Refactoring in Requirements Engineering:

Exploring a methodology for formal verification of safety-critical systems

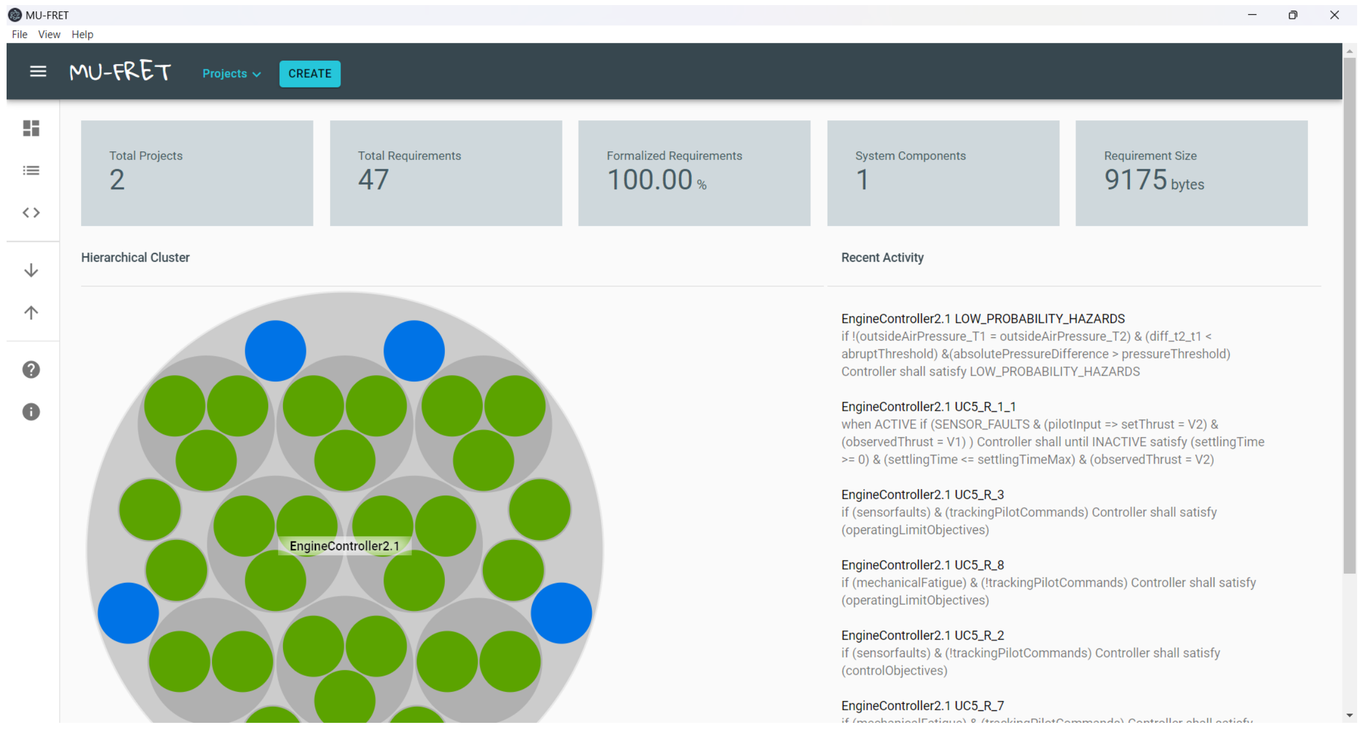
Oisín Sheridan, Rosemary Monahan

Department Of Computer Science, Hamilton Institute

FRET: The Formal Requirements Elicitation Tool

This work is built on the **Formal Requirements Elicitation Tool (FRET)**, a tool developed by NASA for requirements engineering. Requirements for software are commonly expressed in natural language (e.g. plain English), and are often left too vague for direct formalization. One solution to this problem is to use a ***semi-formal*** language; a language that is human-readable, but with a defined structure that allows for automatic translation to a formal logic. FRET allows for requirements written in a structured natural language called **FRETISH**, which is translated into **Linear Temporal Logic (LTL)**. This addresses a common criticism of formal methods; that they are too abstract and too far removed from realistic models used for designs and simulations.

