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

The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained

Anastasia Mavridou
KBR Inc., NASA Ames Research Center

Hamza Bourbouh
KBR Inc., NASA Ames Research Center

Dimitra Giannakopoulou
NASA Ames Research Center

Thomas Pressburger
NASA Ames Research Center

Mohammad Hejase
NASA Ames Research Center

Pierre-Loic Garoche
ENAC, University of Toulouse, France

Johann Schumann
KBR Inc., NASA Ames Research Center

January 5, 2021

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 widely used in industry to capture such systems. In this paper, we report on the application of NASA Ames tools to perform end-to-end analysis of the Ten Lockheed Martin Challenge Problems (LMCPS). LMCPS is a set of industrial Simulink model benchmarks and natural language requirements developed by domain experts. Our framework, which integrates the tools FRET and COCOSIM, is used to: 1) elicit, explain, and formalize the semantics of the given natural language requirements; 2) generate verification code and monitors that can be automatically attached to the Simulink models; 3) perform verification by using SMT-based model checkers. FRET and COCOSIM are open source, and can be used by other researchers and practitioners to replicate our case study. We provide a categorization of recurring patterns in the formalization of the requirements and discuss the strengths and weaknesses of our automated verification approach.

1 Introduction

Cyber-physical systems (CPS) integrate computation with physical processes. MATLAB®/Simulink® [1] is a widely-used framework in industry; in particular, more than 60% of engineers use Simulink for the development and simulation of CPS [2, 3]. To ensure that points of failure are identified 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 analysis, and proceed with model analysis against formalized requirements. Such analyses may result in updates of requirements and/or models.

End-to-end analysis of CPS models is challenging. Requirements are typically written in natural language, which is ambiguous and not easily amenable to formal analysis. Moreover, CPS models typically involve time-varying and real-valued variables, physical system dynamics, or even adaptive behavior. Formal languages supported by analysis tools may not be expressive enough to capture requirements for such systems. Finally, analysis tools may not be able to handle the complexity and scale of CPS models.

Our feasibility study targets the Ten Lockheed Martin Challenge Problems (LMCPS), a set of industrial Simulink model benchmarks and natural language functional requirements developed by domain experts [4, 5]. At the level of requirements, we use the Formal Requirements Elicitation Tool FRET [6, 7]. FRET is an open-source tool developed at NASA Ames for writing, understanding, formalizing, and analyzing requirements. Users write requirements in a restricted natural language, called FRETISH, with precise, unambiguous meaning. For a FRETISH requirement, FRET produces natural language and diagrammatic explanations of its exact meaning, and formalizes the requirement in logics. We investigate whether 1) the LMCPS requirements can be captured in FRET, 2) the process of capturing LMCPS requirements is intuitive to the user, and 3) the produced explanations provide useful feedback.

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]. FRET and COCOSIM are connected: COCOSIM exposes model details to FRET to support the mapping between requirement- and model- variables by FRET users; FRET generates verification code that COCOSIM can process to analyze models against requirements. We study 1) the effectiveness of the connection between FRET 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 FRET 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. FRET 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 FRET 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-VnV/`. 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, [*in*, *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 $8 \times 2 \times 7 = 112$ 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, `Y`ϕ is true iff ϕ holds at the previous time step. `O` refers to at least one past time step, i.e., `O`ϕ is true iff ϕ is true at some past time step including the present time. `H`ϕ is true iff ϕ is always true in the past. ϕ`S`ψ is true iff ψ holds somewhere at step t in the past and for all steps t' (such that $t' > t$) ϕ is true. Finally, ϕ `SI` $\psi \equiv \phi$ `S` $(\psi \,\&\, \phi)$. Timed modifiers constrain an operator's scope to specific intervals: $O_p\ [l, r]\ \phi$, where $O_p \in \{$`O, H, S, SI`$\}$ and $l, r \in \mathbb{N}^0$. For instance, `O` $[l, r]\ \phi$ is true at time t iff ϕ was true in *at least one* of the previous time steps t' such that $t - r \leq t' \leq t - l$. So `O`$[0, 3]$ restricts the scope of `O` to the interval including the step where the interval is interpreted and the previous 3 time steps.

