International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-025-00812-2

GENERAL

Special Issue: ABZ 2023

Integrating formal specifications in the development and testing of UIs by formal model–view–controller pattern

Andrea Bombarda[1] · Silvia Bonfanti[1] · Angelo Gargantini[1]

Accepted: 29 April 2025

Abstract

Ensuring software functionality, maintainability, and modularity is vital while developing UIs. For this reason, architectural patterns, like the Model–View–Controller (MVC), are usually employed. The MVC aims to separate the representation of information (model) from its presentation to the user (view). One can use a formal model to study the system and the UI, but such formal models are separately developed and analyzed, and the results of the analysis cannot be assured for the actual implementation. To address this problem, we introduced the *formal* MVC pattern (*f*MVC), allowing the integration of `Asmeta` specifications into the model of the MVC-designed software. This paper extends the *f*MVC pattern and the framework to better support Java Swing components and enhance error management, including automatic model state rollback on input failure. Moreover, we propose an extension enabling testers to generate and reuse `Avalla` scenarios for UI validation. We demonstrate the framework's application, covering modeling, validation, and verification at the model level for the AMAN case study that inspired us during the definition of the *f*MVC pattern. Moreover, we show how the AMAN UI can be implemented and tested by our approach.

Keywords Model–view–controller · Abstract state machines · Asmeta · Model-based testing

1 Introduction

The use of formal methods for the development of User Interfaces (UIs) for Human–Computer Interaction (HCI) is not very common, although there are several promising approaches, including the development of new formalisms for UI [14], generation of executable code of UIs from formal model [20, 23], and creation of a formal model of informal design artifacts [10]. Since the most common approaches for developing UIs are based on the use of patterns, and in particular on the use of the Model–View–Controller (MVC) pattern, we decided to evaluate the integration of a formal approach based on the Abstract State Machines (ASM) with the MVC pattern [6]. MVC separates the UI from the data that it must show. To be more precise, MVC describes the architecture of a system of objects, and it can be applied not only to UIs but to entire applications. However, it is also less clearly defined than many other patterns, leaving a lot of latitude for alternate implementations. It is more a philosophy than a recipe [11], and it can be easily adapted and tuned for different use case scenarios. One of the first patterns in the MVC family is the Model–View–Controller pattern introduced at the end of 1970's by Reenskaug for the Smalltalk programming language [12]. In UI development, *Model* objects store, encapsulate, and abstract the data of the application, *View* objects display the information in the model to the user, while *Controller* objects implement the application's actions.

In [6] we have introduced the *formal* MVC pattern (*f*MVC), an extension of the classical MVC where the *Model* is a formal specification, written using Abstract State Machines. This pattern is supported by the `AsmetaFMVCLib` library, which allows the user to link the formal model with the view and the controller by using simple Java annotations. The library is integrated into the `Asmeta` framework [1, 7] and includes the *Model* wrapper (that requires the user only to attach the ASM model) and the *Controller* part, which can be used as it is or extended to be adapted to case-specific

✉ A. Bombarda
andrea.bombarda@unibg.it

✉ S. Bonfanti
silvia.bonfanti@unibg.it

✉ A. Gargantini
angelo.gargantini@unibg.it

[1] Department of Management, Information and Production Engineering, University of Bergamo, Bergamo, Italy

Published online: 12 May 2025

behaviors. Moreover, the library provides an interface to be implemented by the *View* component.

The development of UIs using *f*MVC offers several advantages. First, by using the proposed pattern, users can take advantage of the main peculiarities of formal models, e.g., rigorousness, possibility of properties verification, and iterative development approach by refinement. Moreover, one of the advantages of using Asmeta specifications and, in general, ASMs is that the models are executable and, thus, they can be validated even before having the actual UI by simulation, animation, and scenario-based validation. However, the *f*MVC has suffered from some limitations that we have addressed in this paper. First, when designing the view, the developers can only use limited graphical components, as the others are not supported by our framework. Furthermore, the error handling on input failure missed a clear behavioral definition. For this reason, we have further extended the supported graphical components and embedded a new Asmeta simulator allowing to automatically roll-back the simulator state on input failure. Second, *f*MVC has no way to validate the UIs except to manually execute the UI and check if the behavior is as expected. If the developer has introduced some ad hoc code that changes the behavior with respect to that implemented in the model, or has added to the controller some case-specific instruction which may influence the output, automatically checking the conformance between the formal model and the final executable is of paramount importance. For this reason, we have extended AsmetaFMVCLib by introducing a method to write or automatically derive scenarios for the Asmeta specification representing the *Model* to test the correctness of the specification.

Moreover, in this paper, we apply the proposed pattern to the Arrival Manager (AMAN) case study.[1] We show the whole development process, from the Asmeta model specification, its validation through scenario-based testing and verification, to the linking between the Java View and the Model. Additionally, we exploit the AsmetaFMVCLib library to perform model-based testing on the obtained UI.

The remainder of this paper is structured as follows. Section 2 presents the Asmeta framework, by focusing on the tools that are relevant for this work. In particular, it also reports a simple case study of a calculator we use to show how the proposed methodology can be applied. Then, in Sect. 3, we present the *f*MVC pattern and we describe the operations needed when developing a UI software with it. Section 4 shows the process we have devised allowing for the automatic generation and execution of model-based testing activities on UIs. The application of the *f*MVC approach, both in terms of software development and testing, is described in Sect. 5, while a discussion on the approach and its limitations is given in Sect. 6. Finally, Sects. 7 and 8, respectively, report related work on integrating formal specifications during UI development and conclude the paper.

2 The Asmeta framework

The proposed *f*MVC architecture and model-based testing process rely on the Asmeta framework [1]. The framework offers various tools that can be used in different stages of software development: design, development, and operation (see Fig. 1). Modeling, validation, and verification are the activities performed starting from requirements in the design phase. After requirements are formalized and verified, they can be further refined by adding details to the previous model. Then, development and operation can be performed independently. The former includes code/abstract unit test generation and acceptance test generation from models, while the latter supports runtime simulation and runtime monitoring activities. The Asmeta framework supports the Abstract State Machines (ASMs), a rigorous state-based formal method that updates abstract *states* by employing *transition rules*. Each state is composed of algebraic structures, i.e., domains of objects with functions and predicates defined on them.

An example of Asmeta specification is shown in Code 1. It models the behavior of a simple calculator, SimpleCalc, given in terms of an Abstract State machine. The user can insert a number, modeled by the monitored function number and, then, he/she can increment or decrement the calculator result, represented by the controlled function calc_result, by the entered value (number). The two mathematical operations, INC (increment) and DEC (decrement) are stored in the monitored function math_action. Furthermore, the user can also store the current value of the calculator result in local memory, represented by the controlled function mem. This operation is done when the mem_action is equal to MPLUS. The local memory can be reset by the memory action MRESET.

The specification is composed of four main sections:

• Functions and domains are declared in the signature section. Besides predefined domains provided by the StandardLibrary, static/dynamic user-defined domains can be declared (e.g., MathAction). The interaction with the environment is specified using *monitored* functions (e.g., number), written by the environment and read by the machine, and *out* functions written by the machine and sent as output to the environment. Internal functions are those defined as *controlled* (e.g., mem), which are read in the current state and updated in the next state by the machine. Functions for which a value is not provided in the current state are considered undef.

[1] The case study description is available online at https://sites.google.com/view/abz-aman-casestudy/home.

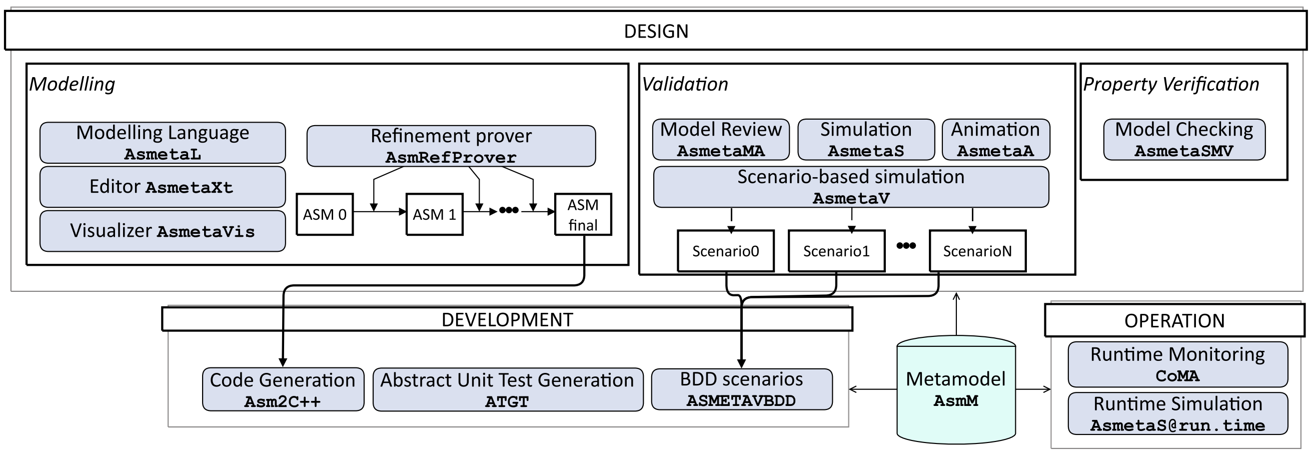

Fig. 1 Phases of The ASM development process powered by the `Asmeta` framework: design, development, and operation

Code 1 An `Asmeta` spec for a simple calculator

```
1  asm SimpleCalc
2  import StandardLibrary
3  signature:
4  // DOMAINS
5  enum domain MathAction = {INC, DEC}
6  enum domain MemoryAction = {MPLUS, MRESET}
7  // FUNCTIONS
8  monitored number: Integer
9  monitored math_action: MathAction
10 monitored mem_action: MemoryAction
11 controlled calc_result: Integer
12 controlled mem: Integer
13 definitions:
14 // INVARIANTS
15 invariant inv_res over calc_result: calc_result >= 0
16 invariant inv_action over math_action, mem_action:
       math_action = undef xor mem_action = undef
17 // MAIN RULE
18 main rule r_Main =
19 par
20         if math_action = INC then calc_result := calc_result
               + number
21     endif
22         if math_action = DEC then calc_result := calc_result
               − number
23     endif
24         if mem_action = MPLUS then mem := calc_result
               endif
25         if mem_action = MRESET then mem := undef
               endif
26 endpar
27 // INITIAL STATE
28 default init s0:
29 function calc_result = 0
30 function mem = undef
```

• The definitions section consists of the specification of functions, domains, transition rules, and possible invariants. Depending on the system behavior, different transition rules can be used, e.g., the *update* rule to assign a specific value to a function, the guarded updates (if-then, switch-case) where the execution of an action depends on conditions, simultaneous updates (par). For example, Code 1 uses the par rule to run in parallel a set of if-then guarded updates.

