The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained
Anastasia Mavridou, Hamza Bourbouh, Dimitra Giannakopoulou, Thomas Pressburger, Mohammad Hejase, Pierre-Loïc Garoche, Johann Schumann

To cite this version:

HAL Id: hal-02967437
https://enac.hal.science/hal-02967437
Submitted on 5 Jan 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Capturing and analyzing requirements of Cyber-Physical Systems (CPS) can be challenging, since CPS models typically involve time-varying and real-valued variables, physical system dynamics, or even adaptive behavior. MATLAB/Simulink is a development and simulation framework that is as early as possible, it is crucial to check that CPS models satisfy their requirements, which are usually expressed in natural language and are riddled with ambiguities.

In this paper, we evaluate the feasibility and benefits of applying automated tools to perform end-to-end analysis of CPS models. End-to-end means that we start with requirements elicitation, formalization, and analyzing requirements. Users write requirements in a restricted language and Logics. CPS requirements involve timing, so it is important to handle this aspect in requirements elicitation tools. The explanations produced by FRETIsh requirements captured our intended semantics. Even though FRETIsh is aimed at being intuitive, it was not always straightforward to turn natural language requirements into FRETIsh. However, most of the LMCPS requirements fall within a small number of patterns, an issue that we have also observed in other studies within our organization. Our logic was not able to capture some aspects of the system, in particular as related to delay blocks, which are heavily used in the models. We were able to shortcut this problem by exposing internal model variables at the requirements level, but a FRETIsh-level solution would be desirable.

For model analysis, we use COCOSIM [8, 9], an open source tool developed at NASA Ames, which analyzes Simulink models by connecting to formal tools such as the MathWorks Simulink Design Verifier (SLDV) [10] and Kind2 [11]. FRETI and COCOSIM are connected: COCOSIM exposes model details to FRETI to support the mapping between requirement- and model-variables by FRETI users; FRETI generates verification code that COCOSIM can process to analyze models against requirements. We study 1) the effectiveness of the connection between FRETI and COCOSIM, both in terms of producing verification code and in transferring verification results back at the requirements level, and 2) whether COCOSIM is able to successfully analyze the LMCPS requirements.

We were able to capture and analyze the majority of the LMCPS requirements with our framework. Our main findings can be summarized as follows:

Language and Logics. CPS requirements involve timing, so it is important to handle this aspect in requirements elicitation tools. The explanations produced by FRETI were instrumental in ensuring that the FRETIsh requirements captured our intended semantics. Even though FRETIsh is aimed at being intuitive, it was not always straightforward to turn natural language LMCPS requirements into FRETIsh. However, most of the LMCPS requirements fall within a small number of patterns, an issue that we have also observed in other studies within our organization. Our logic was not able to capture some aspects of the system, in particular as related to delay blocks, which are heavily used in the models. We were able to shortcut this problem by exposing internal model variables at the requirements level, but a FRETIsh-level solution would be desirable.

Formalization and Verification Code. FRETI formalizations are compact for most of the requirements of the LMCPS challenge; this is because they are optimized for many of the patterns that occur in the system. The automated production of verification code was a very smooth process. Automating the process of generating verification code from requirements has been extremely valuable since it reduces the sources of discrepancy and errors in the various artefacts.

Connecting Requirements to Models for Analysis. Requirements capture should not depend on the existence of a model. In fact, different members of our team worked on requirements capture and Simulink model analysis. We therefore found the capability of importing Simulink models in FRETI a great help during the step of connecting requirements with their targeted models. The most important feature of our integrated framework has been the capability to preserve the component structure of the LMCPS systems, and use it to perform analysis in a modular fashion. This has been instrumental in achieving scalability. Our analysis exposed issues including requirement ambiguities, undefined parts in the models, and small bugs in the checkers invoked by COCOSIM.

All details of our case study are available at [12], and our tools can be obtained, open source, at https://github.com/NASA-SW-YnV/. Our study can therefore be replicated by other researchers and practitioners.
2 Background

2.1 The FRETish language

A FRETish requirement contains up to six fields: scope, condition, component*, shall*, timing, and response*, where mandatory fields are indicated by an asterisk. ‘component’ specifies the component that the requirement refers to. ‘shall’ is used to express that the component’s behavior must conform to the requirement. ‘response’ is a Boolean condition that the component’s behavior must satisfy. ‘scope’ specifies intervals where the requirement is enforced. For instance, ‘scope’ can specify system behavior before a mode occurs, or after a mode ends, or when the system is in a mode. The optional ‘condition’ field is a Boolean expression that triggers the need for a ‘response’ within the scope. When triggered, the response must occur as specified by field timing, e.g., immediately, always, after/for/within N time units.

Each template is designated by a template key with values for fields [scope, condition, timing]. For example, [on, null, always] identifies requirements of the form In M mode, the software shall always satisfy R. Condition null (as opposed to regular), means that the response is triggered at the beginning of each scope interval. The most common key is [null, null, always], i.e., The software shall always satisfy R. Scope null indicates global scope, which means that the requirement is enforced on the entire execution interval. At the time of this case study, FRETish supported 8 values for field mode, 2 values for field condition, and 7 values for field timing, for a total of 8 × 2 × 7 = 112 semantic templates. More details on FRETish and its semantics are available in [7].

