# Create and Play your Pac-Man Game with the GEMOC Studio (Tool Demonstration)

Dorian Leroy, Erwan Bousse, Manuel Wimmer, Benoit Combemale, Wieland Schwinger

## To cite this version:

Dorian Leroy, Erwan Bousse, Manuel Wimmer, Benoit Combemale, Wieland Schwinger. Create and Play your Pac-Man Game with the GEMOC Studio (Tool Demonstration). EXE 2017 - 3rd International Workshop on Executable Modeling, Sep 2017, Austin, United States. pp.1-6. hal-01651801

**HAL Id: hal-01651801**

**https://hal.inria.fr/hal-01651801**

Submitted on 14 Dec 2017

# Create and Play your Pac-Man Game with the GEMOC Studio

*(Tool Demonstration)*

Dorian Leroy*, Erwan Bousse†, Manuel Wimmer‡, Benoit Combemale§, Wieland Schwinger¶
*¶JKU Linz (Austria), †‡TU Wien (Austria), ‡CDL-MINT (Austria), §University of Toulouse (France)
dorian.leroy@jku.ac.at, erwan.bousse@tuwien.ac.at, wimmer@big.tuwien.ac.at,
benoit.combemale@irit.fr, wieland.schwinger@jku.ac.at

***Abstract*—Executable Domain-Specific Languages (DSLs) are used for defining the behaviors of systems. The operational semantics of such DSLs may define how conforming models *react* to stimuli from their environment. This commonly requires adapting the semantics to define both the possible domain-level stimuli, and their handling during the execution. However, manually adapting the semantics for such cross-cutting concern is a complex and error-prone task. In this paper, we present an approach and a tool addressing this problem by augmenting the operational semantics for handling stimuli, and by automatically generating a complete *behavioral language interface* from this augmentation. At runtime, this interface can receive stimuli sent to models, and can safely handle them by interrupting the execution flow. This tool has been developed for the GEMOC Studio, a language and modeling workbench for executable DSLs. We demonstrate how it can be used to implement a Pac-Man DSL enabling the creation and execution of Pac-Man games.**

***Index Terms*—Model Execution; Reactive DSLs; Code Generation**

## I. Introduction

A large number of Domain-Specific Languages (DSLs) geared toward the description of the behavior of systems (*e.g.,* [1], [2], [3], [4], [5]). Enabling the execution of their conforming models allows to make the most out of those models. This requires the definition of the *execution semantics* of these languages, including how conforming models react to *stimuli* from their environment [6]. This includes the definition both of possible domain-specific events—*i.e.,* the different types of stimuli in the considered domain—and of how occurrences of said events are to be handled.

But incorporating events handling logic within operational semantics is a difficult task, as it impacts both the content (*e.g.,* adding event-processing code in existing execution rules) and the scheduling of execution rules (*e.g.,* defining instants in the execution when stimuli should be handled). In addition, at runtime, it is necessary to provide an *interface* to allow external actors (*e.g.,* a simulator, a test engine, other models, etc.) to send event occurrences to models being executed. Depending on how a semantics is structured, this may require a mechanism to temporarily *interrupt* the execution of the model (similarly to *interruptable models* in the DEVS formalism [6]), in order to react to event occurrences by triggering their handling. Overall, manually defining such interface and its integration with the semantics can be a tedious and error-prone task, which must be repeated for each executable DSL.

The demonstrated tool aims to solve this problem. It is developed as an extension to the GEMOC Studio [7], an Eclipse-based language and modeling workbench for executable DSLs. It provides a non-intrusive and modular way to define both the possible domain-specific events and their handling logic within the operational semantics of a DSL. It then generates an interface to safely send event occurrences to a model being executed, *i.e.,* only when the model is in a consistent state. An extension to the execution environment has been developed to use this interface. The use of this tool is illustrated with a Pac-Man DSL ([8], [9]) allowing to define customized versions of the world-famous Pac-Man video game, which can then be played. The tooling as well as the example presented in this paper are available on Github[1][2].

The remainder of this paper is structured as follows. In Section II, we provide an overview of the architecture of the tool. Section III details the Pac-Man use case. Finally, future research directions are given in Section IV.

## II. Architecture

In this section we present the architecture of the tool, which is developed as a reactive extension to the GEMOC Studio and is written in Java and Xtend.

