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

(Tool Demo)

Dorian Leroy
JKU Linz
Austria
dorian.leroy@cis.jku.at

Erwan Bousse, Manuel Wimmer
TU Wien
Austria
erwan.bousse@tuwien.ac.at
wimmer@big.tuwien.ac.at

Benoit Combemale
Université de Rennes 1
France
benoit.combemale@irisa.fr

Wieland Schwinger
JKU Linz
Austria
wieland.schwinger@jku.ac.at

***Abstract*—Executable Domain-Specific Languages (DSLs) are used for defining the behaviors of systems. In particular, 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 demonstrate a tool addressing this problem by allowing the augmentation of 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 automatically interrupting the execution flow. This tool is an extension to 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 to create and play Pac-Man games.**

I. Introduction

Enabling the execution of models conforming to the large number of Domain-Specific Languages (DSLs) geared toward the description of the behavior of systems (*e.g.,* [1], [2], [3], [4], [5]) allows to make the most out of those models. This requires to define the *execution semantics* of these languages, which may need to define how conforming models react to *stimuli* from their environment [6]. This may include 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 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]), and to trigger the handling of a given event occurrence. 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, by extending the language workbench, 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 of the modeling workbench has been developed to use this interface. The use of this tool is illustrated with a Pac-Man DSL ([8], [9]) allowing to define Pac-Man games, 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. It is developed as a reactive extension to the GEMOC Studio and written in Java and Xtend. Figure 1 provides an overview of the architecture of this extension. The figure shows the specific case of the Pac-Man DSL.

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 that can be decomposed in a data structure representing the *model state* and a set of *execution rules*. The model state is defined in an *execution metamodel* and extends the abstract syntax metamodel. Before the execution, it is initialized through a transformation from the abstract syntax to the execution metamodel. The execution of the model is performed by a endogenous, in-place transformation on this model state. In the GEMOC Studio, Ecore [10] is used as a metamodeling language to define the abstract syntax and the execution

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

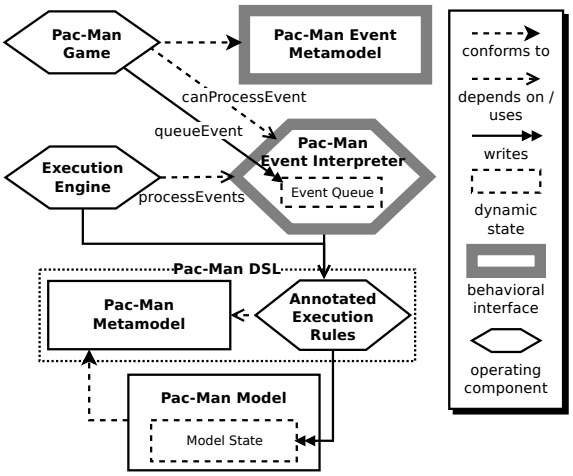

Fig. 1: Architecture of the tool for the Pac-Man example.

metamodel of DSLs. The Kermeta language [11] is used to define the operational semantics and as been extended with an «`eventHandler`» annotation to support the annotation of 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. The execution engine (written in Java) has been extended to add event processing in the execution loop, using the $processEvent$ service, detailed below.

C. Behavioral Interface

The *behavioral language interface* is generated by the *behavioral interface generator* shown on the right of Figure 1. This generator is implemented in Xtend and takes as input the definition of the DSL. It uses the «`eventHandler`» annotation as a flag to detect event handlers and generate the behavioral language interface. The behavioral interface of an executable DSL can be further decomposed as a *domain-specific event metamodel* and a *domain-specific event interpreter*, which is generated as a Java class. The event interpreter 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). External components can use the $canProcessEvent$ service of the event interpreter to check if a stimulus can be handled in the current execution state of the model and can use the $queueEvent$ service to push stimuli 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 interruption of the planned scheduling of execution rules by calling the rule handling the kind of the stimuli in the event queue.

D. External Components

External components use the behavioral language interface to interact with executed models. 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 is such an external component, refreshing the view when receiving notifications from the engine, and using the behavioral interface of the Pac-Man DSL to react to the user's inputs.

III. The Pac-Man example

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

Abstract Syntax: Figure 2 shows two views of the Ecore metamodel of the abstract syntax of the Pac-Man DSL, as a class diagram in the top left corner and in the tree editor in the bottom left corner. A Board is composed of Tiles and Entities. Each Tile has an `x` and an `y` attribute representing its coordinates on the Board. A Tile also has a `passable` attribute indicating whether it is a wall or not, and an `initialPellet` attribute indicating which type of pellet the tile contains, if any. Lastly, a Tile has two bidirectional references to its neighboring Tiles: `right` (the opposite being `left`), and `bottom` (the opposite being `top`). An Entity can either be a Pacman or a Ghost. A Pacman has an initial number of lives (`initialLives`), and a Ghost has a `name` and a `personnality`. Finally, an Entity points to an initial Tile where it starts the game or where its position is reset when a Pacman loses a life.

