On the Execution of Deep Models

Colin Atkinson, Ralph Gerbig, Noah Metzger
University of Mannheim
Software Engineering Group
Mannheim, Germany
Email: $\{atkinson, gerbig, metzger\}$@informatik.uni-mannheim.de

***Abstract*—A variety of tools today support the dynamic execution/simulation of models within a single modeling environment. However, they all suffer from limitations resulting from their implementation on a traditional, two-level modeling platform. The most prominent of these is the inability to represent the specification of the modeling language, the domain model and model execution state at the same time in a uniform and seamless manner. They therefore invariably have to resort to some kind of ad hoc extension mechanism or workarounds to represent all three levels, with corresponding increases in accidental complexity and potential for misunderstandings. In this paper we demonstrate how deep modeling environments provide a conceptually cleaner and more powerful environment for model execution and simulation thanks to their inherent support for the representation of arbitrary numbers of classification levels, and the ability to define customizable, domain specific languages within them.**

I. Introduction

Modeling languages and tools aiming to support the execution of models are available in various different flavors and degrees of maturity. Some tools support the graphical definition of executable models (e.g. fUML [1] in xMOF [2] or graph transformations in AToMPM [3]). Others support the textual definition of execution semantics (e.g. ALF [4]). Sophisticated, industry-quality simulation environments such as Simulink [5] or AnyLogic [6] are also available, with accompanying languages.

All these languages and tools are based on traditional "two-level" modeling technology which limits the modeler to two "physical" levels for modeling - one containing types and one containing instances of those types. Usually the type level is used to define the modeling language (meta-level) and the instance level is used to represent the user model (e.g. UML class or state diagrams). However, with such an arrangement there is no place left to model the instances of the user model which is where the majority of the execution actions conceptually take place. In general, at least three levels are needed to naturally represent model execution in a modeling tool. To address the lack of sufficient modeling levels in traditional OMG-based technologies, workarounds are needed to show run-time data at the level of the executed user model (i.e. execution blueprint).

Deep modeling provides a natural solution to this problem by providing uniform, "out-of-the-box" support for modeling over multiple classification levels. As well as supporting the extra classification level required to store execution information. Deep modeling also provides a number of additional advantages. First, it makes it possible to naturally represent run-time instance information alongside model type information (with inherent support for the semantics of instantiation) without polluting the model execution blueprint. Second, it provides natural support for debugging by allowing model execution traces (i.e. instances) to be directly checked against model execution blueprints (i.e. types). Third, it allows the behavior of the system to be dynamically modified at any time by simply changing model execution instances without changing the blueprint. Fourth, the execution blueprint can be extended dynamically at any time, without the need for any code generation or editor redeployment, simply by adding new types to the language definition. Finally, it allows domains that inherently feature more than three classification levels to be modeled and executed in a natural and uniform way since there is no limitation on the number of levels that can be modeled.

In this paper we illustrate the advantages of the deep modeling approach for model execution by presenting an example of the execution of a simple but intuitive deep model from the domain of gaming. The example represents an executed game in which, from a birds-eye view, two players play against each other on one computer. One uses the mouse to control the game and the other the keyboard. Changes to the state of the game (i.e. the deep model) are immediately visible to all parties, and a third player is able to manipulate the game while the other two players play.

The paper is structured as follows: in the next section (Section II) the deep modeling approach is introduced. The prototype game demonstrating the advantages of deep models for model execution is then shown in Section III and discussed in Section IV. Finally the paper closes with a discussion of related work (Section V) and a few concluding remarks (Section VI).

II. Deep Modeling

Deep modeling environments allow domains with more than one logical class-/instance level to be represented within one physical model. In the domain of model simulation and execution at least three levels are usually required — one defining the modeling language, one describing "the model" and one capturing the execution state of the model in the form of instances. Figure 1 shows an excerpt of the deep model used for modeling the executed game environment. Although this model has only three levels (i.e. O_0, O_1, O_2) the number of levels is unlimited in general.

