MANCHESTER 1824
The University of Manchester

The University of Manchester Research

FRETting and Formal Modelling: A Mechanical Lung Ventilator

DOI:
10.1007/978-3-031-63790-2_28

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
Farrell, M., Luckcuck, M., Monahan, R., Reynolds, C., & Sheridan, O. (2024). FRETting and Formal Modelling: A Mechanical Lung Ventilator. In *International Conference on Rigorous State Based Methods* Springer Cham. https://doi.org/10.1007/978-3-031-63790-2_28

Published in:
International Conference on Rigorous State Based Methods

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact openresearch@manchester.ac.uk providing relevant details, so we can investigate your claim.

Download date:09. May. 2026

FRETting and Formal Modelling: A Mechanical Lung Ventilator*

Marie Farrell[1], Matt Luckcuck[2], Rosemary Monahan[3], Conor Reynolds[1], and Oisín Sheridan[3]

[1] Department of Computer Science, The University of Manchester, Manchester, UK
[2] School of Computer Science, University of Nottingham, Nottingham, UK
[3] Department of Computer Science, Maynooth University/Hamilton Institute, Maynooth, Ireland

Abstract. In this paper, we use NASA's Formal Requirements Elicitation Tool (FRET) and the Event-B formal method to model and verify the requirements for the ABZ 2024 case study, the Mechanical Lung Ventilator. We use the FRET requirements to guide the development of a formal design model in Event-B. We provide details about the artefacts produced and reflect on our experience of using these tools in this case study. We focus on the Functional and Controller requirements for the system, as given in the case study documentation. This paper provides a first step towards using Event-B as part of a FRET-guided verification workflow in a large case study.

1 Introduction

We describe a workflow that captures the requirements of the ABZ 2024 case study, the Mechanical Lung Ventilator,[4] using the Formal Requirements Elicitation Tool (FRET) [16]. Our workflow uses the requirements, written in FRET's structured-natural requirements language FRETISH, to guide the development of a formal design in Event-B. We previously proposed this FRET-guided workflow in [22,29,28], and this paper explores how to use FRETISH requirements to guide the development of a formal design in a large case study.

FRET parses FRETISH requirements and translates them into past- and future-time Metric Temporal Logic (MTL) [16], useful for formal verification. MTL extends Linear-time Temporal Logic (LTL) with timing constraints. MTL properties express that a proposition becomes true within a given interval. FRET also produces a *diagramatic* semantics for each requirement that shows the period in which the requirements should hold, and their triggering and stopping conditions. Both representations help to sanity check the requirement's behaviour.

Software requirements are properties that the software must satisfy; they specify what the system should do, without saying how it should do it [31].

* All authors (listed alphabetically by surname) contributed equally to this work. This work was partially supported by the Royal Academy of Engineering and EPSRC grant EP/Y001532/1, as well as Maynooth University's Hume Doctoral Award.

[4] ABZ 2024 Case Study: `https://abz-conf.org/case-study/abz24/`

But requirements specifications are usually written in natural-language, not as a *formal* specification. FRETISH enables us to step from natural-language requirements, to formal requirements in MTL.

A software design should be built to obey the system's requirements. The novelty of our work lies in using FRETISH to guide the development of a formal design in Event-B, into which we can trace the requirements and prove that they are preserved. We do not have a formal translation between FRETISH and Event-B, but the formality of these artefacts supports the development of a *formal* design that preserves the properties specified by the FRETISH requirements.

The FRETISH requirements provide an unambiguous representation of the properties that the system must obey, and the formal design modelled in Event-B ensures that these properties hold. The requirements are represented by several artefacts in the Event-B model (§4), but the formality of both the requirements and design make verification possible.

We begin our description by providing some background prerequisite information in §2 related to FRET, Event-B and the case study. We also provide a brief overview of related work. In §3 we describe how we formalised the requirements given in the case study documentation, and in §4 we discuss how we modelled these in Event-B. We reflect on our approach in §5, and §6 concludes.

2 Background and Related Work

In this section, we provide overviews of FRET and the Event-B formal method that we used for the Mechanical Lung Ventilator case study. We also outline the case study itself and provide a short description of related work.

2.1 Formal Requirements Elicitation Tool (FRET)

FRET is supported by a structured natural-language, called FRETISH, with which users can express requirements [16]. For each requirement, FRET generates a formal semantics in past- and future-time MTL. To aid usability, FRET also produces a diagrammatic semantics to help users to understand the meaning of the requirements. FRET is open-source and available on GitHub.[5]

FRETISH requirements are composed of the following five fields:

`scope` `condition` `component` shall `timing` `response`

The `component` and `response` fields are mandatory (along with the "shall" keyword) for all FRETISH requirements. Using this syntax, users can express requirements for individual `component`s that pertain to a particular `scope` under some `condition` where the `response` (expected behaviour) can be specific to a defined `timing`. Uses of FRET have typically focused on aerospace use cases [13,25], though some studies exist for other domains including robotics [14,10]. This paper reports the first use of FRET for a medical use case.

FRET provides automated translations from FRETISH requirements to CoCoSpec contracts [12], that can be verified with the Kind2 model checker, and

[5] FRET GitHub Repository: `https://github.com/NASA-SW-VnV/fret`

Copilot runtime monitors [26]. There is no automated support for translating between FRETISH requirements and theorem proving approaches like Event-B, though previous work has shown that these proof-based methods are useful for verifying requirements that are difficult to verify using the current supported CoCoSpec approach [10].

2.2 Event-B

Event-B is a state-based formal method that has been used in the development of safety-critical systems in a variety of sectors, including rail [20], aerospace [23] and medical [18]. The Event-B language is based on set theory and first-order logic [2]. Event-B is supported by its Eclipse-based IDE, the Rodin Platform [3]. Event-B, like FRET, is open-source. Specifications in Event-B are composed of *machines* and *contexts*. A machine specifies dynamic behaviour via variants, invariants, and events. Static behaviour is typically specified in a context, using carrier sets, constants, and axioms.

Event-B supports formal refinement, enabling users to gradually add more detail to their model and discharge proofs of correctness at each stage [27]. Theorem proving is supported in Rodin with most of the generated proof obligations discharged automatically, although some may need manual interaction with Rodin's provers. The semantics of Event-B models is often thought of in terms of the generated proof obligations [17], and a detailed formal semantics for the Event-B language itself is given in [15].

2.3 Case Study: Mechanical Lung Ventilator

The case study proposed for ABZ 2024 is a mechanical lung ventilator. An Abstract State Machine (ASM) model of this use case is provided in [8] and it provides the basis for the case study documentation.[6]

During the COVID-19 pandemic, mechanical lung ventilators were an essential piece of medical equipment, supporting patients who were unable to breathe on their own [9,1]. The system provides two ventilation modes: Pressure Controlled Ventilation (PCV) and Pressure Support Ventilation (PSV). The case study description includes a diagram of the system, as shown in Fig 1. We used this architecture as the basis for our first abstract Event-B machine, discussed in §4. We also used it to inform our understanding of the requirements.

The case study documentation provided a large number of requirements for the system. These were partitioned into Functional Requirements (FUN), Values and Ranges (PER), Sensors and Interfaces (INT), Alarm Requirements (SAV), GUI Requirements (GUI), Controller Requirements (CONT), and Alarms (AL). Some requirements have 'child' requirements; for example, FUN6 is decomposed into FUN6_1–6. Requirements also reference others; for example, CONT4 refers to FUN6. The documentation does not make it clear what these relationships are formally and we will discuss this further later.

[6] Mechanical Lung Ventilator Repository: `https://github.com/foselab/abz2024_casestudy_MLV`

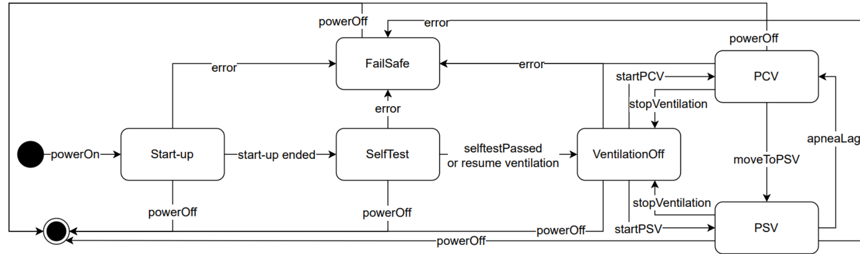

Fig. 1. The controller state machine is labelled as Fig 4.1 in the case study document.

