**Software Tools for Technology Transfer manuscript No.**
(will be inserted by the editor)

# Rigorous development process of a safety-critical system: from ASM models to Java code

**Paolo Arcaini[1⋆], Angelo Gargantini[2], Elvinia Riccobene[3]**

[1] Charles University in Prague, Faculty of Mathematics and Physics, e-mail: arcaini@d3s.mff.cuni.cz
[2] Department of Management, Information and Production Engineering, University of Bergamo, e-mail: angelo.gargantini@unibg.it
[3] Department of Computer Science, Università degli Studi di Milano, e-mail: elvinia.riccobene@unimi.it

Received: date / Accepted: date

**Abstract.** The paper presents an approach for rigorous development of safety-critical systems based on the Abstract State Machine formal method. The development process starts from a high level *formal* view of the system and, through *refinement*, derives more detailed models till the desired level of specification. Along the process, different validation and verification activities are available, as simulation, model review, and model checking. Moreover, each refinement step can be proved correct using an SMT-based approach. As last step of the refinement process, a Java implementation can be developed and linked to the formal specification. The correctness of the implementation w.r.t its formal specification can be proved by means of model-based testing and runtime verification. The process is exemplified by using a Landing Gear System as case study.

## 1 Introduction

In safety-critical systems, human safety depends upon the correct operation of the system. Therefore, they need development methods and processes that could lead to provably correct systems. Rigorous development processes require the use of formal methods which can guarantee, thanks to their mathematical foundation, model preciseness and properties assurance.

However, although there are several cases showing the advantages of applying formal methods in industrial applications [44], many practitioners are still reluctant to adopt them. Besides the well-known lack of training, this skepticism is mainly due to (i) the complex formal notations, (ii) the lack of easy-to-use tools supporting a developer during the life cycle activities of system development, possibly in a seamless manner, and (iii) the lack of a development process guidance and a methodology, which lead the designer from the requirements to the final implementation.

The Abstract State Machine (ASM) method [18] is a system engineering method that can guide the development of software and embedded hardware-software systems seamlessly from requirements capture to their implementation. This is shown by the method's application in a series of cases studies in different application domains [18]. ASMs are transition systems that extend the Finite State Machines (FSMs) [17]; the method has, therefore, a rigorous mathematical foundation, 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. The ASM-based 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.

In the context of safety-critical systems, the Landing Gear System (LGS) was proposed in the ABZ 2014 conference as a real-life case study [14] with the aim of showing how different formal methods can be used for the specification, design and development of a complex system. Many examples of rigorous formalizations have been presented in [15], covering different aspects of specification, verification, and animation.

In this paper, we take advantage of the LGS case study to present a rigorous development process for safety-critical systems based on the use of the ASMs. The process goes from model to code and it is assisted by a series of tools that can be employed for different forms of analysis either at model level w.r.t. the requirements, and at code level w.r.t. the models.

We do not consider the whole description of the case study, since the focus is not on the case study itself – we are not interested in its complete development –, but rather on showing the effectiveness and ease of using the ASMs and the framework ASMETA [9] (a set of tools for the ASMs) for designing and implementing systems in a rigorous and controllable way. We concentrate on those requirements regarding the safety-critical functionalities of the system, and we show the advantages offered by the ASM method and its tools either in terms of *modeling techniques*, which integrates dynamic (*operational*) and static (*declarative*) descriptions, and in terms of *analysis technique* that combines *validation* (by simulation and testing) and *verification* methods at any desired level of detail, as well as *conformance checking* between models

⋆ The work was partially supported by Charles University research funds PRVOUK.

and code (if any). We skip some details concerning the description of the architecture and the hardware of the LGS, not because of their unimportance, but because they would not add any new aspect or potentiality of our approach, while their modeling is only a question of reaching the suitable refinement level.

Among the specification methods reported in [15], the work in [6] already presents an ASM specification of the case, and shows how to prove the required properties by means of integrated formal approaches for verification. Furthermore, techniques for runtime verification, still based on ASM models, are presented in [7] and regard the sub-case study of the voting system of sensors, for which a Java implementation was developed.

Compared with the results in [6, 7], this work (a) describes the complete process from a very abstract model to an executable Java implementation, (b) improves the technique of ASM model refinement since its correctness is checked automatically, and (c) applies conformance checking techniques to the whole system.

After a brief introduction to ASMs in Section 2, Section 3 presents the ASM-based development process, and it overviews the variety of model analysis activities that can be performed by using the ASMETA framework. Section 4 presents the ground model of the LGS while Section 5 reports the chain of refined models. 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 inputs 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. Compared with the results in [6], here we are able to check refinement correctness automatically. Section 6 reports the results of the model validation and verification performed at each level of refinement. We present different forms of model validation (simulation, scenarios construction, model review), and techniques for model checking LTL temporal properties. Section 7 presents how a formal specification can be linked with its implementation, and the two conformance validation techniques, model-based testing and runtime verification, that we have applied after having implemented in Java the last refined model of the LGS. Section 8 discusses the strengths and the weaknesses of the approach w.r.t. other existing approaches, taking advantage of the availability of other formalizations of the LGS case study. Finally, Section 9 concludes the paper.

## 2 Abstract State Machines (ASMs)

ASMs are an extension of Finite State Machines, obtained by replacing unstructured control states by states comprising arbitrarily complex data [17].

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*. State invariants can be used for two purposes: either specifying a property that must be checked in each state (when formulated over controlled and derived functions), or stating a constraint over the input values (when specified over monitored functions).

Concepts briefly recalled here are related to the definition of basic ASMs. There are several extensions to model any kind of computational paradigm: from a single agent executing simultaneous parallel actions, to distributed multiple agents interacting in a synchronous or asynchronous way. A complete presentation of the ASMs can be found in [18].

### 2.1 Ground model and model refinement

For system specification, the ASM method builds upon two main concepts:

- *ground model*, an ASM which is a precise but concise high-level description of the system and can be

considered as "authoritative" reference model for the design;
- *model refinement*, which is a general scheme for stepwise instantiations of model abstractions to concrete system elements, providing controllable links between the more and more detailed descriptions at the successive stages of system development.

The definition of *correct refinement* between ASM models has been given in [16]. According to this definition, to refine an ASM $M$ to an ASM $\widetilde{M}$, 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 $\widetilde{M}$-states $\widetilde{S}$ of interest, i.e., the pairs of states in the runs one wants to relate through the refinement, usually including the correspondence of initial and (if there are any) of final states;
- a notion of abstract *computation segments* $\tau = \tau_1, \ldots, \tau_m$, where each $\tau_i$ represents a single $M$-step, and of corresponding refined computation segments $\sigma = \sigma_1, \ldots, \sigma_n$, of single $\widetilde{M}$-steps $\sigma_j$, 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 $\widetilde{M}$ is a correct refinement of $M$ as follows:

**Definition 1 (Börger's refinement).** Given a notion $\equiv$ of equivalence, an ASM $\widetilde{M}$ is a correct refinement of an ASM $M$ if and only if for each $\widetilde{M}$-run $\widetilde{S}_0, \widetilde{S}_1, \ldots$, there is an $M$-run $S_0, S_1, \ldots$ and sequences $i_0 < i_1 < \ldots$ and $j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv \widetilde{S}_{j_k}$ 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 $i_0 < i_1 < \ldots$ and $j_0 < j_1 < \ldots$ are infinite.

The states $S_{i_k}$ and $\widetilde{S}_{j_k}$ are the corresponding states of interest. They represent the end points of the corresponding computation segments for which the equivalence is defined in terms of a relation between their corresponding locations (those of interest). Fig. 1 shows a run of a refined machine $\widetilde{M}$ and a corresponding run of the abstract machine $M$.

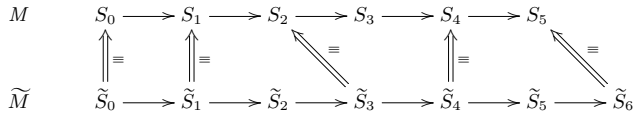



Fig. 1: Börger's refinement ($i_0 = 0$, $i_1 = 1$, $i_2 = 2$, $i_3 = 4$, , $i_4 = 5$ and $j_0 = 0$, $j_1 = 1$, $j_2 = 3$, $j_3 = 4$, , $j_4 = 6$)

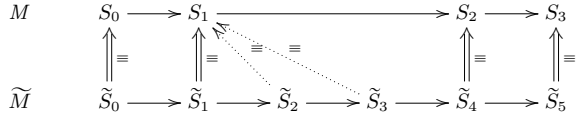



Fig. 2: Stuttering refinement ($j_0 = 0$, $j_1 = 1$, $j_2 = 4$, $j_3 = 5$)

### 2.1.1 Stuttering refinement

The definition of refinement given in Def. 1 is very general. It is applicable for hand-made mathematical proofs, but it is difficult to embed it into a prover for automatic refinement correctness proof, and to find proof patterns. For this reason, in the following, we give a restricted notion of correct model refinement. It does not consider pieces of corresponding runs, but it considers models' runs in the whole.

