Offline Model-based Testing and Runtime Monitoring of the Sensor Voting Module

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

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

Abstract. Formal specifications are widely used in the development of safety critical systems, as the Sensor Voting Module of the Landing Gear System. However, the conformance relationship between the formal specification and the concrete implementation must be checked. In this paper, we show a technique to formally link a Java class with its Abstract State Machine formal specification, and two approaches for checking their conformance: an offline model-based testing approach and an online runtime monitoring approach.

1 Introduction

For safety critical components, formal verification and validation of models must be combined with the validation of the implementation. The user wants to gain confidence that the system has been implemented as specified, i.e., it *conforms* to its requirements. Indeed, regardless the correctness of the model (guaranteed by formal verification, simulation and so on), the implemented system must be validated itself. As aptly stated by Ed Brinksma in his 2009 keynote at the Dutch Testing Day and Testcom/FATES, "Who would want to fly in an airplane with software proved correct, but not tested?".

We here focus on the model-driven design and validation of the sensor voting module (SVM) in a landing gear system [7]. A sensor voting system, similar to that presented in our case study, is verified in [16] using the UML Verification Environment. In this paper, we describe the validation activity of a Java implementation of the SVM, using the Abstract State Machines (ASMs) as formal language. For a complete description of the modeling of the whole landing gear system case study using ASMs through the ASMETA framework, we refer to [5].

In this paper, we first introduce the SVM case study, and give a brief introduction to two conformance validation techniques, model-based testing and runtime monitoring, reporting some related literature (Section 2). Then, we show the ASM model for the SVM and which activities the designer should perform, even before the conformance checking is started, to be sure that the model is correct (Section 3.1). We then implement the SVM in Java (Section 3.2) and we validate it against the ASM model. We describe how the formal specification

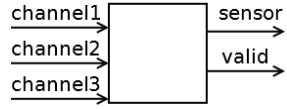


Fig. 1: Sensor Voting Module Interface

can be linked with the implementation (Section 3.3), and then we present the application of model-based testing (Section 4) and of runtime monitoring (Section 5) to the case study. Finally, we compare the strengths and the weaknesses of the two approaches through some experiments (Section 6), and conclude the paper in Section 7.

2 Background

2.1 The sensor voting module

The Landing Gear System (LGS) has proposed in the ABZ conference as a real-life case study [7] with the aim of showing how different formal methods can be used for the specification, design and development of a complex system.

In the LGS the state of the equipments (i.e., doors and gears) is computed by a set of discrete sensors; the digital part of the landing gear system takes decisions and sends commands (e.g., stimulating the electro-valves) relying on the sensor values. In order to prevent sensor failures, each sensor value is based on the values of three *micro-sensors* [7]; a sensor receives the values of the three micro-sensors from three channels. The duty of the Sensor Voting Module (SVM) is to select one of these three values according to the following policy.

Let X be a sensor and $X_i(t)$ $(i = 1, 2, 3)$ the values for X received at time t:

- If at t the three channels are considered as valid and are equal, then the value considered by the control software is this common value.
- If at t one channel is different from the two others for the first time (i.e., the three channels were considered as valid up to t), then this channel is considered as invalid and is definitely eliminated. Only the two remaining channels are considered in the future. At time t, the value considered by the control software is the common value of the two remaining channels.
- If a channel has been previously eliminated, and if at t the two remaining channels are not equal, then the sensor is definitely considered as invalid.

We can represent an SVM by the black box reported in Fig. 1. It has three inputs corresponding to the three channels for the sensor and two outputs: one that represents the value of the sensor and one that informs whether the sensor is valid or invalid.

2.2 Model-based off-line testing

Model-based conformance testing [13,17] 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 off-line 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 (also 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.

MBT for ASM For ASMs, we have developed a tool, called ATGT [11], which is capable of generating tests from ASMs following several testing criteria [10], 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.

2.3 Runtime Monitoring

According to [14], *runtime monitoring* (also *runtime verification*) is "the discipline of computer science that deals with the study, development, and application of those monitoring techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property".

