formal MVC: a pattern for the integration of ASM specifications in UI development

Andrea Bombarda[1 [0000−0003−4244−9319]], Silvia Bonfanti[1 [0000−0001−9679−4551]], and Angelo Gargantini[1 [0000−0002−4035−0131]]

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione,
Università degli Studi di Bergamo, Bergamo, Italy
{andrea.bombarda,silvia.bonfanti,angelo.gargantini}@unibg.it

Abstract. Using architectural patterns is of paramount importance for guaranteeing the correct functionality, maintainability and modularity, especially for complex software systems. The model-view-controller (MVC) pattern is typically used in user interfaces (UIs), since it allows the separation between the internal representation of the information and the way it is shown to users. The main problem of using this approach in a formal setting, where formal models are used to specify the requirements and prove safety properties, is that those models are not directly used within the MVC pattern and, thus, all the activities performed at model-level are somehow lost when implementing the UI. For this reason, in this paper, we present 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`, which allows the user to link the formal model with the view and the controller by using simple Java annotations. We present the application of *f*MVC on a simple example of a calculator for explanatory purposes, then we apply it to the AMAN case study, which has inspired the definition of *f*MVC. We discuss the advantages of *f*MVC and its shortcomings, trying to identify the scenarios where it should be applied and possible alternatives.

1 Introduction

When we planned to apply the formal method of our choice, namely the Abstract State Machines (ASM), to the ABZ2023 case study, we realized that the case study differs from the past case studies because it contains a relevant part regarding the user interface (UI) and the interaction with humans. Thus, we decided to evaluate the use of patterns for developing UIs. In that case, one of the most used patterns is the model-view-controller (MVC). MVC separates the UI from the data that it must show. To be more precise, the 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 [4], and it can be easily adapted and tuned for different use case scenarios. 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.

Even in a more formal setting, if one has developed a formal model for the system to be implemented, MVC can be used by deriving part of the code from the formal specification or by using the formal specification as a guideline for developing the MVC (especially for the part related to the controller and the model). However, a direct integration of the formal model is not expected by the existing implementations of MVC.

In this work, inspired by the case study, we devise an extension of the classical MVC, the *formal* Model-View-Controller (*f*MVC) pattern, where the model is a formal specification, an ASM. The way to integrate the model, the view and the controller is provided by a Java library, called `AsmetaFMVCLib`, which allows user to annotate components in the view in order to link them to the input and output locations of the ASM model. The library is integrated in the Asmeta framework [1] and includes the *Model* wrapper (that requires the user only to attach the ASM model) and the *Controller* part, which can be used as they are or extended to be adapted to case-specific behaviors. Moreover, the library provides an interface to be implemented by the *View* component.

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. Moreover, one of the advantage of using Asmeta specifications and, in general, ASMs is that the models are executable and, thus, they can be tested even before having the actual UI.

We apply the proposed pattern to the Arrival Manager (AMAN) case study[1] by showing the whole development process, from the Asmeta model specification, its validation and verification, to the linking between the Java *View* and the *Model*. With this case study, we are able to discuss the advantages and the disadvantages of the *f*MVC, and we highlight the scenarios in which the proposed pattern better fits and those in which alternatives are preferable.

The remainder of this work is structured as follows. Sect. 2 describes the Asmeta framework with its available tools, and gives an example of a simple specification. Sect. 3 introduces the *formal* Model-View-Controller pattern and explains how to use the annotations provided with the `AsmetaFMVCLib` to exploit the proposed approach. Then, in Sect. 4 we report the activities of modeling and V&V of the Arrival Manager (AMAN) case study and application of the *f*MVC pattern. We discuss the main pros and cons of the proposed approach in Sect. 5, and report the related works on integrating formal methods in MVC pattern and, in general, how formal methods are integrated with the UI in Sect. 6. Finally, Sect. 7 concludes the paper.

2 The Asmeta framework

This work is based on the use of Abstract State Machines (ASMs), an extension of Finite State Machines (FSMs) in which unstructured control states are replaced by states with arbitrarily complex data. In particular, we use the functionalities offered by the Asmeta framework [1] which supports the developer with an analysis process spanning the whole life cycle of the system. The three main phases are design, development, and operation, and each phase integrates different tools (see Fig. 1). For this case study,

[1] `https://abz2023.loria.fr/case-study/`

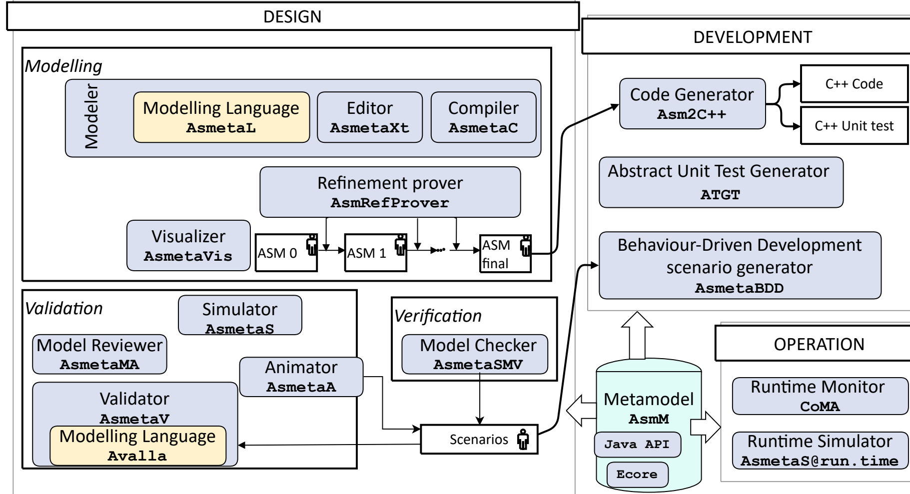

Fig. 1: Phases of The ASM development process powered by the Asmeta framework: design, development and operation.

we only limit to the *design* phase that includes modeling, validation, and verification activities.

