Systematic Construction of Critical Embedded Systems using Event-B

Pascal André, Christian Attiogbé and Arnaud Lanoix

LS2N CNRS UMR 6004 - University of Nantes
✉{firstname.lastname}@univ-nantes.fr

Abstract. We propose a method to build critical embedded control systems in a systematic way. The method covers the modelling of both the digital part and the physical environment of a considered system, and their refinement until more concrete levels. It is based on Event-B in order to benefit from its materials, stepwise refinements and tools. Two main processes are distinguished: one to capture the global model, the other to detail the global model; they are made of several refinement steps which are accompanied with guidelines. The precise description of the interface between the digital and physical parts is used to start the modelling process. The recurrent categories of variables and events in control systems are described and used as guidelines to conduct a systematic construction. We illustrate the method with the landing gear system case study.

Key words: Embedded control systems; Specification method; Event-B patterns

1 Introduction

Modelling and analysis of complex systems without dedicated methods is painful, inefficient and time-consuming. Methods and tools are required for efficient system engineering; this is particularly true for formal software engineering.

Unlike many other types of software, embedded systems are often developed for specific target environments (processors, vehicles, medical devices, etc.) and very often they should run for long times (even years), once they have been implemented in their so called critical environments. Therefore, embedded systems and their construction have stringent robustness requirements; one have to develop them accordingly to get them reliable at runtime. The target environments for the development of each embedded system do not help the advent or the expansion of tools and methods dedicated to this type of software. But, there are numerous models for embedded real-time systems [15],

Considering that *i)* the requirements for reliability and correct construction of the models and the derived embedded systems are of great importance, and that *ii)* the development of these systems lacks of methods to guide the developers, we are motivated to contribute to fill the gap between these needs and the state of the art.

In this work we propose a correct-by-construction method dedicated to critical embedded control systems. This method, based on Event-B, is intended to guide step by step the specifier or the engineer to drive its development from requirements to concrete software, defining abstract models, and refining them in a systematic way.

In Section 2 we introduce the proposed method with the details of each step. Section 3 illustrates the application of the method on a common case study, called the landing gear system. In Section 4 we evaluate the application of the method and the case study and comment related studies. Section 5 concludes this work.

2 A Method to Construct Correct Embedded Systems

We present a stepwise and systematic method (named Heñcher) to construct critical embedded control systems using Event-B. We reuse the approach already established and demonstrated in several case studies [9,10,1], following which complex systems can be constructed by combining 1) horizontal refinement with *feature augmentation* where we have to build a global abstract model of a the whole system (a controller and its physical environment) (Sect. 2.1) and 2) *structural refinement* (making the abstract structures more and more concrete (Sect. 2.2). But we extend it and provide dedicated guidelines at different steps, which help the developers to reach quickly a correct control system. Fig. 1 illustrates the Event-B patterns from the most abstract model which describes only the interface of the controller, to the systematic decomposition into two parts: the Controller and the physical Environment.

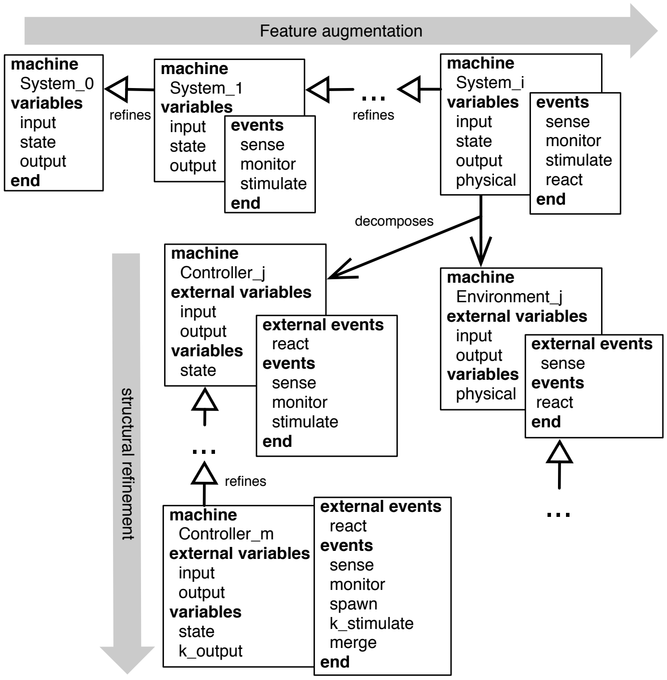

Fig. 1. Synoptic structure of the Event-B models of the construction

2.1 Horizontal Process: Building an Abstract Global Model of the System

The high level state space of any control system can be described by the **elicitation of the interface variables** between the digital part (the controller) and the physical part (the controlled environment) of the considered system.

