Modeling a Landing Gear System in Event-B

Amel Mammar[1] and Régine Laleau[2]

[1] Institut Mines-Télécom/Télécom SudParis, CNRS UMR 5157 SAMOVAR, France
amel.mammar@telecom-sudparis.eu
[2] Université Paris-Est, LACL, IUT Sénart Fontainebleau, France
laleau@u-pec.fr

Abstract. This paper describes the Event-B modeling of the landing gear system of an aircraft whose the complete description can be found in [3]. This real-life case study has been proposed by the ABZ'2014 track that takes place in Toulouse, the European capital of the aeronautic industry. Our modeling is based on the Parnas and Madey's 4-Variable Model that permits to consider the different parts of a system. These parts are incrementally introduced using the Event-B refinement technique. The entire development has been carried out under the Rodin toolset. To validate and prove the different components, we use the Atelier B, SMT and ML provers which are plugged to Rodin.

1 General Overview of the System

The objective of the landing gear system is to permit a safe extension/retraction of the gears when the plane is going to land/fly. Each gear is placed in a landing-gear box equipped with a door that must be open when a gear is extending/retracting and closed when it becomes completely extended/retracted and locked. To this aim, the controller (See Figure 1) reads, periodically through a set of sensors, the states of the different elements (doors, gears, handler, etc.) and sends orders to a set of electro-valves that make, for instance, the gears extend/retract or the doors open/close. More details will be introduced throughout the modeling of this system.

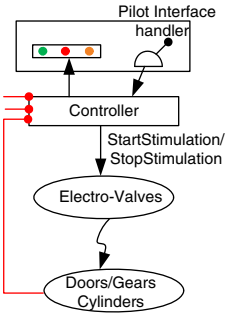

Fig. 1. The overall structure of the landing gear system

F. Boniol et al. (Eds.): ABZ 2014 Case Study Track, CCIS 433, pp. 80–94, 2014.

To model this system in Event-B [2], we suggest following the classification of modeling variables according to the four-variable model of Parnas and Madey [8]. We distinguish two groups of variables *environment* and *controller* variables:

1. *Environment variables*: represent the status of the elements outside the controller. Two kinds of variables are distinguished:
   - *Monitored variables*: the values of these variables are not calculated by the controller but can be monitored. For example, the actual states of the doors/gears.
   - *Controlled variables*: the values of these variables are determined by the controller. For example, the status of the valves and the lights.
2. *Controller variables*: denote values inside the controller system. Mainly, they represent the values of some elements as seen by the controller but also the different orders it sends.
   - *Inputs*: the values stored in the controller and provided by some sensors. For example, $door\_open_i[x], handle_i, gear\_extended_i[x]$, etc.
   - *Outputs*: they are the orders sent by the controller toward the different environmental elements. For example, $general\_EV, retract\_EV$, $gears\_maneuvring$, $anomaly$, etc.

The system can be seen as continuously executing the following sequence of actions:

Do
Read Inputs from some sensors
Process Inputs
Produce outputs
Until *a failure is detected*

In the following sections, we are going to develop the modeling of this system in six main steps:

1. *Modeling the monitored variables*: we describe the behavior of the physical components like the doors, the gears, cylinders, but also the handler, the switch and the shock absorbers. The variables modeling these components are suffixed with "$\_p$" because they represent their actual (physical) status (See Section 3).
2. *Modeling the controlled variables*: we describe in this phase the behavior of the valves that permit to act directly on the doors, gears and cylinders (See Section 4). We also describe the behavior of the lights that inform the pilot about the status of the system in general. Again, the variables modeling these components are suffixed with "$\_p$" because they represent their actual status (See Section 4).
3. *Modeling the controller/output variables*: we describe how the controller reads information from the sensors, sends orders to the valves and how it updates the values of the lights (See Section 5).
4. *Modeling timing aspects*: to facilitate the design, we have chosen to elaborate a first modeling of the system without considering any timed constraints. The timed aspects are taken into account later by refinement (See Section 6).

5. *Modeling the failure cases*: in this step, we take into account the system's anomalies caused by failures on the different elements of the system (See Section 7).
6. Finally, we describe how properties are verified (See Section 8).

In each of the previous steps, the different elements are gradually introduced thanks to the Event-B refinement mechanism. The next section gives a brief description of the Event-B method together with its refinement technique.

2 Event-B Method

Event-B [2] is the successor of the B method [1] permitting to model discrete systems using mathematical notations. The complexity of a system is mastered thanks to the refinement concept that allows to gradually introduce the different parts that constitute the system starting from an abstract model to a more concrete one. An Event-B specification is made of two elements: *context* and *machine*. A context describes the static part of an Event-B specification; it consists of constants and sets (user-defined types) together with axioms that specify their properties. The dynamic part of an Event-B specification is included in a machine that defines variables and events. The possible values that the variables hold are restricted using an invariant written using a first-order predicate on the state variables. An event can be executed if it is enabled, i.e. all the conditions, named guards, prior to its execution hold. Among all enabled events, only one is executed. In this case, substitutions, called actions, are applied over variables. The execution of each event should maintain the invariant. To this aim, proof obligations are generated for each event. To discharge these proof obligations, the Rodin[1] platform offers an automatic prover but also the possibility to plug additional external provers like the SMT and Atelier B provers.

