An Event-B Model of a Mechanical Lung Ventilator

Amel Mammar

An Event-B Model of a Mechanical Lung Ventilator*

Amel Mammar[1][0000−0003−0016−6898]

SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, France
amel.mammar@telecom-sudparis.eu

Abstract. In this paper, we present a formal Event-B model of the Mechanical Lung Ventilator (MLV), the case study provided by the ABZ'24 conference. This system aims at helping patients maintain good breathing by providing mechanical ventilation. For this purpose, two modes are possible: *Pressure Controlled Ventilation* (PCV) and *Pressure Support Ventilation* (PSV). In the former mode, respiratory cycles are completely defined by the patient that is able to start breathing on its own. In the latter mode, the respiratory cycle is constant and controlled by the ventilator. Let us note that it is possible to move from a given mode to the other depending on the breathing capabilities of the patient under ventilation. In this paper, we illustrate the use of a correct-by-construction approach, the Event-B formal method and its refinement process, for the formal modeling and the verification of such a complex and critical system. The development of the formal models has been achieved under the Rodin platform that provides us with automatic and interactive provers used to verify the correctness of the models. We have also validated the built Event-B models using the ProB animator/model checker.

Keywords: Mechanical Lung Ventilator, System modeling, Event-B method, Refinement, Verification

1 Introduction

The paper presents the formal modelling of a Mechanical Lung Ventilator (MLV), a case study proposed in the context of the ABZ'24 conference. The goal of this system is to offer a support for patients that are in intensive therapy and thus need a mechanical ventilation. This system includes two ventilation modes: *Pressure Controlled Ventilation* (PCV) and *Pressure Support Ventilation* (PSV). The use of a given mode depends on the breathing capabilities of the patient. Basically, the PCV mode is used for patients that are not capable of breathing on their own. In this case, the breathing cycle (among others, inspiration and expiration) is entirely controlled by the ventilator. The PSV mode is used for other patients that can initiate breathing cycles. To ensure the safety of a patient, the

* This work was supported by the ANR projet DISCONT

controller can decide to switch from PSV to PCV when it detects that the patient is not able to re-start inspiration. Similarly, the user(technician/doctor) can ask the controller to move to the PSV mode if he/she assesses that the patient is able to breath on its own. The MLV is composed of two main components, the *controller* and the GUI (Graphical User Interface), that interact with each other while having some independent phases/states and some synchronisation points (see Figure 1): the user sends commands to the controller through the GUI, the controller informs the user about the status of the ventilation using the GUI. According to the requirement document(Requirements **GUI.2** and **GUI.13**), any crash that may happen on the GUI does not affect the controller that continues accomplishing at least the activities that do not require user interactions.

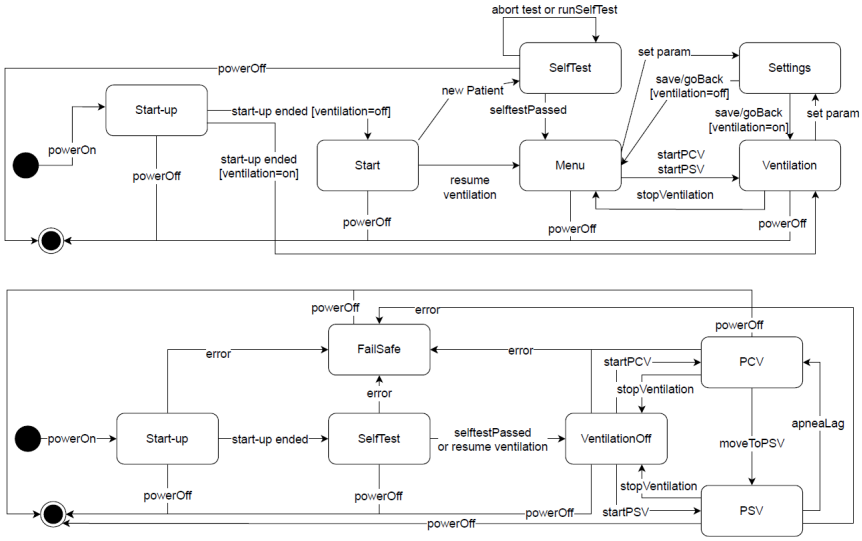

Fig. 1. GUI/Controller state machines taken from [2]

The present paper introduces an Event-B formal model of the MLV system built incrementally thanks to the refinement concept of the Event-B method. The refinement technique permits to master the complexity of a system by gradually introducing its components and characteristics. The obtained model has been validated by simulating some scenarios using ProB [4] and proved under Rodin [3] that provides us with automatic and interactive provers like AtelierB provers, SMT, etc.

The rest of this paper is structured as follows. A brief description the Event-B method is provided in the next section, then Section 3 presents our modelling strategy. Section 4 describes our model in more details. The validation and verification of our model are discussed in Section 5. Finally, Section 6 concludes the paper.

2 EVENT-B Method

Introduced by J-R.Abrial, the formal EVENT-B method [1] is a mathematical concepts-based approach to build correct-by-design discrete systems. An EVENT-B model is made up of components, each of which can either be a *context* or a *machine*. A context models the static part of a system and may define constants and sets (user-defined types) together with axioms that specify their properties. A context is seen by machines that model the dynamic part in terms of variables and a number of events. The type of these variables and the properties that must be satisfied whatever the evolution of the system are specified as invariants using first-order logic and arithmetic.

