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

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

- **TECHNICAL PUBLICATION.** Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- **TECHNICAL MEMORANDUM.** Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- **CONTRACTOR REPORT.** Scientific and technical findings by NASA-sponsored contractors and grantees.

- **CONFERENCE PUBLICATION.** Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- **SPECIAL PUBLICATION.** Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- **TECHNICAL TRANSLATION.** English-language translations of foreign scientific and technical material pertinent to NASA’s mission.

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

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

- Access the NASA STI program home page at [http://www.sti.nasa.gov](http://www.sti.nasa.gov)

- E-mail your question to help@sti.nasa.gov

- Phone the NASA STI Help Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681–2199
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
The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.
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.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Framework overview</td>
<td>1</td>
</tr>
<tr>
<td>2.1</td>
<td>Step-by-step workflow</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Running example in Natural Language (NL), FRETISH, and pmLTL forms.</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>FRET explanations</td>
<td>5</td>
</tr>
<tr>
<td>3.2</td>
<td>FRET variable editor</td>
<td>6</td>
</tr>
<tr>
<td>5.1</td>
<td>Demonstration of COPilot monitor running as X-Plane plugin: cruising.</td>
<td>9</td>
</tr>
<tr>
<td>5.2</td>
<td>Demonstration of COPilot monitor running as X-Plane plugin: stall.</td>
<td>9</td>
</tr>
<tr>
<td>5.3</td>
<td>Demonstration of COPilot monitor running as X-Plane plugin: recovery.</td>
<td>10</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>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.</td>
<td>12</td>
</tr>
<tr>
<td>6.2</td>
<td>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.</td>
<td>12</td>
</tr>
</tbody>
</table>
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.

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.

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.
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\rightarrow Y (((\text{airspeed} < 100) & ((Y (!((\text{airspeed} < 100))) & Fin\_flight))) & (!((\text{airspeed} >= 100)))) \rightarrow (O_{t=10}((((\text{airspeed} < 100) & ((Y (!((\text{airspeed} < 100))) & Fin\_flight))) & (!((\text{airspeed} >= 100)))) \rightarrow (O_{t<10}(\text{Fin}\_flight) & (\text{airspeed} >= 100)))) & Fin\_flight)))) & (!Lin\_flight) S (!Lin\_flight) & 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 ≥ 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 ≤ 100 becoming true). timing specifies when the response must occur (e.g., within 10 seconds).

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].

We extended FRET’s analysis portal 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 $\text{altitude} > 0.0$, where altitude is an input variable. Internal variable assignments can be defined in Lustre or Copilot. 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
prop AvoidStall = ((PTLTL.alwaysBeen (((not (flight)) && ... ))))

spec :: Spec
spec = do
  trigger "handlerpropAvoidStall" (not propAvoidStall) []
```
Chapter 5

Copilot Steps

COPEILOT 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:

```c
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.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>K Result</th>
<th>K Time</th>
<th>Monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG-001</td>
<td>Undecided</td>
<td>Timeout</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>REG-002</td>
<td>Undecided</td>
<td>Timeout</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>REG-003</td>
<td>Valid</td>
<td>10.046 sec</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>REG-004</td>
<td>Undecided</td>
<td>Timeout</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>REG-005</td>
<td>Undecided</td>
<td>Timeout</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>REG-006</td>
<td>Invalid</td>
<td>5.998 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>REG-007</td>
<td>Invalid</td>
<td>5.998 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>REG-008</td>
<td>Invalid</td>
<td>5.998 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>REG-009</td>
<td>Invalid</td>
<td>5.998 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>REG-010</td>
<td>Invalid</td>
<td>5.998 sec</td>
<td>Activated</td>
</tr>
<tr>
<td><strong>Total running time</strong></td>
<td></td>
<td></td>
<td>CoCoSim: Timeout</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>K Result</th>
<th>K Time</th>
<th>Monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM-001v1</td>
<td>Invalid</td>
<td>0.254 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>FSM-001v2</td>
<td>Invalid</td>
<td>0.465 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>FSM-001v3</td>
<td>true up to 28 steps</td>
<td>timeout (2h)</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>FSM-002</td>
<td>Valid</td>
<td>0.252 sec</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>FSM-003</td>
<td>Invalid</td>
<td>0.191 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>FSM-004</td>
<td>Invalid</td>
<td>0.191 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>FSM-005</td>
<td>Valid</td>
<td>0.252 sec</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>FSM-006</td>
<td>Valid</td>
<td>0.252 sec</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>FSM-007</td>
<td>Invalid</td>
<td>0.135 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>FSM-007v2</td>
<td>Valid</td>
<td>0.252 sec</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>FSM-008v1</td>
<td>Invalid</td>
<td>0.165 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>FSM-009</td>
<td>Valid</td>
<td>0.252 sec</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>FSM-010</td>
<td>Valid</td>
<td>0.132 sec</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>FSM-011v1</td>
<td>Invalid</td>
<td>0.110 sec</td>
<td>Activated</td>
</tr>
<tr>
<td>FSM-011v2</td>
<td>Valid</td>
<td>0.132 sec</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>FSM-012</td>
<td>Valid</td>
<td>0.132 sec</td>
<td>Non-Activated</td>
</tr>
<tr>
<td>FSM-013</td>
<td>Valid</td>
<td>0.132 sec</td>
<td>Non-Activated</td>
</tr>
<tr>
<td><strong>Total running time</strong></td>
<td></td>
<td></td>
<td>CoCoSim: 141.09sec</td>
</tr>
</tbody>
</table>
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