2.2 Past-Time Metric Linear Temporal Logic (pmLTL)

We briefly review the main pmLTL operators (Y, O, H, S, SI), which stand for Yesterday, Once, Historically, Since, and Since Inclusive, respectively. Y refers to the previous time step, i.e., at any non-initial time, Yφ is true iff φ holds at the previous time step. O refers to at least one past time step, i.e., Oφ is true iff φ is true at some past time step including the present time. Some of the templates are often defined as the composition of linear systems.

2.3 The Lustre language

Lustre [13] is a synchronous dataflow language. Lustre code consists of a set of nodes that transform infinite streams of input flows to streams of output flows, with possible local variables denoting internal flows. A symbolic “abstract” universal clock is used to model system progress. Two important Lustre operators in this context are the unary right-shift pre (for previous) and the binary initialization → (for followed-by) operator. Their semantics is as follows: at time t = 0, pre p is undefined, while for each time step t > 0 it returns the value of p at − 1. At time t = 0, p → q returns the value of p at t = 0, while for t > 0 it returns the value of q at t.

3 lockheed martin cyber physical systems (lmcps) challenge problems

The 10 Cyber-Physical V&V Challenges [5] were created by Lockheed Martin Aeronutronics to evaluate and improve the state-of-the-art in formal methods tools. Each challenge problem includes: 1) documentation that contains a high-level description and a set of requirements written in plain English; 2) a Simulink model; 3) a set of parameters (in .mat format) for simulating the model.

The challenges were first presented in the 2016 Safe and Secure Systems and Software Symposium (S5) [5]. They consist of a set of problems inspired by flight control and vehicle management systems, which are representative of flight-critical systems. They are publicly available and as such, they provide an excellent basis for discussion and comparison of approaches across the research community.

Although in most cases the specified requirements look relatively straightforward, a closer study revealed many questions regarding their precise meaning. Additionally, even though the Simulink models of the challenge were built with commonly used blocks, their analysis has proven to be challenging. Table 1 summarizes the ten LMCPS challenges. For each challenge it includes a brief description, the number of Simulink blocks in the models, the types of blocks that challenged our analysis, and the seven FRET format keys that we used to formalize the requirements of each challenge.

Vectors and Matrices. LMCPS challenges manipulate signals with multiple dimensions. The use of multi-dimensional signals and matrices is common in CPS Simulink models, since control systems are often defined as the composition of linear systems.

4 The fret-cocosim integrated framework

Figure 1 illustrates the flow of our framework. In the elicitation loop – Step 0 – the user writes and refines requirements in FRET based on the semantic explanations and simulation capabilities supported by FRET. Once the user is satisfied with the requirement semantics, the FRETish requirements are translated in Step 1 into pure Past-Time / Future-Time Metric LTL (pmLTL/fmLTL) formulas. In Step 2, data from the model under analysis is used to produce an architectural mapping between requirement propositions and Simulink signals. In Step 3, the pmLTL formulas and the architectural mapping are used to generate cocosim monitors and traceability data. In Step 4, cocosim [8] imports the generated cocosim monitors and traceability data, along with the Simulink model. cocosim then produces Simulink monitors, attaches them to the model, and produces Lustre code for the complete model (initial model plus attached monitors). cocosim can thus drive both Simulink-based (e.g., Simulink Design Verifier (SLDV)) and Lustre-based (e.g., Kind2 [11], Zustre) verification tools to analyze the target model in Step 5. Counterexamples produced by the analysis can be traced back to cocosim or FRET (Step 6).

We illustrate the entire process through the following requirement from the 6 Degree Of Freedom Dehavilland Beaver Autopilot (AP) LMCPS challenge.

[AP-003c] Natural Language: 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 mode engagement.