2.4 Related Work

Event-B, and its variants, have been used extensively in ABZ case studies, both in isolation and alongside other tools [23,6,24,21,19]. These case studies, as well as industrial applications of Event-B, emphasise its use in multiple distinct safety-critical domains such as rail, healthcare and aerospace. FRET has not been used for an ABZ case study before; its use has mostly been focused on the aerospace domain [13,25], though some robotics use cases exist [14,10].

One of these robotics use cases made an attempt at linking FRETISH requirements with an Event-B model. This focused on a small part of the robotic system (the planner), and Event-B was chosen because the Kind2 model checker that was used to verify the FRETish requirements timed out on properties concerning the planner [10]. In contrast, our work here seeks to represent more requirements and to specifically examine which parts of the Event-B model were impacted by the FRETISH requirements.

Bonfanti *et al.* [8] describe the experience of modelling the behaviour of a Mechanical Ventilator using a compositional modelling and simulation technique [7]. Their approach is supported by the AsmetaComp tool of the ASMETA toolset for ASMs [5]. Here, separate abstract state machines represent the behaviour of interacting subsystems of the ventilator; where they can communicate with each other through I/O events, and co-operate by a precise orchestration schema.

3 FRETISH Requirements

In this section, we describe our journey from the natural-language requirements given in the case study document to formalised FRETISH requirements. At first, we focused on the functional (FUN) requirements that were identified in the documentation. FRET has been mostly used for functional requirements in the past so this seemed like the logical place for us to start. However, there is a strong relationship between these Functional (FUN) and the Controller (CONT) requirements, so we expanded our scope to also examine the CONT requirements.

We mapped 142 Functional and Controller requirements from their natural-language specification into FRETISH. The requirements are labelled FUN⟨X⟩ and

CONT⟨X⟩, respectively ($X \in \mathbb{N}$); the requirement numbers match the documentation, with the dots replaced with underscores in the child requirements for compatibility with FRET. We omit the dot from the top-level requirements; for example, FUN.1 in the natural-language requirements becomes FUN1.

Table 1 shows a subset of the FRETISH requirements that we could formalise; the full set of FRETISH requirements is available in the Appendix. Fig. 2 shows FRET's dashboard, displaying the total number of requirements in the project, and the percentage that have been formalised (green circles). Requirements that could not be formalised are indicated by a white circle (see §3.2). Requirements that are not valid FRETISH are represented as red circles; for example, CONT36, which is an incomplete sentence in the case study document. Circles within other circles indicate a parent-child relationship between requirements.

Fig. 3 shows an example of a requirement that we were able to formalise, FUN10_6. The requirement's "Rationale" (top left) captures its natural-language version. FRET produces a diagrammatic semantics (top right) from the FRETISH requirement (text box, bottom left) that is useful for sanity checking the requirement. FUN10_6's timing condition, `after 15 minutes`, is reflected in both the diagrammatic semantics (top right) and the temporal logic (bottom right). While FRET labels the temporal logic "Future Time LTL" we can see that the formula is MTL, for example it contains "G[0,15]".

Initially, we only included `timing` in the FRETISH requirements where it was explicitly mentioned in the natural-language version, like in FUN10_6. If a FRETISH requirement has no `timing` condition, then it defaults to `eventually`; so, on a second pass, we rechecked the timing conditions and added them explicitly. Each requirement's timing was considered individually, but we usually used: `always` when the requirement had no conditions, `eventually` for events that would take an indeterminate amount of time (such as waiting for a process to finish or for user input), and `at the next timepoint` for a response triggered by an event or button-press. We chose `at the next timepoint` instead of `immediately` to represent the time taken to react to the trigger and generate the response.

3.1 Formalising Requirements with FRET

We mapped the requirements into FRETISH in several stages, producing sequential versions of the requirements set. As mentioned above, we initially focused on formalising specifically the Functional requirements from the case study, from FUN1 up to FUN42. These were first compiled in v0.1 and v0.2 of the requirements set. We made revisions and additions as a group, producing v0.3 and v0.3.1. Further edits were made in v0.4 to better align with the case study document and fix model checking errors arising from invalid variable names (e.g. the variable "I:E" in the case study had to be changed to "ItoE" in FRETISH).

Following this, we realised that the Controller requirements provided additional detail that would be helpful during the development of the Event-B model. We expanded the scope of the FRETISH formalisation to include the Controller requirements, from CONT1 up to CONT46. These were formalised in v0.5. v0.5.1 formalised one additional requirement, CONT43_3, that was initially omitted.

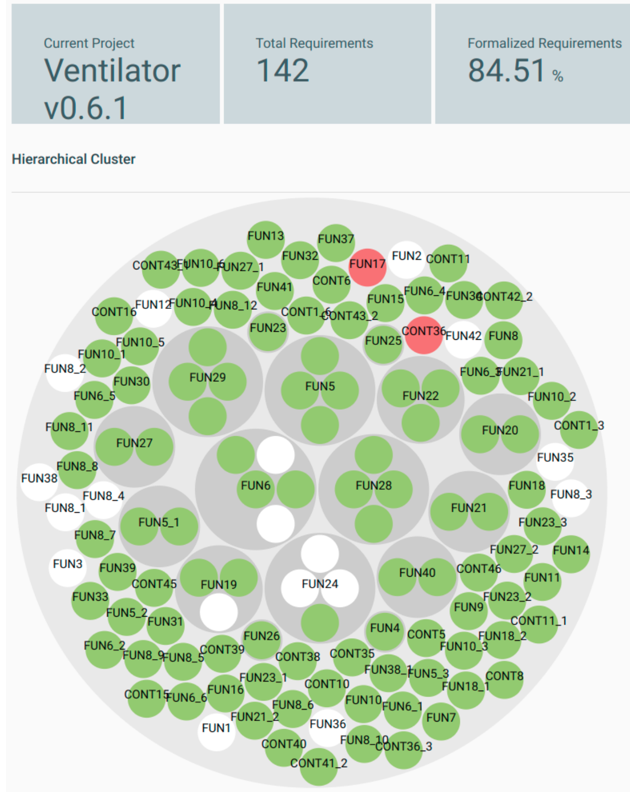

Fig. 2. View of FRET's dashboard for the Ventilator project. Formalised requirements are indicated in green, and those in white have not been formalised. A red circle indicates invalid FRETISH in that requirement.

Finally, we decided to update the FRETISH to include explicit timing conditions in all of the requirements, in v0.6 and v0.6.1.

What follows are some explanations of our methodology when formalising the requirements. For traceability, the names of variables and constants align with those mentioned in the case study, where possible. For example, FUN20:

> *"In PCV mode, the breathing cycle shall be defined by inspiratory pressure P_{insp_PCV} relative to atmosphere, respiratory rate (RR_{PCV}) and the ratio between the inspiratory and expiratory times ($I{:}E_{PCV}$)."*

This natural-language description is accompanied by a note that details how the times are defined. This was formalised in FRETISH as "`in PCVMode System` shall `always satisfy breathingCycleTime = 1/RR_PCV & ExpiratoryTime = breathingCycleTime / (1+ItoE_PCV)`".

