# Model-Driven Gamified Software Modelling Learning

Alfa Yohannis*, Dimitris Kolovos, Fiona Polack
Department of Computer Science
University of York
York, United Kingdom
Email: *ary506@york.ac.uk

***Abstract*—In**

***Keywords*—gamified approach, software modelling learning, model-driven engineering**

## I. Introduction

## II. Rationale

## III. Design

### A. Gameful Design

### B. Pedagogical Design

### C. Model-Driven Design

## IV. Demonstration

In this section, an example is presented to demonstrate the use of the platform. The example is to show the of teaching the concept of substates in UML statechart modelling. In this demonstration, designers are assumed to have proficient knowledge and skills of statechart diagram while learners are assumed already understood the concept of states, transitions, start state, and final state. The following subsections are the order of the statechart from its definition to be fully used in a game play.

### A. Define Statechart Metamodel

In order for the platform to support modelling in statechart diagram, designers have to define its metamodel first. The metamodel is written in Emfatic (https://www.eclipse.org/emfatic) and Eugenia-like annotations [1] are used to define its concrete syntax. For example, the definition of Start State derived from Node class is displayed in Fig. 1. Other basic elements of statechart diagram, such as states, transitions, end states, etc., are also defined but not displayed in the figure.

```
abstract class Node extends Entity {
  ref Edge[*]#source outgoing;
  ref Edge[*]#target incoming;
}

@gmf.node(mxLabel="name", mxShape="startState",
     mxWhiteSpace="wrap", mxHtml="1", mxFillColor="#000000",
      mxWidth="30", mxHeight="30")
class Start extends Node {
}
```

Fig. 1: A definition of Start State derived from Node class using Emfatic and Eugenia-like annotations.

The annotated metamodel then trasformed using Epsilon [2] and EMF (http://www.eclipse.org/modeling/emf/) to generate the Java codes of the metamodel, as the backbone codes for further model operations, and Javascript codes to display the statechart's elements as visual elements in a palette of an MxGraph-based graphical editor (https://jgraph.github.io/mxgraph/).

### B. Create Statechart Model

Using the MxGraph-based graphical editor (Fig. 2), designers can now create statechart models. The editor has a pallete on the left side that contains elements of statechart's elements which they can drag and drop into a drawing area on the centre where they can arrange the elements to construct statechart models. The editor also has a property panel on the right side to modify the attributes and appearance of the models. Models that have been created can be saved and loaded again for further operations. The editor also can be used to create models as input/base models to be embedded in learning activities.

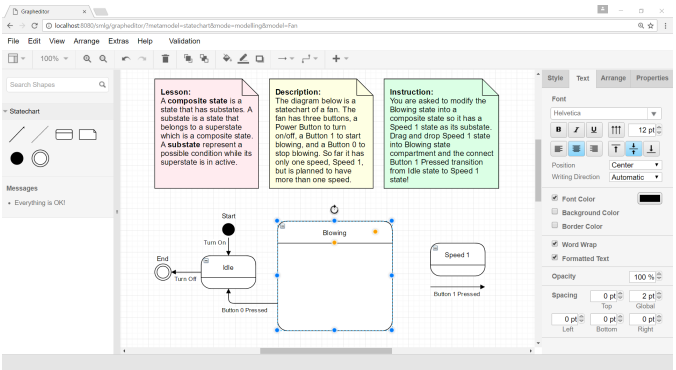



Fig. 2: The MxGraph-based graphical editor to create statechart models.

### C. Create Learning Pattern Model

To teach and learn a concept, designers have to design a learning pattern that consists of ordered learning activities. In this example, a learning pattern to learn the concept of substates is created (Fig. 3). The case that is selected is an electrical fan that has two main states, Idle and Blowing. Learners will be asked to modify the Blowing state so it

becomes composite state that has several substates that represent different speed of blowing. The learning pattern has three activities, namely One-Speed Fan, Two-Speed Fan, and Tree-Speed Fan. The activities are arranged *per se* to accommodate Flow state [3] so learners can start learning from the easiest activity to the hardest one. The activities are explained in more detail in the next following subsections.