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

Operational Semantics: Figure 2 shows on the right a part of the K3 semantics of the Pac-Man DSL. Among other execution rules, 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, and an `enterNextTile` rule that takes care of the actual move and deals with the consequences of this move (*e.g.,* by killing the Pacman if a Ghost reaches the same Tile, by eating the Pellet present on the Tile, etc.). Entities also have a changeDirection rule, which is annotated as an «`eventHandler`» in the case of the Pacman class. This means that events can be sent to the model to change the direction of the pacman. The event metamodel generated as part of the behavioral language interface is shown on Figure 2, in the right frame of the bottom left corner.

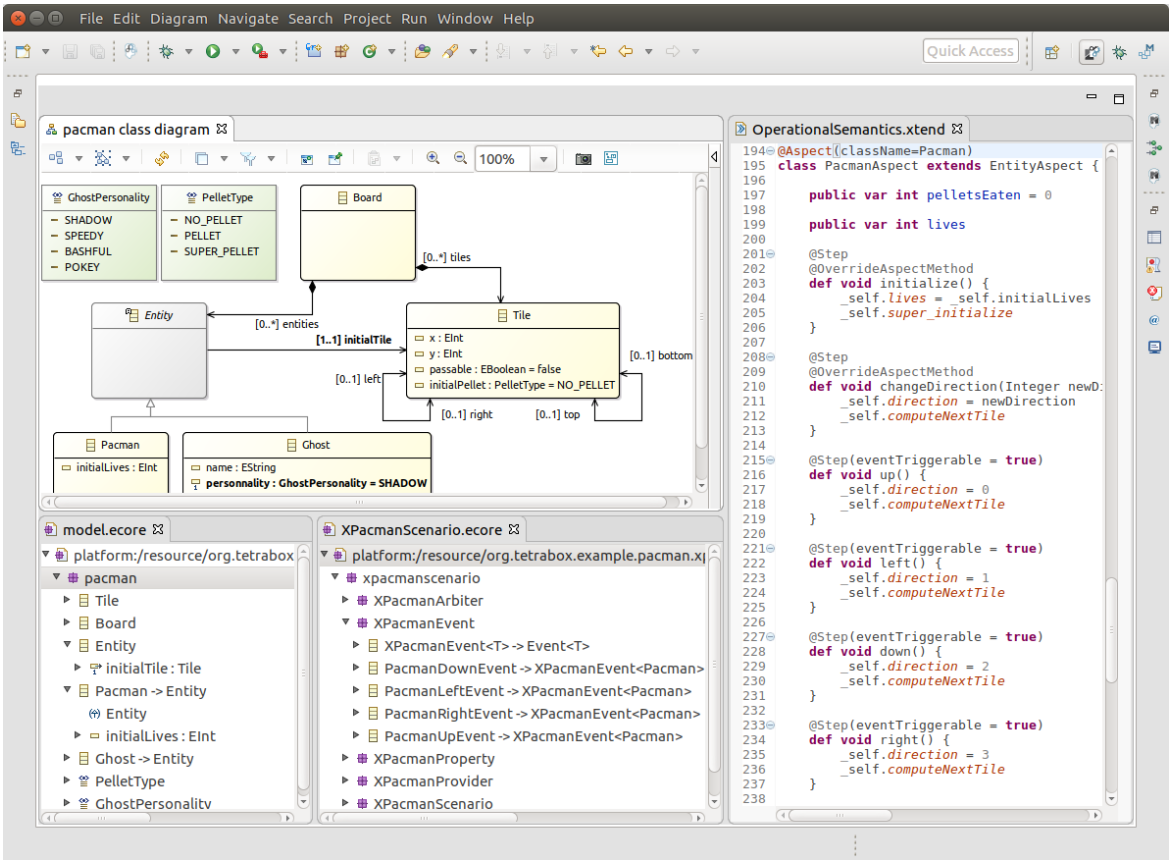

Fig. 2: Abstract syntax and operational semantics of the Pacman DSL.

User Interface: The user interface of the game is in fact a concrete syntax for the Pac-Man DSL. It is implemented in JavaFX and consists of two parts: an editor presenting some useful tools to build a Pac-Man game, and the actual game interface, which receives events from the keyboards and delegates them to the executed model as direction changes for the pacman through the behavioral interface. The editor is thus a concrete syntax for the abstract syntax of the DSL while the game interface is a concrete syntax for the execution metamodel of the DSL. The game interface is implemented as an engine listener and updates its view when notified by the engine.

IV. Future Work

The direct perspectives of this work include the two following research topics. First, the definition and the handling of *output* events occurrences sent by an executed model to its environment. This would enable co-simulation, among other things. Second, the generation of a temporal *domain-specific property language*, which could be used to define temporal properties for several activities that require a behavioral interface, such as testing or runtime monitoring.

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. Van Tendeloo and H. Vangheluwe, "An introduction to classic devs," *arXiv preprint arXiv: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 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, (Amsterdam, Netherlands), p. 8, Oct. 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 & Systems Modeling (SoSyM)*, vol. 14, no. 2, 2013.