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 `pre`vious) operator 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 $t - 1$. 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. COCOSPEC [14] is an extension of the synchronous dataflow language Lustre for the specification of assume-guarantee contracts.

3 lockheed martin cyber physical systems (lmcps) challenge problems

The 10 Cyber-Physical V&V Challenges [5] were created by Lockheed Martin Aeronautics to evaluate and improve the state-of-the-art in formal method toolsets. 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[1] and as such, they provide an excellent basis for discussion and comparison of approaches across the research community.

[1] `https://github.com/hbourbouh/lm\_challenges`

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 challenges 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 7 FRET template keys that we used to formalize the requirements of each challenge.

The LMCPS requirements and models were developed to represent challenges that are typical of CPS systems. The inputs and outputs of CPS systems are modeled through signals, which are functions over time. Most of the LMCPS models are highly numeric and often exhibit non-linear behavior. Next, we present elements of LMCPS that proved challenging for the analysis of the requirements. Section 5 discusses how we handled these challenges.

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.

Non Linear and Non Algebraic Blocks. Trigonometric functions, exponential functions, and the logarithm are typically not supported by SMT solvers. Two of the LMCPS challenges, i.e., AP and EUL, use the *Trigonometric Function* Simulink block to perform common trigonometric functions. The square root, i.e., *sqrt* Simulink block, used in the NLG, AP, and SWIM challenges is usually not well-handled by SMT solvers. Other non-linear Simulink blocks that are challenging for analysis are *Abs*, *MinMax*, *Switch*, and *Saturation*.

Continuous time blocks. Such blocks can be almost arbitrarily mixed with sampled blocks in Simulink. Thus another challenge comes from the fact that CPS models often contain mixes of continuous and discrete parts.

Complex requirement formalizations. As shown in Table 1, we used 7 distinct semantic template keys to express the LMCPS requirements. The formalization that corresponds to each template key is shown in Table 2. For certain template keys, e.g., [*in*, *regular*, *always*] the formalization is complex and potentially challenging for analysis tools.

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 FRETISH 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 COCOSPEC monitors and traceability data. In **Step 4**, COCOSIM [8] imports the generated COCOSPEC 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 1: Summary of LMCPS Challenges (NoB stands for Number of Blocks)

Challenge	**Description**	**NoB**	**Block Types**	**Template Keys**
Triplex Signal Monitor (TSM)	A redundancy management system that prevents errors from propagating past the input of an airborne application.	479	Non-linear (*Switch*), Vectors and Matrices	[*null*, *null*, *always*]
Finite State Machine (FSM)	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.	279	Non-linear (*Switch*)	[*null*, *null*, *always*] [*null*, *regular*, *immediately*]
Tustin Integrator (TUI)	A flight software utility for computing the integration of a signal.	45	Non-linear (*Switch*, *Saturation*), Vectors and Matrices	[*null*, *null*, *always*]
Control Loop Regulators (REG)	A regulators inner loop architecture that is commonly used in many feedback control applications.	271	Non-linear (*Switch*, *Saturation*), Vectors and Matrices, Continuous-time	[*null*, *null*, *always*]
Nonlinear Guidance Algorithm (NLG)	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.	355	Non-linear (*Sqrt*, *Switch*), Vectors and Matrices	[*null*, *null*, *always*] [*null*, *regular*, *always*]
Feedforward Cascade Connectivity Neural Network (NN)	A two-input single-output predictor neural network with two hidden layers arranged in a feedforward architecture.	699	Non-linear (*Saturation*), Vectors and Matrices	[*null*, *null*, *always*] [*null*, *regular*, *for*]
Abstraction of a Control Allocator - Effector Blender (EB)	A control allocation method, which enables the calculation of the optimal effector (surface) configuration for a vehicle, given a control minimization effort problem.	75	Non-linear (*Switch*), Vectors and Matrices	[*null*, *null*, *always*]
6DoF with DeHavilland Beaver Autopilot (AP)	A full, realistic full six degree of freedom simulation of the DeHavilland Beaver airplane with autopilot.	1357	Non-linear (*Switch*, *Sqrt*, *Abs*, *MinMax*, *Saturation*), Non-algebraic (*Trigonometric*), Vectors and Matrices, Continuous-time	[*null*, *null*, *always*] [*in*, *null*, *always*] [*in*, *null*, *immediately*] [*in*, *regular*, *always*] [*null*, *regular*, *immediately*]
System Wide Integrity Monitor (SWIM)	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.	141	Non-linear (*Switch*, *Sqrt*), Vectors and Matrices	[*null*, *null*, *always*]
Euler Transformation (EUL)	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.	97	Non-linear (*Switch*), Non-algebraic (*Trigonometric*), Vectors and Matrices	[*null*, *null*, *always*]