ASM *states* are mathematical structures, i.e., domains of objects with functions and predicates defined on them, and the transition from one state s_i to another state s_{i+1} is obtained by firing *transition rules*. Functions are classified as *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are distinguished between *monitored* (only read by the machine and modified by the environment) and *controlled* (read in the current state and updated by the machine in the next state).

An example of an Asmeta specification modeling a simple calculator is shown in Listing 1. It can multiply or sum, depending on the requested operation, the result of the previous operation (initially equals to 1) by a desired number. The desired operation is given by the monitored function `operation` defined in Line 7 and the number inserted by the user is the monitored function `number` defined in Line 8. The result is stored in `calc_result` (line 9) and updated by the two rules running in parallel defined in the main rule (line 13).

With Asmeta, during the modeling phase, the user implements the system models using the `AsmetaL` language and the editor `AsmetaXt` which provides some useful editing support. Furthermore, in this phase, the ASMs visualizer `AsmetaVis` transforms the textual model into graphs using the ASMs notation. The validation process is supported by the model simulator `AsmetaS`, which allows simulating the specification in an interactive mode or by assigning random values to the monitored functions, the model animator `AsmetaA`, the scenarios executor `AsmetaV`, and the model reviewer `As-`

```
1   asm calculator
2   import StandardLibrary
3   signature:
4     // DOMAINS
5     enum domain Operation = {SUM, MULT}
6     // FUNCTIONS
7     monitored operation: Operation
8     monitored number: Integer
9     controlled calc_result: Integer
10
11  definitions:
12  // MAIN RULE
13  main rule r_Main = par
14    if operation = SUM then
15      calc_result := calc_result + number
16    endif
17    if operation = MULT then
18      calc_result := calc_result * number
19    endif endpar
20  // INITIAL STATE
21  default init s0:
22    function calc_result = 1
```

Listing 1: Example of an Asmeta specification for a calculator

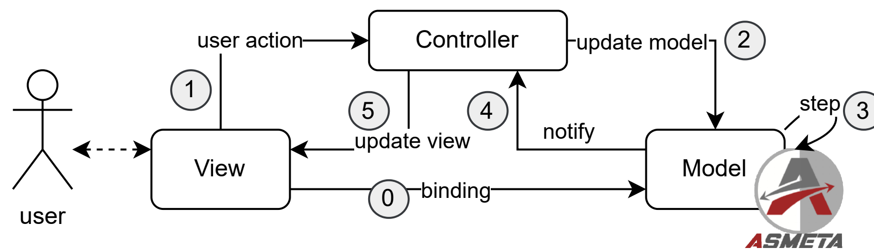

Fig. 2: Formal Model-View-Controller architecture with Asmeta

`metaMA`, which performs the static analysis of the specification and evaluates its quality attributes. Property verification is performed with the `AsmetaSMV` tool. It verifies if the properties derived from the requirements are satisfied by the models. When a property is verified, it guarantees that the model complies with the intended behavior.

3 Formal Model-View-Controller

In this section we explain the overall approach of our *f*MVC framework[2], that is shown in Fig. 2. In our approach, the *View* is a Java graphical container (like a Swing JFrame[3]), with many graphical components that can capture **user actions** (like buttons, text fields, spinners, etc.) and are able to show information regarding the model, including the values of selected controlled locations. The View must implement the interface `AsmetaFMVCView`, which is used to generalize all the possible views and requires the implementation of the method `repaintView` that is called when the GUI needs to be repainted. The *Model* is an Asmeta specification, with its state, including the current values of monitored and controlled functions. In practice, it is an instance of the class

[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`

`AsmetaFMVCModel` that takes an Asmeta file, reads the specification and starts the simulator for the specified ASM model. Finally, the *Controller*, an object that extends the class `AsmetaFMVCController`, controls the flow of information and when it is built, it is linked to the view and the model as well.

Regarding the static part of the architecture, the designer must establish a **binding** between the *View* and the *Model* (step 0 in Fig. 2). This is done by using one or more of the following Java annotations when declaring graphical components in the View:

- `@AsmetaMonitoredLocation`: it links a graphical element (like buttons, text fields, etc.) to a monitored function of the Asmeta model. For each field with this annotation, users must specify the name (`asmLocationName`) of the location in the Asmeta model. The value to be assigned to the `asmLocationName` location can be taken from the graphical element (e.g., if it is a text field) or specified using the annotation attribute `asmLocationValue` (e.g., if it is a button).
- `@AsmetaControlledLocation`: it links a graphical component to specific controlled locations (of the Asmeta model) whose name is specified by the annotation attribute `asmLocationName`.