We used FRET to formalize the requirements for an Aircraft Engine Controller as part of the EU's VALU3S project. While doing so, we found that there were many concepts and definitions that were repeated across the set of requirements, and we felt that there should be a better way to manage this. We started our own fork of FRET, called ***Mu-FRET***, and began work on implementing **refactoring** techniques for FRETISH requirements. In Software Engineering, refactoring is a collection of techniques that improves the structure of code without changing its behaviour, e.g. *extracting* a block of code into a function/method, which is then called where that code originally was. The cleaner code or requirements produced by refactoring are easier to maintain, examine, understand, and update.



Extract Requirement:

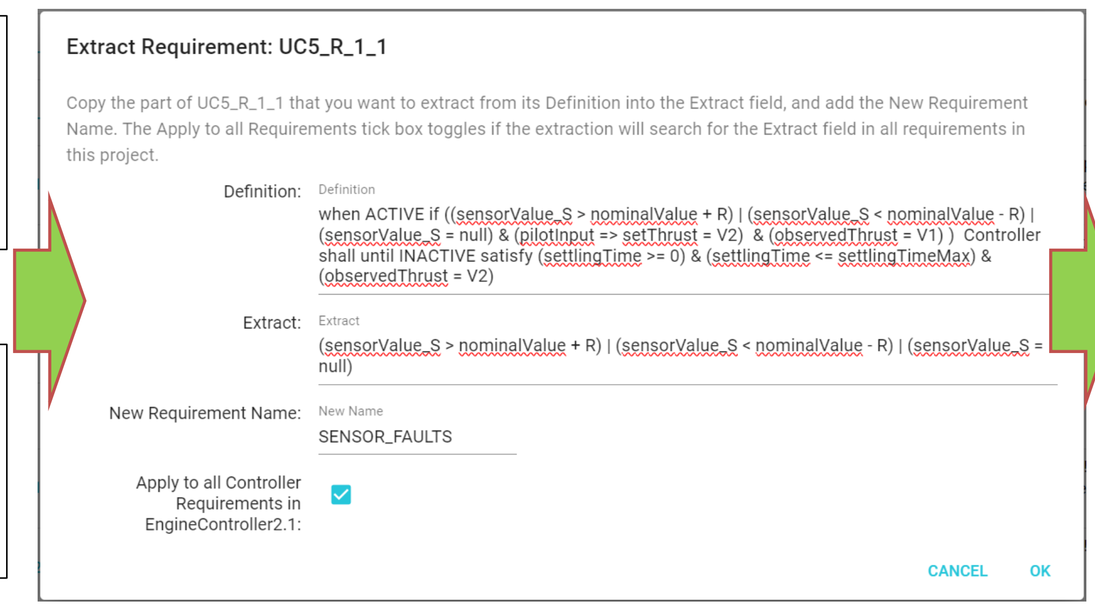
Extract Requirement moves definitions from one requirement (the source requirement) into a newly-created requirement (the destination requirement). In the source requirement, **the extracted parts are replaced with a reference to the new requirement**. Using Extract Requirement to introduce a reference to the newly-created requirement can improve readability, because it can be named to better inform the reader of its intent. This refactoring is especially useful when the definition being extracted **is repeated across several requirements**.