Table 2: Semantic Template Formalizations. FTP (First Time Point) stands for $\neg Y$ TRUE

Template Key	Past-time Temporal Logic
[*null*, *null*, *always*]	$H\psi$
[*null*, *regular*, *immediately*]	$H(\phi \wedge ((Y\neg\phi) \vee \text{FTP})) \Rightarrow \psi)$
[*null*, *regular*, *always*]	$H(H(\neg\phi) \vee (\psi S(\psi \wedge (\phi \wedge ((Y\neg\phi) \vee \text{FTP})))))$
[*null*, *null*, *for*]	$H((O[\leq duration]\text{FTP}) \Rightarrow \psi$
[*in*, *null*, *always*]	$H(\text{mode} \Rightarrow \psi)$
[*in*, *null*, *immediately*]	$H((\text{mode} \wedge (\text{FTP} \vee (Y(\neg\text{mode})))) \Rightarrow \psi)$
[*in*, *regular*, *always*]	$((H((((\neg mode) \wedge (Y mode)) \wedge (Y\text{TRUE})) \Rightarrow$ $(Y((((\neg\phi)S((\neg\phi) \wedge (mode \wedge (\text{FTP} \vee (Y(\neg mode))))))) \vee (\psi S$ $(\psi \wedge (\phi \wedge ((Y(\neg\phi)) \vee (mode \wedge (\text{FTP} \vee (Y(\neg mode)))))))))S$ $((((\neg\phi)S((\neg\phi) \wedge (mode \wedge (\text{FTP} \vee (Y(\neg mode))))))\vee$ $(\psi S(\psi \wedge (\phi \wedge ((Y(\neg\phi)) \vee (mode \wedge (\text{FTP} \vee (Y(\neg mode)))))))))\wedge$ $(mode \wedge (\text{FTP} \vee (Y(\neg mode)))))))))\wedge$ $(((\neg((\neg mode) \wedge (Y mode)))S$ $((\neg((\neg mode) \wedge (Y mode))) \wedge (mode \wedge (\text{FTP} \vee (Y(\neg mode))))))) \Rightarrow$ $((((\neg\phi)S((\neg\phi) \wedge (mode \wedge (\text{FTP} \vee (Y(\neg mode))))))\vee$ $(\psi S(\psi \wedge (\phi \wedge ((Y(\neg\phi)) \vee (mode \wedge (\text{FTP} \vee (Y(\neg mode)))))))))S$ $((((\neg\phi)S((\neg\phi) \wedge (mode \wedge (\text{FTP} \vee (Y(\neg mode))))))\vee$ $(\psi S(\psi \wedge (\phi \wedge ((Y(\neg\phi)) \vee (mode \wedge (\text{FTP} \vee (Y(\neg mode)))))))))\wedge$ $(mode \wedge (\text{FTP} \vee (Y(\neg mode)))))))))$