Step 1: Characterise the abstract model of a considered system

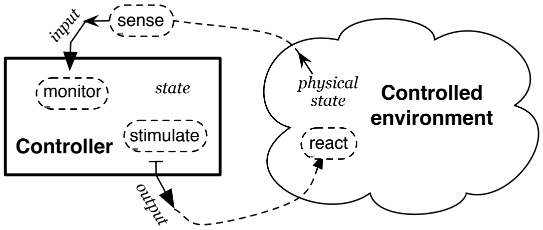

Fig. 2. A generic shape for event-based model of a control system

Fig. 2 depicts a general principle that may govern the organisation of *event-based models* of control systems. The dashed ovals are representative of the parametric events families; They should be replaced by the effective events related to the logic of a specific case study. Besides, the identified physical devices to be controlled should be precisely listed. The behaviour of each one will be specified later.

Step 1.1: Elicit the interface We distinguish in our method, a first step which consists in the *description of the interface variables of the controller*. There are three categories of variables at the interface of a controller with the controlled environment.

- the *input* variables: they give the sensed state of the environment (the value given by sensors); they are read by the controller;
- the *state* variables: they are set and modified by the controller; they are used for monitoring the whole system;
- the order *output* variables: they are those used to send the orders that stimulate the physical environment; their values will be used by actuators.

These categories of variables will be used at different levels of the modelling and refinement of the system at hand: to introduce the events of the first abstract model and, for refining gradually the first model. Additionally, we have the category of *internal* variables, which are only used inside the controller.

Step 1.2: Elicit the global properties of the system The required system properties, including safety, liveness and non-functional properties should be explicitly named and listed in their informal form. These properties will be formalised and gradually introduced with their formal form during the various construction steps of our method.

Step 1.3: Start with a first abstract model Use the interface variables resulting from Step 1.1 to build a first Event-B abstract model. This abstract model comprises in the related B machine clauses, the interface variables together with the appropriate abstract sets and properties which characterise them. This Event-B model will be enriched to obtain a global abstract model of the system including its control part, its physical part and their related properties. Global properties of the system (among those already listed in Step 1.2) should be formalised and introduced according to the available variables. Notice that the enrichment incorporates gradually the details of the physical environment (sensors and actuators) and the corresponding properties.

Step 1.4: List the events of the abstract model The Event-B abstract model resulting from the previous step will be enriched with a series of events built by defining a family of events related to each category of the interface variables: *sense events*, *monitor events* and *stimulate events*.

- **Sense events family** This family gathers the events used to set and to modify the values of the *input* variables. For each variable of this category, define an event named after the variable, with the prefix *sense_*. The link with the physical (state variables of the) environment is done later by refining these events in Step 2.2.
- **Monitor events family** These events modify the appropriate *state* variables. For each variable of this category, define an event named after the variable with the prefix *monitor_*, to set the variable according to the current state of the controller and the input data.
- **Stimulate events family** The events of this family modify the order *output* variables; each variable of this category is set with an event named after the variable with the prefix *stmlt_*. These events use the internal variables, the input variables and the output variables. Associated with these events to stimulate the physical devices, we may have as many events to stop the stimulation of the devices; accordingly these events have their name prefixed with stop_.

These three families of events, together with the **reaction events family** introduced later, are compliant with the standard sense-decision-control of the control cycle.

Step 2: Extend the abstract global model

Use *feature augmentation* [9,10,1] to integrate the controlled environment. This is precisely achieved on the basis of the sense events family, which in turn need the description of the controlled environment. The global properties listed before are also gradually formalised in the model, as invariants, as soon as the appropriate variables are available. Two sub-steps are distinguished but no matter their order during the development.

Step 2.1: Introduce the physical environment and the reaction events family It consists in adding successively to the model, events to propagate the values of interface *output* variables inside the physical environment. These events simulate or stop the behaviours of physical devices via the actuators. The feature augmentation is used to introduce the *physical state* variables, invariants and appropriate behaviours. Depending on the cases, either one simulates the behaviour of the physical devices with an abstract model, or the values of the output variables (from the interface) result in signals sent to the environment. In this last case we do not have dedicated events in the abstract model. Accordingly, the behaviours of the physical devices should be formally described. These behaviours, systematically guarded by the values of the output variables, may impact the state of the environment and finally they may impact the sensors. State automata can often be used to capture the behaviour of a physical device; describing the automata with B events is then straightforward. The description of the physical part behaviour results in the ***family of reaction events***. These events should be named using as prefix the identifier of the physical part that they describe.

Step 2.2: Detail the sense events family Each event of the *sense family*, updates an *input* variable according to the state of the sensors; for this purpose the event needs the model and the behaviour of its related sensor. Therefore the feature augmentation consists in introducing the model of the sensors and their related behaviour, as variables, invariants and related events. The behaviour of the sensors should consider the possible failures (anomalies, malfunctioning or physical defects); specific events should be described for each such possible failure.

Practically during all the refinement steps, it is recommended to proceed incrementally with several small refinement steps dedicated to variables and events. This is necessary to master the proof complexity.

Step 3: Introduce the specific properties

According to the system one has to build, besides the global properties gradually introduced with the variables, additional specific properties should be integrated at the abstract level to constrain the functioning of the system.

