Modeling and Analyzing using ASMs: the Landing Gear System case study

Paolo Arcaini[1], Angelo Gargantini[1], and Elvinia Riccobene[2]

[1] Dipartimento di Ingegneria, Università degli Studi di Bergamo, Italy
{paolo.arcaini,angelo.gargantini}@unibg.it
[2] Dipartimento di Informatica, Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

Abstract. The paper presents an Abstract State Machine (ASM) specification of the Landing Gear System case study, and shows how the ASMETA framework can be used to support the modeling and analysis (validation and verification) activities for developing a rigorous and correct model in terms of ASMs. We exploit the two fundamental concepts of the ASM method, i.e., the notion of ground model and the refinement principle, and we achieve model development and model analysis by the combined use of formal methods for specification and for verification.

1 Introduction

The *Abstract State Machine* (ASM) method is a system engineering method that guides the development of software and embedded hardware-software systems seamlessly from requirements capture to their implementation. Within a precise but simple conceptual framework, the ASM method allows a *modeling technique* which integrates dynamic (*operational*) and static (*declarative*) descriptions, and an *analysis technique* that combines *validation* (by simulation and testing) and *verification* methods at any desired level of detail. The method has been successfully applied in different fields as: definition of industrial standards for programming and modeling languages, design and re-engineering of industrial control systems, modeling e-commerce and web services, design and analysis of protocols, architectural design, language design, verification of compilation schemes and compiler back-ends, etc.

ASMs are an extension of Finite State Machines, obtained by replacing unstructured control states by states comprising arbitrarily complex data [7]. The method has, therefore, a rigorous mathematical foundation [9], but a practitioner needs no special training to use the method since ASMs can be correctly understood as pseudo-code or virtual machines working over abstract data structures.

We here propose an ASM specification of the Landing Gear System (LGS), proposed in the ABZ conference as a real-life case study [5] with the aim of showing how different formal methods can be used for the specification, design and development of a complex system.

The ASM modeling process is based on the concept of a *ground model* representing a precise but concise high-level formalization of the system, and on the

refinement principle that allows to capture all details of the system design by a sequence of refined models till the desired level of detail.

After a brief introduction to ASMs in Section 2, Section 3 presents the modeling approach, and it also overviews a variety of model analysis activities that can be performed by using the ASMETA framework [4,12], a set of tools for the ASMs.

Section 4 reports the results of the modeling activity, and of the model validation and verification performed at each level of refinement. We start from a ground model that is the description of the *core* system, namely one landing set whose behavior is captured in terms of user input and doors' and gears' alleged state. Then we refine the model by adding the actuators' behavior in terms of electro-valves' and cylinders' operations; subsequently the sensors are added. The system with one landing component is then generalized to a system with three landing sets, and in the last refinement the health monitoring is included.

Section 5 discusses the strengths and the weaknesses of the approach, and outlines some future research directions. Since no other solutions of modeling and analysis of the LGS case study are available at the moment of writing this paper, we are not able to report any related work. Of course, many successful applications exist in literature regarding the use of the ASMs for complex system modeling and analysis. Due to their multiplicity, we prefer to refer to [9] for a complete introduction on the ASM method and the presentation of the great variety of its successful applications.

2 Abstract State Machines

Abstract State Machines (ASMs), whose complete presentation can be found in [9], are an extension of FSMs, where unstructured control states are replaced by states with arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e., domains of objects with functions and predicates defined on them. ASM states are modified by *transition relations* specified by "rules" describing the modification of the function interpretations from one state to the next one. There is a limited but powerful set of *rule constructors* that allow to express guarded actions (`if-then`), simultaneous parallel actions (`par`) or sequential actions (`seq`). Appropriate rule constructors also allow nondeterminism (existential quantification `choose`) and unrestricted synchronous parallelism (universal quantification `forall`).

An ASM state s is represented by a set of couples (*location*, *value*). ASM *locations*, namely pairs (*function-name*, *list-of-parameter-values*), represent the abstract ASM concept of basic object containers (memory units). Location *updates* represent the basic units of state change and they are given as assignments, each of the form $loc := v$, where loc is a location and v its new value.

Functions are classified as *derived*, i.e., those coming with a specification or computation mechanism given in terms of other functions, and *basic* which can be *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are

distinguished between *monitored* (only read by the machine and modified by the environment), and *controlled* (read and written by the machine).

A *computation* of an ASM is a finite or infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ of states of the machine, where s_0 is an initial state and each s_{n+1} is obtained from s_n by simultaneously firing all the transition rules which are enabled in s_n. The (unique) *main rule* is a transition rule and represents the starting point of the computation. An ASM can have more than one *initial state*. It is possible to specify state *invariants*.

3 Modeling process and supporting tools

The process of *requirements capture* results in constructing rigorous *ground models* which are precise but concise high-level system blueprints ("system contracts"), formulated in domain-specific terms, using an application-oriented language which can be understood by all stakeholders. The developer starts from the textual description of the informal requirements, and an ASM model is developed simply translating the text in terms of transition rules capturing the behavior of the system at a very high level of abstraction. This sketchy first model is usually neither "correct" nor "complete". Rather, it tries on purpose to expose errors, ambiguities, or incompletenesses in the original text. Correctness can be achieved through an iterative process reasoning on requirements till producing a ground model.

From the ground model, by step-wise refined models, further details are added to capture the major design decisions and provide descriptions of the complete software architecture and component design of the system. In this way the complexity of the system can be always taken under control, and it is possible to bridge, in a seamless manner, the gap between specification and code.