Graphical elements (like buttons) or timers that generate actions causing an **update of the model** are annotated as `@AsmetaRunStep`. Moreover, if a step requires the GUI to be repainted (e.g., because the number or type of components shown needs to be changed), the flag `repaintView` can be set. When it is created, the *Controller* registers itself as an actionListener and changeListener (whichever is applicable) to all the fields annotated with `@AsmetaRunStep` in the view.

Regarding the dynamics of the pattern, the complete action can be described as follows. When the user performs an action on elements annotated with `@AsmetaRunStep`, the *Controller* handles the request (step 1 in Fig. 2). It takes all the values of the view that are bound to the model (with `@AsmetaMonitoredLocation`), and sets all the monitored functions in the current state of the model with those values (step 2 in Fig. 2). It then executes a step of the ASM model by using the simulator embedded in the *Model* component (step 3 in Fig. 2). When the *Model* is updated, it **notifies** the *Controller* (which is declared as an observer of the *Model*) (step 4 in Fig. 2). Then, the *Controller* **updates** the *View* (step 5 in Fig. 2). First it takes all the values of controlled functions, updates the corresponding graphical elements (those annotated with `@AsmetaControlledLocation`) by calling the method `updateView`, and shows the new values. Then, if it is needed, it repaints the view by calling the method `repaintView`.

3.1 A simple example: an UI for the calculator

In this section, we present how the *f*MVC pattern can be applied to the simple example of Listing 1[4]. We want to provide a UI that allows the user to insert the number in a text field, and to execute the MULT operation by pressing a button. The result is shown in another text field. Moreover, it indicates the sign of the result by using the green background color for positive numbers and red for negative ones. The binding between the *Model* and the *View* is shown in Fig. 3 and described in the following.

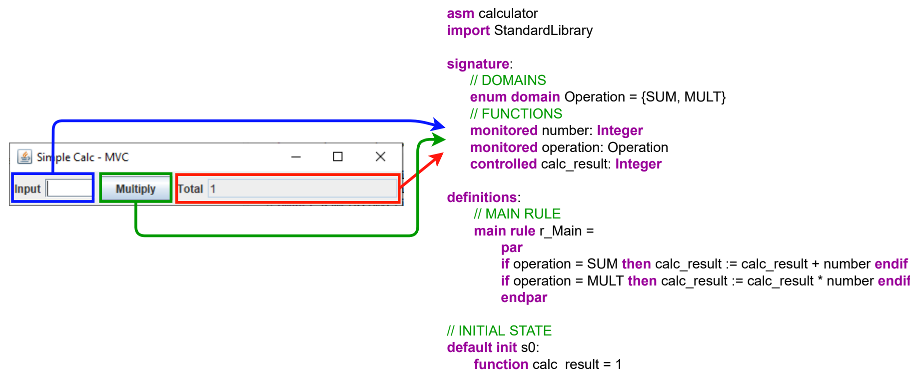

Fig. 3: Bindings between view components and Asmeta locations

The Java code defining the *View* class, implementing the `AsmetaFMVCView` interface, is reported in Listing 2. First, we annotate with `@AsmetaMonitoredLocation` the text field `m_userInputTf`, used to valorize the integer `number` monitored function in the ASM model (see Listing 1). Then, with the `@AsmetaControlledLocation` annotation, we specify that the text field `m_totalTf` shall report the outcome of the ASM computation, stored in the `calc_result` integer controlled function in the ASM model. Finally, using the `@AsmetaRunStep` annotation we set the `m_multiplyBtn` button to be used for requesting the execution of an ASM step. Note that the ASM model (see Listing 1) supports multiple operations (MULT and ADD). For this reason, while executing an ASM step, we need to specify also the operation to be computed. This is done by adding an additional `@AsmetaMonitoredLocation` annotation to the `m_multiplyBtn` button: when the button is clicked, first the `operation` monitored location is set to the `MULT` value (as specified with the `asmLocationValue` field), then the step is performed.

Then, the code for the *Controller* is defined by extending `AsmetaFMVCController`. It redefines the method `update`, which is automatically called when the *Model* notifies a change in values. This extension is needed since the controller in the AsmetaFMVC 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 (such as the color of a text field) has to be managed by the user. This is done by overriding the method `update` as in Listing 3.

The three components are then connected and launched in a main class as in Listing 4.

3.2 Dealing with wrong actions

One of the advantages of the proposed approach is related to the direct use of formal models in the *Model* component of the *f*MVC pattern. In fact, when working with

[4] The source code and the Asmeta model of the multiplier is available online at `https://github.com/asmeta/asmeta_based_applications/tree/main/fMVC/Calculator`

Listing 2: Java Swing View for the multiplier example

```
public class CalcView extends JFrame implements AsmetaFMVCView {
    // bind number with a text field
    @AsmetaMonitoredLocation(asmLocationName="number")
    private JTextField m_userInputTf = new JTextField(5);

    // bind calc_result a the text field
    @AsmetaControlledLocation(asmLocationName="calc_result")
    private JTextField m_totalTf = new JTextField(20);

    // bind operation with a button
    @AsmetaMonitoredLocation(asmLocationName="operation", asmLocationValue = "MULT")
    @AsmetaRunStep
    private JButton m_multiplyBtn = new JButton("Multiply");

    public CalcView() {
        // Adds the component to the Java frame
    }
    @Override
    public void refreshView(boolean firstTime) { }
}
```

Asmeta (see Sect. 2) users may add conditional guards that limit possible input values or invariants that must be always satisfied in every state.