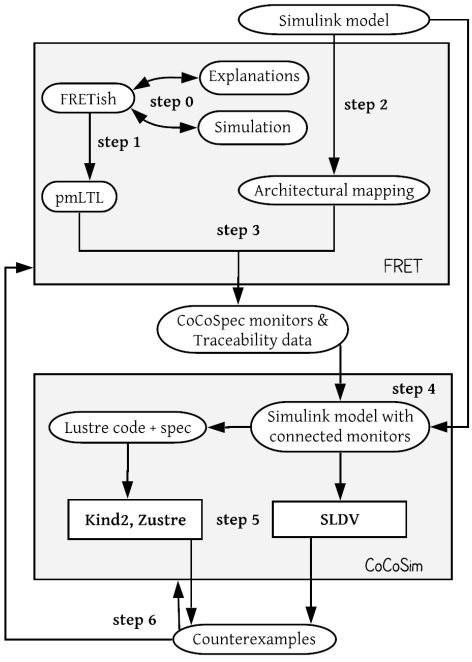

Figure 1: Requirement analysis framework

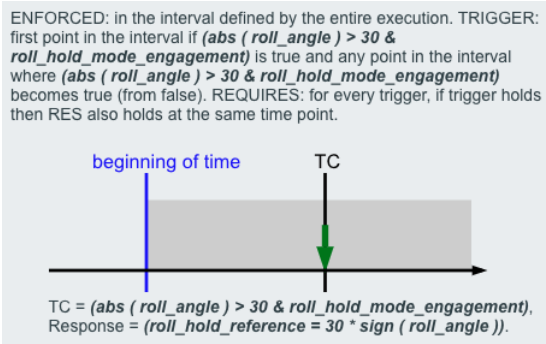

Figure 2: FRET semantics for requirement [**AP-003c-v1**]

`roll_hold_mode_engagement`. Our first attempt at FRETISH for [**AP-003c**] was the following:
[**AP-003c-v1**]:
If abs(roll_angle) > 30 & roll_hold_mode_engagement Autopilot shall immediately satisfy roll_hold_reference = 30 * sign(roll_angle), where function `abs` returns the absolute value of `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., If abs(roll_angle) > 30 & roll_hold_mode_engagement, is a 'trigger': the requirement is only enforced at time points where the condition becomes true from false, or at the first point of the interval if the condition holds there. Timing "immediately" states that the response should hold simultaneously with each trigger 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 abs(roll_angle) > 30 as 'p', roll_hold_mode_engagement as 'q', and roll_hold_reference = 30 * sign(roll_angle) as 'r'. 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 (abs(roll_angle) > 30 & roll_hold_mode_engagement) ⇒ roll_hold_reference = 30 *

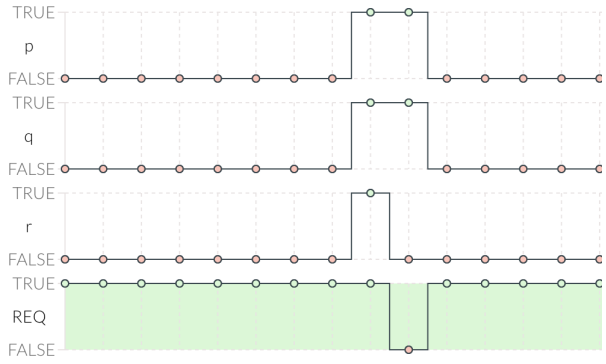

Figure 3: Simulating requirement [**AP-003c-v1**]

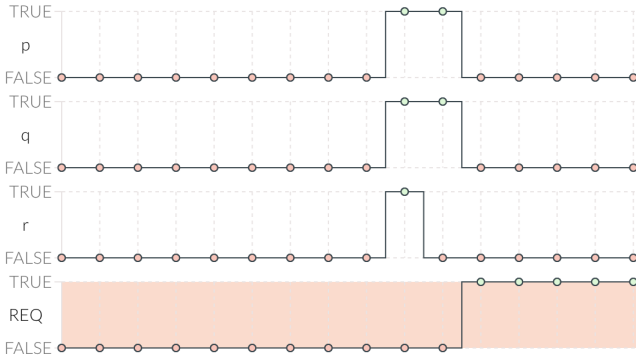

Figure 4: Simulating requirement [**AP-003c-v2**]

sign(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 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:

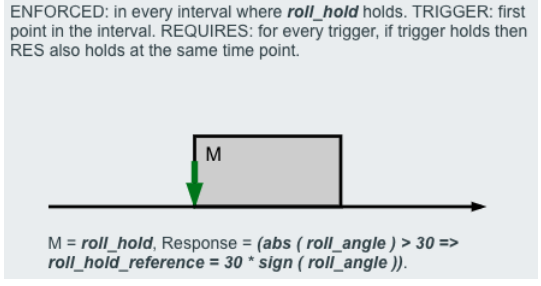

Figure 5: FRET semantics for requirement [**AP-003c-v3**]

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

FRET name	**Model path**
roll_angle	global/Autopilot/Roll_Autopilot/Phi
roll_hold_reference	global/Autopilot/Roll_Autopilot/PhiRef_cmd
roll_hold	global/Autopilot/Roll_Autopilot/RollHold

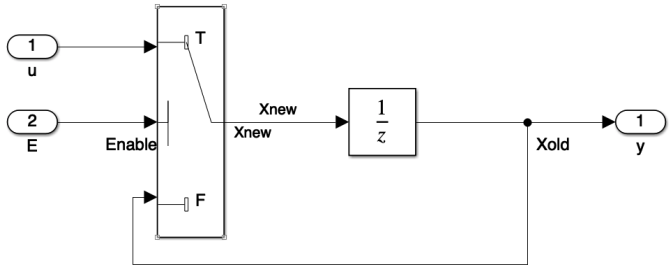

Figure 6: Simulink model for requirement [**AP-003c-v4**]

$(H(roll\_hold\ \&\ (!FTP \mid (Y(!\ roll\_hold))) \Rightarrow (abs(roll\_angle) > 30 \Rightarrow roll\_hold\_reference = 30 * sign(roll\_angle))))$,
where FTP is a predicate that holds at the First Time Point of an execution (equivalent to $\neg\ Y$ TRUE).

Step 2: Architectural mapping. To generate monitors and automatically attach them at the right hierarchical level of the model, we need architectural data from the model. For instance, for [**AP-003c-v3**], we need the path, in the model hierarchy, of the `Autopilot` component mentioned in FRETISH. Additionally, we need information about the signals of the component, e.g., name, type (e.g., `input, output`), datatype (e.g., `boolean, double, bus`) that correspond to the variables mentioned in [**AP-003c-v3**]. Our framework provides a mechanism to automatically extract the required data from a Simulink model. The mapping of the variables used in [**AP-003c-v3**] is shown in Table 3.

Steps 3 & 4: Generation of analysis code and Simulink monitors. To translate [**AP-003c-v3**] into Lustre code, the pmLTL formula generated by FRET gets translated into the following COCOSPEC code, used by COCOSIM for analysis:

```
-- AP-003c-v3 requirement in CoCoSpec
guarantee H((roll_hold and (FTP or (pre (not roll_hold))))
            => abs(roll_angle) > 30 =>
            roll_hold_reference = 30 * sign(roll_angle))
```

The COCOSPEC 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]**

Inputs	**T = 0**	**T = 0.025**
roll_angle	0.0	-90.0
roll_hold	false	true
Outputs		
roll_hold_reference	3.0	0.0