• The main rule is the starting point of each computation step. It calls the other rules (macro rule) based on the defined behavior.

• The default init section contains the initialization of *controlled* functions. An example is the function calc_result which is initialized to 0.

When modeling a system, it is possible to specify state invariants (see line 16 in Code 1) as first-order formulas that must be true in each computational state. A set of safety assertions can be specified as model invariants, and a state is considered *safe* if it does not violate any invariant.

Once the specification is available, it can be validated using the simulator `AsmetaS`. In particular, for this work, we exploit an alternative simulator, namely `AsmetaSimulatorWR` [9], supporting the rollback that can bring back the simulated machine to the previous state. Additionally, the validation can be carried out by using the animator `AsmetaA` [8], by performing a graphical simulation in which the values of the monitored and controlled functions can be kept under control. Specifications can be validated also by writing scenarios in the `Avalla` language [13]. A scenario describes how the state must evolve. It is a sequence of actions executed on the system and the expected behavior of the system. An example is reported in Code 2. `Avalla` scenarios can be executed by the validator `AsmetaV`.

A scenario starts with the keyword scenario followed by the name (e.g., test1) and it is linked to the specification to be tested using the load command. After that, the scenario sets all the input functions (e.g. set math_action := INC), makes the specification execute a step, then it checks that functions have the expected values (e.g. check calc_result = 5). More specifically, the scenario in Code 2 simulates the

Code 2 An Avalla scenario for the simple calculator

```
scenario test1
load SimpleCalculator.asm
// Increment the result
set math_action := INC;
set mem_action := undef;
set number := 5;
step
check calc_result = 5;
// decrement
set math_action := DEC;
set mem_action := undef;
set number := 2;
step
check calc_result = 3;
// store result in memory
set math_action := undef;
set mem_action := MPLUS;
step
check mem = 3;
// reset the memory
set math_action := undef;
set mem_action := MRESET;
step
check mem = undef;
```

execution of 5 steps. Subsequently, at each step, a different command is chosen: INC, DEC, MPLUS, MRESET. After each step, the scenario checks whether the modeled behavior successfully matches the expected one.

Besides validation, verification is used to verify system properties and consequently the correctness of the specification. In the `Asmeta` framework, it is feasible thanks to the model checker `AsmetaSMV`.

Using the model checker or the simulator, the tool `ATGT` can automatically generate abstract unit tests given the specification. In the first case, test sequences are generated to cover the rules and the guards inside the conditional rules, while the simulator chooses randomly the values of monitored functions and performs a given number of steps of the machine as requested by the tester.

3 Formal model–view–controller

The *formal* Model–View–Controller (*f*MVC) is a design pattern allowing the integration of a formal model (in our case, an `Asmeta` model) with a graphical UI developed in Java. It is supported by the `AsmetaFMVCLib`[2] that includes all the classes and methods necessary to apply it. In this section, we introduce and explain the overall approach that is shown in Fig. 2, which gives a view of the dynamic of the *f*MVC. First we explain the *binding* (step 0 in the figure) and then when we present the controller, we will discuss the other steps.

Figure 3 gives a static view of the pattern. As in the classical Model–View–Controller (MVC) design pattern, *f*MVC integrates a *Model* – which is a *formal* one in *f*MVC–, a *View*, and a *Controller*. Thus, the user is required to implement only an app-specific *View* and the *Controller*, namely the classes `SpecificUI` and `SpecificController` marked in gray in the UML diagram in Fig. 3.

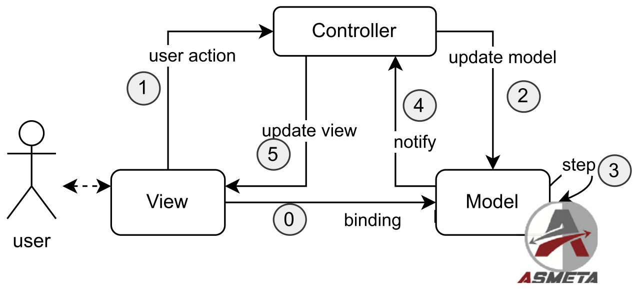

Fig. 2 Formal Model–View–Controller architecture with `Asmeta`

Model In the *f*MVC approach, the model is an instance of the class `AsmetaFMVCModel`, and it embeds an `Asmeta` specification as a formal model. The `AsmetaFMVCModel` keeps track of the current state of the model, with the values of the monitored and controlled functions. Before the execution, the *Model* takes the `Asmeta` model path and parses the specification, and during the execution it simulates step by step the specified ASM model. Note that, differently from what was done for [6], in this enhanced version of the *f*MVC pattern the `Asmeta` simulator supports the rollback when an unsafe state is reached because of unexpected or wrong inputs (see Sect. 3.2) [9].

The `Asmeta` specification used as *Model* can be any model. However, there are some constraints as explained in the paragraph *Actions in the Model* that will follow.

View The *View* is a Java graphical container (like a Swing JFrame[3]), including many graphical components with which the user interacts (e.g., buttons, text fields, spinners, etc.). Those components may be used as inputs, meaning that they are used to assign the value of selected *monitored* locations of the `Asmeta` model, or as outputs, i.e., they are able to show information regarding the current state of the model, including the values of selected controlled locations. The *View* is a

[2] The code of the `AsmetaFMVCLib` is available online at https://github.com/asmeta/asmeta/tree/master/code/experimental/asmeta.fmvclib.

[3] https://docs.oracle.com/javase/7/docs/api/javax/swing/JFrame.html.

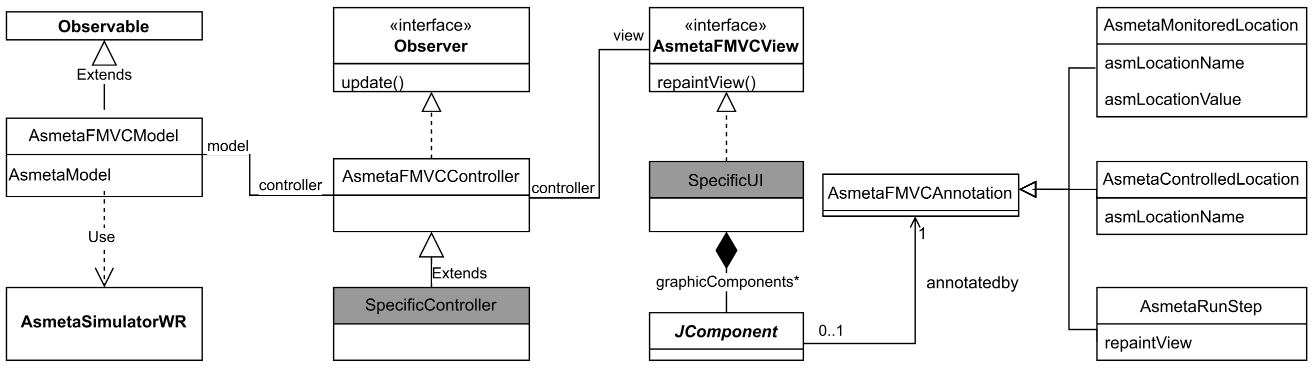

Fig. 3 UML class diagram for the *f*MVC pattern

Java class that implements the interface `AsmetaFMVCView`, which generalizes all the possible views and requires the specific view to implement the method `repaintView` that is called when the GUI needs to be repainted. In particular, this method is used when the *View* changes (in terms of the number or type of components) based on the update of the *Model*. Instead, if only the data shown in the *View* change, no repaint is required.

When developing the *View*, the developer must establish a static binding between the components and the functions in the *Model* (step 0 in Fig. 2). This is done only once, when defining the *View*, while all the other operations (such as the update of the *Model* or the *View*) are cyclically performed at every execution step. In particular, the binding between components and the functions in the *Model* is performed by using one or more of the following Java annotations when declaring a graphical component:

• `@AsmetaMonitoredLocation`: it is used to link a graphical component able to collect some input from the user or to interact with him or her (e.g., button, text field, table, etc.) to a monitored function of the `Asmeta` model. When using this annotation, users have to specify the `asmLocationName`, i.e., the name of the linked location in the `Asmeta` model. During the execution, the value to be assigned to the linked monitored function is taken from the graphical component that contains editable data (e.g., if the component is a text field its text will be used). For instance, the `JTextField number` of Code 3 (line 2) is linked to the `Asmeta` monitored function number, and its value is taken from the text inserted by the user. If the graphical element does not contain editable data (e.g., a button), the value is specified by using the annotation attribute `asmLocationValue`. For example, the `JButton mPlusBtn` in Code 3 (line 13) is linked to the mem_action monitored function, and its value (`MPLUS`) is specified within the annotation by using the attribute `asmLocationValue`.

• `@AsmetaControlledLocation`: it is used to link a graphical component to a specific controlled function of the `Asmeta` model. The name of the location to be linked is specified through the annotation attribute `asmLocationName`.

• `@AsmetaRunStep`: it is used to indicate whether an action (click, change in the text, tick of the timer, and so on) on the component implies the execution of a step of the `Asmeta` model. Some components may require the GUI to be repainted (e.g., because an action causes the change in the number of components to be shown). In this case, the `@AsmetaRunStep` annotation allows setting the flag `repaintView`.

Actions in the model In the *f*MVC pattern, we require only that the model is an `Asmeta` specification. This specification may model more possible behaviors than those allowed by the integration with the graphical UI. For example, if many monitored functions linked to buttons are declared, the formal model may allow changing them all in the same simulation step. However, when integrating a formal specification with a *View* the possible execution paths are restricted, e.g., the user can click only a single button at a time. Thus, in order to restrict the possible behaviors of the `Asmeta` specification only to those actually possible with the final executable, users should add an invariant ensuring that monitored locations can change only as allowed by the UI. In order to make this more systematic and to help the modeler to write this sort of invariants, we introduce the concept of *model action*.