3 Modeling the Monitored Variables

In the system, we have the following monitored variables: *gears, doors, doors/gears cylinders, handler, hydraulic circuit* and *switch*. These elements are introduced according to the following refinement strategy:

- Initial model (Component *Gears*): we start by describing the behavior of the gears, that can be made extended or not, since this is the main objective of the system.
- 1st refinement (Component *GearsIntermediateStates*): we refine the state where a gear is not extended by distinguishing two different sub-states: retracted or partly-extended.
- 2nd and 3rd refinements (Components *Doors* and *DoorsIntermediateStates*): like for the gears, we describe the state of a door as open or not, then we add an intermediate state to model a partly-open door.

[1] `http://www.event-b.org/install.html`

- 4th refinement (Component *Cylinders*): in this step, we introduce the cylinders that allow the motion of the doors and gears.
- 5th refinement (Component *HandlerSwitchShockAbsorber*): we model in this phase the handler, the analogical switch, the hydraulic circuit and the shock absorbers.

In the following sub-sections, we detail each step.

3.1 Gears Modeling: The Initial Model and the First Refinement

We first introduce a context with set `PositionsDG` representing the three possible cases for gears/doors/etc.: *front*, *left* or *right*. Then, we define a Boolean variable *gear_extended_p* to formalize whether a gear is extended or not:

inv1: $gear\_extended\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$

To make the gears extended or not, we define the following two events:

Make_GearExtended	Start_GearRetracting
ANY *po* **WHERE**	**ANY** *po* **WHERE**
$po \in PositionsDG \wedge$	$po \in PositionsDG \wedge$
$gear\_extended\_p(po)$=FALSE	$gear\_extended\_p(po)$=TRUE
THEN	**THEN**
$gear\_extended\_p(po)$:=TRUE	$gear\_extended\_p(po)$:=FALSE
END	**END**

When a gear is not extended, it can be retracted or partly-extended. So, we refine the previous specification by introducing a new Boolean variable *gear_retracted_p* that is true if the gear is entirely retracted. This variable is defined by two invariants (**inv2**) and (**inv3**), where (**inv3**) states that a gear cannot be extended and retracted at the same time:

inv2: $gear\_retracted\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$
inv3: $\forall po.(po \in \texttt{PositionsDG} \Rightarrow \neg(gear\_extended\_p(po) = \texttt{TRUE} \wedge gear\_retracted\_p(po) = \texttt{TRUE}))$

Consequently, the event `Make_GearExtended` is refined by adding the guard ($gear\_retracted\_p(po)$ =FALSE), and we define the two following new events to make a gear start extending (it becomes no longer retracted) or complete its closing.

Start_GearExtending	Make_GearRetracted
ANY *po* **WHERE**	**ANY** *po* **WHERE**
$po \in$ PositionsDG	$po \in$ PositionsDG
$gear\_retracted\_p(po)$ =TRUE	$gear\_extended\_p(po)$ =FALSE
THEN	$gear\_retracted\_p(po)$=FALSE
$gear\_retracted\_p(po)$:=FALSE	**THEN**
END	$gear\_retracted\_p(po)$:=TRUE
	END

3.2 Doors Modeling: The Second and Third Refinements

In this part, we present the modeling of the doors. To this aim, we have proceeded like for the gears by defining two levels. In the first level, we define a new variable *door_open_p* to know if a door is open or not. Then, we refine, in the second level, the state where a door is not open by adding a new variable *door_closed_p* to state if the door is closed or partly-open.

– the third refinement: we define the variable *door_open_p* and express an invariant to state that when a gear is partly-extended then all the doors are open. In other words, it is not possible to start the extending/retracting of a gear until all the doors are open.

inv4: $door\_open\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$
inv5: $\exists po.(po \in \texttt{PositionsDG} \wedge gear\_extended\_p(po) = \texttt{FALSE} \wedge gear\_retracted\_p(po) = \texttt{FALSE}) \Rightarrow door\_open\_p = \texttt{PositionsDG} \times \{\texttt{TRUE}\}$

In order to preserve invariant (**inv5**), we refine the events `Start_GearExtending` and `Start_GearRetracting` by adding the guard ($door\_open\_p$ =PositionsDG× {TRUE}). We also define two events to make a door open and start closing.

Make_DoorOpen	Start_DoorClosing
ANY *po* **WHERE**	**ANY** *po* **WHERE**
$door\_open\_p(po) = \texttt{FALSE}$	$door\_open\_p(po) = \texttt{TRUE}$
THEN	$(gear\_extended\_p = \texttt{PositionsDG} \times \{\texttt{TRUE}\} \vee$
$door\_open\_p(po) := \texttt{TRUE}$	$gear\_retracted\_p = \texttt{PositionsDG} \times \{\texttt{TRUE}\})$
END	**THEN**
	$door\_open\_p(po) := \texttt{FALSE}$
	END

– the fourth refinement: in this level, we define the variable *door_closed_p* and express that a door cannot be open and closed at the same time:

inv6: $door\_closed\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$
inv7: $\forall po.(po \in \texttt{PositionsDG} \Rightarrow \neg(door\_open\_p(po) = \texttt{TRUE} \wedge door\_closed\_p(po) = \texttt{TRUE})$

In order to preserve invariant (**inv7**), we refine the event `Make_DoorOpen` by adding the guard ($door\_closed\_p(po)$ = FALSE). We also define two new events to make a door start opening (it becomes no longer closed) or accomplish its closing.