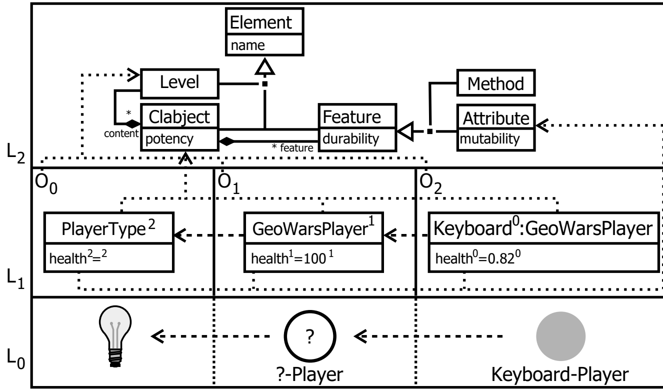

Fig. 1. The orthogonal classification architecture.

The first thing that can be observed from the model is that two kinds of classification relationships are present, one kind represented in the vertical dimension and one kind represented in the horizontal dimension. These two classification dimensions give the underlying architecture of deep modeling environments its name - the Orthogonal Classification Architecture (OCA) [7]. The vertical dimension shows the model from the perspective of a traditional "two-level" implementation of such an environment within a UML or EMF tool. The top level, L_2, defines all model elements that are available in the deep modeling language and is thus called the "linguistic (meta) model". The middle level, L_1, contains the domain content defined by the user and thus actually contains the so called deep model and its multiple "ontological classification levels". All model elements in the L_1 level, except model elements residing at the most abstract ontological classification level, O_0, have two types: an ontological type (horizontal dashed arrows) and a linguistic type (vertical dotted arrows). The lowest linguistic level, L_0, contains the real world entities represented by the deep model (i.e. the ontological content in L_1). When working with a deep model, L_0 and L_2 are usually not visible since they do not contain any information that is immediately relevant to the development and execution of the domain model.

Since model elements in the middle ontological levels are instances of the types at the ontological level above and types for instances at the ontological level below, they are (or play the role of) classes and objects at the same time. In deep modeling they are therefore referred to as "clabjects" (a name derived from "class" and "object") to emphasize that they should be represented and thought of as integrated, unified model elements. The notation used to represent clabjects is designed to be as UML-like as possible whilst being fully level-agnostic. The main notational difference to the UML from an end-user's point of view are the numeric "potencies" next to the names of clabjects, attributes and attribute values as seen in Figure 1. The potency next to the name of a clabject specifies over how many subsequent model levels it can be instantiated. Each instantiation step reduces the potency value by one until 0 is reached. In the example in Figure 1, *PlayerType* has a potency of 2, and is thus instantiable over the following two levels. At the level below, it is instantiated by *GeoWarsPlayer* which has a potency of 1. This, in turn, is instantiated at the level below as *Keyboard* with potency 0. *Keyboard* cannot be instantiated further since the potency of a model element must be a non-negative integer. The potency value next to an attribute's name specifies over how many subsequent levels that attribute can endure and, therefore, has to be possessed by instances of the clabject. Hence, it is also referred to as the attribute's "durability". Finally, the potency next to the value of an attribute specifies over how many levels the value of that attribute can be changed. It is thus also referred to as the "mutability" of that attribute. Durability and mutability follow the same decrementation rule as the potency for model elements.

III. GeoWars: a Deep Model Execution Case-study

Since games can be understood without any domain-expertise the demonstration prototype was chosen from this domain. The game consists of two components as shown in Figure 2: the *Deep Model* representing the executed game (left-hand side) and the *Game Engine* executing the deep model (right-hand side). The model is implemented in the deep modeling environment Melanee [8] and the game engine is implemented in Java. Both parts communicate via sockets, a widely used way to enable communication between two software components. An alternative would be to connect the two components via a file storing exchange data but this has the disadvantage of possible read/write conflicts. Since the Melanee tool is written in Java, the game component could also have been implemented directly in Melanee because it is implemented using Java, too. However, the goal was to create a scenario in which the model is executed by an external execution engine since we assume this is a common situation.

The Melanee component uses the Melanee API to manipulate the deep model. This API offers a transaction-based command framework for manipulating deep model content and several meta-model oriented query operations. In cases where this API is not sufficient, the widely used and well documented capabilities of the Eclipse Platform on which Melanee is built can be employed. Visual appearance is usually not changed through this API but can be. The size and location of model elements can be manipulated through the attached visualizers, and the visual appearance of model elements can be influenced by means of an aspect-oriented, context-sensitive, concrete syntax definition mechanism [9]. Using these features, visualizations of model elements can be dynamically changed based on the values of their attributes.