### *A. Executable DSLs and Event Handlers Annotation*

The tool presented in this paper supports DSLs whose abstract syntax is provided as a *metamodel* and whose execution semantics is provided as an *operational semantics*. The considered operational semantics can be decomposed in *i)* a data structure representing the *model state* and *ii)* a set of *execution rules*. The model state is defined in an *execution metamodel* extending the abstract syntax metamodel. Before the execution, models are initialized through a transformation from the abstract syntax to the execution metamodel. Model execution is performed by an endogenous, in-place transformation on this model state, using the set of execution rules constituting the operational semantics of the DSL. The approach relies on providing an annotation mechanism to tag execution rules as *event handlers*. This allows the identification of both events and their corresponding handler. In the GEMOC Studio, Ecore [10] is used as a metamodeling language to define

[1]https://github.com/eclipse/gemoc-studio
[2]https://github.com/tetrabox/pacman-example.git

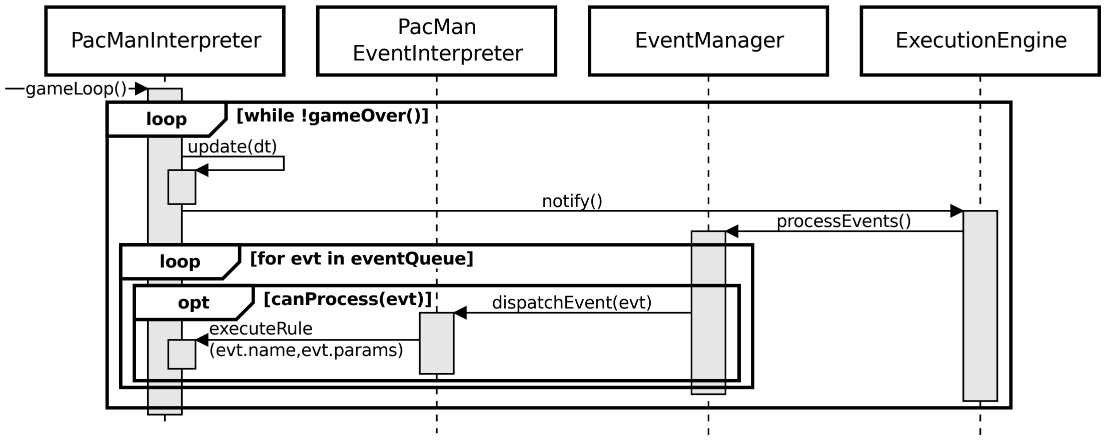


Fig. 1: Sequence diagram of a call to manageEvents.

the abstract syntax and the execution metamodel of DSLs. One goal of the underlying approach is to support multiple metaprogramming languages to define the operational semantics of DSLs. We support the Kermeta language [11] and the xMOF language [12]. For example, in Kermeta, an `@Step` annotation allows to tag methods as execution steps. This annotation has been extended with the `eventHandler` boolean parameter. This parameter fulfills the role required to tag execution rules as event handlers.

### B. Execution Engine

Within the GEMOC Studio, model execution is orchestrated by a component called the *execution engine*. Among other things, this component is responsible for starting and stopping the execution. It is notified whenever the execution of a rule is about to begin or comes to an end. During such notifications, the engine guarantees that there is no ongoing atomic execution step, which means that the executed model is in a *consistent state*. A component can be notified each time the model reaches such a state by registering as a listener to the execution engine. Figure 1 shows how we leverage the notifications sent to the execution engine to introduce event processing in the execution loop. The DSL (Pac-Man in Figure 1) interpreter executes the rules of the operational semantics, and notifies the execution engine when such a rule is executed (here the *update* rule). The execution engine then delegates the handling of events to the event manager, through the *processEvent* service. In turn, the *event manager* iterates over its event queue and delegates for each event the call of the corresponding execution rule to the (generated) DSL *event interpreter*, through the *dispatchEvent* service. These two components are detailed below.

### C. Behavioral Language Interface

The behavioral language interface of an executable DSL allows external actors to send stimuli to executed models. It can be decomposed as a *domain-specific event metamodel* and a *domain-specific event interpreter*. These artifacts are both generated from the annotated execution semantics of the DSL by a generator implemented in Xtend taking the definition of the DSL as input. This generator uses the annotation mechanism provided by the approach to detect execution rules that are event handlers. The outputs of the generator are an Ecore event metamodel and a Java class providing the *dispatchEvent* service. As shown on Figure 1, this service retrieves the parameters stored within the provided event model and calls the corresponding execution rule with these parameters.