When using the *f*MVC pattern, the update of values shown in the *View* is always made by the *Controller*, based on the value of the controlled functions in the ASM model after the execution of a simulation step. In this way, using the mechanisms embedded into the Asmeta framework, actions can be ignored when they violate invariants: an `InvalidInvariantException` is thrown and caught by the `AsmetaFMVCModel`. Similarly, if a conditional guard is not satisfied, the Asmeta simulator embedded into the *Model* component does not update the corresponding controlled locations during the simulation step. With Asmeta, it is possible to deal also with *inconsistent updates* (i.e., when the same location is updated to two different values at the same time). As for the cases previously presented, if an inconsistent update is found, no update is performed at model level and, thus, no action is executed within the simulation step and the *View* does not change.

The effect of this approach is that only valid actions are executed and valid values handled. Thus, the consistency between the ASM model and the *View* is always assured and they both remain in a safe state.

Listing 3: Controller class for the multiplier example

```
public class CalcController extends AsmetaFMVCController{
    public CalcController(AsmetaFMVCModel m, CalcView v)
    throws IllegalArgumentException, IllegalAccessException {
        super(m, v);
    }
    @Override
    public void update(Observable o, Object arg) {
        // Handle the main locations as regularly done by fMVC
        super.update(o, arg);
        // Set the background color of the result based on the sign
        CalcView v = ((CalcView)this.m_view);
        v.getmTotalTf().setBackground(
            Integer.parseInt(v.getmTotalTf().getText()) >= 0 ?
                Color.GREEN : Color.RED);
    }}
```

Listing 4: Definition of the three components for the multiplier example

```
// Define the model with the Asmeta spec
AsmetaFMVCModel asmetaModel =
    new AsmetaFMVCModel(
        "model/calculator.asm");

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

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

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

4 The AMAN case study

We here explain how we have applied the *f*MVC pattern to the Arrival Manager (AMAN) case study[5]. In particular, we first analyze the modeling and V&V activities that we have performed with Asmeta. Then, we describe how we have implemented the *Controller* and the *View* of our AMAN prototype in order to let them interacting with the Asmeta *Model*.

4.1 Formal (Asmeta) Model

We here describe the structure of the Asmeta model and the requirements that we have covered for the AMAN case study. In particular, in the following, we introduce the modeling strategy that we have adopted, we highlight the details of the models that we have obtained by applying the Asmeta development process, and we describe the properties that we have proved on them.

Modeling strategy As normally done in the Asmeta-based development process, we have modeled the AMAN case study using an iterative design process: initially, a simplified model can be developed and, then, the model is refined by adding further details at a later stage. First, we have specified in the most simple model (in the following identified as $AMAN_0$) all the functionalities that we considered, sometimes in a limited way, except from the time, which is not handled at this level. Then, $AMAN_1$ removes all the limitations introduced in $AMAN_0$ but still does not handle the passing of time. Finally, $AMAN_2$ includes the time management [3] as well and, thus, it can be used as the formal *Model* underlying the AMAN implementation based on the *f*MVC pattern. In the following, we describe in more details the structure and the requirements captured by each Asmeta model.

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

Table 1: Models dimension for the AMAN case study

	Functions				Rules
	Monitored	**Controlled**	**Derived**	**Static**	**Rules**
$AMAN_0$	4	5	0	3	5
$AMAN_1$	6	5	1	4	5
$AMAN_2$	6	9	3	4	7

Listing 5: Asmeta rule handling the moving up of an airplane in $AMAN_0$

```
[...]
domain TimeSlot subsetof Integer
domain ZoomValue subsetof Integer
controlled landingSequence: TimeSlot -> Airplane
controlled blocked: TimeSlot -> Boolean
controlled zoomValue : ZoomValue
static search: Prod(Airplane,TimeSlot) -> TimeSlot
static canBeMovedUp: Airplane -> Boolean
[...]
domain TimeSlot = {0 : 10}
domain ZoomValue = {15 : 45}
[...]
rule r_moveUp($a in Airplane) =
  let ($currentLT = search($a, 0)) in
    if $currentLT != -1 and $currentLT < 10 then
      let ($blk = blocked($currentLT + 1)) in
        if $currentLT < zoomValue and not $blk and canBeMovedUp($a) then par
            landingSequence($currentLT + 1):= $a
            landingSequence($currentLT):= undef
            [...]
          endpar endif  endlet endif endlet
```

Model details Tab. 1 shows the models dimension in terms of number of functions and rules for each refinement level, while further details are given here:

- $AMAN_0$: this model implements the basic functionalities of AMAN. It 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 dimension of 10 time instants. It allows moving airplanes up and down, but only for one time instant at a time, and putting them on hold. For example, we here report in Listing 5 the rule used for moving up an airplane, which checks that, given the current landing time `$currentLT = search($a, 0)`, the destination time instant (`$currentLT + 1`) is not blocked and still allows keeping the desired distance between airplanes.
- $AMAN_1$: this model implements the same functionalities of the previous refinement level, but removes all the limitations we set. Indeed, all the 45 possible future

Listing 6: Asmeta rule handling the moving up of an airplane in $AMAN_1$

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

Listing 7: Asmeta rule handling the time passing in $AMAN_2$