The deep model used to describe the game, shown in Figure 3, has four levels: O_0 containing a general language for describing games; O_1 containing the language to describe the GeoWars game featured in this demonstration; O_2 containing the designed game levels which can be played and O_3 containing the current state of an executed GeoWars game level. At level O_0, generic types and attributes common to all games are modelled. An instance of *LevelType* represents

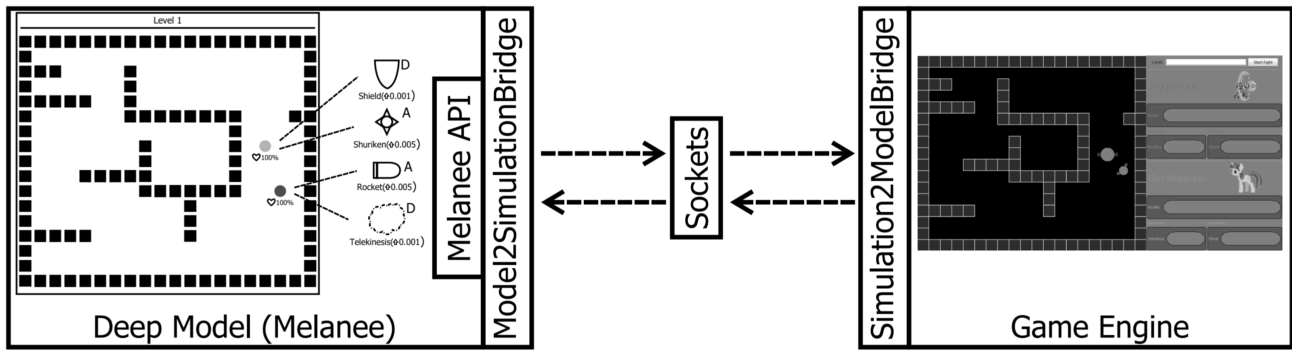

Fig. 2. The GeoWars implementation

the environment in which players play a game. The *name* attribute of *LevelType* is used to identify a particular game level. Levels can have obstacles such as walls, rivers etc., which are represented by instances of *LevelComponentType*, while players are represented as instances of *PlayerType* which is composed of *PlayerComponentTypes*. Such components can be weapons, power-ups etc.

The GeoWars game modeling language is described at level O_1 using instances of the generic game types at level O_0. GeoWars takes place in a level, the *GeoWarsLevel*, which has a unique *name* for identification. It consists of *Walls* which are obstacles that cannot be passed-through by players. Two *GeoWarsPlayers* are located in the level, one mouse-controlled and the other keyboard-controlled as expressed through the *control* attribute. Furthermore, *size* (describing the player radius), *speed* (describing the player movement speed) and *health* (describing the damage that can be tolerated by a player until the end of the game) attributes are specified for *GeoWarsPlayers*. Each player has two *Weapons*, one *AttackWeapon* and one *DefenseWeapon*. Since the corresponding clabjects exist purely for the purpose of grouping weapons they are abstract clabjects as expressed by their potency of *0*. Three *AttackWeapons* and three *DefenseWeapons* are predefined in the game, but new weapons can be added to the model as needed. The available *AttackWeapons* are *Rocket*, *Shuriken* and *Minion* while the available *DefenseWeapons* are *Telekinesis*, *Shield* and *Grenade*.

The visualization of the level designer is realized using Melanee's context-sensitive and aspect-oriented concrete syntax definition features [9]. A model element's visualization information is defined using a graphical, domain-specific visualizer. In the figure, this is shown as a cloud symbol containing the shape of the symbol to be used to represent instances of the clabject. The simplest graphical visualization is the one for *Walls* which are represented as black rectangles. A *GeoWarsPlayer*, on the other hand, is visualized as a solid colored-circle whose color depends on the selected value (*mouse* or *keyboard*) of the *control* attribute. The circle representing the player is indicated by the dotted circle, with *B* being a placeholder for the background-color of the circle. The expression which calculates the color is expressed in square brackets. If the player is keyboard-controlled the color is blue otherwise it is red. Below the circle the current health is indicated as a percentage next to a heart icon. As the *health* attribute is of datatype *real* between 0 and 1 a formula multiplying this value by 100 is used to calculate the actual percentage value.