### D. Event manager

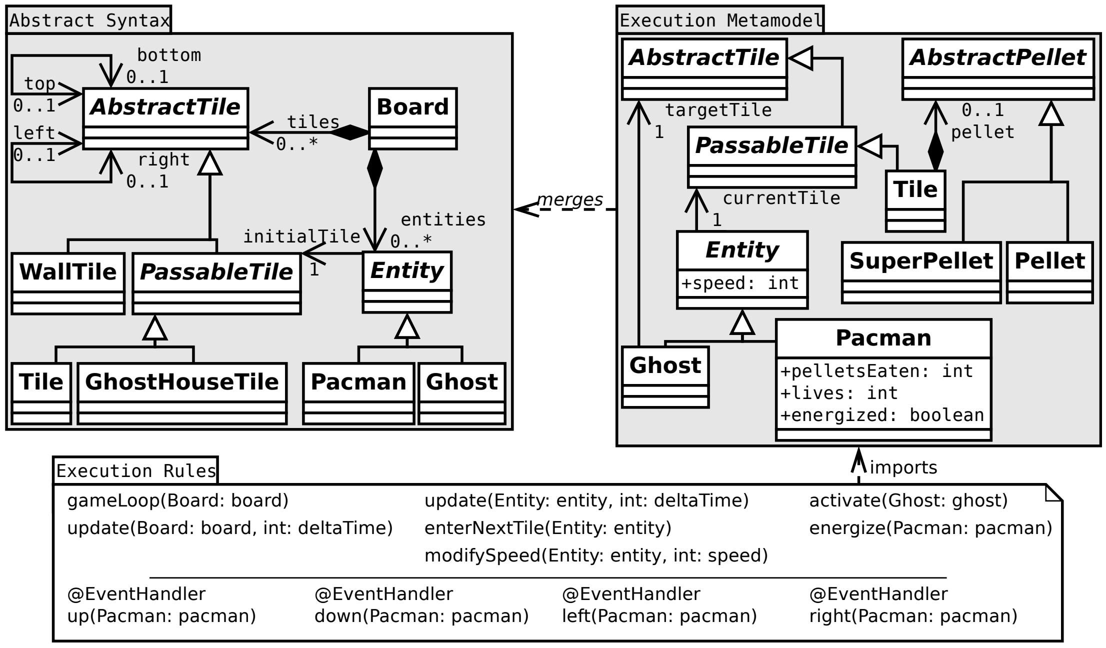
The event manager is the component linking the execution engine to the generated event interpreter. It has a dynamic state composed of the *event queue* which is a list of models conforming to the domain-specific event metamodel (*i.e.,* domain-specific stimuli) part of the behavioral language interface. The event manager provides the *queueEvent* service, which is used by external components to push event occurrences to the event queue. The *processEvents* service of the event interpreter is called by the execution engine each time the execution reaches a new consistent state. It leads to the temporary interruption of the planned scheduling of execution rules in order to call the handling rule corresponding to each event occurrence in the event queue.

### E. External Components

External components use the behavioral language interface to interact with executed models, through the event manager. These components can listen to the execution by registering as listeners to the execution engine. This way they are notified when the execution starts, stops or reaches a consistent state, which allows them to perform their intended task with reliable data (*i.e.,* the model's dynamic state). These components also have access to the services provided by the event interpreter to check if particular stimuli can be processed in the current execution state and to push stimuli to the queue. The Pac-Man GUI (presented below) is such an external component, refreshing the view when receiving notifications from the engine, and forwarding the user's inputs to the behavioral interface of the Pac-Man DSL under the form of instances from its event metamodel.

## III. The Pac-Man example

In this section we detail the Pac-Man DSL used in this tool demonstration.



```
gameLoop(Board: board)                       update(Entity: entity, int: deltaTime)      activate(Ghost: ghost)
update(Board: board, int: deltaTime)         enterNextTile(Entity: entity)               energize(Pacman: pacman)
                                             modifySpeed(Entity: entity, int: speed)

@EventHandler            @EventHandler            @EventHandler            @EventHandler
up(Pacman: pacman)       down(Pacman: pacman)     left(Pacman: pacman)     right(Pacman: pacman)
```

Fig. 2: Overview of the Pac-Man DSL.