**Definition 2 (Stuttering refinement).** An ASM $\widetilde{M}$ is a correct *stuttering refinement* of an ASM $M$ if and only if for each $\widetilde{M}$-run $\widetilde{S}_0, \widetilde{S}_1, \ldots$, there is an $M$-run $S_0, S_1, \ldots$ and sequence $j_0 < j_1 < \ldots$ such that $j_0 = 0$ and, for each $k = 0, 1, \ldots$, it yields $\bigwedge_{r=j_k}^{j_{k+1}-1} S_k \equiv \widetilde{S}_r$ and either

- both runs terminate and their final states are the last pair of equivalent states; or
- both runs and sequence $j_0 < j_1 < \ldots$ are infinite.

This notion of refinement is a particular case of the definition of refinement given in Def. 1. It requires that any refined state $\widetilde{S}_j$ is equivalent with an abstract state $S_i$, and its successor state $\widetilde{S}_{j+1}$ is equivalent with either $S_i$ or with $S_{i+1}$. Therefore, each state is a state of interest, and indexes $j_k$ indicate those positions where the equivalence relation must be checked against a new abstract state.

Fig. 2 shows the equivalence relation between states of a run of the refined machine and a run of the abstract machine. Refined states $\widetilde{S}_0$, $\widetilde{S}_1$, $\widetilde{S}_4$, and $\widetilde{S}_5$ are linked with abstract states $S_0$, $S_1$, $S_2$, and $S_3$. Refined states $\widetilde{S}_2$ and $\widetilde{S}_3$, that are not linked with the next abstract state, are conformant with the previous linked state $S_1$.

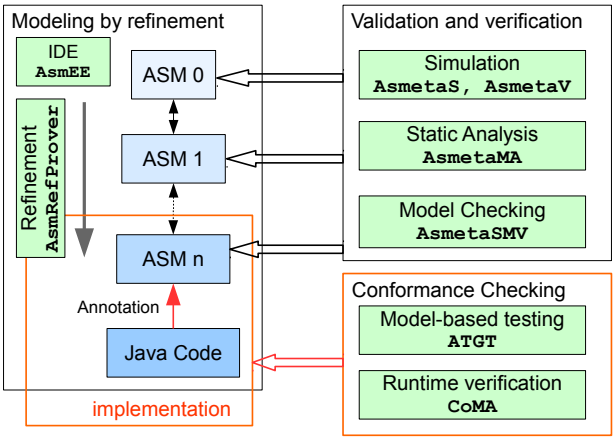



Fig. 3: ASM-based development process

This notion of refinement provides us with two main advantages: it trivially preserves invariant properties and it allows us to automatically prove correctness of model refinement (see Section 3). Indeed, in order to prove that $\widetilde{M}$ refines $M$, it suffices to prove that, for every couple of states $S$ of machine $M$ and $\widetilde{S}$ of machine $\widetilde{M}$ together with their next states $S'$ and $\widetilde{S}'$, the following property holds:

$$\widetilde{S} \equiv S \rightarrow (\widetilde{S}' \equiv S \vee \exists S' : \widetilde{S}' \equiv S') \qquad (1)$$

We have developed an automatic technique able to prove refinement correctness[1].

## 3 Development process and supporting tools

The concepts of ground model and model refinement bring to the definition of a rigorous process for ASM-based system development. The process is depicted in Fig. 3: the modeling activity is complemented with a number of other activities on models and on model/code that help to develop a correct system in a correct way. A set of tools exists to support the developer in the various activities and to make the ASM method useful in practice. Tools are part of the ASMETA (ASM mETA-modeling) framework[2] [9], and are strongly integrated in order to permit reusing information about models during different development phases.

We here explain the fundamental activities (and the related tools) of the process, while in the following sections we show their application to the LGS case study.

The process of *requirements capture* usually 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 high-level *ground model* which is specified using domain-specific terms and can be understood by all stakeholders. This ground model is *abstract*, i.e., it avoids irrelevant details necessary later for the implementation, but *correct*, i.e., it reflects the intended initial requirements, and *consistent*, i.e., it removes ambiguities and incompleteness of the initial requirements. The ground model may not need to be *complete*, i.e., it may leave some given requirements unspecified. These requirements may be captured later.

The IDE `AsmEE` is available to assist the user when editing an ASM model by using the concrete syntax `AsmetaL` (see Section 4).

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. Oftentimes – as it was for the LGS case study –, the requirements are presented in an incremental way, and the chain of refined models reflects this increase of details. Each time a model is specified as refinement of an abstract one, refinement correctness should be checked. This can be done by hand, but we provide an automatic way to achieve this assurance in case of stuttering refinement (see Def. 2). The tool `ASMRefProver` automatically checks stuttering refinement between two ASM models (see Section 2.1.1 for more details).

Modeling activity is supported, at each level of refinement, by model *validation* and *verification* (V&V). Model validation should be applied, already at ground model level, 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. ASM model validation (see Section 6.1) is possible by means of the model simulator `AsmetaS` [30] and by the validator `AsmetaV` [19] that allows to build and execute *scenarios* of expected system behaviors. A further validation technique is *model review* (a form of static analysis) to determine if a model has sufficient *quality* attributes (as minimality, completeness, consistency). Automatic ASM model review is possible by means of the `AsmetaMA` tool [3] (see Section 6.2).

Validation usually precedes 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. Formal verification of ASMs is possible by means of the model checker `AsmetaSMV` [2] (see Section 6.3). *Computa-*

[1] `http://asmeta.sourceforge.net/download/asmrefprover.html`

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

*tion Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas can be proved on models.

When the system is implemented in an actual code, either derived from the model as last low-level refinement step, or externally provided, also *conformance checking* (see Section 7) is possible. Both *model-based testing* and *runtime verification* can be applied to check if the implementation conforms to its specification. We support conformance checking w.r.t. Java code. The tool `ATGT` [29] can be used to automatically generate tests from ASM models[3] and, therefore, to check the conformance *offline*; `CoMA` [4], instead, can be used to perform runtime verification, i.e., to check the conformance *online*.

In the following sections, we present the application of the proposed process to the Landing Gear System case study, whose description can be found in [14] and in the same volume of this paper. We present in sequence all the proposed activities, together with supporting techniques and tools, although they should be used iteratively.

## 4 Ground model

The first activity of the process consists in writing a simple abstract state machine representing the ground model of our modeling effort. Note that among the possible views proposed in the informal requirements – functional, architectural, real time, reliability, etc. – we make some simplifications, in order to keep the presentation concise. From the functional view, we abstract from the analogical switch and the pressure sensor, while, from the architecture view, we simplify the digital architecture by only considering *one* computing module. We also abstract from the sensor voting mechanism that, however, has been separately analyzed in [7].

The ASMETA framework provides the user with a language, `AsmetaL`, its syntax checker, and `AsmEE`, an IDE (Integrated Development Environment) embedded within eclipse as plug-in. Using `AsmEE`, the user can edit ASM models and access all the tools presented in the following sections. `AsmEE` features include syntax highlighting and *new model* wizard; a screenshot is shown in Fig. 4.

An ASM model is structured into three sections, as shown in Code 1: a *header* in which external models can be imported and the signature is declared, a *body* in which functions, domains, and rules are defined, and an *init* which initializes the machine.

The ground model of the LGS, shown in Code 2 and developed according to the template in Code 1, models the doors and the gears, and how their statuses change. The model does not contain a representation for valves, cylinders, sensors, and the health monitoring. Function `doors` represents the status of the doors that can be

[3] Note that sequences generated by `ATGT` could be used to test programs written in any programming language.

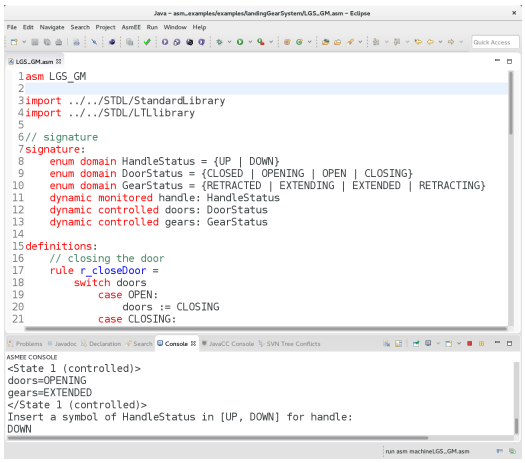



Fig. 4: `AsmEE`, the ASMETA Eclipse Environment

```
asm model
// header
import StandardLibrary
signature:
    dynamic monitored value: Integer
    ...

// body
definitions:
    function ...

    rule r_open = ...

    main rule r_main = ...

// initial state
default init s0:
    ...
```

Code 1: A template for an ASM

`OPEN`, `CLOSED`, `OPENING` 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* [14] is executed, and, instead, as long as the `handle` is `DOWN`, the *outgoing sequence* [14] is executed. Let us 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.

```
asm LGS_GM

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 2: Ground model

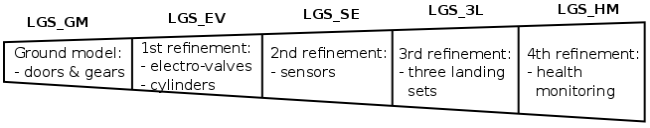



Fig. 5: Models chain

In the model, 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`).

## 5 Modeling by refinement

In this section we present the four steps of the refinement process for modeling the case study[4]. Fig. 5 depicts the relationship existing between the models and, for each model, the system elements introduced with respect to the previous model. The refinement process has been guided by the LGS requirements, that in [14] are presented with an increasing level of detail.

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

We start from the high-level description (the ground model described in Section 4) of the system *core*, i.e., one landing set whose behavior is captured in terms of user inputs and doors' and gears' alleged state. Then, we refine the model by adding the behavior of the actuators: electro-valves and cylinders. In the second refinement, the sensors are added. The third 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 each step, we automatically prove the refinement correctness using the `AsmRefProver`. It exploits an SMT solver and the symbolic representation of ASM states and the relation (transition rules) between two consecutive states presented in [8]. By instantiating an SMT theory with both $S$ and $\widetilde{S}$ together with their generic successor states $S'$ and $\widetilde{S}'$, `AsmRefProver` can prove (or disprove) the validity of property 1.

In all the refinement steps, the equivalence between an abstract and a refined state is defined in terms of all the controlled locations which are declared at both levels of refinement. They act as locations of interest.

