FRET Requirements Refactoring

Matt Luckcuck
Wed 25th Aug 2021

Requirements: EngineController

Starting File: MF-FRET-EngineControllerv1.0.json

Many of the requirements use repeated ideas, things like "sensor faults" or "control objectives", which means that the requirements have a lot of repetition. This is the "Duplicated Activities" problem [Ramos et al. 2007], which is similar to the smell of "Duplicated Code" [Refactoring, Fowler 1999] but we can't apply the usual "Extract Method" refactoring because FRET requirements are not Object-Oriented and FRET doesn't have anything analogous to a method, it only have requirements.

So, to do this refactoring, we're going to have to use other requirements as the destination of the extracted code (FRET requirements, in this case) and use the "Extract Requirement" refactoring from [Ramos et al. 2007]. Since FRET doesn't type check what you give it, we can abuse the notation slightly by saying `if [extracted requirement]`. We already do this in the parent requirements, like `UC5_R_1 = if ((sensorfaults) & (trackingPilotCommands)) Controller shall satisfy (controlObjectives)`; we've just dropped the text of the requirement into FRET, relying on the fact that the we know the child requirements define what `sensorFaults`, `trackingPilotCommands`, and `controlObjectives` mean. But FRET doesn't care and raises no issue with UC5_R_1 or any of the other requirements.

We're going to break this process down into steps.

Step 1: Fragments of FRET

We refer to the repeated ideas/duplicated code as 'fragments' (mainly for the alliteration). These fragments are currently baked into the requirements, so we need to extract them. In our example there are several repeated patterns. The table below shows the first three requirements. For example, requirement R1 specifies: "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"; and R3 specifies "Under sensor faults, while tracking pilot commands, operating limit objectives shall be satisfied (e.g. respecting upper limit in shaft speed)". Both of these requirements are active "Under sensor faults" and "while tracking pilot commands". They only differ in that R1 sates that "control objectives" shall be satisfied, whereas R3 states that "operating limit objectives" shall be satisfied.

Requirement ID	Requirement Text
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

Requirement ID	Requirement Text
R2	Under sensor faults, during regulation of nominal system operation (no change in pilot input), control objectives shall be satisfied (e.g. settling time, overshoot, and steady state error will be within predefined, acceptable limits)
R3	Under sensor faults, while tracking pilot commands, operating limit objectives shall be satisfied (e.g. respecting upper limit in shaft speed)

The table below shows the seven repeated fragments that we discovered.

