NASA/TM–20220000049

Integrating FRET with Copilot: Automated Translation of Natural Language Requirements to Runtime Monitors

Ivan Perez
National Institute of Aerospace, Hampton, Virginia

Anastasia Mavridou
KBR, NASA Ames Research Center, Moffett Field, California

Thomas Pressburger
NASA Ames Research Center, Moffett Field, California

Alwyn Goodloe
NASA Langley Research Center, Hampton, Virginia

*Dimitra Giannakopoulou**
NASA Ames Research Center, Moffett Field, California
**Author contributed to this work prior to joining AWS*

January 2022

NASA STI Program . . . in Profile

NASA/TM–20220000049

Integrating FRET with Copilot: Automated Translation of Natural Language Requirements to Runtime Monitors

Ivan Perez
National Institute of Aerospace, Hampton, Virginia

Anastasia Mavridou
KBR, NASA Ames Research Center, Moffett Field, California

Thomas Pressburger
NASA Ames Research Center, Moffett Field, California

Alwyn Goodloe
NASA Langley Research Center, Hampton, Virginia

*Dimitra Giannakopoulou**
NASA Ames Research Center, Moffett Field, California
**Author contributed to this work prior to joining AWS*

National Aeronautics and
Space Administration

Ames Research Center, Moffett Field, CA 94035

January 2022

Executive Summary

Runtime verification (RV) enables monitoring systems at runtime, to detect property violations early and limit their potential consequences. To provide the level of assurance required for ultra-critical systems, monitor specifications must faithfully reflect the original mission requirements, which are often written in ambiguous natural language. This report presents an end-to-end framework to capture requirements in structured natural language and generate monitors that capture their semantics faithfully. We leverage NASA's Formal Requirement Elicitation Tool (FRET), and the RV system COPILOT. We extend FRET with mechanisms to capture additional information needed to generate monitors, and introduce OGMA, a new tool to bridge the gap between FRET and COPILOT. With this framework, users can write requirements in an intuitive format and obtain real-time C monitors suitable for use in embedded systems. Our toolchain is available as open source.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Safety-critical systems, such as aircraft, automobiles, and power systems, where failure can result in injury or death of a human [1], must undergo extensive assurance. The verification process must ensure that the system satisfies its requirements under realistic operating conditions and that there is no unintended behavior. Verification rests on possessing a precise statement of requirements, arguably one of the most difficult tasks in engineering reliable software. Formal verification techniques are, in principle, one method for achieving the level of reliability required in safety-critical systems. Although there have been considerable advances in industrial-scale formal methods, in most real-world scenarios it is not yet practical to apply formal methods to an entire system due to their exceedingly large complexity and the difficulty in constructing specifications.

Runtime verification (RV) [2, 3, 4] has the potential to enable the safe operation of complex safety-critical systems. RV monitors can be used to detect and respond to property violations during the mission, as well as to verify implementations and simulations at design time. For monitors to be effective, they must faithfully reflect the mission requirements, which is generally difficult for any non-trivial properties, since properties are normally expressed in temporal logic or programming code, and requirements in natural language.

The focus of this report , as shown in Figure 1.1, is to provide an end-to-end framework that takes as input requirements and other necessary data and provides mechanisms to 1) help the user deeply understand the semantics of these requirements, 2) automatically generate formalizations and 3) produce RV monitors that faithfully capture the semantics of the requirements. We leverage NASA's Formal Requirement Elicitation Tool (FRET) [5, 6] and the runtime monitoring system COPILOT [7, 8, 9]. FRET allows users to express and understand requirements through its intuitive structured natural language (named FRETISH) and elicitation mechanisms, and generates formalizations in temporal logic. COPILOT allows users to specify monitors and compile them to hard real-time C code.

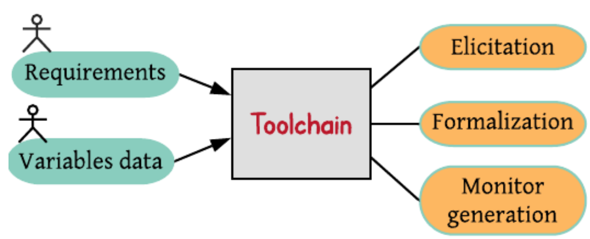

Figure 1.1: Framework overview

The contribution of this report is the tight integration of the FRET-COPILOT tools to support the automated synthesis of executable RV monitors directly from requirement specifications. In particular, we present:

- A new tool, named OGMA, that receives requirement formalizations and variable data from FRET and compiles these into COPILOT monitors.

- An extension of the FRET analysis portal to support the generation and export of specifications that can be directly digested by OGMA.
- Preliminary experimental results that evaluate the proposed workflow.

All tools needed by our workflow are available as open source [10, 11, 12].

Related Work. A number of runtime verification languages and systems have been applied in resource-constrained environments [13, 14, 15, 16, 17, 18]. In contrast to our work, these systems do not provide a direct translation from natural language. Several tools [19, 20, 21, 22, 23] formalize natural-language like requirements, but not for the purpose of generating runtime monitors. The STIMULUS tool [24] allows users to express requirements in an extensible, natural-like language that is syntactic sugar for hierarchical state machines. The machines then act as monitors that can be used to validate requirements during the design and testing phases, but are not intended to be used at runtime. FLEA [25] is a formal language for expressing requirements that compiles to runtime monitors in a garbage collected language, making it harder to use in embedded systems; in contrast, our approach generates hard real-time code.

Chapter 2

Step-by-step Framework Workflow

To integrate FRET and COPILOT, we extended the FRET analysis portal and created the OGMA tool. Figure 2.1 shows the step-by-step workflow of the complete framework - dashed lines represent the newly added steps (2, 3, and 4). Once requirements are written in FRETISH, FRET helps users understand and refine their requirements through various explanations and simulation (step 0). Next, FRET automatically translates requirements (step 1) into pure Past-time Metric Linear Temporal Logic (pmLTL) formulas. Next, information about the variables referenced in the requirements must be provided by the user (step 2). The formulas, as well as the provided variables' data, are then combined to generate the Component Specification (step 3). Based on this specification, OGMA creates a complete COPILOT monitor specification (step 4). COPILOT then generates the C Monitor (step 5), which is given along with other C code (step 6) to a C Compiler for the generation (step 7) of the final object code.