The combined use, over the same graphical elements, of the annotations `@AsmetaRunStep` and `@AsmetaMonitoredLocation` linked to a monitored function mon, identifies that function mon as *action* in the model. These *actions* represent the events that are fired when a user performs a click, update, or general interaction with the UI: a monitored function must be set accordingly, and an update of the *Model* is required. These actions are not free to change as they please, since actions can be set maximum one at a time. For example, the invariant at line 16 in Code 1 is used to ensure that only one of the two possible actions (namely, math_action or mem_action) can be executed in every simulation step.

Note however, that a graphical element annotated `@AsmetaRunStep`, i.e., requiring a step of the model, may not be annotated by `@AsmetaMonitoredLocation`, i.e., that graphical elements is not linked to an action in the model. In this case, the model must allow that a step is performed without any action, i.e., with every action set as undef. For instance, we will present the invariant in the AMAN$_2$ specification that allows also all the actions to be undef.

With this technique, we ensure that the `Asmeta` specification and the *View* are always consistent, and the behaviors allowed in the former are possible also in the latter, and vice versa.

Controller In the *f*MVC approach, the *Controller* is a class extending the `AsmetaFMVCController` and controlling the flow of information. When it is built, the *Controller* is linked both to the *View* and the *Model*. Moreover, the *Controller* component registers itself as an actionListener and changeListener (whichever is applicable) to all the fields annotated with `@AsmetaRunStep` in the *View*.

When an action is performed by the user on elements annotated with `@AsmetaRunStep`, the *Controller* handles the specific request (step 1 in Fig. 2). It gives the value to the monitored functions in the current state of the `Asmeta` model by extracting them from the components annotated with `@AsmetaMonitoredLocation` in the *View* (step 2 in Fig. 2). Then, the *Controller* executes a single simulation step of the `Asmeta` model (step 3 in Fig. 2). In this way, the *Model* is updated and the *Controller* (which is an observer of the model) is notified of the changes (step 4 in Fig. 2). Finally, the *Controller* has the responsibility to update the values shown in the *View* (step 5 in Fig. 2) by extracting them from the `Asmeta` model and assigning the values to the respective components annotated with `@AsmetaControlledLocation`. Moreover, if the component firing the update requires the repaint of the *View*, the method `repaintView` is executed.

In *f*MVC, as shown in Fig. 2, the View has no reference to the Model and it cannot directly query the Model, differently from the original MVC pattern implemented in Smalltalk [11]. According to the classification of MV* patterns [29], our approach is more similar to that of Model–View–Presenter pattern, with a Passive View. In *f*MVC, the controller is a kind of mediator similar to what is used by Apple Cocoa or partly used by Sun Java Swing.

3.1 A simple example: a UI for the simple calculator

In this section, we present how the *f*MVC pattern can be applied to the simple calculator whose model has been presented in Code 1. In particular, we implement a UI allowing the user to insert a number in a text field and to increment or decrement the calculator result of that number, depending

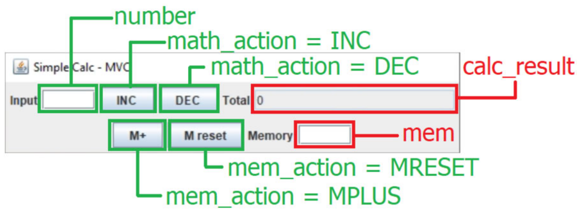

Fig. 4 Mapping between view components and `Asmeta` controlled (red) or monitored (green) locations

on the button pressed. The result of the operation is shown in a text field and can be memorized by the system in local memory when the M$^+$ button is clicked. When a result is memorized, it is shown in a dedicated text field, which can be erased by clicking on the M reset button. The *View* and the binding of its components with the *Model* are shown in Fig. 4 and described in the following.

The Java code defining the *View* and implementing the `AsmetaFMVCView` interface is reported in Code 3. First, we annotate with `@AsmetaMonitoredLocation` the text field `number`, used to valorize the number monitored function in the ASM model (see Code 1). Then, we link the two buttons performing mathematical operations, i.e., `m_INCBtn` and `m_DECBtn`, with the math_action monitored function. The former, when pressed, sets the monitored function to the value INC, while the latter to the value DEC. Note that both buttons fire the update of the *Model* when pressed. For this reason, they are annotated with `@AsmetaRunStep`. Similarly, the two buttons `mPlusBtn` and `mResetBtn` are linked to the monitored function mem_action and set it, respectively, to MPLUS and MRES when pressed, as defined by the `asmLocationValue` attribute. Finally, the text field `memory` shows the content of the mem controlled function, and the text field `calcresult` reports the outcome of the calculator computation contained in the calc_result controlled function. The mapping between the two text fields and the respective controlled locations is done by using the `@AsmetaControlledLocation` annotation.

Then, the code for the Controller is defined by extending `AsmetaFMVCController`. It redefines the method `update` (see Code 4), which is automatically called when the *Model* notifies a change in its values. This extension is needed since the controller in the *f*MVC framework automatically handles the output of the main types of data (e.g., the text to be shown in a text field, in a label, in a table, etc.), but case-specific outputs that are not handled by `AsmetaFMVCLib` (such as the color of a text field) have to be managed by the user. For example, we have defined how the background color of the text field changes based on the current value of the calculator result. Note that the background will never be set as red since all the updates are handled by the model, and, due to the invariant at line 15 in Code 1, the value of the result will never be negative. However, the user may decide

Code 3 View class for the simple calculator example

```
public class SimpleCalculatorView extends JFrame implements AsmetaFMVCView {
    @AsmetaMonitoredLocation(asmLocationName="number")
    private JTextField number = new JTextField(5);

    @AsmetaMonitoredLocation(asmLocationName="math_action", asmLocationValue = "INC")
    @AsmetaRunStep
    private JButton m_INCBtn = new JButton("INC");

    @AsmetaMonitoredLocation(asmLocationName="math_action", asmLocationValue = "DEC")
    @AsmetaRunStep
    private JButton m_DECBtn = new JButton("DEC");

    @AsmetaMonitoredLocation(asmLocationName="mem_action", asmLocationValue = "MPLUS")
    @AsmetaRunStep
    private JButton mPlusBtn = new JButton("M+");

    @AsmetaMonitoredLocation(asmLocationName="mem_action", asmLocationValue = "MRESET")
    @AsmetaRunStep
    private JButton mResetBtn = new JButton("M␣reset");

    @AsmetaControlledLocation(asmLocationName="mem")
    private JTextField memory = new JTextField(5);

    @AsmetaControlledLocation(asmLocationName="calc_result")
    private JTextField calcresult = new JTextField(20);

    public SimpleCalculatorView() {
        // Adds the component to the Java frame
    }

    @Override
    public void repaintView(boolean firstTime) { }
}
```

Code 4 Controller class for the simple calculator example

```
public class CalcController extends AsmetaFMVCController{
    public CalcController(AsmetaFMVCModel model, SimpleCalculatorView view)
    throws IllegalArgumentException, IllegalAccessException {
        super(model, view);
    }

    @Override
    public void update(Observable o, Object arg) {
        // Handle the main parameters as regularly done by the AsmetaFMVCLib
        super.update(o, arg);

        // Set the background color of the result textbox based on the sign
        JTextField res = ((SimpleCalculatorView) this.m_view).getmTotalTf();
        if (Long.parseLong(res) >= 0)
            res.setBackground(Color.GREEN);
        else
            res.setBackground(Color.RED);
    }
}
```

Code 5 Definition of the three components for the simple calculator example

```
// Define the model with the ASM spec
AsmetaFMVCModel model = new AsmetaFMVCModel("model/SimpleCalculator.asm");

// Define the view
CalcView view = new CalcView();

// The controller has both the references of model and view
AsmetaFMVCController controller = new CalcController(model, view);

// Show the view
view.setVisible(true);
```

to write additional checks in the Java code, e.g., because he/she wants to reuse the same view in multiple applications where the *Model* is weaker.

Finally, the three components are connected and launched in a main class as in Code 5.

3.2 Dealing with wrong actions

When a user interacts with a UI he/she may perform some invalid actions, i.e., actions that should not be allowed by the software under analysis or at least that must be ignored and not produce any effect on the model. For example, in the simple calculator reported in Sect. 3.1, we may want to discard operations that make the result negative, and this can be declared at the model level by using the invariant reported in Code 1 line 15. However, the user can insert a value greater than the result and click the decrement button, leading to a possible negative result. We would like to avoid or prevent this type of actions or at least make them ineffective.

This problem can be easily tackled by directly using the formal model in the *Model* component as we propose with the *f*MVC pattern. In fact, when working with `Asmeta` (see Sect. 2), the developer can address this problem in two ways: (1) by inserting in the model some rules that prevent to reach an invalid state (sort of *defensive programming*), (2) by adding the invariants and asking that monitored values that cause their violations are not considered (similar to *design by contract*).

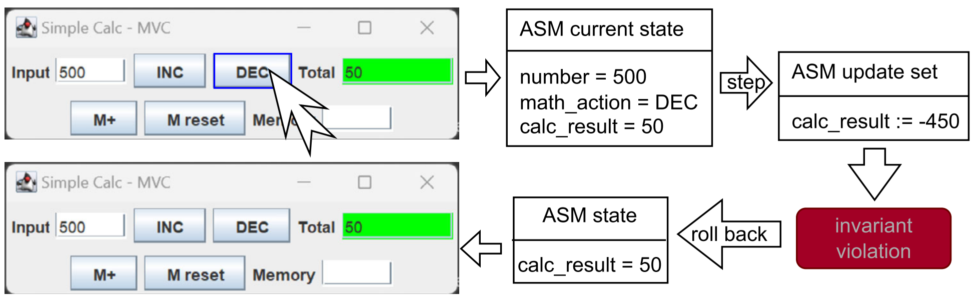

Fig. 5 Example of the invariant violation in the simple calculator

In the former approach, when the values given as input by the user do not satisfy a conditional guard, the `Asmeta` simulator embedded into the *Model* component does not update the corresponding controlled locations when executing a simulation step. In this way, the UI is correctly not updated.