The aim of runtime monitoring is to check that the observed executions of a system ensure some correctness properties. Runtime monitoring is a *lightweight* verification technique that, considering the ability to detects faults, can be classified halfway between those techniques that try to ensure universal correctness of systems – as model checking and theorem proving – and those techniques like testing that ensure the correctness only for a fixed set of executions.

The main difference with techniques like model checking is that, whereas these techniques check all possible executions of a program, runtime monitoring only checks those executions that are actually performed by the program under scrutiny. So, it is possible that, although the program contains a fault, its executions never produce a failure that evidences that fault.

The main difference with testing, instead, is that the number of executions over which the program is checked is not fixed. Sometimes, runtime monitoring is seen as the process of testing the system *forever* [14], since, as in testing, the actual output is checked with respect to an expected output (usually described by an *oracle*), but, unlike testing, every execution of the system is checked.

Finally, whereas traditional validation and verification activities are only executed *offline*, i.e., before the deployment, runtime monitoring can also be executed *online*, i.e, after the deployment of the program.

In order to describe the expected correctness properties, several formalisms have been used in literature as, for example, temporal logics [12,6], extended regular expressions [8], and Z specifications [15].

Coma: conformance monitoring between ASMs and Java In [3] we propose *CoMA*, runtime *Conformance Monitoring* of Java code *by ASM specifications*. The CoMA monitor allows *online* monitoring, namely it considers executions in an incremental fashion. It takes as input an executing Java software system and an ASM formal model. The monitor observes the behavior of the Java system and determines its correctness w.r.t. the ASM specification working as an oracle of the expected behavior. While the software system is executing, the monitor checks conformance between the observed state and the expected state.

2.4 Comparing Offline Testing and Runtime Monitoring

Offline testing is much simpler than runtime monitoring: once the tests are generated, they can be easily reused as long as the model does not change. The test generation time may be an issue, especially if the model is large and the model checker takes a lot of resources for test generation; however, efficient test generator tools can generate tests also for big models. Once the tests are obtained, they can be launched and, if the SUT passes all the tests, the tester can be confident that the implementation is correct and therefore the system can be deployed.

However, the system could strongly depend on the environment in which it is executed [9]. If such environment is not available at testing time or, although available, it is not practically possible to interact with it (because maybe too much time consuming), testing the system could become difficult. In unit testing this problem is sometimes mitigated by using *mock objects* that mimic the behavior of the environment: nonetheless, if the actions of the environment are not fully predictable, also using mock objects could be not useful. Moreover, safety-critical systems as medical devices, aircraft flight controls, nuclear systems, etc., although tested and verified deeply, could require an additional degree of confidence that they behave as expected. Runtime monitoring here acts as a double check that everything goes well [14].

Furthermore, in the presence of nondeterministic systems, an MBT approach, as that described in Section 4, is not suitable because it is not able to correctly judge the implementation output: the implementation could deviate from a test case, taking a different but valid execution path, and the test case would *falsely* fail. For such kind of systems, a runtime monitoring approach able to deal with nondeterminism, as that described in Section 5, can also benefit the testing [4].

3 Specification and implementation of the SVM

The following sections describe the ASM model (Section 3.1) and the Java implementation (Section 3.2) for the SVM. The ASM and the Java implementation

have been developed independently: once we have agreed upon the interface, one author has developed the ASM and another one the Java code. In this way, the two artifacts may be quite different. Finally, Section 3.3 describes how to link the Java code with the ASM; such linking will be exploited in Section 4 and Section 5 for the testing and the runtime monitoring of the implementation.

3.1 ASM model of the SVM

Code 1 reports the ASM model. The signature of the ASM contains the enumerative domain Channel representing the three input channels of the sensor; one unary monitored function channel represents the signals coming from the three channels. The controlled unary function validCh keeps track if each channel is still valid; in the initial state all the channels are valid. The output value of the sensor is computed by the machine and recorded with the function sensor, while its validity is simply defined as a derived function valid, which is true if there exist two different channels still valid.