Still from its ground level, a model can be *validated* and *verified*. Model validation should be applied at the early stages of the system development, in order to ensure that the specification really reflects the user needs and statements about the system, and to detect faults in the specification as early as possible with limited effort. Validation should precede the application of more expensive and accurate methods, like formal requirements analysis and verification of properties, that should be applied only when a designer has enough confidence that the specification captures all informal requirements.

Tools allowing different forms of model analysis can surely help the developer in reaching model correctness. For the ASM method, the ASMETA (ASM mETAmodeling) framework[3] [4,12] provides basic functionalities for ASM models creation and manipulation (as editing, storage, interchange, access, etc.), as well as advanced model analysis techniques (as validation, verification, testing, model review, requirements analysis, runtime monitoring, etc.). The tools are strongly integrated in order to permit reusing information about models during several development phases.

[3] `http://asmeta.sourceforge.net/`

The concrete syntax `AsmetaL` is available for model *editing*. Model *simulation* is possible using `AsmetaS` [11]. The tool allows *invariant checking* to guarantee that the executed model always satisfies given properties, *consistent updates checking* for revealing inconsistent updates, *random simulation* where random values for monitored functions are provided by the environment, and *interactive simulation* when required inputs are provided interactively during simulation.

A more powerful validation approach is based on *scenario construction* by the ASM validator `AsmetaV` [10]. The validator is based on the `AsmetaS` simulator and on the `Avalla` modeling language. This last provides constructs to express execution scenarios in an algorithmic way as interaction sequences consisting of (a) *actions* committed by the user to `set` the environment, to `check` the machine state, and to ask for the `exec`ution of certain transition rules, and (b) the *reaction* of the machine to make one (or a sequence of) `step`(s) in response of the user actions.

A further validation technique is *model review* which aims at determining if a model not only fulfills the intended requirements, but it is of sufficient *quality* to be easy to develop, maintain, and enhance. Model review allows to identify defects early in the system development, reducing the cost of fixing them, so it is useful to apply this technique on models just sketched. The `AsmetaMA` tool [2] permits *automatic* review of ASMs. Typical vulnerabilities and defects a developer can introduce during the modeling activity using the ASMs are checked as violations of suitable meta-properties. The violation of a meta-property means that some attributes (minimality, completeness, redundancy, etc.) are not guaranteed and indicates the presence of actual faults, or only of potential faults.

Formal verification of ASMs is possible by means of `AsmetaSMV` [1]. This tool takes in input models written in `AsmetaL` and maps them into specifications for the model checker NuSMV. `AsmetaSMV` supports both the verification of *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas.

Tools for model-based testing and runtime verification are available in the ASMETA framework; we do not use them in this work, since we do not have any implementation to test. However, such techniques are explained and used in a separate paper regarding the sub-case study of the voting system of sensors [3], for which a Java implementation was developed.

3.1 Model Refinement

For complex systems, the complete specification can be reached by step-wise refinement, namely by a chain of models each of which is proved to be a correct refinement of the previous one. According to the notion of ASM refinement method presented in [6,8], to refine an ASM M to an ASM M^*, the following items must be defined:

- a notion of *refined state*;
- a notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^* of interest, i.e., the pairs of states in the runs one wants to relate through the refinement, including usually the correspondence of initial and (if there are any) of final states;

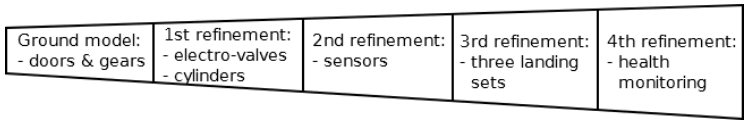

Fig. 1. Models chain

- a notion of abstract *computation segments* $\tau_1, \ldots, \tau_m$, where each τ_i represents a single M-step, and of corresponding refined computation segments $\sigma_1, \ldots, \sigma_n$, of single M^*-steps σ_j, which in given runs lead from corresponding states of interest to (usually the next) corresponding states of interest;
- a notion of *locations of interest* and of *corresponding* locations, i.e., pairs of (possibly sets of) locations one wants to relate in corresponding states;
- a notion of equivalence $\equiv$ of the data in the locations of interest; these local data equivalences usually accumulate to a notion of equivalence of corresponding states of interest.

According to this scheme, an ASM refinement allows one to combine a change of the signature (data refinement) with a change of the control (operation refinement), while many notions of refinement in the literature keep these two features separated.

Once the notions of corresponding states and of their equivalence have been determined, one can define that M^* is a correct refinement of M as follows:

Definition 1. *Fix a notion $\equiv$ of equivalence of states and of initial and final states. An ASM M^* is a correct refinement of an ASM M if and only if for each M^*-run $s_1^*, s_2^*, \ldots$, there is an M-run $s_1, s_2, \ldots$ and sequences $i_0 < i_1 < \ldots, j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv S^*_{j_k}$ for each k and either*
- *both runs terminate and their final states are the last pair of equivalent states; or*
- *both runs and both sequences $i_0 < i_1 < \ldots, j_0 < j_1 < \ldots$ are infinite.*

The states S_{i_k} and $S^*_{j_k}$ are the corresponding states of interest. They represent the end points of the corresponding computation segments (those of interest) for which the equivalence is defined in terms of a relation between their corresponding locations (those of interest).