1. *Reachability property with partial ordering:* specific events (not at the same granularity with the Event-B events) with timestamps may be systematically used to order and to reason on reachability properties.
2. *Non functional properties:* specific properties related to nonfunctional requirements should be gradually introduced here. No matter the way they are described, provided that the mathematical support of Event-B is very large, and that external modules may be used to analyse them.

2.2 Vertical Process: Building the Concrete Parts of the System

The aim of this second process is to build the digital part and possibly the physical part of the system. The global Event-B abstract model resulting from the horizontal process should then be decomposed into various parts leading to specific components. At least we have a decomposition into a control software and a physical part. The decomposition can be performed as soon as one want to go into the details of one of the specific part by considering that the other part will stay as it is; that means no modification of the other part cannot be considered when we are refining a given one. Typically, from a decomposition step, the digital part will be refined until code by considering the events and variables of the physical part as they was at the decomposition step.

Step 4: Refine the global abstract model

We recommend to perform structural refinements as needed by the specific model to be refined. New internal B events may be added to refine the events of each family of events (*sense*, *monitor*, or *stimulate*). The state space variables of the global abstract model may be refined with more details in the invariant. At the end of this step, be sure that, the events of both parts are all in place, that the global required properties of the system are all in place (they cannot be introduced later after the decomposition).

Like with other formal models, an Event-B model can be animated, *i.e.* when appropriate values are provided for the variables in the model, its behaviour can be observed step by step according to the semantics of the model. Animation capabilities are helpful during all the refinement steps where we still have all the events of the global system together; it will not be possible to animate the whole system after its decomposition.

Step 5: Decompose into software and physical parts

A decomposition paradigm is already supported by the Event-B method. It consists in splitting a given machine into several ones which will be refined independently. The decomposition splits the variables of the state space and/or the behaviour of the machine; however resulting machines cannot contradict each other by modifying the variables and their related properties once they have been separated. Two approaches exist for this purpose: the Abrial'style decomposition (called the A-style decomposition) [2] based on shared variables, and the Butler'style decomposition (called the B-style decomposition) [7,19] based on shared events. In the A-style decomposition, events are first split between Event-B sub-components and then shared variables of the sub-components are used to introduce external events in the sub-components; these external events should be refined in the same way. In the B-style decomposition, variables are first partitioned between the sub-components and then shared events (which use the variables of both sub-components) are split between the sub-components according to the used variables.

We adopt the A-style decomposition which is more relevant when considering a list of specific events to be split relatively to a control part and a physical part. The methodological guide to achieve the decomposition is as follows: the digital part is made with all the events defined in the *sense events*, the *monitor events* and the *stimulate events* families whereas the physical environment gathers all the events defined in the *reaction events* families. Moreover, each part must have an abstract view of the other through external variables and events.

Step 6: Refine the control software and the physical environment separately

Step 6.1: Refining the control software Use *Structural refinements* based on the *stimulate* events family to refine the controller. The involved categories of variables are the *input* variables, the *state* variables and the *output* variables. Typically, the values of the *output* variables are synthesised from the other ones. This can be done through simple control functions or through sub-modules.

When there are sub-modules, the *input* variables may be spawned inside the sub-modules; in the same way *output* variables may be updated by promotion from the sub-modules if any. Therefore one have to incorporate successively in the Event-B model the events to set and modify the *output* variables; they describe the result of the behaviour of the control part. State automata help to catch these behaviours; then the events of the B models encode the automata. We give now some recurrent patterns to help in modelling control part.

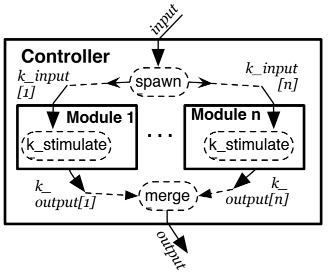

Fig. 3. Modules redundancy

i). *Composition of several redundant sub-modules:* when a controller is made of several redundant modules, it is straightforward to describe a generic module and use an indexing function to compose several instances of such modules (see Fig. 3).
   - *Encasing variables inside modules:* the values coming from outside one or several modules can be systematically encased inside the modules with a dedicated event that spawn the events.
   - *Promoting variables outside a module:* in a symmetric way, the values going outside a module or several modules can be systematically described using a **promotion pattern** (with a dedicated event) for merging the output variables of the internal computing modules.

ii). When the modules are not redundant, each one should be refined separately, but the treatment we have described for the inputs and outputs variables is the same.

