



**Maynooth
University**

National University
of Ireland Maynooth

Refactoring in Requirements Engineering: Exploring a methodology for formal verification of safety-critical systems

Oisín Sheridan

Department of Computer Science, Maynooth University, Ireland

7th of April 2025

Overview

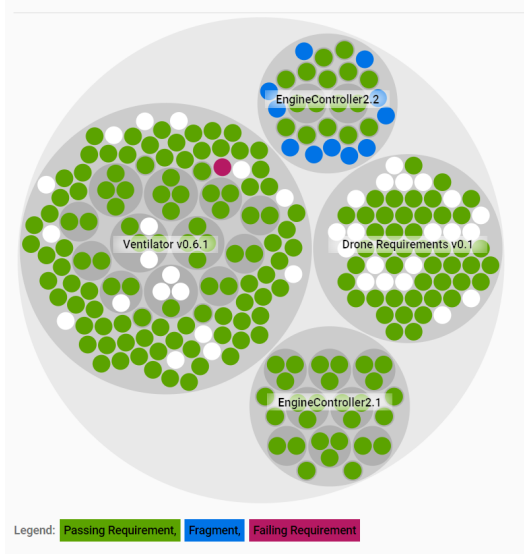
- ▶ Requirements Elicitation with FRET
 - ▶ Started on the VALU3S project
 - ▶ Three phase methodology - Two parallel techniques
 - ▶ Toolchain - (Mu-)FRET, Simulink, Event-B
- ▶ Refactoring of requirements
 - ▶ Four refactorings planned for Mu-FRET
 - ▶ Three currently implemented - Extract, Inline and Rename
 - ▶ Develop theory to expand applicability
- ▶ Case Studies
 - ▶ Aircraft Engine Controller, from VALU3S
 - ▶ Mechanical Lung Ventilator, from ABZ conference
 - ▶ UAV Tilt-Rotor Drone, from the ProVANT Emergentia Project

FRET - the Formal Requirements Elicitation Tool

The Formal Requirements Elicitation Tool (FRET)

FRET

- ▶ FRET is an open source tool for requirements engineering developed by NASA
- ▶ Requirements are written in a structured natural-language called FRETish
- ▶ FRET provides automated translations from FRETISH requirements to CoCoSpec contracts, which can be verified with the Kind2 model checker, and Copilot runtime monitors



The Formal Requirements Elicitation Tool (FRET)

Update Requirement

Requirement ID	Parent Requirement ID	Project
UC5_R_1_1	UC5_R_1	EngineController2.1

Rationale and Comments

Requirement Description

A requirement follows the sentence structure displayed below, where fields are optional unless indicated with "*". For information on a field format, click on its corresponding bubble.

SCOPE

CONDITIONS

COMPONENT*

SHALL*

TIMING

RESPONSES*



when (diff_ref_obs > activeThreshold) if ((sensorValue_S > nominalValue + R) | (sensorValue_S < nominalValue - R) | (sensorValue_S = null) & (pilotInput => setThrust = V2) & (observedThrust = V1)) Controller shall until (diff_ref_obs < inactiveThreshold) satisfy (settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)

SEMANTICS

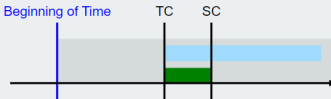
ASSISTANT

TEMPLATES

GLOSSARY

ENFORCED: in the interval defined by the entire execution. TRIGGER: first point in the interval if $((diff_ref_obs > activeThreshold)) \& (((sensorValue_S > nominalValue + R) | (sensorValue_S < nominalValue - R) | (sensorValue_S = null) \& (pilotInput \Rightarrow setThrust = V2) \& (observedThrust = V1)))$ is true and any point in the interval where $((diff_ref_obs > activeThreshold)) \& (((sensorValue_S > nominalValue + R) | (sensorValue_S < nominalValue - R) | (sensorValue_S = null) \& (pilotInput \Rightarrow setThrust = V2) \& (observedThrust = V1)))$ becomes true (from false). REQUIRES: for every trigger, RES must remain true until (but not necessarily including) the point where the stop condition holds, or to the end of the interval. If the stop condition never occurs, RES must hold until the end of the scope, or forever. If the stop condition holds at the trigger, the requirement is satisfied.

Beginning of Time



TC = $((diff_ref_obs > activeThreshold)) \& (((sensorValue_S > nominalValue + R) | (sensorValue_S < nominalValue - R) | (sensorValue_S = null) \& (pilotInput \Rightarrow setThrust = V2) \& (observedThrust = V1)))$, SC = $((diff_ref_obs < inactiveThreshold))$, Response = $((settlingTime >= 0) \& (settlingTime <= settlingTimeMax) \& (observedThrust = V2))$.