To master the complexity of a system, EVENT-B defines a refinement technique that allows for an incremental development. Machines are related by a refinement relation (**refines**) whereas the contexts are linked by an extension link (**extends**). By refinement, new variables, events and properties are introduced along with guard strengthening and nondeterminism reduction. A new event introduced in a model M', which refines a model M, is considered to refine a *skip* event of M. Therefore, this new event cannot modify a variable of M. As a result, any event that needs to modify a variable v must be defined in the same model where v is first introduced.

The EVENT-B models presented in this paper have been built, validated and proved within the Rodin platform [3] that provides editors, provers and plugins for various tasks like animation and model checking with PROB [4].

3 Modelling Strategy

3.1 Control Abstraction

Through the formal modeling of the different ABZ case studies [9,6,7,8] we have carried out, the use of the concepts described by Parnas and Madey in [10] has proved to be very suitable for the modeling of control systems. This is why we propose to reuse the same paradigm to build the EVENT-B formal models for the MLV.

The MLV system can be considered as a control system that uses sensors to acquire information from the environment elements m, called *monitored* variables (*e.g.*, state/value of the valves but also the different orders sent by the user like starting/stopping ventilation, etc), and provides these measures to the controller as an *input* variable i. Depending on the information received by the sensors, the controller sends commands via actuators. The objective of these commands, called *output* variable o, is to modify the value of some characteristics of the environment, called a *controlled* variable c. In this particular case study, the controller also sends commands to the GUI to inform the user about the state of the system. These different elements (environment and controller elements) are modelled as variables in EVENT-B. In this paper, we do not model the delays of the sensors/actuators. In other words, we consider such delays are insignificant compared to other actions.

A control system can be viewed from two different perspectives: a control loop that acquires all inputs at once, at a given moment, and then computes all output commands in the same iteration. But, it can be also viewed as a continuous system that can be interrupted by any change in the environment represented by a new value sent by a sensor. In this paper, we adopt the second view where the controller/GUI reacts to each modification of the environment element. From the Event-B point of the view, we model each of these modifications as an Event-B event.

3.2 Modeling Structure

The Event-B specification of the MLV is composed of 6 levels (6 machines and 7 contexts) built iteratively using refinement. Before elaborating this final structure, we have evaluated several alternatives which differ on the following main points:

1. which component, GUI or the controller, should be modelled at first?
2. which level is the best to introduce time?
3. can the controller event error be dealt with like other events?
4. can the events common to the GUI and the controller (powerOn, powerOff, start-up ended, etc.) be represented by a single Event-B event?

In the following, we give and justify the choices made for each point:

1. As depicted in Figure 1, even if the first step of the controller's behavior is independent from that of the GUI, we model the GUI at first because the main functionalities of the controller depend on the orders received from the GUI. Indeed, some events of the controller's state machine are those received from the GUI (startPCV, stopVentilation, etc.).
2. To master the complexity of Event-B models, a common practice is to introduce the time aspect as late as possible. For this particular case study, this was not possible because when the controller is on backup battery, its state may change (become Off) when the battery level is null. Thus, we introduce the time aspect starting from the first level by defining the event progress that models time progression.
3. Contrary to the other events, the controller can at any moment raise an error and move to a safe state where no action is possible anymore. As an error may be due to several causes (valve failure, backup battery failure, etc.), we chose to represent this event as the refinement of the event progress that can detect at any time malfunctions in the system.
4. Even if two transitions of the GUI and the controller states machines are labelled with the same event start-up ended, this event does not have the same effect (actions to execute) on the GUI and the controller. Indeed, the GUI may crash and require to be re-initiated from the initial state, but the behaviour of the controller is not affected at all. This is why we defined two different events, startUpEndedGui for the GUI and startUpEndedCont for the controller.

Given the above, the first level (context GuiStates + machine GuiSM) of the Event-B specification models the different states and the transitions of the GUI along with the backup/external batteries of the system. Mainly, each transition of the GUI's state machine is translated into an Event-B event. In particular, we modelled all the transitions ongoing/outgoing to/from the state Ventilation with a unique event changeMode that is refined later into several events, that is, one event per transition. At this level, we introduce the time progression by specifying that this event can make the system move into the state Off when specific conditions are fulfilled. In the second level, we model the state machine of the controller (context ContStates + machine ContSM). The third level (contexts {ComParams, PCVParams, PSVParams} + machine Ventilation) models the two ventilation modes (PSV and PCV) along with their associated parameters. The next level (context VentilStates + machine VentilationPhases) details the different phases of a respiratory cycle (*inspiratory*, *expiratory*, *inspiratory/expiratory pauses*, etc.). In the last level (context Alarms + machine MVLWithAlarms), we model some alarms that can be raised either by the GUI or the controller.