In the main rule, if the sensor is not valid, the machine state is no longer updated. Otherwise, if the sensor is valid, the following rules are called:

- r_allValidChannels checks if all the channels are still valid and, in this case, it controls if the values of the three channels are equal. Since the comparison is performed by considering each pair of channels, r_allValidChannels calls r_threeValidChannels three times, in order to actually compare each pair ($vc1 and $vc2); if they are equal, it also checks the third channel ($vc3) and, if necessary, it updates its validity. The sensor value is updated to the majority value of the three channels.
- r_twoValidChannels checks if two channels are still valid, in case the third channel ($nvc) is no longer valid; the rule is called three times, one for each pair of channels. The sensor value is updated only if the two valid channels are equal.

Note that the specification can be easily extended in case there are more than three channels.

Model validation We have performed the following preliminary activities over the ASM model using the framework ASMETA[3], in order to be sure that the model exactly captures the intended behavior of the system. In fact, in model-based testing and in runtime monitoring, it is of extreme importance that the models are correct, otherwise faults in the model jeopardize the entire activity of the implementation validation.

Simulation Through simulation with the ASM simulator AsmetaS, we have simulated the scenarios of a channel becoming invalid and then the entire sensor becoming invalid. Simulation is useful to gain confidence that the specification actually captures the intended behavior. The simulator, at each step, checks that

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

```
asm SensorVotingModule

signature:
    enum domain Channel = {ONE | TWO | THREE}
    dynamic monitored channel: Channel -> Boolean
    dynamic controlled validCh: Channel -> Boolean
    dynamic controlled sensor: Boolean
    derived valid: Boolean

definitions:
    function valid =
        (exist $c1 in Channel, $c2 in Channel with $c1!=$c2 and validCh($c1) and validCh($c2))

    rule r_threeValidChannels($vc1 in Channel, $vc2 in Channel, $vc3 in Channel) =
        if channel($vc1) = channel($vc2) then
            par
                sensor := channel($vc1)
                if channel($vc1) != channel($vc3) then
                    validCh($vc3) := false
                endif
            endpar
        endif

    rule r_allValidChannels =
        if (forall $c in Channel with validCh($c)) then
            par
                r_threeValidChannels[ONE,TWO,THREE]
                r_threeValidChannels[TWO,THREE,ONE]
                r_threeValidChannels[THREE,ONE,TWO]
            endpar
        endif

    rule r_twoValidChannels($nvc in Channel, $vc1 in Channel, $vc2 in Channel) =
        if not(validCh($nvc)) then
            if channel($vc1) = channel($vc2) then
                sensor := channel($vc1)
            else
                par
                    validCh($vc1) := false
                    validCh($vc2) := false
                endpar
            endif
        endif

    invariant over validCh: size({$c in Channel | validCh($c) : $c}) != 1

    main rule r_Main =
        if valid then
            par
                r_allValidChannels[]
                r_twoValidChannels[ONE,TWO,THREE]
                r_twoValidChannels[TWO,THREE,ONE]
                r_twoValidChannels[THREE,ONE,TWO]
            endpar
        endif

default init s0:
    function validCh($c in Channel) = true
```

Code 1: ASM specification of the SVM

all the specified invariants are satisfied. In the model (before the main rule), we have introduced an invariant specifying that it is not possible that only one sin-

```
rule r_twoValidChannels($vc1 in Channel, $vc2 in Channel, $vc3 in Channel) =
    if not(validCh($vc1)) then
        if channel($vc2) = channel($vc3) then
            sensor := channel($vc2)
        else
            par
                validCh($vc1) := false //error
                validCh($vc2) := false
            endpar
        endif
    endif
```

Code 2: Faulty model – Error in the rule r_twoValidChannels

gle channel is valid. The requirements indeed specify that at least two channels must be valid, otherwise the entire sensor must be considered invalid (i.e., all the channels must be considered invalid).

Model Advisor During the development of the model, we have applied the model advisor [2], a tool we developed for looking for common errors that are usually introduced in the model development using ASMs. The model advisor has discovered an error in the model. Code 2 shows the faulty implementation of rule r_twoValidChannels. The model advisor signals that, when the update validCh($vc1) := false is executed, the location validCh($vc1) is always yet *false*. Indeed, the model is faulty and the location that should be updated to *false* is validCh($vc3). We have fixed the error, and we have given more meaningful names to the rule parameters, as shown in Code 1.