Refactoring Requirements in Mu-FRET

Motivation

- ▶ While using FRET to formalise requirements during the VALU3S project, we found that there were definitions repeated many times across the requirements set
 - ▶ e.g. Sensor Faults: "if((sensorValue(S) > nominalValue + R) | (sensorValue(S) < nominalValue - R) | (sensorValue(S) = null))".
This definition was included in 8 different requirements.
- ▶ This creates unfortunate dependencies between requirements, and requires adjustments in multiple places if a definition needs to be changed
- ▶ We believed that this could be solved using refactoring, a technique used to reorganise and improve the structure of software without altering its behaviour
- ▶ We decided to implement refactoring functionality in a fork of FRET, called Mu-FRET. We started with the most applicable refactoring, *Extract Requirement*.

Refactoring Requirements

- ▶ Our refactoring of requirements is primarily based on *Improving the Quality of Requirements with Refactoring* by Ramos et al. (2007)
- ▶ They define five refactorings to be used on requirements; we believe that four are applicable to FRETish.
- ▶ These four are: Extract Requirement, Inline Requirement, Rename Requirement and Move Activity/Definition

3.1. Extract Requirement

Context. A set of inter-related information is used in several places or could be better modularized in a separate requirement. Or a requirement is too large or contains information related to a feature that is scattered across several requirements or is tangled with other concerns.

Solution. Extract the information to a new requirement and name it according to the context.

Motivation. This refactoring should be applied when there are large requirements that can be split into two or more new requirements. These large requirements include a great deal of information that is difficult to understand. Furthermore it is not easy to locate the needed information quickly [Alexander 2002] [Sommerville 1997].

Mechanics. The following activities should be performed:

1. Create a new requirement and name it.
2. Select the information you want to extract.
3. Add the selected information to the new requirement.
4. Remove the information from the original requirement.
5. Make sure the original requirement is acceptable without the removed information.
6. Update the references in dependent requirements.

Rename Requirement

- ▶ Changes a requirement's name, and updates any references to it contained in other requirements
- ▶ FRET already allows the user to rename a requirement, but any references to it in the 'Parent Requirement ID' field of child requirements are not updated, leaving the child with a broken reference
- ▶ This needs to be addressed, since we often create variables that represent specific requirements when refactoring

3.2 Rename Requirement

Context. The name of a requirement is not appropriate for the context, is abbreviated or is used in several places with different semantics.

Solution. Rename the requirement to clearly express its purpose.

Motivation. Good names make communication and understanding system abstractions easier and provide a common vocabulary to the development team [Alexander and Stevens 2002].

Mechanics. The following activities should be performed:

1. Select the requirement you want to rename.
2. Change the name of the requirement.
3. Update the references in dependent requirements.

Move Definition

- ▶ This refactoring takes part of a requirement and moves it to another, focusing a requirement on a single responsibility
- ▶ We are also considering if Move Definition could be specialised into a 'Pull-Up Definition' refactoring that explicitly moves common definitions from a child requirement up to its parent

3.3 Move Activity

Context. A set of activities is better accommodated in another requirement description.

Solution. Move the activity to the desired requirement. If the requirement does not exist yet, create a new requirement with the selected activities using the Extract Requirement refactoring.

Motivation. This refactoring is done to improve modularity and to ameliorate the balance of activities among the requirements. Activities are moved from one requirement to another also when a new requirement is created, either manually or by an Extract Requirement refactoring. This improvement in modularity could lead to a better understanding of the system in the long term [Sommerville 2004].

Mechanics. The following activities should be performed:

1. Select the activities you want to move.
2. Move them to the desired requirement.
3. Update references to these activities if needed.

Inline Requirement

- ▶ Inline Requirement is the opposite of Extract Requirement; it takes one requirement and merges it into another
- ▶ It is useful where a requirement is only referenced in a few places, and is simple enough that its definitions are as clear as its name
- ▶ In such a case, Inline Requirement can simplify the requirement set and improve readability

3.4 Inline Requirement

Context. A requirement is referenced and used in only one place or in a few places in such a way that its existence is not justified in terms of maintenance costs.

Solution. Insert the requirements description into the requirements that use it.

Motivation. This refactoring reduces the complexity of the requirements model by merging requirements. Each software artifact demands time and resources to understand and maintain it [Jacobson 2003]. If a requirement is not useful enough to justify its existence, the developer can inline it, merging its responsibilities with other requirements.

Mechanics. The following activities should be performed:

1. Copy the activities (including pre and post conditions if applicable) described in the requirement to all requirements that uses this one.
2. Update the affected requirements to reflect the inlined activities and other requirements information.
3. Remove references to the inlined requirement.
4. Remove the inlined requirement.