3.3 Considered requirements

The requirements document [2] is very large and contains a huge number of requirements that are classified according to the related elements (GUI, controller, valves, alarms, etc). Some of them are even reported in several sections since they depend on several elements. As it will be difficult to enumerate all the properties that have been considered, we give hereafter the main functionalities we have considered for the development of the Event-B model:

- the different states of the controller and the GUI as depicted in Figure 1: we have considered all the states and also all the possible transitions between them,
- the different ventilation modes (PCV and PSV) and the switching from one to the other,
- the ventilation parameters and their update before and during the ventilation,
- the position of the valves (*in* and *out*) during the ventilation and their failures,
- Alarms related to the following failures:
  - the valves (*in* and *out*),
  - patient connection while the system is in the state `StartUp`,
  - ventilation parameters values that can be outside the allowed values,
  - the backup battery, the switchover, the FI1/FI12/oxygen sensors.

4 Model Details

In this section, we give some excerpts of the Event-B specification of the MLV system. The complete archive of the Event-B project is available in [5]. Except Section that describes the first abstract level and the first refinement, each subsection will describe a specific refinement level.

4.1 Machines GuiSM + ConSM

Machines GuiSM and ConSM model the state transition machines of the GUI and the controller. For the GUI for instance, we create the context GuiStates that defines a set of all its possible states *ModesG* and a constant *possTransG* that represents the allowed transitions between these states. Even if the state machine of the GUI considers a super state `Ventilation`, we split it to two sub-states `PCV` and `PSV` (see Figure 1).

axm1: **partition**(*Ventilation*, {*PCV*}, {*PSV*})
axm2: **partition**(*ModesG*, {`StartUp`}, {*Start*}, {`Menu`}, {`SelfTest`}, {`Settings`},`Ventilation`, {`Off`})
axm3: $possTransG \in \mathbf{BOOL} \rightarrow \mathbb{P}(ModesG \times ModesG)$
axm4: $possTransG = \{\mathbf{TRUE} \mapsto \{Menu \mapsto Settings, Settings \mapsto Menu, \ldots\} \cup (\{Menu, Settings\} \times Ventilation) \cup \ldots \cup (Ventilation \times (\{Menu, Settings\} \cup Ventilation)), \mathbf{FALSE} \mapsto (ModesG \times \{Off\})\}$

The set $possTransG(\mathbf{TRUE})$ (resp. $possTransG(\mathbf{FALSE})$) denotes the allowed transitions when the power is on (the user pushes power button on the ventilator unit), the backup battery or the external power (`AC`) did not fail and the GUI did not crash. This property, deduced from the state machine of the GUI, is specified in the machine GuiSM by the following invariant:

$$modeGP \neq modeG \Rightarrow modeGP \mapsto modeG \in possTransG(\mathbf{bool}(power = \mathbf{TRUE} \wedge crashed = \mathbf{FALSE} \wedge (onAC = \mathbf{TRUE} \vee (switchover = \mathbf{TRUE} \wedge batLev > 0 \wedge batFail = \mathbf{FALSE}))))$$

where the variables are defined as follows:

- *modeG*(resp. *modeGP*) denotes the current (resp. previous) state of the GUI,
- *power*: states whether the power is on or not, that is, whether the user pushes power button on the ventilator unit or not.
- *crashed*: states whether the GUI is crashed or not,
- *OnAC*: states whether the system is powered using the external power `AC` or not,
- *switchover* (resp. *batFail*): denotes the state of the switchover (resp. backup battery),
- *batLev*: denotes the level of the backup battery.

In the machine GuiSM, we define, among others, the event saveBackAbort that corresponds to the transition labelled save/goBack of the GUI state machine (see Figure 1). This event is used to store or abort the parameters update performed by the user in the state `Settings`. As we can remark, this transition has one source state Settings but two possible target states `Menu` and `Ventilation`. In fact, the GUI has to come back to its previous state when the user asks for the parameters setting. In the machine GuiSM, this event is specified as follows where *modeg* is an event parameter that denotes the new state of the GUI after the execution of the event:

Event saveBackAbort $\widehat{=}$
any
$modeg$
where
grd1: $modeG = \texttt{Settings} \wedge modeg \in ModesG$
grd2: $modeg \in Ventilation \cup \{\texttt{Menu}\}$
then
act1: $modeG := modeg$
act2: $modeGP := modeG$
end

The guard grd1 states that this event is enabled when the GUI is in the state `Settings`, This event makes the GUI move to a new state *modeg* that may be *Ventilation* or *Menu* (Guard grd2). Indeed, from the state `Settings`, the system should come back to its previous state. This state can be deduced from the state of the controller: if the controller was ventilating, *modeg* is equal to *Ventilation*, otherwise it is equal to *Menu*. This is modelled by refining the event saveBackAbort in the machine ContSM and adding the following guards with *modeC* and *modec* representing respectively the current and the next state of the controller:

grd1: $modeC \neq FailSafe$
grd2: $modeC \in Ventilation \Rightarrow modec \in Ventilation \wedge modeg = modec$
grd3: $modeC \notin Ventilation \Rightarrow modec = modeC \wedge modeg = Menu$

In order to save/abort the ventilation parameters, the controller should not be in the state *FailSafe* (Guard grd1). After saving/aborting the parameters update, we distinguish the following cases:

- if the controller is ventilating, it will still ventilating and the GUI will be in the same state as the controller. This means that before being in the state `Settings`, the GUI was in the state `Ventilation` and the user asks for a parameters update during ventilation: the transition setParam from the state `Ventilation` to the state `Settings`. Let us remark that after saving/aborting parameters setting, even if the controller is ventilating, it may change the ventilating mode, moving from PCV to PSV, when asked by the user. This is why the guard grd2 does not state that the current/next states *modeC* and *modec* are equal.
- if the controller is not ventilating (state *VentilationOff*, the controller will stay in the same state and the GUI will move (comes back more precisely) to the state *Menu*. This means that before being in the state `Settings`, the GUI was in the state `Menu` and triggered the transition setParam from the state `Menu` to the state `Settings`

Machines GuiSM and ContSM also specify a generic event changeMode that permits switching between modes `Settings` and `Ventilation` for the GUI and between different ventilation modes (PCV and PSV) for the controller. This event is specified as follows (parts in bold are those added by refinement in the machine ContSM):

Event changeMode $\widehat{=}$
refines changeMode
  any
    modeg, *modec*
  where
    grd1: $modeG \in$ `Ventilation`
    grd2: $modeg \in$ `Ventilation` $\cup$ {`Settings`}
    grd3: $\boldsymbol{modeC} \in$ `Ventilation` $\wedge$ $\boldsymbol{modec} \in$ `Ventilation`
  then
    act1: $modeG := modeg$
    act2: $modeGP := modeG$
    act3: $\boldsymbol{modeC := modec}$
    act4: $\boldsymbol{modeCP := modeC}$
  end

From the GUI point of view, the guards grd1 and grd2 state that when the MLV is ventilating, it is possible either to move to state `Settings` in order to modify the ventilating parameters or continue to ventilate. Guard grd3 specifies that this event can be enabled when the controller is ventilating. This event makes the controller continue the ventilation ($modec \in$ `Ventilation`) but possibly by changing its ventilation mode.

Finally, we have the event *progress* that models time progression with the possibility of modifying the state of both the GUI and the controller. The main parts of this event are as follows:

Event progress $\widehat{=}$
refines progress
  any
    step, *modec*, l, *batf*, ...
  where
    grd1: $step \in \mathbb{N}1 \wedge l \in \mathbb{N}1 \wedge batf \in \mathbf{BOOL}$
    grd2: $modec \in$ {`FailSafe`, $modeC$, `Off`, `StartUp`}
    grd3: $(l = 0 \vee batf{=}\mathbf{TRUE} \vee switchover{=}\mathbf{FALSE}) \wedge onAC = \mathbf{FALSE}$
          $\Rightarrow$
          $modec{=}$`Off`
    grd4: $(batLev > 0 \wedge l > 0) \vee switchover = \mathbf{FALSE} \vee onAC = \mathbf{TRUE} \vee power = \mathbf{FALSE}$
          $\Rightarrow$
          $modec \in \{modeC,$ `FailSafe`$\}$
      ...
  then
    act1: $curTime := curTime + step$
    act2: $batLev := l$
    act3: $batFail := batf$
    act4: $modeC := modec$
      ...
  end

Event progress makes time progress by *step* units of time. It stores the new level of the backup battery (l) and its status (*batf* is true if the backup battery fails). The state of the controller may change as stated by guards grd3 and grd4: if

the system is on the backup battery whose level is null or it has a defect, the controller moves to state `Off`. In the conditions specified by the guard grd4, either the controller does not change its states or goes to the state *failSafe* (see Section 4.5).

4.2 Machine Ventilation

In this machine, we mainly model the ventilation parameters and the switch between the PCV to PSV modes. As the parameters of the GUI and those of the controller may be different before saving the GUI parameters, we have defined a separate set of variables for each. The PSV parameters for instance are modelled by the following invariants:

inv1: $psvParamsValC \in 0..curTime \pfun (psvParams \pfun \mathbb{N}_1)$
inv2: $\forall\, x.\ x \in \mathbf{ran}(psvParamsValC) \Rightarrow psvParams \setminus \{\texttt{RRAP}, \texttt{PinspAP}\} \subseteq \mathbf{dom}(x)$
inv3: $modeC = \text{PSV}$
$\Rightarrow$
$\mathbf{dom}(psvParamsValC(\mathbf{max}(\mathbf{dom}(psvParamsValC)))) = psvParams$

Invariant inv1 gives the type of the variable *psvParamsValC* that represents the PSV parameters stored in the controller. As one can notice, we use partial function since these parameters may have no values at given moments (when the controller/GUI crashes for instance). Invariant inv2 states that the parameters that should always have values. Finally, invariant inv3 specifies that all the PSV parameters should be valued when the PSV ventilation mode is selected. A similar variable *psvParamsValG* is defined for the GUI. In this machine, we also model the action of the user that wants to change the ventilation mode from PCV to PSV. Thus, we have introduced a Boolean variable *PCV2PSV* with the following invariant that states that when the user asks for moving from PCV to PSV, the possibility of modifying the parameters is given for the user ($modeG = \texttt{Settings}$); in that case the controller is either ventilating or in the state `FailSafe`. We have deduced this invariant from the state machines of both the GUI and the controller.

inv1: $PCV2PSV = \mathbf{TRUE}$
$\Rightarrow$
$modeG = \texttt{Settings} \land modeC \in \{\texttt{PCV}, \texttt{FailSafe}\}$

In the machine Ventilation, the event changeMode is refined by the event moveToPSV as follows: we state in the guards that the event is enabled when both the GUI and the controller are in the state PCV, then the GUI moves into the state `Settings` while the controller stays in the same state.