4 Models chain of the LGS

In the following sections we present the five steps of the refinement process for modeling the case study[4]. Fig. 1 depicts the relationship existing between the models and, for each model, the system elements introduced with respect to the previous model. We start from the high level description (ground model) of the

[4] All the models are available online at `http://fmse.di.unimi.it/sw/landingGearSystem.zip`

```
asm LandingGearSystemGround

signature:
    enum domain HandleStatus = {UP | DOWN}
    enum domain DoorStatus = {CLOSED | OPENING | OPEN | CLOSING}
    enum domain GearStatus = {RETRACTED | EXTENDING | EXTENDED |
                              RETRACTING}
    dynamic monitored handle: HandleStatus
    dynamic controlled doors: DoorStatus
    dynamic controlled gears: GearStatus

definitions:
    rule r_closeDoor =
        switch doors
            case OPEN: doors := CLOSING
            case CLOSING: doors := CLOSED
            case OPENING: doors := CLOSING
        endswitch

    rule r_retractionSequence =
        if gears != RETRACTED then
            switch doors
                case CLOSED: doors := OPENING
                case CLOSING: doors := OPENING
                case OPENING: doors := OPEN
                case OPEN:
                    switch gears
                        case EXTENDED: gears := RETRACTING
                        case RETRACTING: gears := RETRACTED
                        case EXTENDING: gears := RETRACTING
                    endswitch
            endswitch
        else
            r_closeDoor[]
        endif

    rule r_outgoingSequence =
        if gears != EXTENDED then
            switch doors
                case CLOSED: doors := OPENING
                case OPENING: doors := OPEN
                case OPEN:
                    switch gears
                        case RETRACTED: gears := EXTENDING
                        case EXTENDING: gears := EXTENDED
                        case RETRACTING: gears := EXTENDING
                    endswitch
            endswitch
        else
            r_closeDoor[]
        endif

    invariant over gears, doors:
    (gears = EXTENDING or gears = RETRACTING) implies
    doors = OPEN

    invariant over gears, doors:
    doors = CLOSED implies
    (gears = EXTENDED or gears = RETRACTED)

    main rule r_Main =
        if handle = UP then
            r_retractionSequence[]
        else
            r_outgoingSequence[]
        endif

default init s0:
    function doors = CLOSED
    function gears = EXTENDED
```

Code 1. Ground model

system *core*, i.e., one landing set whose behavior is captured in terms of user input and doors' and gears' alleged state. Then we refine the model by adding the behavior of the actuators: electro-valves and cylinders. In the third step the sensors are added. The fourth refinement generalizes the system, moving from one landing component to a system with three equal landing sets. In the last refinement, the health monitoring is included.

For the first two refinement steps we prove that a model is a correct refinement of the more abstract one. For the further levels, the proof technique is similar and it has been skipped. On the ground model we apply different validation techniques (simulation, scenario construction, model review) that, due to lack of space, are not repeated in the other levels. If a refinement step is proved correct, all the properties already verified in the high-level model do not need to be verified again in the refined model. However, since the refinement process was guided by the requirements, and each refinement introduces new elements in the model, new properties regarding the newly added requirements have been added and verified at each suitable level.

4.1 Ground model

In the first model we have only modeled the doors and the gears and how their status changes. The model does not contain valves, cylinders, sensors, and the health monitoring. The complete ground model in shown in Code 1. Function `doors` represents the status of the doors that can be `OPEN`, `CLOSED`, `OPENING`

```
rule r_retractionSequence =
    if gears != RETRACTED then
        switch doors
            ...
            case OPEN:
                switch gears
                    case RETRACTING: gears := EXTENDED //error. It should be RETRACTED
                    ...
```

Code 2. Wrong ground model – Error in `r_retractionSequence`

or `CLOSING`. Function `gears` represents the status of the gears that can be `EXTENDED`, `RETRACTED`, `RETRACTING` or `EXTENDING`.

The state transitions are driven by the value of the monitored function `handle`. As long as the `handle` is `UP`, the *retraction sequence* [5] is executed, and, instead, as long as the `handle` is `DOWN`, the *outgoing sequence* [5] is executed. Let's see, as an example, how the retraction sequence works: so we assume that, in each state, the `handle` is `UP`. In the initial state, the `doors` are `CLOSED` and the `gears` are `EXTENDED`; then the `doors` start `OPENING`. When the `doors` become `OPEN`, the `gears` start `RETRACTING`. When the `gears` become `RETRACTED`, the `doors` start `CLOSING`. The retraction sequence terminates with the `doors CLOSED` and the `gears RETRACTED`. The outgoing sequence behaves similarly. Note that, a retraction (resp. an outgoing) sequence can be always interrupted by switching the value of the `handle`; in this case, an outgoing (resp. a retraction) sequence begins, starting from the status of the `doors` and the `gears` reached in the previous sequence.

An invariant checks that, if the `gears` are moving (i.e., they are `EXTENDING` or `RETRACTING`), the `doors` must be `OPEN`; another invariant checks that, if the `doors` are `CLOSED`, then the `gears` must be stopped (i.e., they are `EXTENDED` or `RETRACTED`).

Model review As first validation activity, we have checked the model with the model advisor. The first model we wrote actually contained an error, as shown in Code 2. Indeed, during a retraction sequence, the `gears` became `EXTENDED` instead of `RETRACTED`. The model advisor has discovered two meta-property violations (among the seven proposed in [2]):

- MP_5 requires that, for every domain element e, there exists a location which has value e. In the faulty model, MP_5 is violated since element `RETRACTED` of domain `GearStatus` is never used.
- MP_6 requires that every controlled location can take any value in its codomain. In the faulty model, MP_6 is violated since function `gears` does not take the value `RETRACTED` of its codomain.