Refactoring Examples

Refactoring - Extract Requirement

Req ID	FRETish Requirement
UC5_R_1	<code>if ((sensorfaults) & (trackingPilotCommands)) ControlledSystem shall satisfy (controlObjectives)</code>
UC5_R_1_1	<code>when (diff_ref_obs > activeThreshold) if ((sensorValue_S > nominalValue + R) (sensorValue_S < nominalValue - R) (sensorValue_S = null) & (pilotInput => setThrust = V2) & (observedThrust = V1)) Engine shall until (diff_ref_obs < inactiveThreshold) satisfy (settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)</code>



UC5_R_1_1



when (diff_ref_obs > activeThreshold) if ((sensorValue_S > nominalValue + R) | (sensorValue_S < nominalValue - R) | (sensorValue_S = null) & (pilotInput => setThrust = V2) & (observedThrust = V1)) Controller shall until (diff_ref_obs < inactiveThreshold) satisfy (settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)

Refactoring - Extract Requirement

Extract Requirement: UC5_R_1_1

Copy the part of UC5_R_1_1 that you want to extract from its Definition into the Extract field, and add the New Requirement Name. The Apply to all Requirements tick box toggles if the extraction will search for the Extract field in all requirements in this project.

Definition: Definition

when (diff_ref_obs > activeThreshold) if ((sensorValue_S > nominalValue + R) |
(sensorValue_S < nominalValue - R) | (sensorValue_S = null) & (pilotInput => setThrust = V2) &
(observedThrust = V1)) Engine shall until (diff_ref_obs < inactiveThreshold) satisfy
(settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)

Extract:

Extract

New Requirement Name:

New Name

Apply to all Requirements in
EngineController2.1:

☐

CANCEL

OK

Refactoring - Extract Requirement

Extract Requirement: UC5_R_1_1

Copy the part of UC5_R_1_1 that you want to extract from its Definition into the Extract field, and add the New Requirement Name. The Apply to all Requirements tick box toggles if the extraction will search for the Extract field in all requirements in this project.

Definition: Definition

when (diff_ref_obs > activeThreshold) if ((sensorValue_S > nominalValue + R) | (sensorValue_S < nominalValue - R) | (sensorValue_S = null) & (pilotInput => setThrust = V2) & (observedThrust = V1)) Engine shall until (diff_ref_obs < inactiveThreshold) satisfy (settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)

Extract: Extract

(sensorValue_S > nominalValue + R) | (sensorValue_S < nominalValue - R) | (sensorValue_S = null) & (pilotInput => setThrust = V2)

New Requirement Name: New Name

SENSOR_FAULTS

Apply to all Requirements in
EngineController2.1:



CANCEL

OK



Refactoring - Extract Requirement

Check Types Before Extracting:

Requirements to be refactored: UC5_R_1_1, UC5_R_1_2, UC5_R_1_3

UC5_R_1_1 Definition:

when (diff_ref_obs > activeThreshold) if ((sensorValue_S > nominalValue + R) |
(sensorValue_S < nominalValue - R) | (sensorValue_S = null) & (pilotInput => setThrust = V2)
& (observedThrust = V1)) Engine shall until (diff_ref_obs < inactiveThreshold) satisfy
(settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)

- diff_ref_obs :undefined  ▼
- activeThreshold :undefined  ▼
- sensorValue_S :integer ▼
- nominalValue :integer ▼
- R :integer ▼
- null :integer ▼
- pilotInput :boolean ▼

OK

Successfully Extracted Requirement: UC5_R_1_1
















The checks have passed and the refactoring is complete. You may Close this dialogue.



Checks Passed. The original and new requirements behave the same.

CLOSE

Refactoring - Extract Requirement

Status	ID ↑			Summary
	SENSORFAULTS			if (sensorValue_S > nominalValue + R) (sensorValue_S < nominalValue - R) (sensorValue_S = null) Controller shall satisfy SENSORFAULTS
	UC5_R_1			if ((sensorfaults) & (trackingPilotCommands)) Controller shall satisfy (controlObjectives)
	UC5_R_1_1			when (diff_ref_obs > activeThreshold) if (SENSORFAULTS & (pilotInput => setThrust = V2) & (observedThrust = V1)) Controller shall until (diff_ref_obs < inactiveThreshold) satisfy (settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)
	UC5_R_1_2			when (diff_ref_obs > activeThreshold) if (SENSORFAULTS & (pilotInput => setThrust = V2) & (observedThrust = V1)) Controller shall until (diff_ref_obs < inactiveThreshold) satisfy (overshoot >= 0) & (overshoot <= overshootMax) & (observedThrust = V2)
	UC5_R_1_3			when (diff_ref_obs > activeThreshold) if (SENSORFAULTS & (pilotInput => setThrust = V2) & (observedThrust = V1)) Controller shall until (diff_ref_obs < inactiveThreshold) satisfy (steadyStateError >= 0) & (steadyStateError <= steadyStateErrorMax) & (observedThrust = V2)

Inline Requirement: SENSORFAULTS

Response: SENSORFAULTS

This is a fragment

Definition: Definition

if (sensorValue_S > nominalValue + R) | (sensorValue_S < nominalValue - R) | (sensorValue_S = null) Controller shall satisfy
SENSORFAULTS

This is a fragment. Would you like to inline it in place of the variable SENSORFAULTS? YES

CANCEL

OK

Refactoring - Inline Requirement

Inline Requirement: SENSORFAULTS

Response: SENSORFAULTS

This is a fragment

Definition:

Definition

if (sensorValue_S > nominalValue + R) | (sensorValue_S < nominalValue - R) |
(sensorValue_S = null) Controller shall satisfy SENSORFAULTS

This is a fragment. Would you like to inline it in place of the variable SENSORFAULTS? **YES**

UC5_R_1_2

Definition

when (diff_ref_obs >
activeThreshold) if
(SENSORFAULTS & (pilotInput
=> setThrust = V2) &
(observedThrust = V1))
Controller shall until
(diff_ref_obs <
inactiveThreshold) satisfy
(overshoot >= 0) &
(overshoot <=
overshootMax) &
(observedThrust = V2)

INLINE HERE

UC5_R_1_3

Definition

when (diff_ref_obs >
activeThreshold) if
(SENSORFAULTS & (pilotInput
=> setThrust = V2) &
(observedThrust = V1))
Controller shall until
(diff_ref_obs <
inactiveThreshold) satisfy
(steadyStateError >= 0) &
(steadyStateError <=
steadyStateErrorMax) &
(observedThrust = V2)

INLINE HERE

UC5_R_2_1

Definition

when (diff_ref_obs >
activeThreshold) if
(SENSORFAULTS &
(!pilotInput => setThrust =
V1) & (observedThrust =
V2)) Controller shall until
(diff_ref_obs <
inactiveThreshold) satisfy
(settlingTime >= 0) &
(settlingTime <=
settlingTimeMax) &
(observedThrust = V1)

INLINE HERE

UC5_R_2_2

Definition

when (diff_ref_obs >
activeThreshold) if
(SENSORFAULTS &
(!pilotInput => setThrust =
V1) & (observedThrust =
V2)) Controller shall until
(diff_ref_obs <
inactiveThreshold) satisfy
(overshoot >= 0) &
(overshoot <=
overshootMax) &
(observedThrust = V1)

INLINE HERE

UC5_R_2_3

Definition

when (diff_ref_obs >
activeThreshold) if
(SENSORFAULTS &

UC5_R_3_1

Definition

when (diff_ref_obs >
activeThreshold) if
(SENSORFAULTS &

UC5_R_4_1

Definition

when (diff_ref_obs >
activeThreshold) if
(SENSORFAULTS &

SENSORFAULTS

Definition

if (sensorValue_S >
nominalValue + R) |
(sensorValue_S <

CANCEL

OK

Refactoring - Inline Requirement

Inline Requirement: SENSORFAULTS

Response: SENSORFAULTS
This is a fragment

SENSORFAULTS: Definition

```
if (sensorValue_S >
nominalValue + R) |
(sensorValue_S < nominalValue
- R) | (sensorValue_S = null)
Controller shall satisfy
SENSORFAULTS
```

Inlined version of UC5_R_1_2:
Definition

```
when (diff_ref_obs > activeThreshold) if (((
sensorValue_S > nominalValue + R) | (
sensorValue_S < nominalValue - R) | (
sensorValue_S = null )) & (pilotInput =>
setThrust = V2) & (observedThrust = V1))
Controller shall until (diff_ref_obs <
inactiveThreshold) satisfy (overshoot >= 0)
& (overshoot <= overshootMax) &
(observedThrust = V2)
```

- SENSORFAULTS :boolean ▾
- V1 :integer ▾
- V2 :integer ▾
- activeThreshold :integer ▾
- diff_ref_obs :integer ▾

CANCEL OK

Refactoring - Inline Requirement

Requirement ID	Parent Requirement ID	Project
UC5_R_1_2	UC5_R_1	EngineController2.1

Rationale and Comments

Requirement Description

A requirement follows the sentence structure displayed below, where fields are optional unless indicated with "**". For information on a field format, click on its corresponding bubble.



when (diff_ref_obs > activeThreshold) if (((sensorValue_S > nominalValue + R) | (sensorValue_S < nominalValue - R) | (sensorValue_S = null))& (pilotInput => setThrust = V2) & (observedThrust = V1)) Controller shall until (diff_ref_obs < inactiveThreshold) satisfy (overshoot >= 0) & (overshoot <= overshootMax) & (observedThrust = V2)

SEMANTICS

Methodology & Case Studies

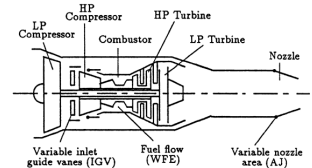
Research Questions

- ▶ **RQ1:** To what extent are software refactorings applicable to functional requirements?
 - ▶ **RQ2:** Under precisely what conditions are requirement refactoring techniques applicable? What elements of the structure of a requirement might allow or prevent the use of one or more refactoring techniques?
 - ▶ **RQ3:** How can refactoring techniques developed for FRETISH requirements be generalised for other requirements languages and formalisms?
-
- ▶ The first step in answering these questions is to complete the implementation of the four refactorings in Mu-FRET.
 - ▶ Then we will evaluate the applicability of each refactoring on a number of case studies. We have developed three such sets of requirements to date.
 - ▶ We will also examine the effect that refactoring has on the underlying meaning of the requirements, represented by the LTL formulae generated by FRET.