Step 0: Elicitation. Understanding the above natural language requirement and making it precise is not straightforward. We first identify the variables involved. By reading the first part of the requirement ‘The roll hold reference shall be set to 30 degrees in the same direction as the actual roll angle’ we identify two variables: roll_hold_reference and roll_angle, and express this part of the requirement as roll_hold_reference = 30 + sign(roll_angle), where function sign returns the sign, e.g., −1 or 1 of the roll_angle in order to determine its direction. For the second part of the requirement, i.e., if the actual roll angle is greater than 30 degrees at the time of roll hold mode engagement, we identify variables: roll_angle and
<table>
<thead>
<tr>
<th>Challenge</th>
<th>Description</th>
<th>NoB</th>
<th>Block Types</th>
<th>Template Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triplex Signal Monitor (TSM)</td>
<td>A redundancy management system that prevents errors from propagating past the input of an airborne application.</td>
<td>479</td>
<td>Non-linear (Switch), Vectors and Matrices</td>
<td>[null, null, always]</td>
</tr>
<tr>
<td>Finite State Machine (FSM)</td>
<td>An abstraction of an advanced autopilot system interacting with an independent sensor platform to ensure a safe automatic operation in the vicinity of hazardous obstacles.</td>
<td>279</td>
<td>Non-linear (Switch)</td>
<td>[null, null, always]</td>
</tr>
<tr>
<td>Tustin Integrator (TUI)</td>
<td>A flight software utility for computing the integration of a signal.</td>
<td>45</td>
<td>Non-linear (Switch, Saturation), Vectors and Matrices</td>
<td>[null, null, always]</td>
</tr>
<tr>
<td>Control Loop Regulators (REG)</td>
<td>A regulators inner loop architecture that is commonly used in many feedback control applications.</td>
<td>271</td>
<td>Non-linear (Switch, Saturation), Vectors and Matrices</td>
<td>[null, null, always]</td>
</tr>
<tr>
<td>Nonlinear Guidance Algorithm (NLG)</td>
<td>A nonlinear guidance algorithm that generates commands in order to guide an Unmanned Aerial Vehicle (UAV) to follow a moving target respecting a specific safety distance.</td>
<td>355</td>
<td>Non-linear (Sqrt, Switch), Vectors and Matrices</td>
<td>[null, null, always]</td>
</tr>
<tr>
<td>Feedforward Cascade Connectivity Neural Network (NN)</td>
<td>A two-input single-output predictor neural network with two hidden layers arranged in a feedforward architecture.</td>
<td>699</td>
<td>Non-linear (Saturation), Vectors and Matrices</td>
<td>[null, null, always]</td>
</tr>
<tr>
<td>Abstraction of a Control Allocator - Effector Blender (EB)</td>
<td>A control allocation method, which enables the calculation of the optimal effector (surface) configuration for a vehicle, given a control minimization effort problem.</td>
<td>75</td>
<td>Non-linear (Switch), Vectors and Matrices</td>
<td>[null, null, always]</td>
</tr>
<tr>
<td>6DoF with DeHavilland Beaver Autopilot (AP)</td>
<td>A full, realistic full six degree of freedom simulation of the DeHavilland Beaver airplane with autopilot.</td>
<td>1357</td>
<td>Non-linear (Switch, Sqrt, Abs, MinMax, Saturation), Non-algebraic (Trigonometric), Vectors and Matrices, Continuous-time</td>
<td>[null, null, always]</td>
</tr>
<tr>
<td>System Wide Integrity Monitor (SWIM)</td>
<td>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.</td>
<td>141</td>
<td>Non-linear (Switch, Sqrt), Vectors and Matrices</td>
<td>[null, null, always]</td>
</tr>
<tr>
<td>Euler Transformation (EUL)</td>
<td>A component that creates a Rotation Matrix describing a rotation about the z-axis, y-axis, and finally x-axis of an Inertial frame in Euclidean space.</td>
<td>97</td>
<td>Non-linear (Switch), Non-algebraic (Trigonometric), Vectors and Matrices</td>
<td>[null, null, always]</td>
</tr>
</tbody>
</table>
### Table 2: Semantic Template Formalizations. FTP (First Time Point) stands for \( \neg Y \) TRUE.

<table>
<thead>
<tr>
<th>Template Key</th>
<th>Past-time Temporal Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>[null, null, always]</td>
<td>( H(\phi \land ((Y \neg \phi) \lor FTP)) \Rightarrow \psi )</td>
</tr>
<tr>
<td>[null, regular, immediately]</td>
<td>( H(H(H(\neg \phi) \lor FTP)) \Rightarrow \psi )</td>
</tr>
<tr>
<td>[null, regular, always]</td>
<td>( H(H(H(\neg \phi) \lor FTP))) \Rightarrow H(H(H(\neg \phi) \lor FTP)) \Rightarrow \psi )</td>
</tr>
<tr>
<td>[null, null, for]</td>
<td>( H(\neg \psi) )</td>
</tr>
<tr>
<td>[in, null, immediately]</td>
<td>( H((\psi \land FTP) \lor (Y \neg \phi)) \Rightarrow \psi )</td>
</tr>
<tr>
<td>[in, regular, always]</td>
<td>( H((\neg \psi) \land FTP) \lor (Y \neg \phi)) \Rightarrow \psi )</td>
</tr>
</tbody>
</table>

### Figure 1: Requirement analysis framework

![Figure 1: Requirement analysis framework](image)

### Figure 2: fret semantics for requirement [AP-003c-v1]

**Roll hold node engagement.** Our first attempt at fretish for [AP-003c] was the following:

**[AP-003c-v1]:**
If \( \text{abs}(\text{roll\_angle}) > 30 \) & \( \text{roll\_hold\_mode\_engagement} \) then \( \text{autopilot shall immediately satisfy} \) \( \text{roll\_hold\_reference} = 30 \) * sign(\( \text{roll\_angle} \)), where function \( \text{abs} \) returns the absolute value of \( \text{roll\_angle} \).

**FRET displays requirement semantics in a variety of forms:** English descriptions, diagrammatic representations, logic formulas (metric temporal logics with pure future-time / pure past-time operators). Figure 2 shows the English and diagrammatic descriptions generated by FRET for [AP-003c-v1]. **TC** denotes a triggering condition.

Since scope is not specified, the requirement is 'enforced' in the interval defined by the entire execution, i.e., beginning of time to the end of the execution. Notice that the condition of the requirement, i.e., \( \text{if abs}(\text{roll\_angle}) > 30 \) & \( \text{roll\_hold\_mode\_engagement} \), becomes true (from false). **REQUIRES:** for every trigger, if trigger holds then \( \text{RES} \) also holds at the same time point.