```
[...]
domain Minutes subsetof Integer
controlled timeShown: TimeSlot -> Minutes
controlled lastTimeUpdated : Minutes
[...]
domain Minutes = {0 : 59}
[...]
rule r_update_time_shown = par
  forall $t in TimeSlot do timeShown($t) :=
    mod(currentTimeMins + $t + 1, 60)
  // If times have been shifted, shift all the airplanes too
  if lastTimeUpdated != currentTimeMins then par
    lastTimeUpdated := currentTimeMins
    forall $a in Airplane do r_moveDown[$a, false, 1]
    forall $t in TimeSlot with $t > 0 do
      blocked($t − 1) := blocked($t) endpar endif endpar
```

time instants are shown in the landing sequence, and airplanes can be moved up or down of more than a single time instant. At this refinement level, the rule reported in Listing 5 is modified as shown in Listing 6. Instead of searching the landing time using the static function `search`, we here introduce a derived function `landingTime` which associates to each airplane its corresponding current landing time. Moreover, the rule now uses an additional input parameter `$nMov` which indicates the number of moves to be done.

– $AMAN_2$: this last model refinement implements the handling of time, by exploiting the functionalities offered by the Asmeta TimeLibrary [3]. In this way, the specification can be used as the *Model* within the *f*MVC pattern to show the current time, and it is able to automatically shift the time instants every minute to show the passing of time. The rule handling the time passing is shown in Listing 7.

All the requirements we have captured in the ASM models have been proven using the LTL properties (as described in the following) reported in Tab. 2. Note that the requirements we report are those directly captured by the model, while others (REQ17, REQ18, REQ20, REQ21, REQ22, and REQ23) that are automatically guaranteed by how we have implemented the GUI (i.e., with Java Swing) are not reported.

Safety property verification One of the main advantages in using Asmeta (or, in general, a formal notation) is that the models can be used for proving safety properties. Moreover, if the formal model is directly used in the implementation, the obtained software behavior is correct (w.r.t. the proved properties) by construction.

In this case study, we have proved the safety properties on the $AMAN_0$ model, since the `AsmetaSMV` module exploits the `NuSMV` model checker which is not able to deal with infinite domains (such as the integers used by the Asmeta TimeLibrary [3] to store the time). However, the particular type of refinement used, namely the *stuttering* refinement [1], preserves in the refined model the properties proved for the more abstract one.

Table 2: LTL properties for the AMAN case study

REQ	Description and LTL property
REQ3	Airplanes can be moved earlier or later on the timeline **LTLSPEC** (**forall** $a **in** Airplane, $t **in** Time **with** **g**(search($a, 0) = $t **and** selectedAirplane=$a **and** action = UP **and** canBeMovedUp($a) **implies** **x**(search($a, 0) = ($t + 1)))) **LTLSPEC** (**forall** $a **in** Airplane, $t **in** Time **with** **g**(search($a, 0) = $t **and** selectedAirplane=$a **and** action = DOWN **and** canBeMovedDown($a) **implies** **x**(search($a, 0) = ($t - 1)))))
REQ4	Airplanes can be put on hold by the PLAN ATCo **LTLSPEC** (**forall** $a **in** Airplane, $t **in** Time **with** **g**(search($a, 0) = $t and selectedAirplane=$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 != $t2 **and** search($t1, 0) != -1 **and** search($t2, 0) != -1 **and** **not** **isUndef**(search($t1, 0)) **and** **not** **isUndef**(search($t2, 0))) **implies** ((search($t1, 0)-search($t2, 0)>=3) **or** (search($t1, 0)-search($t2, 0)<=-3))))
REQ6	An aircraft label cannot be moved into a blocked time period **LTLSPEC** (**forall** $a **in** Airplane, $t **in** TimeSlot **with** **g**(search($a, 0) = $t **implies** **not** blocked($t)))
REQ15	The HOLD button must be available only when one aircraft label is selected **LTLSPEC** (**forall** $a **in** Airplane, $t **in** TimeSlot with **g**(search($a, 0) = $t **and** **isUndef**(selectedAirplane) **and** action = HOLD **implies** **x**(search($a, 0) = $t)))
REQ16	The zoom value cannot be bigger than 45 and smaller than 15 **LTLSPEC** **g**(zoomValue >= 15 **and** zoomValue<=45)
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)

Tab. 2 reports the properties we have verified, corresponding to a subset of the AMAN requirements given in the document presenting the case study. In particular the properties we here report are those that can be verified with the aspects we have included in our Asmeta model.

4.2 View

For our experiments, we have implemented a simplified version for the GUI of the AMAN software as in Fig. 4. The mapping between the components and Asmeta locations is reported in Listing 8 and described in the following.

The zoom level is managed using the `zoom` slider, whose value is used to set the `zoom` monitored variable. When the zoom changes, a simulation step is executed and the GUI is repainted (`repaintView = true`) in order to show only the desired number

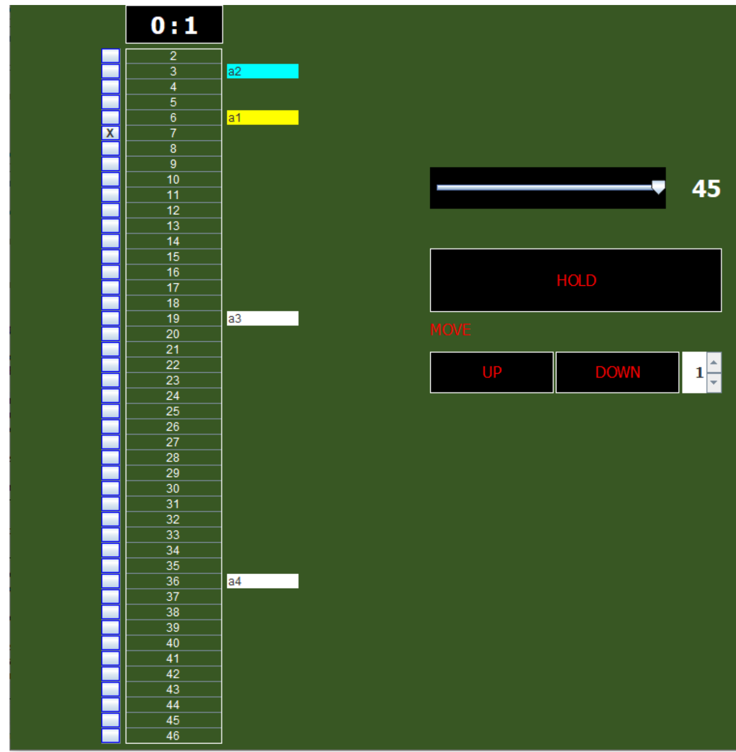

Fig. 4: The GUI of AMAN developed using the *f*MVC pattern

of time instants with the corresponding landing airplanes. Note that, at each simulation step, the ASM model checks which is the action to be executed. For this reason, when the zoom changes, the additional `action` monitored function is set to NONE. Then, a label (`lblZoomValue`) shows the `zoomValue` controlled function containing the current value set for the zoom. We emphasize that its value is set through the model when the slider controlling the zoom is moved, and not directly by the view itself.

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

The hold of an airplane is handled through a button `btnHold` which makes the simulator do a simulation step and sets the `action` to be performed to 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 (up or down) for an airplane is stored in the `numMoves` monitored function through the `spnrNumMoves` spinner. This is a simplification that we have decided to apply w.r.t. the nominal behavior of AMAN, in which the user can drag an airplane label and drop it in the desired location.