We created FRETISH requirements for all of the Functional and Controller requirements, even those that could not be formalised in FRET. The main exceptions to this were CONT4 and CONT39. For example, CONT39 states:

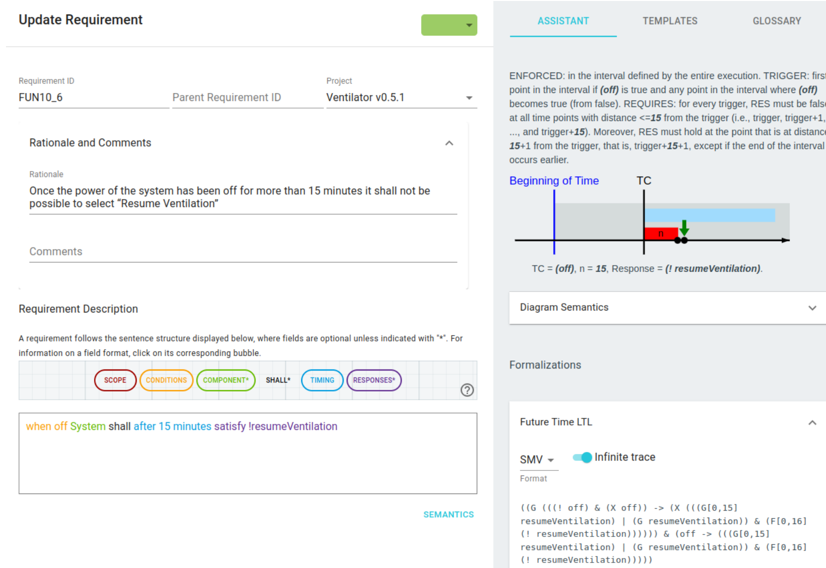

Fig. 3. A screenshot of the FRET window illustrating the formalisation of requirement FUN10_6 documenting the requirement rationale and FRETISH description, as well as the associated diagrammatic semantics and MTL generated by FRET. While the heading says "Future Time LTL", the formula is MTL (e.g. "G[0,15]").

> *"When the ventilator is in an Inspiration state, the out valve shall be closed and the in valve pressure shall be set to target inspiratory pressure (P_{insp} of the corresponding mode)."*

and then each of its three child requirements specify the correct pressure for each mode, e.g. CONT39_2 states *"P_{insp_PSV} if current mode is PSV"*. This format does not lend itself well to FRETISH, as none of these are full requirements on their own. Additionally, a conflict would arise between CONT39_1 (*"P_{insp_PCV} if current mode is PCV"*) and CONT39_3 (*"P_{insp_AP} if current mode is PCV from apnea backup"*) if they were translated directly, requiring additional information to be added to CONT39_1. As a result, these requirements were combined into a single CONT39 as shown in Table 1. CONT4 was a similar case.

3.2 Unformalised Requirements

The unformalised requirements often related to capabilities of the overall system, rather than specifiable behaviour. For example, we did not formalise FUN1:

> *"The system shall provide ventilation support for patients who require mechanical ventilation and weigh more than 40 kg (88 lbs). Rationale: ventilation of children and infants is more challenging",*

because it refers to things that are not within the scope of the functioning system. For example, whether patients require ventilation or not should be determined

by a medical practitioner and not the system. Similarly, the ventilator has no way to determine or check the weight of a given patient. Perhaps a doctor could input this to the machine and it could then issue a warning to the user if the patient were outside the permitted weight range, but we did not find references to any such functionality in the requirements.

We chose not to translate the "Measured and displayed parameters" requirements from §2.2 of the case study, although they are labelled FUN43–FUN58. These requirements specify parameters that the system should measure and display for the patient, e.g. FUN56: *"Value of the temperature inside the system unit is reported"*. There was no meaningful way to capture these requirements in FRET without a more detailed understanding of the sensors and GUI. Since they are in a separate subsection and don't seem to add to the description of the core functionality of the ventilator system, we felt it best to omit them at this time. We chose not to formalise other sections of the case study for this paper, such as the Alarm requirements in §2.5, for similar reasons. These requirements are straightforwardly expressed as boolean conditions in Event-B so they would not have added to our exploration of using FRET to scaffold an Event-B model.

3.3 Analysis

In some cases, once formalised, a CONT requirement matched a FUN requirement exactly, apart from the `component` field. For example, FUN4 and CONT1 both list the operating modes that the system should implement, and were both ultimately formalised as "`System/Controller` shall `always` `satisfy` `StartUpMode | SelfTestMode | StandbyMode | PCVMode | PSVMode | FailSafeMode`".

An obstacle we encountered in some of these cases was that, even when the (seemingly) same underlying behaviour is being described, the language is not entirely consistent. Here, the mode that comes after the self test has passed and before the system moves to PCV or PSV mode is called "Standby Mode" in the FUN requirements, but is named "VentilationOff" in the CONT requirements. We chose to maintain consistency and use StandyByMode in FRETISH.

In other cases, the Controller requirements contained more detail than Functional requirements that described the same behaviour, or vice-versa, or they were structured differently to put more focus on certain aspects of the behaviour. This sometimes led to requirements gaining extra detail. For example, our initial version of FUN5_1 simply read "`in StartUpMode` `System` shall `satisfy initStart`". However, when formalising CONT13 and CONT14, which list FUN5_1 as their "Input ref.", we realised that the description of FUN5_1 contained more functionality that could be included. As such, FUN5_1 was updated to "`in StartUpMode` `System` shall `satisfy initStart & checkCommsSensors & checkCommsValves & checkCommsGUI`" (with explicit timing also added later).

However, there were also cases where a lack of detail led to an apparent conflict between two requirements. This occurred with CONT24 and FUN22, which both specify the "Recruitment Maneuver (RM)" behaviour. FUN22:

> *"In PCV mode it shall be possible to initiate with the push of a single button a lung recruitment procedure, termed Recruitment Maneuver (RM)"*

We interpreted this to mean that RM begins immediately when the corresponding GUI button is pressed during PCV mode. However, CONT24 states:

> *"At the end of an inspiration phase, if inspiratory pause is not required and the Recruitment Maneuver is set by the GUI, a Recruitment Maneuver shall start"*

Note that CONT24 is found in the subsection of Controller requirements that apply to the PCV mode. The Controller requirements only refer to the recruitment manoeuver starting at the end of the inspiration phase as seen here, leading to an apparent conflict with the earlier requirement.

Formalising requirements in a structured language like FRETISH helps to find cases like these where a requirement lacks important details. A similar case is found with CONT19, which states *"If the SelfTest fails, the Controller shall not be able to proceed to ventilation"*. A naive implementation of this could easily create a deadlock, as there is no stated endpoint where the failure stops applying. We added a timing condition "`until off`" to the FRETISH CONT19 for this reason.

4 Event-B Model

This section describes our Event-B modelling approach using both the FRETISH and natural-language requirements. Our primary focus was to explore how the FRETISH requirements are represented in the corresponding Event-B model.

4.1 Modelling the Architecture

A key feature of Event-B is formal refinement: the stepwise development of a system from an abstract to a concrete specification [27]. Event-B model development thus typically begins with an abstract machine capturing the high-level operation of the system. The case study document provides two useful starting-points for a formal model: Fig. 2.1, a "high-level operation diagram"; and Fig. 4.1, the "controller state machine". The controller state machine at first appears as if it is a refinement of the high-level operation diagram, but there are some inconsistencies.

As mentioned earlier, one such inconsistency is reflected in the requirements themselves. Specifically, the "Standby Mode" in Fig. 2.1 appears to correspond to the "VentilationOff" mode in Fig. 4.1. Fig. 2.1 has a transition from "StandBy Mode" to "SelfTest", while Fig. 4.1 has no transition from "VentilationOff" to "SelfTest". This transition does not seem possible and is not described anywhere else in the case study document. After examining the case study documentation, we have concluded that the requirements listed there are more consistent with the controller state machine than with the high-level operation diagram.

As a result of these inconsistencies, we chose to use the controller state machine in Fig. 1 (Fig. 4.1 in the case study documentation) as our canonical high-level operation diagram for the system. The initial Event-B model, `mac00` in Fig. 4 and its context in Fig. 5, correspond directly to Fig. 1. A single variable