Obviously, both meta-property violations are caused by the same error in the model. Note that behavioral faults often reveal themselves as stylistic defects and therefore they can be captured by the model advisor.

Simulation By simulation we were able to identify the state in which the erroneous rule was executed. Fig. 2 shows the simulation trace of the wrong ground

```
Insert a symbol of HandleStatus
in [UP, DOWN] for handle:
UP
<State 0 (monitored)>
handle=UP
</State 0 (monitored)>
<State 1 (controlled)>
doors=OPENING
gears=EXTENDED
</State 1 (controlled)>
```

```
Insert a symbol of HandleStatus
in [UP, DOWN] for handle:
UP
<State 1 (monitored)>
handle=UP
</State 1 (monitored)>
<State 2 (controlled)>
doors=OPEN
gears=EXTENDED
</State 2 (controlled)>
```

```
Insert a symbol of HandleStatus
in [UP, DOWN] for handle:
UP
<State 2 (monitored)>
handle=UP
</State 2 (monitored)>
<State 3 (controlled)>
doors=OPEN
gears=RETRACTING
</State 3 (controlled)>
```

```
Insert a symbol of HandleStatus
in [UP, DOWN] for handle:
UP
<State 3 (monitored)>
handle=UP
</State 3 (monitored)>
<State 4 (controlled)>
doors=OPEN
gears=EXTENDED
</State 4 (controlled)>
```

Fig. 2. Simulation of the wrong ground model

```
Insert a symbol of HandleStatus
in [UP, DOWN] for handle:
UP
<State 0 (monitored)>
handle=UP
</State 0 (monitored)>
<State 1 (controlled)>
doors=OPENING
gears=EXTENDED
</State 1 (controlled)>
Insert a symbol of HandleStatus
in [UP, DOWN] for handle:
UP
<State 1 (monitored)>
handle=UP
</State 1 (monitored)>
<State 2 (controlled)>
doors=OPEN
gears=EXTENDED
</State 2 (controlled)>
```

```
Insert a symbol of HandleStatus
in [UP, DOWN] for handle:
UP
<State 2 (monitored)>
handle=UP
</State 2 (monitored)>
<State 3 (controlled)>
doors=OPEN
gears=RETRACTING
</State 3 (controlled)>
Insert a symbol of HandleStatus
in [UP, DOWN] for handle:
UP
<State 3 (monitored)>
handle=UP
</State 3 (monitored)>
<State 4 (controlled)>
doors=OPEN
gears=RETRACTED
</State 4 (controlled)>
```

```
Insert a symbol of HandleStatus
in [UP, DOWN] for handle:
UP
<State 4 (monitored)>
handle=UP
</State 4 (monitored)>
<State 5 (controlled)>
doors=CLOSING
gears=RETRACTED
</State 5 (controlled)>
Insert a symbol of HandleStatus
in [UP, DOWN] for handle:
UP
<State 5 (monitored)>
handle=UP
</State 5 (monitored)>
<State 6 (controlled)>
doors=CLOSED
gears=RETRACTED
</State 6 (controlled)>
```

Fig. 3. Simulation of the correct ground model – Complete retraction sequence

```
scenario lgsGround1
load LandingGearSystemGround.asm

set handle := UP;
step
check doors = OPEN and gears = EXTENDED;

set handle := UP;
step
check doors = OPEN and gears = RETRACTING;

set handle := UP;
step
check doors = OPEN and gears = RETRACTED;
```

Code 3. Scenario reproducing the simulation that leads to the error

model. During an interactive simulation, at each step the user is asked for the values of the monitored functions (in this case the function `handle`).

Fig. 3 shows the simulation, over the correct ground model, of the complete retraction sequence described previously.

Scenario-based validation We have then built a scenario describing the simulation that brings to the execution of the erroneous rule shown in Code 2; a scenario permits to automatize the execution of a run that must be executed more than once. Code 3 shows the scenario in which, before each step, the value of the monitored function `handle` is set to `UP`, and, after the simulation step, the values of functions `doors` and `gears` are checked. The scenario execution consists in a simulation, similar to that seen in Fig. 2. However, the simulation is not interactive, since the values of the monitored functions are set according to the values specified in the scenario. Moreover, the scenario execution also checks for the specified properties. Fig. 4 shows the output of the scenario execution over the faulty ground model; we can notice that, in the fourth step, the specified

```
<State 1 (controlled)>
doors=OPENING
gears=EXTENDED
handle=UP
</State 1 (controlled)>
"check succeeded: doors = OPENING and gears = EXTENDED"
<State 2 (controlled)>
doors=OPEN
gears=EXTENDED
handle=UP
</State 2 (controlled)>
"check succeeded: doors = OPEN and gears = EXTENDED"
```

```
<State 3 (controlled)>
doors=OPEN
gears=RETRACTING
handle=UP
</State 3 (controlled)>
"check succeeded: doors = OPEN and gears = RETRACTING"
<State 4 (controlled)>
doors=OPEN
gears=EXTENDED
handle=UP
</State 4 (controlled)>
"CHECK FAILED: doors = OPEN and gears = RETRACTED at step 4"
```

Fig. 4. Execution of the scenario shown in Code 3 over the wrong ground model

property has been violated. We have later executed the scenario over the correct model and all the checks have been successful. Scenarios may be thought as use cases that drive the development of the model in a sort of Behaviour-Driven Development: a model is enhanced and/or fixed until all the scenarios execute without failures.