Case Study 1 - Aircraft Engine Controller

Aircraft Engine Controller - Overview

- ▶ VALU3S Use Case 5 focused on the software controller for a high-bypass civilian aircraft turbofan engine
- ▶ An example of a Full Authority Digital Engine Control (FADEC) system, which monitors and controls everything about the engine, including thrust control, fuel control, power management, health monitoring of the engine, thrust reverser control, etc.
- ▶ Engine is expected to maintain proper operation in the presence of sensor faults, perturbation of system parameters and environmental hazards (e.g. sudden drop in air pressure)



Case Study 1 - Aircraft Engine Controller

Table 2.3 Aerospace (UC5) SCP requirements

Requirement ID	Corresponding evaluation scenario ID(s)	Textual Description	Supporting information
UC5_R_1	VALU3S_WP1_Aerospace_1	Under sensor faults, while tracking pilot commands, control objectives shall be satisfied (e.g. settling time, overshoot, and steady state error will be within predefined, acceptable limits)	Safety

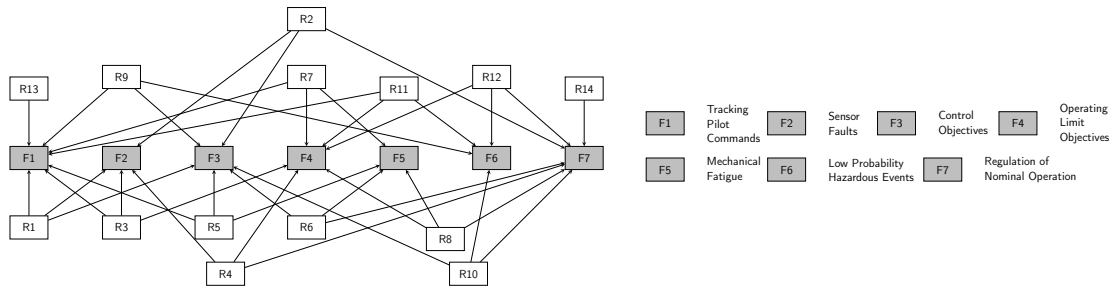
Table 2.4 Aerospace (UC5) test cases

Test case ID	Corresponding requirement ID(s)	Textual Description	Supporting information
UC5_TC_1	UC5_R_1	<ul style="list-style-type: none">* Preconditions: Aircraft is in operating mode M and sensor S value deviates at most +/- R % from nominal value* Input conditions / steps: Observed aircraft thrust is at value V1 and pilot input changes from A1 to A2* Expected results: Observed aircraft thrust changes and settles to value V2, respecting control objectives (settling time, overshoot, steady state error)	Safety
UC5_TC_2	UC5_R_1	<ul style="list-style-type: none">* Preconditions: Aircraft is in operating mode M and sensor S value is not available (sensor is out of order)* Input conditions / steps: Observed aircraft thrust is at value V1 and pilot input changes from A1 to A2* Expected results: Observed aircraft thrust changes and settles to value V2, respecting control objectives (settling time, overshoot, steady state error)	Safety

Case Study 1 - Aircraft Engine Controller

Req ID	FRETish Requirement
UC5_R_1	if ((sensorfaults) & (trackingPilotCommands)) ControlledSystem shall satisfy (controlObjectives)
UC5_R_1_1	when (diff_ref_obs > activeThreshold) if ((sensorValue_S > nominalValue + R) (sensorValue_S < nominalValue - R) (sensorValue_S = null) & (pilotInput => setThrust = V2) & (observedThrust = V1)) Engine shall until (diff_ref_obs < inactiveThreshold) satisfy (settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)
UC5_R_14	if (!trackingPilotCommands) Controller shall satisfy satisfy newMode=nominal newMode=surgeStallPrevention
UC5_R_14_2	in surgeStallPrevention mode when (diff_setNL_observedNL < NLmax) if (!pilotInput => !surgeStallAvoidance) Controller shall until (diff_setNL_observedNL > NLmin) satisfy newMode=nominal

Case Study 1 - Aircraft Engine Controller



Requirements and Fragments

- ▶ From the original 14 natural-language requirements and 20 associated test cases, we formalised 42 requirements using FRET
- ▶ We identified seven definitions that were repeated across a number of requirements (referred to as 'fragments').