<table>
<tr><th>Req ID</th><th>FRETISH</th></tr>
<tr><td>CONT1</td><td>Controller shall always satisfy StartUpMode | SelfTestMode | StandbyMode | PCVMode | PSVMode | FailSafeMode</td></tr>
<tr><td>CONT1_3</td><td>in StandbyMode Controller shall always satisfy ventilationOff & inValveClose & outValveOpen</td></tr>
<tr><td>CONT1_6</td><td>in FailSafeMode Controller shall always satisfy inValveClose & outValveOpen</td></tr>
<tr><td>CONT4</td><td>in SelfTestMode if selfTestPassed | GUIResumeRequest Controller shall at the next timepoint satisfy StandbyMode</td></tr>
<tr><td>CONT13</td><td>in StartUpMode Controller shall eventually satisfy checkCommsSensors & checkCommsValves</td></tr>
<tr><td>CONT14</td><td>in StartUpMode Controller shall eventually satisfy checkCommsGUI</td></tr>
<tr><td>CONT19</td><td>in SelfTestMode if SelfTestFail Controller shall until off satisfy !StandbyMode & !ventilating</td></tr>
<tr><td>CONT39</td><td>while inspiratoryPhase Controller shall always satisfy outValveClose & (PCVMode & !apnea => P_insp = P_isnpPCV) & (PSVMode => P_insp = P_inspPSV) & (PCVMode & apnea => P_insp = P_inspAP)</td></tr>
<tr><td>FUN4</td><td>System shall always satisfy StartUpMode | SelfTestMode | StandbyMode | PCVMode | PSVMode | FailSafeMode</td></tr>
<tr><td>FUN5</td><td>when powerButton & (breathingCircuitConnected & !(patientConnected) & airSupplyConnected & powerConnected) System shall at the next timepoint satisfy StartUpMode</td></tr>
<tr><td>FUN5_1</td><td>in StartUpMode System shall eventually satisfy initStart & checkCommsSensors & checkCommsValves & checkCommsGUI</td></tr>
<tr><td>FUN5_3</td><td>System shall always satisfy (StartUpMode | SelfTestMode) -> !patientConnected</td></tr>
<tr><td>FUN6</td><td>in SelfTestMode System shall eventually satisfy selfTestPassed | selfTestFailed</td></tr>
<tr><td>FUN6_1</td><td>in SelfTestMode System shall eventually satisfy testPowerSwitchPass | testPowerSwitchFail | testPowerSwitchSkip</td></tr>
<tr><td>FUN7</td><td>in SelfTestMode if selfTestFail System shall at the next timepoint satisfy OutOfServiceWarning & FailSafeMode</td></tr>
<tr><td>FUN10</td><td>when startUpDone System shall eventually satisfy newPatient | resumeVentilation</td></tr>
<tr><td>FUN10_1</td><td>when newPatient System shall eventually satisfy patientAttributesEntered & SelfTestMode & ((testPowerSwitchPass & testLeaksPass & testFl2Pass & testPSExpPass & testOxygenSensorPass & testAlarmsPass) => selfTestPassed)</td></tr>
<tr><td>FUN10_2</td><td>when resumeVentilation System shall at the next timepoint satisfy loadLastParams</td></tr>
<tr><td>FUN10_3</td><td>when resumeVentilation System shall eventually satisfy SelfTestMode & (((testPowerSwitchPass | testPowerSwitchSkip) & (testLeaksPass | testLeaksSkip) & (testFl2Pass | testFl2Skip) & (testPSExpPass | testPSExpSkip) & (testOxygenSensorPass | testOxygenSensorSkip) & (testAlarmsPass | testAlarmsSkip)) => selfTestPassed)</td></tr>
<tr><td>FUN10_4</td><td>when selfTestPassed System shall at the next timepoint satisfy StandbyMode</td></tr>
<tr><td>FUN10_6</td><td>when off System shall after 15 minutes satisfy !resumeVentilation</td></tr>
<tr><td>FUN20</td><td>in PCVMode System shall always satisfy breathingCycleTime = 1/RR_PCV & ExpiratoryTime = breathingCycleTime / (1+ItoE_PCV)</td></tr>
</table>

Table 1. Selected FRETISH requirements, the full list is in the Appendix.

```
1  MACHINE mac00
2  SEES ctx00
3  VARIABLES mode
4  INVARIANTS
5     typeof__mode: mode ∈ Mode
6  EVENTS
7  Initialisation
8    then
9       act1: mode := PoweredOff
10  Event PowerOn ≙
11    when
12       grd0_1: mode = PoweredOff
13    then
14       act0_1: mode := StartUp
15   Event StartUpEnded ≙
16     when
17        grd0_1: mode = StartUp
18     then
19        act0_1: mode := SelfTest
20   Event ResumeVentilation ≙
21     when
22        grd0_1: mode = SelfTest
23     then
24        act0_1: mode := VentilationOff
25   Event SelfTestPassed ≙
26     when
27        grd0_1: mode = SelfTest
28     then
29        act0_1: mode := VentilationOff
30   Event StartPCV ≙
31     when
32        grd0_1: mode = VentilationOff
33                ∨ mode = PSV
34     then
35        act0_1: mode := PCV
36   Event StartPSV ≙
37     when
38        grd0_1: mode = VentilationOff
39                ∨ mode = PCV
40     then
41        act0_1: mode := PSV
42   Event StopVentilation ≙
43     when
44        grd0_1: mode = PCV
45                ∨ mode = PSV
46     then
47        act0_1: mode := VentilationOff
48   Event MoveToPSV ≙
49     when
50        grd0_1: mode = PCV
51     then
52        act0_1: mode := PSV
53   Event ApneaLag ≙
54     when
55        grd0_1: mode = PSV
56     then
57        act0_1: mode := PCV
58   Event Error ≙
59     when
60        grd0_1: mode ≠ PoweredOff
61        grd0_2: mode ≠ Failsafe
62     then
63        act0_1: mode := Failsafe
64   Event PowerOff ≙
65     when
66        grd0_1: mode ≠ PoweredOff
67     then
68        act0_1: mode := PoweredOff
69 END
```

Fig. 4. Abstract Event-B machine corresponding to the ventilator behaviour in Fig. 1.

mode indicates the current mode, and transitions between these modes are triggered by events. This method of modelling state-machine diagrams is similar to the approach taken by the iUML-B plugin (though we do not use iUML-B here) [30]. This initial Event-B model captures requirements FUN4 and CONT1.

By encoding the state machine abstractly and supporting new behaviour via refinement of this state machine, we ensure that the refined models switch modes according to the state-transition diagram.

4.2 Encoding the Requirements

We encoded the FRETISH requirements into Event-B in different ways, depending on what they specified. Some requirements were easily represented in a context, others became part of the behavioural event specifications, and some became invariant specifications. We outline how the requirements were represented in

```
1  CONTEXT ctx00
2  SETS  Mode
3  CONSTANTS
4    Failsafe, PoweredOff, VentilationOff
5    PCV, PSV, SelfTest, StartUp
6  AXIOMS
7    axm0_1: partition(Mode, {StartUp},
8              {SelfTest}, {VentilationOff},
9              {PCV}, {PSV}, {Failsafe},
10             {PoweredOff})
11 END
```

Fig. 5. Context for the abstract machine, capturing FUN4/CONT1.

```
1  CONTEXT ctx01
2    EXTENDS ctx00
3    SETS ValveState, TestResult
4    CONSTANTS
5      ValveOpen, ValveClosed, TestPassed
6      TestFailed, TestSkipped
7    AXIOMS
8       axm1_1: partition(ValveState,
9                 {ValveOpen}, {ValveClosed})
10      axm1_2: partition(TestResult,
11                {TestPassed}, {TestFailed},
12                {TestSkipped})
13 END
```

Fig. 6. Extending context to capture necessary sets and constants related to the selftest process (FUN6_1–6).

Event-B in Table 2. Since Event-B does not have direct support for temporal logic, we focused on the FRETISH requirements and associated diagrammatic semantics, rather than attempting to represent MTL directly in Event-B. Hence, we do not provide a systematic translation from FRETISH requirements to Event-B. This is left as future work.

We were curious about the parent-child relationships between requirements that were present in the document. We found that, for FUN10, some of its child requirements were actually specified via event refinement in the Event-B model. For example, FUN10_3 is defined in natural language as

> *If "Resume Ventilation" is selected, every step of the self-test procedure FUN.6 can be skipped or optionally rerun individually.*

The corresponding FRETISH requirement (Table 1) is straightforward and has a response indicating that each step can either be passed or skipped. This requirement prompted us to add new sets and constants by extending `ctx00` using `ctx01` as shown in Figs 5 and 6, respectively. Specifically, we add the `TestPassed`, `TestFailed` and `TestSkipped` constants to allow us to keep track of which tests have passed/failed. In the concrete machine (`mac01`), we include the variables: `testPowerSwitch, testLeaks, testFF12, testPS_EXP, testOxygenSensor` and `testAlarms`; which are updated in the `SelfTestPassed-OrSkipped` (Fig. 7), `SelfTestFailed` and `SelfTestPassed` events. The latter directly models the test passing specification of FUN10_1.