```
asm LGS_EV

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

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

    function cylGears =
        switch gears
            case RETRACTED: CYL_RETRACTED
            case EXTENDING: CYL_EXTENDING
            case EXTENDED: CYL_EXTENDED
            case RETRACTING: CYL_RETRACTING
        endswitch
```

```
    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 3: First refinement: cylinders and electro-valves

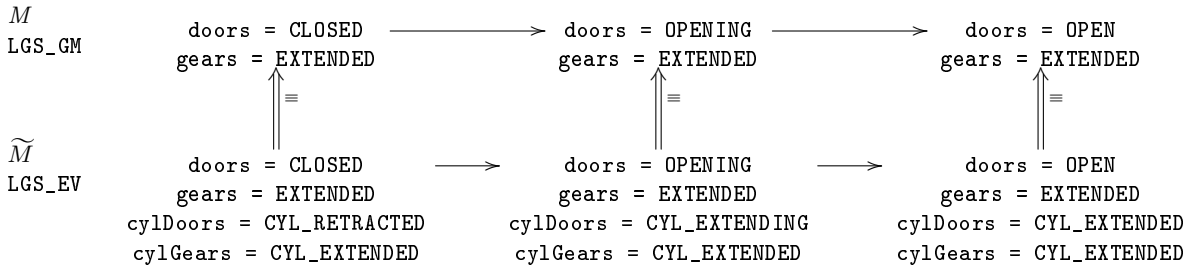



Fig. 6: First refinement – Example of refined run

### *5.1 First refinement: adding the electro-valves and the cylinders*

In this model, LGS_EV, we have refined the ground model by adding the representation of the electro-valves and of the cylinders. Code 3 shows the new elements introduced in the model. We have added the functions for the general electro-valve (`generalEV`), and for 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 `cylDoors` and `cylGears` 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.

*Correctness of the model refinement* The model LGS_EV is a correct stuttering refinement of LGS_GM. If a state $\widetilde{S}$ of LGS_EV is conformant with a state $S$ of LGS_GM, a step in the refined model leads to a state $\widetilde{S}'$ that is conformant with $S'$, the next state of $S$ in the abstract model. Indeed, the refined machine only extends the signature of the abstract model, but does not modify the transition relation. Fig. 6 shows an example of refined run.

### *5.2 Second refinement: adding the sensors*

The model LGS_SE presented in this section extends the model described in Section 5.1 by adding the modeling of the sensors. Code 4 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 LGS 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.

```
asm LGS_SE

signature:
    ...
    dynamic monitored doorsOpen: Boolean
    dynamic monitored doorsClosed: Boolean
    dynamic monitored gearsExtended: Boolean
    dynamic monitored gearsRetracted: 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 4: Second refinement: sensors

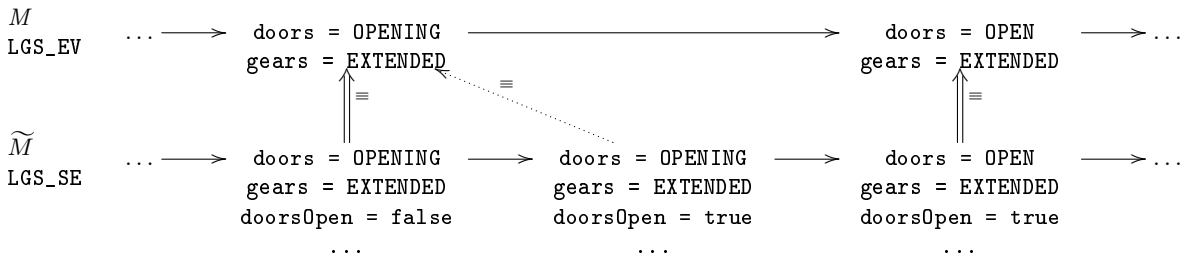



Fig. 7: Second refinement – Example of refined run

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 4 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).

Sensor values are computed through a voting mechanism that has been modeled and analyzed in [7]. In this paper, we abstract from the voting mechanism and we assume to observe the values computed by the mechanism. 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 4). 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.

*Correctness of the model refinement* The model LGS_SE is a correct stuttering refinement of LGS_EV. In each state, the refined machine can move to a state in which the doors status or the gears status are either changed (if the sensors detect the changing) or unchanged (if the sensors do not detect any change). Therefore, if a state $\widetilde{S}$ of the refined model LGS_SE is conformant with a state $S$ of the abstract model LGS_EV, a step in the refined model can lead to a state $\widetilde{S}'$ that is either conformant with $S$ (if the sensors do not detect any change) or with the next state $S'$ of the abstract model (if the sensors detect the changing). Fig. 7 shows an example of refined run.

### *5.3 Third refinement: adding the three landing sets*

The model LGS_3L presented in this section extends the model described in Section 5.2 by adding the modeling of the three landing sets. Code 5 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

```
asm LGS_3L

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 5: Third refinement: three landing sets

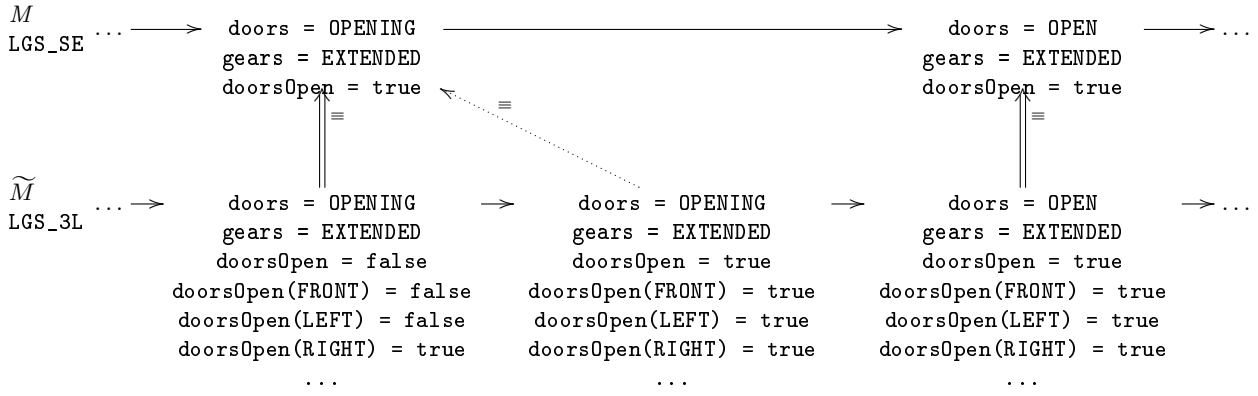



Fig. 8: Third refinement – Example of refined run

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 LGS_SE 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.

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 model LGS_3L is a correct stuttering refinement of LGS_SE. In the previous refinement step, we have already proved that the introduction of the sensors produces a correct stuttering refinement. In this step, we have only modified the policy to read the sensors: a sensor type is considered activated if all the sensors on the three landing sets are activated. Fig. 8 shows an example of refined run.

### *5.4 Fourth refinement: adding the health monitoring system*

The model LGS_HM presented in this section extends the model LGS_3L described in Section 5.3, by adding the modeling of the health monitoring system (Section 4.3 of the case study in [14]). 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 [7]. Since the analogical switch and the pressure sensor are not considered in this work, we do not model their monitoring.

