AnimUML as a UML Modeling and Verification Teaching Tool

Frédéric Jouault
ERIS Team, ESEO, Angers, France
frederic.jouault@eseo.fr

Valentin Sebille
ERIS Team, ESEO, Angers, France
valentin.sebille@eseo.fr

Valentin Besnard
Ateme, Vélizy, France
valentin.besnard.49@gmail.com

Théo Le Calvar
Univ Angers, LERIA, F-49000 Angers, France
DIRO, Université de Montréal
theo.lecalvar@univ-angers.fr

Ciprian Teodorov
Lab STICC, SL Department, ENSTA Bretagne, Brest, France
ciprian.teodorov@ensta-bretagne.fr

Matthias Brun
ERIS Team, ESEO, Angers, France
matthias.brun@eseo.fr

Jerome Delatour
ERIS Team, ESEO, Angers, France
jerome.delatour@eseo.fr

***Abstract*—Practice with feedback is essential to most learning activities. Although invaluable, an instructor's availability to give feedback is necessarily time-limited, but can sometimes be complemented by automated feedback. This is actually the case when learning a new programming language: students can get automated low-level feedback on their production from compilers, interpreters, and program output. However, this is generally not possible when learning a modeling language. Even for UML, which has multiple available execution engines, getting automated feedback from a model's execution requires it to be virtually as precise and complete as a program. In previous work, we presented AnimUML, which makes it possible to animate incomplete and inconsistent models. The work presented here shows how AnimUML works in practice, and how it can be used when teaching modeling. With it, students can observe the behavior or existing models, thus getting a first hands-on experience with the UML semantics. They can then start creating and animating their own models, all along getting a similar level of automated feedback as when learning a programming language. Finally, because it can be connected to a model checker, AnimUML can also help teach model verification.**

***Index Terms*—model animation, model verification, teaching modeling**

I. Introduction

In order to help modelers create UML models, we proposed AnimUML [1][1] at MODELS 2020. AnimUML makes it possible to animate incomplete and inconsistent models, by leveraging specific semantic relaxation points. The rationale is that being able to partially execute a model gives valuable behavioral feedback to users on their models. Since then, we have improved the tool, notably for enhanced model debuggability, and used it with students during UML courses. The work presented here builds on this experience in order to propose a demonstration of AnimUML based on a robot model. More information on the AnimUML principles, as well as a comparison to related work is available in the original paper [1].

The paper is organized as follows. The running example is presented in Section II. Section III presents how model animation works in AnimUML. Model creation is then explained in Section IV. Section V briefly describes model analysis, and finally Section VI gives some concluding remarks.

[1]The current version of AnimUML is available at: https://github.com/fjouault/AnimUML.

II. Running Example

This section presents the UML model example that is used throughout the paper: a robot model. Figure 1 represents its structural diagram. This model contains two actors: `user`, that represents a user, and `simulateurRobot`, that represents a robot simulator. It also contains four objects: `adminUI`, `pilot`, `robot`, and `copilot`, that handle user messages in order to drive the robot. This diagram borrows from the communication diagram in order to show which messages can be sent by which entity (actor or object) to which other entity. For instance, `user` can send the `goAutoMode` message to `adminUI`. Conversely, `adminUI` can send the `displayScreen` message to `user`.

Each of the four objects has a behavior specified as a state machine. Figure 2 gives the state machine of `adminUI`. This object starts in its `mainScreen` state, and can go either to `autoTrip`, if `user` sends `goAutoMode`, or to `manuMode`, if `user` sends `goManuMode`. `user` can also send `backMainScreen` from each of these modes to go back to `mainScreen`. Each of these modes has additional behavior in the form of internal transitions that will not be described here. The other state machines are not shown here as their details are not necessary to understand the tool.