`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 COCOSPEC code from pmLTL logic formulas. Our translation keeps the structure of the original formulas, and is based on defining pmLTL operators in COCOSPEC. 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 created a library of timed operators in Lustre. For instance for the timed version of `O`, we added the following nodes to the library:

```
--Timed Once: general case
node OT(const L: int; const R: int; X: bool;)
     returns (Y: bool);
   var D:bool;
let
   D = delay(X,R);
   Y = OTlore(L-R,D);
tel

--Timed Once: less than or equal to N
node OTlore(const N: int; X: bool; ) returns (Y: bool);
    var C:int;
let
    C = if X then 0
        else (-1 -> pre C + (if pre C <0 then 0 else 1));

    Y = 0 <= C and C <= N;
tel
```

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 `OTlore` (Once Timed less than or equal to) node to check the valuation of `X` in the interval defined by the 0 (current) time step and the left bound `L-R`. `OTlore` 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 `OT` using the usual temporal logic equivalences.

To provide assurance that the COCOSPEC code generated is correct, we extend FRET's formula verification framework to also handle COCOSPEC 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 COCOSPEC, 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 7. 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]

Inputs	**T = 0**
standby	true
supported	true
good	true
state	ap_transition_state
Outputs	
STATE	ap_standby_state

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 change states from TRANSITION to STANDBY when the pilot is in control (standby).*
[FSM-002] fretish: FSM shall always satisfy (standby & state = ap_transition_state) => STATE = ap_standby_state.

[FSM-003] Natural Language: *The autopilot shall change states from TRANSITION to NOMINAL when the system is supported and sensor data is good.*
[FSM-003] fretish: FSM shall always satisfy (state = ap_transition_state & good & supported) => STATE = ap_nominal_state.

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 & !standby) (notice the addition of !standby), 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.

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 operation when it is not in reset mode, and the output is within the specified limits.

[TUI-004] Natural Language: *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.*

This requirement could not be expressed directly in FRETISH. The "*After 10 seconds of computation at an execution frequency of 10 Hz*" part of the requirement constitutes a condition that must persist for

a time duration (10 seconds). Such conditions are not yet supported in FRETISH.

NN is a two-input, single-output, two-hidden-layer feed-forward nonlinear neural network. Neural networks of this form are common utilities in modeling 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 xt, yt and zt. The NN specification file consisted of four requirements. We show below the FRETISH 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 LTL operator.

[NN-004]: NN shall for 200 sec satisfy (x = xt & y = yt => AbsoluteErrorZtMinusZ $\leq$ 0.01).

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

```
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 );
```

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 modular analysis, since the specification generation mechanism of FRET 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 with Kind2. Simulink, on the other hand, treats algebraic loops as algebraic constraints which it solves numerically by using the ODE (Ordinary Differential Equation) solver. However, this is not considered a good programming practice since the behavior is defined by the simulator engine.

As shown in Table 6[2], we were able to get results only for the requirements that were analyzed against a sub component (local scope, e.g., Roll AP). Requirements that were analyzed against the top AP component (global scope), e.g., **[AP-000]** were either unsupported or undecided. In particular, analysis with Kind2 was not supported due to the algebraic-loop, while analysis with SLDV was undecided.

5.2 Challenges

Vectors and Matrices SMT solvers do not typically support multi-dimensional signals as native objects. Signals and matrices need to be split into individual scalar variables making analysis harder depending on the operations that need to be performed. For instance, the EB and AP challenges use blocks that compute the inverse of matrices, while AP also manipulates quaternions with some advanced quaternion operations (e.g. *Quaternion Modulus*, *Quaternion Norm* and *Quaternion Normalize*).

We experienced the same problem at the level of requirements. In order to generate specifications that can be used for analysis by SMT solvers, we had to encode and expand matrix operations as non-matrix formulas. For instance, requirement **[EUL-001]** describes the computation of the DCM321 matrix as the product of the 3x3 Euler_Roll_Rotation and the 3x3 Euler_Pitch_Rotation matrices. In order to perform analysis on this requirement, we specified it in FRETISH by first decomposing it into nine sub-requirements, i.e., one for each element of the DCM321 matrix. Such decomposition naturally results in a considerably larger specification, as compared to the original natural language requirement.