We additionally use the FRET simulator, which allows users to interactively set values of requirement variables over a time interval and observe the consequences on the value of the requirement formulas. Let us abbreviate \( \text{abs}(\text{roll\_angle}) > 30 \) as \( 'p' \), \( \text{roll\_hold\_mode\_engagement} \) as \( 'q' \), and \( \text{roll\_hold\_reference} = 30 \) * sign(\( \text{roll\_angle} \)) as \( 'v' \). Figure 3 shows the FRET simulator. We can see that even when condition \( ('p' \) and \( 'q' \)) holds for two consecutive points, response is only required to hold at the first point (where condition becomes true), for the requirement (REQ) to hold. The green color of the last row (REQ) indicates that the requirement holds with the selected valuation of variables (it would be red otherwise).

Having thus used the aids that FRET provides for understanding fretish requirements, we realized that [AP-003c-v1] did not capture our intentions. Instead of a trigger we wanted the response to hold every time the condition is true and not only when it becomes true from false. Thus, we rewrote the requirement as follows:

**[AP-003c-v2]:** Autopilot shall always satisfy \( \text{abs}(\text{roll\_angle}) > 30 \) & \( \text{roll\_hold\_mode\_engagement} \) \( \Rightarrow \) \( \text{roll\_hold\_reference} = 30 \) * sign(\( \text{roll\_angle} \))
The response is now expressed as an implication. The requirement is no longer true in the scenario of Figure 3, as indicated by the red color of the last row (REQ) in Figure 4. Version 2 already seems closer to the intended semantics of the initial requirement, however, there is a temporal sub-property that we have not expressed explicitly yet — it is hidden in the roll_hold_mode_engagement variable. In order to elicit the full meaning of the requirement, it is ideal to express explicitly all sub-properties through the fretish grammar. The roll_hold_mode_engagement variable captures the fact that there is a mode of operation, i.e., the roll_hold mode, and the scope of our requirement only applies to the intervals in which roll_hold mode is true. The roll_hold_reference must be set to 30 degrees in the direction of the roll angle at the time of roll hold mode engagement, in other words, immediately upon entering the roll_hold mode. We unfold this subproperty within the fretish requirement as follows: [AP-003c-v3]: when in roll_hold mode Autopilot shall immediately satisfy (abs(roll_angle) > 30) ⇒ roll_hold_reference = 30 * sign(roll_angle).

The fret semantics for requirement [AP-003c-v3] is shown in Figure 5. The requirement is ‘enforced’ in every interval where roll_hold holds. The ‘trigger’ defines the first point in the mode interval: when roll_hold becomes true from false. Response is ‘required’ to hold at that same point.

Step 1: Formalization. The pmLTL formula that fret generates for [AP-003c-v3] is an instantiation of the [in, null, immediately] template key in Table 2:

$$\text{ENFORCED: in every interval where roll\_hold holds, TRIGGER: first point in the interval. REQUIRE: for every trigger, if trigger holds then RES also holds at the same time point.}$$

![Figure 5: fret semantics for requirement [AP-003c-v3]](image)

Table 3: FRET to Model Variables mapping for Autopilot (abbr. ap_12BAdapted/GlobalScope by global)

<table>
<thead>
<tr>
<th>FRET name</th>
<th>Model path</th>
</tr>
</thead>
<tbody>
<tr>
<td>roll_angle</td>
<td>global/Autopilot/Roll_Autopilot/Phi</td>
</tr>
<tr>
<td>roll_hold_reference</td>
<td>global/Autopilot/Roll_Autopilot/PlaRef_cmd</td>
</tr>
<tr>
<td>roll_hold</td>
<td>global/Autopilot/Roll_Autopilot/RollHold</td>
</tr>
</tbody>
</table>

The coconspec code then gets compiled into a Simulink monitor block, which is attached to the original model.

Steps 5 & 6: Analysis and counterexample generation. Requirement [AP-003c-v3] was shown to be invalid by the Kind2 model checker. Kind2 returned the counterexample shown in Table 4, which shows that at the time of roll hold mode engagement, (when T = 0.025, roll_hold becomes true and the absolute value of roll_angle is greater than 30), roll_hold_reference is not set to 30 degrees in the same direction as the roll_angle, i.e., roll_hold_reference is not equal to 30. Instead we noticed that at the time of roll hold engagement, roll_hold_reference is equal to 0.0, which is the value of roll_angle at the previous step. Based on this counterexample we modified the requirement as follows: [AP-003c-v4]: when in roll_hold mode Autopilot shall immediately satisfy (abs(roll_angle) > 30) ⇒ roll_hold_reference = previous(roll_angle), where previous is a function that returns the value of roll_angle at the previous time step. Requirement [AP-003c-v4] was proven valid.

In order to understand why the output is based on the previous value of roll_angle, we looked at the Simulink model. Figure 6 shows the Simulink model responsible for returning the roll_angle, where u is the roll angle and E is the condition ‘not engaged in roll 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’. Thus, we believe that [AP-003c-v3] is not satisfied due to an incomplete/erroneous model.

Note that we could not express the temporal subproperty previous.
Table 4: Counterexample of requirement [AP-003c-v3]

<table>
<thead>
<tr>
<th>Inputs</th>
<th>T = 0</th>
<th>T = 0.025</th>
</tr>
</thead>
<tbody>
<tr>
<td>roll_angle</td>
<td>0.0</td>
<td>-90.0</td>
</tr>
<tr>
<td>roll_hold</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>roll_hold_reference</td>
<td>3.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

roll_angle of [AP-003c-v4] directly through the fretish language and thus, we expressed it through an internal variable, which we defined directly in cocospec.