A very generic symbol is provided for weapons through the *Weapon* clabject. Two join-points are defined, J_I holding the icon to represent the weapon and J_T indicating whether a *Weapon* is an *AttackWeapon* or a *DefenseWeapon*. The name of the weapon and its regeneration speed (indicated by the upwards facing arrow) are displayed at the bottom. The subclasses of *Weapon* provide aspects of type *around*, enabling them to replace the content in the joinpoints. Whether a weapon is for attacking or defense is indicated through an *A* in the first case or *D* in the later case. The icons are provided by the specific weapon classes (e.g. *Rocket*, *Shuriken*, *Shield* and *Telekinesis*). The icons for the other two *Weapon* subclasses are not shown in the figure for space reasons.

Level O_2 shows a blueprint of a game level modeled using the GeoWars DSL. A game level with two players each possessing one attack weapon and one defense weapon has been created. This game level is instantiated for execution at the O_3 level. In the example, the state of the executing game is represented at O_3 indicating that the keyboard-controlled player has a health value of 64% and the mouse-controlled player a health value of 32%. Both players have also moved from the starting position, which is visible from the position of their icons. Multiple instances of the same game level can be displayed side-by-side and analyzed by reasoning services to check if the current execution state is valid. The simulation can also be paused and resumed based on the information stored in the executing game model content.

A pragmatic approach is used to define the semantics of the *Deep Model* by translating each concept in the deep-model to one concept in the data model of the *Game Engine* with clear execution semantics defined in Java. Deep modeling, however, allows semantics to be defined in a translational, denotational and operational style (cf. [10]) but the pragmatic approach was used here as it was the most suitable.

The final deep model and game implementation is shown in Figure 4. The top of the figure shows the deep model, which is an instance of a game level description at level O_3, opened in Melanee. A one-to-one representation of this game