[2]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 6: AP Analysis Results with Kind2

Reqs	Scope	Kind2 Result	Kind2 Time
[AP-000]	Global	Unsupported	
[AP-001]	Roll AP	Valid	$<$ 1 sec
[AP-002]	Roll AP	Valid	$<$ 1 sec
[AP-003a]	Roll AP	Invalid	$<$ 1 sec
[AP-003b]	Roll AP	Invalid	$<$ 1 sec
[AP-003c]	Roll AP	Invalid	$<$ 1 sec
[AP-003d]	Roll AP	Valid	$<$ 1 sec
[AP-004]	Global	Unsupported	
[AP-005]	Global	Unsupported	
[AP-006]	Global	Unsupported	
[AP-007]	Roll AP	Valid	$<$ 1 sec
[AP-008]	Roll AP	Valid	$<$ 1 sec
[AP-010]	Global	Unsupported	
Total running time		CoCoSim: 40.589s	

Non Linear and Non Algebraic Blocks Trigonometric functions, exponential functions, and the logarithm are typically not supported by SMT solvers. To be able to perform meaningful analysis on LMCPS models that contain trigonometric and square root blocks, we abstracted their behavior by providing a surrogate version, which is a sound abstraction. For instance, instead of block *sqrt*, which defines the signal $x = sqrt(y)$, we encoded properties $x * x = y \wedge x \geq 0$. Similarly for trigonometric functions, we provided bounds for the values depending on the input range.

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 template keys like [*in*, *regular*, *always*] correspond to really 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 [*in*, *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 experience and lessons learned below.

a) Can LMCPS requirements be captured in fret? We captured 69 out of 74 LMCPS requirements in FRET. 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, FRETISH 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 by Kind2. Analysis results categorized by **V**alid/**IN**valid/**UN**decided. Timeout (TO) was set to 2 hours

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

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? FRET provides 112 semantic template keys out of which we used only 7. Among these template keys, the [*null*, *null*, *always*] key was used the most (75% of the formalizations). 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 customize 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-cocosim 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 COCOSIM). We found the ability to interpret and trace counterexamples both at the model 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 `H` $(A \Rightarrow B)$, we used two approaches: 1) check a weaker property, e.g., by strengthening the preconditions, i.e., $A' \subset A$ and check whether the invariant `H` $(A' \Rightarrow B)$ is satisfied, and 2) check feasibility of B with bounded model checking, i.e., `H` $(\neg B)$, 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 vacuity 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 `H` $(\texttt{R} \Rightarrow \texttt{E})$. Our case study showed that it is interesting to check whether the activation of a mode `R` is reachable. If not, the property is trivially true. So, in terms of analysis, showing 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-COCOSIM 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 QVtest from QRA Corp). In contrast, our work focuses more on support for requirements elicitation and formalization, though 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 R1.3, as the invariant $G\{0,T\}(\text{Phi} > 30 \Rightarrow \text{PhiRef} = 30)$. 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 SpeAR [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 controllers whereas formalizations in this paper are used with COCOSIM to verify the Simulink models against requirements. SpeAR 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 synthesize 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 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

[1] "Simulation and model-based design," 2020. [Online]. Available: https://www.mathworks.com/products/simulink.html

[2] X. Zheng, C. Julien, M. Kim, and S. Khurshid, "Perceptions on the state of the art in verification and validation in cyber-physical systems," *IEEE Systems Journal*, vol. 11, no. 4, pp. 2614–2627, 2015.

[3] S. Nejati, K. Gaaloul, C. Menghi, L. C. Briand, S. Foster, and D. Wolfe, "Evaluating model testing and model checking for finding requirements violations in Simulink models," in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19*. New York, NY, USA: Association for Computing Machinery, 2019, p. 1015–1025.

[4] C. Elliott, "On example models and challenges ahead for the evaluation of complex cyber-physical systems with state of the art formal methods V&V, Lockheed Martin Skunk Works," in *Safe & Secure Systems and Software Symposium (S5)*, A. F. R. Laboratory, Ed., 2015.