Case Study 1 - Aircraft Engine Controller

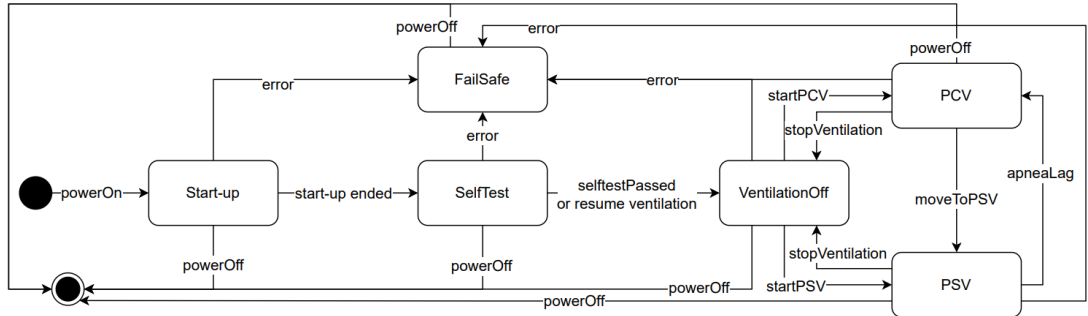
ID	Fragment Name	№ of (Re)Definitions	
		Before Refactoring	After Refactoring
F1	<i>Sensor Faults</i>	8	1
F2	<i>Tracking Pilot Commands</i>	13	1
F3	<i>Control Objectives</i>	18	1
F4	<i>Regulation Of Nominal Operation</i>	14	1
F5	<i>Operating Limit Objectives</i>	6	1
F6	<i>Mechanical Fatigue</i>	8	1
F7	<i>Low Probability Hazardous Events</i>	8	1
F8	<i>Active</i>	28	1
F9	<i>Not Active</i>	28	1
Total (Re)Definitions		132	9

Table 1: The number of times each fragment's definition occurs in a child requirement. Note that this table includes *Active* and *Not Active*.

Case Study 2: Mechanical Lung Ventilator

Mechanical Lung Ventilator

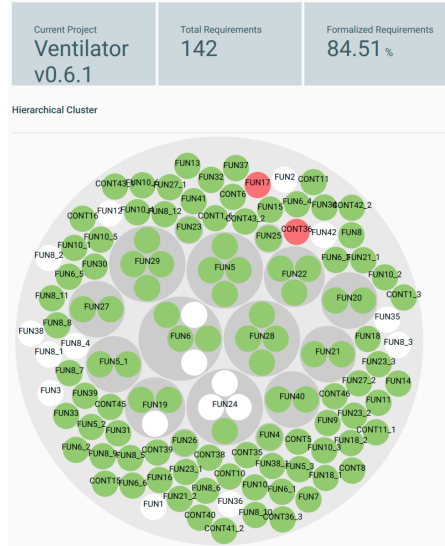
- ▶ Case study proposed for the ABZ 2024 conference
- ▶ Crucial during the Covid-19 pandemic for supporting patients who were unable to breathe on their own



Case Study 2: Mechanical Lung Ventilator

Mechanical Lung Ventilator - Requirements

- ▶ The case study documentation provided a large number of requirements, partitioned into 58 Functional Requirements, 25 Values and Ranges, 58 Sensors and Interfaces, 107 GUI Requirements, 45 Controller Requirements, and 26 Alarms.
- ▶ We decided to focus on the Functional (FUN) and Controller (CONT) requirements
- ▶ We also constructed an Event-B model of the system using both the natural-language and FRETish requirements as a base.



Case Study 3: ProVANT Emergentia Drone

UAV Drone Requirements

- ▶ Tilt-rotor drone capable of Vertical Take-Off and Landing (VTOL) and cruise flight
- ▶ Began with a set of 66 high-level requirements focused on data monitoring, simulation, timing constraints on the control loop and operation under failure conditions.
- ▶ The requirements were formalised over four versions of the FRETISH requirements set.

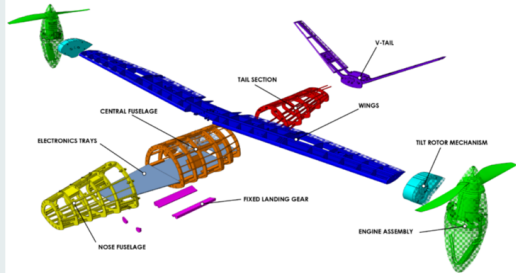


Figure 9.18: Disassembled ProVANT Emergentia Drone

Case Study 3: ProVANT Emergentia Drone

ID	Final fretish text
REQ001	Allow failure simulations between nucleo/jetson and nucleo/nucleo
	in <code>SimulationMode</code> mode whenever <code>SimulateFailureTransitions</code> <code>System</code> shall <code>eventually</code> satisfy <code>JetsonFailureTransitionToNucleo</code> <code>NucleoFailureSwitchActiveNucleo</code>
REQ018	The control loop will complete within 12 milliseconds
	upon <code>ControlLoopStart</code> <code>System</code> shall <code>within 12 milliseconds</code> satisfy <code>ControlLoopFinish</code>
REQ019	The control algorithm will complete within 6 milliseconds
	upon <code>ControlAlgorithmStart</code> <code>System</code> shall <code>within 6 milliseconds</code> satisfy <code>ControlAlgorithmFinish</code>
REQ033	Monitor linear velocities (ground speed and relative wind speed)
	upon <code>ControlLoopStart</code> <code>ActiveNucleo</code> shall <code>before ControlLoopFinish</code> satisfy <code>(MonitorGroundSpeed & SendGroundSpeedData) & (MonitorWindSpeed & SendWindSpeedData)</code>