UC5_R_1_1

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

UC5_R_1_2

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



UC5_R_1_1

when ACTIVE if (SENSOR_FAULTS & (pilotInput => setThrust = V2) & (observedThrust = V1)) Controller shall until INACTIVE satisfy (settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)

UC5_R_1_2

when ACTIVE if (SENSOR_FAULTS & (pilotInput => setThrust = V2) & (observedThrust = V1)) Controller shall until INACTIVE satisfy (overshoot >= 0) & (overshoot <= overshootMax) & (observedThrust = V2)

SENSOR_FAULTS

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

Inline Requirement:

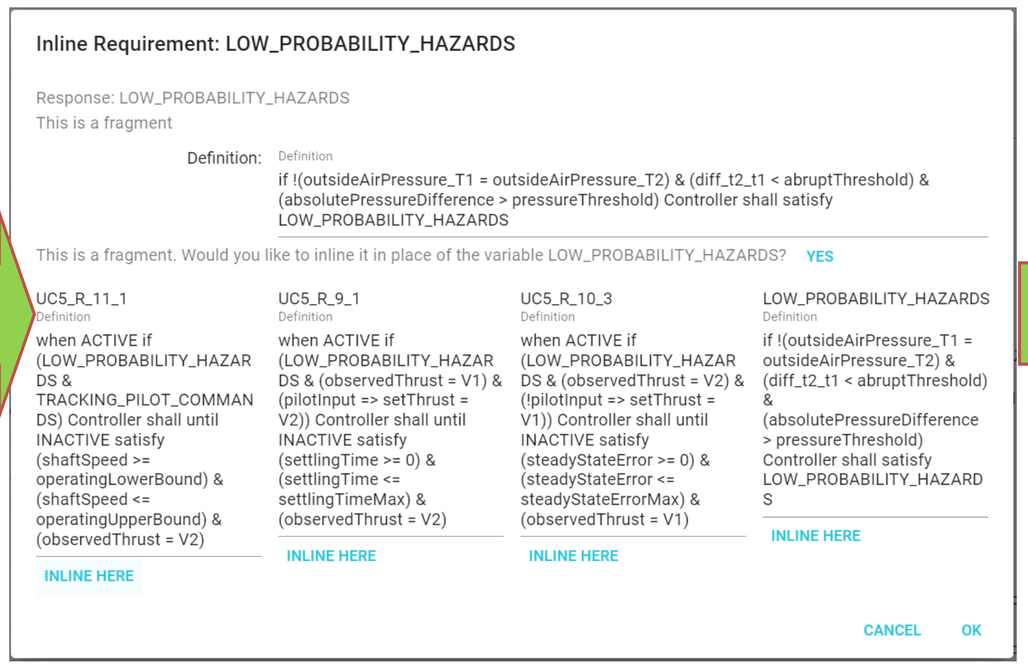
Inline Requirement is the opposite of Extract Requirement; it **merges the entire source requirement into one or more destination requirements**. Inline Requirement is useful where a requirement is **only referenced in a few places**, and is simple enough that its definitions are as clear as its name. Currently, it is only possible to inline FRETISH requirements that follow our specific structure, but we plan to investigate how Inline Requirement might be more generally applicable.

UC5_R_11_1

when ACTIVE if (LOW_PROBABILITY_HAZARDS & TRACKING_PILOT_COMMANDS) Controller shall until INACTIVE satisfy (shaftSpeed >= operatingLowerBound) & (shaftSpeed <= operatingUpperBound) & (observedThrust = V2)

LOW_PROBABILITY_HAZARDS

if !(outsideAirPressure_T1 = outsideAirPressure_T2) & (diff_t2_t1 < abruptThreshold) &(absolutePressureDifference > pressureThreshold) Controller shall satisfy LOW_PROBABILITY_HAZARDS



UC5_R_11_1

when ACTIVE if ((! (outsideAirPressure_T1 = outsideAirPressure_T2) & (diff_t2_t1 < abruptThreshold) & (absolutePressureDifference > pressureThreshold)) & TRACKING_PILOT_COMMANDS) Controller shall until INACTIVE satisfy (shaftSpeed >= operatingLowerBound) & (shaftSpeed <= operatingUpperBound) & (observedThrust = V2)

This research is currently funded by Maynooth University's John and Pat Hume Fellowship.

Initial research was funded by the European Union's Horizon 2020 research and innovation programme under the VALU3S project (grant No 876852), and by Enterprise Ireland (grant No IR20200054). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.