Start_DoorOpening	Make_DoorClosed
ANY *po* **WHERE**	**ANY** *po* **WHERE**
$door\_closed\_p(po) = \texttt{TRUE}$	$door\_closed\_p(po) = \texttt{FALSE}$
THEN	$door\_open\_p(po) = \texttt{FALSE}$
$door\_closed\_p(po) := \texttt{FALSE}$	**THEN**
END	$door\_closed\_p(po) := \texttt{TRUE}$
	END

3.3 Cylinders Modeling: The Fourth Refinement

The motion of the gears and the doors is performed by a set of cylinders. A door (resp. gear) cylinder is locked when the door is closed (resp. extended or retracted). Of course, before starting moving, the cylinder, associated with the door/gear, should not be locked. So in the next refinement, we define two new variables *door_cylinder_locked_p* and *gear_cylinder_locked_p* with the following invariant:

inv8: $door\_cylinder\_locked\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$
inv9: $gear\_cylinder\_locked\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$
inv10: $\forall po.(door\_cylinder\_locked\_p(po) = \texttt{TRUE} \Rightarrow door\_closed\_p(po) = \texttt{TRUE})$
inv11: $\forall po.(gear\_cylinder\_locked\_p(po) = \texttt{TRUE} \Rightarrow (gear\_extended\_p(po) = \texttt{TRUE} \vee gear\_retracted\_p(po) = \texttt{TRUE}))$
inv12: $\forall po.(gear\_cylinder\_locked\_p(po) = \texttt{FALSE} \Rightarrow door\_open\_p = \texttt{PositionsDG} \times \{\texttt{TRUE}\})$

In order to satisfy (**inv11**), we have refined the events *Start_GearExtending* and *Start_GearRetracting* by adding the guard ($gear\_cylinder\_locked\_p(po) =$ FALSE). Similarly, we have refined the event *Start_DoorClosing* by adding the guard ($gear_cylinder_locked_p =$PositionsDG $\times$ {TRUE}) to make (**inv12**) satisfied. Finally, we have defined four new events to lock/unlock door/gear cylinders. For the sake of space, we provide only those associated with gears.

UnlockGearCylinder	LockGearCylinder
ANY *po* **WHERE**	**ANY** *po* **WHERE**
$po \in$ PositionsDG	$po \in$ PositionsDG
$gear_cylinder_locked_p(po) =$TRUE	$gear_cylinder_locked_p(po)=$FALSE
$gear_extended_p(po) =$TRUE $\vee$	$gear_extended_p(po) =$TRUE $\vee$
$gear_retracted_p(po)=$TRUE	$gear_retracted_p(po) =$TRUE
$door_open_p =$PositionsDG $\times$ {TRUE}	**THEN**
THEN	$gear_cylinder_locked_p(po) :=$TRUE
$gear_cylinder_locked_p(po) :=$FALSE	**END**
END	

3.4 Handler/Switch/Shock Absorbers/Hydraulic Circuit Modeling: The Fifth Refinement

In this step, we continue the modeling of the monitored variables by introducing the handler, the analogical switch, the shock absorbers and the hydraulic circuit. First, we extend the context by defining two new sets PositionsHandler and PositionsSwitch to denote respectively the possible positions for the handler, *up* and *down*, and for the switch, *open*, *closed*. So, we define two Boolean variables *handler_p* and *analogical_switch_p* to model the position of the handler and the switch respectively. Since the analogical switch closes each time the handler changes its position, we add a Boolean variable *handle* which memorizes the handler shift. The events we define for the handler are: PutHandlerUp and PutHandlerDown. For instance, under the guard $handler\_p = down$, the event PutHandlerUp sets the variable *handler_p* to *up* and assigns TRUE to the variable *handle*.

To model the physical behavior of the analogical switch depicted in Figure 2, we define two additional Boolean variables $Intermediate_1$ and $Intermediate_2$ that cannot be true at the same time as follows:

inv13: $\neg(Intermediate_1 =$TRUE $\wedge\ Intermediate_2 =$TRUE$)$
inv14: $(Intermediate_1 =$TRUE $\vee\ Intermediate_2 =$TRUE$) \Rightarrow analogical_switch_p = open$

Each transition is translated into an event whose guard corresponds to its source state and includes (*handle*=TRUE) if it is triggered by the handler shift. The action of this event consists in assigning FALSE to the source state and TRUE to the target one. For the sake of space, we only provide the Event-B translation of two transitions.

close_Switch	HandleFromIntermediate2ToIntermediate1
WHEN	**WHEN**
$Intermediate_1 =$TRUE	$Intermediate_2 =$TRUE
THEN	$handle =$TRUE
$analogical\_switch\_p := closed$	**THEN**
$Intermediate_1 :=$FALSE	$handle :=$FALSE
END	$Intermediate_2 :=$FALSE
	$Intermediate_1 :=$TRUE
	END