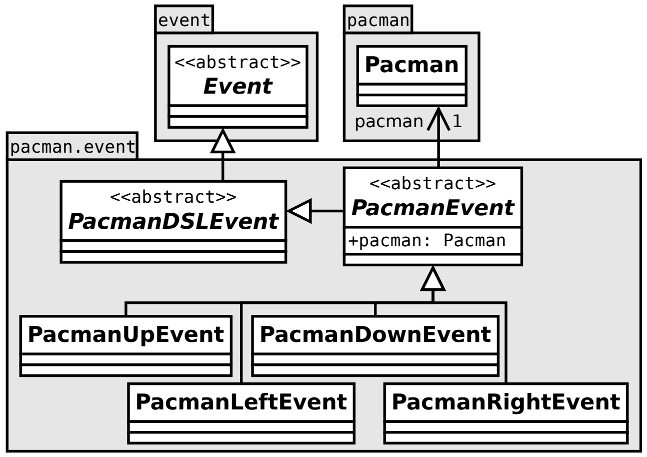


Fig. 3: Event metamodel generated for the Pacman DSL.

*Abstract Syntax:* Figure 2 shows the abstract syntax of the Pac-Man DSL as well as its execution metamodel. A Board is composed of AbstractTiles and Entities. An AbstractTile can be a PassableTile or a WallTile. PassableTiles are further refined into Tiles and GhostHouseTiles. Tile has an `initialPellet` attribute indicating which type of pellet the tile contains, if any. Lastly, an AbstractTile has two bidirectional references to its neighboring AbstractTiles: `right` (the opposite being `left`), and `bottom` (the opposite being `top`). An Entity can either be a Pacman or a Ghost and points to an initial PassableTile where it starts the game and where its position is reset when a Pacman loses a life or a Ghost is eaten. Not shown on the class diagram, a Ghost also has a `personality` dictating its behavior during the game,

**Algorithm 1:** gameLoop

```
  Input: $board$
1 begin
2   $previousTime \leftarrow nanoTime()$
3   while $\neg gameOver()$ do
4     $currentTime \leftarrow nanoTime()$
5     $\delta t \leftarrow currentTime - previousTime$
6     $board.update(\delta t)$
7     if $\delta t < targetFrameRate$ then
8       $sleep(targetFrameRate - \delta t)$
9     $previousTime \leftarrow currentTime$
```

and a Pacman has a number of `initialLives`.

The domain targeted by this DSL is thus the definition of Pac-Man games, including the topology of the level (walls, ghost house and pellets), the number of ghosts, their behavior and the starting positions and number of ghosts and pacmen.

*Operational Semantics:* The real-time aspect of the execution semantics is obtained by using a classic game loop, illustrated by Algorithm 1. Thread sleeps are used in order to reach a target frame rate. The lower part of Figure 2 shows a subset of the execution rules of the Pac-Man DSL. Entities have an `update` rule used to check whether they reached a new Tile according to their speed and the time they already spent in their current Tile. Entities also have an `enterNextTile` rule that takes care of the actual move and deals with the consequences of this move (*e.g.,* killing the Pacman if a Ghost reaches the same Tile, eating the Pellet present on the Tile, etc.). The Pacman entity has a set of rules (`up`, `down`, `left` and `right`) allowing it to change its direction. These rules are annotated as an event handlers, which means that events can be sent to the model to change the direction of the pacman. The event metamodel generated from these annotations is shown in Figure 3.

*User Interface:* The user interface has been specifically developed for the Pac-Man DSL. It is implemented in JavaFX and consists of two parts: an editor presenting useful tools to build a Pac-Man game, and the actual game interface, which receives events from the keyboards and delegates them to the behavioral interface through the event manager. The game interface is implemented as an execution engine listener and updates its view when notified by the engine that a call to the update execution rule of the Board has been completed.

## IV. Future Work

The direct perspectives of this work include the three following research topics. First, defining and the handling of *output* events occurrences sent by an executed model to its environment. This would be a first step toward enabling co-simulation in a generic way. Second, the combined use of behavioral language interfaces and temporal property languages would allow to investigate activities such as testing or runtime monitoring for executable DSLs. Third, the implementation of the underlying approach with another meta-programming language such as xMOF.

***Acknowledgements*—This work has been funded by: the Austrian Science Fund (FWF): P 28519-N31; the Austrian Agency for International Mobility and Cooperation in Education, Science and Research (OeAD) on behalf of the Federal Ministry for Science, Research and Economy (BMWFW) under the grand number FR 08/2017, by the French Ministries of Foreign Affairs and International Development (MAEDI), and the French Ministry of Education, Higher Education and Research (MENESR); Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development.**

## References

[1] Object Management Group, "Semantics of a Foundational Subset for Executable UML Models, V 1.1," August 2013.