Formal Property Verification We have been able to formally prove some properties by using the model checker AsmetaSMV [1]. The first property simply checks that the specified invariant is satisfied in all the states. Indeed, by default AsmetaSMV translates each invariant φ in the Computation Tree Logic (CTL) formula $\mathbf{ag}(\varphi)$.
The following temporal properties have also been proved:

- Once the sensor becomes invalid, then it will always remain invalid in the future:
  CTLSPEC ag(not(valid) implies ag(not(valid)))
- There exists a path in which the sensor eventually becomes invalid:
  CTLSPEC ef(not(valid))
- There exists a path in which the sensor always remains valid:
  CTLSPEC eg(valid)

3.2 Java implementation

Code 3 shows the Java implementation of the SVM. The method computeSensorValue, given the values of the three parameters s1, s2, and s3 (representing the input channels), updates the value of the sensor and marks if the sensor is no more valid (field sensorValid). The boolean array chValid records which channels are still valid. Two methods return the values of fields value and sensorValid.

```
@Asm(asmFile = "models/SensorVotingModule.asm")
public class Sensor {
    private boolean value;
    private boolean sensorValid;
    private boolean[] chValid;

    @StartMonitoring
    public Sensor() {
        sensorValid = true;
        chValid = new boolean[]{true, true, true};
    }

    @RunStep
    public void computeSensorValue(@Param(func = "channel", args={"ONE"}) boolean s1,
                                   @Param(func = "channel", args={"TWO"}) boolean s2,
                                   @Param(func = "channel", args={"THREE"}) boolean s3) {
        if (sensorValid) {
            if (chValid[0] && chValid[1] && chValid[2]) {
                if (s1 == s2 && s2 == s3) {
                    value = s1;
                } else if (s1 != s2 && s2 == s3) {
                    chValid[0] = false; // first channel invalid
                    value = s2;
                } else if (s2 != s1 && s1 == s3) {
                    chValid[1] = false; // second channel invalid
                    value = s3;
                } else {
                    chValid[2] = false; // third channel invalid
                    value = s1;
                }
            } else if (!chValid[0]) {
                if (s2 == s3)
                    value = s2;
                else
                    sensorValid = false;
            } else if (!chValid[1]) {
                if (s1 == s3)
                    value = s1;
                else
                    sensorValid = false;
            } else if (!chValid[2]) {
                if (s1 == s2)
                    value = s2;
                else
                    sensorValid = false;
            }
        }
    }

    @MethodToFunction(func = "sensor")
    public boolean getValue() {
        return value;
    }

    @MethodToFunction(func = "valid")
    public boolean isValid() {
        return sensorValid;
    }
}
```

Code 3: Java implementation of the SVM

3.3 Linking Java code and ASM specifications

Linking a Java code with its ASM formal specification permits to establish a conformance relation between the ASM and the implementation. In the following, we provide an informal description; a complete description of the technique with all the formal definitions can be found in [3].

We use *Java annotations* to establish this link; Java annotations are metadata 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 [3]. 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 compiler and by any program through reflection. In the tools developed for supporting our model-based testing and runtime monitoring 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 Sensor (Code 3) is linked to the ASM specification SensorVotingModule (Code 1).

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[4] with a function/location, using the method annotation `@MethodToFunction`, having the same attributes of `@FieldToFunction`. In the presented case study, pure methods getValue and isValid are respectively linked to functions sensor and valid.

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 getValue and isValid.

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[5] which builds the desired observed initial state of the object. The annotation `@RunStep`, instead, permits to identify the 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[6]. Both linked constructors

[4] 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.

[5] 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`.