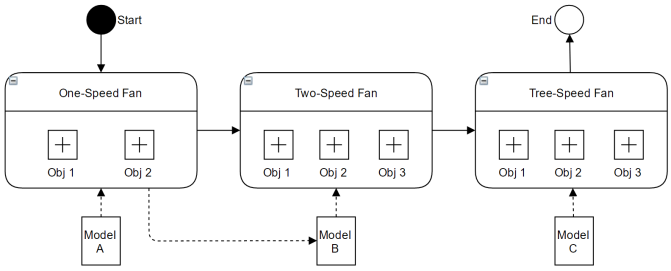



Fig. 3: A learning activity pattern for learning the concept of Substates in state-machine modelling.

Each activity has lesson and instruction properties. Lesson contains explanation about the concepts that are being taught and instruction contains commands or questions that learners need to execute or answer. In the process of satisfying the instruction in each activity, one or more objectives have to be met by learners in order to move to next activity. Also, each activity can consume existing models (Model A, B, and C in Fig. 3) as its base models so learners do not have to create a model from scratch and produce model (Model B in Fig. 3) to be used in its next activity. Each of the models also has a description property to describe itself which is very useful for learners to understand about the model. An example of the lesson, model description, and instruction properties of the One-Speed Fan activity (Fig. 3) is displayed in Fig. 4.

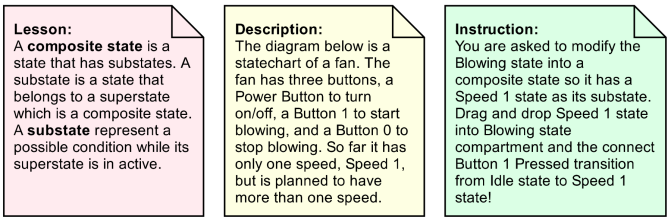
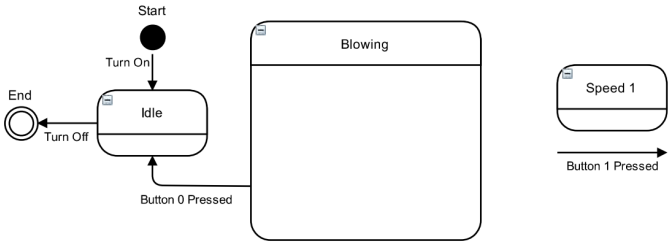


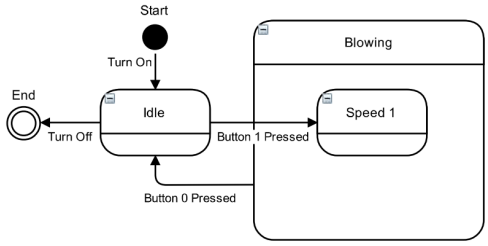

Fig. 4: Lesson, model description, and instruction.

*1) One-Speed Fan Activity:* In this activity (Fig. 5), learners are introduced to one substate only.The activity start with a base model and learners are required to modify the base model (Fig. 5a) to meet the target model (Fig. 5b). The base model corresponds to Model A in Fig. 3, which refers to an existing model that already created previously and will be loaded once the activity is executed so leaners do not need to create the model from the start.

The example case in this activity is a fan that has three buttons, a Power Button to turn on/off, a Button 1 to start blowing, and a Button 0 to stop blowing. So far it has only one speed, Speed 1, but is planned to have more than one speed. Learners are asked to modify the Blowing state in Fig. 5a into a composite state through moving the Speed 1 state into the Blowing state compartment as well as to connect the Button 1 Pressed transition from the Idle state to the Speed 1 state. Since in Fig. 3 this activity is designed to have only two objectives, the two objectives are adjusted and defined as follow: Objective 1 "Blowing state contains Speed 1 substate" and Objective 2 "Button 1 Pressed transition connects Idle state to Speed 1 substate".




