Authoring, Analyzing, and Monitoring Requirements for a Lift-Plus-Cruise Aircraft

Tom Pressburger[1], Andreas Katis[2], Aaron Dutle[3], and Anastasia Mavridou[2]

[1] NASA Ames Research Center, Moffett Field, CA, USA
[2] KBR, NASA Ames Research Center, Moffett Field, CA, USA
[3] NASA Langley Research Center, Hampton, VA, USA

Abstract. [Context & Motivation] Requirements specification and analysis is widely applied to ensure the correctness of industrial systems in safety critical domains. Requirements are often initially written in natural language, which is highly ambiguous, and as a second step transformed into a language with rigorous semantics for formal analysis. **[Question/problem]** In this paper, we report on our experience in requirements creation and analysis, as well as run-time monitor generation using the Formal Requirement Elicitation Tool (FRET), on an industrial case study for a Lift-Plus-Cruise concept aircraft. **[Principal ideas/results]** We study the creation of requirements directly in the structured language of FRET without a prior definition of the same requirements in natural language. We focus on requirements describing state machines and discuss the challenges that we faced, in terms of creating requirements and generating monitors. We demonstrate how realizability, i.e., checking whether a requirements specification can be implemented, is crucial for understanding temporal interdependencies among requirements. **[Contribution]** Our study is the first complete attempt at using FRET to create industrial, realizable requirements and generate run-time monitors. Insight from lessons learned was materialized into new features in the FRET and JKind analysis frameworks.

1 Introduction

The process of writing requirements for safety critical systems can be an arduous task, as engineers need to avoid ambiguous semantics and ensure that the resulting specification excludes unsafe system behavior. Formal specification can help engineers overcome both obstacles, as requirements are translated into unambiguous constructs using mathematical logic. Still, writing requirements using a formal language is not straightforward, especially when the author lacks a solid background in logical concepts. Furthermore, the analysis of such requirements can often leave engineers in a state of confusion. as they struggle with the interpretation of both positive and negative results.

The Formal Requirements Elicitation Tool (FRET) [9] is an active, open-source research project [1] developed at NASA Ames, providing a highly accessible requirements engineering and analysis framework. It is designed so that

engineers with varying levels of experience in formal methods can express requirements using structured natural language, observe their behavior through interactive simulation, and analyze their correctness with respect to their realizability; i.e., answer whether a system implementation exists that is guaranteed to conform to the given specification no matter the inputs received from its environment. The requirements can then be used to generate run-time monitors, which are programs that detect the violation of a particular requirement during the execution of the system. A tool chain beginning with FRET allows for partially automatic generation of such monitors.

In this paper, we briefly present FRET (§ 2) and report our experience using it to author and analyze requirements (§ 4) for an industrial case study on a Lift-Plus-Cruise concept aircraft (§ 3) as well as generate run-time monitors (§ 5). There is a focus on formulating state machines using FRET. We showcase challenges that we encountered, corresponding to common problems in requirements engineering, from expressing said requirements, to checking their realizability and actually interpreting the analysis results. We furthermore discuss how we were able to address issues, not only through the process of refining the specification, but also in terms of improving existing features of FRET to improve explainability of analysis results. This top-to-bottom study provided us with valuable insights, which we describe through lessons learned (§ 6).

Related Work: Previous work explored using FRET for industrial-level case studies, in cases where natural requirements specification already existed [20,6]. In contrast, in this paper we create requirements from informal diagrams and incorporate realizability checking as part of the workflow. Similar studies have been conducted in the past for other requirements specification tools. Previous works in the RAT [24] SPECTRA [19] and LTSA [17] tools has showed how requirements expressed in Linear Temporal Logic could be evolved, guided by the results of consistency and realizability analysis in a Boolean setting. The EARS-CTRL [18] tool provides a natural language, and its analysis for synthesizing controllers is also in a Boolean setting. In comparison, requirements written in FRET's language can deal with linear arithmetic expressions over unbounded integer and real numbers. A study analyzing control software in the AGREE framework identified errors in specification using realizability checking [3]. Notably, the checking algorithm used is known to be unsound w.r.t. unrealizable results [8], whereas FRET employs sound procedures [12,16]. The CLEAR [4] tool also uses a constrained natural language to formalize requirements. It can check completeness and consistency, but not realizability.

2 Background

FRET provides a collection of features for the creation, management and analysis of requirements. We next present the features that were used in this paper, namely the FRETish language, the realizability checking component and finally the requirements export functionality to achieve synthesis of run-time monitors.