4.1 Reliable tool integration

A key feature in the integration of fret and cocosim is the capability to generate cocosim code from pmlTL logic formulas. Our translation keeps the structure of the original formulas, and is based on defining pmlTL operators in cocosim. This process is described in [15] for untimed operators. To support timed modifiers that constrain an operator’s scope to a specific interval [l, r], we have extended the process described in [15] and defined a library of timed operators in LMCPS. For instance for the timed version of 0, we added the following nodes to the library:

```
1 - Timed Case: general case
2 node DT(const L: int; const K: int; const I: bool;) returns (F: bool);
3 case 
4 var D: bool;
5 set
6 D = delay(l,K);
7 Y = DTlore(L,R,D);
8 return Y;
9
10 - Timed Case: less than or equal to K
11 node DTlore(const K: int; const I: bool;) returns (F: bool);
12 case
13 var C: int;
14 set
15 C = if X then 0 else (-1 -> pre C) (if pre C 0 then 0 else 1);)
16 set
17 Y = G <= C and C <= K;
18 return Y;
```

The delay function delays input X by R time units to define the right bound of the interval in which the valuation of X must be checked. Once the input X has been delayed by R time steps, we can treat the R bound as zero and use the DTlore (Once Timed less than or equal to no) node to check the valuation of X in the interval defined by the 0 (current) time step and the left bound L. DTlore is implemented using an integer counter C, which counts the number of time steps that occurred since the last occurrence of property X. If the event has never occurred, the counter keeps its initial value of -1. Other time-constrained operators are defined through DT using the usual temporal logic equivalences.

To provide assurance that the cocosim code generated is correct, we extend fret’s formula verification framework to also handle coco- spec code. The framework presented in [7] automatically generates test cases each consisting of an execution trace t, a template key k, and an expected truth value e, reflecting the semantics of k applied to t. Formulas corresponding to k are then evaluated on t using a model checker, to ensure that the result agrees with e. In our extension for cocosim, this step uses the Kind2 model checker. Our verification framework helped us detect and correct discrepancies between the cocospec generated formulas and the intended fret template key semantics.

5 Analysis - Selected use cases and requirements

Our case study encompasses the following tasks: 1) eliciting requirements in fretish; 2) making the mapping between fretish variables and model variables; 3) performing analysis; 4) interpreting counterexamples at requirements level; 5) interpreting requirements at model level. Three researchers were involved: 1) a control engineer, considered the domain expert; 2) a requirements expert; and 3) a verification expert. Tasks 1 and 2 were performed together by the requirements and domain experts. Tasks 3 and 5 were performed by the verification expert. Finally, Task 4 was performed by the requirements expert.

The verification results are summarized in Table 5. The analysis was carried out on a MacBook Pro with 3.1 GHz Intel Core i7 and 16 GB Memory, with a R2019b MATLAB/Simulink, and a v1.1.0 Kind2. Kind2 was configured to timeout after 2 hours. In this section, we highlight analysis results of a subset of the LMCPS challenges and discuss how we approached the challenging elements presented in Section 3.

Table 5: Counter example of requirement [FSM-003]

<table>
<thead>
<tr>
<th>Inputs</th>
<th>T = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>standby</td>
<td>true</td>
</tr>
<tr>
<td>supported</td>
<td>true</td>
</tr>
<tr>
<td>good</td>
<td>true</td>
</tr>
<tr>
<td>state</td>
<td>ap_transition_state</td>
</tr>
<tr>
<td>STATE</td>
<td>ap_standby_state</td>
</tr>
</tbody>
</table>

5.1 Requirements and Verification

Our presentation focuses on the following LMCPS components: FSM, TUI, NN, and AP. We include the FSM and TUI challenges because they exhibit cases of unrealizable requirements, as well as a requirement that we were not able to express directly in fretish. The NN challenge describes a machine learning model. Verification of models that are inferred by machine learning techniques is currently considered an open area for research. Finally, AP is the most complex of the LMCPS challenges in terms of number and types of blocks used, and its fretish requirements involve a variety of template keys.

FSM represents an abstraction of an advanced autopilot system interacting with an independent sensor platform 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.

All FSM requirement examples were written in fretish using the [null, null, always] semantic key pattern. Let us look into the following FSM requirements.

```
[FSM-002] Natural Language: The autopilot shall always satisfy (state = ap_transition_state) => \[ FSM-002 \]

[FSM-003] Natural Language: The autopilot shall change states from TRANSITION to NOMINAL when the system is supported and sensor data is good.
```

Note that non-mutually-exclusive requirements are not necessarily problematic, since requirements are often complementing each other to make up a system’s specification. In fact, we found several pairs of requirements that were not mutually exclusive in the LMCPS challenge.

TUI represents a flight software utility for computing the integration of a signal. The algorithm executed by the utility bounds the allowable integration range with a position limiter. The integrator is in normal mode of operation. If we form a weaker property, i.e., strengthen the precondition as follows (state = ap_transition_state & good & supported & Istandby) (notice the addition of Istandby), then [FSM-002] and [FSM-003] become mutually exclusive and requirement [FSM-003] is proven valid.