Model checking In the ground model we have been able to verify four normal mode requirements among those reported in the case study: $R_{11}bis$, $R_{12}bis$, R_{21}, and R_{22}. For example, requirement $R_{11}bis$ requires that, *when the command line is working (normal mode), if the landing gear command handle has been pushed DOWN and stays DOWN, then eventually the gears will be locked down and the doors will be seen closed.*

We have verified the following four CTL properties:

ag(ag(handle = DOWN) implies af(gears = EXTENDED and doors = CLOSED)) //$R_{11}bis$
ag(ag(handle = UP) implies af(gears = RETRACTED and doors = CLOSED)) //$R_{12}bis$
ag(ag(handle = DOWN) implies ax(ag(gears != RETRACTING))) //R_{21}
ag(ag(handle = UP) implies ax(ag(gears != EXTENDING))) //R_{22}

4.2 First refinement: adding the electro-valves and the cylinders

In this model we have refined the ground model by adding the representation of the electro-valves and of the cylinders. Code 4 shows the new elements introduced in the model. We have added the functions for the general electro-valve (`generalEV`) and the electro-valves related to the opening/closing of the doors (`openDoorsEV` and `closeDoorsEV`) and the retracting/extending of the gears (`retractGearsEV` and `extendGearsEV`), that represent the actuators of the system. These functions have been declared controlled.

Functions `cylindersDoors` and `cylindersGears` represent the status of cylinders that move the doors and the gears. The functions have been declared as *derived*, since they can be defined in terms of the values of functions `doors` and `gears`. For example, the cylinders of the doors are extended/retracted when the doors are open/closed, and extending/retracting when the doors are opening/-closing. A similar relation exists between the gears and their cylinders.

```
asm LandingGearSystemWithCylAndValves

signature:
    ...
    enum domain CylinderStatus =
    {CYL_EXTENDING | CYL_RETRACTING |
    CYL_RETRACTED | CYL_EXTENDED}
    derived cylindersDoors: CylinderStatus
    derived cylindersGears: CylinderStatus
    dynamic controlled generalEV: Boolean
    dynamic controlled openDoorsEV: Boolean
    dynamic controlled closeDoorsEV: Boolean
    dynamic controlled retractGearsEV: Boolean
    dynamic controlled extendGearsEV: Boolean

definitions:
    function cylindersDoors =
        switch doors
            case OPEN: CYL_EXTENDED
            case OPENING: CYL_EXTENDING
            case CLOSING: CYL_RETRACTING
            case CLOSED: CYL_RETRACTED
        endswitch

    function cylinderGearStatus = ...

    rule r_closeDoor =
        switch doors
            case OPEN:
                par
                    closeDoorsEV := true
                    doors := CLOSING
                endpar
            ...

    rule r_retractionSequence =
        if gears != RETRACTED then
            switch doors
                case CLOSED:
                    par
                        generalEV := true
                        openDoorsEV := true
                        doors := OPENING
                    endpar
                ...
```

Code 4. Model with cylinders and electro-valves

Model review The model advisor signals that functions `cylindersDoors` and `cylindersGears` are useless, since they are never used (never read). Indeed, we have added the cylinders only for documentation purposes, but they could be omitted from the model, since their status is given by a straightforward relation with the status of the doors/gears.

Correctness of the model refinement Let us call M the ground model `LandingGearSystemGround` and M^* the refined model `LandingGearSystemWithCylAndValves`. At M-level, the locations of interest are those for functions `doors` and `gears`, which have corresponding locations for the same function names at level M^* (since the refinement simply extends the signature of machine M). Two states $s \in S$ and $s^* \in S^*$ are equivalent, i.e., $s \equiv s^*$, iff $[\![\texttt{doors}]\!]_s = [\![\texttt{doors}]\!]_{s^*} \wedge [\![\texttt{gears}]\!]_s = [\![\texttt{gears}]\!]_{s^*}$. In order to prove the correctness of the refinement, we apply Def. 1.

Let $s_0^*, s_1^*, \ldots, s_n^*$ be an M^* run. Let us consider the sequence $t = ([\![\texttt{handle}]\!]_{s_0^*}, [\![\texttt{handle}]\!]_{s_1^*}, \ldots, [\![\texttt{handle}]\!]_{s_{n-1}^*})$. If we apply sequence t to M, we obtain a run $s_0, s_1, \ldots, s_n$ such that $s_i \equiv s_i^*, \forall i = 0, \ldots, n$.