Requirement specification and formalization. Users write requirements in FRETish, i.e., a restricted natural language with standard mathematical expressions [9]. A FRETish requirement is described using up to six sequential fields (the * symbol designates mandatory fields): 1) `scope` specifies the time intervals where the requirement is enforced, 2) `condition` is a Boolean expression that triggers the `response` to occur at the time the expression's value becomes true from false, or is true at the beginning of the scope interval, 3) `component*` is the system component that the requirement is levied upon, 4) `shall*` is used to express that the component's behavior must conform to the requirement, 5) `timing` specifies when the response shall happen, subject to the constraints defined in `scope` and `condition` and 6) `response*` is the Boolean expression that the component's behavior must satisfy.

FRETish provides 8 scopes: *global*, *in*, *before*, *after*, *notin*, *only in*, *only before*, and *only after*. The scope *global* means *always*; the others are with respect to when the system is in a mode or satisfies a Boolean expression. The optional condition field is introduced by any of the words *upon*, *when*, or *if*, which are synonymous. FRETish provides 10 timings: *immediately* (meaning: at the same time point), *at the next timepoint*, *always*, *eventually*, *never*, *for* N time steps, *within* N time steps, *after* N time steps, *until* *bool_expr*, and *before* *bool_expr*. When the scope is omitted it is taken as *global*; when the condition is omitted, it is taken as `true`; when the timing is omitted, it is taken as *eventually*.

The Boolean expressions use the standard logical symbols, as well as standard arithmetic symbols and relations. FRETish also supports several predefined predicate symbols: *preBool(init,x)* (resp., *preReal(init,x)*) denotes, at the first time point, the value of the Boolean (resp., real) expression *init*, and subsequently the value of the Boolean (resp., real) expression x at the previous time point; *absReal(x)* denotes the absolute value of the real-valued expression x; and *FTP* is true at the first time point in the execution, and is false otherwise.

FRETish requirements are based on rigorous semantics and thus, have a precise, unambiguous meaning. Once the requirements are specified, FRET produces formalizations in several logics. In this study, we make use of past-time metric linear temporal logic (pmLTL).

To capture commonly occurring requirement patterns, FRET provides a *template* facility. This allows the user to construct FRETish requirements by instantiating placeholders in a template.

Realizability checking. Informally, a specification is realizable if we can implement a system, such that it always conforms to the given requirements, while considering inputs from an uncontrollable environment (sensors, user input, etc.). A proof of realizability not only establishes the truth that a system can be implemented for the given requirements, but also the fact that, given proper care in the system implementation, the requirements themselves are free of conflicts that would translate into unsafe behavior.

The analysis portal in FRET provides means to examine specifications in terms of their realizability, as well as generate artifacts that help engineers further understand the analysis results. The following features are available [13]:

— *Compositional analysis:* As a preprocessing step, FRET decomposes the specification, if possible, into a set of connected components, based on the outputs exercised in each requirement. This decomposition is sound w.r.t. realizability, allowing the independent analysis of each one of the computed components [21].
— *A portfolio of engines and algorithms that support infinite theories.* FRET uses both the KIND 2 [5,16] and JKIND [7,12] model checkers for realizability analysis. Both engines are SMT-based, supporting unbounded theories of integer and real arithmetic, while also providing means to compute counterexamples from unrealizable specifications, in the form of deadlocking execution traces.
— *Diagnosis of unrealizable specifications.* FRET employs diagnostic algorithms to provide further feedback in unrealizable requirements. This is achieved through the computation and simulation of minimal sets of unrealizable requirements (known as minimal conflicts) [14,15]. Counterexamples that demonstrate unrealizability can be graphically displayed in the interactive simulator in FRET.

Exporting requirements for monitor synthesis. Having created a set of realizable requirements, we can now generate runtime monitors. To this end, FRET generates and exports a specification that can be digested by the OGMA tool [23] for the generation of COPILOT monitors [22]. This specification contains formalized requirements and information about the variable types referenced in the requirements. OGMA then produces an input specification for COPILOT, and finally COPILOT generates C code suitable for use in hard real-time systems, running without dynamic memory allocation in predictable space and time. The C code accepts inputs to be monitored and invokes user-provided handlers when the requirements are violated. The creation and integration of these monitors is intended to be as seamless as possible; the properties to be monitored are written in FRETISH, and little to no code is required to be written by hand.