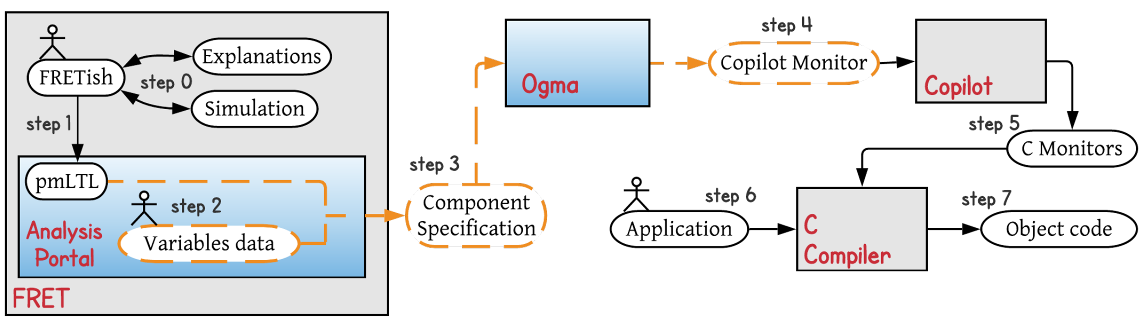

Figure 2.1: Step-by-step workflow

Running Example. The next sections illustrate each workflow step using a flight-critical system requirement: airplanes should always avoid stalling (a stall is a sudden loss of lift, which may lead to a loss of control). To avoid stalls, they should fly above a certain speed, known as *stall speed* (as well as stay below a critical angle of attack). Our running requirement example is captured in natural language in Figure 2.2. For the purposes of this example, we consider the airspeed threshold to be 100 m/s and the correction time to be 10 seconds.

NL: "*While flying, if the airspeed is below 100 m/s, the autopilot shall increase the airspeed to at least 100 m/s within 10 seconds.*"

FRETish: `in flight mode if airspeed < 100 the aircraft shall within 10 seconds satisfy (airspeed >= 100)`

pmLTL: `H (Lin_flight→(Y (((O`$_{[=10]}$`(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → (O`$_{[<10]}$`(Fin_flight | (airspeed ≥ 100)))) S (((O`$_{[=10]}$`(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → (O`$_{[<10]}$`(Fin_flight | (airspeed ≥ 100)))) & Fin_flight)))) & ((!Lin_flight) S ((!Lin_flight) & Fin_flight)) → (((O`$_{[=10]}$`(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → (O`$_{[<10]}$`(Fin_flight | (airspeed ≥ 100)))) S (((O`$_{[=10]}$`(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → (O`$_{[<10]}$`(Fin_flight | (airspeed ≥ 100)))) & Fin_flight)),`

where `Fin_flight` (First timepoint in flight mode) is `flight & (FTP | Y !flight)`, `Lin_flight` (Last timepoint in flight mode) is `!flight & Y flight`, `FTP` (First Time Point) is `! Y true`.

Figure 2.2: Running example in Natural Language (NL), FRETISH, and pmLTL forms.

Chapter 3

FRET Steps

Next we discuss FRET, the requirements tool that constitutes our frontend.

Step 0: fretish and semantic nuances. A FRETISH requirement (see running example in Figure 2.2) contains up to six fields: `scope`, `condition`, `component*`, `shall*`, `timing`, and `response*`. Fields marked with * are mandatory.

`component` specifies the component that the requirement refers to (e.g., aircraft). `shall` expresses that the component's behavior must conform to the requirement. `response` is of the form *satisfy* R, where R is a Boolean condition (e.g., satisfy airspeed $\geq$ 100). `scope` specifies the period when the requirement holds during the execution of the system, e.g., when "in flight mode". `condition` is a Boolean expression that further constrains when the `response` shall occur (e.g., the requirement becomes relevant only upon airspeed $\leq$ 100 becoming true). `timing` specifies when the `response` must occur (e.g., within 10 seconds).

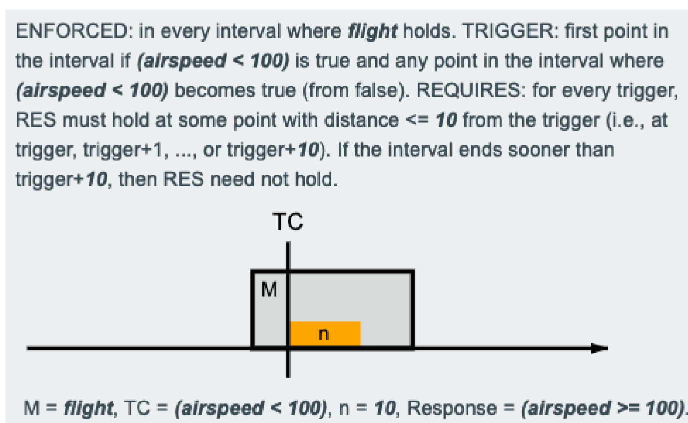

Figure 3.1: FRET explanations

Getting a temporal requirement right is usually a tricky task since such requirements are often riddled with semantic subtleties. To help the user, FRET provides a simulator and semantic explanations [5]. For example, the diagram in Figure 3.1 explains that the requirement is only relevant within the grayed box M (while in flight mode). TC represents the triggering condition (airspeed $<$ 100) and the orange band, with a duration of n=10 seconds, states that the response (airspeed $>=$ 100) is required to hold at least once within the 10 seconds duration, assuming that flight mode holds for at least 10 seconds.

Step 1: fretish to pmLTL. For each FRETISH requirement, FRET generates formulas in a variety of formalisms. For the COPILOT integration, we use the generated pmLTL formulas (Figure 2.2) Clearly, manually writing such formulas can be quite error-prone, while the FRET formalization process has been extensively tested through its *formalization verifier* [5].

Steps 2 & 3: Variables data and Component Specification. We extended FRET's analysis portal [26] to capture the information needed to generate Component Specifications for OGMA. To generate a specification, the user must indicate the type (i.e., input, output, internal) and data type (integer, Boolean, double, etc) of each variable (Figure 3.2). Internal variables represent expressions of input and output variables; if the same expression is used in multiple requirements, an internal variable can be used to substitute it and simplify the requirements. The user must *assign* an expression to each internal variable. In our example, the `flight` internal variable is defined by the expression `altitude > 0.0`, where `altitude` is an input variable. Internal variable assignments can be defined in Lustre [27] or Copilot [7]. Integrated Lustre and Copilot parsers identify parsing errors and return feedback (Figure 3.2). Note that FRET asks users for variables data only for the connection with analysis tools (e.g., Copilot). Other FRET functionalities such as requirement formalization do not require this information. Once steps 1 and 2 are completed, FRET generates a Component Specification, which contains all requirements in pmLTL and Lustre code, as well as variable data that belong to the same system component.

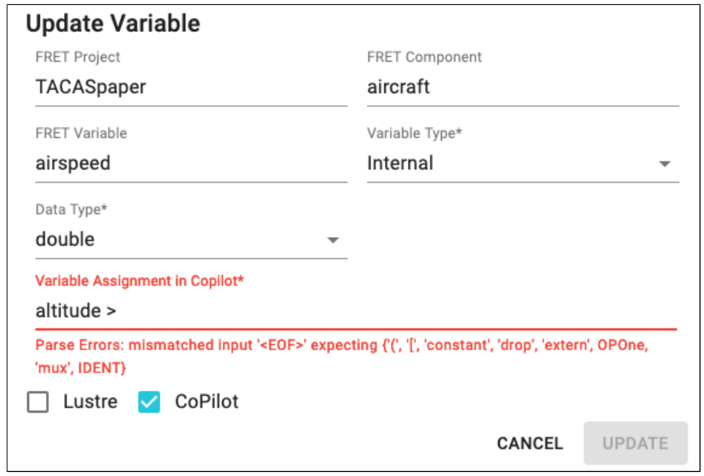

Figure 3.2: FRET variable editor

Chapter 4

Ogma Steps

OGMA is a command-line tool to produce monitoring applications. OGMA generates monitors in COPILOT, and also supports integrating them into larger systems, such as applications built with NASA's core Flight System (cFS) [28].