978-1-6654-2484-4/21/$31.00 ©2021 IEEE
DOI 10.1109/MODELS-C53483.2021.00094

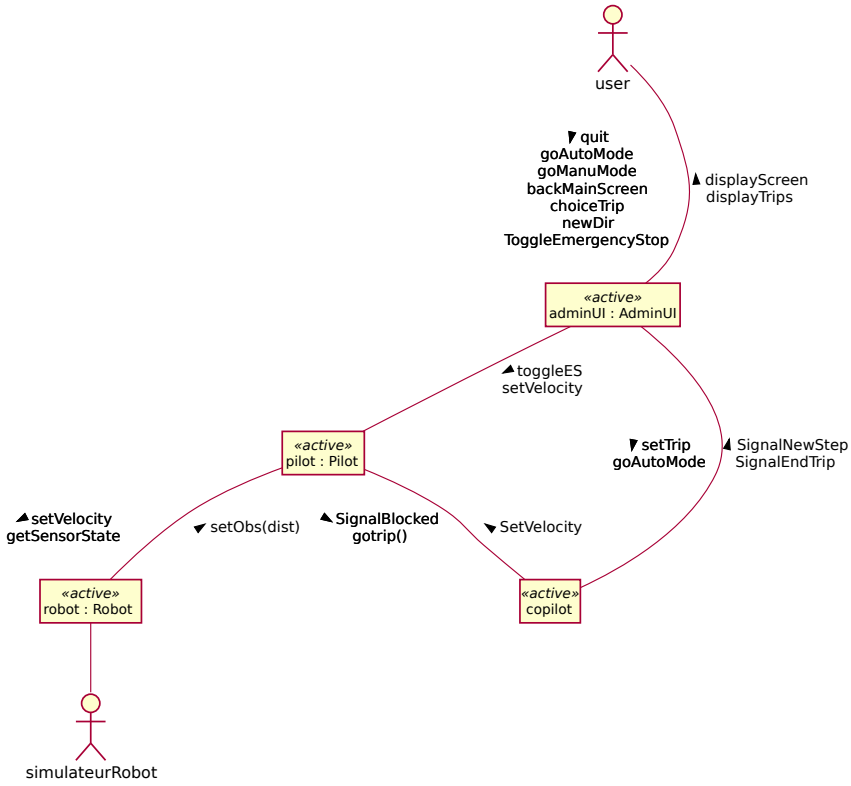

Fig. 1. Structural diagram

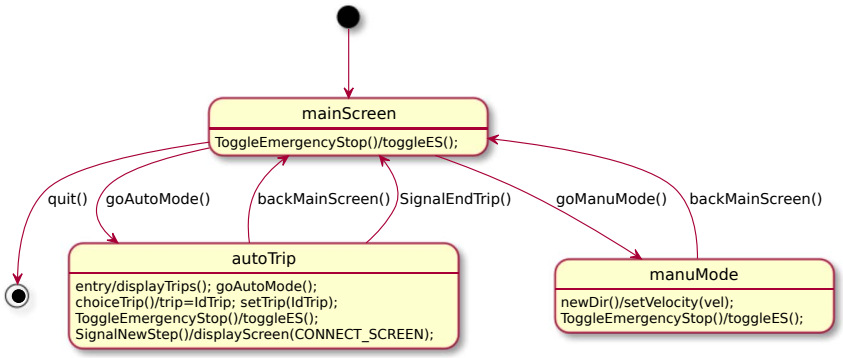

Fig. 2. `AdminUI` state machine

III. Model Animation

The main AnimUML feature is model animation, which can even be performed on incomplete or inconsistent models. Specific semantic relaxation points (see [1]) enable animation to proceed with such models. For instance, AnimUML allows sending messages to unspecified objects. This can be seen on Figure 2, in one of `autoTrip`'s internal transitions, where the `setTrip(idTrip)` signal can be sent without specifying to which object. Such messages end up in a special place called the *Ether* from where they can be consumed by any object. A more complete model could specify to which object the message should be sent to in the following way: `copilot.setTrip(idTrip)`, which would place the message in `copilot`'s event pool, as per the standard UML semantics.

AnimUML can export plain UML diagrams, as seen in Figures 1 and 2, but its user interface shows additional elements. For instance, Figure 3 contains the following AnimUML-specific elements:

- **Operations** are shown on objects, although standard UML would only show them on classes. AnimUML allows this for two main reasons: because 1) it lets

users model objects before modeling classes, and 2) this enables some specific interactions, such as explicitly accepting an operation call.
- **Hyperlinks**, colored in blue and underlined, which can be used to interact with the model.
- The **Ether block** on object `pilot`, which represents messages sent without specifying a target object, and not yet received. Such messages are only shown on objects that can actually consume them using a transition trigger.

Other elements could also be shown, depending on the model, and its current state. These include: property values (like in standard UML), and event pool messages (unlike standard UML, which has no syntax for event pools).

The messages that `user` can send to `adminUI` are shown as hyperlinks on Figure 3 so that they can be sent by clicking on them. All other messages are not shown as hyperlinks because they can be sent by the objects' state machines. Users can choose to always show them as hyperlinks, which can be useful when animating incomplete models. For instance, if a state machine is missing from the model, then users can emulate its behavior by sending messages explicitly using hyperlinks. In this case, because all state machines are already modeled, it makes more sense to only allow objects to send messages. This makes it easier to check whether specific messages can actually be sent by animating the state machines.