Step 6.2: Refining the controlled (physical) environment Many cases can be considered depending on the system to be studied; either the physical devices are already available, or one has to build the physical devices from the formal models, or one has to build a part of the physical devices. Nevertheless, the exchange of signal with actuators is the standard way to act on physical devices.

i). In the case where the physical devices are available, with the related actuators, the refinement is straightforward; it consists for the events of the physical part to output the correct signal values (for example on/off values are encoded as voltage) as input of the actuators. But the physical devices may be emulated in preliminary studies before implementing the control part on the real devices. Mathematical models and dedicated system engineering tools are available as explained hereafter.

ii). When the output variables of the controller cannot be immediately encoded as signal values, transformation can be achieved via appropriate mathematical functions and models. This should be done, starting from the requirements and the properties of the previous model, for example by external modules or functions written with tools like Matlab or Simulink[1] or SciLab[2]; related works can be found in [21]. These tools generate executable codes dedicated to the target devices. They are also equipped with specific functions to handle time requirements.

3 A Running Case Study

The proposed method is applied on the *Landing Gear* (LG) case study, a benchmarking example proposed at the ABZ'2014 conference to compare different formal methods in terms of expressivity, performance, and ease of use. A prerequisite for reading this section is the detailed specification of this critical embedded system given in [5]. A summary of the LG system is depicted in Fig. 4.

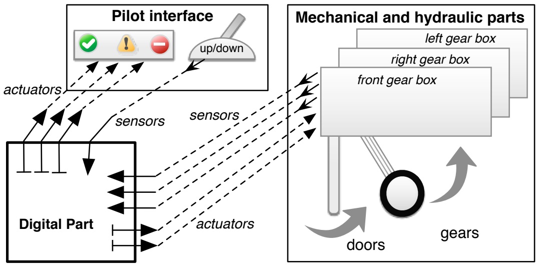

Fig. 4. Global architecture of the LG system

The LG system is in charge of manoeuvring 3 landing boxes: front, left and right. Each landing box contains a landing gear, an associated door and the corresponding hydraulic cylinders in charge to move gears and doors. The system is made of a controller (the *digital part*) and the controlled physical environment (i.e. the 3 landing gear boxes and a pilot interface) which interact via sensors and actuators; the sensors provide to the digital part the information on the state of its physical part; the actuators engage the orders of the controller on the physical part. The physical devices already exist, we will not build them; the challenge deals with the digital control part only (see page 2 of [5]).

[1] http://uk.mathworks.com/products/control/

[2] http://www.scilab.org

3.1 Horizontal Process: Building an Abstract Global Model of the System

We give the main elements resulting from the successive application of the steps proposed in the method (Sect. 2).

Step 1.1: Elicitation and modelling of the interface variables The requirement document listed several triplicated *input* variables: handle, analogical switch, gear states, doors, $\cdots$. We model (**Step 1.3**) them with a type $TRIPLE = \{1,2,3\}$ used as an index of the function variables:

$GEAR = \{FG, LG, RG\}$	$analogical\_switch \in TRIPLE \rightarrow AnalSWSTATE$
$DOOR = \{FD, RD, LD\}$	$handle \in TRIPLE \rightarrow HSTATE$
$HSTATE = \{hDown, hUp\}$	$gear\_extended \in (TRIPLE \times GEAR) \rightarrow BOOL$
$AnalSWSTATE = \{openSW, closedSW\}$	$door\_closed \in (TRIPLE \times DOOR) \rightarrow BOOL$
$\ldots$	$\ldots$

The function variable $handle \in TRIPLE \rightarrow HSTATE$ captures precisely the requirement $handle_i \in \{hDown, hUp\}$ with $i \in \{1,2,3\}$. The *state* variables are the states of the gears, doors, anomalies, etc. They are modelled as follows:

$$gears\_locked\_down \in BOOL \ \wedge \ gears\_maneuvering \in BOOL \ \wedge \ anomaly \in BOOL \wedge \cdots$$

The *output* variables hold the values computed for various electro-valves:

$$general\_EV \in BOOL \ \wedge \ close\_EV \in BOOL \ \wedge \ open\_EV \in BOOL \wedge \cdots$$

The lights which indicate the position of the gears and doors to the pilot are described as *internal* variables: *greenLight*, *orangeLight*, *redLight*. These variables are bound to the output state variable *gears_locked_down* with an invariant predicate. Another *internal* variable *order* is used to record the action of the pilot on the handle.

The LG system is controlled digitally in the *normal* mode until an anomaly is detected. A permanent failure leads to an *emergency* mode where the system is controlled analogically. Accordingly the *internal* boolean variable *anomaly* is used to denote that an anomaly has been detected or not.

Step 1.2: Elicitation of the global properties of the LG system Most of the normal mode requirements are safety properties. Some identified ones are the following:

R_{21}	We can not observe a retraction sequence (consequence of the order hUp) if the handle is down. Using the enumerated set $HSTATE$ which permits only one value from two for the variable *order*.
R_{31}	The gears outgoing event occurs if doors are open locked.
R_{41}	Opening and closing doors electro-valve are not stimulated simultaneously.
R_{51}	It is not possible to stimulate the manoeuvring EV (opening, closure, outgoing or retraction) without stimulating the general EV.

The first Event-B abstract model resulting from **Step 1.3**, gathers all the variables of the interface, their related invariants and initialisations. Event-B contexts are used to model the static part with the various sets and definitions that we have introduced.