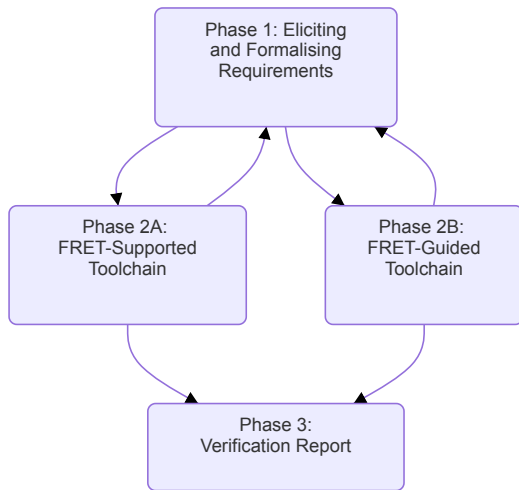
- ▶ There are two immediate tasks underway, which will complete the implementation of Mu-FRET refactoring:
 - ▶ Implement Move Definition
 - ▶ Incorporate NuSMV checks into Inline, Rename, and Move
- ▶ We have described the steps involved with Move Definition in previous work, though some questions remain about the specifics of the implementation (e.g. what parts of a requirements can be moved, which other requirements are valid options for movement)
- ▶ Since we already use Nu-SMV during Extract Requirement, incorporating it into the other refactorings is mainly a matter of deciding exactly what properties should be checked, and ensuring that the implementation is robust when multiple different refactorings are performed over time.

Appendix - Bonus

The Three-Phase Methodology

Methodology

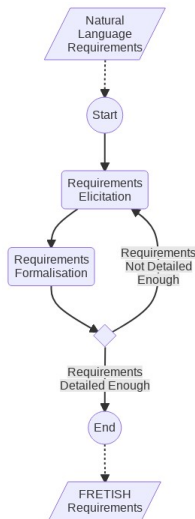
- ▶ We developed a three-phase methodology for verification based around FRET
- ▶ Phase 1: Requirements...
 - ▶ Initial requirements
 - ▶ Eliciting detail
 - ▶ Structure and formalise
- ▶ Phase 2: Verification...
 - ▶ Automatic output from FRET (2A)
 - ▶ Guided by requirements in FRET (2B)
- ▶ Phase 3: Reporting...
 - ▶ Verification results and evidence
 - ▶ Traceability evidence



The Three-Phase Methodology

Phase 1: Eliciting and Formalising Requirements

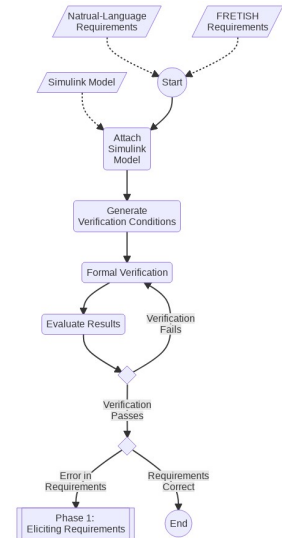
- ▶ Begin with requirements in natural-language
 - ▶ e.g. R1: "Under sensor faults, while tracking pilot commands, control objectives shall be satisfied (e.g. settling time, overshoot, and steady state error will be within predefined, acceptable limits)"
- ▶ Formalise and structure requirements in FRET
- ▶ Iterate in collaboration with partners to address ambiguities and problems with the formalised requirements
- ▶ FRETish acts as an intermediary language for translation to other formalisms - LTL, CoCoSpec contracts



The Three-Phase Methodology

Phase 2A: FRET-Supported Verification

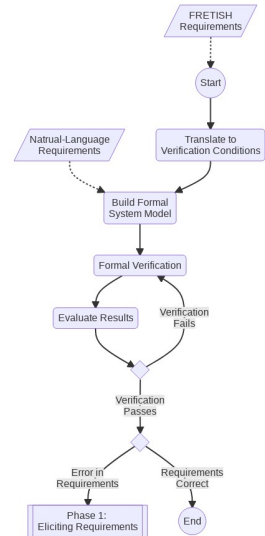
- ▶ This phase makes use of FRET's built-in translation functionality to produce CoCoSpec contracts
- ▶ Contracts can be attached to the Simulink model of the system, mapping components mentioned in requirements to blocks and signals
- ▶ Model can then be verified to check if the contracts hold
- ▶ If not, calls for a re-evaluation of the model and/or of the requirements