In the concrete `SelfTestPassedOrSkipped` event (Fig. 7), we add the timing parameter, `timePoweredOff` (line 3), which is an integer (line 12) that is less than 15 (line 13), to capture requirement FUN10_6: `when off System` shall `after 15 minutes satisfy !resumeVentilation`. The response part was already captured in the abstract event specification as an action on line 15 of Fig. 7 (parts of the abstract version of this event are in gray) hence the refinement from FUN10 to FUN10_6 in this case is a *superposition* refinement that constrains the event further [4]. Event-B does not have a native way to address timing properties, in contrast to FRETISH, so we assume the existence of a clock from which the

```
1  Event SelfTestPassedOrSkipped ≙
2    REFINES  SelfTestPassed
3    any timePoweredOff
4      when
5        grd0_1: mode = SelfTest
6        grd1_1: testPowerSwitch ∈ {TestPassed, TestSkipped}
7        grd1_2: testLeaks ∈ {TestPassed, TestSkipped}
8        grd1_3: testFF12 ∈ {TestPassed, TestSkipped}
9        grd1_4: testPS_EXP ∈ {TestPassed, TestSkipped}
10       grd1_5: testOxygenSensor ∈ {TestPassed, TestSkipped}
11       grd1_6: testAlarms ∈ {TestPassed, TestSkipped}
12       grd1_7: timePoweredOff ∈ ℤ
13       grd1_8: timePoweredOff ≤ 15 ∧ is_new_patient = FALSE
14     then
15       act0_1: mode := VentilationOff
16       act1_1: in_valve := ValveClosed
17       act1_2: out_valve := ValveOpen
18 END
```

Fig. 7. `SelfTestPassedOrSkipped` event after the first refinement step. This captures requirements FUN10 and some of it's children, along with FUN6. The components in light gray font are included from the abstract event.

```
1 cont1_3: mode = VentilationOff  ⇒  in_valve = ValveClosed ∧ out_valve = ValveOpen
2 cont1_6: mode = Failsafe  ⇒  in_valve = ValveClosed ∧ out_valve = ValveOpen
```

Fig. 8. Invariants for CONT1_3 and CONT1_6.

`timePoweredOff` parameter gets its value. Using a clock variable also appears in related work that integrates timing constraints for Event-B [11].

We did not model FUN10_2 explicitly in Event-B since we assume that global variable updates are sufficient to keep the last parameters available. Interestingly, FUN10_4 was already present in the abstract event (line 15, Fig. 7) since `StandbyMode` corresponds to `VentilationOff` mode.

Some of the parent-child relationships between the requirements could be modelled (and verified) via refinement in Event-B. For example FUN10 and FUN10_6 as described earlier. However, not all of these relationships could be easily represented by refinement; for example FUN5 and FUN5_3. These requirements were both captured in the same Event-B machine as they seemed to us to be at the same level of abstraction, FUN5_3 caused us to add functionality related to `SelfTestMode` which was not mentioned in FUN5. FUN5_3 also inspired the addition of an invariant. Exploring the role that refinement plays in the parent-child requirements relationship more generally is left as future work.

Due to time constraints, we did not explore the formalisation of all of the requirements in Event-B. That said, we have identified over 60 of the remaining FUN and CONT requirements that should be formalisable in future refinement steps. These include, for example FUN18 and its children, and CONT40–45. We intend to continue working on this case study and explore this further.

FRETISH Requirement ID	Context(s)	Event(s)	Invariant(s)	Event-B File(s)
FUN4	✓	✓		mac00, ctx00
FUN5		✓		mac01
FUN5_3		✓	✓	mac01
FUN6	✓	✓		mac00, mac01, ctx01
FUN6_1–FUN6_6	✓	✓		mac01, ctx01
FUN7		✓		mac01
FUN10		✓		mac00
FUN10_1	✓	✓		mac01, ctx01
FUN10_3–FUN10_6		✓		mac01
FUN23		✓		mac01
FUN27		✓		mac01
CONT1	✓	✓		mac00, ctx00
CONT1_1		✓		mac01
CONT1_3			✓	mac01
CONT1_6			✓	mac01
CONT3		✓		mac00
CONT4		✓		mac00
CONT12		✓		mac00, mac01
CONT18	✓	✓		mac01, ctx01
CONT19		✓		mac01
CONT38			✓	mac01
CONT46		✓		mac01

Table 2. Moving between FRETISH and Event-B contexts, events and invariants.

4.3 Verification

The Rodin Platform generates proof obligations for Event-B models, which can be discharged automatically or interactively using Rodin's theorem provers. These proof obligations relate to properties including well-definedness, event feasibility, invariant preservation, and refinement [17]. We were able to discharge all 79 proof obligations generated by Rodin automatically.

Some requirements were verified by construction. Specifically, adherence to the controller state machine (Fig. 1) is obtained by constructing an Event-B model that evolves following the mode changes indicated by the arrows in Fig. 1. There is no corresponding proof obligation in Rodin; it is purely a matter of modelling the behaviour according to the document. Thus, we consider requirements referring to this sequence of states, e.g. FUN4, to be correct-by-construction.

Other requirements are verified more directly, by inspecting the guard or action of the event that corresponds to the behaviour described by that requirement. For example, the start-up procedure may only proceed if the system is connected to the breathing circuit, air supply, and power source. The guard of the `StartUpProcedure` event in `mac01` ensures these three things are `true` and otherwise cannot proceed. This is similar to the correct-by-construction approach above; but it also involves event refinement, so proof obligations related to event refinement (e.g. guard strengthening) were discharged for these requirements.

We modelled some requirements as invariants in `mac01`. Fig. 8 shows the Event-B encoding of the invariants for CONT1_3 and CONT1_6. Invariants must be `true` before and after every event. This is a classical correctness property

in Event-B, and Rodin generates specific invariant preservation proof obligations to ensure that the user considers this property.

5 Discussion

This section discusses how we link requirements with our formal model, and reflects on how some of the functional requirements contain elements related to both the controller and GUI.

Linking Requirements and Formal Specification
One key aim of this work was to examine how the requirements can be integrated into a formal design model. As previously mentioned, the Event-B model captures the requirements in a variety of ways. Table 2 summarises how each FRETISH requirement was captured in the Event-B model; in a context, as an event, as an invariant, or as a combination of these three things.

Modelling the transitions between the states in Fig. 1 was straightforward, and captured FUN4 and other requirements relating to mode changes. These requirements were easily specified using events to describe the required behaviour. In `ctx00`, the `Mode` set enumerates the various modes that the system can be in (line 2, Fig. 1). The current mode is represented by the `mode` variable in `mac00` (Fig. 4). The `mode` is updated when an event triggers a mode change, for example `PowerOn` on lines 10–14 of Fig. 4.

The refined machine, `mac01` (supported by `ctx01`) captured more requirements; for example FUN6_1, which was added via event refinement. In this refinement step, we were able to directly translate some requirements into a behavioural specification in the Event-B model. For example, a requirement's `condition` often became part of the guard of an event, and the `response` often became part of the event's action.

Some requirements were translated into the refined Event-B model as invariants. As outlined in §1, requirements are properties that the software must satisfy; so we initially suspected that this would be the most common way to express FRETISH requirements in Event-B. However, for this set of requirements at least, Event-B seems to more naturally capture requirements in events. In our future work with this case study, it will be interesting to see if certain classes of requirements are more likely to become invariants. Perhaps the functional and controller requirements were more specific to behaviour than others.

When constructing the Event-B model, we used both the FRETISH and the natural-language requirements as sources of information. Both were useful to help with modelling. First, mapping the natural-language requirements into FRETISH enabled us to clearly express our understanding of the requirements. However, we were unable to clarify some details in the requirements without a domain expert. To help fill gaps in our group's knowledge, we consulted more general guidance on how mechanical lung ventilators work. In previous work [13], we were able to collaborate closely with the authors of the natural-language requirements; we see the benefit of that collaboration through its absence here.

Controller and GUI Attributes in Functional Requirements
Our work focuses on requirements for the Mechanical Lung Ventilator's control software, so we have captured the Functional (FUN) and Controller (CONT) requirements and used them to build a formal design. We leave the GUI requirements as future work.