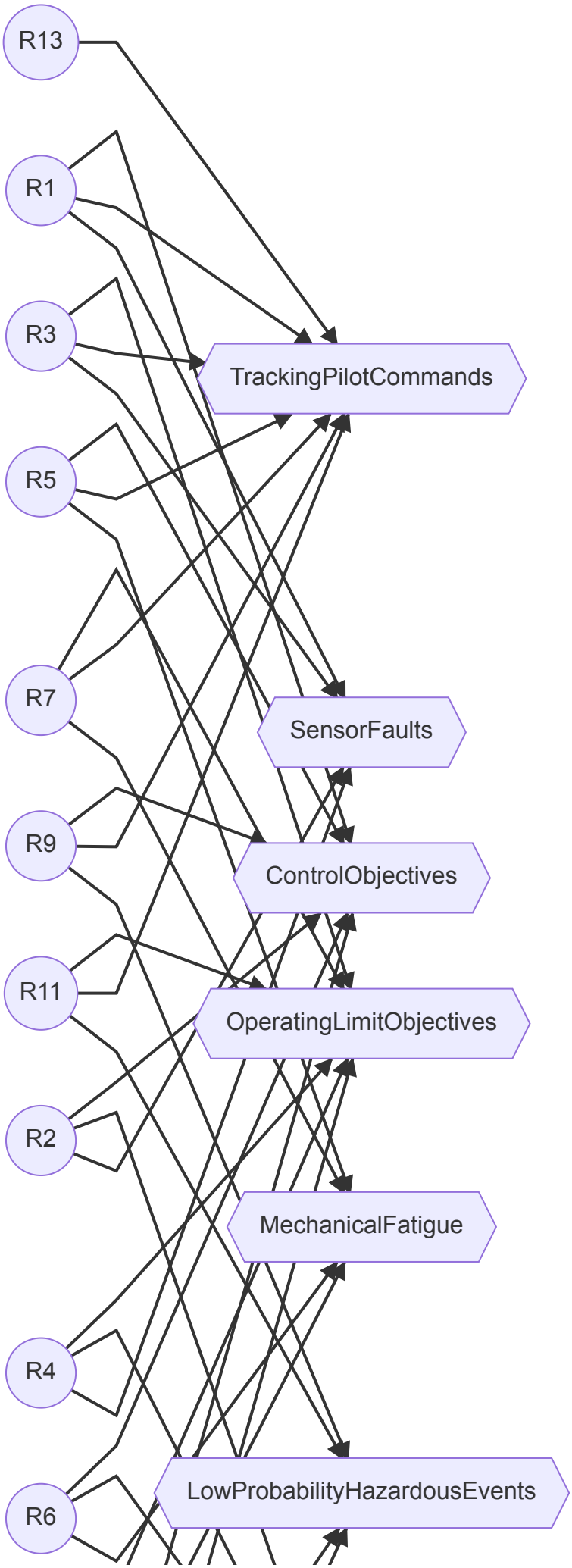
Fragment ID	Description
F1	Sensor Faults
F2	Tracking Pilot Commands
F3	Control Objectives
F4	Regulation Of Nominal Operation
F5	Operating Limit Objectives
F6	Mechanical Fatigue
F7	Low Probability Hazardous Events

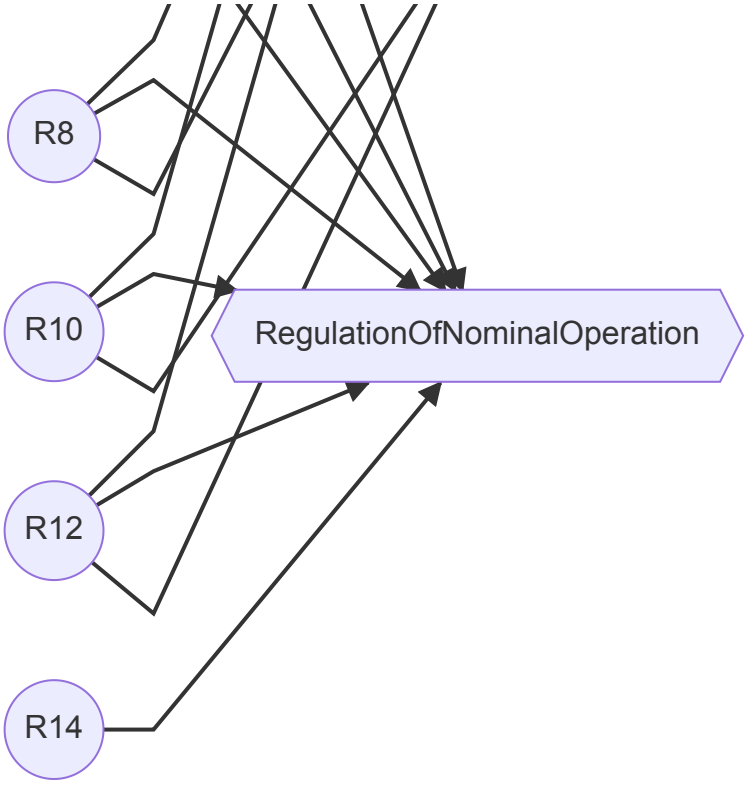
The figure below shows a dependency graph, mapping the requirements to the fragment they depend on. These dependencies are hidden within the requirements, which makes maintaining their definitions troublesome. First, we collect the parent requirements and the fragments into two sets:

$$Reqs = \{R1, R2 \dots R13, R14\}$$
$$Frags = \{F1, F2 \dots F6, F7\}$$

We then provide a relation between these two sets:

$$Dependencies = \{(R1, F1), (R1, F2), (R1, F3) \dots\}$$

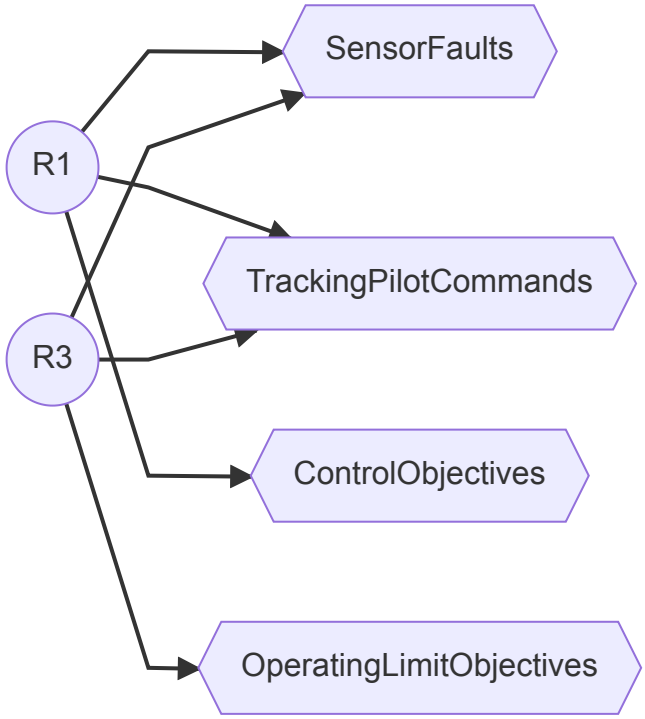

R13
R1
R3
R5
R7
R9
R11
R2
R4
R6
TrackingPilotCommands
SensorFaults
ControlObjectives
OperatingLimitObjectives
MechanicalFatigue
LowProbabilityHazardousEvents



Specifically for our example of requirements R1 and R3, we have:

$$Dependencies = \{(R1, F1), (R1, F2), (R1, F3), (R3, F1), (R3, F2), (R3, F5)\}$$

which produces the dependency graph below (simply a subset of the full graph above).



Step 2: Apply Extract Requirement

In FRETish, R1 becomes `if ((sensorfaults) & (trackingPilotCommands)) Controller shall satisfy (controlObjectives)`. We then implemented R1 as three separate child requirements, for example R1.1 `when (diff(r(i),y(i)) > E) if ((sensorValue(S) > nominalValue + R) | (sensorValue(S) < nominalValue - R) | (sensorValue(S) = null)`

`& (pilotInput => setThrust = V2) & (observedThrust = V1) ) Controller shall until (diff(r(i),y(i)) < e) satisfy (settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)`, which contains the detailed specification of `sensorfaults` (after the `if` and before `Controller`) and `trackingPilotCommands` (the `when()` and `untill()` clauses) but only a partial specification of `controlObjectives`. In the textual version of the requirements there are three control objectives listed, so the child requirements were used to manage the complexity of the specification. However, this further increases the complexity of maintaining the repeated fragments.

Sensor Faults

Applying the Extract Requirement refactoring to the sensor faults fragment was relatively simple, because its definition is simply repeated across several requirements. This definition was extracted to a new requirement `SENSOR_FAULTS = where (sensorValue(S) > nominalValue + R) | (sensorValue(S) < nominalValue - R) | (sensorValue(S) = null) Controller shall satisfy SensorFaults`.

After creating the new requirement, the original repeated definitions were replaced with a 'call' to this new requirement. For example requirements R1 and R1.1 become:

```
R1 = if (SENSOR_FAULTS & (trackingPilotCommands)) Controller shall satisfy (controlObjectives)
R1.1 = when (diff(r(i),y(i)) > E) if (SENSOR_FAULTS & (pilotInput => setThrust = V2) & (observedThrust = V1) ) Controller shall until (diff(r(i),y(i)) < e) satisfy (settlingTime >= 0) & (settlingTime <= settlingTimeMax) & (observedThrust = V2)
```

Control Objectives

Applying the Extract Requirement refactoring to the control objectives fragment was a little more difficult, because it's detailed definition was spread over several child requirements, but was still achievable. Each of the child requirements that used `controlObjectives` (e.g. R1.1, R1.2, and R1.3) contains the detail of one part of the `controlObjectives`: settling time, overshoot, or steady state error, respectively.