Code 6 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

```
asm LGS_HM

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 6: Fourth refinement: failure mode

the `timeout` has elapsed and the electro-valve responsible for the doors opening is turned on and the doors are not open (`openDoorsEV and not(doorsOpen)`), 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 derived functions `aDoorOpen`, `aDoorClosed`, `aGearExtended`, and `aGearRetracted` that signal if at least one of the corresponding sensors is turned on.

*Correctness of the model refinement* The model LGS_HM is a correct stuttering refinement of LGS_3L. LGS_HM implements two modes: normal mode and failure mode (when an anomaly is detected). In normal mode, the refined model behaves as LGS_3L: in this case the refinement proof is straightforward. When the model enters failure mode, it does not modify the status of the doors and of the gears anymore: therefore, we need to prove that such behaviors (i.e., runs) are also allowed by the abstract model. The refined model can enter in failure mode only when the doors or the gears are moving. In the abstract model, when the doors or the gears are moving, they change their status only if the corresponding sensors are turned on (e.g., if the `doors` are `OPENING`, they become `OPEN` only when the sensor `doorsOpen` is seen turned on). Therefore, whenever the refined model LGS_HM is in a state $\widetilde{S}$ conformant with a state $S$ of the abstract model LGS_3L in which the sensors are turned off and it moves to a state $\widetilde{S}'$ in failure mode, then $\widetilde{S}'$ is conformant with $S'$, the next state of $S$. Fig. 9 shows an example of refined run in which the refined model enters failure mode.

[5] In the case study [14], 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.*

## 6 Validation and verification

We here describe all the validation and verification activities we have performed on the developed specifications. Each activity has been performed on all the models (all the refinements), unless stated otherwise.

### 6.1 Simulation

`AsmetaS` permits to perform either *interactive simulation*, where required inputs are provided interactively by the user during simulation, and *random simulation*, where inputs values are chosen randomly by the simulator itself. The simulator, at each step, performs *consistent updates checking* to check that all the updates are consistent: in an ASM, two updates are inconsistent if they update the same location to two different values at the same time [18]. Moreover, at each step the simulator also checks that all the invariants hold. If an invariant specified over the controlled part of the model does not hold, the simulation is interrupted because a state violating the desired property has been reached: this is a signal of a fault in the model (or in the invariant). Instead, if during an interactive simulation an invariant specified over the monitored part of the model (i.e., the

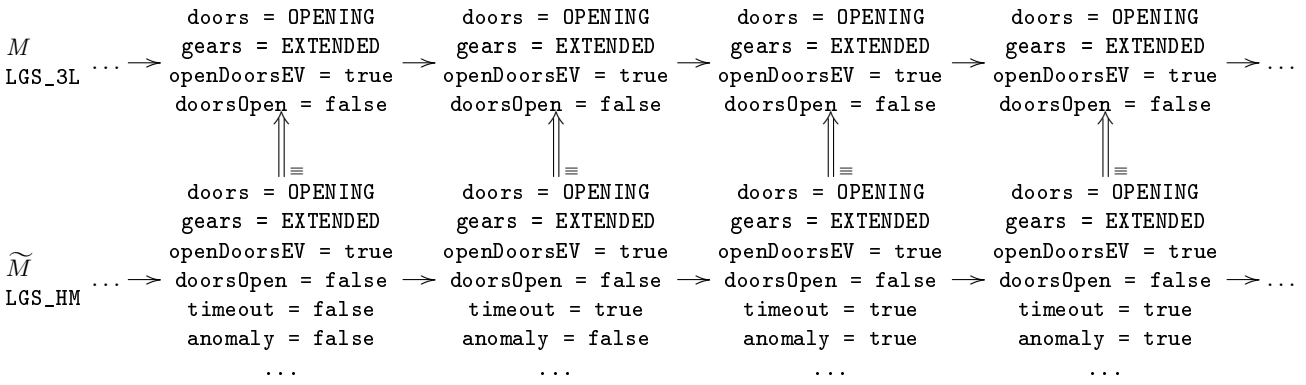



Fig. 9: Fourth refinement – Example of refined run

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

Code 7: Wrong ground model – Error in `r_retractionSequence`

```
scenario lgsGround1
load LGS_GM.asm

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

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 8: Scenario reproducing the simulation that leads to the error

environment) is violated, the simulator asks for another (valid) set of monitored values: in this case, the user has provided a set of inputs that cannot be observed in the environment (e.g., sensors `doorsClosed` and `doorsOpen` both turned on at the same time).

*Interactive* In an interactive simulation, at each step the user is asked for the values of the monitored functions.

By interactive simulation we were able to identify an error in a preliminary version of the ground model LGS_GM (see Code 2). Fig. 10 shows the simulation trace of the wrong model (shown in Code 7). The error was due to the fact that, during a retraction sequence, the `gears` became `EXTENDED` instead of `RETRACTED`. Fig. 11 shows the simulation, over the correct ground model, of the complete retraction sequence.

*Random* In random simulation, the simulator itself randomly chooses the values for monitored functions. Such kind of simulation is particularly useful because it does not require the user intervention and possibly permits to find consistency violations (i.e., inconsistent updates) and/or invariant violations with ease.

*Scenario-based validation* In scenario-based validation the designer provides a set of scenarios specifying the expected behavior of the models (using the textual notation `Avalla`). These scenarios are used for validation by instrumenting the simulator `AsmetaS`. During simulation, `AsmetaV` captures any check violation and, if none occurs, it finishes with a *PASS* verdict. `Avalla` provides constructs to express execution scenarios in an algorithmic way as interaction sequences consisting of actions committed by the user to `set` the environment (i.e., the values of monitored/shared functions), to `check` the machine state, to ask for the `execution` of certain transition rules, and to enforce the machine itself to make one `step` (or a sequence of steps by command `step until`) as reaction of the actor actions.

We have built several scenarios describing different configurations of the LGS. For example, Code 8 shows the scenario for the ground model 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. Such scenario reproduces the situation that brings to the error described previously. The scenario execution consists in a simulation, similar to that seen in Fig. 10. However, the simulation is not interactive, since the values of the monitored functions are set according to the values specified

```
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. 10: 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. 11: Simulation of the correct ground model – Complete retraction sequence

```
<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. 12: Execution of the scenario shown in Code 8 over the wrong ground model

in the scenario. Moreover, the scenario execution also checks the specified assertions. Fig. 12 shows the output of the scenario execution over the faulty ground model; we can notice that, in the fourth step, the specified assertion has been violated. We have later executed the scenario over the correct model and all the assertion 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.

### *6.2 Static analysis*

The aim of *model review* is to determine if a model is of sufficient *quality* to be easy to develop, maintain, and enhance. This technique allows to identify defects early in the system development, reducing the cost of fixing them, so it should be applied also on models just sketched. The `AsmetaMA` tool [3] (based on `AsmetaSMV`) allows *automatic* review of ASMs. Typical vulnerabilities and defects that can be introduced during the modeling activity using ASMs are checked as violations of suitable *meta-properties* (*MPs*, defined in [3] as CTL formulae). The violation of a meta-property means that a quality attribute (minimality, completeness, consistency) is not guaranteed, and it may indicate the presence of an actual fault (i.e., the ASM is indeed faulty), or only of a *stylistic defect* (i.e., the ASM could be written in a better way). An inconsistent update (meta-property MP1), for example, is a signal of a real fault in the model; the presence of functions that are never read nor updated (meta-property MP7), instead, may simply indicate that the model is not minimal, but not that it is faulty.

During the development process, we have detected several deficiencies of our models. Most of them regarded minimality violations and indicated some stylistic violations, but not real faults. For example, in the first refinement LGS_EV (see Code 3), the model advisor signals that functions `cylDoors` and `cylGears` are never read (violation of meta-property MP7), and, therefore, are useless. Indeed, in that refinement step 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.

However, we have also found some faults in our models. For example, a preliminary version of the ground model LGS_GM contained an error, as shown in Code 7 (the same error we found by simulation in Section 6.1). Indeed, during a retraction sequence, the `gears` became `EXTENDED` instead of `RETRACTED`. The model advisor has discovered two meta-property violations for the same model:

- $MP_5$ 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.

### *6.3 Property verification*

As further analysis activity, we have verified the specifications through model checking. `AsmetaSMV` [2] is a tool that translates ASM specifications into models of the NuSMV model checker, and thus it allows the verification of *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulae. Invariants specified over the controlled part of the model are translated as *invariant specifications* that specify properties that must always hold in the NuSMV model; invariants specified over the monitored part of the model, instead, are translated as *invariant constraints* that constraint the set of reachable states in the NuSMV model.

We have verified the requirements reported in [14]. Each requirement has been specified as an LTL property and proved as soon as possible in the chain of refinements, i.e., in the first model describing all the elements involved in the requirement.

*Ground model* In the ground model LGS_GM (see Code 2) 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 $R_{22}$.

```
-- specification G ( G handle = UP ->
F (doors = CLOSED & gears = RETRACTED)) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 3.1 <-
doors = CLOSED
gears = EXTENDED
handle = DOWN
-> State: 3.2 <-
handle = UP
-> State: 3.3 <-
doors = OPENING
-- Loop starts here
-> State: 3.4 <-
doors = OPEN
-> State: 3.5 <-
gears = RETRACTING
-> State: 3.6 <-
gears = EXTENDED
```

Fig. 13: Counterexample showing the falsification of property $R_{12}bis$ over the faulty ground model

```
g(g(handle = DOWN) implies
        f(gears = EXTENDED and doors = CLOSED)) //R11bis
g(g(handle = UP) implies
        f(gears = RETRACTED and doors = CLOSED)) //R12bis
g(g(handle = DOWN) implies x(g(gears != RETRACTING))) //R21
g(g(handle = UP) implies x(g(gears != EXTENDING))) //R22
```

For example, requirement $R_{12}bis$ requires that *when the command line is working (normal mode), if the landing gear command handle has been pushed UP and stays UP, then eventually the gears will be locked retracted and the doors will be seen closed.* In the preliminary faulty ground model (see Code 7), such property is falsified since the gears never become `RETRACTED`. As counterexample, the model checker returns the trace shown in Fig. 13. Note that this fault has also been discovered with simulation and model review. However, more complicated faults may be difficult to find with those validation techniques and only verification could reveal them.