We found that some of the FUN natural-language requirements specify functionality for both the Controller and the GUI at the same time. This is not overly surprising, since the FUN requirements are a high-level specification of the system's requirements, which are 'implemented' by the CONT and GUI requirements. However, this means that our translation into FRETISH captures the parts of the FUN requirements that were needed for the controller, but treats parts that seem to relate to the GUI more trivially.

A good example of our approach is FUN7, which says: "*if the self-test mode fails, the user shall be warned that the system is out-of-service. In addition, any other operations shall be not allowed*". When we captured this in FRETISH (see Table 1), the "*if the self-test mode fails*" is the `condition` that triggers the requirement. But the clause after the comma, "*the user shall be warned that the system is out-of-service*", reads like a requirement of the GUI, so it is captured with a boolean that trivially states that there is a warning. The second sentence, however, "*In addition, any other operations shall be not allowed*", reads like a Controller requirement that signals a mode change. The full FRETISH requirement is shown in Table 1. The `FailSafeMode` variable denotes that the system should change to Fail Safe mode; this was our interpretation of "any other operations shall be not allowed" given the state machine in Fig. 1.

Other examples of this are requirements FUN13–17. They all "measure... and display..." a sensor reading. "Measure" seems to suggest there is a sensor, and a variable in the Controller will hold the result of reading the sensor. "Display" seems like a GUI requirement, for which we are not creating a design. Each of these requirements follow a similar pattern, for example FUN14: `System` shall `always` `satisfy` `measureO2% & displayO2%`.

6 Conclusion

This paper described our use of FRET and Event-B to formalise and model the requirements for the ABZ 2024 Mechanical Lung Ventilator case study.[7] We discuss our approach to providing traceability from natural-language requirements to a formal design model, using the FRETISH structured requirements language as an intermediate step. We reflect on the requirements that we could and could not formalise with FRET. Then we examine how these requirements were specified and subsequently verified in the Event-B model. This work allowed us to explore the link between formal requirements and formal specifications. In this case, requirements became both behavioural specifications and correctness conditions (invariants) in the formal model. This case study provides a basis for linking FRET and Event-B which we intend to explore in future work.

[7] FRET and Event-B artefacts: `https://github.com/mariefarrell/abz2024`

References

1. A Abba, C Accorsi, P Agnes, E Alessi, P Amaudruz, A Annovi, F Ardellier Desages, S Back, C Badia, J Bagger, et al. The novel mechanical ventilator milano for the covid-19 pandemic. *Physics of Fluids*, 33(3), 2021.
2. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.
4. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.
5. Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. *The ASMETA Approach to Safety Assurance of Software Systems*, pages 215–238. Springer, 2021.
6. Richard Banach. The Landing Gear Case Study in Hybrid Event-B. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 126–141. Springer, 2014.
7. Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Compositional Simulation of Abstract State Machines for Safety Critical Systems. In *Formal Aspects of Component Software*, pages 3–19. Springer, 2022.
8. Silvia Bonfanti, Elvinia Riccobene, Davide Santandrea, and Patrizia Scandurra. Modeling the MVM-Adapt System by Compositional I/O Abstract State Machines. In *Rigorous State-Based Methods*, pages 107–115. Springer, 2023.
9. Walter Bonivento, Angelo Gargantini, Reiner Krücken, and Alessandro Razeto. The mechanical ventilator milano. *Nuclear Physics News*, 31(3):30–33, 2021.
10. Hamza Bourbouh, Marie Farrell, Anastasia Mavridou, Irfan Sljivo, Guillaume Brat, Louise A. Dennis, and Michael Fisher. Integrating Formal Verification and Assurance: An Inspection Rover Case Study. In *NASA Formal Methods*, pages 53–71. Springer, 2021.
11. Dominique Cansell, Dominique Méry, and Joris Rehm. Time Constraint Patterns for Event-B Development. In *B 2007: Formal Specification and Development in B*, pages 140–154. Springer, 2006.
12. Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. CoCoSpec: A Mode-Aware Contract Language for Reactive Systems. In *International Conference on Software Engineering and Formal Methods*, pages 347–366. Springer, 2016.
13. Marie Farrell, Matt Luckcuck, Oisín Sheridan, and Rosemary Monahan. FRETting About Requirements: Formalised Requirements for an Aircraft Engine Controller. In *Requirements Engineering: Foundation for Software Quality*, pages 96–111. Springer, 2022.
14. Marie Farrell, Nikos Mavrakis, Angelo Ferrando, Clare Dixon, and Yang Gao. Formal Modelling and Runtime Verification of Autonomous Grasping for Active Debris Removal. *Frontiers in Robotics and AI*, 2022.
15. Marie Farrell, Rosemary Monahan, and James F Power. Building Specifications in the Event-B Institution. *Logical Methods in Computer Science*, 18, 2022.
16. Dimitra Giannakopoulou, Anastasia Mavridou, Julian Rhein, Thomas Pressburger, Johann Schumann, and Nija Shi. Formal Requirements Elicitation with FRET. In

International Conference on Requirements Engineering: Foundation for Software Quality, 2020.

17. Stefan Hallerstede. On the Purpose of Event-B Proof Obligations. In *International Conference on Abstract State Machines, B and Z, ABZ*, pages 125–138, 2008.
18. Thai Son Hoang, Colin Snook, Lukas Ladenberger, and Michael Butler. Validating the Requirements and Design of a Hemodialysis Machine Using iUML-B, BMotion Studio, and Co-Simulation. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 360–375, 2016.
19. Thai Son Hoang, Colin Snook, Asieh Salehi, Michael Butler, and Lukas Ladenberger. Validating and Verifying the Requirements and Design of a Haemodialysis Machine using the Rodin Toolset. *Science of Computer Programming*, 158:122–147, 2018.
20. Tibor Kiss and Katalin Tünde Jánosi-Rancz. Developing Railway Interlocking Systems with Session Types and Event-B. In *International Symposium on Applied Computational Intelligence and Informatics, SACI*, pages 93–98. IEEE, 2016.
21. Lukas Ladenberger, Dominik Hansen, Harald Wiegard, Jens Bendisposto, and Michael Leuschel. Validation of the ABZ Landing Gear System Using ProB. *International Journal on Software Tools for Technology Transfer*, 19:187–203, 2017.
22. Matt Luckcuck, Marie Farrell, Oisíin Sheridan, and Rosemary Monahan. A Methodology for Developing a Verifiable Aircraft Engine Controller from Formal Requirements. In *IEEE Aerospace Conference*, pages 1–12, 2022.
23. Amel Mammar and Régine Laleau. Modeling a Landing Gear System in Event-B. *International Journal on Software Tools for Technology Transfer*, 19:167–186, 2017.
24. Amel Mammar and Michael Leuschel. Modeling and Verifying an Arrival Manager using Event-B. In *International Conference on Rigorous State-Based Methods*, pages 321–339. Springer, 2023.
25. Anastasia Mavridou, Hamza Bourbouh, Dimitra Giannakopoulou, Tom Pressburger, Mohammad Hejase, Pierre-Loic Garoche, and Johann Schumann. The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained. In *Proceedings of the 28th IEEE International Requirements Engineering Conference*, 2020.
26. Ivan Perez, Anastasia Mavridou, Tom Pressburger, Alwyn Goodloe, and Dimitra Giannakopoulou. Automated Translation of Natural Language Requirements to Runtime Monitors. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 387–395. Springer, 2022.
27. Steve Schneider, Helen Treharne, and Heike Wehrheim. The Behavioural Semantics of Event-B Refinement. *Formal Aspects of Computing*, 26:251–280, 2014.
28. Oisín Sheridan. Exploring a Methodology for Formal Verification of Safety-Critical Systems. In *Rigorous State-Based Methods*, pages 361–365. Springer, 2023.
29. Oisín Sheridan, Rosemary Monahan, and Matt Luckcuck. A Requirements-Driven Methodology: Formal Modelling and Verification of an Aircraft Engine Controller. In *Integrated Formal Methods*, page 352–356. Springer, 2022.
30. Colin Snook and Michael Butler. UML-B and Event-B: an Integration of Languages and Tools. In *IASTED International Conference on Software Engineering*, pages 336–341, 2008.
31. Ian Sommerville. *Software Engineering*. International Computer Science Series. Addison-Wesley, 1982.

A Formalised Requirements

Table 3: FRETISH Requirements.