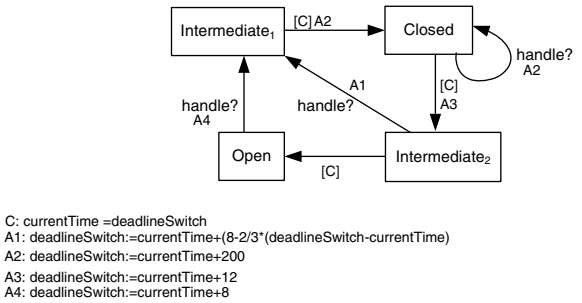

Fig. 2. Physical behavior of the analogical switch

The hydraulic circuit is modeled with a Boolean variable *circuit_pressurized_p*, and two events *Unpressurise* and *Pressurise*. For instance, under the guard (*circuit_pressurized_p*=TRUE) , the event *Unpressurise* sets the variable *circuit_ pressurized_p* to FALSE. In addition, we have refined each event related to the doors/gears/ motion and lock/unlock cylinders by adding a guard (*circuit_pressu rized_p* = TRUE). Finally, we model gears shock absorbers by a Boolean variable that gives for each position the state of its associated shock absorber according to the following invariant stating that a gear shock absorber is on ground only if its gear is extended:

inv15: $\forall po.(po \in \texttt{PositionsDG} \wedge gear\_shock\_absorber\_p(po) = \texttt{TRUE} \Rightarrow gear\_extended\_p(po) = \texttt{TRUE})$

So, to preserve invariant (**inv15**), the event Start_GearRetracting is refined by adding an action that set the variable *gear_shock_absorber_p*(*po*) to FALSE. In addition, to make the state of a shock absorber evolve, we have defined two new events: a first to set it to FALSE and a second to TRUE under the guard that its gear is extended.

4 Modeling the Controlled Variables: The Sixth Refinement

This section deals with the modeling of valves and lights that are controlled by the system (Component *ValvesLights*). We describe how a valve becomes active/not active and how a light becomes on/off. Each valve is modeled with a Boolean variable (*general_EV_p*, *open_EV_p*, *close_EV_p*, *extend_EV_p*, *retract _EV_p*) and two events to make it active or not. For the sake of space, we describe the events that activate the open door valve and deactivate the extend valve; the others are very similar.

MakeOpenDoorValveActive	MakeExtendValveActive
WHEN	**WHEN**
open_EV_p= FALSE	*extend_EV_p*= FALSE
circuit_pressurized_p= TRUE	*circuit_pressurized_p*= TRUE
THEN	**THEN**
open_EV_p:= TRUE	*extend_EV_p*:= TRUE
END	**END**

In addition, we refine each event related to the motion of doors/gears by adding a guard to specify that the corresponding valve is active and its opposite is deactivated. For instance, we refine the event `Start_GearExtending` by adding the guard ($extend\_EV\_p$ = `TRUE` $\wedge$ $retract\_EV\_p$ = `FALSE`). We also refine the events related to lock/unlock door/gear cylinders by adding the adequate guard. For instance, we refine the event `LockGearCylinder` by adding the guard:

$$(gear\_extended\_p(po) = \texttt{TRUE} \wedge extend\_EV\_p = \texttt{TRUE}) \vee$$
$$(gear\_retracted\_p(po) = \texttt{TRUE} \wedge retract\_EV\_p = \texttt{TRUE})$$

Finally, we refine the event `Pressurise_HydrolicCircuit` (resp. `Unpressurise_HydrolicCircuit`) by adding the guard ($general\_EV\_p$ = `TRUE`) (resp. $general\_EV\_p$ = `FALSE`).

The lights are dealt with similarly to the valves. We model each of them by a Boolean variable ($greenLight\_p$, $orangeLight\_p$, $redLight\_p$) and define two events for green and orange lights; one to set the light on and the other to set it off. For the red light, only the event that makes it on is defined since this state is kept forever.

5 Modeling the Controller/Output Variables: The Seventh Refinement

In this section, we describe how the controller takes its decisions about the setting of the light and the activation/deactivation of the valves according to the information it gets from the sensors that it periodically reads (Component *Sensor*). To do that, the controller reads the status of the handler, the switch, the hydraulic circuit, the doors and the gears[2]. So, we introduce for each of these elements a new variable that represents its state as seen by the controller. Such variables are suffixed by "$\_ind$" and are of the same type and have the same constraints as their associated variables suffixed by "$\_p$". For instance, a door cannot be seen open and closed at the same time. The controller acquires information from the sensors as follows:

```
ReadInput
  ANY
    handler_sensor_value , analogical_switch_sensor_value, circuit_pressurized_sensor_value,
    gear_extended_sensor_valueF, gear_extended_sensor_valueL, gear_extended_sensor_valueR,
          ...
  WHERE
    handler_sensor_value ∈ PositionsHandler
                ...
    gear_extended_sensor_valueF ∈ BOOL ∧
          gear_extended_sensor_valueF=TRUE ⇒
        gear_extended(front)=TRUE ∧ gear_cylinder_locked_p(front)=TRUE
                ...
  THEN
    handler_sensor_ind:=handler_sensor_value
                ...
    gear_extended_sensor_p:= {front ↦ gear_extended_sensor_valueF,
        left ↦ gear_extended_sensor_valueL,right ↦ gear_extended_sensor_valueR}
                ...
  END
```

[2] In this paper, we make the assumption that there is a unique sensor on each of these elements.