3 The Lift Plus Cruise Case Study

Because of its ability to be used at many stages of the development lifecycle, and the familiarity of the researchers with the tool, FRET was chosen as a main component in a NASA project studying safety assurance for a novel Lift-Plus-Cruise (LPC) electric Vertical Takeoff and Landing (eVTOL) aircraft. There are several different concepts for VToL aircraft being investigated by the aviation community, including NASA [26]. One such design has a number of lifting rotors attached to the wings and a forward pushing propeller on the rear of the aircraft (Figure 1). NASA is developing models of the flight characteristics of this LPC concept, as well as simulation capabilities, and control schemes [11]. The project investigated aspects of safety assurance

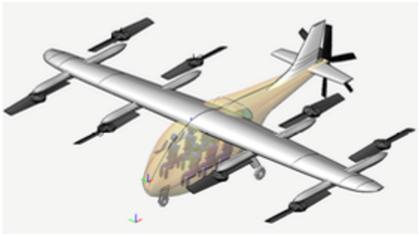

Fig. 1. The LPC vehicle.

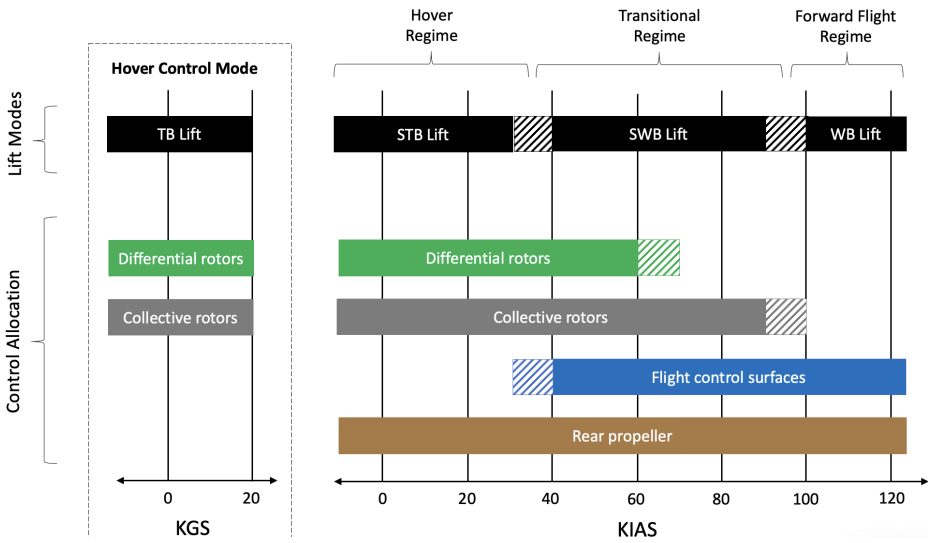

Fig. 2. Control Allocation Schedule. KGS/KIAS: ground/indicated air speed (knots).

of the aircraft including hazard analysis, requirements capture, formal modeling, and runtime monitoring. FRET was used to capture requirements for the vehicle, and the collection of requirements served as a model of how the vehicle was expected to behave. Some of these requirements were then used to generate runtime monitors for use in the simulation environment.

Due to the design of the aircraft, several distinct control regimes may apply at different phases of flight. For example, during takeoff and landing, the aircraft motion is controlled by the lifting rotors only, and the flight surfaces (wings, ailerons, etc) have no effect (thrust-borne mode, TB). On the other hand, during the higher speeds of the en-route phase, the wings provide lift, the rear propeller provides thrust, and the lifting rotors are inactive (wing-borne mode, WB). *Collective* control means that all of the rotors are commanded to increase or decrease torque, leading to more or less "heave" (vertical climb). *Differential* control means that the rotors are commanded to have differing amount of torques, enabling control of pitch, yaw and roll.

Figure 2 shows the ranges of air/ground speeds for the control regimes. The hashed areas indicate regions of hysteresis; i.e., control lag. For example, if the vehicle is slowing down from the wing-borne mode (WB), the transition to semi-wing-borne (SWB) kicks in at an indicated airspeed of 90 knots (kias $<=$ 90.0), whereas if the vehicle is speeding up from a SWB mode, the transition to WB mode occurs at kias $>$ 100.0 knots; similarly for the transitions between semi-thrust-borne (STB) mode and SWB mode. The vehicle remains in the thrust-borne mode (TB) as long as kgs $<=$ 20.0 knots and Hover Control (HC) mode is selected.

The main research questions that we aim to answer through this work are: 1. *Can we take informal descriptions of how the vehicle is supposed to operate and behave, and (through FRET) turn this into a formal description/model that can be analyzed?* and 2. *Can we use this formal model to easily create monitors?*

4 Writing Requirements for LPC

The work presented in this paper was the result of multiple iterations between requirements formalization and their respective analysis in terms of realizability. Requirements development was done iteratively, over a period of eight months part-time, with the requirements researchers meeting with the aircraft controls researchers regularly to refine both the requirements and controls. The requirements development revealed some ways that FRET could be enhanced to better capture the types of requirements needed, and to analyze them, so FRET additional feature development occurred concurrently. While the current work is a research project, the overall concept of formally capturing and analyzing requirements for a developer to test against, and using these requirements as runtime monitors, is envisioned as a method to help assure safety of future aircraft.