```
// Zoom management
@AsmetaMonitoredLocation(asmLocationName = "action",
    asmLocationValue = "NONE")
@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 = "action",
    asmLocationValue = "NONE")
@AsmetaMonitoredLocation(asmLocationName =
    "timeToLock")
@AsmetaRunStep
private ButtonColumn isLockedColumn;

// Timer causing the update of AMAN due to time passing
@AsmetaMonitoredLocation(asmLocationName = "action",
    asmLocationValue = "NONE")
@AsmetaRunStep
private Timer guiTimer;
```

Listing 8: Mapping with the proposed annotation between View components and Asmeta locations

AMAN shows the airplanes approaching the landing runway using the `airplaneLabels` table. This is used both as an output, i.e., it shows the values of the controlled function `landingSequence`, and as an input, i.e., it is used to assign to the `selectedAirplane` monitored function the value of the selected cell, which has to be moved or put on hold. Note that the table showing the landing sequence also handles the background color of each cell, representing the status of an airplane (freezed, stable, or unstable). However, this is a very case-specific aspect, and we have decided to manage it using the controller (see Sect. 4.3). Next to the airplane labels, the following time instants are stored in the `timeShown` controlled function and shown in the `times` table. Similarly, the blocked time instants are reported in the `isLockedColumn` button column (a table with only one column composed of buttons). As for the zoom, `isLockedColumn` sets two different monitored functions, namely `timeToLock` (the index of the time which the user has requested to lock with a click on the button) and `action` (set to NONE, since no move or hold of an airplane is requested). Note that when a button in the `isLockedColumn` is clicked, a simulation step is performed. In this way, the update of the text on the buttons is checked by the model (e.g., we assume that a time instant in which there is an airplane cannot be blocked) and updated by the controller (see

Sect. 4.3). In particular, the blocked time instants are shown with an X on the buttons, while the non-blocked ones do not have any label on the associated button.

Finally, the `guiTimer` is used to refresh the view every minute. For this reason, it sets to NONE the `action` and execute a single simulation step.

4.3 Controller

For adopting the *f*MVC pattern in the AMAN case study, we have extended the `AsmetaFMVCController` included into the `AsmetaFMVCLib` by adding case-specific behaviors for outputs that are not explicitely mapped to graphical components in the *View*. In particular, when the *Controller* 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 Listing 9.

In both the additional setting procedures, the adopted pattern is the same. First, 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 to 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 Discussion

In this section, we discuss the results obtained with the application of the *f*MVC pattern to the AMAN case study and the possible threats to validity. Moreover, we analyze potential benefits in using *f*MVC and possible alternatives adoptable when the proposed solution is not the best fit.

The main threat to the validity of our proposal is *external* [7], which concerns whether we can generalize the results outside the scope of our study, i.e., if the approach we propose in this paper can be applied to other case studies different from AMAN. In this paper, we have presented a first simple example (see Sect. 3.1) in which we have shown that the *f*MVC approach can be applied to a system having different behavior than AMAN. However, since our intention with this paper is to show a methodology, rather than propose a solution that fits in all the possible case studies, the `AsmetaFMVCLib` may be extended in future in order to work with additional graphical components or properties of already supported components. Indeed, the `AsmetaFMVCLib` library supports only a limited number of components (i.e., those we have used in the two proposed examples) and to handle only limited interactions among those normally available in a UI. Nevertheless, we believe that including additional behaviors is easily doable by extending the proposed annotations or their support to new components.

Note that for user interactions (UI components, properties, and actions) supported by the `AsmetaFMVCLib`, using the *f*MVC approach makes the UI development easier. In fact, in that case, if the formal model is already available (e.g., because the specifications have been written for V&V purposes), the user has only to write the view and to link graphical components to model locations.

Listing 9: Controller for the AMAN case study