[6] 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

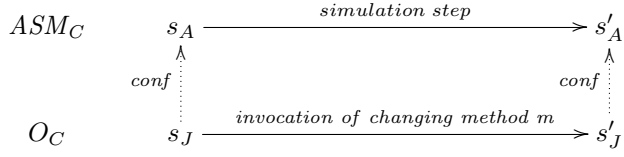
and linked methods can have some parameters, that can be linked to the ASM as well. The annotation `@Param` can be used to link parameters to monitored functions/locations of the ASM; it has a mandatory attribute `func` to specify the name of a monitored function of the ASM model, and an optional attribute `args` to specify the function arguments. In the case study, the parameters of method computeSensorValue are linked to the locations of function channel.

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 1. ***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 methods are equal to the values of the corresponding functions/locations.

Definition 2. ***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.

4 Offline testing

4.1 Test generation

We have used ATGT to generate tests from the SVM model, 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

ASM model, and `toValue`, specifying a value of the function codomain. `setFunction` should have the same value in all the annotations, while `toValue` must assume different values.

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 in the inner conditional rule of rule r_threeValidChannels is specified by the following test predicate.

BR_r_threeValidChannels_TTT21:
valid and (validCh(ONE) and validCh(TWO) and validCh(THREE)) and
(channel(ONE) = channel(TWO)) and (channel(ONE) != channel(THREE))

ATGT has derived, for the entire specification, 38 test predicates (20 for the BRC and 18 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 SVM, 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 38 test sequences, while, with coverage evaluation, ATGT produces only 11 test sequences.

4.2 Test concretization

We devise a novel technique that derives a concrete Java test, consisting of a sequence of method calls with suitable checks (i.e., asserts), from each abstract test sequence *ATS*; in this work, we automatically build JUnit tests. The test concretization leverages the linking between the Java class and the ASM (see Section 3.3) and the definitions of state conformance (Def. 1) and step conformance (Def. 2).

First, it 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 C sut = **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.

The procedure that identifies the inputs in the *ATS* and maps them in method invocations with values for their parameters exploits the Java annotations `@RunStep` and `@Param`. For each state of the *ATS*, the method annotated

```
----- state 0 -----
 -- controlled --
valid = true
 -- monitored --
channel(ONE) = false
channel(TWO) = false
channel(THREE) = true
----- state 1 -----
 -- controlled --
sensor = false
valid = true
```

(a) Abstract test sequence

```
@Test
public void test() {
    // state 0
    Sensor sut = new Sensor();
    assertEquals(true, sut.isValid());
    sut.computeSensorValue(false, false, true);
    // state 1
    assertEquals(false, sut.getValue());
    assertEquals(true, sut.isValid());
}
```

(b) JUnit test case

Fig. 2: Example of test concretization for BR_r_threeValidChannels_TTT21

with `@RunStep` is called[7]. 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. For instance, the formal parameters s1, s2 and s3 of changing method computeSensorValue are connected to the monitored locations channel(ONE), channel(TWO), and channel(THREE).

After each method invocation and after the object instantiation, the oracle is built, exploiting the annotations `@FieldToFunction` and `@MethodToFunction`. For each state of the *ATS*:

- given a function/location linked with an annotation, we obtain its value v from the *ATS*;
- if the annotation annotates a field f, we build an assertion as follows: assertEquals(v, sut.f);
- if the annotation annotates a pure method m, we build an assertion as follows: assertEquals(v, sut.m());

Fig. 2 shows the translation of the ATS built for covering the test predicate BR_r_threeValidChannels_TTT21 (Fig. 2a) in a JUnit test case (Fig. 2b).

5 Runtime Monitoring

Although a model-based testing approach as that described in Section 4 can give enough confidence that the implementation is correct, for safety-critical systems as the sensor voting module, we may want to continue checking the conformance of the implementation with respect to its specification also after the deployment.

[7] 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, since only method computeSensorValue is annotated with `@RunStep`, it is always called.

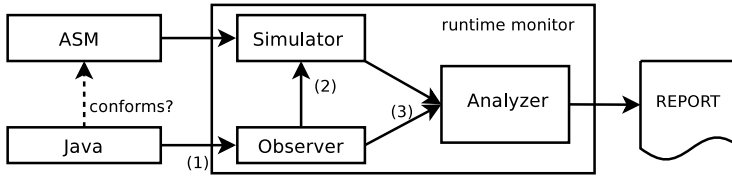


Fig. 3: The CoMA runtime monitor for Java

We propose CoMA [3], a runtime monitoring approach for Java code using ASMs. The schema of the proposed runtime framework is shown in Fig. 3. 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.