Model checking In this model, we have been able to verify the normal mode requirements R_{31}, R_{32}, R_{41}, R_{42}, and R_{51}. For example, requirement R_{31} requires that, *when the command line is working (normal mode), the stimulation of the gears outgoing or the retraction electro-valves can only happen when the three doors are locked open.*

We have verified the following four CTL properties:

ag((extendGearsEV or retractGearsEV) implies doors = OPEN) //R_{31}
ag((openDoorsEV or closeDoorsEV) implies
(gears = RETRACTED or gears = EXTENDED)) //R_{32}
ag(not(openDoorsEV and closeDoorsEV)) //R_{41}
ag(not(extendGearsEV and retractGearsEV)) //R_{42}
ag((openDoorsEV or closeDoorsEV or extendGearsEV or retractGearsEV)
implies generalEV) //R_{51}

```
asm LandingGearSystemWithCylValvesAndSensors

signature:
    ...
    dynamic monitored gearsExtended: Boolean
    dynamic monitored gearsRetracted: Boolean
    dynamic monitored doorsClosed: Boolean
    dynamic monitored doorsOpen: Boolean

definitions:

    rule r_closeDoor =
        switch doors
            case CLOSING:
                if doorsClosed then
                    par
                        generalEV := false
                        closeDoorsEV := false
                        doors := CLOSED
                    endpar
                endif
            ...
    rule r_retractionSequence =
        if gears != RETRACTED then
            switch doors
                case CLOSED:
                    par
                        generalEV := true
                        openDoorsEV := true
                        doors := OPENING
                    endpar
                case OPENING:
                    if doorsOpen then
                        par
                            openDoorsEV := false
                            doors := OPEN
                        endpar
                    endif
                ...

invariant over doorsClosed, doorsOpen: not(doorsClosed and doorsOpen)
invariant over gearsExtended, gearsRetracted: not(gearsExtended and gearsRetracted)
```

Code 5. Model with cylinders, electro-valves, and sensors

4.3 Second refinement: adding the sensors

The model presented in this section extends the model described in Section 4.2 by adding the modeling of the sensors. Code 5 shows the new elements introduced in the model. Four boolean monitored functions are used to indicate whether the gears are extended (`gearsExtended`) or retracted (`gearsRetracted`), and whether the doors are closed (`doorsClosed`) or open (`doorsOpen`). In ASMs, monitored functions represent quantities that are not determined by the system, but that come from the *environment*; usually, they are used in transitions rules (e.g., in the guard of a conditional rule or in the right part of an update rule) to modify the state of the system. For this reason, we chose to model the sensors as monitored functions, because, in the landing gear system, the sensors can be seen as inputs that determine the status of the system: for example, whenever the sensor `gearsExtended` is seen turned on, the gears are considered extended by the system.

In this model, we have refined some rules by adding the reading of sensors. Some update rules have been guarded by conditional rules checking the value of the monitored functions; for example, we can see in Code 5 that, if the `doors` are `CLOSING`, they become `CLOSED` only if the sensor `doorsClosed` is turned on (i.e., the guard of conditional rule is true).

In this paper, we do not model the sensor voting module, that is modeled and analysed in [3]. Moreover, we assume that impossible combinations of sensor values (e.g., both sensors `doorsClosed` and `doorsOpen` turned on) cannot appear. In order to check that only admissible combinations of sensor values are provided by the environment, we add to the model two invariants specifying that `doorsClosed` and `doorsOpen` cannot be turned on together, and that `gearsExtended` and `gearsRetracted` cannot be turned on together (see Code 5). An alternative solution could be to make the model more robust, by accepting any combination of sensor values, but modifying the ASM state only upon the observation of correct combinations: this would require to make the guards of the transition rules more complex.

```
asm LandingGearSystemWithCylValvesAndSensors3LS

signature:
    ...
    enum domain LS = {FRONT | LEFT | RIGHT}
    dynamic monitored gearsExtended: LS -> Boolean
    dynamic monitored gearsRetracted: LS -> Boolean
    dynamic monitored doorsClosed: LS -> Boolean
    dynamic monitored doorsOpen: LS -> Boolean
    derived gearsExtended: Boolean
    derived gearsRetracted: Boolean
    derived doorsClosed: Boolean
    derived doorsOpen: Boolean

definitions:
    function gearsExtended =
        (forall $s in LS with gearsExtended($s))

    function gearsRetracted =
        (forall $s in LS with gearsRetracted($s))

    function doorsClosed =
        (forall $s in LS with doorsClosed($s))

    function doorsOpen =
        (forall $s in LS with doorsOpen($s))
    ...
```

Code 6. Model with cylinders, electro-valves, and sensors – Three landing sets

Correctness of the model refinement Let us call M the model `LandingGearSystemWithCylAndValves` and M^* the refined model `LandingGearSystemWithCylValvesAndSensors`. At M-level, the locations of interest are those as in the previous refinement. Two states s and s^* are equivalent, i.e., $s \equiv s^*$, iff $[\![\texttt{doors}]\!]_s = [\![\texttt{doors}]\!]_{s*} \wedge [\![\texttt{gears}]\!]_s = [\![\texttt{gears}]\!]_{s*}$. Let $updLocs(s_i^*)$ be the set of locations that are non-trivially updated in state s_i^* (so having a different value in state s_{i+1}^*). In order to prove the correctness of the refinement, we apply Def. 1.

Let $s_0^*, s_1^*, \ldots, s_n^*$ be an M^* run; we say that a state s_i^* is of interest if $i = 0 \vee \texttt{doors} \in updLocs(s_{i-1}^*) \vee \texttt{gears} \in updLocs(s_{i-1}^*)$. Given the sequence of states of interest $s_{j_0}^*, s_{j_1}^*, \ldots, s_{j_m}^*$, such that $j_0 = 0$ and $j_0 < j_1 < \ldots < j_m \leqslant n$, we build the sequence $t = ([\![\texttt{handle}]\!]_{s_{j_1-1}^*}, [\![\texttt{handle}]\!]_{s_{j_2-1}^*}, \ldots, [\![\texttt{handle}]\!]_{s_{j_m-1}^*})$. If we apply sequence t to M, we obtain a run $s_0, s_1, \ldots, s_m$ such that $s_i \equiv s_{j_i}^*, \forall i = 0, \ldots, m$.