4.1 Initial Formalization

To validate the control scheme concept, and facilitate use in further development and refinement of control software, we undertook the formal modeling in FRET of the control allocation of the LPC concept during the landing transition phase. This phase transitions from fully wing-borne flight to fully thrust-borne, with intermediate phases semi-wing-borne and semi-thrust-borne.

Our task is to develop realizable requirements for the control schedule (Figure 2). The complete set of requirements is in the technical report [25]. The variables used in these requirements, as well as their types are shown in Table 1. For the purposes of realizability checking and monitor generation, we need to declare each variable as either an *input* or *output*. An output is a variable that the system controls. An input is a monitored variable, one whose value is set by the environment that the system has no control over.

Table 1. LPC Variables.

cr	boolean	output
dr	boolean	output
fcs	boolean	output
HC	boolean	output
rearprop	boolean	output
kgs	double	output
kias	double	output
wind_speed	double	input
lift_mode	integer	output
TB	integer	constant 0
STB	integer	constant 1
SWB	integer	constant 2
WB	integer	constant 3

We start with a requirement that the vehicle be in one of the lift modes at each time point. Note that integer constants in Table 1 are used to simulate a lift-mode enumerated type.

[LIFT_MODE]: The vehicle `shall` always satisfy lift_mode = TB | lift_mode = STB | lift_mode = SWB | lift_mode = WB

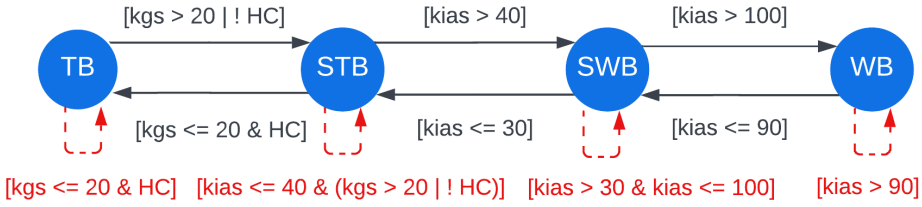

Fig. 3. Lift Mode State Machine derived from Fig. 2. The acronyms are: HC = hover control, B = borne, T = thrust (rotors), W = wing, S = semi-, kgs = ground speed (knots), kias = indicated air speed (knots).

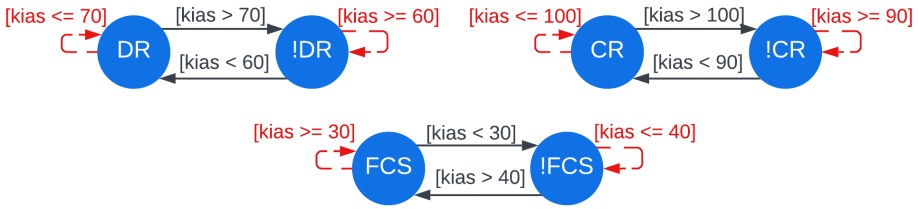

Fig. 4. Differential Rotors (DR), Collective Rotors (CR), and Flight Control Surface (FCS) state machines derived from Fig. 2.

We also require that the rear propeller be always used, except in HC mode: **[REARPROP]:** The vehicle `shall` always satisfy rearprop xor HC

To specify the control schedule requirements, we chose the clear and succinct way that state machines provide, and expressed those state machines in FRETish. Initially, we transformed what is shown in Figure 2 into state machines. E.g., for the required behavior of the lift modes, we created Figure 3: the four states correspond to the lift modes and the black, solid-line guarded transitions define when a mode change may happen. For instance, when in STB mode and the ground speed is less than or equal to 20 knots (kgs <= 20), the pilot, or an automated control system, can switch to HC mode, allowing the aircraft to enter the TB lift mode. Similarly, the control allocation state machines are represented in Figure 4. The guards on the transitions refer to the conditions on the indicated airspeed in knots (kias) and ground speed (kgs). Initially, we designed Figures 3 and 4 without the red, dashed-line loop transitions.