We focus now on the latter approach. With *f*MVC, the update of values shown in the *View* is always performed by the *Controller*, based on the value of the controlled functions in the ASM model after the execution of a simulation step. Using the mechanisms embedded into the `Asmeta` framework, we have extended `AsmetaFMVCLib` such that actions are ignored when their execution causes the violation of one or more invariants: an `InvalidInvariantException` is thrown by the `Asmeta` simulator and caught by the AsmetaFMVCModel. Furthermore, a rollback is executed by the `Asmeta` simulator which brings back the *Model* in a safe and correct state.

Example
Consider the user interaction shown in Fig. 5 with the Simple Calculator. Let us assume that the user wants to decrement (by clicking the DEC button) 500 from 50. After the click of the button, the `Asmeta` *specification will compute the next state. However, the new state would have calc_result negative, which is against the invariant, so the Simulator will throw an exception. The previous state will be restored with calc_result equal to 50 and the View will be updated accordingly – since the state has actually not changed, the view remains unchanged.*

`Asmeta` handles similarly also *inconsistent updates*, i.e., when the same location is updated to two different values at the same time. When inconsistent updates could not be detected using the `AsmetaMA` model advisor, they may happen at runtime. In this case, the `Asmeta` simulator raises an exception, it rolls back to the previous state and the *View* does not change.

As a consequence, using the *f*MVC approach and embedding the `Asmeta` model into the final system, only valid actions are executed and valid values handled. This makes the consistency between the ASM model and the *View* always assured, i.e., the *View* reports the same values as those in the *Model* state, and guarantees that both of them always remain in a safe state.

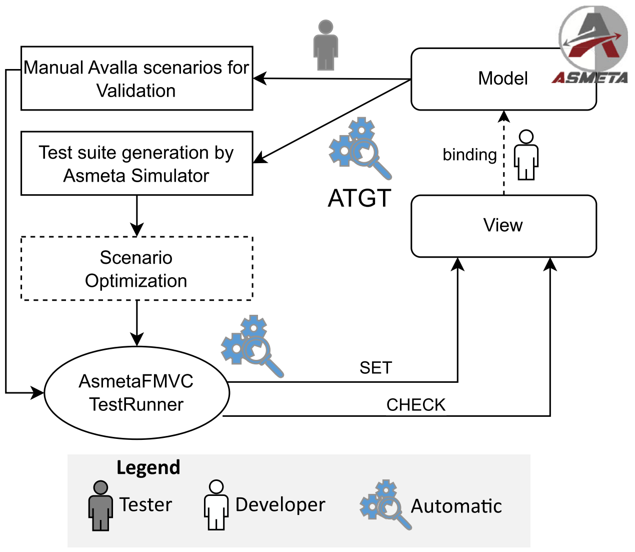

Fig. 6 Model-based testing process for UIs

4 Model-based testing of UIs

We have improved the *f*MVC framework to support a novel method for testing UIs, by extending the `AsmetaFMVCLib`. Once the *Model* has been developed and linked to the *View*, as explained in Sect. 3, the model-based testing approach uses `Avalla` scenarios as test scripts for the UI. The scenarios, which simulate the interaction of the user with the *View* (such as with the more classical scripts derived by capture–replay testing techniques [26]), can be automatically derived from the *Model* or manually written. In particular, the model-based testing process of the UI we propose follows the workflow presented in Fig. 6 and formalized in Algorithm 1.

Manually and automatically building test scenarios in `Avalla` Test scenarios can be manually written by the tester who wants to test particular conditions of the system (as that reported in Code 2 for the example case study),

Algorithm 1 Model-based testing process supported by Asmeta and the *f*MVC framework

```
Require: $model$ the model
Require: $view$ the view
Require: $controller$ the controller
Require: $nStep$ the number of steps for each scenario
Require: $nTest$ the number of scenarios to generate
Require: $actions$ the list of mutually-exclusive actions
Require: $stepTime$ the time interleaving each command execution
Require: $manualTests$ the Avalla scenarios written manually

      ▷ Generate the test suite
1: $ts \leftarrow$ AsmTGBySimulationOneAction.GETTS($model, nStep, nTest, actions$)
      ▷ Optimize the test scenarios
2: $ts \leftarrow$ OPTIMIZESCENARIOS($ts$)
3: $ts \leftarrow ts \cup manualTests$
4: for all $test \in ts$ do
      ▷ Run the test
5:    $runner \leftarrow AsmetaFMVCTestRunner(view, controller, test, stepTime)$
6:    $runner.runTest()$
7: end for
```

or automatically generated by ATGT, which exploits either the model-checker-based or random simulation to derive scenarios. For this work, we have extended the ATGT random test generation by Asmeta simulator by allowing the generation of test sequences mimicking the interaction of a user with the UI: a user interacting with the GUI can execute one single action for each execution step. Thus, the AsmTGBySimulationOneAction class generates scenarios by considering a list of mutually-exclusive *actions*, meaning that only one of them is not undef in each scenario step (line 1 in Algorithm 1). Moreover, in order to mimic a real interaction, the action allowing the step to be performed is always set as the last one in the scenarios. This is what a human user would do: first, he/she sets all the values needed for the current simulation step; then, he/she launches the step. This procedure can be executed to generate a configurable number of scenarios, each one with the desired number of steps.

Example
An example scenario automatically generated for the simple calculator case study by our fMVC framework and supported by the Asmeta *tools is reported in Code 6. In practice, we have generated 10 scenarios, each composed of 10 steps.*

Scenario optimization Given automatic test scenarios generated by the Asmeta simulator, we optionally perform the test optimization (line 2 in Algorithm 1). This optimization does not change the semantics of the tests but improves their readability and executability, especially in complex systems, where the functions to be checked are many. More specifically, our framework performs the *Check Optimization* [5]: it removes the check of unchanged controlled locations. If a controlled location in state s_i has the same value it had in the state s_{i-1} (in which it was already checked and the check was not removed), the corresponding check is removed, if present.

Test scenario execution After the scenarios are optimized, the runTest method of AsmetaFMVCTestRunner is used to run the Avalla scenarios on the UI (lines 5 and 6 in Algorithm 1). It reads each scenario line by line and interacts with the *View* accordingly with the Avalla instruction (**set**, **check**, or **step**). The **check** instructions are translated into one or more asserts which read the values shown by the *View* in the components annotated with @AsmetaControlledLocation and compares them with those expected by the scenario. Then, the **set** directives are translated into one or more instructions assigning the values provided by the Avalla scenario to the components annotated with @AsmetaMonitoredLocation. In addition, if the **set** is referred to a component annotated with @AsmetaRunStep, a simulation step is performed. That said, the **step** instructions are skipped, as the execution step is performed by the *View* components with associated actions. Indeed, the practical effect of a **step** instruction is carried out by the execution of a **set** on an action, which is the last one we execute at each execution step in our scenarios.

Table 1 reports the list of components[4] supported by the *f*MVC MBT framework and how the AsmetaFMVCTestRunner maps **set** and/or **check** directives when executing the scenarios.

[4] ButtonColumn is a custom-made component made up of a table with a single column of selectable buttons. It is available at https://github.com/asmeta/asmeta/blob/master/code/experimental/asmeta.fmvclib/src/asmeta/fmvclib/controller/ButtonColumn.java.

Table 1 Mapping of check and sets directives into asserts and value setting. Set actions marked with $*$ are those firing a step after their execution

Component	Set [= *val*]	Check [= *val*]
JTextField	tField.setText(val)	assert(tField.getText().equals(val))
JLabel	label.setText(val)	assert(label.getText().equals(val))
JSpinner	spinner.setValue(val)	//
JSlider	slider.setValue(val) slider.getChangeListeners().stateChanged()*	//
JButton	button.doClick()*	//
Timer	timer.getActionListeners().actionPerformed()*	//
ButtonColumn	btnCol.setEditingRow(index) btnCol.actionPerformed()*	assert(btnCol.getValueAt(in-dex).equals(val))
JTable	table.setValueAt(val, index)	assert(table.getValueAt(index).equals(val))

Code 6 An `Avalla` scenario for the simple calculator generated by the proposed MBT approach

```
load SimpleCalculator.asm
check calc_result = 0;
set number := −1843136949;
set mem_action := undef;
set math_action := DEC;
[...]
check calc_result = 987545668;
check mem = undef;
set mem := undef;
set math_action := undef;
set mem_action := MRESET;
[...]
set math_action := undef;
set mem := undef;
set mem_action := MPLUS;
step
check mem = 987545668;
check calc_result = 987545668;
set math_action := undef;
set mem_action := MPLUS;
step
check mem = 987545668;
check calc_result = 987545668;
step
```

Table 2 Coverage reached by executing the 10 scenarios automatically generated by our MBT approach on the simple calculator example

Statement	Branch	Method
92.4%	50.0%	57.1%

When executing the test scenario, `AsmetaFMVCTestRunner` allows for specifying the step time interleaving each line execution (see line 5 in Algorithm 1). In this way, the test execution speed can be adapted and made as slow as needed to allow the observability of the execution process.

Example
In the case of the simple calculator, we have executed the optimized scenarios setting the `AsmetaFMVCTestRunner` *with a step time of* 50 *ms. The coverage reached by the 10 test scenarios that we have generated,*[5] *in terms of methods, branches, and statements is reported in Table 2.*

[5] The test scenarios are available at https://github.com/asmeta/asmeta_based_applications/tree/main/fMVC/SimpleCalculator/scenarios.

5 The AMAN case study

In this section, we explain how we have applied the *f*MVC pattern to the Arrival Manager (AMAN) case study[6] proposed at ABZ 2023 conference [28]. In particular, we first analyze the modeling and V&V activities we have performed with `Asmeta`, including property verification and model validation through manually written `Avalla` scenarios. Then, we delve into the implementation of the *Controller* and the *View* of our AMAN prototype in order to let them interact with the `Asmeta` *Model*. Finally, we introduce the MBT activities that we have performed on the UI starting from automatically generated scenarios.

5.1 Formal (asmeta) model

In this section, we describe the structure of the `Asmeta` model and the requirements we have implemented in our solution. First, we present the modeling strategy we have adopted and we present the details of all models we have obtained.

[6] The source code, the `Asmeta` model of the AMAN case study, and the `Avalla` scenarios are available online at https://github.com/asmeta/asmeta_based_applications/tree/main/fMVC/AMAN.

The entire model development process has been carried out by applying the Asmeta framework as presented in Sect. 2. Then, we present the scenario-based validation activities and, finally, we describe all the properties we have proved on the models.