Step 4: Copilot Monitors. OGMA provides a command `fret-component-spec` to process Component Specifications and generates a corresponding Copilot specification. For example:

```
$ ogma fret-component-spec --fret-file-name reqs.json > Monitor.hs
```

The command traverses the Abstract Syntax Tree of the Component Specification, and converts each tree node into its COPILOT counterpart. Input and output variables in FRET become *extern* streams in COPILOT, or time-varying sources of information needed by the monitors:

```
airspeed :: Stream Double
airspeed = extern "airspeed" Nothing

flight :: Stream Bool
flight = extern "flight" Nothing
```

Internal variables are also mapped to streams. Each requirement's pmLTL formula is translated into a Boolean stream, paired with a C handler *triggered* when the requirement is violated. In the example below, the property we monitor is associated with a handler, `handlerpropAvoidStall`, which must be implemented separately in C by the user to determine how to address property violations:

```
propAvoidStall :: Stream Bool
propAvoidStall = ((PTLTL.alwaysBeen ((((not (flight)) && ... )))))

spec :: Spec
spec = do
  trigger "handlerpropAvoidStall" (not propAvoidStall) []
```

Chapter 5

Copilot Steps

Copilot is a stream-based runtime monitoring language. Copilot streams may contain data of different types. At the top level, specifications consist of pairs of Boolean streams, together with a C handler to be called when the current sample of a stream becomes true. For a detailed introduction to Copilot, see [7].

Step 5: C Monitors. Ogma generates self-contained Copilot monitoring specifications, which can be further compiled into C99 by just compiling and running the Copilot specifications with a Haskell compiler. This process produces two files: a C header and a C implementation.

Step 6: Larger Applications. The C files generated by Copilot are designed to be integrated into larger applications. They provide three connections end-points: extern variables, a `step` function, and handler functions, which users implement to handle property violations. The code generated has no dynamic memory allocation, loops or recursive calls, it executes in predictable memory and time. For our running example, the header file generated by Copilot declares:

```
extern bool flight;
extern float airspeed;

void handlerpropAvoidStall(void);
void step(void);
```

Users are not expected to modify the files generated by Copilot, but simply use the above interface to connect them to the system being monitored.

Commonly, the calling application will poll sensors, write their values to global variables (in the example above, `flight` and `airspeed`), call the `step` function, and implement handlers that log property violations or execute corrective actions (i.e., `handlerpropAvoidStall`). Users are responsible for compiling and linking the Copilot code together with their application (step 7).

We used the running requirement in this report to monitor a flight in the simulator X-Plane. We wrote an X-Plane plugin to show the state of the C monitor and some additional information on the screen (Fig. 5.1). To test the code, we brought an aircraft to a stall by increasing the angle of attack, which also lowered the airspeed (Fig. 5.2). After 10 seconds below the specified threshold, the monitor became active, remaining on after executing a stall recovery (Fig. 5.3).

Figure 5.1: Demonstration of COPILOT monitor running as X-Plane plugin: cruising.

Figure 5.2: Demonstration of COPILOT monitor running as X-Plane plugin: stall.

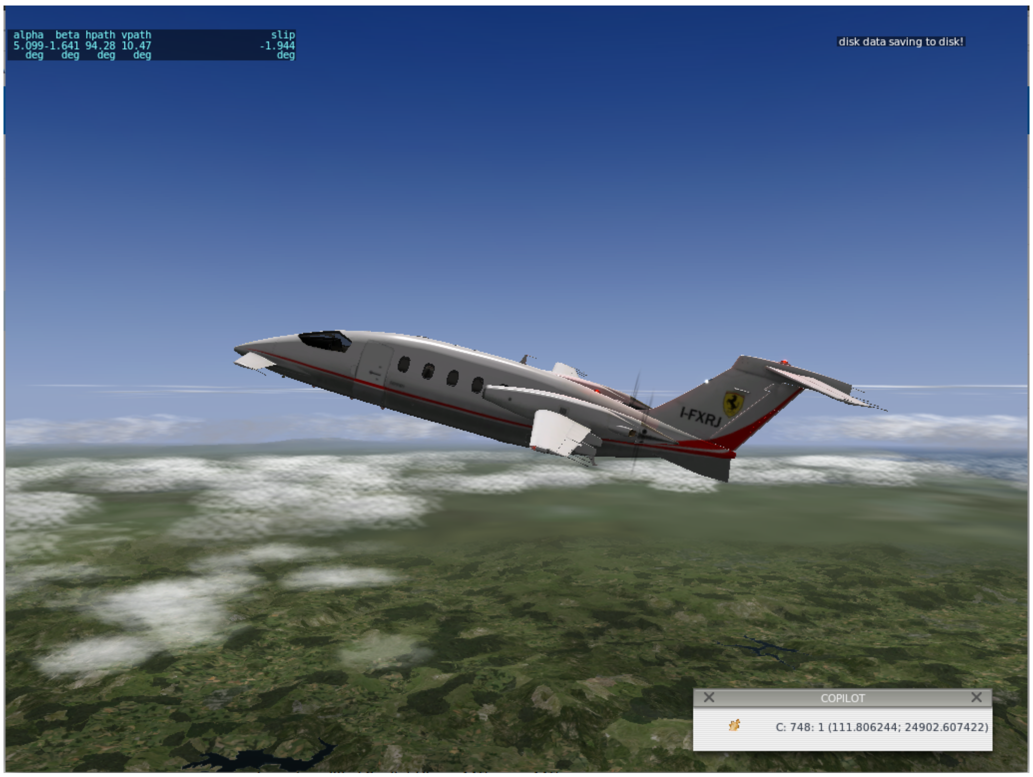

Figure 5.3: Demonstration of COPILOT monitor running as X-Plane plugin: recovery.

Chapter 6

Preliminary Results

We report on experiments with monitors generated from the publicly available Lockheed Martin Cyber-Physical System (LMCPS) challenge problems [29, 30], which are a set of industrial Simulink model benchmarks and natural language requirements developed by domain experts. LMCPS requirements were previously written in FRETISH [31, 32] by a subset of the authors and were analyzed against the provided models using model checking.