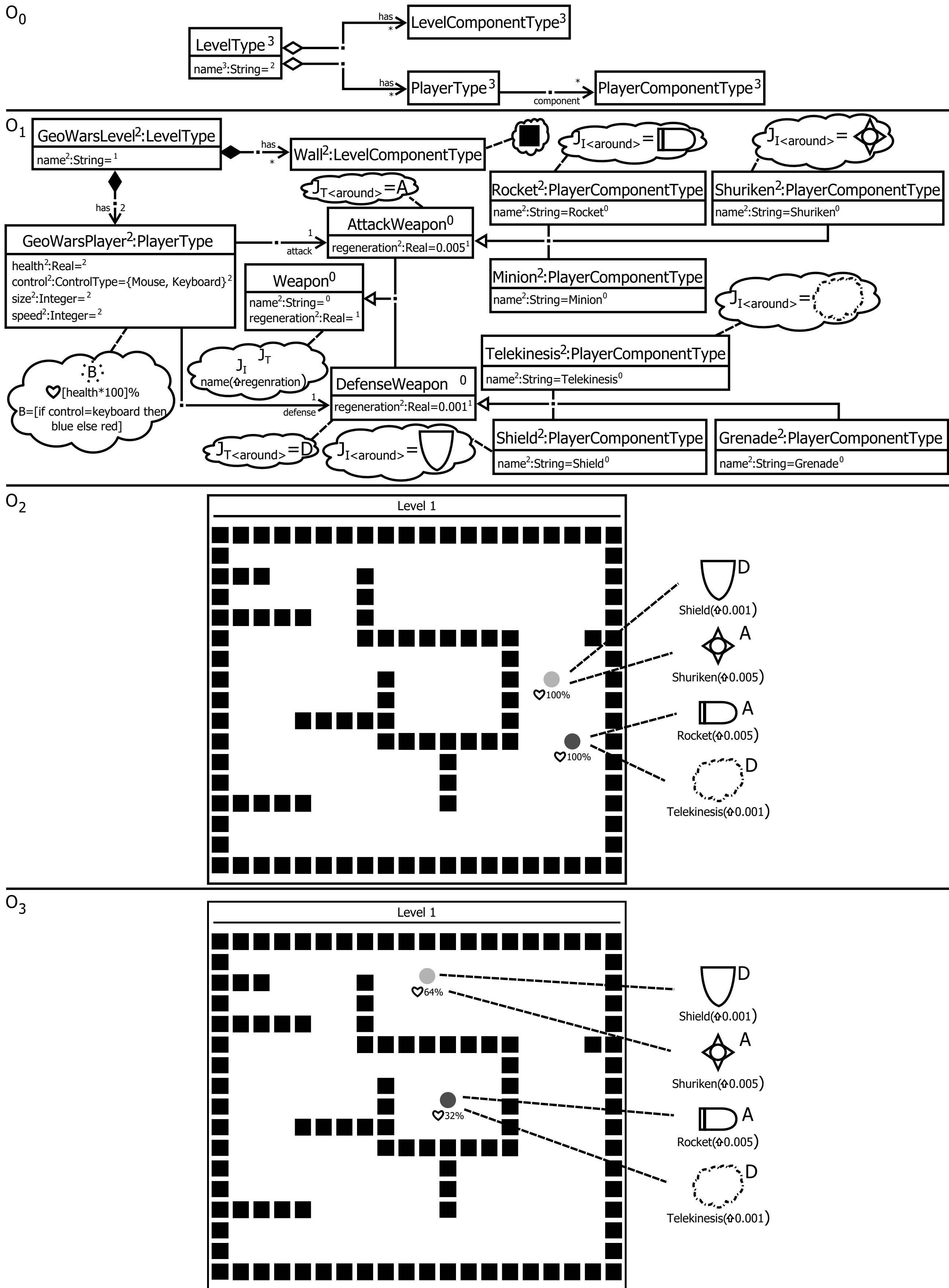

Fig. 3. The GeoWars example model.

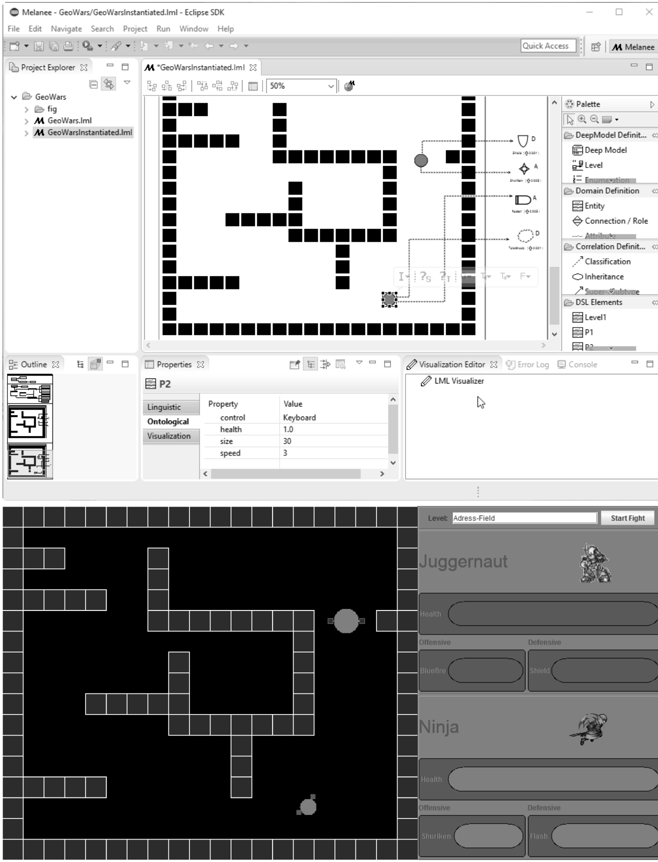

Fig. 4. The GeoWars example running in Melanee and the GeoWars Engine.

level executed by the game engine is shown at the bottom of the figure. A video of the running tool can be found on the Melanee homepage [11] while the game and source are available at [11] and [12].