While animating a model, a sequence diagram is built by the tool. Figure 4 shows how this diagram looks like for the robot model after initialization. Current state, and sent messages can be seen with the usual sequence diagram syntax. Here, `pilot` can be seen to have sent a `sendMvt` message to the Ether as part of its initial transition. Hyperlinks can be used to select a specific object to be shown, or to go back to a previous state.

Additionally, notes can be shown on each entity so that users can animate the model from the sequence diagram instead of having to click on the structural or state machines diagram. Available actions are represented as hyperlinks, and unavailable actions in stroked out red. After having animated some more steps, the current sequence diagram can then be exported without all the animation hyperlinks and notes.

Figure 5 shows a runtime version of the `adminUI` state machine from Figure 2. It is shown here along with its owning object, so that its event pool can be seen, but this is optional. This runtime state occurs after sending `goManuMode()` to `adminUI` by clicking on the corresponding message on the edge between `user` and `adminUI`. `goManuMode` is shown here to be in the object's event pool rather than in the Ether because it was specifically sent to `adminUI`. The current state: `mainScreen`, is shown with a different background color (green) or shade (if colors cannot be seen). The transitions that have the current state as source are either fireable, when their labels are shown as hyperlinks, or shown with stroked out red text to denote that they cannot be fired. Here, only `goManuMode` has been received by `adminUI`, which is why only the transition that consumes this message is fireable. The other transitions from `mainScreen` are shown with their triggers in stroked out red text because they do not match any available event pool or Ether event. If a transition cannot be fired because its guard is false, then the guard would be shown in stroked out red text instead.

IV. Model Creation

Model animation can already be useful to helper understand a specific model, or even how UML works. However, it is even more useful when creating models because users can then check with animation whether their models have the behavior they intend them to have or not.

They are multiple ways to edit AnimUML models, including a prototype visual editor, and a conversion tool from Eclipse UML (e.g., to work with Papyrus), but the most efficient way to create models is to use the JavaScript object notation. This can be done in JSON[2], but also in regular JavaScript syntax. The latter is more permissive, and also allows multiple strings delimited by backquotes characters. Listing 1 gives an excerpt of the robot model specified as JavaScript code embedded in HTML elements. A model object is first stored into the model variable. A link to AnimUML embedding this model in JSON is then built and used in an iframe. This file can be directly loaded into a web browser, which will load AnimUML with the specified model. State machines may be defined as JavaScript objects, but this is relatively verbose. In order to address this issue, a specific textual syntax inspired from PlantUML[3] can be used (see lines 14 to 27).

Listing 1. AnimUML syntax excerpt

```
1   <script>
2       const model = {
3           "name": "RobotV3",
4           "objects": [
5               {
6                   "name": "user",
7                   "isActor": true
8               },
9               {
10                  "name": "adminUI",
11                  "class": "AdminUI",
12                  "isActor": false,
13                  behavior:`
14                      [*] --> mainScreen
15                      mainScreen --> manuMode : goManuMode
16                      manuMode --> mainScreen : backMainScreen
17                      mainScreen --> autoTrip : goAutoMode
18                      autoTrip --> mainScreen : backMainScreen
19                      autoTrip --> mainScreen : SignalEndTrip
20                      mainScreen --> [*] : quit
21                      mainScreen : ToggleEmergencyStop / toggleES();
22                      autoTrip : entry / displayTrips(); goAutoMode();
23                      autoTrip : choiceTrip() / trip=IdTrip; setTrip(IdTrip);
24                      autoTrip : ToggleEmergencyStop / toggleES();
25                      manuMode : newDir() / setVelocity(vel);
26                      autoTrip : SignalNewStep/displayScreen(CONNECT_SCREEN);
27                      manuMode : ToggleEmergencyStop / toggleES();
28
29                  `,
30                  "operationByName": {
31                      "displayScreen": {},
32                      "displayTrips":{},
33                      "setTrip":{},
34
35                  }
36              },

[...]

204     };
205     document.write(`
206         <iframe
207                 width="100%"
208                 height="100%"
209                 src='http://localhost/AnimUML.html#${
210                         encodeURIComponent(JSON.stringify(model))
211                 }'
212         ></iframe>
```