To capture *transition* requirements, we created the following FRET template to express transitions from state s_0 to a state s_1 under condition p:

> Upon state = s_0 & p the vehicle `shall` at the next timepoint satisfy state = s_1

E.g., the transition originating from state WB to state SWB in Figure 3 can be written as follows: **[WB_TO_SWB]:** Upon lift_mode = WB & kias <= 90.0 the vehicle `shall` at the next timepoint satisfy lift_mode = SWB

4.2 Refinement using Realizability Checking

Using the realizability analyzer over this initial set of requirements, led us to discover that we also need a *stay* requirement that says the state remains s_0 if none of the exit transition conditions from s_0 hold. Otherwise, the required behavior is under-specified, and hence anything could happen after a transition to a particular state when no transition condition applies. In particular, realizability analysis, as shown in Table 2, reported a realizable trace where the aircraft state transitions from wing-borne mode directly to thrust-borne mode without visiting intermediate modes. The stay requirements are necessary for specification completeness: the behavior under all conditions must be specified, so the disjunction of the guards of the transitions from a state needs to be a valid formula [10]. In the past, FRET had a template for writing state-machine transition requirements, which originated from a set of given natural-language requirements [20] that were neither realizable nor complete. We improved on this template, by having a simplified transition requirement template that uses a single state variable, as well as adding a template for stay requirements. The new templates allow for complete specifications of state machine requirements.

One could express the stay requirement in FRETish as: When state $= s_0 \,\&\, P$, the vehicle `shall` at the next timepoint satisfy state $= s_0$, where $P = !p_1 \,\&\, \ldots \,\&\, !p_n$ is the conjunction of negated guards that belong to outgoing transitions of s_0. However, this would only constrain the value of state when the condition transitioned from false to true, not whenever the condition held. Instead, the stay requirement can be expressed with the following FRET template (see [25]):

> Vehicle `shall` always satisfy if preBool(false, *state* $= s_0 \,\&\, P$) then state $= s_0$

Currently, this is formalizable by FRET only in pmLTL. This was adequate for this case study, since both realizability analysis and monitor generation rely on the past-time formalization. If, in a different situation, a future-time formula is needed, it can be expressed in FRETish without preBool as Upon state $= s_0$ & P the vehicle `shall` until state $= s_0$ & $!P$ satisfy state $= s_0$. This says that the system, upon entering state s_0 when no transition condition applies, will remain in state s_0 until *and including* the time point where a transition condition holds. The two formulations were shown, using the NuSMV model-checker, to be equivalent. Although equivalent logically, realizability analysis using the second formulation was 15 to 100 times slower; we are investigating the cause. We show below stay transition requirements from the wing-borne mode (Figure 3) and flight control surfaces (Figure 4). Other stay and transition requirements were written in a similar manner. These requirements correspond to the dashed-line loop transitions (Figures 3 and 4).

[WB_STAY_ON_pre]: The vehicle `shall` always satisfy
if preBool(false, lift_mode = WB & kias > 90.0) then lift_mode = WB

[WB_STAY_ON_until]: Upon lift_mode = WB & kias > 90.0 the vehicle `shall` until lift_mode = WB & kias <= 90.0 satisfy lift_mode = WB

Table 2. Example trace from incomplete specification.

Variable \ Step	0	1	2	3	4	5	6
HC	false	false	false	false	false	false	false
kgs	120	120.25	110.25	111.5	103.5	100	90.25
kias	120	120.25	110.25	111.5	103.5	100	90.25
lift_mode	WB	TB	STB	SWB	WB	SWB	SWB

Table 3. Example trace from the final specification.

Variable \ Step	0	1	2	3	4	5	6	7	8	9	10	11
HC	false	false	false	false	false	false	false	false	false	false	true	true
kgs	120	110	100	90	80	70	60	50	40	30	20	20
kias	120	110	100	90	80	70	60	50	40	30	20	20
lift_mode	WB	WB	WB	WB	SWB	SWB	SWB	SWB	SWB	SWB	STB	TB

[FCS_STAY_OFF]: The vehicle `shall` always satisfy if preBool(false, !fcs & kias <= 40.0) then !fcs

[FCS_TURN_ON]: Upon !fcs & kias > 40.0 the vehicle `shall` at the next time-point satisfy fcs

So far, we have specified the required behavior for transitioning from wing-borne lift mode to thrust-borne mode. Still, we are not done: we need to specify initial conditions, as well as a time target before which the transition should complete. We try the scenario where the initial airspeed is 120 knots, the initial lift mode is wing-borne, the ground speed always equals the airspeed, and the airspeed changes by no more than 10 knots in consecutive time points:

[INIT_KIAS]: The vehicle `shall` immediately satisfy kias = 120.0

[INIT_LIFT_MODE]:
The vehicle `shall` immediately satisfy lift_mode = WB <=> kias >= 90.0

[KIAS_KGS]: The vehicle `shall` always satisfy kias = kgs

[KIAS_DERIVATIVE]: The vehicle `shall` always satisfy
FTP | absReal(preReal(0.0, kias) − kias) <= 10.0

All that is now left is to define a possible goal about *lift_mode*:

[REACH_HOVER]: The vehicle `shall` within 10 ticks satisfy lift_mode = TB

We now claim that we have a complete formalization. Is it realizable, though? Careful inspection should result in a "no" answer, as 10 ticks is not enough time to complete the transition. The realizability analysis supported this claim: the requirements are unrealizable for 10 ticks and realizable for 11 ticks. Table 3 shows a positive trace from the latter result, where the system is able to complete the transition from wing-borne to thrust-borne in a proper manner, exercising the intended intermediate mode transitions.