The valuations ap_transition_state, ap_standby_state, ap_nominal_state of the state and STATE variables represent the TRANSITION, STANDBY, and NOMINAL states of the autopilot. Requirement [FSM-002] was shown to be valid. However, when checking requirement [FSM-003], analysis returned the counterexample shown in Table 5. It is interesting to note that the valuation of the input variables of the counterexample satisfies the preconditions of both [FSM-002] and [FSM-003]. While these requirements are not mutually exclusive, their expected responses are conflicting, which makes them unrealizable [16]. If we form a weaker property, i.e., strengthen the precondition as follows (state = ap_transition_state & good & supported & Istandby) (notice the addition of Istandby), then [FSM-002] and [FSM-003] become mutually exclusive and requirement [FSM-003] is proven valid.

Note that non-mutually-exclusive requirements are not necessarily problematic, since requirements are often complementing each other to make up a system’s specification. In fact, we found several pairs of requirements that were not mutually exclusive in the LMCPS challenge.
a time duration (10 seconds). Such conditions are not yet supported in frestih.

NN is a two-input, single-output, two-hidden-layer feed-forward non-linear neural network. Neural networks of this form are common utilities in machine learning and simulation for capturing complex numerical dependencies. In this challenge, a single variable, $z$, is computed based on two independent parameters $x$ and $y$. This challenge comes with a truth table in the form of a Matlab matrix file with reference values $z_t$ and $z$. The NN specification file consisted of four requirements. We show below the frestest version of requirement [NN-004], for which we used the for 200 sec metric timing, which resulted in CoCoSpec code that uses the once timed (OT) metric RTL operator. [NN-004]: NN shall for 200 sec satisfy ($x = x_t \land y = y_t \implies \text{AbsoluteErrorZtMinusZ} \leq 0.01$).

Below is the generated CoCoSpec code for [NN-004]:

Kind2 and SLDV did not return an answer for any of the four NN requirements.

AP AP represents a complete system and, as shown in Table 1, it contains Simulink blocks that involve non-linearities, non-algebraic math, and manipulation of matrices. AP is a full six degree of freedom simulation of a single-engined high-wing propeller-driven airplane with autopilot. A six degree of freedom simulation enables movement and rotation in the three-dimensional space. The AP model and requirements also capture the plant mode of the airplane, i.e., the physical model, as well as environmental aspects such as wind that influence the motion of the airplane. AP requirements define the required behavior of the model in terms of changes in the three perpendicular position axes (forward/backward, left/right, up/down) combined with changes through rotation (yaw, pitch, and roll).

In the AP and FSM challenges, we performed model order reduction, since the specification generation mechanism of frest provides specifications at any desired level along with full traceability information regarding where the corresponding monitor should be deployed in the model. Additionally, the Cocosim compiler preserves the hierarchy of the model which allows performing analysis at different hierarchy levels.

The AP model 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 signals/variables (block outputs and inputs) in the same time-step. Lustre forbids such constructs; no circular dependencies are allowed. Strangely though, Kind2 was not able to detect the algebraic loop. We contacted a Kind2 developer and confirmed that there is a bug in the algebraic loop detection algorithm. Once the bug was fixed, top-level analysis was not possible and confirmed that there is a bug in the algebraic loop detection algorithm. We also used SLDV but the MathWorks license prevents publication of empirical results comparing with SLDV, so we omit the SLDV results from Table 6.

<table>
<thead>
<tr>
<th>Req</th>
<th>Scope</th>
<th>Kind2 Result</th>
<th>Kind2 Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP-000</td>
<td>Global</td>
<td>Unsupported</td>
<td></td>
</tr>
<tr>
<td>AP-001</td>
<td>Roll AP</td>
<td>Valid</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>AP-002</td>
<td>Roll AP</td>
<td>Valid</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>AP-003a</td>
<td>Roll AP</td>
<td>Invalid</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>AP-003b</td>
<td>Roll AP</td>
<td>Invalid</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>AP-003c</td>
<td>Roll AP</td>
<td>Invalid</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>AP-003d</td>
<td>Roll AP</td>
<td>Valid</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>AP-004</td>
<td>Global</td>
<td>Unsupported</td>
<td></td>
</tr>
<tr>
<td>AP-005</td>
<td>Global</td>
<td>Unsupported</td>
<td></td>
</tr>
<tr>
<td>AP-006</td>
<td>Global</td>
<td>Unsupported</td>
<td></td>
</tr>
<tr>
<td>AP-007</td>
<td>Roll AP</td>
<td>Valid</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>AP-008</td>
<td>Roll AP</td>
<td>Valid</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>AP-010</td>
<td>Global</td>
<td>Unsupported</td>
<td></td>
</tr>
</tbody>
</table>

The total running time in CoCosim was 40.589s.

Continuous time blocks Our analyses are based on the synchronous dataflow model and can only address discrete-time components. Thanks to the modular feature of our analyses, requirements associated to discrete-time components can be properly addressed. However, in the case of continuous-time components (defined using continuous blocks such as Integrators, Transfer functions, or State space blocks), we first replace them with their discrete counterparts using Simulink discretization functions.