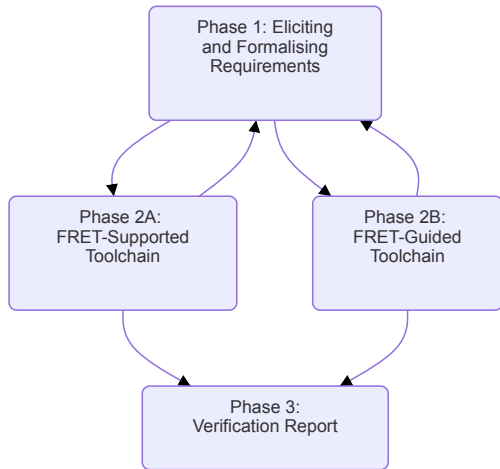
The Three-Phase Methodology

Phase 2B: FRET-Guided Verification

- ▶ This phase uses the requirements and model to drive formal modelling in another formalism, e.g. Event-B
- ▶ A model of the system is constructed based on the Simulink model, requirements and other sources
- ▶ FRETISH requirements are used as a basis for the properties to be verified in the chosen formalism
- ▶ Formal verification is performed using the appropriate tool and method, e.g. model checking, theorem proving, etc.
- ▶ This external model corroborates verification results and may be suited to verifying different properties



The Three-Phase Methodology



Phase 3: Verification Report

- ▶ Finally, a report on the verification results is assembled, describing the verification process and which properties of the system could and could not be verified
- ▶ The report compiles the results of Phases 2A and 2B. As the system may be verified differently in each phase, the report offers a complete view of the results
- ▶ The report may take different forms depending on the intended purpose and audience, e.g. stakeholders, regulators, academics, consumers

Extract Requirement

- ▶ Moves definitions from one requirement into a newly created requirement. In the original, the extracted parts are replaced with a reference to the new requirement.
- ▶ We can extract the same definition from multiple requirements into a single fragment
- ▶ Also serves the intent of the Extract Alternative Flows refactoring, which is aimed at modularising Use Case descriptions

3.1. Extract Requirement

Context. A set of inter-related information is used in several places or could be better modularized in a separate requirement. Or a requirement is too large or contains information related to a feature that is scattered across several requirements or is tangled with other concerns.

Solution. Extract the information to a new requirement and name it according to the context.

Motivation. This refactoring should be applied when there are large requirements that can be split into two or more new requirements. These large requirements include a great deal of information that is difficult to understand. Furthermore it is not easy to locate the needed information quickly [Alexander 2002] [Sommerville 1997].

Mechanics. The following activities should be performed:

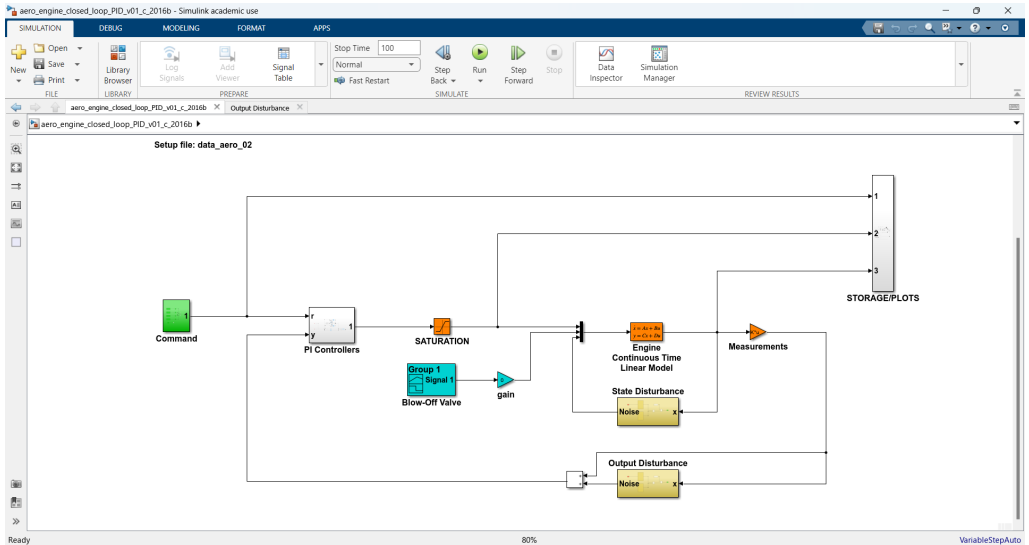
1. Create a new requirement and name it.
2. Select the information you want to extract.
3. Add the selected information to the new requirement.
4. Remove the information from the original requirement.
5. Make sure the original requirement is acceptable without the removed information.
6. Update the references in dependent requirements.

VALU3S

- ▶ 'Verification and Validation of Automated Systems' Safety and Security'
- ▶ 24 industrial partners, 6 research institutes, 10 universities
- ▶ Use Case Domains: automotive (3) , agriculture (1), railway (2), healthcare (1), aerospace (1) and industrial automation and robotics (4)
- ▶ Maynooth team worked on the aerospace use case (UC5) - controller for a civilian aircraft engine
- ▶ Industrial partner provided a model of the engine controller in Simulink, as well as 14 natural -language requirements and 20 test cases



Case Study 1 - Aircraft Engine Controller



Case Study 1 - Aircraft Engine Controller