Req ID	FRETISH
CONT1	Controller shall always satisfy StartUpMode | SelfTestMode | StandbyMode | PCVMode | PSVMode | FailSafeMode
CONT1_1	in StartUpMode Controller shall eventually satisfy initStart & (initDone | (initFail & OutOfServiceWarning & FailSafeMode))
CONT1_3	in StandbyMode Controller shall always satisfy ventilationOff & inValveClose & outValveOpen
CONT1_6	in FailSafeMode Controller shall always satisfy inValveClose & outValveOpen
CONT2	if powerOff & !powerButton Controller shall always satisfy !StartUpMode
CONT3	in StartUpMode when initDone Controller shall at the next timepoint satisfy SelfTestMode
CONT4	in SelfTestMode if selfTestPassed | GUIResumeRequest Controller shall at the next timepoint satisfy StandbyMode
CONT5	in StandbyMode if PSVModeSelected Controller shall at the next timepoint satisfy PSVMode
CONT6	in StandbyMode if PCVModeSelected Controller shall at the next timepoint satisfy PCVMode
CONT7	in PCVMode if inspiratoryPhaseEnd & PSVModeSelected Controller shall at the next timepoint satisfy ventilating & PSVMode
CONT8	in PCVMode if stopVentilation Controller shall at the next timepoint satisfy StandbyMode
CONT9	in PSVMode when breathingTime >= apneaLagTime Controller shall at the next timepoint satisfy PCVMode & RR = RR_AP & P_insp = P_inspAP & ItoE = ItoE_AP
CONT10	in PSVMode if stopVentilation Controller shall at the next timepoint satisfy StandbyMode
CONT11	if powerOff Controller shall at the next timepoint satisfy FinalState
CONT11_1	in FinalState Controller shall eventually satisfy parametersStored & off
CONT12	in StartUpMode Controller shall eventually satisfy defaultParamsLoaded
CONT13	in StartUpMode Controller shall eventually satisfy checkCommsSensors & checkCommsValves
CONT14	in StartUpMode Controller shall eventually satisfy checkCommsGUI
CONT15	in StartUpMode if pressureSensorRetries >= 5 & (pressureSensorConnFailure | pressureSensorError) Controller shall at the next timepoint satisfy FailSafeMode
CONT16	in StartUpMode if ADCRetries >= 5 & (ADCConnFailure | ADCError) Controller shall at the next timepoint satisfy FailSafeMode

CONT18	in SelfTestMode Controller shall eventually satisfy selfTestPassed | selfTestFailed
CONT19	in SelfTestMode if SelfTestFail Controller shall until off satisfy !StandbyMode & !ventilating
CONT20	in PCVMode Controller shall always satisfy breathingCycleTime = 1/RR_PCV & ExpiratoryTime = breathingCycleTime / (1+ItoE_PCV)
CONT21	in PCVMode when BreathingCycleStart Controller shall at the next timepoint satisfy inspiratoryPhaseStart
CONT22	in PCVMode Controller shall always satisfy inspiratoryTime = 60*(ItoE_PCV/(RR_PCV * (1 + ItoE_PCV)))
CONT23	while PCVMode & inspiratoryPauseButton when inspiratoryPhaseEnd Controller shall for 40 seconds satisfy (inspiratoryPauseButton => inValveClose & outValveClose)
CONT24	while PCVMode & RMButton when inspiratoryPhaseEnd Controller shall at the next timepoint satisfy RM
CONT25	in PCVMode Controller shall always satisfy (breathingCycleDone | patientBreathingRequest) => (breathingCycleStart & inspiratoryPhaseStart)
CONT26	while PCVMode when dropPAW > ITS_PCV Controller shall at the next timepoint satisfy patientBreathingRequest
CONT27	in PCVMode when expiratoryPauseButton & ExpiratoryPhaseEnd & !patientBreathingRequest Controller shall until buttonUnPressOr60Seconds satisfy expirationPhaseEnd & inValveClose & outValveClose
CONT28	in PCVMode Controller shall always satisfy P_insp = P_inspPCV
CONT30	in PSVMode when BreathingCycleStart Controller shall at the next timepoint satisfy inspiratoryPhaseStart
CONT32	in PSVMode Controller shall until (P_insp >= MaxP_insp | inspClock >= inspiratoryTime) satisfy inspiratoryPhase
CONT33	in PSVMode when V_E < ExpiratoryTriggerSensitivity*PeakV_E Controller shall at the next timepoint satisfy expirationPhaseStart
CONT34	while PSVMode & inspiratoryPauseButton when inspiratoryPhaseEnd Controller shall for 40 seconds satisfy (inspiratoryPauseButton => inValveClose & outValveClose)
CONT35	while PSVMode & RMButton when inspiratoryPhaseEnd Controller shall at the next timepoint satisfy RM
CONT36_1	while PSVMode & expiratoryPhase when dropPAW > ITS_PSV Controller shall at the next timepoint satisfy patientBreathingRequest
CONT36_2	while PSVMode when expiratoryPauseButton & expClock <= apneaLagTime & !patientBreathingRequest Controller shall until buttonUnPressOr60Seconds satisfy expirationPhaseEnd & inValveClose & outValveClose
CONT36_3	in PSVMode Controller shall always satisfy minExpiratoryTime >= 0.4 & minExpiratoryTime <= 2 & (expiratoryPhase => minExpiratoryTime = inspClock/2)

<table>
<tr><td>CONT37</td><td>in PSVMode when expClock >= apneaLagTime Controller shall at the next timepoint satisfy apnea & PCVMode & RR = RR_AP & P_insp = P_inspAP & ItoE = ItoE_AP</td></tr>
<tr><td>CONT38</td><td>while StartUpMode | StandbyMode Controller shall always satisfy inValveClose & outValveOpen</td></tr>
<tr><td>CONT39</td><td>while inspiratoryPhase Controller shall always satisfy outValveClose & (PCVMode & !apnea => P_insp = P_isnpPCV) & (PSVMode => P_insp = P_inspPSV) & (PCVMode & apnea => P_insp = P_inspAP)</td></tr>
<tr><td>CONT40</td><td>while expiratoryState Controller shall always satisfy inValveClose & outValveOpen</td></tr>
<tr><td>CONT41</td><td>while (PCVMode | PSVMode) when inspiratoryPauseButton Controller shall eventually satisfy inspiratoryPause | !inspiratoryPauseButton</td></tr>
<tr><td>CONT41_1</td><td>while (PCVMode | PSVMode) & inspiratoryPauseButton when inspiratoryPhaseEnd Controller shall at the next timepoint satisfy (inspiratoryPauseButton => inspiratoryPause & inValveClose & outValveClose)</td></tr>
<tr><td>CONT41_2</td><td>when inspiratoryPause Controller shall after 40 seconds satisfy !inspiratoryPause & expirationPhaseStart</td></tr>
<tr><td>CONT42</td><td>while (PCVMode | PSVMode) when expiratoryPauseButton Controller shall eventually satisfy expiratoryPause | !expiratoryPauseButton</td></tr>
<tr><td>CONT42_1</td><td>while (PCVMode | PSVMode) & expiratoryPauseButton when expiratoryPhaseEnd Controller shall at the next timepoint satisfy (expiratoryPauseButton => expiratoryPause & inValveClose & outValveClose)</td></tr>
<tr><td>CONT42_2</td><td>when expiratoryPause Controller shall after 40 seconds satisfy !expiratoryPause & inspiratoryPhaseStart</td></tr>
<tr><td>CONT43</td><td>while (PCVMode | PSVMode) & RMButton & !inspiratoryPauseButton when inspiratoryPhaseEnd Controller shall at the next timepoint satisfy RM</td></tr>
<tr><td>CONT43_1</td><td>while RM if !RMButton Controller shall at the next timepoint satisfy !RM & expirationPhaseStart</td></tr>
<tr><td>CONT43_2</td><td>when RM Controller shall after 10 seconds satisfy !RM & expirationPhaseStart</td></tr>
<tr><td>CONT43_3</td><td>while RM Controller shall always satisfy outValveClose & inValveOpen</td></tr>
<tr><td>CONT44</td><td>if P_insp > MaxP_insp Controller shall at the next timepoint satisfy inspiratoryPhaseEnd & expirationPhaseStart</td></tr>
<tr><td>CONT45</td><td>when expirationPhaseStart Controller shall after 700 milliseconds satisfy monitorInhaleTrigger</td></tr>
<tr><td>CONT46</td><td>after FailSafeMode Controller shall until off satisfy !(StartUpMode | SelfTestMode | StandbyMode | PCVMode | PSVMode)</td></tr>
<tr><td>FUN4</td><td>System shall always satisfy StartUpMode | SelfTestMode | StandbyMode | PCVMode | PSVMode | FailSafeMode</td></tr>
<tr><td>FUN5</td><td>when powerButton & (breathingCircuitConnected & !(patientConnected) & airSupplyConnected & powerConnected) System shall at the next timepoint satisfy StartUpMode</td></tr>
<tr><td>FUN5_1</td><td>in StartUpMode System shall eventually satisfy initStart & checkCommsSensors & checkCommsValves & checkCommsGUI</td></tr>
</table>