APPENDIX

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 (shown in Figure 3).

In the second phase of the demonstration, we will show how a game can be defined using an editor (shown in Figure 4) we implemented on top of the abstract syntax of the Pac-Man DSL.

We will then show how to create a launch configuration to execute a model conforming to the Pac-Man DSL (that is play a specific Pac-Man game).

We will then demonstrate the result, first by playing a Pac-Man game, and then by feeding a scenario (that is a predefined sequence of stimuli) to the executed model which will control the Pac-Man on its own.

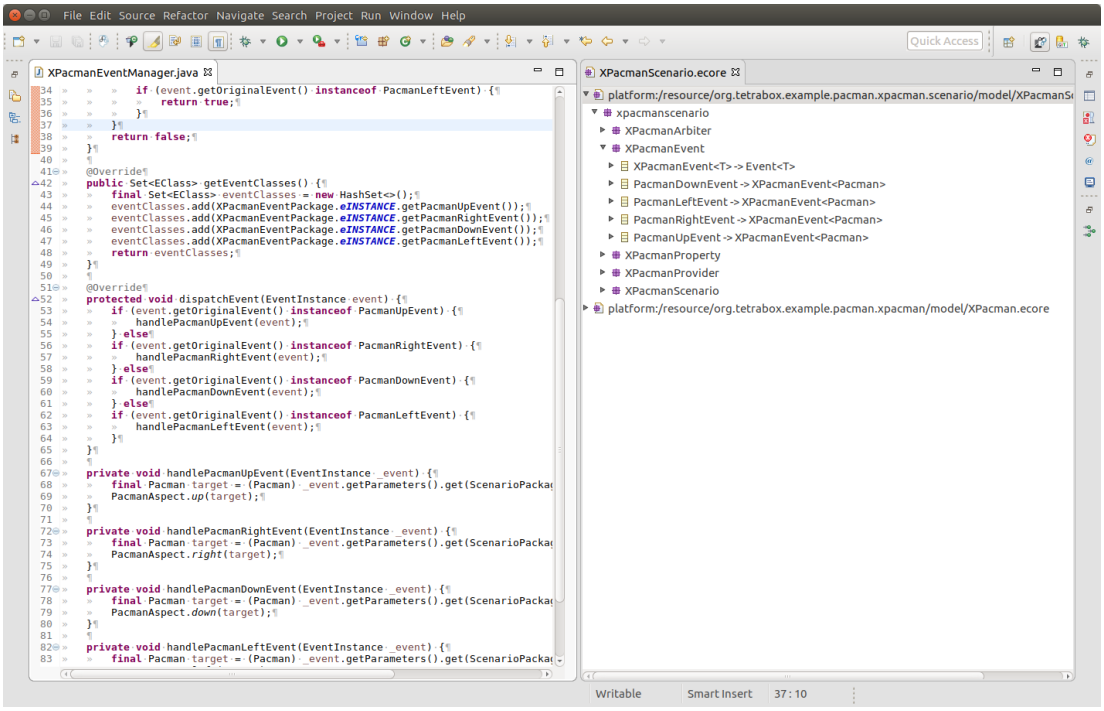

Fig. 3: Artifacts constituting the behavioral interface for the Pac-Man DSL.

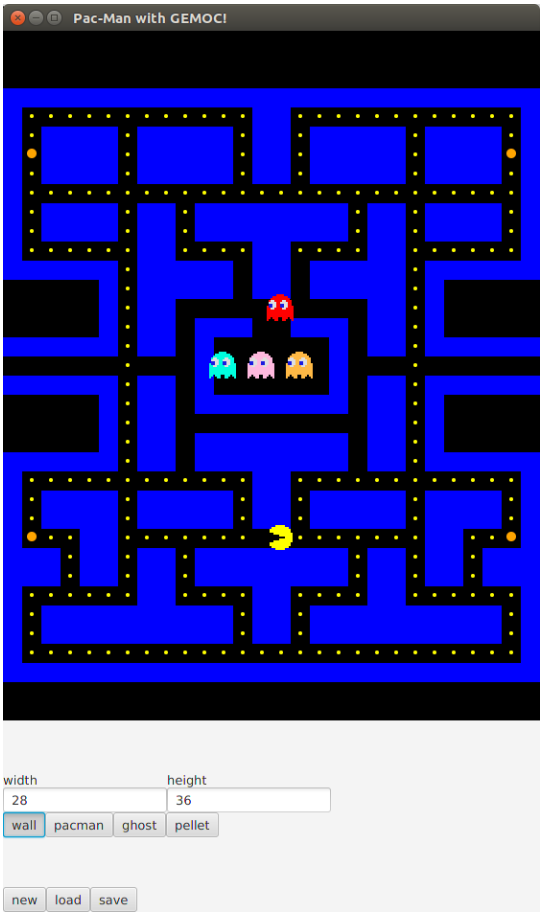

Fig. 4: Pac-Man editor in JavaFX.