Event moveToPSV $\widehat{=}$
refines changeMode
any
modeg, *modec*
where

grd1: ...
grd2: ...
grd3: ...
grd4: $modeG = \texttt{PCV} \wedge modeC = \texttt{PCV}$
grd5: $modeg = \texttt{Settings} \wedge modec = modeC$

then

act1: ...
act2: ...
act3: ...
act4: ...
act5: $PCV2PSV := \mathbf{TRUE}$

end

Similarly, the event saveBackAbort is refined by distinguishing the cases where the parameters are saved or not. Mainly, we introduce a new Boolean event parameter *sv* with the following semantics: if *sv* is true, the controller parameters become equal to those of the GUI, otherwise (the user aborts the parameters update) the GUI parameters become equal to their previous values that are those of the controller. Moreover, if the variable *PCV2PSV* is true, the controller should move to the PSV mode. The refinement of the event saveBackAbort is as follows where $\mathbf{max}(\mathbf{dom}(psvParamsValG))$ is used to denote the moment of the last update of the parameters:

Event saveBackAbort $\widehat{=}$
refines saveBackAbort

any

$modeg$, $modec$, sv, $psvC$, $psvG$,...

where

grd1: ...
...
grd6: $PCV2PSV = \mathbf{TRUE} \wedge modeC = \text{PCV} \Rightarrow modec{=}\text{PSV}$
grd7: $PCV2PSV = \mathbf{FALSE} \vee modeC \neq \text{PCV} \Rightarrow modec{=}modeC$
grd8: $psvG{=}\{\mathbf{TRUE} \mapsto psvParamsValG(\mathbf{max}(\mathbf{dom}(psvParamsValG))),$
$\mathbf{FALSE} \mapsto psvParamsValC(\mathbf{max}(\mathbf{dom}(psvParamsValC)))\}(sv)$
grd9: $psvC{=}\{\mathbf{TRUE} \mapsto psvParamsValG(\mathbf{max}(\mathbf{dom}(psvParamsValG))),$
$\mathbf{FALSE} \mapsto psvParamsValC(\mathbf{max}(\mathbf{dom}(psvParamsValC)))\}(sv)$
grd10: $modec = \text{PSV} \Rightarrow \mathbf{dom}(psvC){=}psvParams$

then

act1: ...
...
act5: $psvParamsValG(curTime) := psvG$
act6: $psvParamsValC(curTime) := psvC$
act7: $PCV2PSV := \mathbf{FALSE}$

end

4.3 Machine VentilationPhase

In this level, we detail the different breathing phases: *inspiration*, *inspiration pause*, *expiration*, *expiration pause*, etc. To this end, we define a variable *cycles* that denotes the set of the breathing cycles. A new breathing cycle is created

for each new inspiration; its current phase and mode are stored in the variables *cycleMode* and *ventilPhase* defined as follows:

axm1: **partition**(*ventSates*,{`inspBeg`},{`inspEnd`},{`expBeg`},{`expEnd`},...)
inv1: $cycleMode \in cycles \rightarrow Ventilation$
inv2: $ventilPhase \in cycles \rightarrow ventSates$
inv3: $inspBegT \in cycles \rightarrow \mathbb{N}$ inv4: $inspEndT \in cycles \rightarrow \mathbb{N}$

One can wonder why we have to store the mode of each cycle. We do that because the breathing mode may change during any breathing cycle c, but the characteristics of this cycle should not be modified. Let us consider for instance the requirement **CONT.22** for the mode PCV: *The cycle starts with the inspiration phase that lasts an inspiratory time* $I = 60 \times IE_{PCV} \div ((RR_{PCV} * (1 + IE_{PCV})))$. A naive modeling of this requirement would be (RR_{PCV} denotes the Respiratory Rate, IE_{PCV} is the ratio of inspiratory time to expiratory time):

$$\begin{array}{l}\forall\ c.\ c \in cycles \wedge modeC = \text{PCV} \Rightarrow \\ inspEndT(c) \text{ - } inspBegT(c) = 10 * 60 \times (pcvParamsValC(curTime))(IE_{PCV}) \div \\ \qquad ((pcvParamsValC(curTime))(RR_{PCV}) * \\ \qquad\qquad (1 + (pcvParamsValC(curTime))(IE_{PCV}))) \end{array} \quad \textbf{(InspDur)}$$

where *inspBegT* (resp. *inspEndT*) gives the start (resp. end) time of a cycle. This modeling is inadequate because during a breathing cycle both values of variables *pcvParamsValC* and *modeC* may change. This is why we need to define an expression that depends on the values of the parameters taken at the beginning of the inspiration phase. So, we propose instead the following invariant where t is used to denote the values of the parameters at the moment where the breathing cycle c starts. As one can remark, this invariant uses values $pcvParamsValC(t)$ and *cycleMode* that never change even if the PCV parameters and ventilation mode are updated.