FUN5_2	in StartUpMode System shall eventually satisfy initDone | (initFail & OutOfServiceWarning & FailSafeMode)
FUN5_3	System shall always satisfy (StartUpMode | SelfTestMode) -> !patientConnected
FUN6	in SelfTestMode System shall eventually satisfy selfTestPassed | selfTestFailed
FUN6_1	in SelfTestMode System shall eventually satisfy testPowerSwitchPass | testPowerSwitchFail | testPowerSwitchSkip
FUN6_2	in SelfTestMode System shall eventually satisfy testLeaksPass | testLeaksFail | testLeaksSkip
FUN6_3	in SelfTestMode System shall eventually satisfy testFl2Pass | testFl2Fail | testFl2Skip
FUN6_4	in SelfTestMode System shall eventually satisfy testPSExpPass | testPSExpFail | testPSExpSkip
FUN6_5	in SelfTestMode System shall eventually satisfy testOxygenSensorPass | testOxygenSensorFail | testOxygenSensorSkip
FUN6_6	in SelfTestMode System shall eventually satisfy testAlarmsPass | testAlarmsFail | testAlarmsSkip
FUN7	in SelfTestMode if selfTestFail System shall at the next timepoint satisfy OutOfServiceWarning & FailSafeMode
FUN8	System shall always satisfy logParams & saveLog & loadLog
FUN8_5	System shall always satisfy if user = operator then !eraseLog
FUN8_6	when ventilatorSettingsChanged System shall at the next timepoint satisfy logVentilatorSettings
FUN8_7	when alarmSettingsChanged System shall at the next timepoint satisfy logAlarmSettings
FUN8_8	when patientChanged System shall at the next timepoint satisfy logPatientChange
FUN8_9	when powerSupplyChanged System shall at the next timepoint satisfy logPowerSupply
FUN8_10	when preUseCheckDone System shall at the next timepoint satisfy logPreUseCheck
FUN8_11	System shall always satisfy logO2SensorUse
FUN8_12	System shall always satisfy logVentilationParams & logAlarmParams & logCalibrationParams
FUN9	when selfTestPassed System shall at the next timepoint satisfy startMonitoring & startReportingHealthParams & StandbyMode
FUN10	when startUpDone System shall eventually satisfy newPatient | resumeVentilation
FUN10_1	when newPatient System shall eventually satisfy patientAttributesEntered & SelfTestMode & ((testPowerSwitchPass & testLeaksPass & testFl2Pass & testPSExpPass & testOxygenSensorPass & testAlarmsPass) => selfTestPassed)
FUN10_2	when resumeVentilation System shall at the next timepoint satisfy loadLastParams

FUN10_3	when resumeVentilation System shall eventually satisfy SelfTestMode & (((testPowerSwitchPass | testPowerSwitchSkip) & (testLeaksPass | testLeaksSkip) & (testFl2Pass | testFl2Skip) & (testPSExpPass | testPSExpSkip) & (testOxygenSensorPass | testOxygenSensorSkip) & (testAlarmsPass | testAlarmsSkip)) => selfTestPassed)
FUN10_4	when selfTestPassed System shall at the next timepoint satisfy StandbyMode
FUN10_5	in StandbyMode System shall always satisfy ventilationOff & ventilationParmsAdjustable
FUN10_6	when off System shall after 15 minutes satisfy !resumeVentilation
FUN11	System shall always satisfy GBPS <= 5.2
FUN13	System shall always satisfy measureRR & displayRR
FUN14	System shall always satisfy measureO2% & displayO2%
FUN15	System shall always satisfy measurePSins
FUN16	System shall always satisfy measureTV & displayTV
FUN17	System shall always satisfy measureFl1 & display Fl1
FUN18	System shall always satisfy (if enableLeakCompensation then leakCompensation) | !leakCompensation
FUN18_1	System shall always satisfy (if enableLeakCompensation then leakCompensation) & (if disableLeakCompensation then (!leakCompensation & !enableLeakCompensation))
FUN18_2	when leakCompensationEnable if MinPEEPAlarm System shall at the next timepoint satisfy leakCompensationActive
FUN20	in PCVMode System shall always satisfy breathingCycleTime = 1/RR_PCV & ExpiratoryTime = breathingCycleTime / (1+ItoE_PCV)
FUN21	in PCVMode System shall always satisfy (breathingCycleDone | patientBreathingRequest) => breathingCycleStart
FUN21_1	when inspiratoryPressure < InhaleTriggerSensitivityPCV System shall at the next timepoint satisfy breathingCycleStart
FUN21_2	when patientBreathTrigger System shall at the next timepoint satisfy breathingTimerReset
FUN22	in PCVMode when RMButton System shall at the next timepoint satisfy RM
FUN23	in PCVMode when PSVModeSelected System shall at the next timepoint satisfy ventilating & PSVMode
FUN23_1	in PCVMode when PSVModeSelected System shall eventually satisfy confirmPSVParameters
FUN23_2	in PCVMode System shall always satisfy ((confirmPSVParameters & PSVMode) | (!confirmPSVParameters & PCVMode)) & ventilating
FUN23_3	in PCVMode when PSVModeSelected System shall at the next timepoint satisfy !(PCVInspTimeEnd & PSVMode)
FUN25	in PSVMode when inspiratoryPressure < InhaleTriggerSensitivityPSV System shall at the next timepoint satisfy breathingCycleStart
FUN26	in PSVMode when Fl1 < ExpiratoryTriggerSensitivity System shall at the next timepoint satisfy expirationPhaseStart

FUN27	in PSVMode when breathingTime >= apneaLagTime System shall at the next timepoint satisfy apnea
FUN27_1	if apnea System shall at the next timepoint satisfy apneaAlarm
FUN27_2	if apnea System shall at the next timepoint satisfy PCVMode & RR = RR_AP & P_insp = P_inspAP & ItoE = ItoE_AP
FUN28	when expiratoryPauseButton & (ExpiratoryPhaseEnd) System shall until buttonUnPressOr60Seconds satisfy expirationPhaseEnd & inValveClose & outValveClose
FUN29	while inspiratoryPauseButton when (inspiratoryPhaseEnd) System shall for 40 seconds satisfy (inspiratoryPauseButton => inValveClose & outValveClose)
FUN30	System shall always satisfy if StartUpMode then (if newPatient then SelfTestMode & if !newPatient then StandbyMode) & if SelfTestMode then (if selfTestPassed then StandbyMode) & if StandbyMode then (if startPCV then PCVMode & if startPSV then PSVMode & if runSelfTest then SelfTestMode) & if error then FailSafeMode & if powerOff then off
FUN31	System shall always satisfy patientSafe
FUN32	in FailSafeMode System shall always satisfy patientSafe
FUN33	if powerFailure System shall at the next timepoint satisfy patientSafe
FUN34	if gasSupplyFailure System shall at the next timepoint satisfy patientSafe
FUN37	if powerFailure System shall for 120 minutes satisfy !off
FUN38_1	if param_V > paramMax_V | param_V < paramMin_V System shall at the next timepoint satisfy paramAlarm_V
FUN39	before PSVMode | PCVMode System shall eventually satisfy enterAlarmThresholds
FUN40	if P_insp > MaxP_insp System shall at the next timepoint satisfy inspiratoryPhaseEnd & expirationPhaseStart
FUN41	if GUIFailue | !GUIConnected System shall at the next timepoint satisfy ventilating & highPriorityAlarm