(a) Base model

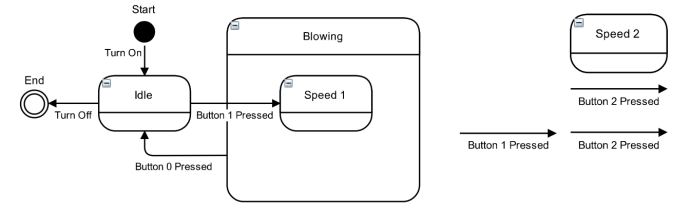



(b) Target model

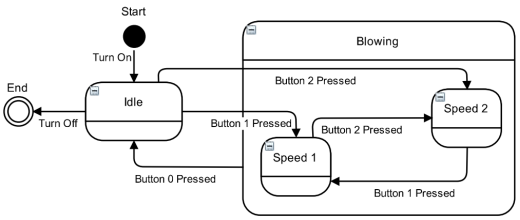
Fig. 5: The One-Speed Fan example.

*2) Two-Speed Fan Activity:* The Two-Speed Fan activity (Fig. 6) consumes the model that has been produced in the first activity (Model B in Fig. 3). Therefore, any model produced in the previous activity will become the base model for modelling in this activity. Inline to the Flow concept [3], this second activity has to be more challenging. Therefore, the activity (Fig. 6) challenge learners with one additional state and three new transitions. Now the case has changed. The fan has an additional button, Button 2, to support 2-speed blowing. When Button 1 is pressed, the fan blows in speed 1. When Button 2 is pressed, the fan blows in speed 2. Thus, Learners are required modify the Blowing state into a composite state so it has two speed states. The fan can move from the Idle state to the Speed 1 state, from the Idle state to the Speed 2 state, and from the Speed 1 state to the Speed 2 state or *vice versa*.

*3) Three-Speed Fan Activity:* The Three-Speed Fan activity (Fig. 7) should be more difficult than the second activity. The fan now supports 3 speeds of blowing, but it has been modified so it cannot go directly to Speed 2 and Speed 3 without firstly go the state with lower speed. In other words, transition from Idle can only go to Speed 1 for the fan to start blowing–Speed 2 and Speed 3 will not be working if transition comes from Idle state. Thus, starting from the model C as the base model as shown in Fig. 3, learners are required to modify the Blowing state into a composite state so it has 3 speeds of blowing and



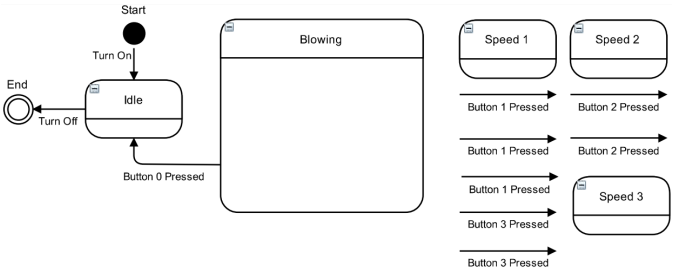

(a) Base model



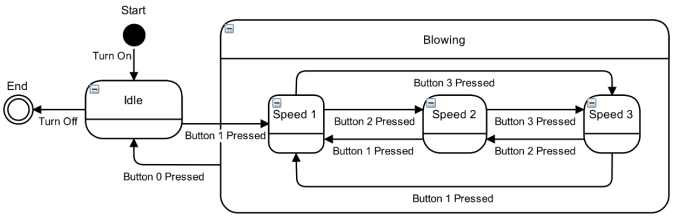

(b) Target model

Fig. 6: The Two-Speed Fan example.

only allow transition from Idle to Speed 1 and from a speed that is lower from the intended one in order to blow.




(a) Base model




(b) Target model

Fig. 7: The Three-Speed Fan example.