Model checking The introduction of the sensors do not require to verify any further requirement.

4.4 Third refinement: adding the three landing sets

The model presented in this section extends the model described in Section 4.3 by adding the modeling of the three landing sets. Code 6 shows the new elements introduced in the model and how some functions have been modified. The enumerative domain `LS` represents the three landing sets (`FRONT`, `LEFT`, and `RIGHT`). The sensors have been refined by explicitly modeling, for each sensor type, the sensor on each landing set; four new unary monitored functions with domain `LS` have been added to the model. For example, the unary monitored function `gearsExtended` represents the three sensors associated with the three landing sets, that detect the extension of the gears: specifically, each location of the function (`gearsExtended(FRONT)`, `gearsExtended(LEFT)`, and `gearsExtended(RIGHT)`) is a sensor of a landing set.

The 0-ary functions that in the previous model (Section 4.3) are declared as monitored, in this model are declared as derived, because now their value depends on the value of the corresponding unary functions having the same name. Indeed, each derived function describes if all the corresponding sensors on the three landing sets are turned on, or if at least one is turned off.

```
asm LandingGearSystemWithHealthMon3LS

signature:
    ...
    derived aGearExtended: Boolean
    derived aGearRetracted: Boolean
    derived aDoorClosed: Boolean
    derived aDoorOpen: Boolean
    derived greenLight: Boolean
    derived orangeLight: Boolean
    derived redLight: Boolean
    dynamic monitored timeout: Boolean
    dynamic controlled anomaly: Boolean

definitions:
    function aGearExtended = (exist $s in LS with gearsExtended($s))
    function aGearRetracted = (exist $s in LS with gearsRetracted($s))
    function aDoorClosed = (exist $s in LS with doorsClosed($s))
    function aDoorOpen = (exist $s in LS with doorsOpen($s))

    function greenLight = (gears = EXTENDED)
    function orangeLight = (gears = EXTENDING or gears = RETRACTING)
    function redLight = anomaly
    ...
```

```
    rule r_healthMonitoring =
        if timeout then
            if (openDoorsEV and not(doorsOpen)) or
                (closeDoorsEV and aDoorOpen) or
                (retractGearsEV and aGearExtended) or ...
                anomaly := true
            endif
        endif

    main rule r_Main =
        if not(anomaly) then
            par
                if handle = UP then
                    r_retractionSequence[]
                else
                    r_outgoingSequence[]
                endif
                r_healthMonitoring[]
            endpar
        endif

default init s0:
    function anomaly = false
    ...
```

Code 7. Model with cylinders, electro-valves, and sensors – With failure mode

Note that AsmetaL permits function overloading, i.e., having different functions with the same name, but a different arity and/or a different domain.

Correctness of the model refinement The proof of the correctness of the model refinement is straightforward, and it should be done as seen for the two previous models.

Model checking The introduction of the three landing sets do not require to verify any further requirement.

4.5 Fourth refinement: adding the health monitoring system

The model presented in this section extends the model described in Section 4.4, by adding the modeling of the health monitoring system (Section 4.3 of the case study in [5]). We only consider the doors motion monitoring and the gears motion monitoring. A possible way to model the monitoring of the sensors is described in [3]. Since the analogical switch and the pressure sensor are not considered in this work, we do not model their monitoring.

Code 7 shows the new elements introduced in the model. The health monitoring is executed by rule `r_healthMonitoring` that, whenever a *timeout* has occurred, checks that the values of the sensors are as expected. The detection of an anomaly in the system is modeled by the update to *true* of the boolean function `anomaly`; in the main rule, the system is executed only if there is no anomaly (i.e., `anomaly` is false). The timeout is modeled through the boolean monitored function `timeout`. Note that, at this level of abstraction, we do not need to explicitly handle the time, neither to distinguish between different time intervals: it is sufficient to know if, in a given system configuration, the maximum allowed time interval, after which the system configuration should be observed changed, has elapsed. For example, if the electro-valve responsible for the opening of the doors

is turned on and the doors are not open (openDoorsEV and not(doorsOpen)), if the `timeout` has elapsed, then an anomaly has been detected[5].

In the monitoring rules, sometimes we need to know if, given a sensor type, at least one single sensor is turned on. For example, one monitoring rule states that *if the control software does not see the value door_open*$[x] = false$ *for all* $x = \{front, left, right\}$ *...*; in order to implement this rule, we must check if at least one door is open, but this can not be inferred through function `doorsOpen`. In order to model this kind of rules, we have introduced in this model the functions `aDoorOpen`, `aDoorClosed`, `aGearExtended`, and `aGearRetracted` that signal if there is at least one of the corresponding sensors turned on.

Correctness of the model refinement The proof of the correctness of the model refinement is straightforward.