[2] R. Bendraou, B. Combemale, X. Crégut, and M. P. Gervais, "Definition of an executable SPEM 2.0," in *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC'07)*, pp. 390–397, IEEE, 2007.

[3] T. Fischer, J. Niere, L. Torunski, and A. Zündorf, "Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java," in *Proceedings of the 6th International Workshop Theory and Application of Graph Transformations (TAGT'98)*, vol. 1764, pp. 157–167, 2000.

[4] D. Harel, H. Lachover, A. Naamad, A. Pnuelli, M. Politi, R. Sherman, A. Shtull-trauring, and M. Trakhtenbrot, "STATEMATE: a working environment for the development of complex reactive systems," *IEEE Transactions on software engineering*, vol. 16, no. 4, pp. 403–414, 1990.

[5] OASIS, "Web Services Business Process Execution Language Version 2.0," 2007.

[6] Y. V. Tendeloo and H. Vangheluwe, "An introduction to classic DEVS," *CoRR*, vol. abs/1701.07697, 2017.

[7] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale, "Execution Framework of the GEMOC Studio (Tool Demo)," in *Proceedings of the 9th International Conference on Software Language Engineering (SLE'16)*, SLE 2016, p. 8, 2016.

[8] R. Heckel, "Graph transformation in a nutshell," *Electr. Notes Theor. Comput. Sci.*, vol. 148, no. 1, pp. 187–198, 2006.

[9] E. Syriani and H. Vangheluwe, "A modular timed graph transformation language for simulation-based design," *Software and System Modeling*, vol. 12, no. 2, pp. 387–414, 2013.

[10] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework, 2nd Edition*. Eclipse Series, Addison-Wesley Professional, 2008.

[11] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet, "Mashup of metalanguages and its implementation in the Kermeta language workbench," *Software and Systems Modeling*, vol. 14, no. 2, 2013.

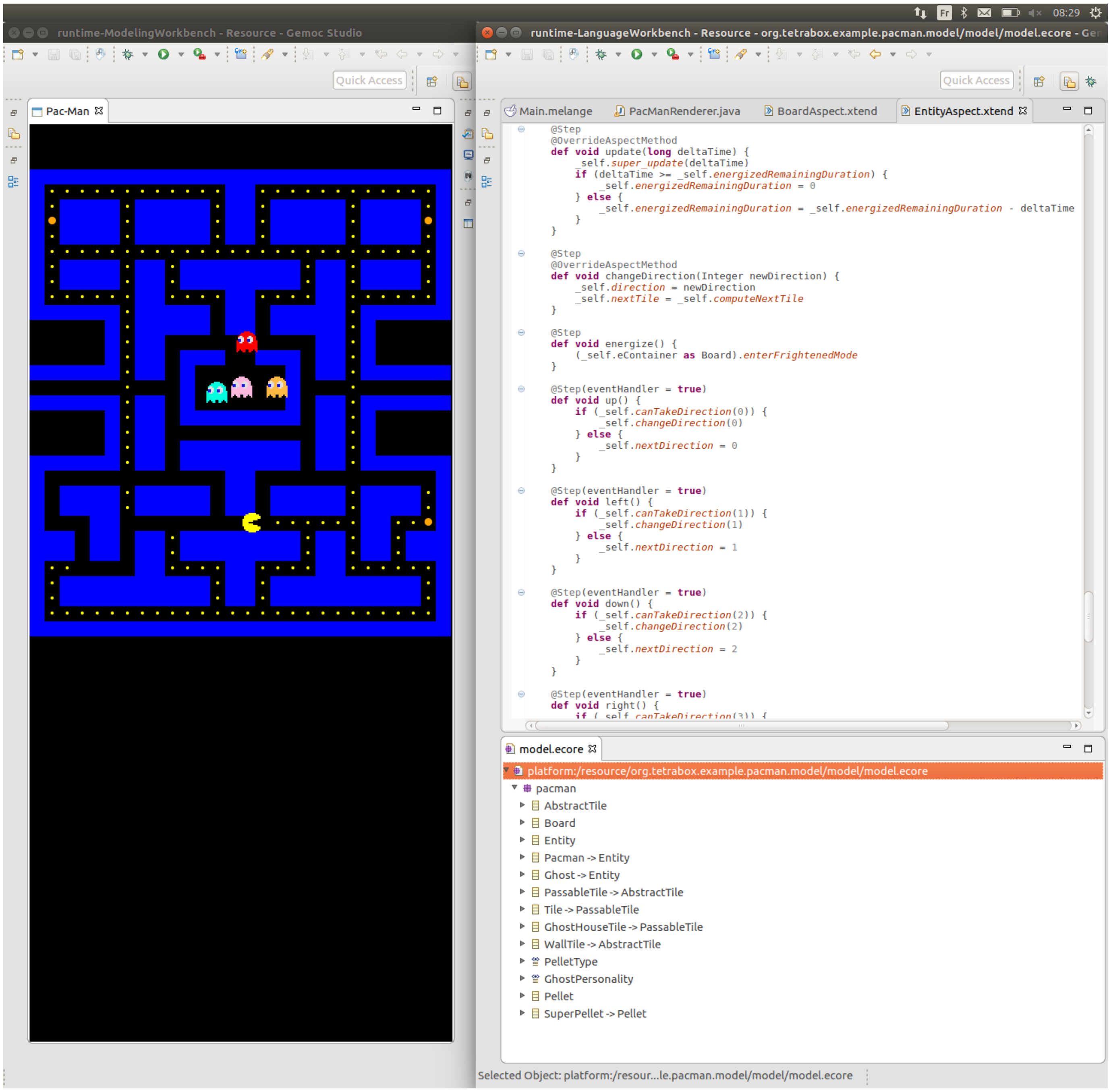
[12] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel, "xMOF: Executable DSMLs based on fUML," in *Proceedings of the 6th International Conference on Software Language Engineering (SLE'13)*, vol. 8225 of *LNCS*, Springer, 2013.