The technique exploits the linking described in Section 3.3 and the definitions of state conformance (Def. 1) and step conformance (Def. 2). In the following, we give the definition of runtime conformance.

Definition 3. ***Runtime conformance*** *We say that C is runtime conforming to its specification 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. 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.

6 Experimental comparison

We have executed the 38 Junit tests, obtained as explained in Section 4, and applied CoMA, as explained in Section 5. In CoMA, we have simulated the environment by instantiating 10 times a new sensor and computing 10 times the sensor value by the method `computeSensorValue`, passing three random values as inputs for the three channels. We have measured the code coverage by EclEmma and the mutation score by PIT[8]. In both cases, we found line and branch code coverage of 100%, and mutation score of 57 killed mutants over 74. The not killed mutants involve code inserted by AspectJ and are not relevant for the case study. We can state that both techniques are equivalent regarding detecting faults inserted by the standard PIT mutation operators. However, we have simulated a delayed short circuit fault that causes isValid to return true after 5 times it is called. We have modified the code as follows:

```
int nvCount = 0;
boolean isValid() {
    return valid | nvCount++ > 5;
}
```

The tests produced from the specification do not detect this fault, since the rule coverage of the specification does not imply the coverage of this faulty behavior in the implementation. However, monitoring the code with CoMA exposes the failure by any run in which valid becomes false and isValid is called at least 5 times. In general, we can assume that unforeseen and unspecified anomalous behaviors of the implementation are better detected by runtime monitoring than by MBT.

7 Conclusions

We have presented the model-driven development and validation activity of a critical module in the Landing Gear System. We have applied the formal method of ASMs from the design to the conformance checking of the implementation. We have presented two methodologies for actual system validation (model-based testing and runtime monitoring) and briefly compared them.

References

1. P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In *Abstract State Machines, Alloy, B and Z, 2nd Int. Conference (ABZ 2010)*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2010.

[8] `http://www.eclemma.org/` and `http://pitest.org/`

2. P. Arcaini, A. Gargantini, and E. Riccobene. Automatic Review of Abstract State Machines by Meta Property Verification. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, pages 4–13. NASA, 2010.
3. P. Arcaini, A. Gargantini, and E. Riccobene. 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.
4. P. Arcaini, A. Gargantini, and E. Riccobene. Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism. In *IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 178–187, 2013.
5. P. Arcaini, A. Gargantini, and E. Riccobene. Modeling and analyzing using ASMs: the Landing Gear System case study. In *Proceedings of 4th International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2014) – Case study track*, Communications in Computer and Information Science. Springer, 2014.
6. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software and Methodology (TOSEM)*, 20, 2011.
7. F. Boniol and V. Wiels. Landing gear system `http://www.irit.fr/ABZ2014/landing_system.pdf`. Technical report, ONERA-DTIM, 2014.
8. F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 357–372. Springer Berlin / Heidelberg, 2004.
9. S. Colin and L. Mariani. Run-time verification. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer Berlin / Heidelberg, 2005.
10. A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. *J. Universal Computer Science*, 7:262–265, 2001.
11. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *Proceedings of ASM 2003*, volume 2589 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.
12. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6:158–173, August 2004.
13. R. Hierons and J. Derrick. Editorial: special issue on specification-based testing. *Software Testing, Verification and Reliability*, 10(4):201–202, 2000.
14. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.
15. H. Liang, J. Dong, J. Sun, and W. E. Wong. Software monitoring through formal specification animation. *Innovations in Systems and Software Engineering*, 5:231–241, 2009.
16. C. Mrugalla, O. Robbe, I. Schinz, T. Toben, and B. Westphal. Formal verification of a sensor voting and monitoring UML model. In *Proceedings of the 4th International Workshop on Critical Systems Development Using Modeling Languages (CSDUML 2005)*. Technische Universität München, Sept. 2005.
17. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2006.