Each of the part-specifications of `controlObjectives` was extracted into a single requirement `CONTROL_OBECTIVES` and replaced the part-specifications in the child requirements with a 'call' to the new requirement. For example requirements R1 and R1.1 become:

```
R1 = if (SENSOR_FAULTS & (trackingPilotCommands)) Controller shall satisfy CONTROL_OBECTIVES
R1.1 = when (diff(r(i),y(i)) > E) if (SENSOR_FAULTS & (pilotInput => setThrust = V2) & (observedThrust = V1) ) Controller shall until (diff(r(i),y(i)) < e) satisfy CONTROL_OBECTIVES & (observedThrust = V2)
```

Tracking Pilot Commands

The final fragment, tracking pilot commands, was more difficult to apply the Extract Requirement refactoring to then the previous two fragments. This was because fragment specifies an interval during which the requirement is active and so occurs in two places in the requirement: the `when()` at the beginning and the `until()` just before the response/post-condition of the requirement.

To overcome this challenge, we applied Extract Requirement to each of these sub-fragments so that the structure book-ending the application of the requirement could remain in tact. The first sub-fragment, the 'start condition' for the interval, `when (diff(r(i),y(i)) > E)` was extracted into `TRACKING_PILOT_COMMANDS_START`; and the second sub-fragment, the 'stop condition' for the interval, `until (diff(r(i),y(i)) < e)` was extracted into `TRACKING_PILOT_COMMANDS_STOP`. Each of these was then substituted into the relevant location in the child requirements R1 remains the same and R1.1 becomes:

```
R1.1 = when TRACKING_PILOT_COMMANDS_START if (SENSOR_FAULTS & (pilotInput => setThrust = V2)  & (observedThrust = V1) )  Controller shall until TRACKING_PILOT_COMMANDS_STOP satisfy CONTROL_OBECTIVES & (observedThrust = V2)
```

Step 3: Apply Pull Up Requirement

After the refactoring in Step 2, we were left with a discontinuity between child and parent requirements. The tracking pilot commands start and stop conditions had been extracted to two requirements, but the parent requirement was left with the abstract `trackingPilotCommands`. For example compare `R1 = if (SENSOR_FAULTS & (trackingPilotCommands)) Controller shall satisfy CONTROL_OBECTIVES` with R1.1 in the previous step. This is similar to [code smell] from [Refactoring, Fowler 1999].

To deal with the smell, we apply the Pull Up Requirement refactoring, which is based on the Pull Up Method refactoring in [Refactoring, Fowler 1999]. This is effectively the Move Activity from [Ramos et al. 2007] but here we are explicitly moving the requirement up, from the child requirements to their parent.

We pull the calls to the two tracking pilot commands fragments, and their positioning, up into the parent requirement. For example, R1 becomes:

```
R1 = when TRACKING_PILOT_COMMANDS_START if SENSOR_FAULTS Controller shall until TRACKING_PILOT_COMMANDS_STOP satisfy (CONTROL_OBECTIVES)
```

Step 4: Remove Redundant Requirements

Now that we've extracted the repeated fragments into separate requirements, and pulled up a common requirement, we're left with a lot of identical child requirements, so we can simplify by removing them. For example requirements R1.1, R1.2, and R1.3 all read: `when TRACKING_PILOT_COMMANDS_START if (SENSOR_FAULTS & (pilotInput => setThrust = V2) & (observedThrust = V1)) Controller shall until TRACKING_PILOTC_COMMANDS_STOP satisfy CONTROL_OBJECTIVES & (observedThrust = V2)` showing that we only need one child requirement to implement the detail of requirement R1.

Limitations

This seems to make it much more difficult to export to CoCoSim, because there is no way of telling FRET that you've referenced another requirement. Is this related to parent id?