Step 1.4: The families of events of the abstract model A thorough analysis of the two action sequences (outgoing sequence and retraction sequence) described in the LG system helps us to capture the behaviour of the digital part and to derive the events. We use here state automata to make it clear the interaction between the different components (actions of the pilot, the controller, the orders received by the environment).

In the *sense event* family we have listed for example the event sense_gear to modify the input variable *gear_extended* listed above. In the same way we have listed the other events sense_door, etc. Examples of events we have identified for the *control events* family are: stmlt_general_EV to stimulate the general electro valve, stmlt_door_opening, stmlt_gear_outgoing, stop_stmlt_general_EV, stop_ stmlt_gear_outgoing, etc.
Each one modifies its related variable, for instance the event stop_stmlt_gear_outgoing sets the variable *extend_EV* to *FALSE*. Examples of event we have classified in the *monitor events* family are: monitor_ anomaly, monitor_gears_locked_Down, monitor_ gears_maneuvering.

Step 2: Extension of the abstract global model with the event families We achieve many refinement steps, by feature augmentation, to integrate gradually the variables and events related to the physical devices: the sensors, the doors and the gears.

Following **Step 2.1**, we define the behaviours of physical devices. For instance, the door behaviour is first captured with a state automata; the transitions of the automata are then described as events. For this purpose we use a transition function $doorState \in DOOR \rightarrow DSTATE$ where $DSTATE = \{ClosedLocked, ClosedUnlocked, OpenUnlocked\}$ is the enumerated set of the identified door states. The set $DOOR$ contains the three door identifiers. The function $doorState$ is a total function; this captures the requirement that all the three doors are controlled via the state transition.

The starting transition of the door behaviour is enabled by the *open_EV* order given by the digital part. Therefore there is a synchronisation between the digital part and the motion of the doors. We only give below the description of the starting event Door_openDoor_cl2cu; the other necessary events are similar.

```
event Door_openDoor_cl2cu
/* Door's Behaviour (for the three doors). The first transition of the Door Automata */
where
    @g1  $open\_EV = TRUE$ // all the doors Electro Valves are on
    @g2  $ran(doorState) = \{notOpenLocked\}$
then
    @a1  $doorState := DOOR \times \{notOpenNotLocked\}$ // door is being opened
end
```

The following event describes an event of the *control event* family.

```
event stmlt_gear_outgoing
/* stimulate gear outgoing electro valve once the three doors are in the open position */
where
    @g0  $general\_EV = TRUE$
    @g1  $order = hDown$
    @g2  $ran(handle) = \{hDown\}$
    @g3  $ran(door\_closed) = \{FALSE\}$ // the three doors are in the open position
    @g4  ran(door_open) = {TRUE}
    @next  nextOGseq = 3
    @gano  anomaly = FALSE // no anomaly detected
    @notretract  retract_EV = FALSE
then
    @a1  extend_EV := TRUE
    @a2  nextOGseq := nextOGseq + sequenceStep
end
```

The variable *nextOGseq* controls the evolution of the outgoing sequence; it indicates in the event guards, the next step in the outgoing sequence. We note that the events in the *sense event* family anticipate their real future specifications, which are related to the physical part introduced later. When we have introduced the various events families and the related variables, it becomes clear for us that we have the complete control loop. Following **Step 2** the properties (listed in **Step 1.2** above) are formalised as first order predicates, integrated into the invariant of the abstract model and, proved along the horizontal refinement. As an example, the requirement R_{51} is described as follows.

$$\begin{array}{l}((open\_EV = TRUE \lor close\_EV = TRUE \lor extend\_EV = TRUE \lor retract\_EV = TRUE) \\ \quad \Rightarrow general\_EV = TRUE)\end{array}$$

To sum up, the global Event-B abstract model results from a series of refinement of contexts and machines.

Step 3: Dealing with specific properties In this case study, reachability is one of the specific properties. Based on the idea of Lamport's logical clocks [16], we implement a technique that captures the reachability requirement R_1 given in page 13 of [5]. For that purpose, we introduce the notion of *control cycle*, a period of time during which one can observe several events, especially a chain of events denoting an outgoing sequence or a retraction sequence; a typical control cycle is one starting with an event (*downH*) which denotes the *hDown* order and terminating by an event (*dcge*) which denotes the fact that "*the gears are locked down and the doors are seen closed*"; similarly, another control cycle is started when the handle triggers an order *hUp*. A dedicated variable *endCycle* is used to control the start and the end of each control cycle.