```
1  public class AMANController extends AsmetaFMVCController {
2
3      public AMANController(AsmetaFMVCModel model, AMANView view)
4          throws IllegalArgumentException, IllegalAccessException { ... }
5
6      @Override
7      public void update(Observable o, Object arg) {
8          // Handle the main parameters as regularly done by the Asmeta FMVCLib
9          super.update(o, arg);
10         // Set the text on buttons based on the value in the TableModel
11         updateBlockedStatus();
12         // Set the color of cells
13         setAirplaneLabelColors();
14     }
15     public void setAirplaneLabelColors() {
16         m_model.computeValue("landingSequenceColor", LocationType.INTEGER);
17         List<Entry<String, String>> values = m_model.getValue("landingSequenceColor");
18         JTable table = ((AMANView) this.m_view).getAirplaneLabels();
19         ArrayList<String> colors = new ArrayList<>();
20         // Iterate over the results and set the background of each cell
21         ...
22     }
23     public void updateBlockedStatus() {
24         m_model.computeValue("blocked", LocationType.INTEGER);
25         List<Entry<String, String>> value = m_model.getValue("blocked");
26         JTable table = ((AMANView) this.m_view).getIsLocked();
27         IsLockedModel model = (IsLockedModel) table.getModel();
28         // Iterate over the results and set the label on each button
29         ...
30         table.repaint();
31     }
```

While performing our experiments and designing the *f*MVC pattern, we felt that the user interface and the formal methods are very different, but it is possible to implement patterns and strategies to link and let them communicate, as we did for the work presented in this paper. However, our impression is that the part which automatize the communication is hardly generalizable, both for the ASM and Java side, since the number of components and properties to be handled is significantly high and users may define their graphical components that are unknown a-priori. Thanks to the experience gained during the work presented in this paper, we can say that having a formal model underlying the actual software is very useful, but having a general controller is not possible. This is the reason why, in the `AsmetaFMVCLib`, we allow users to extend the controller part and to handle in ad-hoc manner additional values.

A threat to *conclusion* validity is that we are experts in using formal methods and in particular the Asmeta framework. Still, we consider the application of *f*MVC to be suitable when the safety of the system is a major concern and for a core critical part of the system. In this case, *f*MVC can benefit from the main advantages of using a formal notation with a precise semantics and with a set of tools for the validation and verification of models.

There are still valuable alternatives for *f*MVC. One consists in transforming the formal specification to source code in a generic programming language and then embed that program in the UI by using a classical MVC pattern in which the model is the generated code. Unfortunately, Asmeta does not support the translation to Java (only C++) for now, but we believe that this path can be viable and we plan to investigate it in future works.

6 Related work

The integration of formal methods in MVC pattern as presented in this paper, has never been proposed, to the best of our knowledge. There exist approaches where the MVC pattern and formal models are combined [9,13] . In those works, each MVC component is formally developed by applying stepwise refinement, until the executable code of each component is generated starting from the formal model previously validated and verified. The whole approach is formalized using Event-B and relies on the Rodin tools for V&V activities. This approach, compared to the one proposed in this paper, does not use the formal model directly as *Model*, but all components are used to generate the initial version of the code. Another approach based on the generation of *verified* code for the UI is presented in [8]. It focuses more on modeling and verifying the behavioral aspect of user interfaces (UIs) and it does not exploit the MVC pattern. A tool that aims at generating MVC prototypes (with the GUI written in Java) from requirements models automatically presented in [14]. The generated UI is generic and differently from *f*MVC it cannot be personalized.

Expanding the analysis to the application of formal methods for design and verification of GUI and human-computer interaction, we have found some relevant works. The contribution of different formal approaches in the field of human-computer interaction is presented in [12]. That paper gives an historical perspective of the main contribu-

tions in the area of formal methods in the field of human-computer interaction without emphasis on the UI development.

A black box approach for the verification of GUIs is presented in [2]. A formal model for the behavior of the GUI application is derived by dynamic analysis (even without the UI code). V&V activities are then performed on the derived model. In our work we try to follow the opposite path: validate first the model to have then the correct UI. Formal methods and tools have been also used for systematically analyzing control panel interface in [5]: the authors propose a convenient notation for describing the interface, and describe a set of tools allowing the analysis (in terms of credibility, feedback, consistency of actions, etc.) of the case-specific interface. Similarly, [11] proposed a user interface description language and a Petri nets-based tool for the engineering and development of usable and reliable user interfaces. These are used to support prototyping phases, for instance when the models and the interactive application evolve significantly to meet late user requirements, as well as the operation phase, after the system is deployed. In particular, the notation proposed can be used to describe interaction techniques, interactive components and behavioral parts of interactive applications.