Complex requirement formalizations As shown in Table 1, some templates like [null, regular, always] correspond to tally complex formulas. This template key was used for the specification of requirements [AP-004a] and [AP-010a] of the AP challenge. The scope of these requirements is the top level component of the model, and thus, they could not be analyzed by Kind2 (due to the algebraic loop) nor by SLDV, which returned undecided for both requirements. Note that however even simpler formalizations, such as the ones that correspond to the [null, null, always] key template, could not be analyzed globally. We also tried to verify specifications that correspond to [null, regular, always] at a local level and interestingly, we were able to analyze them, which shows that modular verification can be effective even for complex specifications.

6 Lessons Learned

The application of our framework to an externally-provided and challenging system has been very informative. We summarize our experiences and lessons learned below.

a) Can LMCPs requirements be captured in frest? We captured 69 out of 74 LMCPs requirements in frest. As mentioned, we were not able to formalize requirement [TUI-004] that contains a temporal condition. Additionally, several requirements refer to the previous value of a variable (e.g., see [AP-003c-v4]), defined in pmlTL with the Y operator, in Lustre with the pre operator, and in Simulink as a delay block. Currently, frestih cannot express ‘previous value of a variable’. To shortcut this limitation, we used internal/auxiliary variables which we defined at the Cocospec level, but a
Table 7: LMCPs verification results. $N_R$: #requirements, $N_F$: #formalized requirements, $N_A$: #requirements analyzed and Kind2. Analysis results categorized by Valid/INValid/UNDetermined. Timeout (TO) was set to 2 hours.

<table>
<thead>
<tr>
<th>Name</th>
<th>$N_R$</th>
<th>$N_F$</th>
<th>$N_A$</th>
<th>Kind2 V/IN/UN</th>
<th>Kind2 (t/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSM</td>
<td>6</td>
<td>6</td>
<td>6/10</td>
<td>37.7</td>
<td></td>
</tr>
<tr>
<td>FSM</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>7/6/0</td>
<td>141.1</td>
</tr>
<tr>
<td>TUI</td>
<td>4</td>
<td>3</td>
<td>2/10</td>
<td>19.2</td>
<td></td>
</tr>
<tr>
<td>REG</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>1/5/4</td>
<td>23/23/18</td>
</tr>
<tr>
<td>NLG</td>
<td>9</td>
<td>5</td>
<td>2/10</td>
<td>0/0/4</td>
<td>40.6</td>
</tr>
<tr>
<td>NN</td>
<td>4</td>
<td>4</td>
<td>4/0/1</td>
<td>TO</td>
<td></td>
</tr>
<tr>
<td>EB</td>
<td>5</td>
<td>3</td>
<td>0/0/3</td>
<td>TO</td>
<td></td>
</tr>
<tr>
<td>AP</td>
<td>14</td>
<td>13</td>
<td>8/5/0</td>
<td>40.6</td>
<td></td>
</tr>
<tr>
<td>SWIM</td>
<td>3</td>
<td>3</td>
<td>2/10</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>EUL</td>
<td>8</td>
<td>6</td>
<td>1/6/0</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>74</td>
<td>69</td>
<td>64/23/18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fretish solution** would be desirable. Additionally, we did not capture the following requirements: 1) [AP-009] since it was out of scope of the Simulink model; 2) [EB-003] since it is trivial; 3) [EUL-005] and [EB-005] were unclear: we were not able to interpret their meaning even with the help of the domain expert.

### b) Is fretish intuitive?

Fretish provides 112 semantic template keys to help translate template keys both at the model level and condition fields that were specified locally, but none of the properties that were specified globally. We have had a similar experience with a NASA mission that we collaborate with: their requirements fall into recurring patterns. We are currently extending Fret with the capability to define typical requirement patterns within a domain or project, and allowing users to import and use these patterns within the editor. This makes requirements capture a more natural and intuitive process.

### c) Are fret explanations useful?

We extensively relied on the semantic descriptions and diagrammatic representations, as well as the Fret simulation capabilities in order to understand the meaning of requirements throughout the LMCPs study. It helped us identify and understand several semantic nuances of the Fretish fields. The use of modes and condition fields as first-level constructs of the Fretish language was particularly useful. As shown in the elicitation phase of [AP-003c], unfolding the roll/hold/engagement subproperty through the Fretish mode field allowed us to elicit the full meaning of the requirement (see [AP-003c-v2] and [AP-003c-v3] requirements in Section 4).

### d) How effective is the fret-cocospec integration?

We were able to generate specifications and traceability data for all LMCPs challenges. These were used to automatically generate and attach monitors on the Simulink models (through COCOSPEC). We found the ability to interpret and trace counterexamples both at the model level and requirement levels particularly useful. In some cases, counterexamples uncovered conflicting requirements, and the model was not needed for understanding the problem (see Section 5). The Fret simulator was particularly useful in understanding these counterexamples. In other cases, counterexamples needed to be interpreted at the model level, since they exposed behaviors in the model that violated the requirements.

### e) How did we deal with model and specification complexity? To deal with complexity we performed modular analysis whenever possible, i.e., for non-system requirements (requirements that could be applied locally). Our architectural mapping approach allows us to deploy COCOSPEC specifications at different levels of the model behavior. This is especially important for complex models where verification does not scale for global scopes. We applied modular verification to 20 out of the 69 requirements. For instance, in the FSM challenge problem, we generated three different contracts that we deployed at three different hierarchical levels of the model. Similarly, in the AP challenge we generated two different contracts; one that we deployed at the top level component of the model and one that we deployed at a sub-component level. We were able to analyze all properties that were specified locally, but none of the properties that were specified globally.