IV. Discussion

This GeoWars game example shows that it is possible to naturally implement sophisticated model execution environments using deep modeling tools such as Melanee. Melanee provides all the capabilities required for a model execution scenario including: 1) services for model query and manipulation, 2) services for accessing and manipulating graphical model representation definitions, 3) extensibility via a plug-in framework supporting a general purpose programming language (Java) and 4) support for defining execution semantics through action languages and transformations. In areas where the Melanee API is not powerful enough the well-established technologies of the Eclipse Platform can be used as a fallback. The tools ongoing use for other projects allows missing features to be continuously discovered and added to the Melanee modeling environment and made available via the Melanee API.

The model execution scenario shown here can also be implemented with traditional modeling technologies available today. These "two-level" modeling technologies provide a type level which would contain the GeoWars modeling language located at O_1 in Figure 3 and an instance level which would contain the executed level blueprint which is located at level O_2 in Figure 3. The execution information is then displayed as additional information in the model execution blueprint (i.e. the model game level) by applying workarounds such as UML Profiles or the annotation model approach [13] because a dedicated classification level for representing execution information is not available in such a modeling approach.

From a conceptual point of view the deep modeling approach is much cleaner than model execution approaches based on the aforementioned traditional "two-level" modeling approach. In particular, it naturally includes all the classification levels needed to represent the currently executed model. Changes to a running model execution can be defined at the level of a specific model instance to influence the state and behavior of that one particular game only. The model instances can also be used to pause and resume model executions. Such a stack of classification levels is not available in "two-level" modeling technologies and thus data about the state of the executed model has to be saved externally outside the modeling stack. Moreover, modifications to the executed model on a per-instance basis cannot be defined without applying workarounds. In such an architecture, changes can only be performed at the blueprint level which then effects all executions of the model and not a single instance only.

Visualizing the current state of an executed model is also typically done at the blueprint level rather than at the instance level today. This highlights the conceptual problem faced in the representation of executing models within a two-level environment as the user gets the impression that the blueprint is executed instead of a model instance. Deep modeling on the other hand supports the visualization of model instances during model execution and leaves the blueprint untouched.

To summarize, the example demonstrates that deep modeling naturally supports the key features needed for model execution which are: 1) the availability of additional classification levels dedicated to executed model instances, 2) the presentation of the current execution state of a system at the instance level, 3) storing instance information of executed models in the model itself and 4) level-agnostic action and transformation languages to define execution semantics. All of these features are available in today's tools but as extensions to, and workarounds on, the underlying modeling approach which is not optimal for the tasks previously mentioned. Unnatural extensions and workarounds increase the accidental complexity [14], [15] of models.

V. Related Work

Three different areas of work are relevant to the technology presented in this paper: deep modeling tools with model execution capabilities, standard modeling tools with built-in execution/simulation support and approaches to connect pure modeling tools with third party execution/simulation tools.

The only deep modeling tool besides Melanee which allows execution of models is Metadepth [16] which is shipped with

an action language from the Eclipse Epsilon Framework [17]. However, MetaDepth, itself only provides textual visualizations which limits the kind of applications that it can support. The example shown in this paper is mainly graphical but Melanee can support multi-format visualizations (e.g. text, forms, tables).

In the two-level modeling space, Atom3 [18] and its successor AtomPM [3], are the best known academic tools with simulation and execution capabilities. The best known industrial tools are AnyLogic [6] and Simulink [5]. AtomPM allows a model to be simulated/executed by specifying a graph transformation which is much more intuitive and easier to realize than writing a plug-in for the Melanee tool. On the other hand, we believe that this approach is less flexible and powerful when it comes to complex computations or the use of third party components as shown in this paper. One of the most advanced academic tools supporting model execution is xMOF [2], which operates on fUML [1] specifications. However, because of its exclusive focus on UML it cannot be applied to arbitrary domain-specific languages unlike Melanee.

An approach for connecting modeling tools with external execution capabilities is described by Fritzsche et. al. [19]. They present an approach which allows simulation information to be attached to any model and subsequently exported and simulated/executed in another external tool. The model can then be updated by importing the simulation/execution trace back into the modeling tool. However, this approach adds a lot of extra effort to modify existing modeling environments if an integrated tool experience is desired.