5.1.1 Modeling

The modeling strategy we have adopted while developing the AMAN model is based on an iterative design process, in which, at every refinement step, new behaviors are added. We emphasize that the Asmeta models we report in this section aim to be the representation of the AMAN behavior, including user inputs, but do not consider external events, such as airplane rescheduling.

AMAN$_0$ Initially, we have specified in the most simple model (in the following identified as $AMAN_0$ and reported in Code 7) all the functionalities that we considered, sometimes in a limited way, except the passing of time, which we did not handle at this level. Note that, even if Code 7 references to TimeSlots, these represent only the different positions, i.e., the indexes, in the landing sequence and do not have a direct correspondence with the time. In particular, this model entirely manages the landing sequence (i.e., labels of the airplanes, color of each airplane, and status of each time instant – blocked or not blocked), with a maximum size of 20 time instants, as defined by the range declared at line 13 in Code 7. This model supports the movement up or down of an airplane, only for a single time instant, as well as the hold functionality. An excerpt of the Asmeta specification, with the rule allowing an airplane to be moved up, r_moveUp, is shown in Code 7 starting from line 33. This rule checks, given the current landing time $lt of the selected airplane, that the destination time instant is not locked or too much closer to another landing airplane. Most of these checks are carried out by the derived function canBeMovedUp (see line 16), which ensures an airplane can be moved only if the required minimum distance between it and other airplanes is maintained, and if the destination slot is not blocked. Specifically, the function returns false if any of the next four available time slots are already occupied by another airplane or if the destination slot is blocked, i.e., if the airplane cannot be moved, and true otherwise.

AMAN$_1$ In this refinement step, we have removed all the limitations previously introduced in $AMAN_0$, except for the passing of time. More specifically, with $AMAN_1$ all the 45 time instants requested by the official requirements are available, but they still represent only the indexes in the landing sequence and do not have a direct correspondence with the time. Furthermore, $AMAN_1$ allows for moving airplanes for

Code 7 Asmeta rule handling the moving up of an airplane in AMAN$_0$

```
 1  [...]
 2  domain TimeSlot subsetof Integer
 3  domain ZoomValue subsetof Integer
 4  // given a time slot, return the aiplane in it or undef if empty
 5  controlled landingSequence: TimeSlot -> Airplane
 6  // given a time slot, return true iif blocked
 7  controlled blocked: TimeSlot -> Boolean
 8  controlled zoomValue : ZoomValue
 9  // compute the position of a given airplane
10  derived position: Airplane -> TimeSlot
11  derived canBeMovedUp: Airplane -> Boolean
12  [...]
13  domain TimeSlot = {0 : 20}  // range of TimeSlot from 0 to 20
14  domain ZoomValue = {15 : 45}  // zoom from 15 to 45
15  [...]
16  function canBeMovedUp($a in Airplane) =
17      let ($lt = position($a)) in
18      if $lt >= 20 then false else
19          let ($blk = blocked($lt + 1)) in
20          if $blk then false else
21              if isDef(landingSequence($lt + 1)) then false
22              else if ($lt + 2) <= 20 then
23                  if isDef(landingSequence($lt + 2)) then
                        false
24                  else if ($lt + 3) <= 20 then
25                      if isDef(landingSequence($lt + 3)) then
                            false
26                      else if ($lt + 4) <= 20 then
27                          if isDef(landingSequence($lt + 4))
                                then false
28                          else true endif endif endif endif
                                endif endif endif endif
29          endlet
30      endif
31      endlet
32  [...]
33  rule r_moveUp($a in Airplane) =
34      let ($lt = position($a)) in
35          if $lt != −1 and $lt < 15 then
36              if $lt < zoomValue and canBeMovedUp($a) then
                    par
37                  landingSequence($lt + 1):= $a
38                  landingSequence($lt):= undef
39                  [...]
40              endpar endif endif endlet
41  [...]
42  default init s0:
43          function landingSequence($t in TimeSlot) =
44      if $t = 5 then fr1988 else
45      if $t = 2 then u21748 else undef endif endif
46          function zoomValue = 30
47          function action = NONE
48          function selectedAirplane = undef
49          function statusOutput($t in Airplane) =
50      if $t = fr1988 then UNSTABLE else
51      if $t = u21748 then FREEZE else STABLE endif endif
52          function landingSequenceColor($t in TimeSlot) =
53      if $t = 5 then YELLOW else
54      if $t = 2 then CYAN else WHITE endif endif
55          function blocked($t in TimeSlot) =
56      if $t = 6 then true else false endif
```

Code 8 `Asmeta` rule handling the moving up of an airplane in AMAN$_1$

```
rule r_moveUp($a in Airplane, $manual in Boolean, $nMov
    in TimeSlot) =
        let ($currentLT = landingTime($a)) in
        if ($currentLT != undef and $nMov != undef
            and canBeMovedUp($a, $nMov) !=
            undef) then
            if $currentLT < zoomValue and
                $currentLT + $nMov <= 45
                and not blocked($currentLT +
                $nMov) and canBeMovedUp(
                $a, $nMov) then
            par
                landingSequence(
                    $currentLT + $nMov)
                    := $a
                landingSequence(
                    $currentLT):= under
                [...]
            endpar
            endif endif endlet
```

more than a single time slot. For this reason, in this refinement level, we have added a new monitored function numMoves defining the number of movement steps. Additionally, to make the definition of the rules for the movement of airplanes easier, we have added a derived function landingTime which associates to each airplane its corresponding current landing time. Both the two new functions are used when moving an airplane up or down, as shown in Code 8.

AMAN$_2$ In the last refinement step reported in Code 9, we have implemented the handling of time passing, by exploiting the `Asmeta` TimeLibrary [4]. This integration allows us to use the $AMAN_2$ specification in the *Model* component when using the *f*MVC pattern, as we can show the current time and automatically move the airplanes together with their landing time along the landing sequence as time passes.

As introduced in Sect. 3, in order to embed this `Asmeta` specification in the final executable, we have also added in the $AMAN_2$ model an invariant (see line 5.1.1), restricting the model behavior to the same possible by the user interaction with the UI. In particular, it represents the constraints posed by the use of a graphical interface: only a single event can occur per time. If the user locks a time instant, during the same step she/he cannot change the zoom level or move an airplane in a different time slot. Similarly, if the user changes the zoom level, he/she cannot lock a time instant or move an airplane, and so on.

5.1.2 Scenario-based validation

After having modeled the system as described in Sect. 5.1 with an ASM, we have performed scenario-based validation activities. In particular, with scenarios, we are able to check the behavior of the system against the expected one, in terms of values assumed by controlled functions when the value of the monitored ones is set and one or more simulation steps are executed.

During the development process of the `Asmeta` model, for the last refinement level, namely $AMAN_2$, we have written four `Avalla` scenarios, aiming at testing all the possible operations allowed by the AMAN UI and executing at least once all the rules of the `Asmeta` specification. An example of these scenarios is shown in Code 10 which reports an execution trace in which a time instant is locked (line 6) and the scenario tries to move an airplane in that time instant (line 16), but nothing changes since, due to system requirements, it is not possible to move an airplane into a blocked time slot (from line 18 to 20). Thanks to the execution of the scenarios through the `AsmetaV` tool, we have measured which rules are covered by each scenario. We report this summary in Table 3. Note that we cover every rule in the `Asmeta` specification by the scenarios.[7]

5.1.3 Property verification

During the development of the `Asmeta` model, we have kept track of all requirements we covered among those listed in the document presenting the case study. Table 4 reports all the requirements we have addressed. Note that some of them are guaranteed by the way in which we have implemented our ASM model (those proved with an LTL property), while others are granted by the fact that we have used Java Swing to implement the *View* (see Sect. 5.2), or by the operating system.

For what concerns the requirements provable with a temporal property, this is one of the main advantages of using a formal specification (such as in the case of `Asmeta`) in the development of a system. In particular, with the *f*MVC approach, if one proves the property on the formal specification and uses that specification in the *Model* component linked to the *View*, the obtained software behavior is correct (with respect to the proved properties) by construction.

More specifically, for the AMAN case study, we have proved the safety properties on the first model (i.e., $AMAN_0$). Indeed, the `AsmetaSMV` tool uses the NuSMV model checker which is not able to deal with infinite domains (such as the integers used by the `Asmeta` TimeLibrary [4] to store the time). However, the usefulness of this process is still preserved: since the refinements we have performed on the model are *stuttering refinements* [1], the refined model preserves the properties proven on the more simple ones.

5.2 View

To test the application of the *f*MVC pattern on the AMAN case study, we have implemented a simplified version of the

[7] All manually-written scenarios are available at https://github.com/asmeta/asmeta_based_applications/tree/main/fMVC/AMAN/model.

Code 9 `Asmeta` specification for AMAN$_2$

```
asm aman2
import StandardLibrary
import TimeLibrary
import aman1

signature:
 domain Minutes subsetof Integer
 domain Hours subsetof Integer
 derived currentTimeMins: Integer
 derived currentTimeHours: Integer
 derived zoomChanged: Boolean
 controlled timeShown: TimeSlot −> Minutes
 controlled lastTimeUpdated : Minutes
 controlled mins : Minutes
 controlled hours : Hours

definitions:
 domain Hours = {0 : 23}
 domain Minutes = {0 : 59}
 function currentTimeMins = rtoi(mCurrTimeSecs/60) mod 60
 function currentTimeHours = rtoi(mCurrTimeSecs/3600) mod
      24
 function zoomChanged =
  if zoom != zoomValue then true else undef endif

 rule r_update_time =
  par
   mins := currentTimeMins
   hours := currentTimeHours endpar

 rule r_update_time_shown =
  par
   forall $t in TimeSlot do
    timeShown($t) := mod(currentTimeMins + $t + 1, 60)
   if lastTimeUpdated != currentTimeMins then
    par
     lastTimeUpdated := currentTimeMins
     forall $a in Airplane do r_moveDown[$a, false, 1]
     forall $time in TimeSlot with $time > 0 do
                         blocked($time − 1) := blocked($time)
    endpar endif endpar
invariant inv_action over timeToLock, zoomChanged, action
      : (timeToLock != undef implies (zoomChanged=undef
      and action=undef)) and (zoomChanged != undef
      implies (timeToLock=undef and action=undef)) and (
      action != undef implies (timeToLock=undef and
      zoomChanged=undef))
main rule r_Main =
 par
  if not isUndef(timeToLock) then r_update_lock[] endif
  if not isUndef(zoom) then r_update_zoom[] endif
  r_update_time[]
  r_update_time_shown[]
  if selectedAirplane != undef then
   if action = UP then
   r_moveUp[selectedAirplane, true, numMoves] else
   if action = DOWN then
   r_moveDown[selectedAirplane, true, numMoves] else
   if action = HOLD then
   r_hold[selectedAirplane] endif endif endif endif endpar
default init s0:
 function landingSequence($t in TimeSlot) = if $t = 5 then
      fr1988 else
          if $t = 2 then u21748 else
          if $t = 18 then fr1989 else
          if $t = 35 then u21749 else
          undef endif endif endif endif
 function zoomValue = 30
 function action = undef
 function selectedAirplane = undef
 function timeShown($t in TimeSlot) = ($t + 1)
 function lastTimeUpdated = currentTimeMins
 function statusOutput($t in Airplane) = if $t = fr1988 then
      UNSTABLE else if $t = u21748 then FREEZE else
      STABLE endif endif
 function landingSequenceColor($t in TimeSlot) = if $t = 5
      then YELLOW else
       if $t = 2 then CYAN else WHITE endif endif
 function blocked($t in TimeSlot) = if $t = 6 then true else
      false endif
 function mins = 0
 function hours = 0
```