### f) Which types of property reasoning/checking did we find helpful? Having a tight integration between requirement and verification activities allows us to use different approaches to interpret violated properties. In particular, we found useful the combination of reasoning at the level of requirements and counterexample simulation at the model level. When a property was shown to be invalid, we tried to understand the reason; i.e., is it because of a faulty requirement or a faulty model? Since in most cases, our formalized requirements were invariants of the form $A \Rightarrow B$, we used two approaches: 1) check a weaker property, e.g., by strengthening the preconditions, i.e., $A' \Rightarrow A$ and check whether the invariant $H (A' \Rightarrow B)$ is satisfied, and 2) check feasibility of $B$ with bounded model checking, i.e., $H \Rightarrow B$, in which case the model checker returns counterexamples that could help construct stronger preconditions for $B$ to be satisfied. Our case study showed that using these approaches was helpful for reasoning about violated properties. Furthermore, simulation of counterexamples was helpful for identifying weaker properties and producing meaningful reasoning scenarios.

Additionally, we used COCOSPEC modes to perform vacancy checking [17]. A COCOSPEC mode has preconditions that describe the activation of the mode (Requires) and actual conditions to be checked (Ensures) of the form $R (\Rightarrow E)$. Our case study showed that it is interesting to check whether the activation of a mode $R$ is reachable. So, in some cases, the property is infeasible if $R$ is reachable. Checking that $R$ is reachable allows us to have a better understanding of whether the property is meaningful for the current model. For instance, we discovered that in the AP challenge, one of the modes was never reachable.

### g) Were the abstraction techniques useful?

We used abstractions of non-linear functions, e.g., trigonometric functions and the sqrt function, to perform analysis with the Kind2 model checker. This proved to be helpful for three challenges: REG, SWIM, EUL. For instance, in the SWIM case study we were able to prove two more requirements by using a square root abstraction. In other cases, for instance in the EUL challenge, the abstractions would generate nonsensical counterexamples. For example, when we abstracted the cosine function with the interval $[-1,1]$, we got the following nonsensical counterexample: $\cos(0) = 0$.

### 7 Related Work

Our poster paper [15] describes the technical parts of the FRET-COCOSPEC integration: 1) the FRET interface through which the architectural mapping can be performed, 2) the library of (non-metric) pmlTL operators that we defined, 3) the generation of Simulink monitors through COCOSIM. The focus of this paper is very different; it describes in detail the LMCPs study and discusses challenges and lessons learned.

The components of the LMCPs challenge have also been analyzed in [3]. However, that work focused on comparing the efficacy of verification tools for model testing versus model checking (the latter using QVest from QRA Corp). In contrast, our work focuses more on the requirement formalization and model checking components. We also performed model checking using Kind2 and SLDV. There appears to be several differences in our formalization of requirements versus [3]. For example, in their companion material (reference 2 of [3]), they formalize $[AP-003c]$, which they call $R_{1.3}$, as the invariant $G[0,7](\Phi > 30) \Rightarrow \Phiref = 30$. We believe that this misrepresents the part of the natural-language requirement that mentions that the roll angle $\phi$ should be considered at the time of roll-hold engagement, as in our $[AP-003c-v3]$. The capability to explore the exact meaning of the requirements that are written in Fretish through provided explanations gives us confidence in our requirements capture. Moreover, our framework formalizes requirements automatically, sparing its users the error-prone effort of producing complex formulas for elaborate template keys.

Similar to FRET, the SpecAR [18], ASSERT™ [19], STIMULUS [20], RERD [21] and EARS-CTRL [22] tools provide natural-language like formal languages to express requirements and properties. The ARSENAL tool [23] attempts to formalize general natural language requirements, as opposed to FRET and the others mentioned where a constrained natural-language like formal language is used to express requirements. Except for STIMULUS, they do not appear to handle metric time, so would not be able to express some of the neural network properties. The goal of EARS-CTRL is to synthesize concrete controllers whereas formalizations in this paper are used with COCOSIM to verify the Simulink models against requirements. SpecAR and ASSERT can perform semantics checks on requirements, such as consistency and entailment, producing counter examples when such checks are violated. Checking for requirements realizability is in our plans. ASSERT generates test cases, and STIMULUS simulates sets of requirements. None of the tools automatically synthesizes monitors so that models can be model checked against requirements.
8 Conclusions
The LMCPS benchmark provides a valuable case study to evaluate requirements elicitation and analysis tools. We found that using an end-to-end automatic framework significantly simplifies requirements elicitation and model analysis. Requirements formalizations can easily become complex, and writing complex formulas by hand, or translating them in other logics can be hard and error-prone. Eliciting requirements with unambiguous and as-intended semantics is not an easy task. Explanations and interactive exploration of written requirements is a great tool for facilitating this task.

The ultimate purpose of formal requirements is to enable analysis. Requirements of CPSs can be complex to analyze, so it is important to provide modular analysis techniques to achieve scalability. Space projects at NASA Ames are currently starting to use our framework and we have already received valuable feedback. For example, desired customizations are customizable requirement patterns and the ability to express that a condition persists until some event. The advantage of a close collaboration with mission scientists during the development of requirements will allow us to further evaluate and improve the usability of our framework.

References