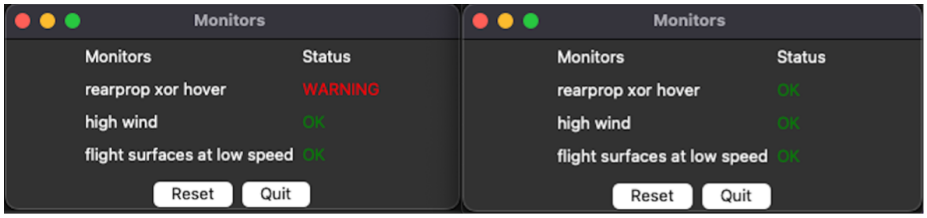

Fig. 5. Runtime monitor displays: monitor violation (left), no violation (right).

4.3 Reasoning about the System's Environment

Notably, the requirements presented thus far do not constrain the system's input. We experimented with additional requirements involving wind, changing **[KIAS_KGS]** to specify that kgs is the sum of kias and the wind_speed input variable (hence uncontrollable). Furthermore, we added the following assumption on the environment: **[WIND_SPEED_assumption]:** The vehicle `shall` always satisfy absReal(wind_speed) <= 30.0

We expected that this assignment for kgs would prohibit entering TB mode because the wind would prevent kgs <= 20. However, in about a minute, realizability checking said that it was realizable. Examination of a positive trace revealed that this was due to kias becoming negative; i.e., the vehicle flying backwards. The diagram we were initially given (Fig 2) is misleading: the vehicle can only maneuver backwards slowly, while in TB mode, to make small corrections while landing. When a requirement was included that said kias >= 0, the requirements were shown to be unrealizable, even when increasing the time limit in **[REACH_HOVER]** to 16 ticks. Strengthening the assumption to |wind_speed| <= 20 fixed the issue, as the requirements were shown to be realizable within 13 ticks, which makes sense as kias needs to be reduced to zero for kgs to be <= 20, for any valid wind speed.

5 Generation of Run-Time Monitors

We integrated the C code generated by Copilot into the FlightDeckZ Vehicle Simulation Environment [2], monitoring three requirements described earlier: **[REARPROP]**, **[FCS_STAY_OFF]**, and **[WIND_SPEED_assumption]**. FlightDeckZ is a system that incorporates physics models of the LPC concept with flight controllers, and a visualization system, to allow for fairly realistic flight simulation of the LPC vehicle model with experimental controllers.

The first two monitors express requirements that we expect from the control system of the LPC model, while the last monitor expresses an environmental property that may be of interest to a pilot during an actual flight (as most eVToL systems are not designed to take off in high winds). This difference here is intentional. The first two monitors are likely more useful to a system developer, and so can likely be removed from use once a stable and trustworthy control

system is in place. The last monitor is something that may be integrated into a system display on a real aircraft. The status of the monitors is displayed to the users with a simple window frame, with descriptions of the monitors and their current status displayed side-by-side (Figure 5).

6 Lessons Learned

We list below lessons and FRET needs and improvements resulting from the experience of using FRET in this case study.

Expressiveness of FRETish: In this effort, requirements were written directly in FRETish based on informal diagrams describing desired behavior, rather than being translated from an initial natural language description. Thus, we were interested in understanding whether FRETish provides adequate expressiveness and clarity and whether we are able to capture requirements that observe complex interaction behavior for generating meaningful runtime monitors.

We were able to express in FRETish all the requirements of the control allocation schedule. Writing these requirements directly in FRETish made them more detailed while avoiding ambiguities; a lot of attention was given to understanding their semantics and how small changes in their syntax affect it.

We also found limitations: FRET lacks an enumerated type facility; a workaround with integer constants was used instead for the lift modes in Table 1. Also, a condition that enforces the response whenever the condition is true, not just triggering the response upon the condition becoming true from false, would have been useful, as discussed in § 4.2.

Usefulness of tool assistance in writing requirements: Crucial to the requirement formulation process were the interactive simulator of FRET and the realizability checking mechanism that guided the discussion to corner cases, important sanity checks, and complete requirement sets (see § 4.2).