In this report, we reuse the FRETISH requirements to generate monitors and compare our runtime verification results with the model checking results of [32]. We experimented with the Finite State Machine (FSM) and the Control Loop Regulators (REG) LMCPS challenges. We used FRET to generate the Component Specifications for both LMCPS challenges. We provide all FRET-generated Component Specifications in the Appendix. For each Simulink model we generated C code through the automatic code generation feature of Matlab/Simulink. We then attached the generated C monitors to the C code and used the property-based testing system QuickCheck [33] to generate random streams of data, feed them to the system under observation, and report if any of the monitors were activated, based on [34, 35, 36].

For both LMCPS challenges, our results are consistent with the model checking results - QuickCheck was able to find input vectors that activated the monitors, indicating that certain requirements are not satisfied. Additionally, we were able to return results within seconds in cases where model checkers timed out. See [37] for a reproducible artifact.

Table 6.1 shows the model checking results of the Regulators (REG) LMCPS challenge problem as reported in [32], where we used the Kind2 [38] SMT-based model checker. Column four shows the analysis results from runtime monitoring the same requirements for 2000 inputs. As shown in Table 6.1, Kind2 was able to return a result for most requirements and timed out for requirements REG-001, REG-002, REG-004, REG-005.

Through the approach presented in this report, we generated a monitor per requirement. For runtime verification we used QuickCheck to generate input vectors - we tested the C system code with 2000 different inputs. Our monitors were activated for requirements REG-006, REG-007, REG-008, REG-009, and REG-010, a result consistent with the KIND 2 results. For requirements, REG-001 to REG-005 the corresponding monitors were not activated for any of the 2000 inputs.

Similarly, Table 6.2 shows the model checking results of the Finite State Machine (FSM) LMCPS challenge problem as reported in [32], where we used the Kind2 [38] SMT-based model checker. Column four shows the analysis results from runtime monitoring the same FRETISH requirements for 2000 inputs.

Table 6.1: Regulators analysis results with the Kind2 (abbr. by K) model checker and runtime monitors. Timeout was set to 2 hours for the model checkers and the monitors were tested for up to 2000 different inputs.

Requirement	K Result	K Time	Monitors
[REG-001]	Undecided	Timeout	Non-Activated
[REG-002]	Undecided	Timeout	Non-Activated
[REG-003]	Valid	10.046 sec	Non-Activated
[REG-004]	Undecided	Timeout	Non-Activated
[REG-005]	Undecided	Timeout	Non-Activated
[REG-006]	Invalid	5.998 sec	Activated
[REG-007]	Invalid	5.998 sec	Activated
[REG-008]	Invalid	5.998 sec	Activated
[REG-009]	Invalid	5.998 sec	Activated
[REG-010]	Invalid	5.998 sec	Activated
Total running time	CoCoSim: Timeout		2000 inputs

Table 6.2: FSM analysis results with the Kind2 (abbr. by K) model checker and runtime monitors. Timeout was set to 2 hours for the model checkers and the monitors were tested for 2000 different inputs.

Requirement	K Result	K Time	Monitors
[FSM-001v1]	Invalid	0.254 sec	Activated
[FSM-001v2]	Invalid	0.465 sec	Activated
[FSM-001v3]	true up to 28 steps	timeout (2h)	Non-Activated
[FSM-002]	Valid	0.252 sec	Non-Activated
[FSM-003]	Invalid	0.191 sec	Activated
[FSM-004]	Invalid	0.191 sec	Activated
[FSM-005]	Valid	0.252 sec	Non-Activated
[FSM-006]	Valid	0.252 sec	Non-Activated
[FSM-007]	Invalid	0.135 sec	Activated
[FSM-007v2]	Valid	0.252 sec	Non-Activated
[FSM-008v1]	Invalid	0.165 sec	Activated
[FSM-009]	Valid	0.252 sec	Non-Activated
[FSM-010]	Valid	0.132 sec	Non-Activated
[FSM-011v1]	Invalid	0.110 sec	Activated
[FSM-011v2]	Valid	0.132 sec	Non-Activated
[FSM-012]	Valid	0.132 sec	Non-Activated
[FSM-013]	Valid	0.132 sec	Non-Activated
Total running time	CoCoSim: 141.09sec		2000 inputs

Chapter 7

Conclusion

We described an end-to-end framework in which requirements written in structured natural language can be equivalently transformed into monitors and be analyzed against C code. Our framework ensures that requirements and analysis activities are fully aligned: C monitors are derived directly from requirements and not handcrafted. The design of our toolchain facilitates extension with additional front-ends (e.g., JKind Lustre [39]), and backends (e.g., R2U2 [40]). In the future, we plan to explore more use cases, including some from real drone test flights.

Bibliography

1. Knight, J. C.: Safety Critical Systems: Challenges and Directions. *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, ACM, 2002, pp. 547–550.

2. Havelund, K.; and Goldberg, A.: *Verify Your Runs*, Springer Berlin Heidelberg, Berlin, Heidelberg. 2008, pp. 374–383.

3. Goodloe, A.; and Pike, L.: Monitoring Distributed Real-Time Systems: A Survey and Future Directions. NASA/CR-2010-216724, NASA Langley Research Center, July 2010.

4. Bartocci, E.; Falcone, Y.; Francalanza, A.; and Reger, G.: Introduction to Runtime Verification. *Lectures on Runtime Verification - Introductory and Advanced Topics*, Springer, vol. 10457 of *Lecture Notes in Computer Science*, 2018, pp. 1–33.

5. Giannakopoulou, D.; Pressburger, T.; Mavridou, A.; Rhein, J.; Schumann, J.; and Shi, N.: Formal Requirements Elicitation with FRET. *Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020)*, 2020.

6. Giannakopoulou, D.; Pressburger, T.; Mavridou, A.; and Schumann, J.: Automated formalization of structured natural language requirements. *Inf. Softw. Technol.*, vol. 137, 2021, p. 106590.

7. Perez, I.; Dedden, F.; and Goodloe, A.: Copilot 3. NASA/TM–2020–220587, NASA Langley Research Center, April 2020.

8. Pike, L.; Wegmann, N.; Niller, S.; and Goodloe, A.: Copilot: Monitoring Embedded Systems. *Innov. Syst. Softw. Eng.*, vol. 9, no. 4, Dec. 2013, p. 235–255.

9. Pike, L.; Goodloe, A.; Morisset, R.; and Niller, S.: Copilot: A Hard Real-Time Runtime Monitor. *Proceedings of the 1st Intl. Conference on Runtime Verification*, LNCS, Springer, November 2010.

10. FRET: Formal Requirements Elicitation Tool. `https://github.com/NASA-SW-VnV/fret/`. Accessed Oct 04, 2021.

11. Copilot. `https://github.com/Copilot-Language/copilot/`. Accessed Oct 04, 2021.

12. Ogma. `https://github.com/nasa/ogma/`. Accessed Oct 04, 2021.

13. Torfah, H.: Stream-Based Monitors for Real-Time Properties. *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, B. Finkbeiner and L. Mariani, eds., Springer, vol. 11757 of *Lecture Notes in Computer Science*, 2019, pp. 91–110.