*LGS_EV* In the first refinement LGS_EV (see Code 3), we have been able to verify five more normal mode requirements: $R_{31}$, $R_{32}$, $R_{41}$, $R_{42}$, and $R_{51}$.

```
g((extendGearsEV or retractGearsEV) implies doors = OPEN) //R31
g((openDoorsEV or closeDoorsEV) implies
        (gears = RETRACTED or gears = EXTENDED)) //R32
g(not(openDoorsEV and closeDoorsEV)) //R41
g(not(extendGearsEV and retractGearsEV)) //R42
g((openDoorsEV or closeDoorsEV or
      extendGearsEV or retractGearsEV) implies generalEV) //R51
```

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.*

*LGS_SE* The introduction of the sensors in the second refinement LGS_SE (see Code 4) did not allow us to

prove any further requirement, but required us to add some constraints to the model. The sensors are used to check that the doors have become open/closed and that the gears have become extended/retracted. Temporal properties for $R_{11}bis$ and $R_{12}bis$ are falsified in the first refinement because there are some execution runs in which the sensors are never turned on. Therefore, we need to add to the model five *justice* constraints (also called *fairness* constraints), specifying that the sensors must be true infinitely often.

```
JUSTICE gearsExtended
JUSTICE gearsRetracted
JUSTICE doorsClosed
JUSTICE doorsOpen
JUSTICE gearsShockAbsorber
```

Note that the need to modify some temporal properties or to add some fairness constraints has also been discussed in [32], where the ProB model checker is used to prove properties over Event-B specifications of the LGS.

*LGS_HM* In the fourth refinement LGS_HM, we have modified the justice constraints because we want to model sensor misbehaviors when they keep their values unchanged. However, we cannot simply remove the justice constraints. We must still assure that each sensor is repeatedly turned on unless the timeout occurs (and therefore the anomaly is detected). An exception is the `gearsShockAbsorber` which is not subject to timeout. The modified constraints follow.

```
JUSTICE gearsExtended or timeout
JUSTICE gearsRetracted or timeout
JUSTICE doorsClosed or timeout
JUSTICE doorsOpen or timeout
JUSTICE gearsShockAbsorber
```

For this refinement, 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 $R_{74}$.

```
g((openDoorsEV and aDoorClosed and timeout) implies
                                        x(g(anomaly))) //R61
g((closeDoorsEV and aDoorOpen and timeout) implies
                                        x(g(anomaly))) //R62
g((retractGearsEV and aGearExtended and timeout) implies
                                        x(g(anomaly))) //R63
g((extendGearsEV and aGearRetracted and timeout) implies
                                        x(g(anomaly))) //R64
g((openDoorsEV and not(doorsOpen) and timeout) implies
                                        x(g(anomaly))) //R71
g((closeDoorsEV and not(doorsClosed) and timeout) implies
                                        x(g(anomaly))) //R72
g((retractGearsEV and not(gearsRetracted) and timeout) implies
                                        x(g(anomaly))) //R73
g((extendGearsEV and not(gearsExtended) and timeout) implies
                                        x(g(anomaly))) //R74
```

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.*

The introduction of the health monitoring system has falsified the properties specified for requirements $R_{11}bis$, $R_{12}bis$, $R_{21}$, and $R_{22}$, requiring that, if the `handle` remains `UP` (or `DOWN`), the `doors` and the `gears` will reach a given configuration and never assume other configurations. Those properties were specified for a model without the health monitoring system, so assuming that the system always evolves; in the current refinement, since the occurrence of an anomaly blocks the evolution of the system, those properties do not hold anymore. Therefore, in this model we need to specify that those requirements hold only in normal mode, as follows:

```
g(g(handle = DOWN and not(anomaly)) implies
        f(gears = EXTENDED and doors = CLOSED)) //R11bis
g(g(handle = UP and not(anomaly)) implies
        f(gears = RETRACTED and doors = CLOSED)) //R12bis
g(g(handle = DOWN and not(anomaly)) implies
                              x(g(gears != RETRACTING))) //R21
g(g(handle = UP and not(anomaly)) implies
                               x(g(gears != EXTENDING))) //R22
```

## 7 Conformance checking

In this section we show the last step of the development process, i.e., the development of an implementation for the LGS, and the checking of the conformance of the implementation w.r.t. the specification. We propose two approaches for checking the conformance: an *offline* approach based on testing, and an *online* approach based on runtime verification.

Code 9 shows the Java implementation that has been developed as last step of low-level refinement starting from LGS_HM. The program is implemented as a `TimerTask`, i.e., a task that can be scheduled to be repeatedly executed with a given frequency. In our case, we execute the program (i.e., the `run` method) every half a second. The program retrieves the values from the sensors (by the method `getSensorValues`) and, if no anomaly has been previously found, executes a check of the system and updates its internal state according to the position of the `handle` (by the method `checkAndUpdate`).

### *7.1 Linking Java code and ASM specifications*

Linking a Java code with its ASM formal specification permits to establish a conformance relation between the implementation the its model. In the following, we provide an informal description of a technique we have proposed in [4] to link ASM to Java code.

We use *Java annotations* to establish this link; Java annotations are meta-data tags that can be used to add some information to code elements as class declarations, field declarations, etc. In addition to the standard ones, annotations can be defined by the user similarly as classes. For our purposes, we have defined a set of annotations [4]. The retention policy (i.e., the way to signal how and when the annotation can be accessed) of all our annotations is *runtime*: annotations can be read by the

```
import org.asmeta.monitoring.*;

@Asm(asmFile = "models/LGS_HM.asm")
public class LandingGearSystem extends TimerTask {
    private GearStatus gearsStatus;
    private DoorStatus doorsStatus;
    @Monitored(func = "doorsOpen", args = {"FRONT"})
    boolean doorsOpenFront;
    @Monitored(func = "doorsOpen", args = {"LEFT"})
    boolean doorsOpenLeft;
    @Monitored(func = "doorsOpen", args = {"RIGHT"})
    boolean doorsOpenRight;
    ...
    @Monitored(func = "sensorTimeout")
    boolean sensorTimeout;
    private boolean anomaly;
    private boolean generalEV;
    ...

    public LandingGearSystem() { ... }

    @Override
    public void run() {
        getSensorValues();
        checkAndUpdate();
    }

    private void getSensorValues() { //sensor values retrieval
    }

    @RunStep
    public void checkAndUpdate() {
        if (!anomaly) {
            healthMonitoring();
            if(handle == HandleStatus.UP)
                moveHandleUp();
            else
                moveHandleDown();
        }
    }

    private void moveHandleUp() { ... }
    private void moveHandleDown() { ... }
    private void healthMonitoring() { ... }

    @MethodToFunction(func = "doors")
    public DoorStatus getDoorsStatus() {
        return doorsStatus;
    }

    @MethodToFunction(func = "gears")
    public GearStatus getGearsStatus() {
        return gearsStatus;
    }

    public static void main(String[] args) {
        LandingGearSystem lgs = new LandingGearSystem();
        new Timer().scheduleAtFixedRate(lgs, 500, 500);
    }
}
```

Code 9: Java implementation of the Landing Gear System

compiler and by any program through reflection. In the tools developed for supporting our model-based testing and runtime verification approaches, we read the annotations in order to discover the relation between the ASM and the Java code.

In order to link a Java class with its corresponding ASM specification, first the class must be annotated with the annotation `@Asm`, having the path of the ASM model as string attribute (`asmFile`). The Java class `LandingGearSystem` (Code 9) is linked to the ASM specification `LGS_HM` (Code 6).

Then the class data must be connected with the signature of the ASM. A field of the Java class can be connected with a function/location of the ASM, through the field annotation `@FieldToFunction`; the annotation has a mandatory attribute `func` for specifying the function name, and an optional attribute `args`, for specifying the arguments' values (if one wants to connect the field to a specific location). Moreover, it is also possible to link a pure method[6] with a function/location, using the method annotation `@MethodToFunction`, having the same attributes of `@FieldToFunction`. In our case, pure methods `getDoorsStatus` and `getGearsStatus` are respectively linked to functions `doors` and `gears`. Linked fields (those annotated with `@FieldToFunction`) and linked methods (those annotated with `@MethodToFunction`) constitute the *observed Java state*. In the case study, the observed Java state is given by the methods `getDoorsStatus` and `getGearsStatus`.

In a Java class some fields take their values from input streams (e.g., file, socket, etc.). Such fields can be linked to monitored functions using the annotation `@Monitored` which has the same attributes of `@FieldToFunction`. In this way, the monitored functions mimic the behavior of the input streams (they act as mock objects). In our case study, all the fields representing the sensor values are connected with the sensor locations; for example, the field `doorsOpenFront` is linked with the location `doorsOpen(FRONT)`.

Finally, the execution of the Java code must be linked with an execution (i.e., a run) of the ASM. The annotation `@StartMonitoring` is used to select one constructor[7] which builds the desired observed initial state of the object. The annotation `@RunStep`, instead, permits to identify a method (called *changing* method) that changes the observed state, i.e., the values of the linked fields and the return values of the linked pure methods[8]. In our case study, method `checkAndUpdate` is a changing method.

Both linked constructors and linked changing methods can have some parameters, that can be linked to the ASM as well. The annotation `@Param` can be used to link parameters to monitored functions/locations of the ASM; similarly to `@FieldToFunction`, it has a mandatory attribute `func` and an optional attribute `args` to identify an ASM function/location. In our case study,

[6] Pure methods are side effect free methods with respect to the object/program state. They return a value but do not assign values to fields.