AMAN GUI (see Fig. 7) which resembles the one presented in the document describing the case study. The binding between the *View* and the *Model* locations is done by using the annotations presented in Sect. 3 and as shown in Code 11.

In the view, we have used all the components supported by the `AsmetaFMVCLib` and we have extended some of them, in order to show how the library supports not only the components already defined by Java Swing, but others derived from them as well.

In particular, the zoom level is handled by the `zoom` slider, which is graphically shown as a `CustomSliderUI`[8] and assigns its value to the `zoom` monitored function. The `zoom` slider is also one of the mutually-exclusive actions we support in our UI. Indeed, since when the zoom changes, the number of time slots to be shown changes as well, we have annotated it with the `@AsmetaRunStep` annotation. Moreover, the annotation sets the flag `repaintView` to true because the entire UI is repainted to show only the desired number of time slots. In this way, when the slider changes its position, only a single action is executed and the value assumed by the controlled function zoomValue is shown as text of the `lblZoomValue` label.

The current time, stored in two controlled functions (mins and hours) is shown, respectively, in the `lblCurrentTimeMins` and `lblCurrentTimeHours` labels on the view.

[8] The custom class `CustomSliderUI` is available online at https://github.com/asmeta/asmeta_based_applications/blob/main/fMVC/AMAN/src/customcomponents/CustomSliderUI.java.

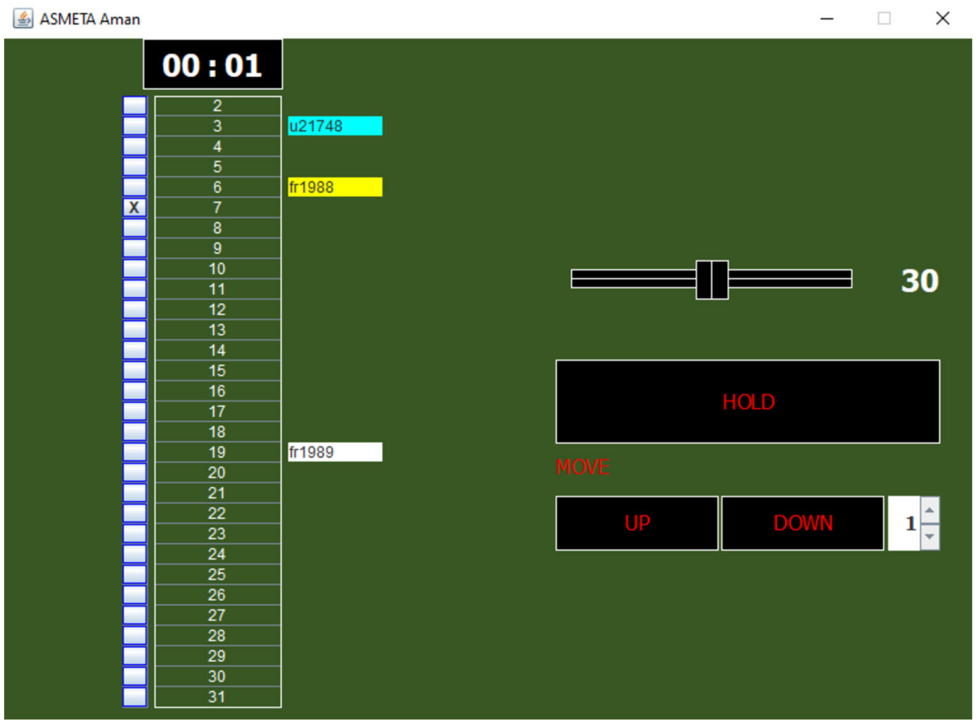

Fig. 7 The GUI of AMAN developed using the *f*MVC pattern

Code 10 Example of an `Avalla` scenario written to test the locking functionality of the $AMAN_2$ model

```
scenario ScenarioLOCK
load aman2.asm
check landingSequence(2) = u21748;
check landingSequence(5) = fr1988;
set action := undef;
set timeToLock := 1;
step
check blocked(1) = true;
check blocked(0) = false;
set timeToLock := 0;
step
check blocked(1) = true;
check blocked(0) = true;
set selectedAirplane := u21748;
set numMoves := 1;
set action := DOWN;
step
check blocked(1) = true;
check landingSequence(2) = u21748;
check landingSequence(5) = fr1988;
```

Table 3 Rules covered by each `Avalla` scenario

Rule	ScenarioH	ScenarioMU	ScenarioMD	ScenarioL
`r_main`	×	×	×	×
`r_update_time_shown`	×	×	×	×
`r_update_time`	×	×	×	×
`r_moveUp`		×		
`r_moveDown`			×	×
`r_hold`	×			
`r_update_lock`				×
`r_update_zoom`		×	×	

For these two components, we have redefined the method `setText()` to let it show the value of minutes and hours with two digits, even if the controlled functions in the `Asmeta` specification are integers and leading zeros are absent.

The movements and actions on airplanes are handled through buttons. In particular, the airplane to be moved is selected through the `airplaneLabels` table, which is mapped to the `Asmeta` monitored function selectedAirplane, for what concerns the selection, and to the controlled landingSequence function, for what concerns the visualization of the airplanes that are landing. In this way, the `airplaneLabels` table is used both as an input and as an output. After having selected an airplane, the button `btnHold` (which sets the action monitored function to HOLD) puts it on hold. Similarly, the airplanes can be moved up or down using the `btnMoveUp` and `btnMoveDown` buttons, that run the simulator for a step and set the action monitored function accordingly. The number of movements, when an airplane is moved up or down, is set through the `spnrNumMoves` which gives the value to the numMoves monitored function.

Next to the airplane labels, the following time instants are stored in the timeShown controlled function and shown in the `times` table, while the `isLockedColumn` button column

Table 4 Requirements captured and temporal properties for the AMAN case study. Requirements marked with ∗ are those granted by Java Swing or by the Operating System

REQ	Description (and LTL property)
REQ3	Airplanes can be moved earlier or later on the timeline
	LTLSPEC (**forall** $a **in** Airplane, $t **in** TimeSlot **with** **g**(position($a) = $t **and** selectedAirplane=$a **and** action = UP **and** canBeMovedUp($a) **implies** **x**(position($a) = ($t + 1))))
	LTLSPEC (**forall** $a **in** Airplane, $t **in** TimeSlot **with** **g**(position($a) = $t **and** selectedAirplane=$a **and** action = DOWN **and** canBeMovedDown($a) **implies** **x**(position($a) = ($t - 1))))
REQ4	Airplanes can be put on hold by the PLAN ATCo
	LTLSPEC (**forall** $a **in** Airplane, $t **in** Time **with** **g**(position($a) = $t and selectedAirplane=$a **and** action = HOLD **implies** **x**(**isUndef**(landingSequence($t)))))
REQ5	Aircraft labels should not overlap
	LTLSPEC (**forall** $t1 **in** Airplane, $t2 **in** Airplane **with** **g**(($t1 != $t2 **and** position($t1) != -1 **and** position($t2) != -1 **and not** **isUndef**(position($t1)) **and not** **isUndef**(position($t2))) **implies** ((position($t1) - position($t2) >=3) **or** (position($t1)-position($t2) <=-3))))
REQ6	An aircraft label cannot be moved into a blocked time period
	LTLSPEC (**forall** $a **in** Airplane, $t **in** TimeSlot **with** **g**(position($a) = $t **implies not** blocked($t)))
REQ15	The HOLD button must be available only when one aircraft label is selected
	LTLSPEC (**forall** $a **in** Airplane, $t **in** TimeSlot with **g**(position($a) = $t **and** **isUndef**(selectedAirplane) **and** action = HOLD **implies** **x**(position($a) = $t)))
REQ16	The zoom value cannot be bigger than 45 and smaller than 15
	LTLSPEC **g**(zoomValue >= 15 **and** zoomValue<=45)
REQ17∗	Aircraft labels must always be positioned in front of a small bar of the timeline
REQ18∗	Lift of the zoom slider should always be located on the slider bar
REQ19	The value displayed next to the zoom slider must belong to the list of seven acceptable values for the zoom
	LTLSPEC **g**(zoomValue = 15 **or** zoomValue = 20 **or** zoomValue = 25 **or** zoomValue = 30 **or** zoomValue = 35 **or** zoomValue = 40 **or** zoomValue = 45)
REQ20∗	Each movement of the mouse on the ATCo table must be reflected by a movement of the cursor on the screen
REQ21∗	There must be one and only one mouse cursor on the screen
REQ22∗	Hold(aircraft) function can only be triggered after a mouse press and a mouse release have been performed on the HOLD button.
REQ23∗	Hold(aircraft) function must not be triggered if there is not a mouse press and a mouse release performed on the HOLD button.

reports the status of each time slot. It shows an X when the corresponding time instant is locked, and no symbol when the time instant is available for the airplanes. When clicking on the buttons in the `isLockedColumn` button column, the timeToLock monitored function is set.