[2]https://www.json.org/

[3]https://plantuml.com/

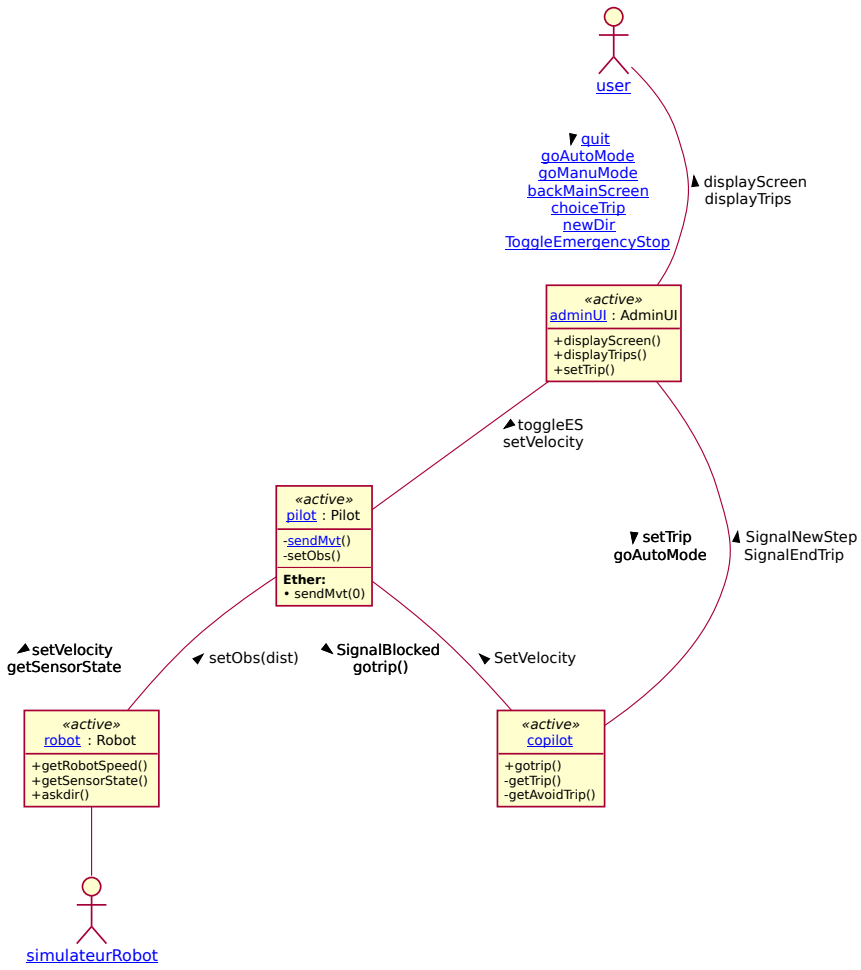

Fig. 3. Structural diagram

The recommended workflow is described in AnimUML's documentation, and is sketched below:

1) Create an HTML file (e.g., YourModel.html) from one of the given templates. The server URL (towards the end of the template) may need to be changed when not using the default server.
2) Load this file in a web browser.
3) If there is a syntax error, go directly to step 5 in order to fix it, otherwise continue with step 4. Remark: some syntax error messages are directly displayed in the browser tab, but others may only appear in the JavaScript console of the browser's web development tools because they are encountered by the browser when loading the page.
4) Evaluate if the displayed model corresponds to what is intended. if that is the case, then this model creation process is finished.
5) Open the HTML file into a text editor (or switch back to an already opened editor window), and modify it.
6) Use the browser's refresh button to reload the model.
7) Go back to step 3.

We used this approach with students by asking them to model a subset of the cruise control case study from [2]. They were able to have a working model within a couple of hours. It would be interesting to precisely evaluate the respective impacts of using a textual syntax, and of being able to animate the model. But from this limited experience, we were satisfied with the students' results, and the speed at which they were able to achieve them.

V. Model Analysis

AnimUML offers multiple model analysis tools that range from simple watch expression, to state space exploration, and even LTL model checking. All of these activities can be performed from the web browser, and mostly run within the web browser itself, except for the LTL model checking. Because we have not implemented model checking algorithms in the browser, we must rely on an external model checker: OBP[4]. The connection between AnimUML in the web browser, and