[7] We do not consider the default constructor. If the class does not have any constructor, the user has to specify an empty constructor and annotate it with `@StartMonitoring`.

[8] The user can identify several changing methods, but, in this case, each changing method must be linked with a different monitored value by the two annotation attributes `setFunction`, specifying the name of a 0-ary monitored function of the ASM model, and `toValue`, specifying a value of the function codomain: `setFunction` must identify the same function in all the annotations, while `toValue` must assume different values.

the constructor and the changing methods do not have parameters.

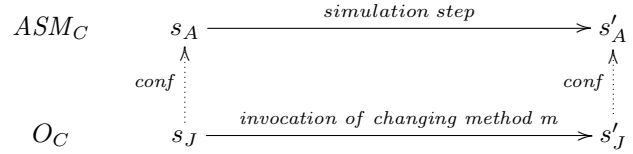
### *7.2 State and step conformance*

The linking previously described allows the following notion of conformance between an instance $O_C$ of a class $C$ and the ASM specification $ASM_C$ linked to $C$.

**Definition 3 (State conformance).** We say that a state $s_J$ of $O_C$ *conforms* to a state $s_A$ of $ASM_C$, i.e., $conf(s_J, s_A)$, if all the observed elements of $C$ (fields annotated with `@FieldToFunction` and methods annotated with `@MethodToFunction`) have values in $O_C$ conforming to the values of the functions in $ASM_C$ linked to them.

Intuitively, the Java state and the ASM state are conformant, if the values of the linked fields and the values returned by linked pure methods are equal to the values of the corresponding ASM functions/locations.

**Definition 4 (Step conformance).** Given the execution of a *changing* method $m$ (i.e., a method annotated with `@RunStep`) and a step of simulation of the ASM, we say that the Java step ($s_J$, $s'_J$) and the ASM step ($s_A$, $s'_A$) are conformant if $conf(s_J, s_A)$ and $conf(s'_J, s'_A)$.




Intuitively, a Java object is step conformant with the corresponding ASM specification, if their states are conformant before and after the changing method execution and the ASM simulation step.

### *7.3 Offline testing*

Model-based conformance testing [33,43] of reactive systems consists in taking benefit from the model for mechanizing both test data generation and verdicts computation (i.e., to solve the oracle problem). In offline approaches, test suites are pre-computed from the model and stored under a format that can be later executed on the System Under Test (SUT). The model can be used both to guide the test generation, in order to discover which aspects of the model must be covered, and to decide when to stop testing, when coverage of the model has reached a certain level.

A classical technique to generate tests from models exploits the use of model checkers. In this case, the model of the system is translated to the language of the model checker, and a suitable property (called *trap property*) is proved false by the model checker by means of a counterexample. This counterexample represents a possible system behavior and it can be translated to a test through a concretization process.

For ASMs, we have developed a tool, called `ATGT` [29], which is capable of generating tests from ASMs following several testing criteria [28], like rule coverage, update rule coverage, parallel rule coverage, etc. For example, a test suite satisfies the *rule coverage* criterion if, for every rule $r_i$, there exists at least one state in a test sequence in which $r_i$ fires, and there exists at least one state in a test sequence (possibly different from the previous one) in which $r_i$ does not fire.

#### 7.3.1 Test generation

We have used `ATGT` to generate tests from model LGS_HM (see Code 6), using the basic rule coverage (BRC) and the update rule coverage (URC). BRC requires that every rule is executed at least once, while URC requires that every update is executed at least once without being trivial, i.e., by actually changing the value of the location that it updates. For every coverage goal (e.g., a rule to execute in BRC), `ATGT` computes a *test predicate* which is a predicate over the state of the machine, representing the condition that must be reached to cover that particular goal. For instance, the basic rule coverage of the update rule doors := OPENING (when the doors are CLOSING) in rule `r_retractionSequence` is specified by the following test predicate:

`BR_r_Main_TTBR_r_retractionSequence_T_CLOSING_T1`: not(anomaly) and handle = UP and gears != RETRACTED and doors = CLOSING

`ATGT` has derived, for the entire specification, 116 test predicates (62 for the BRC and 54 for the URC). For every test predicate *tp*, `ATGT` has built, if possible, an abstract test sequence, which is a valid sequence of states, leading to a state where *tp* becomes true. `ATGT` exploits the SPIN model checker and its capability to produce counterexamples upon property violations. If a test predicate cannot be covered, we say that it is *unfeasible* and it means that there is no valid system behavior that can cover that case. Unfeasible test predicates must be discarded and no longer considered. For the LGS, we found no unfeasible test predicates.

In order to reduce the test suite size, `ATGT` can perform a coverage evaluation of the tests, by checking if a test sequence, generated for a test predicate, unintentionally covers also other test predicates. Without coverage evaluation, `ATGT` produces 116 test sequences (one for each test predicate), while, with coverage evaluation, `ATGT` produces only 22 test sequences.

#### 7.3.2 Test concretization

We here use a technique, originally introduced in [5], that derives, from each abstract test sequence *ATS*, a concrete Java test (a JUnit test), consisting of a sequence of

method calls with suitable checks (i.e., asserts). The test concretization leverages the linking between the Java class and the ASM (see Section 7.1), and the definitions of state conformance (Def. 3) and step conformance (Def. 4).

The technique first identifies the constructor annotated with `@StartMonitoring`, builds an instance of the class, and associates it to the reference variable `sut`. For example, given a class `C` whose constructor without parameters is annotated with `@StartMonitoring`, the produced statement is

$$\text{C sut} = \textbf{new C}();$$

If the constructor has some parameters, these must be annotated with `@Param`. The technique identifies the actual parameters to use in the object instantiation by reading, in the first state of the abstract test sequence, the values of the monitored functions that are linked with the parameters.

Then, for each state of the *ATS*, three sets of instructions are added to the test: the setting of the monitored fields, the execution of the changing method, and the checking of the conformance of the observed state.

**Monitored variables setting** Before calling the changing method, the inputs of the program must be set. Fields annotated with `@Monitored` are updated with the value of the function linked in the annotation. For example, if a field `mf` is linked to a function whose value is `v` in the current state, the following statement is built:

$$\text{sut.mf} = \text{v};$$

**Step execution** If there is only one changing method `cm`, that method is called as follows:

$$\text{sut.cm}();$$

Otherwise, if there are several changing methods, the value `v` of the monitored function/location linked in the `@RunStep` annotations identifies what method must be called (the method having value `v` in the annotation argument `toValue`). In our case study, there is only one changing method (i.e., `checkAndUpdate`) that, therefore, is always called. The (possible) actual parameters in the method invocation are fixed by the values of the monitored functions/locations linked in the `@Param` annotations of the method formal parameters.

**Observed state checking** After each method invocation (and after the object instantiation), the oracle is built, exploiting the annotations `@FieldToFunction` and `@MethodToFunction`:

- given an observed function/location (i.e., linked with the annotation `@FieldToFunction`/`@MethodToFunction`), we obtain its value `v` from the *ATS*;
- if the annotation annotates a field `of`, we build the following assertion

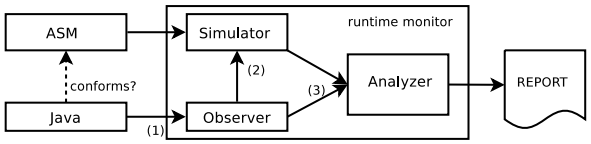



Fig. 15: The CoMA runtime monitor for Java

$$\text{assertEquals(v, sut.of)};$$

which states that the value of `sut.of` must be equal to `v`; if the values are not equal, the test fails and a conformance violation has been found;
- if the annotation annotates a pure method `om`, we build an assertion as follows:

$$\text{assertEquals(v, sut.om())};$$

which states that the value returned by `sut.om()` must be equal to `v`; as before, if the values are not equal, the test fails and a conformance violation has been found.

Fig. 14 shows the translation of the ATS built for covering the test predicate `BR_r_Main_TTBR_r_retractionSequence_T_OPENING_TT1` in a JUnit test case. The abstract test sequence and the test are splitted in the three iterative replicated phases. Note that not all controlled functions are linked to the observed Java state; therefore, some controlled functions of the ASM state are not considered in the conformance checking phase.

For the LGS case study, we have translated the 22 test sequences in JUnit test cases; the execution of the test suite covers 96.9% of the program instructions and 72.3% of the branches.

### *7.4 Runtime verification*

Although a model-based testing approach as that described in Section 7.3 can give enough confidence that the implementation is correct, for safety-critical systems as the LGS, we may want to continue checking the conformance of the implementation with respect to its specification also after the deployment.

In [4], we proposed CoMA, a runtime verification approach for Java code using ASMs. The schema of the proposed runtime framework is shown in Fig. 15. The monitor is composed of: an *observer* that evaluates when the Java (observed) state is changed (1), and leads the abstract ASM to perform a machine step (2), and an *analyzer* that evaluates the step conformance between the Java execution and the ASM simulation (3). When the monitor detects a violation of conformance, it reports the error. It can also produce a trace in form of counterexample, which may be useful for debugging. Note that the use of CoMA can be twofold, since also faults in the specification can be discovered by monitoring the software. For instance, by analysing and re-executing counterexamples, faults in the model can be exposed.