Given a set *obsEvents* of events (for instance the starting of an outgoing sequence, a door closed, a gear locked in a position, etc.) and a logical clock modelled as a natural number, the occurrences of the events can be ordered by the timestamps given by the clock. In our case two events cannot happen at the same time. We use a partial function $ldate \in obsEvents \rightarrow\!\!\!\!\!+ \mathbb{N}$ to record the timestamps of the events. We can compare and reason on the timestamps of any events happening during a sequence and specifically within the specific event sequence called *control cycle*. An example is as follows.

$$\begin{array}{l}\forall dj.(((dj \in \mathbb{N}) \land (dcge \in dom(ldate)) \land (dj = ldate(dcge)) \\ \qquad \land (endCycle = TRUE) \land dj < llc) \Rightarrow \\ \quad \exists di.((di \in \mathbb{N}) \land (downH \in dom(ldate)) \land (di = ldate(downH)) \land (di < dj) \land \\ \qquad\quad \forall ii.(ii \in \mathbb{N} \land di \leq ii \land ii < dj \Rightarrow ldate \sim [\{ii\}] \neq \{upH\})))\end{array}$$

To put into practice in Event-B with Rodin, we defined the set *obsEvents* in the context of our machines, and the above property is included in the invariants of the abstract model.

3.2 Vertical Process: Building the Concrete Parts of the LG System

The vertical process includes several refinements (in **Step 4**) described below following the proposed method. We end our process with the **Step 5**. The **Step 6** was not performed for the LG case study because only the digital part will be refined with the objective to build the software part. The variables and events which are specific to the behaviour of the physical part are not refined but we keep them in the model in order to preserve *animation capabilities*. This approach is very pragmatic.

Step 4: Structural refinements of the global abstract model The requirement document details the inner structure of the digital part; it is made of two redundant computing modules. We achieve structural refinement steps to overcome the details of the behaviour of the digital part.

a) Introducing the two computing modules with refinements Both modules have the same interface (*input* and *output* variables) inherited from the abstract model of the digital part. Each interface variable of a module k (where $k \in \{1,2\}$) is inherited from a variable (for instance $gear\_extended$) of the digital part of the abstract model and it is denoted by $k\_gear\_extended(k)$ where k is an index. An enumerated set $CompModule = \{1,2\}$ is used for the indexes. Therefore each interface variable of the computing modules is specified with the following shape:

$$k\_gear\_extended \in CompModule \rightarrow ((TRIPLE \times GEAR) \rightarrow BOOL)$$

The binding between the two modules interface variables and those of the abstract module is achieved via refinements where new variables and related events are introduced.

b) Spawning the inputs inside the computing modules with refinements We introduced new events (prefixed with `spawn_`) to push the value of each *input* variable (for example $handle$) at the abstract level, in the corresponding variable (for example k_handle) of each computing module. As the inputs of the modules should be the same, an invariant is defined in each case of variable spawning in order to guarantee the correctness of the binding between the *input* variable of the digital part and the same input of the computing modules. The following event pattern spawns the variables at the interfaces of the computing modules.

```
event spawn_handleDown // spawn handleDown within the k CompModules
where @g1   $ran(handle) = \{hDown\}$
then
    @a1   $k\_handle := \{1 \mapsto (TRIPLE \times (ran(handle))), 2 \mapsto (TRIPLE \times (ran(handle)))\}$
end
```

We have identified a reusable **specification rule**: a new event is introduced along with each new k-indexed variable. This event should copy the variable at high level (the digital part) into the indexed variables at the low level. Furthermore, the existing events, whose guards or actions involve the spawned variables, should be refined by extending their guards and actions in order to satisfy the binding between the variables and the associated k-indexed variables. One noticeable feature in this case is that when we have a non-deterministic event of abstract level (as for the value of the sensors), then in the refinement the event should be refined (not extended). This is another reusable **specification rule** we have identified.

c) Merging the outputs of the computing modules with refinements As depicted in Fig. 3, the k-indexed *output* variables (for example $k\_extend\_EV(1)$ and $k\_extend\_EV(2)$) are merged using a logical OR to set the corresponding variable (for example $extend\_EV$) at the output of the digital part. Therefore the event that sets the variable should be guarded by the availability of the merged value. As explained before, a binding invariant should be provided for each variable and the related k-indexed variable. Several refinements are used to introduce the appropriate events.

d) Specifying the behaviour of the computing modules The two computing modules have the same behaviour. It is made of the events that monitor the system and set accordingly the state output variables and the input variables of the digital part, the events that give orders (control decision) to the physical part through the order output variables. It results in the k-indexed form of the events related to the three categories of the interface variables and the internal variables.

We stopped our construction at this stage. However following the guidelines provided in the method, it remains to perform the decomposition step in the basis of the *sense*, *monitor*, *control* events families (**Step 6**). Fortunately, the decomposition modules of Rodin provide assistance for this purpose. In our case the Abrial's style of decompostion which is based on share variables [2] is the most appropriate. Indeed, the decompostion is precisely based on the families of events: the *sense* family should be used for a (physical) machine while *monitor* and *control* families should be used for another (software) machine.