The key point of the event `ReadInput` is that each sensor does not give information that goes against the security of the system (the sensors are intrinsically safe), that means that if it says that a door/gear is {open, close}/{extended/retracted} then it is really the case. If the sensor is faulty, it should say: I do not know!, that is, it will return `FALSE` for the doors and the gears. From these inputs, the controller takes decisions about sending orders to the valves. Each order to a valve is modeled by a Boolean variable (*general_EV*, *close_EV*, etc.) such that:

inv16: $\neg(open\_EV{=}\texttt{TRUE} \wedge close\_EV{=}\texttt{TRUE})$//Req R_{41}
inv17: $\neg(extend\_EV{=}\texttt{TRUE} \wedge retract\_EV{=}\texttt{TRUE})$//Req R_{42}
inv18: $(open\_EV = \texttt{TRUE} \vee close\_EV = \texttt{TRUE}) \Rightarrow general\_EV = \texttt{TRUE}$
inv19: $(extend\_EV = \texttt{TRUE} \vee retract\_EV = \texttt{TRUE}) \Rightarrow open\_EV = \texttt{TRUE}//inv18 + inv19 = R_{51}$

For instance, the controller sends orders to the general and extend valves as follows:

- when the analogical switch is closed, it sends a start stimulation to the general valve if it reads that the handler is `up` (resp. `down`) but the gears are not locked up (resp. down). It should also maintain the stimulation of the general valve if the open/close valve is still stimulated. The event that model starting/stopping the stimulation of the general valve is as follows:

```
OutputGeneralValve
  ANY general_EV_value WHERE
    general_EV_value =bool((analogical_switch_ind =closed
        ∧ ((handler_ind=down ∧ gear_extended ≠PositionsDG × {TRUE}) ∨
      (handler_ind=up ∧ gear_retracted_ind ≠ PositionsDG × {TRUE}))) ∨
        (open_EV =TRUE ∨ close_EV =TRUE)
    general_EV ≠ general_EV_value THEN
    general_EV := general_EV_value
  END
```

- if the open door valve is stimulated and the doors are seen open, it sends a stimulation order to the extend valve if it sees that the handler is down but one of the gear is not extended and locked in the down position, otherwise it stops it:

```
OutputExtendGearValve
  ANY extend_EV_value WHERE
    extend_EV_value =bool(handler_ind =down ∧
    gear_extended_ind ≠ PositionsDG×{TRUE} ∧
    open_EV =TRUE ∧ retract_EV =FALSE ∧
    door_open_ind=PositionsDG×{TRUE}) ∧
    extend_EV ≠ extend_EV_value THEN
    extend_EV := extend_EV_value
  END
```

Similarly to the valves, the controller sends orders to the lights. At this level, we only introduce the order to the green and orange lights; the red one is achieved later when we model failures. For instance, when the controller sees the gears extended and locked, it sends order *gears_locked_down* as follows:

```
gears_locked_down :=bool(gear_extended_sensor_valueF =TRUE ∧
                         gear_extended_sensor_valueL =TRUE ∧
                         gear_extended_sensor_valueR =TRUE)
```

In this step, we refine each event related to making a valve active/not active by adding a guard to specify that its related order has been sent from the controller. We also refine the event acting on the lights by adding a guard that expresses that the setting order has been received from the controller.

6 Introducing Timing Aspects: The Eighth Refinement

In this system, timing aspects are four folds: (1) the analogical switch takes time to move from open to close and vice versa (2) the start/stop stimulation of valves should be separated by some time, (3) the valves take time to be active, the cylinders take time to move and be locked/unlocked, (4) the controller has to read some inputs at given moments to be sure that the system behaves correctly as expected or not. In this section, we deal with the first three aspects (Component *TimedAspects*) and postpone the last one to the next section. Let us notice that real-time cannot be explicitly modeled in Event-B, thus we approximate it by using discrete time: a natural variable $currentTime$ represents the current time.

6.1 Timing Constraints on the Analogical Switch

To introduce timing constraints on the switch, we add a natural variable *deadline Switch* that represents the deadline at which the switch changes its state according to Figure 2. To move from a state to another, $currentTime$ should be equal to $deadlineSwitch$, then the deadline is updated adequately. For instance, the event `HandleFromIntermediate2ToIntermediate1` is refined by adding the action $deadlineSwitch := currentTime + (8 - (2/3) \times (deadlineSwitch - currentTime))$[3]. Similarly, the event `close_Switch` is refined by adding the guard $currentTime = deadlineSwitch$ and action $deadlineSwitchcurrentTime + 200$.