## Appendix: Demonstration

In the first phase of the demonstration, we will show how the Pac-Man DSL is defined in the language workbench of the GEMOC Studio. We will first show how the non-reactive language is defined, and then show how it is made reactive by annotating execution rules from the operational semantics as event handlers. Then, we will give an overview of the artifacts generated from these annotation and the definition of the DSL, that is the behavioral interface for the Pac-Man DSL.

In the second phase of the demonstration, we will show how a game can be defined using an editor we implemented on top of the abstract syntax of the Pac-Man DSL. We will then demonstrate the result by playing a Pac-Man game.

We will then go back to the language workbench, and annotate the *modifySpeed* execution rule as an event. After regenerating the behavioral language interface, we will bind the new event to a key and then launch another game, in which we will be able to send the new event to cheat by increasing the speed of the pacman.



runtime-ModelingWorkbench - Resource - Gemoc Studio

runtime-LanguageWorkbench - Resource - org.tetrabox.example.pacman.model/model/model.ecore - Ge

Main.melange | PacManRenderer.java | BoardAspect.xtend | EntityAspect.xtend

```
    @Step
    @OverrideAspectMethod
    def void update(long deltaTime) {
        _self.super_update(deltaTime)
        if (deltaTime >= _self.energizedRemainingDuration) {
            _self.energizedRemainingDuration = 0
        } else {
            _self.energizedRemainingDuration = _self.energizedRemainingDuration - deltaTime
        }
    }

    @Step
    @OverrideAspectMethod
    def void changeDirection(Integer newDirection) {
        _self.direction = newDirection
        _self.nextTile = _self.computeNextTile
    }

    @Step
    def void energize() {
        (_self.eContainer as Board).enterFrightenedMode
    }

    @Step(eventHandler = true)
    def void up() {
        if (_self.canTakeDirection(0)) {
            _self.changeDirection(0)
        } else {
            _self.nextDirection = 0
        }
    }

    @Step(eventHandler = true)
    def void left() {
        if (_self.canTakeDirection(1)) {
            _self.changeDirection(1)
        } else {
            _self.nextDirection = 1
        }
    }

    @Step(eventHandler = true)
    def void down() {
        if (_self.canTakeDirection(2)) {
            _self.changeDirection(2)
        } else {
            _self.nextDirection = 2
        }
    }

    @Step(eventHandler = true)
    def void right() {
        if (_self.canTakeDirection(3)) {
```

model.ecore

```
platform:/resource/org.tetrabox.example.pacman.model/model/model.ecore
  pacman
    AbstractTile
    Board
    Entity
    Pacman -> Entity
    Ghost -> Entity
    PassableTile -> AbstractTile
    Tile -> PassableTile
    GhostHouseTile -> PassableTile
    WallTile -> AbstractTile
    PelletType
    GhostPersonality
    Pellet
    SuperPellet -> Pellet
```

Selected Object: platform:/resour...le.pacman.model/model/model.ecore

Fig. 4: Overview of the language workbench and of a Pac-Man game being played.