4 Assessment and Discussion

Coverage Applying the proposed method helped us a lot in mastering the case study. The resulting Event-B model presented in this article covers the main aspects of the landing system: the digital part with modules redundancy, its physical part and their interactions. The model covers mainly the safety properties of the LG system; liveness properties are treated by adapting Lamport's logical clocks [16]; but we have not deal with time constraints. Code generation was out of the scope of the current work. Nevertheless the management of huge B models is still tedious, since modifying the models already equipped with a lot of variables an events, at more abstract level requires redoing several steps of modelling, refinements and proving.

Experimentation with Rodin and statistics The Rodin tool is very efficient for proving the Event-B models; a very high percentage ($\sim 90\%$) of proof obligations was automatically discharged. The specifications are available online[3]. The current version is partial as we focus on representative events instead of being exhaustive. Statistics on Proof Obligations are given in Tab. 1. From a total of 619 POs, 547 of them were automatically discharged by Rodin and 6 of them were interactively discharged. Most of the POs at the abstract levels were proved. The undischarged POs are related to the structural refinement and specifically they are related to the binding invariants.

[3] http://www.lina.sciences.univ-nantes.fr/aelos/softwares/LGS/index_en.php

Managing very large models requires a rigorous slicing and several small steps of refinements. This is the reason why we have introduced many refinements, but it is still not enough, the slicing should be of finer grain. Moreover a good naming discipline is necessary at each level of the modelling. As far as the ProB animation tool (integrated in Rodin) is concerned, it is very helpful to tune the Event-B models.

	Total	Auto	Manual	Review.	Undis.
LandingSys5	619	547	6	0	66
Abstract model					
Landing_DP_Ctx	0	0	0	0	0
LandingSysDP_A	115	114	1	0	0
LandingSysDP_SWITCH_A	5	3	0	0	2
LandingSysDP_DOOR_A	42	42	0	0	0
LandingSysDP_DOOR_GEAR_A	79	79	0	0	0
LandingSysDP_DOOR_GEAR_TIME_A	2	2	0	0	0
Models of the vertical refinement					
LandingSysDP_DGT_R1_In	52	50	0	0	2
LandingSysDP_DGT_R2_INOUT	56	56	0	0	0
LandingSysDP_DGT_R3_INOUTDOOR	128	81	5	0	42
LandingSysDP_DGT_R3INOUTDOORGEAR	140	120	0	0	20

Table 1. Statistics of PO generated and proved with Rodin

Related Works The state of the art lacks of assistance methods. The four-variable model of software-controlled embedded systems originally proposed by Parnas and Madey has been used successfully in the development of safety-critical applications in various industries. But as mentioned by [18], the model does not explicitly specify the software requirements, but rather bounds them by specifying the system requirements and the input and output hardware interfaces of the system. We share the same the motivations with [13]. However the authors propose a method to synthesise the controller from the environment. They introduced the controller and its interface as a solution to the problem of maintaining a desired behaviour of an *autonomous system*. In our approach the controller is not synthesised to maintain a specific behaviour; it is built simultaneously with the environment according to the given control requirements, but the environment behaviour is less constrained by the controller. As far as method is concerned in the treatment of the LG system benchmark, all the related B specifications of the LG System are based on refinements. They do not describe a precise methodological process. Often, the authors need about ten refinements to include properties and requirements. The distinction between them is rather the way the refinements are organised rather than on methodological assumptions.

Su and Abrial [20] mentioned that there is no definitive answer for applying some recipe since the question varies from one project to another. They propose a light methodology with three steps: informal requirements, refinement strategy and formal model. They excluded features like redundancy and simplified time constraints. The systematic refinement strategy integrates progressively the devices, which is specific to the case study. Accordingly, our method focuses on a more general refinement strategy.

Mammar and Laleau adopted the four-variable model of software-controlled embedded systems originally proposed by Parnas and Madey. They used a series of refinements [17] first according to a variable classification first (monitor, control, output) then including timing aspects, failure cases and last properties. Mammar and Laleau focused on the control part only. Since it appears to be a logical organisation, a separation of concerns, this ordering delays most of the proof work to the last refinements. It lost modularity and extensibility. R. Banach used Hybrid Event-B to lead his study [4]; this extension enables one to carry continuous varying behaviours. R. Banach proposed a proof of concept of the language extension rather than a method or a full answer to the case study. However, hybrid-B seems adequate to refine the physical part of our current specification and especially to model time requirements.

Hansen et al. focused on the validation of the case using ProB rather than on the methodology of specifying with B [12]. As a matter of fact, the temporal properties are naturally introduced using LTL expressions. Another interesting feature is the ability to visualise the system execution. The counterpart is a simplified specification (no redundancy, no physical part, no failure). The refinements start with physical devices (door, gears, electro-valves), then the output, sensor and controllers are introduced as refinements and finally the general control (switch, valves, lights).

In [8] the authors present a technique for feature interactions for telecommunication services; it is a very close approach but our method is more general than the feature interactions. We plan to investigate more the connection of our approach with works on system engineering approaches [14], and cyber-physical systems where interconnected entities are interacting with the physical world [11].