[5] ——, "An example set of cyber-physical V&V challenges for S5, Lockheed Martin Skunk Works," in *Safe & Secure Systems and Software Symposium (S5)*, A. F. R. Laboratory, Ed., 2016.

[6] D. Giannakopoulou, A. Mavridou, T. Pressburger, J. Rhein, J. Schumann, and N. Shi, "Formal requirements elicitation with FRET," in *26th Intl Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020, Tool)*, 2020.

[7] D. Giannakopoulou, T. Pressburger, A. Mavridou, and J. Schumann, "Generation of formal requirements from structured natural language," in *26th Intl Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020)*, 2020.

[8] CoCo-team, "CoCoSim – automated analysis framework for Simulink." https://github.com/NASA-SW-VnV/CoCoSim.

[9] H. Bourbouh, P.-L. Garoche, T. Loquen, É. Noulard, and C. Pagetti, "CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models," in *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, 2020.

[10] G. Hamon, B. Dutertre, L. Erkok, J. Matthews, D. Sheridan, D. Cok, J. Rushby, P. Bokor, S. Shukla, A. Pataricza *et al.*, "Simulink design verifier-applying automated formal methods to simulink and stateflow," in *Third Workshop on Automated Formal Methods*, 2008.

[11] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, "The Kind 2 model checker," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, 2016, pp. 510–517.

[12] A. Mavridou, H. Bourbouh, P.-L. Garoche, and M. Hejase, "Evaluation of the FRET and CoCoSim tools on the ten Lockheed Martin cyber-physical challenge problems," NASA, Tech. Rep., oct 2019, 84 pages. [Online]. Available: https://drive.google.com/open?id=1DhEk-A0PgDBU9nU1Jhlb-UFYC8Af71W2

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[14] A. Champion, A. Gurfinkel, T. Kahsai, and C. Tinelli, "Cocospec: A mode-aware contract language for reactive systems," in *International Conference on Software Engineering and Formal Methods*. Springer, 2016, pp. 347–366.

[15] A. Mavridou, H. Bourbouh, P.-L. Garoche, D. Giannakopoulou, T. Pressburger, and J. Schumann, "Bridging the gap between requirements and Simulink model analysis," in *26th Intl Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020, Poster)*, 2020.

[16] A. Gacek, A. Katis, M. W. Whalen, J. Backes, and D. Cofer, "Towards realizability checking of contracts using theories," in *NASA Formal Methods Symposium*. Springer, 2015, pp. 173–187.

[17] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 2, pp. 224–233, 2003.

[18] A. W. Fifarek, L. G. Wagner, J. A. Hoffman, B. D. Rodes, M. A. Aiello, and J. A. Davis, "SpeAR v2.0: Formalized past LTL specification and analysis of requirements," in *NASA Formal Methods Symposium*, 2017, pp. 420–426.

[19] A. Crapo, A. Moitra, C. McMillan, and D. Russell, "Requirements capture and analysis in ASSERT^{TM}," in *2017 IEEE 25th International Requirements Engineering Conference (RE)*, Sep. 2017, pp. 283–291.

[20] B. Jeannet and F. Gaucher, "Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, Jan. 2016.

[21] E. Stachtiari, A. Mavridou, P. Katsaros, S. Bliudze, and J. Sifakis, "Early validation of system requirements and design through correctness-by-construction," *Journal of Systems and Software*, vol. 145, pp. 52 – 78, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412121830150X

[22] L. Lúcio, S. Rahman, S. bin Abid, and A. Mavin, "EARS-CTRL: generating controllers for dummies," in *Proceedings of MODELS 2017 Satellite Event*, ser. CEUR Workshop Proceedings, vol. 2019. CEUR-WS.org, 2017, pp. 566–570.

[23] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner, "ARSENAL: automatic requirements specification extraction from natural language," in *NASA Formal Methods Symposium*, 2016, pp. 41–46.