### D. Generate Game

After finishing constructing the learning pattern (Fig. 3), designers can now generate a path of levels (or stages) that is usually found in many games. Each generated level corresponds to an activity defined in the learning pattern. The generation also produces an EVL [4] template for each level for its validation to determine whether the level has been completed, all of its objectives have been met, or not. The EVL template is shown in Fig. 8 which can be extended by designers to write code that fits with the level scenarios and objectives. The number of constraints and operations in the template corresponds to the number of objectives defined in the level's learning pattern activity in Fig. 3. As an example implementation of validation, Fig. 9 shows the EVL code of checking whether Blowing state has already Speed 1 substate as intended by Objective 1 in One-Speed activity.

```
context Statechart {
    constraint obj_1 {
        check:
            self.obj_1()
        message:
            "FAIL: ob_1"
    }
    constraint obj_2 {
        check:
            self.obj_2()
        message:
            "FAIL: obj_2"
    }
}
operation Statechart obj_1(): Boolean {
    return true;
}
operation Statechart obj_2(): Boolean {
    return true;
}
```

Fig. 8: Validation template for objectives in One-Speed Fan activity/level.

```
context Statechart {
    constraint obj_1 {
        check:
            self.obj_1()
        message:
            "FAIL: Blowing state contains Speed 1 substate"
    }
    ...
}
operation Statechart obj_1(): Boolean {
    for (state in State.allInstances.select(state | state.
         name == "Blowing")) {
        if (state.substates.notEmpty() and state.substates.
            select(substate | substate.name == "Speed 1").
            notEmpty()){
            return true;
        }
    }
    return false;
}
...
```

Fig. 9: Validation realisation for Objective 1 in One-Speed Fan activity/level.

### E. Play Game

After generating the learning pattern into a path of ordered levels and defining its levels' validation, Learners can choose the path and play its levels as depicted in Fig. 10. When learners choose the first level, an MxGraph-based editor is displayed. Lesson, model description, instruction, objectives, and base model are presented to learners. Learners then can start to modify the base model to reach the target model.

Every attempt to change the model will trigger an operation to validate the current model using the defined EVL constraints to assess whether the current model has met the current level's objectives. If not, error messages are displayed to learners

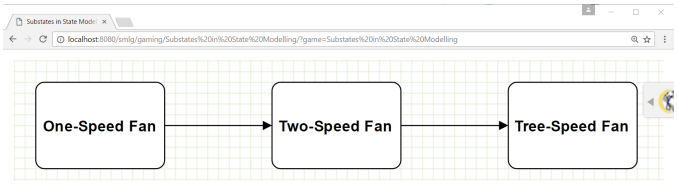



Fig. 10: The path for learning substate concept with its levels.

indicating the objectives that have not been satisfied. For example, in Fig. 11, the Speed 1 substate has not been put into the Blowing state even though the Button 1 Pressed transition has connected the Idle state to the Speed 1 substate. If all objectives have been fulfilled, learners have completed the level. A message that congratulate the learners is displayed to give positive reinforcement. Learners then are brought back to the learning path where they can choose the next level to play.

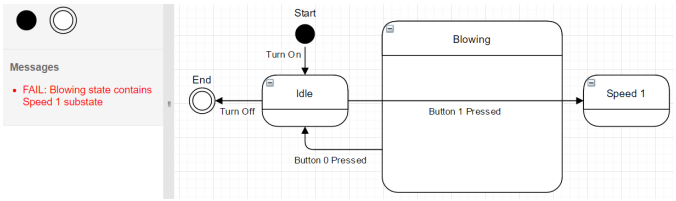



Fig. 11: A message is displayed to learners indicating the objective that has not been fulfilled.

## V. Discussion

## VI. Future Works

## VII. Conclusion

The conclusion goes here.

## Acknowledgment

The authors would like to thank...

## References

[1] D. S. Kolovos, A. García-Domínguez, L. M. Rose, and R. F. Paige, "Eugenia: towards disciplined and automated development of gmf-based graphical model editors," *Software & Systems Modeling*, pp. 1–27, 2015.

[2] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez, "The epsilon book," *Structure*, vol. 178, pp. 1–10, 2010.

[3] M. Csikszentmihalyi, "Toward a psychology of optimal experience," in *Flow and the foundations of positive psychology*. Springer, 2014, pp. 209–226.

[4] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Eclipse development tools for epsilon," in *Eclipse Summit Europe, Eclipse Modeling Symposium*, vol. 20062, 2006, p. 200.