Formulating correctly the FRETish for state transitions involved some subtlety, but once the FRET templates were devised, they were used to specify 26 out of the 53 LPC requirements. Since state machines are frequently used in requirements development, we expect that the FRET templates could be useful to others who wish to formulate complete and realizable state machine requirements. On the other hand, instead of formulating such requirements in FRETish, the ability to express requirements in a state-machine notation directly could be a useful addition to FRET.

Usability of feedback from realizability analysis in the form of positive and negative traces, and minimal conflicts: In several cases (e.g., the cases mentioned in § 4.3 that revealed negative air speed, and in § 4.2 the need for "stay" requirements for completeness), we needed evidence to understand why a specification was realizable. This motivated a new feature in JKind and the FRET analysis portal that computes and displays a satisfying, i.e., positive, trace showing how

the requirements are realizable. We achieved this by using the proof produced by realizability checking. More specifically, when a specification is proved realizable by the underlying tools, a symbolic fixpoint of "good system states" is computed. We reuse this fixpoint to compute and present to the users valid system execution traces of bounded length, that can be seen as indicative runs of a system that is, by definition, guaranteed to always comply with the specification. Examples of such generated positive traces were shown in Tables 2 and 3. Furthermore, we enabled the use of the simulator to interact with the requirements in context, starting from the satisfying trace (see [25]).

Unrealizable results also contributed to the refinement of the specification. E.g., note how we allow the vehicle to control the HC variable (i.e., declared as an output). The fact that it needed to be an output was pointed out by the realizability analysis: the specification was unrealizable when the variable was originally declared as an input, because the environment could decide to never switch to hover control mode. When a set of requirements is unrealizable, it is left up to the FRET user to puzzle out from a negative trace and experimentation why the requirements don't allow a positive trace. In particular, minimal conflicts can be subsets of requirements that discard necessary requirements, for example, **LIFT_MODE**. Further research is needed in the area of providing helpful counterexamples. We sometimes found it sufficient to find the cause just by examining which requirements were in the conflict set.

Dealing with requirement versions: During realizability analysis, we refined our requirements multiple times. Thus, we ended up with several different versions of the same requirements that can be used within different subsets of requirements. This motivated a new feature in the FRET analysis portal that allows the user to easily select which requirement versions should be included in each realizability check (see [25]). In certain cases, we ended up with logically equivalent requirement versions. We thus think that there should be a capability of the FRET interface to test the equivalence of requirements, without the user needing to escape to other tools.

Easy monitoring: FRETish allows for the easy specification of many complex and time-based interactions inside a system. For example, in the LPC model, if one of the lifting rotors fails, the mirrored rotor on the other wing should be turned off, so that a thrust imbalance does not occur. In FRETish, one could easily specify a property that says "Upon rotor_1_fail, the vehicle shall within 5 seconds satisfy rotor_4_power_off". Such a monitor could then be automatically generated, and requirements violations could be detected without post-simulation analysis, or even without the need for manual writing of code that collects and assesses the state of the system over periods of time.

Monitor semantics mismatch: We discovered an issue with FRET-generated Copilot monitors during the integration and testing process. Due to the fact that the requirements are turned into pmLTL, the interpretation of each requirement is the statement "always in the past, requirement x holds." What this

means is that at each time step, the monitor is determining if there has ever been a violation. Hence even if the system returns to a state that is determined to be safe, the monitor is still considered violated. For example, if the wind ever goes above 30 knots, then even after the wind calms, the statement "The wind shall always be below 30 knots" is false, so the monitor stays on. Currently, a workaround "reset" button restarts the monitors, effectively erasing all past history, mitigating the issue.

7 Conclusion

This experience report paper showed how certain aspects of a concept Lift-Plus-Cruise aircraft were captured in requirements written in FRETish and how realizability analysis was crucial for guiding the evolution of the requirements. The main requirements engineering challenges that we encountered stemmed from the iterative process of refining requirements with respect to realizability. These challenges were not apparent until after the step of analysis was reached.

To answer our main research questions: we were successful in turning informal descriptions into an analyzable formal model through FRET and subsequently using this formal model to easily create monitors. To this end, the FRET model did fulfill its purpose. Additionally, experience with this case study led us to improve FRET as well as to point to future work such as adding to the expressiveness of FRETish and revisiting the semantics of run-time monitoring.

Acknowledgements: We acknowledge Michael Feary, John Kanishige, and Kimberlee Shish who explained the vehicle used in this study and provided Figure 2, and Dimitra Giannakopoulou who did an early requirements development. Thanks also to the anonymous reviewers who provided detailed improvement suggestions. This work was supported by the Advanced Air Mobility and System Wide Safety projects in the NASA Aeronautics Mission Directorate's Airspace Operations and Safety Program. Andreas Katis and Anastasia Mavridou were supported by contract NASA 80ARC020D0010.