Finally, the `guiTimer` is used to refresh the view every minute without requiring any other action by the user.

5.3 Controller

To adopt the *f*MVC pattern in the AMAN case study, we have extended the `AsmetaFMVCController` included into the `AsmetaFMVCLib` by defining the class `AMANController` as required; see Fig. 3. The `AMANController` adds case-specific behaviors for outputs that are not explicitly mapped to graphical components in the *View*. In particular, when the `AMANController` is notified by the *Model*, it updates the background color of the airplane names in the table on the *View* (line 15) and sets the labels of the buttons signaling the blocked time instants (line 23), as reported in Code 12.

In both the additional setting procedures, the adopted pattern is the same. First, by using the `model.computeValue(...)` method, we compute the value of a specific function in the current simulator state. Then, we obtain the list of all the locations associated with the desired function together with their values using the `model.getValue(...)` method. Finally, we iterate over all the results and we set the properties of graphical components accordingly.

5.4 MBT of the AMAN case study

In this section, we present the results obtained by applying the testing framework presented in Sect. 4 to the AMAN case study.

Initially, thanks to the functionalities offered by the `AsmTGBySimulationOneAction` class, we have generated

Code 11 Mapping with the proposed annotation between View components and `Asmeta` locations

```
// Zoom management
@AsmetaMonitoredLocation( asmLocationName = "zoom")
@AsmetaRunStep(repaintView = true)
private JSlider zoom;

// Current value set for the zoom
@AsmetaControlledLocation(asmLocationName = "zoomValue")
private JLabel lblZoomValue;

// Labels showing the current time
@AsmetaControlledLocation(asmLocationName = "mins")
private JLabel lblCurrentTimeMins;
@AsmetaControlledLocation(asmLocationName = "hours")
private JLabel lblCurrentTimeHours;

// Buttons moving (UP or DOWN) or removing (HOLD)
// airplanes from the landing sequence
@AsmetaMonitoredLocation(asmLocationName = "action",
        asmLocationValue = "HOLD")
@AsmetaRunStep
private JButton btnHold;
@AsmetaMonitoredLocation(asmLocationName = "action",
        asmLocationValue = "DOWN")
@AsmetaRunStep
private JButton btnMoveUp;
@AsmetaMonitoredLocation(asmLocationName = "action",
        asmLocationValue = "UP")
@AsmetaRunStep
private JButton btnMoveDown;

// Number of movements (up or down)
@AsmetaMonitoredLocation(
        asmLocationName = "numMoves")
private JSpinner spnrNumMoves;

// Table showing the landing sequence (i.e., which airplane lands in
// which time). It is used also as input, to select the airplane to be
// moved/removed
@AsmetaControlledLocation(
        asmLocationName = "landingSequence")
@AsmetaMonitoredLocation(
        asmLocationName = "selectedAirplane")
private JTable airplaneLabels;

// Table showing the following time instants
@AsmetaControlledLocation(asmLocationName =
        "timeShown")
private JTable times;

// Time instants blocking (both visualization and setting)
@AsmetaMonitoredLocation(asmLocationName =
        "timeToLock")
@AsmetaRunStep
private ButtonColumn isLockedColumn;

// Timer causing the update of AMAN due to time passing
@AsmetaRunStep
private Timer guiTimer;
```

Table 5 Coverage reached by executing the 10 scenarios automatically generated by our MBT approach on the AMAN case study

Statement	Branch	Method
93.7%	71.1%	100.0%

10 random `Avalla` scenarios, each with 10 steps, which are available in our online repository.[9]

Then, we have executed the scenarios by setting the `AsmetaFMVCTestRunner` with a step time of 50 ms (see Code 13). For the execution of the test scenarios, we have first defined the *Model*, *View*, and *Controller* objects (from line 4 to 7). Then, in order to let the complete update of all data be managed by the scenarios, we have disabled the embedded timer (line 9) and, finally, we have run the experiments with the test runner object (line 14). With our MBT strategy, we have reached the coverage reported in Table 5, in terms of methods, branches, and statements.

Note that, whenever the coverage was below 100%, we checked the code and we found that uncovered statements and branches were only related to exceptions or parts of the methods in the `CustomSliderUI` class that are not used in our AMAN implementation (e.g., sliders placed vertically, or with inverse bounds).

6 Discussion

In this section, we discuss potential threats to the validity of the *f*MVC approach we propose in this paper.

With our experience in applying the *f*MVC to the AMAN case study, we have found that directly using formal models for designing user interfaces may pose several challenges, since the two aspects are very different, and it may be difficult to generalize all the possible interactions, especially with complex systems. For instance, in this paper, we propose to integrate the `Asmeta` simulator's rollback feature (see Sect. 2) to prevent actions from moving the model into an unsafe or inconsistent state. One could argue that actions could simply be disabled if they are not permitted. However, this would require the developer to add numerous GUI-dependent details to the *Model*, reducing its readability and usability for validation purposes. Let us consider the example of an airplane's movement during the landing sequence in the AMAN case study. In this scenario, the *View* of our *f*MVC approach in principle permits moving an airplane into

[9] The scenarios for the AMAN case study are available at https://github.com/asmeta/asmeta_based_applications/tree/main/fMVC/AMAN/scenarios.

Code 12 Controller for the AMAN case study

```
public class AMANController extends
        AsmetaFMVCController {

    public AMANController(AsmetaFMVCModel model,
            AMANView view)
        throws IllegalArgumentException,
            IllegalAccessException { ... }
    }
    @Override
    public void update(Observable o, Object arg) {
        // Handle the main parameters as regularly
            done by the AsmetaFMVCLib
        super.update(o, arg);
        // Set the text on buttons based on the value
            in the TableModel
        updateBlockedStatus();
        // Set the color of cells
        setAirplaneLabelColors();
    }
    public void setAirplaneLabelColors() {
        m_model.computeValue("
            landingSequenceColor", LocationType.
            INTEGER);
        List<Entry<String, String>> values =
            m_model.getValue("
            landingSequenceColor");
        JTable table = ((AMANView) m_view).
            getAirplaneLabels();
        ArrayList<String> colors = new ArrayList
            <>();
        // Iterate over the results and set the
            background of each cell
        ...
    }
    public void updateBlockedStatus() {
        m_model.computeValue("blocked",
            LocationType.INTEGER);
        List<Entry<String, String>> value =
            m_model.getValue("blocked");
        JTable t = ((AMANView) m_view).
            getIsLocked();
        IsLockedModel mod = (IsLockedModel) t.
            getModel();
        // Iterate over the results and set the label on
             each button
        ...
        t.repaint();
    }
```

an already occupied time slot. However, when the *Model* receives and processes the command, no update is performed because this would bring the model in an unsafe state. For such a scenario, disabling the command to move the airplane would necessitate the *Model* having an additional Boolean function to store whether the airplane's movement is allowed for each ⟨Airplane, DestinationTimeSlot⟩ combination. We believe that the proposed approach reduces the risk of *over-specification* that happens when a formal specification tries to specify some aspects of a system that could be left undetermined [18].

Code 13 Code running each scenario

```
public void runTestScenario(String scenario) {
    // Define the model
    AsmetaFMVCModel model =
      new AsmetaFMVCModel(MODEL_AMAN);
    AMANView view = new AMANView();
    AMANController cntrllr =
      new AMANController(model, view);
    cntrllr.updateAndSimulate(null);
    view.getTimer().stop();
    view.setVisible(true);
    // Create the test runner
    AsmetaFMVCTestRunner runner =
      new AsmetaFMVCTestRunner(view, cntrllr, scenario,
          50);
    runner.runTest();
}
```

For the same reasons, capturing all behaviors – especially the dynamic ones – by using the *f*MVC approach can be difficult. For example, the implementation we propose in this paper does not handle the drag-n-drop of airplanes along the landing sequence timeline (with the creation of the corresponding "ghost object"), but substitutes this mechanism with two buttons, namely that used to move the selected airplane up and that to move it down. Handling the original behavior would have required considering the coordinates of the mouse pointer in the `Asmeta` model as well, in order to bind them with the position in the landing sequence. Similarly, our *AMAN model* does not consider the dynamism of the flights, such as their insertion or removal from the landing sequence. As reported by the case study description [28], this is handled by external events that we are not considering in our implementation. With a more complex model, however, we believe that external events may be considered as *monitored* functions which could be read by the `Asmeta` model and used to update the landing sequence.

Lastly, in this paper we have applied the *f*MVC approach by using `Asmeta` as the underlying formalism. However, we think that the same principles we exploit for this work can be applied to any other state-based formalism providing a simulator that can be integrated into the application under development.

7 Related work

In this paper, we have presented our approach embedding a formal `Asmeta` specification in the development of UI applications exploiting the MVC pattern. This approach has been introduced in our previous work [6] and has now been extended to better support graphical components, invariants, and model-based test generation. To the best of our knowledge, no prior work has proposed the same kind of approach.

In this paper, we have used the AMAN case study to experiment the integration between a UI and the *f*MVC pattern. For this case study, in the literature, Alloy [17] or Event-B models [20, 23] have been validated using domain-specific views. The work presented in this paper exploits the MVC pattern and the formal specification runs in the final software in the *Model*. Moreover, the approach we present allows for integrating the system behavior with general-purpose programming languages (Java, in the AMAN case study), overcoming the limitations of models written using only formal languages.

In the context of UI development, other approaches attempted to integrate formal models with the MVC pattern, such as in [21, 22]. However, in those works, each MVC component is formally developed by applying stepwise refinement, until the executable code of each component is generated. In particular, the whole approach is formalized using Event-B and relies on the Rodin tools for V&V activities. Thus, formal methods are only used during the design, but no formal specification is embedded in the final executable as, instead, we do in the *Model* component of the *f*MVC approach. Similarly, in [19], the authors generated verified code for UIs. They focused more on modeling and verifying behavioral aspects of user interfaces but did not take into account the MVC pattern. Alternatively, an approach exploiting the MVC approach is presented in [30]. However, the UI generated from the formal specification is generic and differently from *f*MVC it cannot be personalized. Moreover, no description of the *Model* component is given in that approach.

Expanding the analysis to the application of formal methods on the design and verification of GUI and human–computer interaction, [27] presents the contribution of different formal approaches in the field of human–computer interaction. That paper gives a historical perspective of the main contributions in the area of formal methods in the field of human–computer interaction but without any emphasis on UI development.