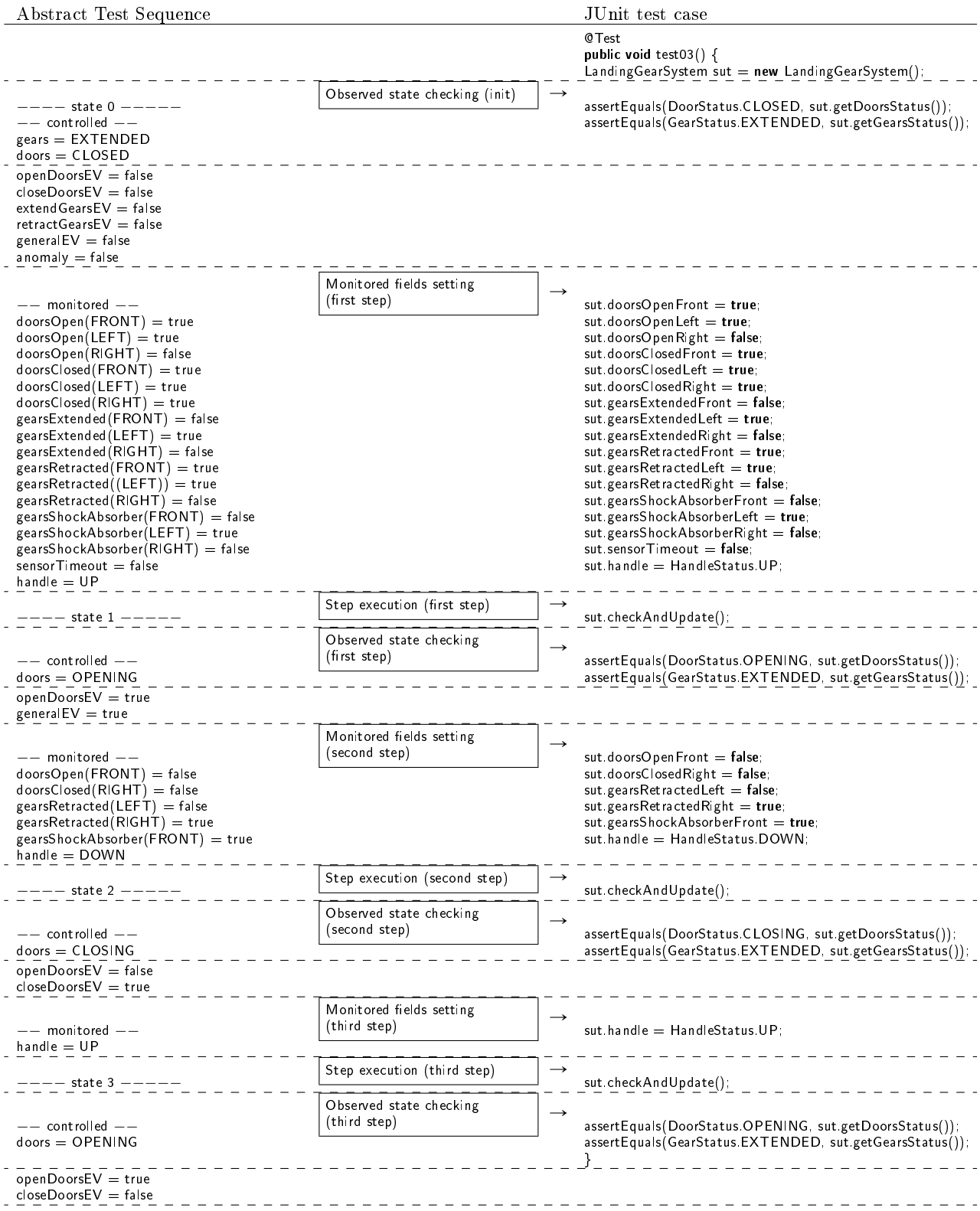



Fig. 14: Test concretization of the ATS obtained for the test predicate `BR_r_Main_TTBR_r_retractionSequence_T_CLOSING_T1`

| Technique | Total test length | Coverage | | Fault detection | | |
|---|---|---|---|---|---|---|
| | | statement | branch | F1 | F2 | F3 |
| Offline testing | 93 steps (22 tests) | 96.9% | 72.3% | NO | YES | YES |
| CoMA with random inputs | 50 | 55.6% | 31.1% | 2% | 12% | 69% |
| | 500 | 75.2% | 52.4% | 2% | 81% | 100% |
| | 5000 | 83.2% | 60.3% | 3% | 100% | 100% |
| | 50000 | 83.4% | 60.5% | 4% | 100% | 100% |

Table 1: Coverage and fault detection

The technique exploits the linking described in Section 7.1 and the definitions of state conformance (Def. 3) and step conformance (Def. 4). In the following, we give the definition of runtime conformance.

**Definition 5 (Runtime conformance).** Given a Java class $C$, an object of $C$ $O_C$, and and ASM specification $ASM_C$, we say that $C$ is runtime conforming to $ASM_C$ if the following conditions hold:

1) the initial state $s_J^0$ of the computation of $O_C$ conforms to *one and only one* initial state $s_A^0$ of the computation of $ASM_C$, i.e., $\exists!\, s_A^0$ initial state of $ASM_C$ such that $conf(s_J^0, s_A^0)$;
2) for every Java step $(s_J, s'_J)$ induced by the execution of a changing method $m$, $\exists!\, (s_A, s'_A)$ step of $ASM_C$ with $s_A$ the current state of $ASM_C$, such that the two steps are conformant.

The runtime framework has been implemented using AspectJ [35]. By means of an *aspect*, AspectJ allows to specify different *pointcuts*, i.e., points of the program execution one wants to capture. For each pointcut, it is possible to specify an *advice*, i.e., the actions that must be executed when a pointcut is reached (*before* or *after* the execution of the code specified by the pointcut). In our runtime framework, we have defined some pointcuts for identifying the instantiation of a class under monitoring (when a constructor annotated with `@StartMonitoring` is called) and the execution of a changing method (i.e., a method annotated with `@RunStep`). Moreover, for each pointcut we have defined an advice actually implementing the monitoring:

- when a monitored object is instantiated, the corresponding advice creates an instance of the ASM simulator `AsmetaS`;
- when a changing method is executed, the corresponding advice forces a step of simulation of the ASM, and it checks the conformance between the obtained Java state and the ASM states that can be reached in one step.

### *7.5 Experimental results*

We have executed some experiments for measuring the coverage and the fault detection achieved by the model-based testing approach and the runtime verification approach. For model-based testing, we have used the test suite obtained in Section 7.3.2, consisting of 22 tests having a total length of 93 steps. For runtime verification, we have executed the monitored class for an increasing number of steps (50, 500, 5000, and 50000), randomly choosing the sensors' values, and allowing the timeout to occur only in the last ten steps; since sensors' values are randomly chosen, each experiment of runtime verification has been executed for 100 runs and the results are the average over the runs.

Table 1 reports the results of the experiments. We have measured the statement and the branch coverage of both techniques over a correct implementation of the case study. Offline testing has better statement and branch coverage than CoMA (with any setting): this outcome is expected, since in model-based testing the tests are derived with the aim of maximizing the coverage over the model and, usually, there is a direct relationship between the coverage of the model and the coverage of the implementation.

For measuring the fault detection, we have used three faulty versions of the implementation that we actually produced during the development process: version `F1` wrongly retrieves the values coming from the sensors related to door opening, version `F2` wrongly updates the status of the gears, and version `F3` does not contain a `break` statement in a `switch` statement. Offline testing can detect faults `F2` and `F3`, but not `F1`. Different runs of CoMA can either find or not find a given fault. Fault `F1` is particularly hard to find, and CoMA (with the best setting) detects it only 4% of the times; note that, however, offline testing is not able to discover `F1` because the produced tests do not exercise the code necessary for discovering the conformance deviation. The other two faults, instead, can always be discovered by CoMA with a sufficient number of steps.

## 8 Related work

Abstract State Machines (ASMs) are a formal method that has been successfully applied for the high-level design and analysis in different system areas: definition of industrial standards for programming and modeling

languages [41], design and re-engineering of industrial control systems, modeling e-commerce and web services, modeling cloud and adaptive systems, design and analysis of protocols, architectural design, language design, verification of compilation schemes and compiler backends, etc. Due to the multiplicity of applications, we prefer to refer to [18] for a complete introduction on the ASM method and the presentation of the great variety of its successful applications.

A comparison of the ASM method with other formal methods is out of the scope of this paper; we refer to [31] for some comparisons between the ASMs and different state-based formal methods. However, we take advantage of the landing gear case study to show some analogies/differences of our ASM-based development process w.r.t. other formal methods frameworks and identify possible future improvements of our process.

The ASMs method provided us a mathematical founded, yet easy to use, notation for reasoning on the requirements of the system and developing a correct by construction implementation of the case study. Moreover, the ASMETA framework allowed us to perform several validation and verification activities at all refinement levels. Another ASM formalization of the LGS has been proposed in [34]. In that work, ASMs have been used to carefully analyze the requirements and find possible inconsistencies; however, the lack of tool-support for model validation and verification only allowed to make handmade proofs of the requirements. Tool-support for ASMs modeling is instead also provided by CoreASM [27], an extensible plugin-based framework that provides simulation, debugging [23], and aspect orientation [22] facilities.