$$\begin{array}{l}\forall\ c, t.\ c \in cycles \wedge t = \mathbf{max}(\{x \mid x \in \mathbf{dom}(pcvParamsValC) \wedge x \leq inspBegT(c)\}) \wedge \\ \qquad cycleMode(c) = \text{PCV} \\ \qquad\qquad \Rightarrow \\ inspEndT(c) \text{ - } inspBegT(c) = 10 \times 60 * (pcvParamsValC(t))(IE_{PCV}) \div \\ \qquad ((pcvParamsValC(t))(RR_{PCV}) * \\ \qquad\qquad (1 + (pcvParamsValC(t))(IE_{PCV})))\end{array}$$

In the machine VentilationPhase we define two events (Start and End) for each ventilation phase. Below, we give the specification of the event inspStart that represents the beginning of a new inspiration. Basically, the event start by creating a new cycle *cy* for which a mode (*modeC*), an inspiration phase (`inspBeg`), a beginning (`curTime`) and end (*inspT*) times are assigned. This event can be enabled when the system is ventilating (Guard grd1), all others cycles reach their end states (grd2: **ran**(*ventilPhase*) $\subseteq$ {`expEnd`,`expPauseEnd`}). In that case, the end inspiration time is calculated according to the guard grd3 with the last values of *pcvParamsValC* before the inspiration time ($t = \mathbf{max}(\mathbf{dom}(pcvParamsValC))$). Let us remark that we cannot replace t with *curTime* since *pcvParamsValC* is

not updated continuously but only at some moments. Thus, *pcvParamsValC* can be not valued at *curTime*, this is why we take the value of the last update of this variable.

Event inspStart $\widehat{=}$
any
cy, *inspT*, *t*
where
grd1: $modeC \in \texttt{Ventilation}\ cy \in Cycles \setminus cycles$
grd2: $t = \mathbf{max}(\mathbf{dom}(pcvParamsValC)) \wedge$
$\mathbf{ran}(ventilPhase) \subseteq \{\texttt{expEnd},\texttt{expPauseEnd}\}$
grd3: $modeC = \text{PCV}$
$\Rightarrow$
$inspT = curTime + 10 \times 60 \times (pcvParamsValC(t))(I : E_{PCV}) \div$
$((pcvParamsValC(t))(RR_{PCV}) \times$
$(1 + (pcvParamsValC(t))$
then
act1: $cycles := cycles \cup \{cy\}$
act2: $cycleMode(cy) := modeC$
act3: $ventilPhase(cy) := \texttt{inspBeg}$
act4: $inspBegT(cy) := curTime$
act5: $inspEndT(cy) := inspT$
end

4.4 Machine Valves

In the machine Valves, we model the state/position of the valves during the different breathing phases. We define two Boolean variables *inValve* and *outValve* for *in* and *out* valves respectively: **TRUE** (resp. **FALSE**) for an open (resp.closed) valve. Moreover, we define two Boolean variables *inValveF*/*outValveF* to model the failure of these valves. For each requirement on the position of a given valve during a breathing phase, we define a particular invariant. For instance, the requirement stating that the *in* (resp. *out*) valve is open (resp. closed) during the inspiration is modelled by the following invariant:

inv1: $(\exists\, c.\ c \in cycles \wedge ventilPhase(c) \in \{\texttt{inspBeg}, \texttt{inspEnd}\}) \wedge inValveF = \mathbf{FALSE} \wedge modeC = \texttt{Ventilation} \Rightarrow inValve = \mathbf{TRUE}$
inv2: $(\exists\, c.\ c \in cycles \wedge ventilPhase(c) \in \{\texttt{inspBeg}, \texttt{inspEnd}\}) \wedge outValveF = \mathbf{FALSE} \wedge modeC = \texttt{Ventilation} \Rightarrow outValve = \mathbf{FALSE}$

To maintain these invariants, each event representing the beginning/end of a breathing phase is refined by adding adequate actions that open/close a valve if it is not defective. Event inspStart is refined by adding the following actions:

$inValve := \{\mathbf{FALSE} \mapsto \mathbf{TRUE}, \mathbf{TRUE} \mapsto inValve\}(inValveF)$
$outValve := \{\mathbf{FALSE} \mapsto \mathbf{FALSE}, \mathbf{TRUE} \mapsto outValve\}(outValveF)$

4.5 Machine MVLAlarms

Machine MVLAlarms models the different undesirable situations that may occur on the system. The case study specifies a huge number of such undesirable states. Roughly speaking, when an undesirable situation happens, the controller/GUI emits an alarm and bring the system in a safe state by closing the *in* valve and opening the *out* one. At the time of the submission of this paper, we have mainly modelled alarms related to the valves, the backup battery failure, GUI failure and Controller failure. To this end, we define in a new context the set of all the possible failures we consider as follows:

axm1: $\mathbf{partition}(Alarms, \{\texttt{guiFailure}\}, \{\texttt{contFailure}\}, \{\texttt{inValveFailure}\}, \{\texttt{patConnected}\}, \ldots)$

Then, in the machine MVLAlarms, we introduce a new varaible *alarmRaised* as a Boolean total function that indicates whether an given alarm is raised or not: $alarmRaised \in Alarms \rightarrow \mathbf{BOOL}$. For each type of alarm, we specify the conditions under which it must be raised. For the *in* valve for instance, we specify that the `inValveFailure` must be raised in the following two conditions:

1. the *in* valve is closed while there is a cycle c in one of the following breathing phase: *inspiration* or *recruitment maneuver*,
2. the *in* valve is open while there is a cycle c in one of the following breathing phase: *expiration* or *inspiratory/expiratory pause*.