6.2 Timing Constraints on the Start/Stop Stimulation of Valves

In this system, the time between starting the stimulation of the general valve and the others should be separated by at least 2 *u.t* (units of time). Since the open valve is the first to stimulate just after the general valve, it is sufficient to respect this time between them. Similarly, the time between stopping the stimulation of the general valve and the others should be separated by at least 10 *u.t*. Since the open/close valve is the last valve that stops stimulation just before stopping the general valve, it is sufficient to respect this time only between them. In addition, the stimulation of contrary orders should be separated by at least 1 *u.t*. So, we have defined five natural variables $allowedStopGeneralEv$, $allowedStartOpenEV$, $allowedStartCloseOpenEV$, $allowedStartExtedEV$ and $allowedStartRetractEV$ that are updated as follows: when the general (resp. open, close, extend, retract) valve is stimulated, then the variables $allowedStartOpenEV$ and $allowedCloseOpenEV$ (resp. $allowedCloseOpenEV$, $allowedStartOpenEV$, $allowedStartRetractEV$, $allowedStartExtedEV$) are updated with the adequate value. In addition, when open (resp. close) valve is stopped then the variable $allowedStopGeneralEv$ is also updated with the adequate value. So, the event

[3] Since, the type *Real* is not provided in Event-B, all computations are done in fixed-point arithmetic with a scale of 10.

OutputGeneralValve has been refined by adding the guard ($currentTime \geq allowedStopGeneralEv$) and actions:

$$\boxed{\begin{array}{l} allowedStartOpenEv := \{\texttt{FALSE} \mapsto \texttt{TRUE} \mapsto currentTime + 2, \texttt{TRUE} \mapsto \texttt{FALSE} \mapsto 0\} \\ \qquad\qquad (general\_EV \mapsto general\_EV\_value) \end{array}}$$

6.3 Timing Constraints on the Activation of the Valves

A valve takes some time to be active/not active after starting/stopping its stimulation. So, we have associated with each kind of valves (general, door, gear) a natural variable that states the time at which the valve can be active/not active. For instance, we have defined for the extend/retract valves the variable $deadlineStimulationRetractExtendEv$ that is updated to the adequate time when the controller sends an order for these valves, and it is reset to 0 when the valve becomes active/not active. Basically, we have refined the event OutputExtendGearValve by adding the action:

$$\boxed{\begin{array}{l} deadlineStimulationRetractExtendEv := \{\texttt{FALSE} \mapsto \texttt{TRUE} \mapsto currentTime + 10 \\ \qquad \texttt{TRUE} \mapsto \texttt{FALSE} \mapsto currentTime + 36\}(extend\_EV \mapsto extend\_EV\_value) \end{array}}$$

Finally, we have refined the events that make the extend/retract valve active/not active by adding the guard ($currentTime = deadlineStimulationRetractExtendEv$) and action ($deadlineStimulationRetractExtendEv := 0$) to reset the deadline after executing the event.

6.4 Timing Constraints on Cylinders

The gears/door cylinders take some time to lock/unlock but also to move from high to down and vice versa. To consider the time taken by a gear cylinder to lock/unlock, we have defined a variable $deadlineUnlockLockGearsCylinders$. This variable is set by events that make extend and retract valves active in order to launch the deadline for unlocking the cylinders, and the events Make_GearExended and Make_GearRetracted to launch the deadline for locking the cylinders. Similarly, we have defined a variable $deadlineGearsRetractingExtending$ to consider the time tacken by the gear to move from down to up and vice versa. So, the event MakeExtendValveActive is refined by adding the guard ($currentTime = deadlineStimulationRetractExtendEv$) and the two following actions that permit to reset the deadline and to set the moment at which the gear cylinders become unlocked.

$$\boxed{\begin{array}{l} deadlineStimulationRetractExtendEv := 0 \\ deadlineUnlockLockGearsCylinders := PositionsDG \times \{currentTime + 4\} \end{array}}$$

In addition, we refine the event Start_GearExtending by adding the action:

$$\boxed{\begin{array}{l} deadlineGearsRetractingExtending(po) := \\ \qquad \{front \mapsto currentTime + 12, left \mapsto currentTime + 16, right \mapsto 16\}(po) \end{array}}$$

and the guard ($deadlineUnlockLockGearsCylinders(po) = 0$). Finally to make the time progress, we have defined the event passingTime that increases the variable $currentTime$ when there is at least one non-null deadline. The time progresses by an amount *step* without exceeding any non-null deadline in order to avoid the starvation problem. More details can be found at:

http://www-public.it-sudparis.eu/~mammar_a/LandingGearsSystem.html

7 Introducing Failures: The Ninth Refinement

7.1 Modeling Failures

So far, we have considered all the physical elements as working correctly as expected. However in practice, each of them can fail: the switch, the cylinders and the valves can fail at any time. To take such failures into account, we have added for each of these elements an event that makes it fail (Component *Failures*). For example, for the switch and the door cylinders, we have defined the two following events where the variables $analogical\_switch\_fail$ and $door\_cylinder\_fail$ denote Boolean variables that say respectively whether the switch or a door cylinder has failed:

MakeSwitchFail	MakeDoorCylinderFail
WHEN $analogical\_switch\_fail$ =FALSE	**ANY** po **WHERE** $po \in$ PositionsDG **THEN**
$analogical\_switch\_fail$:=TRUE	$door\_cylinder\_fail(po)$:=TRUE
END	**END**

Consequently, we refine each event related to the behavior of the switch, the valves and the cylinders by adding a guard stating that the element change its status only if it has not failed. For instance, we have added the guard ($door\_cylinder\_fail(po)$:=FALSE) for the events Make_DoorOpen, Start_DoorOpening, Start_DoorClosing, etc.

