROGRAM ELEMENT NUMBER:

6. AUTHOR(S)

Anastasia Mavridou, Hamza Bourbouh, Pierre-Loc Garoche, Mohammad Hejase

5d. PROJECT NUMBER:

5e. TASK NUMBER:

5f. WORK UNIT NUMBER:

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Ames Research Center, Moffett Field, CA 94035
Onera, The French Aerospace Lab, FR

8. PERFORMING ORGANIZATION REPORT NUMBER: L–

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S): NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S): NASA/TM–2019–220374

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-Unlimited
Subject Category 59
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES

An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT

We connect the Formal Requirements Elicitation Tool (FRET) with the CoCoSim verification tool to enable the automated analysis of hierarchical dataflow models against requirements written in a restricted English language. Our framework supports 1) automatic extraction of Simulink model information and association of high-level requirements with target model signals and components; 2) translation of temporal logic formulas into synchronous dataflow CoCoSpec specifications as well as Simulink monitors; and 3) interpretation of counterexamples produced by the analysis both at the requirement and model level. We report on the lessons learned from the application of our approach to the Lockheed Martin Cyber-Physical, aerospace-inspired challenge problems. For the analysis, we used the Kind2, Zustre, and Simulink Design Verifier (SLDV) tools.

15. SUBJECT TERMS

requirements engineering, verification, simulation, Simulink, Lustre

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Information Desk (email: help@sti.nasa.gov)
U	U	U	UU		**19b. TELEPHONE NUMBER** ***(Include area code)*** (757) 864-9658