These two conditions are modelled using the following invariant:

$$
\begin{array}{l}
alarmRaised(\texttt{inValveFailure}) = \mathbf{bool}(\exists\, c.\ (c \in cycles \wedge \\
\quad ((((ventilPhase(c) = \texttt{inspBeg} \wedge curTime > inspBegT(c)) \vee \\
\qquad (ventilPhase(c) = \texttt{rmBeg} \wedge curTime > rmBegT(c))) \wedge \\
\qquad\qquad inValveP = \mathbf{FALSE} \\
\quad)) \vee \\
\quad (((ventilPhase(c) = \texttt{expPauseBeg} \wedge curTime > expPauseBegT(c)) \vee \\
\quad (ventilPhase(c) = \texttt{expBeg} \wedge curTime > expBegT(c)) \vee \\
\quad (ventilPhase(c) = \texttt{inspPauseBeg} \wedge curTime > inspPauseBegT(c))) \wedge \\
\qquad\qquad inValveP = \mathbf{TRUE}))))
\end{array}
$$

Let us remark the use of a new variable *inValveP* introduced in order to express this dynamic property. It is used to store the previous state of the *in* valve. Indeed, when the controller detects a defect on a valve it forces it to move into a safe position. Therefore, it not possible to express this property using the variable *inValve* as its position changes when an alarm is raised on it. To maintain this invariant, the event progress is refined by adding the following guards: grd1 defines an event parameter to verify if the conditions to raise the *in* valve alarm are fulfilled. If so, the second guard grd2 puts the controller into the state *FailSafe*.

$$
\begin{array}{l}
\text{grd1: } alarmInV = \mathbf{bool}(\exists\, c.\ (c \in cycles \wedge (\\
\qquad (ventilPhase(c) \in \{inspBeg, inspEnd, rmBeg, rmEnd\} \wedge inValve = \mathbf{FALSE})
\end{array}
$$

$$\vee$$
$$(ventilPhase(c) \in \{expPauseBeg, inspPauseBeg, inspPauseEnd\} \wedge inValve = \textbf{TRUE}))))$$

grd2: $alarmInV = \textbf{TRUE} \Rightarrow modec = \texttt{FailSafe}$

We also update the variables *alarmRaised* and *inValveP* in the event progress by assigning true to the alarms to raise. The variable is updated as follows:

$$alarmRaised := alarmRaised \lhd \{..., inValveFailure \mapsto alarmInV\}$$
$$inValveP := inValve$$

5 Validation and Verification

The verification and validation of the built Event-B specification have been achieved in three steps detailed hereafter. It is worth noting that these steps are performed in an iterative manner to detect bugs as soon as possible.

5.1 Model Checking of the Specification

For complex systems as the one presented in this paper, model-checking the specification is not only useful for verifying the preservation of the invariant but this task also helps us during the development of the models, that is, finding the adequate invariants/guards/actions. Indeed, when the invariant depends on several variables that are modified by the same event, determining the right actions/guards is sometimes difficult. Basically, we proceed as follows. We write an initial specification for each event, then we use the ProB model checker to ensure that its guards/actions are sufficient for preservation of the invariant. Proceeding like this also avoids providing guards that are too strong. When the invariant is violated, ProB displays a trace (sequence of events), along with the values of each variable, that starting from the initial state of the machine, leads to a state that violates the related invariant. Analysing such a trace allows us to tweak the specification by revising the guards/actions of events but also sometimes the invariants itself that may be too strong. For this particular case study, the use of ProB helps us find the invariants corresponding to the duration of each breathing phase (**InspDur**). Indeed, as stated before (see Section 4.2), this invariant must not depend on the current values of the PCV/PSV parameters but on the last values just before the breathing cycle starts. The counterexample in that case is as follows: (1) a breathing cycle starts with a duration fulfilling the invariant **InspDur**; (2) the user updates the parameters; (3) the inspiration phase terminates with the previous value of the parameters making the invariant **InspDur** violated.

5.2 Proof of the Specification

The absence of invariant violation during the model-checking does not ensure that the specification is consistent. Indeed, ProB works with a timeout that

may prevent us from finding complex scenarios with more events. Therefore, we need to proceed with the proof of the specification in order to verify that each event does preserve the invariant and that the guard of each refined event is stronger than that of the abstract one. These proof obligations are automatically generated by RODIN: 1322 proof obligations have been generated, of which 23% (312) were automatically proved by the various provers. The interactive proof of the remaining proof obligations took about three weeks since they are more complex and require several inference steps and need the use of external provers (like the Mono Lemma prover, *Dis-prove* with PROB and STM provers). During an interactive proof, users ask the internal prover to follow specific steps to discharge a proof obligation. A proof step consists in applying a deductive rule, adding a new hypothesis that is in turn proved or calling external provers. The external Mono Lemma prover has been very useful for arithmetic formulas. The more complex proofs for this particular case study have been those related to the breathing phase duration since we have to distinguish several cases depending on the ventilation mode.