[4]http://www.obpcdl.org/

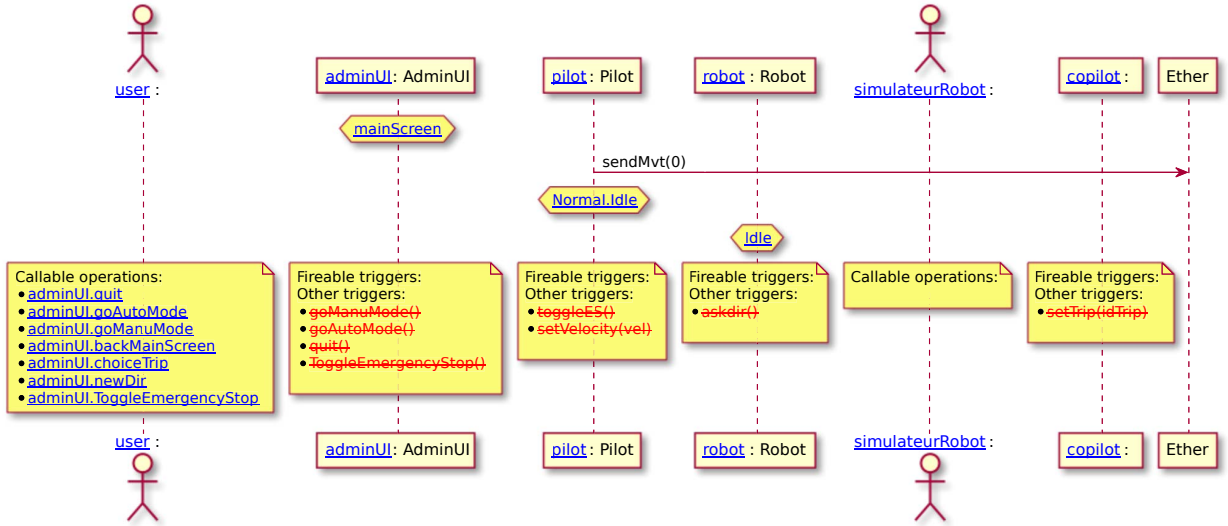

Fig. 4. Driving model animation from sequence diagram

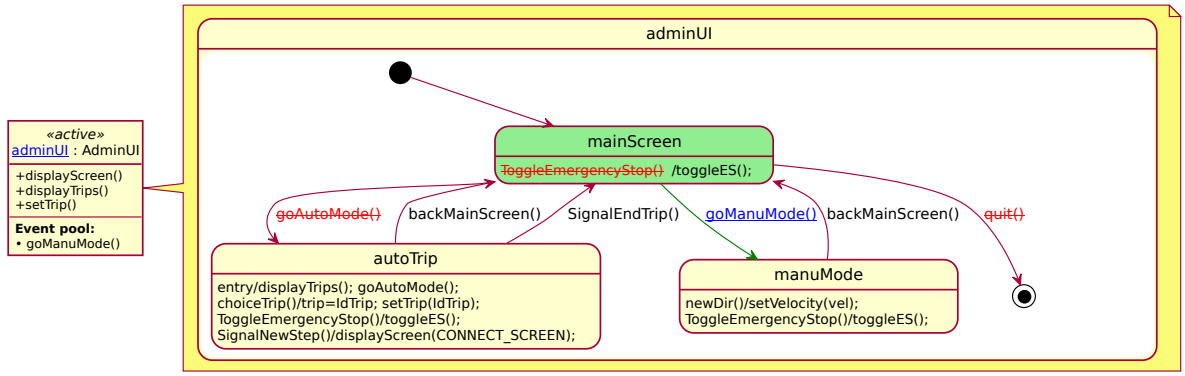

Fig. 5. `AdminUI` state machine for animation

OBP running in a Java Virtual Machine is performed via Web Socket[5].

We asked the same students who modeled the cruise control case study from [2] to additionally verify a few LTL properties on their models. This was done as part of a formal modeling class, and most students managed to verify their models with AnimUML.

VI. Conclusion

This paper has briefly presented AnimUML, and some of its advantages when teaching UML modeling. We believe that partial model execution can actually not only help students, but also anyone creating UML models. Other features than those described here are also available, such as code generation. Finally, we are working on new ways to make AnimUML more attractive to students, such as connecting it to Webots[6] simulators.

[5]https://datatracker.ietf.org/doc/html/rfc6455
[6]https://cyberbotics.com/

References

[1] F. Jouault, V. Besnard, T. Le Calvar, C. Teodorov, M. Brun, and J. Delatour, "Designing, Animating, and Verifying Partial UML Models," in *23rd International Conference on Model Driven Engineering Languages and Systems (MODELS 2020)*, Virtual event, Canada, Oct. 2020, pp. 211–217. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02931876

[2] F. Houdek and A. Raschke, "Adaptive exterior light and speed control system," in *Rigorous State-Based Methods*, A. Raschke, D. Méry, and F. Houdek, Eds. Cham: Springer International Publishing, 2020, pp. 281–301.