14. Faymonville, P.; Finkbeiner, B.; Schledjewski, M.; Schwenger, M.; Stenger, M.; Tentrup, L.; and Torfah, H.: StreamLAB: Stream-based Monitoring of Cyber-Physical Systems. *Computer Aided Verification*, I. Dillig and S. Tasiran, eds., Springer International Publishing, Cham, 2019, pp. 421–431.

15. Baumeister, J.; Finkbeiner, B.; Schirmer, S.; Schwenger, M.; and Torens, C.: RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft. *Computer Aided Verification*, S. K. Lahiri and C. Wang, eds., Springer International Publishing, Cham, 2020, pp. 28–39.

16. Biewer, S.; Finkbeiner, B.; Hermanns, H.; Köhl, M. A.; Schnitzer, Y.; and Schwenger, M.: RTLola on Board: Testing Real Driving Emissions on your Phone. *Tools and Algorithms for the Construction and Analysis of Systems*, J. F. Groote and K. G. Larsen, eds., Springer International Publishing, Cham, 2021, pp. 365–372.

17. Reinbacher, T.; Rozier, K. Y.; and Schumann, J.: Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems. *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, eds., Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 357–372.

18. Moosbrugger, P.; Rozier, K. Y.; and Schumann, J.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods in System Design*, vol. 51, no. 1, 2017, pp. 31–61.

19. Lúcio, L.; Rahman, S.; Cheng, C.-H.; and Mavin, A.: Just Formal Enough? Automated Analysis of EARS Requirements. *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, May 2017, pp. 427–434.

20. Fifarek, A.; Wagner, L. G.; Hoffman, J. A.; Rodes, B. D.; Aiello, M. A.; and Davis, J. A.: SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements. *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, 2017, pp. 420–426.

21. Ghosh, S.; Elenius, D.; Li, W.; Lincoln, P.; Shankar, N.; and Steiner, W.: ARSENAL: Automatic Requirements Specification Extraction from Natural Language. *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, S. Rayadurgam and O. Tkachuk, eds., Springer, vol. 9690 of *Lecture Notes in Computer Science*, 2016, pp. 41–46.

22. Lignos, C.; Raman, V.; Finucane, C.; Marcus, M.; and Kress-Gazit, H.: Provably Correct Reactive Control from Natural Language. *Auton. Robots*, vol. 38, no. 1, jan 2015, p. 89–105.

23. Boteanu, A.; Howard, T.; Arkin, J.; and Kress-Gazit, H.: A model for verifiable grounding and execution of complex natural language instructions. *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 2649–2654.

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

25. Cohen, D.; Feather, M. S.; Narayanaswamy, K.; and Fickas, S. S.: Automatic monitoring of software requirements. *Proceedings of the 19th International Conference on Software Engineering*, 1997, pp. 602–603.

26. FRET: Formal Requirements Elicitation Tool - User Manual. `https://github.com/NASA-SW-VnV/fret/blob/master/fret-electron/docs/_media/userManual.md`. See Section "Exporting for Analysis". Accessed Oct 04, 2021.

27. Halbwachs, N.; Caspi, P.; Raymond, P.; and Pilaud, D.: The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, vol. 79, no. 9, 1991, pp. 1305–1320.

28. Wilmot, J.: A Core Flight Software System. *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '05, ACM, New York, NY, USA, 2005, pp. 13–14.

29. Elliott, C.: 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. *Safe & Secure Systems and Software Symposium (S5)*, A. F. R. Laboratory, ed., 2015.

30. Elliott, C.: An Example Set of Cyber-Physical V&V Challenges for S5, Lockheed Martin Skunk Works. *Safe & Secure Systems and Software Symposium (S5)*, A. F. R. Laboratory, ed., 2016.

31. Mavridou, A.; Bourbouh, H.; Giannakopoulou, D.; Pressburger, T.; Hejase, M.; Garoche, P.-L.; and Schumann, J.: The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained. *2020 IEEE 28th International Requirements Engineering Conference (RE)*, 2020, pp. 300–310.

32. Mavridou, A.; Bourbouh, H.; Garoche, P. L.; and Hejase, M.: Evaluation of the FRET and CoCoSim tools on the Ten Lockheed Martin Cyber-Physical Challenge Problems. TM-2019-220374, National Aeronautics and Space Administration, February 2020.

33. Claessen, K.; and Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM Sigplan Notices*, vol. 46, no. 4, 2011, pp. 53–64.

34. Perez, I.; Goodloe, A.; and Edmonson, W.: Fault-tolerant swarms. *2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, IEEE, 2019, pp. 47–54.

35. Perez, I.; and Goodloe, A. E.: Fault-tolerant functional reactive programming (extended version). *Journal of Functional Programming*, vol. 30, 2020.

36. Perez, I.; and Nilsson, H.: Runtime verification and validation of functional reactive systems. *Journal of Functional Programming*, vol. 30, 2020, p. e28.

37. Perez, I.; Mavridou, A.; Pressburger, T.; Goodloe, A.; and Giannakopoulou, D.: Artifact for Automated Translation of Natural Language Requirements to Runtime Monitors. `https://doi.org/10.5281/zenodo.5888956`.

38. Champion, A.; Mebsout, A.; Sticksel, C.; and Tinelli, C.: The Kind 2 model checker. *International Conference on Computer Aided Verification*, Springer, 2016, pp. 510–517.

39. Gacek, A.; Backes, J.; Whalen, M.; Wagner, L.; and Ghassabani, E.: The JK ind model checker. *International Conference on Computer Aided Verification*, Springer, 2018, pp. 20–27.

40. Schumann, J.; Moosbrugger, P.; and Rozier, K. Y.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Runtime Verification*, Springer, 2015, pp. 233–249.