5.3 Validation with Scenarios

In this step, we ensure that the built models does represent/prevent the desired/undesirable behaviours, that is, we have built the right models that behave as expected. Unfortunately, the requirements document does not contain enough scenarios that can be used as oracles during this task. Indeed, the provided scenarios are only related to some very basic behaviors, mainly some transitions of the state machines of the GUI and the controller. Therefore we have defined our own scenarios based on our understanding of the system. According to the state machines of the GUI and the controller, we have defined the scenarios that permit covering the different paths. Using PROB, we have validated the following functionalities of the system:

- The system permits to proceed with the ventilation of patient when no error is detected by the controller and the GUI never crashes. This corresponds to the *nominal behaviour*.
- The controller continues to work even if the GUI crashes. When the GUI re-initialises, its next state is `Ventilation` if the controller is ventilating, `Start` otherwise.
- It is possible to update the ventilation parameters during ventilation, and this does not affect the current cycle if any.
- In case of error, the controller moves into the state `FailSafe` and no action is possible anymore.
- The position of the valves are appropriate for each breathing phase and any failure makes the controller move into the state `FailSafe` with the valves in an adequate position.

6 Conclusion

In this paper, we presented an Event-B formal model of the Mechanical Lung Ventilator (MLV), the case study provided in the ABZ'2024 conference. Our specification takes most requirements and functionalities into account even if some alarms are not modelled. The main difficulty of this case study in the fact that some data can be modified during the ventilation phase while ensuring that the current breathing cycle continues with the previous values. From a practical point of view, this is reasonable since the cycle duration is very small compared to that of the parameters update. However as we do not model the update duration, we got a counterexample when the parameters are updated during the breathing cycle.

Compared to the previous ABZ case studies [7,9], the present case study is time-dependent as the state of the GUI/controller may change by time progression when the system is powered using the backup battery. This is why we introduced time from the first specification level along with the event progress that makes the time evolve. Moreover, ProB fails to find counterexamples on the last 3 refinement levels. This is due probably to the high number of invariants/events to check.

We think that the requirements document should be improved on several points. First, the document is not clear on the conditions under which the controller must move to the state `FailSafe`. For instance, apart from emitting an alert in case of some undesirable situations, the document does not clearly specify for which kinds of alarms the system has to move to the state `FailSafe`. Second, one can wonder whether the controller continuously checks the presence of undesirable events or not. The state machine of the controller specifies that errors may happen in any state, but it is not clear which ones can happen in each specific state. For instance, does the controller continuously checks the communication with the GUI or only in the state `StatUp`?

As future work, we plan to model more alarms to cover more error cases. We think that is not difficult since the remaining alarms are related to values of sensors that are independent of each other. So, we just have to add a variable that monitors the sensor value and raise an alarm if the read value is not in the range of the desired ones. Future improvements also include exploring the use of decomposition plugins available in Rodin for decomposing the models into smaller and thus more manageable units.

References

1. Abrial, J.: Modeling in Event-B. Cambridge University Press (2010)
2. Bonfanti, S., Gargantini, A.: The Mechanical Lung Ventilator Case Study. In: Rigorous State-Based Methods 10th International Conference, ABZ 2024, Bergamo, Italy, June 25–28, 2024, Proceedings, Lecture Notes in Computer Science, vol. 14759. Springer (2024)
3. Consortium, E.B.: `http://www.event-b.org/`

4. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From Animation to Data Validation: The ProB Constraint Solver 10 Years On. In: Boulanger, J.L. (ed.) Formal Methods Applied to Complex Systems: Implementation of the B Method, chap. 14, pp. 427–446. Wiley ISTE, Hoboken, NJ (2014)
5. Mammar, A.: An Event-B Model of Mechanical Lung Ventilator. Available at `https://github.com/AmelMammar/MechanicalLungVentilator` (February 2024)
6. Mammar, A., Frappier, M., Fotso, S.J.T., Laleau, R.: An Event-B Model of the Hybrid ERTMS/ETCS Level 3 Standard. `http://info.usherbrooke.ca/mfrappier/abz2018-ERTMS-Case-Study` (February 2018)
7. Mammar, A., Frappier, M., Laleau, R.: An Event-B Model of an Automotive Adaptive Exterior Light System. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods - 7th International Conference, ABZ 2020, Ulm, Germany, May 27-29, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12071, pp. 351–366. Springer (2020)
8. Mammar, A., Leuschel, M.: Modeling and Verifying an Arrival Manager Using Event-B. In: Glässer, U., Campos, J.C., Méry, D., Palanque, P.A. (eds.) Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, May 30 - June 2, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14010, pp. 321–339. Springer (2023)
9. Mammar, A., Laleau, R.: Modeling a Landing Gear System in Event-B. STTT (2015)
10. Parnas, D.L., Madey, J.: Functional Documents for Computer Systems. Science of Computer Programming **25**(1), 41–61 (1995)