References

1. FRET, `https://github.com/NASA-SW-VnV/fret.git`
2. Archdeacon, J., Iwai, N., Feary, M.: Aerospace cognitive engineering laboratory (ACELAB) simulator for electric vertical takeoff and landing (eVTOL) research and development. In: AIAA Aviation Forum (2020)
3. Backes, J., Cofer, D., Miller, S., Whalen, M.W.: Requirements analysis of a quad-redundant flight control system. In: NFM 2015
4. Bhatt, D., Ren, H., Murugesan, A., Biatek, J., Varadarajan, S., Shankar, N.: Requirements-driven model checking and test generation for comprehensive verification. In: NFM 2022
5. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: CAV 2016
6. Farrell, M., Luckcuck, M., Sheridan, O., Monahan, R.: Fretting about requirements: Formalised requirements for an aircraft engine controller. In: REFSQ 2022

7. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKind model checker. In: CAV 2018
8. Gacek, A., Katis, A., Whalen, M.W., Backes, J., Cofer, D.: Towards Realizability Checking of Contracts Using Theories. In: NFM. Springer (2015)
9. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Automated formalization of structured natural language requirements. Information and Software Technology **137**, 106590 (2021)
10. Heitmeyer, C.L., Archer, M., Bharadwaj, R., Jeffords, R.D.: Tools for constructing requirements specifications: the SCR toolset at the age of ten. Intl. Journal Comput. Syst. Sci. and Eng. **20**(1), 19–35 (2005)
11. Kanishege, J., Lombaerts, T., Shish, K., Feary, M.: Command and control concepts for a lift plus cruise electrical vertical takeoff and landing vehicle. In: AIAA Aviation Forum and Exposition, San Diego, CA (June 2023)
12. Katis, A., Fedyukovich, G., Guo, H., Gacek, A., Backes, J., Gurfinkel, A., Whalen, M.W.: Validity-guided synthesis of reactive systems from assume-guarantee contracts. In: TACAS (2018)
13. Katis, A., Mavridou, A., Giannakopoulou, D., Pressburger, T., Schumann, J.: Capture, analyze, diagnose: Realizability checking of requirements in FRET. In: CAV 2022
14. Könighofer, R., Hofferek, G., Bloem, R.: Debugging unrealizable specifications with model-based diagnosis. In: Haifa Verification Conference (2010)
15. Könighofer, R., Hofferek, G., Bloem, R.: Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. International Journal on Software Tools for Technology Transfer **15**(5-6), 563–583 (2013)
16. Larraz, D., Tinelli, C.: Realizability checking of contracts with Kind 2 (2022)
17. Letier, E., Heaven, W.: Requirements modelling by synthesis of deontic input-output automata. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 592–601. IEEE (2013)
18. Lúcio, L., Rahman, S., bin Abid, S., Mavin, A.: EARS-CTRL: Generating controllers for dummies. In: MODELS (Satellite Events). pp. 566–570 (2017)
19. Maoz, S., Ringert, J.O.: Synthesizing a lego forklift controller in gr (1): A case study. arXiv preprint arXiv:1602.01172 (2016)
20. Mavridou, A., Bourbouh, H., Giannakopoulou, D., Pressburger, T., Hejase, M., Garoche, P.L., Schumann, J.: The ten Lockheed Martin cyber-physical challenges: Formalized, analyzed, and explained. In: Proceedings of the 28th IEEE International Requirements Engineering Conference (2020)
21. Mavridou, A., Katis, A., Giannakopoulou, D., Kooi, D., Pressburger, T., Whalen, M.W.: From partial to global assume-guarantee contracts: Compositional realizability analysis in FRET. In: Formal Methods (2021)
22. Perez, I., Dedden, F., Goodloe, A.: Copilot 3. Tech. Rep. NASA/TM 2020-220587 (April 2020)
23. Perez, I., Mavridou, A., Pressburger, T., Goodloe, A., Giannakopoulou, D.: Automated translation of natural language requirements to runtime monitors. In: TACAS 2022
24. Pill, I., Semprini, S., Cavada, R., Roveri, M., Bloem, R., Cimatti, A.: Formal analysis of hardware requirements. In: DAC 2006
25. Pressburger, T., Katis, A., Dutle, A., Mavridou, A.: Using FRET to create, analyze and monitor requirements for a lift plus cruise case study. Tech. Rep. NASA/TM 20220017032 (2023)
26. Silva, C., Johnson, W.R., Solis, E., Patterson, M.D., Antcliff, K.R.: VTOL urban air mobility concept vehicles for technology development. In: AIAA 2018