In this work, we have extended the *f*MVC framework to support the validation of UIs with the same formal model embedded into the final executable. In this way, developers can check whether the `Asmeta` functions have been mapped on the correct graphical components and whether their update is always consistent with the *Model*. Similarly, other formal approaches have been proposed to verify GUIs with formal models. For example, in [2], an approach for deriving the model of the behavior of UIs by dynamic analysis is presented. After the automatic derivation of a formal model, it executes V&V activities. In our work, we follow the opposite path: we first validate the model and we embed it directly within the UI.

Formal methods and tools have been also used for systematically analyzing a control panel interface in [15]: the authors introduce a convenient notation used to describe the interface together with a set of tools allowing the analysis (for what concerns the credibility, feedback, consistency of actions, etc.) of the case-specific interface. Similarly, [25] proposed a language used to describe user interfaces and a Petri nets-based tool for the engineering and development of user interfaces with higher usability and reliability. In [3] a formal approach – supported by a tool – is used to model also the behavior of graphical widgets used in Cockpit Display Systems conforming to the ARINC 661 specification. The authors propose these tools to support prototyping and operation phases, focusing in particular on describing interaction techniques, interactive components, and behavioral parts of interactive applications.

Another tool used to design, prototype, and analyze UIs is PVSio-web [24]. It provides a library of widgets supporting the development of realistic user interfaces. The toolkit is based on the PVS theorem proving system for what concerns the analysis, and the PVS-io component for what concerns the simulation. PVSio-web has been applied successfully to many real-world case studies in critical domains, such as for the analysis of medical devices, to identify latent design anomalies that could lead to use errors. In [16], the toolset has been compared with, CIRCUS and IVY, showing that PVSio-web is more suitable to rapid prototyping using PVSio for formal verification. This makes our *f*MVC approach similar to it, but the formal validation and verification are carried on in `Asmeta`. However, as presented in this work, *f*MVC also supports validation of UIs, by exploiting the same test scenarios used to validate `Asmeta` models.

8 Conclusion

In the development of UIs, architectural patterns are in most cases considered the best approach as they allow to obtain the highest modularity and maintainability of the software. In the case in which the software includes a graphical interface, the MVC pattern is normally adopted since it separates data from the way in which they are shown to the user. However, if the behavior of the software has been previously modeled in a formal way, e.g., because the developers want to validate or verify it before the actual implementation, it cannot be directly embedded into the *Model* component.

To overcome this limitation, in this paper, we have further extended the *formal* Model–View–Controller approach previously introduced in [6]: it integrates an `Asmeta` specification into the *Model* and, thanks to the `AsmetaFMVCLib` library, provides a way to annotate graphical components in the *View* in order to link them to `Asmeta` locations. Moreover, our library includes a *Controller* class allowing the handling of general components, which can be extended to be adapted to case-specific graphical elements. In this paper, we have enhanced the *f*MVC approach by improving the interaction with some Java Swing components, introducing the roll-back of the state of the `Asmeta` *Model* when some invariant is violated by user actions, and by developing the `AsmetaFMVCTestRunner` component. It allows for reusing the same `Avalla` scenarios used for the validation of the `Asmeta` specification to validate the UI by simulating user actions.

We have applied the *f*MVC pattern to the AMAN case study, starting from the modeling and V&V activities, including safety property verification and scenario-based validation, with the tools provided by the `Asmeta` framework, to the implementation of the *View* and its binding with ASM locations. Our UI validation process, executing `Avalla` scenarios on the graphical interface, has shown to be effective and with only 10 scenarios has allowed us to cover all the reachable code.

In conclusion, with the presented experience, we have found that directly using formal models for designing user interfaces may pose several challenges, since the two aspects are very different, and it may be difficult to generalize all the possible interactions. However, especially for prototype implementations, having mechanisms allowing the linking between graphical components and formal models is valuable, and the introduction of automatic testing methodologies supported by the `Asmeta` framework is effective in discovering possible faults in that linking and useful for performing regression testing, as the test scenarios can be re-executed when the *Model* or the *View* change.

Funding information Open access funding provided by Università degli studi di Bergamo within the CRUI-CARE Agreement. This work has been partially funded by PNRR – ANTHEM (AdvaNced Technologies for Human-centrEd Medicine) – Grant PNC0000003 – CUP: B53C22006700001 – Spoke 1 – Pilot 1.4.

References

1. Arcaini, P., Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: The ASMETA approach to safety assurance of software systems. In: Logic, Computation and Rigorous Methods, pp. 215–238. Springer, Berlin (2021)
2. Arlt, S., Ermis, E., Feo-Arenis, S., Podelski, A.: Verification of GUI applications: a black-box approach. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, pp. 236–252. Springer, Berlin (2014)
3. Barboni, E., Conversy, S., Navarre, D., Palanque, P.: Model-based engineering of widgets, user applications and servers compliant with arinc 661 specification. In: Proceedings of the 13th International Conference on Interactive Systems: Design, Specification, and Verification, DSVIS'06, pp. 25–38. Springer, Berlin (2006)
4. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E.: Extending ASMETA with time features. In: Rigorous State-Based Methods, pp. 105–111. Springer, Berlin (2021)
5. Bombarda, A., Bonfanti, S., Gargantini, A.: Automatic test generation with ASMETA for the Mechanical Ventilator Milano controller. In: Lecture Notes in Computer Science, pp. 65–72. Springer, Berlin (2022)
6. Bombarda, A., Bonfanti, S., Gargantini, A.: formal MVC: a pattern for the integration of ASM specifications in UI development. In: Rigorous State-Based Methods: 9th International Conference, ABZ 2023, Proceedings, Nancy, France. May 30 – June 2, 2023, pp. 340–357. Springer, Switzerland (2023)
7. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Asmeta tool set for rigorous system design. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods, pp. 492–517. Springer, Switzerland (2025)
8. Bonfanti, S., Gargantini, A., Mashkoor, A.: AsmetaA: Animator for Abstract State Machines, pp. 369–373. Springer, Berlin (2018)
9. Bonfanti, S., Riccobene, E., Scandurra, P.: A component framework for the runtime enforcement of safety properties. J. Syst. Softw. **198**, 111605 (2023)
10. Bowen, J., Reeves, S.: Formal models for informal GUI designs. Electron. Notes Theor. Comput. Sci. **183**, 57–72 (2007)
11. Bucanek, J.: Model-view-controller pattern. In: Learn Objective-C for Java Developers, pp. 353–402. Apress (2009)
12. Burbeck, S.: Applications programming in Smalltalk-80 (TM): How to use model-view-controller (MVC). Technical report, Smalltalk-80 v2. 5. ParcPlace (1992)
13. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A Scenario-Based Validation Language for ASMs pp. 71–84. Springer, Berlin (2008)
14. Courtney, A.: Functionally modeled user interfaces. In: Jorge, J.A., Nunes, N.J., Falcão e Cunha, J. (eds.) Interactive Systems. Design, Specification, and Verification, pp. 107–123. Springer, Berlin (2003)
15. Creissac Campos, J., Harrison, M.D.: Systematic analysis of control panel interfaces using formal tools. In: Interactive Systems. Design, Specification, and Verification, pp. 72–85. Springer, Berlin (2008)
16. Creissac Campos, J., Fayollas, C., Harrison, M.D., Martinie, C., Masci, P., Palanque, P.: Supporting the analysis of safety critical user interfaces: an exploration of three formal tools. ACM Trans. Comput.-Hum. Interact. **27**(5), 1–48 (2020)
17. Cunha, A., Macedo, N., Kang, E.: Task Model Design and Analysis with Alloy, pp. 303–320. Springer, Switzerland (2023)
18. Dix, A.J.: Formal Methods for Interactive Systems. Computers and People Series. Academic Press, London (1991)
19. Ge, N., Dieumegard, A., Jenn, E., da Ausbourg, B., Ait-Ameur, Y.: Formal development process of safety-critical embedded human machine interface systems. In: Intl. Symposium on Theoretical Aspects of Software Engineering (TASE). IEEE Press, New York (2017)
20. Geleßus, D., Stock, S., Vu, F., Leuschel, M., Mashkoor, A.: Modeling and Analysis of a Safety-Critical Interactive System Through Validation Obligations, pp. 284–302. Springer, Switzerland (2023)
21. Geniet, R., Kumar Singh, N.: Refinement based formal development of human–machine interface. In: Software Technologies: Applications and Foundations, pp. 240–256. Springer, Berlin (2018)

22. Kumar Singh, N., Ait-Ameur, Y., Geniet, R., Méry, D., Palanque, P.: On the benefits of using MVC pattern for structuring Event-B models of WIMP interactive applications. Interact. Comput. **33**(1), 92–114 (2021)
23. Mammar, A., Leuschel, M.: Modeling and Verifying an Arrival Manager Using Event-B pp. 321–339. Springer, Switzerland (2023)
24. Masci, P., Oladimeji, P., Zhang, Y., Jones, P., Curzon, P., Thimbleby, H.: PVSio-web 2.0: joining PVS to HCI. In: Computer Aided Verification, pp. 470–478. Springer, Berlin (2015)
25. Navarre, D., Palanque, P., Ladry, J.-F., Barboni, E.: ICOs. ACM Trans. Comput.-Hum. Interact. **16**(4), 1–56 (2009)
26. Nedyalkova, S., Bernardino, J.: Open source capture and replay tools comparison. In: Proceedings of the International C* Conference on Computer Science and Software Engineering, C3S2E13. ACM, New York (2013)
27. Oliveira, R., Palanque, P., Weyers, B., Bowen, J., Dix, A.: State of the art on formal methods for interactive systems. In: Human–Computer Interaction Series, pp. 3–55. Springer, Berlin (2017)
28. Palanque, P., Campos, J.C.: Aman case study. In: Glässer, U., Creissac Campos, J., Méry, D., Palanque, P. (eds.) Rigorous State-Based Methods, pp. 265–283. Springer, Switzerland (2023)
29. Syromiatnikov, A., Weyns, D.: A journey through the land of model-view-design patterns. In: 2014 IEEE/IFIP Conference on Software Architecture. IEEE Press, New York (2014)
30. Yang, Y., Li, X., Liu, Z., Ke, W.: RM2pt: a tool for automated prototype generation from requirements model. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE Press, New York (2019)