Appendix

7.1 FSM Specification

```
{
  "FSMSpec": {
    "Internal_variables": [
      {
        "name": "autopilot",
        "type": "bool",
        "assignmentLustre": "(! standby) & supported & (! apfail)",
        "assignmentCopilot": ""
      },
      {
        "name": "htlore3_autopilot",
        "type": "bool",
        "assignmentLustre": "HT(3,0,autopilot)",
        "assignmentCopilot": ""
      },
      {
        "name": "htlore3_notpreprelimits",
        "type": "bool",
        "assignmentLustre": "HT(3,0,(false  -> pre (false  -> not pre_limits)))",
        "assignmentCopilot": ""
      },
      {
        "name": "pre_autopilot",
        "type": "bool",
        "assignmentLustre": "false -> pre autopilot",
        "assignmentCopilot": ""
      },
      {
        "name": "pre_limits",
        "type": "bool",
        "assignmentLustre": "false -> pre limits;",
        "assignmentCopilot": ""
      }
    ],
    "Other_variables": [
      {
        "name": "apfail",
        "type": "bool"
      },
      {
        "name": "limits",
        "type": "bool"
      },
      {
        "name": "standby",
        "type": "bool"
      },
      {
        "name": "supported",
        "type": "bool"
      },
```

```
      {
        "name": "pullup",
        "type": "bool"
      }
    ],
    "Functions": [],
    "Requirements": [
    {
      "name": "FSM-001v1",
      "ptLTL": "(H (( limits & (! standby) & (! apfail) & supported ) -> pullup))",
      "CoCoSpecCode": "(H((( limits and not standby and not apfail and supported ) => pullup)))",
      "fretish": "FSM  shall  always  satisfy (limits & !standby & !apfail & supported) => pullup"
    },
      {
        "name": "FSM-001v2",
        "ptLTL": "(H (((autopilot & pre_autopilot & pre_limits) & ((Y (! (autopilot & pre_autopilot & pre_limits))) | (! (Y TRUE)))) -> (pullup)))",
        "CoCoSpecCode": "(H((((autopilot and pre_autopilot and pre_limits) and (false -> pre (not (autopilot and pre_autopilot and pre_limits)))) or ((autopilot and pre_autopilot and pre_limits) and FTP)) => (pullup)))",
        "fretish": "if autopilot & pre_autopilot & pre_limits FSM shall immediately satisfy pullup"
      },
      {
        "name": "FSM-001v3",
        "ptLTL": "(H (((htlore3_autopilot & htlore3_notpreprelimits & pre_limits) & ((Y (! (htlore3_autopilot & htlore3_notpreprelimits & pre_limits))) | (! (Y TRUE)))) -> (pullup)))",
        "CoCoSpecCode": "(H(((htlore3_autopilot and htlore3_notpreprelimits and pre_limits) and ((pre ( not (htlore3_autopilot and htlore3_notpreprelimits and pre_limits))) or FTP)) => (pullup)))",
        "fretish": "if  htlore3_autopilot  &  htlore3_notpreprelimits  &  pre_limits FSM  shall  immediately satisfy pullup"
      }
    ]
  }
}
```

7.2 FSM Manager Specification

```
{
  "FSM_AutopilotSpec": {
    "Internal_variables": [
      {
        "name": "ap_maneuver_state",
        "type": "real",
        "assignmentLustre": "2.0;",
        "assignmentCopilot": ""
      },
      {
        "name": "ap_nominal_state",
        "type": "real",
        "assignmentLustre": "1.0;",
        "assignmentCopilot": ""
      },
      {
        "name": "ap_standby_state",
        "type": "real",
        "assignmentLustre": "3.0;",
        "assignmentCopilot": ""
      },
      {
        "name": "ap_transition_state",
```

```
        "type": "real",
        "assignmentLustre": "0.0;",
        "assignmentCopilot": ""
      }
    ],
    "Other_variables": [
      {
        "name": "apfail",
        "type": "bool"
      },
      {
        "name": "good",
        "type": "bool"
      },
      {
        "name": "standby",
        "type": "bool"
      },
      {
        "name": "state",
        "type": "real"
      },
      {
        "name": "supported",
        "type": "bool"
      },
      {
        "name": "STATE",
        "type": "int"
      }
    ],
    "Functions": [],
    "Requirements": [
      {
        "name": "FSM-002",
        "ptLTL": "(H (( standby & state = ap_transition_state ) -> STATE =
            ap_standby_state))",
        "CoCoSpecCode": "(H((( standby and state = ap_transition_state ) => STATE =
            ap_standby_state)))",
        "fretish": "FSM_Autopilot shall always satisfy (standby & state =
            ap_transition_state) => STATE = ap_standby_state"
      },
      {
        "name": "FSM-005",
        "ptLTL": "(H (( state = ap_nominal_state & standby ) -> STATE =
            ap_standby_state))",
        "CoCoSpecCode": "(H((( state = ap_nominal_state and standby ) => STATE =
            ap_standby_state)))",
        "fretish": "FSM_Autopilot shall always satisfy (state=ap_nominal_state &
            standby)  => STATE = ap_standby_state"
      },
      {
        "name": "FSM-003",
        "ptLTL": "(H (( state = ap_transition_state & good & supported ) -> STATE =
            ap_nominal_state))",
        "CoCoSpecCode": "(H((( state = ap_transition_state and good and supported )
            => STATE = ap_nominal_state)))",
        "fretish": "FSM_Autopilot shall always satisfy (state = ap_transition_state &
             good & supported) => STATE = ap_nominal_state"
      },
      {
        "name": "FSM-008v1",
        "ptLTL": "(H (( state = ap_standby_state & ! standby ) -> STATE =
            ap_transition_state))",
        "CoCoSpecCode": "(H((( state = ap_standby_state and not standby ) => STATE =
            ap_transition_state)))",
        "fretish": " FSM_Autopilot shall always satisfy (state = ap_standby_state & !
            standby) => STATE = ap_transition_state"
```

```
      },
      {
        "name": "FSM-009",
        "ptLTL": "(H (( state = ap_standby_state & apfail ) -> STATE = ap_maneuver_state))",
        "CoCoSpecCode": "(H((( state = ap_standby_state and apfail ) => STATE = ap_maneuver_state)))",
        "fretish": "FSM_Autopilot shall always satisfy (state = ap_standby_state & apfail )=> STATE = ap_maneuver_state"
      },
      {
        "name": "FSM-004v2",
        "ptLTL": "(H (( state = ap_nominal_state & ! good & ! standby ) -> STATE = ap_maneuver_state))",
        "CoCoSpecCode": "(H((( state = ap_nominal_state and not good and not standby ) => STATE = ap_maneuver_state)))",
        "fretish": "FSM_Autopilot shall always satisfy (state = ap_nominal_state & ! good & ! standby) => STATE = ap_maneuver_state"
      },
      {
        "name": "FSM-008v2",
        "ptLTL": "(H (( state = ap_standby_state & ! standby & ! apfail ) -> STATE = ap_transition_state))",
        "CoCoSpecCode": "(H((( state = ap_standby_state and not standby and not apfail ) => STATE = ap_transition_state)))",
        "fretish": "FSM_Autopilot shall always satisfy (state = ap_standby_state & ! standby & ! apfail ) => STATE = ap_transition_state"
      },
      {
        "name": "FSM-007",
        "ptLTL": "(H (( state = ap_maneuver_state & supported & good ) -> STATE = ap_transition_state))",
        "CoCoSpecCode": "(H((( state = ap_maneuver_state and supported and good ) => STATE = ap_transition_state)))",
        "fretish": "FSM_Autopilot shall always satisfy (state = ap_maneuver_state & supported & good) => STATE = ap_transition_state"
      },
      {
        "name": "FSM-004",
        "ptLTL": "(H (( ! good & state = ap_nominal_state ) -> STATE = ap_maneuver_state))",
        "CoCoSpecCode": "(H((( not good and state = ap_nominal_state ) => STATE = ap_maneuver_state)))",
        "fretish": "FSM_Autopilot shall always satisfy (! good & state = ap_nominal_state) => STATE = ap_maneuver_state"
      },
      {
        "name": "FSM-007v2",
        "ptLTL": "(H (( state = ap_maneuver_state & supported & good & ! standby ) -> STATE = ap_transition_state))",
        "CoCoSpecCode": "(H((( state = ap_maneuver_state and supported and good and not standby ) => STATE = ap_transition_state)))",
        "fretish": "FSM_Autopilot shall always satisfy (state = ap_maneuver_state & supported & good & ! standby) => STATE = ap_transition_state"
      },
      {
        "name": "FSM-006",
        "ptLTL": "(H (( state = ap_maneuver_state & standby & good ) -> STATE = ap_standby_state))",
        "CoCoSpecCode": "(H((( state = ap_maneuver_state and standby and good ) => STATE = ap_standby_state)))",
        "fretish": "FSM_Autopilot shall always satisfy (state = ap_maneuver_state & standby & good) => STATE = ap_standby_state"
      }
    ]
  }
}
```