Another tool used to design, prototype, and analyses UIs is PVSio-web [10]. It provides a library of widgets to support the development of realistic user interfaces. Underneath, the toolkit uses the PVS theorem proving system for analysis, and the PVS-io component for simulation. PVSio-web has been applied successfully to the analysis of medical devices, to identify latent design anomalies that could lead to use errors. A comparison with two other tools, CIRCUS and IVY, showed that PVSio-web is more suitable to rapid prototyping using PVSio for formal verification [6]. This makes our approach *f*MVC similar to it, where the formal validation and verification is carried on in Asmeta.

7 Conclusions

Devoloping UIs using architectural patterns is universally recognized to be the best solution allowing the higher modularity and maintainability of software. Among all the patterns proposed in the literature, the MVC, or one of its variants, is commonly adopted when the software to be developed includes a graphical interface, since it separates data from how they are shown to users. However, the MVC pattern does not well support formal models: even if a *Model* component is present, it is not a formal version. This may limit the reuse of specifications that have been previously written by using a formal notation and does not exploit all the verification and validation activities performed.

For this reason, in this paper, we have proposed the *formal* Model-View-Controller pattern, in which the *Model* is written using Abstract State Machines (ASMs). The pattern is supported by the `AsmetaFMVCLib`, embedded into the Asmeta framework, which allows users to annotate components in the *View* in order to link them to the input and output locations of the ASM model. It includes a wrapper for the *Model*, the *Controller* and an interface to be implemented by the *View*.

In this paper, we have applied the *f*MVC pattern to the AMAN case study, starting from the modeling and V&V activities with the tools provided by the Asmeta framework, to the implementation of the *View* and its binding with ASM locations. Moreover,

we have discussed the pros and the cons of this solution, and highlighted the scenarios in which the proposed pattern better fits and those in which alternatives are preferable. In conclusion, we have found that directly using formal models for designing user interfaces poses 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, as we have done in the work presented here.

References

1. Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. The ASMETA approach to safety assurance of software systems. In *Logic, Computation and Rigorous Methods*, pages 215–238. Springer, 2021.
2. Stephan Arlt, Evren Ermis, Sergio Feo-Arenis, and Andreas Podelski. Verification of GUI applications: A black-box approach. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 236–252. Springer, 2014.
3. Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, and Elvinia Riccobene. Extending ASMETA with time features. In *Rigorous State-Based Methods*, pages 105–111. Springer, 2021.
4. James Bucanek. Model-view-controller pattern. In *Learn Objective-C for Java Developers*, pages 353–402. Apress, 2009.
5. J. Creissac Campos and M. D. Harrison. Systematic analysis of control panel interfaces using formal tools. In *Interactive Systems. Design, Specification, and Verification*, pages 72–85. Springer, 2008.
6. José Creissac Campos, Camille Fayollas, Michael D. Harrison, Célia Martinie, Paolo Masci, and Philippe Palanque. Supporting the analysis of safety critical user interfaces: An exploration of three formal tools. *ACM Transactions on Computer-Human Interaction*, 27(5):1–48, aug 2020.
7. Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research - an initial survey. In *SEKE*, 2010.
8. Ning Ge, Arnaud Dieumegard, Eric Jenn, Bruno daAusbourg, and Yamine Ait-Ameur. Formal development process of safety-critical embedded human machine interface systems. In *Intl. Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, sep 2017.
9. Romain Geniet and Neeraj Kumar Singh. Refinement based formal development of human-machine interface. In *Software Technologies: Applications and Foundations*, pages 240–256. Springer International Publishing, 2018.
10. Paolo Masci, Patrick Oladimeji, Yi Zhang, Paul Jones, Paul Curzon, and Harold Thimbleby. PVSio-web 2.0: Joining PVS to HCI. In *Computer Aided Verification*, pages 470–478. Springer International Publishing, 2015.
11. David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. ICOs. *ACM Transactions on Computer-Human Interaction*, 16(4):1–56, nov 2009.
12. Raquel Oliveira, Philippe Palanque, Benjamin Weyers, Judy Bowen, and Alan Dix. State of the art on formal methods for interactive systems. In *Human–Computer Interaction Series*, pages 3–55. Springer, 2017.
13. Neeraj Kumar Singh, Yamine Ait-Ameur, Romain Geniet, Dominique Méry, and Philippe Palanque. On the benefits of using MVC pattern for structuring Event-B models of WIMP interactive applications. *Interacting with Computers*, 33(1):92–114, jan 2021.
14. Yilong Yang, Xiaoshan Li, Zhiming Liu, and Wei Ke. 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, may 2019.