The use of the refinement approach helped us to manage the complexity of the case study. Actually, the refinement was guided by the LGS requirements themselves, that in [14] are presented with an increasing level of detail. The particular kind of refinement we consider (i.e., stuttering refinement) allowed us to automatically prove each refinement step with the SMT-based tool `AsmRefProver`. Formalizations of the LGS based on refinement have been proposed in [32,42,37,38] using the Event-B method. The main difference between the notion of refinement in ASMs and in Event-B is that refinement correctness proof is a central notion of the Event-B method [1], whereas in ASMs refinement is a modeling methodology and its correctness proof is performed after the modeling phase [39].

In the first stages of model development, we found very useful to simulate our models to understand if we were developing what we had in mind. Different simulators are available also for other formal notations. The ProB tool, for example, has been successfully applied to the simulation (and the model checking) of the LGS case study in the Event-B method [32]; with respect to our simulator `AsmetaS`, ProB has the advantage of also providing an animator that permits to visually represent the simulation: this feature is particularly useful since it does not force the user to read a (possibly long) simulation trace and can be understood by all the stakeholders.

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), are 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 temporal properties can be directly expressed as part of the ASM model itself.

Model checking of temporal properties for the LGS has been also executed in [32,38] over Event-B specifications using the model checker ProB that, as our model checker `AsmetaSMV`, supports LTL and CTL properties.

Model checking usually suffers from the well-known state explosion problem [21]. Note that our translation from ASM to NuSMV introduces an overhead that usually even worsens the state explosion problem; as future work, in addition to the optimization of the translation from ASMs to NuSMV, we plan to apply some abstractions directly at the level of the ASM model, instead of applying common abstractions (e.g., cone of influence [21]) to the obtained NuSMV model. An approach that tries to mitigate the state explosion problem has been used in [25], where the requirements of the LGS case study have been specified with the *Context Description Language* and verified over Fiacre specifications of the system, using *context-aware model checking*.

We have not been able to verify real-time properties. Although reactive timed ASMs [40] have been proposed for dealing with time in ASMs, they are not supported by our tools for model analysis. Therefore, for normal mode requirements $R_1$ (see Section 5.1 in [14]), 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 [14]). Real-time properties of the LGS could be instead verified in [13] over Fiacre specifications using the model checker Tina. A formalization of the LGS case study with an explicitly representation of time is also provided in [10] using hybrid Event-B machines.

As novelty aspect, our formal process does not only focus on the modeling and the verification of the requirements, but also comprises the development of the implementation (seen as last refinement step): we provide a technique to link the implementation with the specification, and we check the conformance using model-based testing and runtime verification. In most approaches deal-

ing with runtime verification of software, the required behavior of the system is specified by means of correctness properties [24,26]: temporal logic-based formalisms are very popular in runtime verification for specifying these properties [20], especially variants of linear temporal logic [12]. Instead, the use of *operational* notations (as ASMs) for runtime verification has not been investigated so deeply [11,36]. However, we claim that operational specifications offer some advantages with respect to declarative specifications of properties, especially when designers are more accustomed to them: as we have shown for the case study, they allow to perform validation and verification at the early stages of system development and permit to trace, by means of a step-wise model refinement, the relation between the specification and the implementation.

## 9 Conclusions

The paper presents the application of an ASM-based rigorous development method to the Landing Gear System (LGS) case study [14]. A chain of refined ASM models is presented: starting from a high level view of the system (i.e., the *ground model*), more detailed models have been obtained through *refinement*. As last step of the refinement process, a Java implementation for the LGS has been developed. Throughout all the development process, different validation and verification activities have been performed, as simulation, model review, and model checking. Moreover, each refinement step has been proved correct using an SMT-based approach. The correctness of the implementation w.r.t its formal specification, instead, has been proved by means of two techniques: model-based testing and runtime verification.

As future work, we plan to improve the support to refinement. In particular, we want to provide the user with a framework in which the refinement decisions can be documented and the refinement proof obligations can be easily derived from the models (now the user must specify the locations over which the refinement proof obligations must be built). Moreover, we plan to add some animation facilities to our simulator and support infinite-state model checking.

## References


1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
2. 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.
3. 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), NASA/CP-2010-216215*, pages 4–13. NASA, 2010.
4. P. Arcaini, A. Gargantini, and E. Riccobene. CoMA: Conformance monitoring of Java programs by Abstract State Machines. In S. Khurshid and K. Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 2012.
5. P. Arcaini, A. Gargantini, and E. Riccobene. Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Workshops Proceedings, Luxembourg, March 18-22, 2013*, pages 178–187. IEEE, 2013.
6. P. Arcaini, A. Gargantini, and E. Riccobene. Modeling and Analyzing Using ASMs: The Landing Gear System Case Study. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 36–51. Springer International Publishing, 2014.
7. P. Arcaini, A. Gargantini, and E. Riccobene. Offline Model-Based Testing and Runtime Monitoring of the Sensor Voting Module. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 95–109. Springer International Publishing, 2014.
8. P. Arcaini, A. Gargantini, and E. Riccobene. Using SMT for dealing with nondeterminism in ASM-based runtime verification. *ECEASST*, 70, 2014.
9. 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.
10. R. Banach. The Landing Gear Case Study in Hybrid Event-B. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 126–141. Springer International Publishing, 2014.
11. M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.
12. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software and Methodology (TOSEM)*, 20, 2011.
13. B. Berthomieu, S. Dal Zilio, and Ł. Fronc. Model-Checking Real-Time Properties of an Aircraft Landing Gear System Using Fiacre. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 110–125. Springer International Publishing, 2014.
14. F. Boniol and V. Wiels. The Landing Gear System Case Study. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 1–18. Springer International Publishing, 2014.

15. F. Boniol, V. Wiels, Y. A. Ameur, and K.-D. Schewe. *ABZ 2014: The Landing Gear Case Study Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z Toulouse, France, June 2-6, 2014, Proceedings*. Springer International Publishing, 2014.
16. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
17. E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Proceedings of Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005*, volume 3717 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2005.
18. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
19. 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.
20. F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Methods and Software Engineering*, volume 3308 of *LNCS*, pages 357–372. Springer Berlin / Heidelberg, 2004.
21. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
22. M. Dausend and A. Raschke. Introducing Aspect-Oriented Specification for Abstract State Machines. In Y. Ait Ameur and K.-D. Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, volume 8477 of *Lecture Notes in Computer Science*, pages 174–187. Springer Berlin Heidelberg, 2014.
23. M. Dausend, M. Stegmaier, and A. Raschke. Debugging Abstract State Machine Specifications: An Extension of CoreASM. In F. Mazzanti and G. Trentanni, editors, *Proceedings of iFM 2012 & ABZ 2012 - Posters & Tool demos Session*, pages 21–25, 2012.
24. N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.
25. P. Dhaussy and C. Teodorov. Context-Aware Verification of a Landing Gear System. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 52–65. Springer International Publishing, 2014.
26. Y. Falcone, K. Havelund, and G. Reger. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 141–175. IOS Press, 2013.
27. R. Farahbod and U. Glässer. The CoreASM modeling framework. *Softw., Pract. Exper.*, 41(2):167–178, 2011.
28. A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science*, 7:262–265, 2001.
29. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003*, volume 2589 of *Lecture Notes in Computer Science*, pages 263–277. Springer Berlin Heidelberg, 2003.
30. A. Gargantini, E. Riccobene, and P. Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *Journal of Universal Computer Science*, 14(12):1949–1983, 2008.
31. U. Glässer, S. Hallerstede, M. Leuschel, and E. Riccobene. Integration of Tools for Rigorous Software Construction and Analysis (Dagstuhl Seminar 13372). *Dagstuhl Reports*, 3(9):74–105, 2013.
32. D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, and M. Leuschel. Validation of the ABZ Landing Gear System Using ProB. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 66–79. Springer International Publishing, 2014.
33. R. Hierons and J. Derrick. Editorial: special issue on specification-based testing. *Software Testing, Verification and Reliability*, 10(4):201–202, 2000.
34. F. Kossak. Landing Gear System: An ASM-Based Solution for the ABZ Case Study. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 142–147. Springer International Publishing, 2014.
35. R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.
36. H. Liang, J. Dong, J. Sun, and W. Wong. Software monitoring through formal specification animation. *Innovations in Systems and Soft. Eng.*, 5:231–241, 2009.
37. A. Mammar and R. Laleau. Modeling a Landing Gear System in Event-B. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 80–94. Springer International Publishing, 2014.
38. D. Méry and N. K. Singh. Modeling an Aircraft Landing System in Event-B. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 154–159. Springer International Publishing, 2014.
39. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *Journal of Universal Computer Science*, 7(11):952–979, 2001.
40. A. Slissenko and P. Vasilyev. Simulation of Timed Abstract State Machines with predicate logic model-checking. *Journal of Universal Computer Science*, 14(12):1984–2006, 2008.
41. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.
42. W. Su and J.-R. Abrial. Aircraft Landing Gear System: Approaches with Event-B to the Modeling of an Industrial System. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear*

*Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 19–35. Springer International Publishing, 2014.
43. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2006.
44. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, Oct. 2009.