7.3 FSM Sensor pecification

```
{
  "FSM_SensorSpec": {
    "Internal_variables": [
      {
        "name": "sen_fault_state",
        "type": "real",
        "assignmentLustre": "2.0",
        "assignmentCopilot": ""
      },
      {
        "name": "sen_nominal_state",
        "type": "real",
        "assignmentLustre": "0.0",
        "assignmentCopilot": ""
      },
      {
        "name": "sen_transition_state",
        "type": "real",
        "assignmentLustre": "1.0",
        "assignmentCopilot": ""
      }
    ],
    "Other_variables": [
      {
        "name": "limits",
        "type": "bool"
      },
      {
        "name": "request",
        "type": "bool"
      },
      {
        "name": "senstate",
        "type": "real"
      },
      {
        "name": "MODE",
        "type": "bool"
      },
      {
        "name": "SENSTATE",
        "type": "real"
      }
    ],
    "Functions": [],
    "Requirements": [
      {
        "name": "FSM-010",
        "ptLTL": "(H (( (senstate = sen_nominal_state) & limits ) -> SENSTATE = sen_fault_state))",
        "CoCoSpecCode": "(H((( (senstate = sen_nominal_state) and limits ) => SENSTATE = sen_fault_state)))",
        "fretish": "FSM_Sensor shall always satisfy (senstate = sen_nominal_state & limits) => SENSTATE = sen_fault_state"
      },
      {
        "name": "FSM-011v1",
        "ptLTL": "(H (( (senstate = sen_nominal_state) & (! request) ) -> SENSTATE = sen_transition_state))",
        "CoCoSpecCode": "(H((( ( senstate = sen_nominal_state ) and not request ) => SENSTATE = sen_transition_state)))",
        "fretish": "FSM_Sensor shall always  satisfy (senstate = sen_nominal_state & (!request)) => SENSTATE = sen_transition_state"
      },
```

```
      {
        "name": "FSM-011v2",
        "ptLTL": "(H (( ( senstate = sen_nominal_state ) & (! request) & (! limits) )
             -> SENSTATE = sen_transition_state))",
        "CoCoSpecCode": "(H((( ( senstate = sen_nominal_state ) and not request and
            not limits ) => SENSTATE = sen_transition_state)))",
        "fretish": "FSM_Sensor shall always satisfy (senstate = sen_nominal_state &
            (!request) & (!limits)) => SENSTATE = sen_transition_state"
      },
      {
        "name": "FSM-012",
        "ptLTL": "(H (( ( senstate = sen_fault_state ) & (! request) & (! limits) )
            -> SENSTATE = sen_transition_state))",
        "CoCoSpecCode": "(H((( ( senstate = sen_fault_state ) and not request and not
             limits ) => SENSTATE = sen_transition_state)))",
        "fretish": "FSM_Sensor shall  always  satisfy (senstate = sen_fault_state &
            (!request) & (!limits)) => SENSTATE = sen_transition_state"
      },
      {
        "name": "FSM-013",
        "ptLTL": "(H (( ( senstate = sen_transition_state ) & request & MODE ) ->
            SENSTATE = sen_nominal_state))",
        "CoCoSpecCode": "(H((( ( senstate = sen_transition_state ) and request and
            MODE ) => SENSTATE = sen_nominal_state)))",
        "fretish": "FSM_Sensor shall always satisfy (senstate = sen_transition_state
            & request & MODE) => SENSTATE = sen_nominal_state"
      }
    ]
  }
}
i
```

7.4 Regulators Specification

```

{
  "RegulatorSpec":
  {

"Internal_variables":
[
    {
      "name": "count_roll_output_exceeding_50",
      "type": "int",
      "assignmentLustre": "0",
      "assignmentCopilot": "0"
    },
    {
      "name": "count_pitch_output_exceeding_50",
      "type": "int",
      "assignmentLustre": "0 -> if (mcvdt_cmd_fcs_dps2  > 50.0)  then  pre
          count_pitch_output_exceeding_50 + 1 else 0",
      "assignmentCopilot": "mux (mcvdt_cmd_fcs_dps2 > 50.0) (([0] ++
          count_pitch_output_exceeding_50) + 1) 0"
    },
    {
      "name": "count_yaw_output_exceeding_50",
      "type": "int",
      "assignmentLustre": "0 -> if (ncvdt_cmd_fcs_dps2  > 50.0)  then  pre
          count_yaw_output_exceeding_50 + 1 else 0",
      "assignmentCopilot": "mux (ncvdt_cmd_fcs_dps2 > 50.0) (([0] ++
          count_yaw_output_exceeding_50) + 1) 0"
    },
    {
      "name": "count_airspeed_output_exceeding_32",
```

```
      "type": "int",
      "assignmentLustre": "0 -> if (xcvdt_cmd_fcs_fps2 > 32.0) then pre count_airspeed_output_exceeding_32 + 1 else 0",
      "assignmentCopilot": "mux (xcvdt_cmd_fcs_fps2 > 50.0) (([0] ++ count_airspeed_output_exceeding_32 ) + 1) 0"
    },
    {
      "name": "count_height_output_exceeding_32",
      "type": "int",
      "assignmentLustre": "0 -> if (hcvdt_cmd_fcs_fps2 > 32.0) then pre count_height_output_exceeding_32 + 1 else 0",
      "assignmentCopilot": "mux (hcvdt_cmd_fcs_fps2 > 50.0) (([0] ++ count_height_output_exceeding_32) + 1) 0"
    },
    {
      "name": "roll_command_acceleration",
      "type": "real",
      "assignmentLustre": "0.0 -> (lcvdt_cmd_fcs_dps2 - pre lcvdt_cmd_fcs_dps2) * 100.0",
      "assignmentCopilot": "mux ftp (constant 0) ((lcvdt_cmd_fcs_dps2 - ([0] ++ lcvdt_cmd_fcs_dps2)) * 100.0)"
    },
    {
      "name": "pitch_command_acceleration",
      "type": "real",
      "assignmentLustre": "0.0 -> (mcvdt_cmd_fcs_dps2 - pre mcvdt_cmd_fcs_dps2) * 100.0",
      "assignmentCopilot": "mux ftp (constant 0) ((mcvdt_cmd_fcs_dps2 - ([0] ++ mcvdt_cmd_fcs_dps2)) * 100.0)"
    },
    {
      "name": "yaw_command_acceleration",
      "type": "real",
      "assignmentLustre": "0.0 -> (ncvdt_cmd_fcs_dps2 - pre ncvdt_cmd_fcs_dps2) * 100.0",
      "assignmentCopilot": "mux ftp (constant 0) ((ncvdt_cmd_fcs_dps2 - ([0] ++ ncvdt_cmd_fcs_dps2)) * 100.0)"
    },
    {
      "name": "airspeed_command_acceleration",
      "type": "real",
      "assignmentLustre": "0.0 -> (xcvdt_cmd_fcs_fps2 - pre xcvdt_cmd_fcs_fps2) * 100.0",
      "assignmentCopilot": "mux ftp (constant 0) ((xcvdt_cmd_fcs_fps2 - ([0] ++ xcvdt_cmd_fcs_fps2)) * 100.0)"
    },
    {
      "name": "height_command_acceleration",
      "type": "real",
      "assignmentLustre": "0.0 -> (hcvdt_cmd_fcs_fps2 - pre hcvdt_cmd_fcs_fps2) * 100.0",
      "assignmentCopilot": "mux ftp (constant 0) ((hcvdt_cmd_fcs_fps2 - ([0] ++ hcvdt_cmd_fcs_fps2)) * 100.0)"
    }
],

"Other_variables":
[
    {"name":"lcvdt_cmd_fcs_dps2", "type":"real"},
     {"name":"hcvdt_cmd_fcs_fps2", "type":"real"},
     {"name":"xcvdt_cmd_fcs_fps2", "type":"real"},
     {"name":"ncvdt_cmd_fcs_dps2", "type":"real"},
     {"name":"mcvdt_cmd_fcs_dps2", "type":"real"}
],

"Functions":
[