7.2 Detecting Anomalies

As stated in the previous section, physical elements can fail. The controller does not have any information about that but it can deduce it by monitoring the status of the switch, the doors, and the gears. In fact, if the controller sends an order to stimulate the open valve but the doors are not seen open after a given time, then it can assert that a problem has happened (in at least one physical element) by displaying the *anomaly* information to the pilot. To this aim, the controller has to read, through the sensors, the status of these elements at well-defined times. For instance, the controller has to verify that the switch is closed 10 $u.t$ after the handler has changed its position otherwise an anomaly is detected. To model that, we add a natural variable $nextInputReadForOpenSwitch$ that memorizes the time at which the controller must not see the switch open. This variable is updated by the event ReadInput which we refine by adding the following action:

$$nextInputReadForOpenSwitch := \{\texttt{FALSE} \mapsto nextInputReadForOpenSwitch, \texttt{TRUE} \mapsto currentTime{+}10\}(\mathbf{bool}(handler\_ind \neq handler\_sensor\_value))$$

As for other deadlines, to avoid the starvation problem, we refine the event passingTime by adding a guard stating that if $nextInputReadForOpenSwitch$ is not null then the time can progress but without exceeding it. In addition, the event ReadInput resets the variable $nextInputReadForOpenSwitch$ when this deadline is reached and the verification performed. So, we add the following actions to the event ReadInput:

$$nextInputReadForOpenSwitch := \{\texttt{FALSE} \mapsto nextInputReadForOpenSwitch, \texttt{TRUE} \mapsto 0\} \Leftarrow (\mathbf{bool}(currentTime = nextInputReadForOpenSwitch))$$
$$anomaly := \mathbf{bool}(currentTime = nextInputReadForOpenSwitch \wedge analogical\_switch\_ind = open)$$

The other anomalies on the doors, the gears, the hydraulic circuit are dealt with similarly. In addition, we have refined the event `ReadInput` and the events sending orders to the valves by adding the guard ($anomaly = FALSE$) in order to stop the system. Indeed according to the description of the system, the anomaly message has to be maintained forever. From a modeling point of view, we introduce a deadlock such that no operation becomes possible.

8 Properties Verification: The Tenth Refinement

Most properties to verify are temporal properties that refer to several moments of the system. A model checker like ProB [7] would be very useful for such purpose. Nevertheless, we have chosen to stay in a same framework of proof by modeling them as invariants (Component *PropertyVerification*). Moreover to distinguish the specification of the system from the verification of properties, we have created a new refinement level that defines such properties as invariants. For the sake of space, this paper illustrates the verification of the properties through one example, the verification of the other properties can be found at:

`http://www-public.it-sudparis.eu/~mammar_a/LandingGearsSystem.html`

R_{74}. If one of the three gears is not seen locked in the down position more than 10 seconds after stimulating the outgoing electro-valve, then the Boolean output normal mode is set to false.

To specify this property, we have defined a new variable *TimeStimulationExtendRetractEv* to memorize the time at which the extend/retract valve is stimulated. This variable is set by the event `OutputExtendGearValve` by adding the action ($TimeStimulationExtendRetractEv := currentTime$). Then, the property is specified as follows:

$$(currentTime > TimeStimulationExtendRetractEv + 100 \wedge extend\_EV = TRUE) \Rightarrow (anomaly = TRUE \vee gear\_extended\_ind = PositionsDG \times \{\texttt{TRUE}\})$$

To discharge this invariant, the following intermediate lemmas have been added:

$$(currentTime > TimeStimulationExtendRetractEv + 100 \wedge extend\_EV = \texttt{TRUE}) \Rightarrow nextInputReadForGearEndExtendingRetracting = 0$$

$$(nextInputReadForGearEndExtendingRetracting = 0 \wedge extend\_EV = TRUE) \Rightarrow (anomaly = \texttt{TRUE} \vee gear\_extended\_ind = \texttt{PositionsDG} \times \{\texttt{TRUE}\})$$

The first invariant ensures that the time does not progress beyond the deadline ($TimeStimulationExtendRetractEv + 100$) without reading the state of the gears since the variable *nextInputReadForGearEndExtendingRetracting* is reset when the gears are read. The second one states that the controller sets the variables *anomaly* and *gear_extended_ind* correctly when the deadline is reached. Table 1 gives the results of the verification activities.