5 Conclusion

We proposed a method (named Heñcher) to guide step by step the construction of embedded control systems with Event-B. We build on the well-known structure of control systems and on the experiments of several case studies where the Event-B was used and where some methodological guidelines was provided [10,9]. We provide a systematic use of the interface of the controller to build the components of the abstract model of the control system and, also how the features of the control system should be used to guide the successive refinements of the abstract model. A non trivial case study served as illustration and assessment of the proposal.

One flaw of the Event-B top-down approach is the constraint imposed by the evolution of the global abstract model defined before its refinement to the concrete models. This constraint prevents for an incremental model evolution. Indeed, if we miss some features in the abstract state, we will have to reconsider completely the structural refinements. It would be interesting to be able to mix both horizontal and vertical refinements in an incremental view of the design method. In [3] Back, have proposed guidelines for this purpose; an adaptation of this work to Event-B is likely to be interesting.

The reuse of existing independent models, with a bottom-up approach, would be interesting for managing large Event-B models. A typical example is the composition of existing models to build a given abstract model where each part can be modified and refined separately.

We plan to develop an assistance tool to help the user with various patterns, in the form of Event-B machines derived from the interface variables which will be extracted from a sketched graphical view of its control system (as in Fig. 2).

References

1. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)

2. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundam. Inform. **77**(1-2), 1–28 (2007)
3. Back, R.J.: Software Construction by Stepwise Feature Introduction. In: 2nd International Conference of B and Z Users, LNCS, vol. 2272, pp. 162–183. Springer (2002)
4. Banach, R.: The landing gear case study in hybrid event-b. In: Boniol et al. [6], pp. 126–141. DOI 10.1007/978-3-319-07512-9_9
5. Boniol, F., Wiels, V.: The landing gear system case study. In: Boniol et al. [6], pp. 1–18. DOI 10.1007/978-3-319-07512-9_1
6. Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.D. (eds.): ABZ 2014: The Landing Gear Case Study, CCIS, vol. 433. Springer International Publishing (2014)
7. Butler, M.: Decomposition Structures for Event-B. In: 7th International Conference on Integrated Formal Methods, IFM 2009, LNCS, vol. 5423, pp. 20–38. Springer (2009)
8. Cansell, D., Méry, D.: Playing with Abstraction and Refinement for Managing Features Interactions. In: ZB 2000: Formal Specification and Development in Z and B, pp. 148–167. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
9. Damchoom, K., Butler, M.J.: Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In: 12th Brazilian Symposium on Formal Methods, SBMF 2009, LNCS, vol. 5902, pp. 134–152. Springer (2009)
10. Damchoom, K., Butler, M.J., Abrial, J.R.: Modelling and Proof of a Tree-Structured File System in Event-B and Rodin. In: 10th International Conference on Formal Engineering Methods, ICFEM 2008, LNCS, vol. 5256, pp. 25–44. Springer (2008)
11. Edward A. Lee and Sanjit A. Seshia: Introduction to Embedded Systems - A Cyber-Physical Systems Approach. MIT Press (2017)
12. Hansen, D., Ladenberger, L., Wiegard, H., Bendisposto, J., Leuschel, M.: Validation of the abz landing gear system using prob. In: Boniol et al. [6], pp. 66–79
13. Hudon, S., Hoang, T.S.: Development of Control Systems Guided by Models of Their Environment. Electron. Notes Theor. Comput. Sci. **280**, 57–68 (2011)
14. I. Dragomir and I. Ober and D. Lesens: A Case Study in Formal System Engineering with SysML. In: 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems, pp. 189–198 (2012)
15. Jard, C., Roux, O.H. (eds.): Communicating Embedded Systems: Software and Design. Wiley-ISTE (2009)
16. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM **21**(7), 558–565 (1978)
17. Mammar, A., Laleau, R.: Modeling a landing gear system in event-b. In: Boniol et al. [6], pp. 80–94. URL http://dx.doi.org/10.1007/978-3-319-07512-9_6
18. Patcas, L.M., Lawford, M., Maibaum, T.: From system requirements to software requirements in the four-variable model. ECEASST **66** (2013). URL http://journal.ub.tu-berlin.de/eceasst/article/view/887
19. Silva, R., Butler, M.: Shared Event Composition/Decomposition in Event-B. In: 9th International Symposium onFormal Methods for Components and Objects, FMCO 2010, LNCS, vol. 6957, pp. 122–141. Springer (2012)
20. Su, W., Abrial, J.R.: Aircraft landing gear system: Approaches with event-b to the modeling of an industrial system. In: Boniol et al. [6], pp. 19–35. URL http://dx.doi.org/10.1007/978-3-319-07512-9_2
21. Su, W., Abrial, J.R., Zhu, H.: Complementary Methodologies for Developing Hybrid Systems with Event-B. In: T. Aoki, K. Taguchi (eds.) Formal Methods and Software Engineering, LNCS, vol. 7635, pp. 230–248. Springer Berlin Heidelberg (2012)