Appendix

7.1 FSM Specification

```json
{  
"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"  
}  
]  
}
```
7.2 FSM Manager Specification

```json
{
  "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": ""
      }
    ]
  }
}
```
"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 ) \rightarrow STATE = ap_standby_state))",
"CoCoSpecCode": "(H((( standby and state = ap_transition_state ) \Rightarrow STATE = ap_standby_state)))",
"fretish": "FSM_Autopilot shall always satisfy (standby & state = ap_transition_state) \Rightarrow STATE = ap_standby_state"
},
{
"name": "FSM-005",
"ptLTL": "(H (( state = ap_nominal_state & standby ) \rightarrow STATE = ap_standby_state))",
"CoCoSpecCode": "(H((( state = ap_nominal_state and standby ) \Rightarrow STATE = ap_standby_state)))",
"fretish": "FSM_Autopilot shall always satisfy (state=ap_nominal_state & standby) \Rightarrow STATE = ap_standby_state"
},
{
"name": "FSM-003",
"ptLTL": "(H (( state = ap_transition_state & good & supported ) \rightarrow STATE = ap_nominal_state))",
"CoCoSpecCode": "(H((( state = ap_transition_state and good and supported ) \Rightarrow STATE = ap_nominal_state)))",
"fretish": "FSM_Autopilot shall always satisfy (state = ap_transition_state & good & supported) \Rightarrow STATE = ap_nominal_state"
},
{
"name": "FSM-008v1",
"ptLTL": "(H (( state = ap_standby_state & ! standby ) \rightarrow STATE = ap_transition_state))",
"CoCoSpecCode": "(H((( state = ap_standby_state and not standby ) \Rightarrow STATE = ap_transition_state)))",
"fretish": " FSM_Autopilot shall always satisfy (state = ap_standby_state & ! standby) \Rightarrow 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_standby_state))",
  "CoCoSpecCode": "(H((( state = ap_maneuver_state and supported and good ) => STATE = ap_standby_state)))",
  "fretish": "FSM_Autopilot shall always satisfy (state = ap_maneuver_state & supported & good) => STATE = ap_standby_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 specification

```json
{
  "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": "(( (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": "(( (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"
      }
    ]
  }
}
```
7.4 Regulators Specification

```json
{
  "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"
}
}

"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_dps 2 > 50.0) then pre
      count_pitch_output_exceeding_50 + 1 else 0",
      "assignmentCopilot": "mux ( mcvdt_cmd_fcs_dps 2 > 50.0) (([0] ++
      count_pitch_output_exceeding_50 + 1) 0"
    },
    {
      "name": "count_yaw_output_exceeding_50",
      "type": "int",
      "assignmentLustre": "0 -> if ( ncvdt_cmd_fcs_dps 2 > 50.0) then pre
      count_yaw_output_exceeding_50 + 1 else 0",
      "assignmentCopilot": "mux ( ncvdt_cmd_fcs_dps 2 > 50.0) (([0] ++
      count_yaw_output_exceeding_50 + 1) 0"
    },
    {
      "name": "count_airspeed_output_exceeding_32",
      "type": "int",
      "assignmentLustre": "0 -> if ( mcvdt_cmd_fcs_dps 2 > 50.0) then pre
      count_pitch_output_exceeding_50 + 1 else 0",
      "assignmentCopilot": "mux ( mcvdt_cmd_fcs_dps 2 > 50.0) (([0] ++
      count_airspeed_output_exceeding_32 + 1) 0"
    }
  ]
}
```
"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"}
}
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.