```

```
],

"Requirements":
[


{ "name": "REG-004", "ptLTL": "(H (count_airspeed_output_exceeding_32 <= 100))", "CoCoSpecCode": "(H((count_airspeed_output_exceeding_32 <= 100)))", "fretish": "Regulator shall always satisfy count_airspeed_output_exceeding_32 <= 100"},



{ "name": "REG-001", "ptLTL": "(H (count_roll_output_exceeding_50 <= 100))", "CoCoSpecCode": "(H((count_roll_output_exceeding_50 <= 100)))", "fretish": "Regulator shall always satisfy count_roll_output_exceeding_50 <= 100"},



{ "name": "REG-006", "ptLTL": "(H (roll_command_acceleration <= 50))", "CoCoSpecCode": "(H((roll_command_acceleration <= 50.0)))", "fretish": "Regulator shall always satisfy roll_command_acceleration <= 50.0"},



{ "name": "REG-005", "ptLTL": "(H (count_height_output_exceeding_32 <= 100))", "CoCoSpecCode": "(H((count_height_output_exceeding_32 <= 100)))", "fretish": "Regulator shall always satisfy count_height_output_exceeding_32 <= 100"},



{ "name": "REG-007", "ptLTL": "(H (pitch_command_acceleration <= 50))", "CoCoSpecCode": "(H((pitch_command_acceleration <= 50.0)))", "fretish": "Regulator shall always satisfy pitch_command_acceleration <= 50.0"},



{ "name": "REG-008", "ptLTL": "(H (yaw_command_acceleration <= 50))", "CoCoSpecCode": "(H((yaw_command_acceleration <= 50.0)))", "fretish": "Regulator shall always satisfy yaw_command_acceleration <= 50.0"},



{ "name": "REG-009", "ptLTL": "(H (airspeed_command_acceleration <= 32))", "CoCoSpecCode": "(H((airspeed_command_acceleration <= 32.0)))", "fretish": "Regulator shall always satisfy airspeed_command_acceleration <= 32.0"},



{ "name": "REG-010", "ptLTL": "(H (height_command_acceleration <= 32))", "CoCoSpecCode": "(H((height_command_acceleration <= 32.0)))", "fretish": "Regulator shall always satisfy height_command_acceleration <= 32.0"},



{ "name": "REG-002", "ptLTL": "(H (count_pitch_output_exceeding_50 <= 100))", "CoCoSpecCode": "(H((count_pitch_output_exceeding_50 <= 100)))", "fretish": "Regulator shall always satisfy count_pitch_output_exceeding_50 <= 100 "},



{ "name": "REG-003", "ptLTL": "(H (count_yaw_output_exceeding_50 <= 100))", "CoCoSpecCode": "(H((count_yaw_output_exceeding_50 <= 100)))", "fretish": "Regulator shall always satisfy count_yaw_output_exceeding_50 <= 100"}


]
}
}
```

REPORT DOCUMENTATION PAGE						*Form Approved* *OMB No. 0704–0188*
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**						
1. REPORT DATE *(DD-MM-YYYY)* 01-01-2022		**2. REPORT TYPE** Technical Memorandum				**3. DATES COVERED** *(From - To)*
4. TITLE AND SUBTITLE Integrating FRET with Copilot: Automated Translation of Natural Language Requirements to Runtime Monitors						**5a. CONTRACT NUMBER**
						5b. GRANT NUMBER
						5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Ivan Perez, Anastasia Mavridou, Tom Pressburger, Alwyn Goodloe, Dimitra Giannakopoulou						**5d. PROJECT NUMBER**
						5e. TASK NUMBER
						5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Ames Research Center, Moffett Field, CA 94035						**8. PERFORMING ORGANIZATION REPORT NUMBER** L–
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001						**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
						11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/TM–2022–20220000049
12. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified-Unlimited Subject Category 59 Availability: NASA STI Program (757) 864-9658						
13. SUPPLEMENTARY NOTES An electronic version can be found at http://ntrs.nasa.gov.						
14. ABSTRACT Runtime verification (RV) enables monitoring systems at runtime, to detect property violations early and limit their potential consequences. To provide the level of assurance required for ultra-critical systems, monitor specifications must faithfully reflect the original mission requirements, which are often written in ambiguous natural language. This report presents an end-to-end framework to capture requirements in structured natural language and generate monitors that capture their semantics faithfully. We leverage NASA's Formal Requirement Elicitation Tool (FRET), and the RV system COPILOT. We extend FRET with mechanisms to capture additional information needed to generate monitors, and introduce OGMA, a new tool to bridge the gap between FRET and COPILOT. With this framework, users can write requirements in an intuitive format and obtain real-time C monitors suitable for use in embedded systems. Our toolchain is available as open source.						
15. SUBJECT TERMS realizability checking, FRET, requirement analysis, compositional						
16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** STI Information Desk (email: help@sti.nasa.gov)	
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU		**19b. TELEPHONE NUMBER** *(Include area code)* (757) 864-9658	

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18