Table 1. Verification results

Requirement	Verified?	Method	Comment
R_{11}	✓	Animation	Proof seems to be too hard since it needs several intermediate lemmas.
R_{12}	✓	Animation	Proof seems to be too hard since it needs several intermediate lemmas.
R_{21}	✓	Proof	It is verified from the instant where the controller sees the position of the handler down
R_{22}	✓	Proof	It is verified from the instant where the controller sees the position of the handler up
R_{31}	✓	Proof	It is not valid on the physical elements since the controller can start extending/retracting the gears when the doors are actually open but the close valve does not stop completely. Thus, we express it according to the internal variables.
R_{32}	✓	Proof	It is not valid on the physical elements since the controller can start opening/closing the doors when the gears are actually extended/ retracted but the extend/retract valve does not stop completely. Thus, we express it according to the internal variables.
R_{41}, R_{42}, R_{51}	✓	Proof	
R_{61}, R_{62}, R_{63}, R_{64}	✓	Proof	
R_{71}, R_{72}, R_{73}, R_{74}	✓	Proof	

9 Conclusion: Limits and Future Work

In this paper, we have presented a modeling of a landing gear system in the formal language Event-B. To this aim, we have proceeded into 3 main phases: (1) modeling the system without timed concerns and possible failures; (2) taking timed concerns into account; (3) considering the possible faults on the different elements of the system. From a design point of view, the main difficulty was to define a method to tackle the complexity of the case study. The combination of the four-variable model of Parnas and of the Event-B refinement process has proved very relevant for this type of problem. The former allows to classify the variables that represent the system and its environment and the latter allows to gradually introduced these variables. This approach has been used by Butler [4] but with a very simple case study. Contrary to Butler's work, we have chosen to consider time constraints later in the design, since it seemed to us simpler for the proof activity. Finally, failures have been introduced at the end of the process following the idea of considering first the nominal system behavior as advised by [6,9]. From a technical point of view, we have defined 66 variables and 48 events split into 10 refinement levels that give rise to 285 proof obligations, 72% of which have been discharged automatically; we have accomplished the remaining proofs interactively thanks to the Atelier B, SMT and ML provers which are Rodin plugins. We think the modeling can be improved if Event-B and the Rodin framework, under which this development has been achieved, offer real-time aspects. In addition, it would be interesting

to deeper study the use of one of the structuring mechanisms proposed for Event-B: decomposition [10] or modularization [5], in order to structure the specification into logical units.

As stated before, regarding the description of the case study, we make the assumption that each sensor is unique and not triplicated. This is not a strong assumption and does not affect the modeling; it can be easily relaxed by only adapting the event `ReadInput`. For the handler for instance, we will define two functions *handler_sensors* and *handler_sensors_valid* to memorize the values of the sensors and its validity:

$$handler\_sensors \in 1..3 \longrightarrow \texttt{BOOL} \wedge handler\_sensors\_valid \in 1..3 \longrightarrow \texttt{BOOL}$$

Then, the event `ReadInput` is updated as follows (value `TRUE` (resp. `FALSE`) represents position `up` (resp. `down`)):

ANY $handler\_ind\_value, handler\_sensor\_valid\_value$ **WHERE**
$handler\_ind\_value =$
$((card(handler\_sensor\_valid^{-1}[\{\texttt{TRUE}\}])=3 \wedge$
$((handler\_sensor(1) =\texttt{TRUE} \wedge (handler\_sensor(2)=\texttt{TRUE} \vee handler\_sensor(3) =\texttt{TRUE})) \vee$
$\vee$ (handler_sensor(2)=TRUE $\wedge$ handler_sensor(3)=TRUE)))
$\vee$ $(card(handler\_sensor\_valid^{-1}[\{\texttt{TRUE}\}])=2 \wedge$
$card(handler\_sensor[handler\_sensor\_valid^{-1}[\{\texttt{TRUE}\}]])=1)))$
$handler\_sensor\_valid\_value$=... **THEN**
$handler\_ind$:=$\{\texttt{TRUE} \mapsto \texttt{up}, \texttt{FALSE} \mapsto \texttt{down}\}(handler\_ind\_value)$
$handler\_sensor\_valid := handler\_sensor\_valid\_value$
...

References

1. Abrial, J.-R.: The B-book, Assigning Programs to Meanings, pp. I–XXXIV, 1–779. Cambridge University Press (2005)
2. Abrial, J.-R.: Modeling in Event-B - System and Software Engineering, pp. I–XXVI, 1–586. Cambridge University Press (2010)
3. Boniol, F., Wiels, V.: The Landing Gear System Case Study. In: Boniol, F. (ed.) ABZ 2014 Case Study Track. CCIS, vol. 433, pp. 1–18. Springer, Heidelberg (2014)
4. Butler, M.: Using Event-B Refinement to Verify a Control Strategy, Working Paper. ECS, University of Southampton (2009)
5. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting reuse in event B development: Modularisation approach. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 174–188. Springer, Heidelberg (2010)
6. Jeffords, R.-D., Heitmeyer, C.-L., Archer, M., Leonard, E.-I.: Model-Based Construction and Verification of Critical Systems using Composition and Partial Refinement. Formal Methods in System Design 37(2-3), 265–294 (2010)
7. Leuschel, M., Butler, M.-J.: ProB: An Automated Analysis Toolset for the B Method. STTT 10(2), 185–203 (2008)
8. Lorge Parnas, D., Madey, J.: Functional Documents for Computer Systems. Sci. Comput. Program. 25(1), 41–61 (1995)
9. Miller, S.-P., Tribble, A.-C.: Extending the Four-Variable Model to Bridge the System-Software Gap. In: Proceedings of the 20th Digital Avionics Systems Conferene (DASC 2001), Daytona Beach, Florida (2001)
10. Silva, R., Pascal, C., Hoang, T.-S., Butler, M.: Decomposition tool for Event-B. Softw., Pract. Exper. 41(2), 199–208 (2011)