Model checking In this model, we have been able to verify the failure mode requirements R_{61}, R_{62}, R_{63}, R_{64}, R_{71}, R_{72}, R_{73}, and R_{74}. For example, requirement R_{61} requires that, *if one of the three doors is still seen locked in the closed position more than 7 seconds after stimulating the opening electro-valve, then the boolean output normal mode is set to false.*

We have verified the following eight CTL properties:

ag((openDoorsEV and aDoorClosed and timeout) implies ax(ag(anomaly))) //R_{61}
ag((closeDoorsEV and aDoorOpen and timeout) implies ax(ag(anomaly))) //R_{62}
ag((retractGearsEV and aGearExtended and timeout) implies ax(ag(anomaly))) //R_{63}
ag((extendGearsEV and aGearRetracted and timeout) implies ax(ag(anomaly))) //R_{64}
ag((openDoorsEV and not(doorsOpen) and timeout) implies ax(ag(anomaly))) //R_{71}
ag((closeDoorsEV and not(doorsClosed) and timeout) implies ax(ag(anomaly))) //R_{72}
ag((retractGearsEV and not(gearsRetracted) and timeout) implies ax(ag(anomaly))) //R_{73}
ag((extendGearsEV and not(gearsExtended) and timeout) implies ax(ag(anomaly))) //R_{74}

5 Discussion and conclusion

The paper presents an ASM specification of the Landing Gear System case study [5]. The modeling process exploits the two fundamental concepts of the ASM method, i.e., the concept of ground model and the refinement principle.

The use of the refinement approach helped us to manage the complexity of the case study and to achieve the verification of the given requirements. Actually the refinement was guided by the requirements to be verified, since they gave the hint on how to proceed in adding details at each refinement step. Indeed, every refinement step came with a set of suitable novel properties to be verified. Even though, thanks to the proof of refinement correctness, properties already

[5] In the case study, this behavior is described as follows: *if the control software does not see the value door_closed*$[x] = false$ *for all* $x \in \{front, left, right\}$ *7 seconds after stimulation of the opening electro-valve, then the doors are considered as blocked and an anomaly is detected.*

verified at a given step were guaranteed in the refined steps, we have kept the whole set of properties and verified them by model checking at each step.

Among the possible views proposed in the informal requirements – functional, architectural, real time, reliability, etc. – we do not cover real time aspects. Although reactive timed ASMs [13] have been proposed for dealing with time in ASMs, they are not supported by our tools for model analysis. This is the reason why, for properties R_1 (see Section 5.1 in [5]), we verified the weaker version. We modeled the time passing by means of a suitable monitored function *timeout* which was enough for achieving the automatic verification of all the properties regarding *failure mode requirements* (see Section 5.2 of [5]).

From the functional view, we abstracted from the analogical switch and the pressure sensor, while, from the architecture view, we simplified the digital architecture by only considering *one* computing module. Both abstractions are not due to limitations of the method, but simply to the lack of space. Both these functional and architectural abstractions are, however, straightforward to detail. Abstracting from the analogical switch and the pressure sensor also influenced the modeling of the health monitoring. Therefore, regarding the system reliability, we did not deal with scenarios involving these two devices.

In the specification presented here, also the model of the sensor voting is missing. Indeed it has been considered as case study in a separate paper [3] to present two approaches for checking the implementation conformance: an offline model-based testing approach and an online runtime monitoring approach.

By using the simulator and the validator for scenarios construction, we were able to reproduce the expected scenarios of the LGS operating in normal mode (see Section 4.1 in [5]), even if this simulation is not reported here.

The model development and the model analysis have been made possible by the combined use of formal methods for modeling and for verification. In fact, the behavioral specification is expressed in terms of ASMs, while the verification of the properties, as well as other forms of model analysis (e.g., model review), is conducted by the use of the NuSMV model checker. The advantage, in our case, is that all methods are integrated in the same framework, ASMETA, so the user does not need to worry about translating the ASM specification into the language of the model checker. The mapping from an ASM model into a NuSMV model is automatic and the CTL properties can be directly expressed as part of the ASM model itself.

What is missing in the method, apart from the real time aspects, is a mechanical support by theorem provers for verifying the refinement correctness, and the definition of refinement patterns that could be useful to guide the refinement process. For this case study, the refinement steps were suggested by the properties to verify and the refinement correctness was proved by hand. These topics will be arguments for future research, as well as the possibility to integrate the ASM method with other approaches, as the Event-B, that are better structured in this respect.

References

1. P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In *Proceedings of the 2nd International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010)*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2010.
2. P. Arcaini, A. Gargantini, and E. Riccobene. Automatic Review of Abstract State Machines by Meta Property Verification. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, pages 4–13. NASA, 2010.
3. P. Arcaini, A. Gargantini, and E. Riccobene. Offline model-based testing and runtime monitoring of the sensor voting module. In *ABZ Case Study*, volume 433 of *Communications in Computer Information Science*. Springer, 2014.
4. P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.
5. F. Boniol and V. Wiels. The Landing Gear System Case Study. In *ABZ Case Study*, volume 433 of *Communications in Computer Information Science*. Springer, 2014.
6. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
7. E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Proc. of FroCoS 2005*, volume 3717 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2005.
8. E. Börger. Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. *Formal Aspects of Computing*, 19:225–241, 2007.
9. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
10. A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A Scenario-Based Validation Language for ASMs. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z (ABZ 2008)*, volume 5238 of *Lecture Notes in Computer Science*, pages 71–84. Springer-Verlag, 2008.
11. A. Gargantini, E. Riccobene, and P. Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *J. Universal Computer Science*, 14(12):1949–1983, 2008.
12. A. Gargantini, E. Riccobene, and P. Scandurra. Model-Driven Language Engineering: The ASMETA Case Study. In *Int. Conf. on Software Engineering Advances, ICSEA*, pages 373–378, 2008.
13. A. Slissenko and P. Vasilyev. Simulation of Timed Abstract State Machines with predicate logic model-checking. *J.UCS*, 14(12):1984–2006, 2008.