VI. Conclusion

This work shows the advantages of deep modeling for supporting the simulation and execution of models. The prototype implementation shows that even the prototype deep modeling tools available today are already up to the task. In the paper we described the conceptual shortcomings of current modeling technologies and explained how deep modeling overcomes them. The shortcomings mainly revolve around the inability of traditional "two-level" modeling environments to naturally support more than one type and one instance level. The proposed deep modeling approach addresses this limitation through 1) the availability of additional classification levels dedicated to executed model instances, 2) the presentation of the current execution state of a system at the instance level, 3) storing instance information of executed models in the model itself and 4) level-agnostic action and transformation languages to define execution semantics. We are continually enhancing the capabilities of the Melanee deep modeling tool and hope the example in this paper will encourage other researchers to investigate the benefits of this technology for model simulation and execution.

References

[1] OMG, "Semantics of a foundational subset for executable uml models (fuml)," http://www.omg.org/spec/FUML/1.1/, 2013.

[2] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel, "xmof: Executable dsmls based on fuml," in *Software Language Engineering*, ser. Lecture Notes in Computer Science, M. Erwig, R. Paige, and E. Van Wyk, Eds. Springer International Publishing, 2013, vol. 8225, pp. 56–75. [Online]. Available: \url{http://dx.doi.org/10.1007/978-3-319-02654-1_4}

[3] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. V. Mierlo, and H. Ergin, "Atompm: A web-based modeling environment," in *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, vol. 1115, 2013, pp. 21–25.

[4] E. Seidewitz, "Uml with meaning: Executable modeling in foundational uml and the alf action language," *Ada Lett.*, vol. 34, no. 3, pp. 61–68, Oct. 2014. [Online]. Available: http://doi.acm.org/10.1145/2692956.2663187

[5] J. B. Dabney and T. L. Harman, *Mastering simulink*. Pearson, 2004.

[6] A. Borshchev, *The Big Book of Simulation Modeling: Multimethod Modeling with AnyLogic 6*. AnyLogic North America, 2013.

[7] C. Atkinson, M. Gutheil, and B. Kennel, "A flexible infrastructure for multilevel language engineering," *Software Engineering, IEEE Transactions on*, vol. 35, no. 6, 2009.

[8] C. Atkinson and R. Gerbig, "Melanie: Multi-level modeling and ontology engineering environment," in *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, ser. MW '12. New York, NY, USA: ACM, 2012, pp. 7:1–7:2.

[9] ——, "Aspect-oriented concrete syntax definition for deep, user-defined modeling languages," ser. MULTI'15, 2015.

[10] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 1st ed. Addison-Wesley Professional, 2008.

[11] Melanee, "Project Website," http://www.melanee.org, 2015.

[12] ——, "Source Code," https://melanee2.informatik.uni-mannheim.de/stash, 2015.

[13] M. Fritzsche, J. Johannes, U. Aßmann, S. Mitschke, W. Gilani, I. Spence, J. Brown, and P. Kilpatrick, "Systematic usage of embedded modelling languages in automated model transformation chains," in *Software Language Engineering*, ser. Lecture Notes in Computer Science, D. Gasevic, R. Lämmel, and E. Van Wyk, Eds. Springer Berlin Heidelberg, 2009, vol. 5452.

[14] J. Brooks, F.P., "No silver bullet essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[15] C. Atkinson and T. Khne, "Reducing accidental complexity in domain models," *Software & Systems Modeling*, vol. 7, no. 3, pp. 345–359, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10270-007-0061-0

[16] J. de Lara and E. Guerra, "Deep meta-modelling with metadepth," in *Proceedings of the 48th international conference on Objects, models, components, patterns*, ser. TOOLS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 1–20.

[17] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The epsilon object language (eol)," in *Model Driven Architecture Foundations and Applications*, ser. Lecture Notes in Computer Science, A. Rensink and J. Warmer, Eds. Springer Berlin Heidelberg, 2006, vol. 4066, pp. 128–142. [Online]. Available: http://dx.doi.org/10.1007/11787044_11

[18] J. Lara and H. Vangheluwe, "Atom3: A tool for multi-formalism and meta-modelling," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, R.-D. Kutsche and H. Weber, Eds. Springer Berlin Heidelberg, 2002, vol. 2306, pp. 174–188.

[19] M. Fritzsche, "Performance related Decision Support for Process Modelling," Ph.D. dissertation, Queen's University Belfast, 2010.